

WINDOWS POWERSHELL W MIESIĄC

WYDANIE III



DONALD W. JONES
JEFFREY HICKS

Tytuł oryginału: Learn Windows PowerShell in a Month of Lunches, 3rd Edition

Tłumaczenie: Grzegorz Kowalczyk

ISBN: 978-83-283-4649-9

Original edition copyright © 2017 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2018 by HELION SA.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz HELION SA dolożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

HELION SA
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/wipom3.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/wipom3_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

<i>Przedmowa</i>	13
<i>Podziękowania</i>	15
<i>O książce</i>	17
<i>O autorach</i>	19
Rozdział 1. Zanim zaczniesz	21
1.1. Dlaczego nie możesz sobie pozwolić na ignorowanie powłoki PowerShell	22
1.1.1. Życie bez powłoki PowerShell	22
1.1.2. Życie z powłoką PowerShell	23
1.2. Od teraz już tylko PowerShell!	24
1.3. Czy ta książka jest dla Ciebie?	24
1.4. Jak korzystać z tej książki	25
1.4.1. Główne rozdziały	25
1.4.2. Ćwiczenia praktyczne	26
1.4.3. Przykładowe kody	26
1.4.4. Materiały uzupełniające	26
1.4.5. Dalsze pogłębianie swojej wiedzy	26
1.4.6. Dla zainteresowanych	27
1.5. Konfigurowanie środowiska testowego	27
1.6. Instalowanie Windows PowerShell	28
1.7. Kontakt z nami	30
1.8. Natychmiastowe działanie z powłoką PowerShell	30
Rozdział 2. Poznaj powłokę PowerShell	31
2.1. Wybierz swoją broń	31
2.1.1. Okno konsoli	33
2.1.2. Zintegrowane środowisko skryptowe	35

2.2.	Wracamy do wpisywania poleceń z klawiatury	38
2.3.	Najczęściej spotykane problemy	39
2.4.	Jaka to wersja?	40
2.5.	Ćwiczenia	41
Rozdział 3. Korzystanie z systemu pomocy		43
3.1.	System pomocy — jak poznawać polecenia powłoki PowerShell?	43
3.2.	Aktualizowanie systemu pomocy	45
3.3.	Korzystanie z systemu pomocy	47
3.4.	Zastosowanie systemu pomocy do wyszukiwania poleceń	48
3.5.	Interpretacja treści plików pomocy	51
3.5.1.	Zestawy parametrów i parametry wspólne	51
3.5.2.	Parametry obligatoryjne i opcjonalne	52
3.5.3.	Parametry pozycyjne	53
3.5.4.	Wartości parametrów	55
3.5.5.	Wyszukiwanie przykładów poleceń	58
3.6.	Dostęp do ogólnych tematów pomocy	59
3.7.	Dostęp do pomocy online	60
3.8.	Ćwiczenia	60
3.9.	Odpowiedzi	62
Rozdział 4. Uruchamianie poleceń		65
4.1.	Nie tworzymy skryptów, ale uruchamiamy polecenia	65
4.2.	Anatomia polecenia	67
4.3.	Konwencja tworzenia nazw poleceń cmdlet	68
4.4.	Aliaszy — skrócone nazwy poleceń	69
4.5.	Tworzenie skrótów	71
4.5.1.	Używanie skróconych nazw parametrów	71
4.5.2.	Używanie aliasów nazw parametrów	71
4.5.3.	Używanie parametrów pozycyjnych	72
4.6.	Trochę oszukiwania — polecenie Show-Command	73
4.7.	Obsługa poleceń zewnętrznych	75
4.8.	Jak radzić sobie z błędami	78
4.9.	Najczęściej spotykane problemy	79
4.9.1.	Wpisywanie nazw poleceń cmdlet	79
4.9.2.	Wpisywanie parametrów	80
4.10.	Ćwiczenia	80
Rozdział 5. Praca z dostawcami		83
5.1.	Czym są dostawcy?	83
5.2.	Jak zorganizowany jest system plików	85
5.3.	W czym system plików jest podobny do innych magazynów danych?	87
5.4.	Poruszanie się w systemie plików	88
5.5.	Używanie symboli wieloznacznych i dokładnych ścieżek	90

5.6.	Praca z innymi dostawcami	91
5.7.	Ćwiczenia	94
5.8.	Co dalej?	94
5.9.	Odpowiedzi	95
Rozdział 6. Potoki — łączenie poleceń		97
6.1.	Łączenie poleceń ze sobą — mniej pracy dla Ciebie	97
6.2.	Eksportowanie wyników działania polecenia do pliku CSV lub XML	98
6.2.1.	Eksportowanie wyników działania do pliku CSV	99
6.2.2.	Eksportowanie wyników działania do pliku XML	100
6.2.3.	Porównywanie plików	101
6.3.	Przesyłanie wyników działania polecenia do pliku lub na drukarkę	104
6.4.	Konwersja na pliki w formacie HTML	105
6.5.	Używanie poleceń cmdlet do modyfikowania systemu — zakończenie działania procesów i zatrzymywanie usług	106
6.6.	Najczęściej spotykane problemy	108
6.7.	Ćwiczenia	110
6.8.	Odpowiedzi	111
Rozdział 7. Dodawanie poleceń		113
7.1.	Jak jedna powłoka może zrobić wszystko	113
7.2.	Powłoki przeznaczone do zarządzania danym produktem	114
7.3.	Rozszerzenia — wyszukiwanie i dodawanie przystawek	115
7.4.	Rozszerzenia — wyszukiwanie i dodawanie modułów	117
7.5.	Konflikty poleceń i usuwanie rozszerzeń	120
7.6.	Jak to wygląda w systemach operacyjnych innych niż Windows	121
7.7.	Używanie nowego modułu	121
7.8.	Skrypty profilów powłoki PowerShell — wstępne ładowanie rozszerzeń podczas uruchamiania powłoki	123
7.9.	Pobieranie modułów z Internetu	125
7.10.	Najczęściej spotykane problemy	126
7.11.	Ćwiczenia	126
7.12.	Odpowiedzi	127
Rozdział 8. Obiekty — dane pod inną nazwą		129
8.1.	Czym są obiekty?	129
8.2.	Dlaczego powłoka PowerShell używa obiektów?	131
8.3.	Odkrywanie obiektów — polecenie Get-Member	133
8.4.	Używanie atrybutów i właściwości obiektów	134
8.5.	Używanie akcji i metod obiektu	135
8.6.	Sortowanie obiektów	136
8.7.	Wybieranie żądanych właściwości	137
8.8.	Obiekty aż do końca	138

8.9. Najczęściej spotykane problemy	141
8.10. Ćwiczenia	141
8.11. Odpowiedzi	142
Rozdział 9. Potoki — zaglądamy nieco głębiej	143
9.1. Potoki — większe możliwości przy mniejszej liczbie poleceń	143
9.2. Jak powłoka PowerShell przekazuje dane za pomocą potoku	144
9.3. Plan A — przekazywanie danych ByValue	145
9.4. Plan B — przekazywanie danych ByPropertyName	148
9.5. Gdy dane do siebie nie pasują — właściwości niestandardowe	153
9.6. Polecenia w nawiasach	156
9.7. Wyodrębnianie wartości z jednej właściwości	157
9.8. Ćwiczenia	163
9.9. Co dalej?	164
9.10. Odpowiedzi	165
Rozdział 10. Formatowanie wyników działania	167
10.1. Formatowanie — upiększanie tego, co widzisz	167
10.2. Praca z formatowaniem domyślnym	168
10.3. Formatowanie tabel	171
10.4. Formatowanie list	173
10.5. Formatowanie szerokich list	174
10.6. Tworzenie niestandardowych kolumn i elementów list	175
10.7. Przesyłanie danych na wyjście: do pliku, drukarki lub na ekran	178
10.8. Jeszcze jeden typ wyjścia: GridViews	178
10.9. Najczęściej spotykane problemy	178
10.9.1. <i>Formatuj dane na końcu</i>	179
10.9.2. <i>Jeden typ obiektu jednocześnie</i>	181
10.10. Ćwiczenia	182
10.11. Co dalej?	183
10.12. Odpowiedzi	183
Rozdział 11. Filtrowanie i porównywanie danych	185
11.1. Jak sprawić, aby powłoka dała Ci to, czego potrzebujesz	185
11.2. Filtrowanie z lewej	186
11.3. Korzystanie z operatorów porównania	187
11.4. Filtrowanie obiektów poza potokiem	189
11.5. Używanie iteracyjnego modelu wiersza poleceń	190
11.6. Najczęściej spotykane problemy	192
11.6.1. <i>Filtr z lewej, proszę</i>	193
11.6.2. <i>Kiedy używanie symbolu \$_ jest dozwolone?</i>	193
11.7. Ćwiczenia	194
11.8. Co dalej?	195
11.9. Odpowiedzi	195

Rozdział 12. Trochę praktyki	197
12.1. Definiowanie zadania	197
12.2. Wyszukiwanie odpowiednich poleceń	198
12.3. Uczymy się, jak korzystać z nowych poleceń	199
12.4. Wskazówki dotyczące samodzielnej nauki	201
12.5. Ćwiczenia	201
12.6. Odpowiedzi	202
Rozdział 13. Komunikacja zdalna — jeden-do-jednego i jeden-do-wielu	203
13.1. Koncepcja komunikacji zdalnej w powłoce PowerShell	204
13.2. Usługa WinRM	206
13.3. Używanie poleceń Enter-PSSession i Exit-PSSession do połączeń zdalnych typu jeden-do-jednego	210
13.4. Zastosowanie polecenia Invoke-Command do komunikacji zdalnej typu jeden-do-wielu	213
13.5. Różnice między poleceniami zdalnymi i lokalnymi	216
13.5.1. Polecenie Invoke-Command kontra parametr -computerName	216
13.5.2. Przetwarzanie lokalne kontra zdalne	217
13.5.3. Obiekty po deserializacji	219
13.6. Ale poczekaj, jest jeszcze coś więcej	220
13.7. Opcje komunikacji zdalnej	221
13.8. Najczęściej spotykane problemy	222
13.9. Ćwiczenia	223
13.10. Co dalej?	224
13.11. Odpowiedzi	224
Rozdział 14. Zastosowanie mechanizmu WMI oraz standardu CIM	225
14.1. Podstawowe elementy WMI	226
14.2. Złe wieści na temat usługi WMI	228
14.3. Eksplorowanie WMI	230
14.4. Wybierz swoją broń: WMI lub CIM	232
14.5. Używanie polecenia Get-WmiObject	233
14.6. Używanie polecenia Get-CimInstance	237
14.7. Dokumentacja WMI	237
14.8. Najczęściej spotykane problemy	237
14.9. Ćwiczenia	238
14.10. Co dalej?	239
14.11. Odpowiedzi	239
Rozdział 15. Wielozadaniowość z zadaniami działającymi w tle	241
15.1. Uruchamianie wielu zadań powłoki PowerShell w tym samym czasie	241
15.2. Zadania synchroniczne i asynchroniczne	242
15.3. Tworzenie zadań lokalnych	243

15.4.	Zapytania WMI jako zadania działające w tle	244
15.5.	Komunikacja zdalna jako zadanie działające w tle	245
15.6.	Pobieranie wyników działania zadania uruchomionego w tle	246
15.7.	Praca z zadaniami podrzędnymi	249
15.8.	Polecenia do zarządzania zadaniami	251
15.9.	Zaplanowane zadania	253
15.10.	Najczęściej spotykane problemy	254
15.11.	Ćwiczenia	256
15.12.	Odpowiedzi	256
Rozdział 16.	Praca z wieloma obiektami jednocześnie	259
16.1.	Automatyzacja zarządzania wieloma obiektami docelowymi	259
16.2.	Preferowany sposób: polecenia „wsadowe”	260
16.3.	Wykorzystanie mechanizmów CIM/WMI — wywoływanie metod	262
16.4.	Plan B — wyliczanie obiektów	266
16.5.	Najczęściej spotykane problemy	271
16.5.1.	Który sposób postępowania jest właściwy?	271
16.5.2.	Metody WMI a polecenia powłoki	273
16.5.3.	Dokumentacja metod	273
16.5.4.	Najczęstsze problemy z poleceniem ForEach-Object	274
16.6.	Ćwiczenia	275
16.7.	Odpowiedzi	275
Rozdział 17.	Bezpieczeństwo wykonywania skryptów	277
17.1.	Zapewnienie bezpieczeństwa powłoki PowerShell	277
17.2.	Model bezpieczeństwa powłoki Windows PowerShell	278
17.3.	Polityka wykonywania skryptów i podpisywanie kodu	280
17.3.1.	Ustawienia polityki wykonywania skryptów	280
17.3.2.	Cyfrowe podpisywanie kodu	283
17.4.	Inne mechanizmy bezpieczeństwa	287
17.5.	Inne luki w zabezpieczeniach?	287
17.6.	Zalecenia bezpieczeństwa	288
17.7.	Ćwiczenia	289
Rozdział 18.	Zmienne — miejsca do przechowywania danych	291
18.1.	Wprowadzenie do zmiennych	291
18.2.	Przechowywanie wartości w zmiennych	292
18.3.	Używanie zmiennych: zabawne sztuczki z apostrofami i cudzysłowami	294
18.4.	Przechowywanie wielu obiektów w zmiennej	297
18.4.1.	Praca z pojedynczymi obiektami w zmiennej	297
18.4.2.	Praca z wieloma obiektami w zmiennej	298
18.4.3.	Inne sposoby pracy z wieloma obiektami	299
18.4.4.	Rozwijanie właściwości i metod obiektów w powłoce PowerShell v3	300
18.5.	Więcej trików z cudzysłowami	301

18.6. Deklarowanie typu zmiennej	303
18.7. Polecenia do pracy ze zmiennymi	305
18.8. Dobre praktyki w pracy ze zmiennymi	306
18.9. Najczęściej spotykane problemy	306
18.10. Ćwiczenia	307
18.11. Co dalej?	307
18.12. Odpowiedzi	307
Rozdział 19. Wejście i wyjście	309
19.1. Pobieranie i wyświetlanie informacji	309
19.2. Polecenie Read-Host	310
19.3. Polecenie Write-Host	313
19.4. Polecenie Write-Output	315
19.5. Inne sposoby wyświetlania danych	317
19.6. Ćwiczenia	318
19.7. Co dalej?	319
19.8. Odpowiedzi	319
Rozdział 20. Sesje — ułatwienie komunikacji zdalnej	321
20.1. Ułatwienie komunikacji zdalnej z użyciem powłoki PowerShell	321
20.2. Tworzenie sesji wielokrotnego użytku i praca z nimi	322
20.3. Używanie sesji z poleceniem Enter-PSSession	323
20.4. Używanie sesji z poleceniem Invoke-Command	325
20.5. Niejawna komunikacja zdalna — importowanie sesji	327
20.6. Korzystanie z rozłączonych sesji	328
20.7. Ćwiczenia	330
20.8. Co dalej?	331
20.9. Odpowiedzi	332
Rozdział 21. Nazywasz to pisanie skryptów?	333
21.1. Nie tworzymy programów, ale raczej pliki wsadowe	333
21.2. Tworzenie poleceń wielokrotnego użytku	334
21.3. Parametryzowanie poleceń	336
21.4. Tworzenie sparametryzowanego skryptu	337
21.5. Dokumentowanie skryptu	339
21.6. Jeden skrypt, jeden potok	340
21.7. Szybkie spojrzenie na zasięg	344
21.8. Ćwiczenia	346
21.9. Odpowiedzi	346
Rozdział 22. Ulepszanie sparametryzowanego skryptu	349
22.1. Skrypt bazowy	349
22.2. Zmuszamy powłokę PowerShell do ciężkiej pracy	350

22.3.	Definiowanie parametrów obligatoryjnych	351
22.4.	Dodawanie aliasów parametrów	354
22.5.	Sprawdzanie poprawności wartości parametru	355
22.6.	Wyświetlanie szczegółowych wyników działania	356
22.7.	Ćwiczenia	358
22.8.	Odpowiedzi	358
Rozdział 23. Zaawansowana konfiguracja komunikacji zdalnej		361
23.1.	Korzystanie z innych punktów końcowych	361
23.2.	Tworzenie niestandardowych punktów końcowych	362
23.2.1.	Tworzenie konfiguracji sesji	363
23.2.2.	Rejestrowanie sesji	364
23.3.	Włączanie mechanizmu komunikacji zdalnej za pośrednictwem hostów pośrednich	367
23.4.	Zaawansowane uwierzytelnianie na komputerach zdalnych	368
23.4.1.	Ustawienia domyślne wzajemnego uwierzytelniania	368
23.4.2.	Wzajemne uwierzytelnianie przez SSL	369
23.4.3.	Wzajemne uwierzytelnianie za pośrednictwem TrustedHosts	369
23.5.	Ćwiczenia	370
23.6.	Odpowiedzi	371
Rozdział 24. Zastosowanie wyrażeń regularnych do parsowania plików tekstowych		373
24.1.	Przeznaczenie wyrażeń regularnych	374
24.2.	Podstawy składni wyrażeń regularnych	374
24.3.	Używanie wyrażeń regularnych z operatorem -Match	376
24.4.	Używanie wyrażeń regularnych z poleceniem Select-String	377
24.5.	Ćwiczenia	379
24.6.	Co dalej?	380
24.7.	Odpowiedzi	381
Rozdział 25. Różne wskazówki, techniki, sztuczki i chwytły		383
25.1.	Profile, podpowiedzi i kolory — dostosowywanie powłoki	383
25.1.1.	Profile powłoki PowerShell	383
25.1.2.	Dostosowywanie znaku zachęty powłoki	386
25.1.3.	Modyfikowanie ustawień kolorów	386
25.2.	Operatory: -as, -is, -replace, -join, -split, -in, -contains	388
25.2.1.	Operatory -as i -is	388
25.2.2.	Operator -replace	389
25.2.3.	Operatory -join i -split	389
25.2.4.	Operatory -contains i -in	390
25.3.	Operowanie na ciągach znaków	391
25.4.	Operowanie na datach	392
25.5.	Operowanie na datach WMI	394

25.6. Ustawianie domyślnych wartości parametrów	395
25.7. Zastosowanie bloków skryptu	396
25.8. Więcej wskazówek, trików i technik	397
Rozdział 26. Korzystanie ze skryptów innych użytkowników	399
26.1. Skrypt	400
26.2. Analiza wiersz po wierszu	404
26.3. Ćwiczenia	405
26.4. Odpowiedzi	407
Rozdział 27. To jeszcze nie koniec	409
27.1. Pomysły na dalszą eksplorację powłoki PowerShell	409
27.2. „Od czego mam zacząć, kiedy przeczytałem już tę książkę?”	410
27.3. Inne zasoby, o których warto pamiętać	411
Rozdział 28. PowerShell — podręczna ściągawka	413
28.1. Interpunkcja	413
28.2. Pliki pomocy	417
28.3. Operatory	417
28.4. Niestandardowe właściwości i kolumny	418
28.5. Pobieranie parametrów z potoku	419
28.6. Kiedy używać symbolu zastępczego \$_	420
Dodatek A. Ćwiczenia podsumowujące	423
Skorowidz	435

Przedmowa

Już od bardzo dawna uczymy naszych studentów posługiwania się powłoką PowerShell. Kiedy Don zaczął rozważać napisanie pierwszego wydania tej książki, zdał sobie sprawę, że większość autorów książek oraz instruktorów prowadzących szkolenia z powłoki PowerShell — w tym także on sam — próbuje zmusić słuchaczy do potraktowania powłoki jako swego rodzaju języka programowania. W większości książek dotyczących powłoki PowerShell zaczyna się omawiać „skrypty” już w trzecim lub czwartym rozdziale, co powoduje, że coraz więcej studentów nie jest zadowolonych z takiego podejścia zorientowanego na programowanie. Ci użytkownicy chcą po prostu używać powłoki jako powłoki, przynajmniej na początku, a my, instruktorzy, nie dostarczamy im odpowiednich szkoleń, które odpowiadałyby ich potrzebom.

Don postanowił zatem podjąć wyzwanie. Na blogu na stronie Windows IT Pro umieścił post, w którym przedstawił propozycję spisu treści tej książki. Liczne opinie, uwagi i sugestie od użytkowników bloga pozwoliły dopasować się do ich potrzeb i znacząco przyczyniły się do powstania książki w takiej postaci, którą trzymasz w ręku. Don chciał, aby każdy rozdział był krótki, skupiony na kilku najważniejszych zagadnieniach i łatwy do opanowania w krótkim czasie, ponieważ wiemy, że administratorzy nie mają zbyt dużo wolnego czasu i często muszą się uczyć „w przelocie”. Pojawienie się powłoki PowerShell v3 było oczywiście dobrym momentem, aby zaktualizować tę książkę, więc Don zwrócił się do Jeffery’ego Hicksa, wieloletniego współpracownika i posiadacza tytułu MVP, o pomoc.

Obaj chcieliśmy książki, która skupiałaby się na samej powłoce PowerShell, a nie na niezliczonych technologiach, których PowerShell dotyka, takich jak Exchange Server, SQL Server czy System Center. Jesteśmy przekonani, że ucząc się poprawnego korzystania z powłoki, możesz jednocześnie nauczyć się zarządzania wszystkimi tymi produktami serwerowymi w „powershellowy” sposób. Z tego powodu ta książka skupia się na pracy z powłoką PowerShell. Nawet jeżeli korzystasz także z książek zawierających gotowe

do użycia schematy i sposoby realizacji określonych zadań często wykonywanych przez administratora, to i tak nasza książka pomoże Ci lepiej zrozumieć, jak działają te przykłady, co może znakomicie ułatwić Ci modyfikowanie tych rozwiązań, aby osiągnąć inne cele, a w ostatecznym rozrachunku — ułatwić również tworzenie własnych poleceń i skryptów.

Mamy jednak nadzieję, że niniejsza książka nie będzie jedynym źródłem Twojej wiedzy na temat powłoki PowerShell. Jesteśmy także współautorami książki *Learn PowerShell Toolmaking in a Month of Lunches*, która oferuje takie samo podejście do nauczania sposobów tworzenia skryptów i narzędzi powłoki PowerShell. Możesz także poszukać filmów, które zamieszczaliśmy w serwisie YouTube, oraz czytać artykuły, które pisaliśmy dla takich witryn jak Petri IT Knowledgebase czy Windows IT Pro, nie wspominając już o kursach na platformie Pluralsight.

Jeżeli chciałbyś jeszcze bardziej pogłębiać swoją wiedzę i umiejętności, zachęcamy do zajrzenia na stronę <http://www.PowerShell.org>. Obaj odpowiadamy tam na pytania na kilku forach dyskusyjnych i chętnie pomożemy Ci wydostać się z tego, na czym utknąłś. Ta strona jest naprawdę świetnym portalem dla solidnej i aktywnej społeczności użytkowników powłoki PowerShell; możesz się tam dowiedzieć o darmowych e-bookach, znaleźć informacje, relacje i podcasty z PowerShell + DevOps Summit, a także poczytać o wszystkich regionalnych i lokalnych grupach użytkowników oraz wydarzeniach związanych z powłoką PowerShell, które mają miejsce w ciągu całego roku. Zaangażuj się — to świetny sposób, aby uczynić powłokę PowerShell mocną częścią swojej kariery.

Powodzenia i dobrej zabawy w pracy z powłoką PowerShell!

Podziękowania

Książki same się nie piszą, nie redagują i nie publikują. Don chciałby podziękować wszystkim pracownikom wydawnictwa Manning Publications, którzy zdecydowali się podjąć wyzwanie i przyczynili się do powstania tej książki. Jeff chciałby tutaj podziękować Donowi za zaproszenie go do wspólnej pracy nad książką, a także społeczności użytkowników powłoki PowerShell za ich entuzjazm i wsparcie. Don i Jeff są wdzięczni wydawnictwu Manning za umożliwienie im kontynuowania serii *Month of Lunches*, co zaowocowało pojawieniem się już trzeciej edycji tej książki.

Dziękujemy naszym recenzentom, którzy czytali manuskrypt podczas jego tworzenia i przekazywali nam wiele sugestii i porad — są to: Bennett Scharf, Dave Pawson, David Moravec, Keith Hill i Rajesh Attaluri. Ponadto składamy podziękowania dla Eriki Bricker, Geralda Macka, Henry’ego Phillipsa, Hugo Durany, Josepha Tingsanchaliego, Noreen Dertinger, Oliviera Deveaulta, Stefana Hellwegera, Stevena Presleya i Tiklu Ganguly’ego, którzy przekazali nam wiele cennych komentarzy.

Wreszcie składamy również podziękowania Jamesowi Berkenbile’owi i Trentowi Whiteley’owi za przeprowadzenie korekty technicznej rękopisu i kodów źródłowych podczas przygotowywania książki do druku.

O książce

Większość z tego, co powinieneś wiedzieć o tej książce, zostało omówione w rozdziale 1., ale jest kilka rzeczy, o których powinniśmy wspomnieć już teraz.

Po pierwsze, jeżeli zamierzasz samodzielnie sprawdzać omawiane w książce przykłady i wykonywać ćwiczenia praktyczne, będziesz potrzebować maszyny wirtualnej lub komputera z systemem Windows 8.1, Windows Server 2012 lub nowszym. Dokładniej omówimy to w rozdziale 1. W zasadzie możesz również korzystać z komputera działającego pod kontrolą systemu Windows 7, ale w takiej sytuacji nie będziesz w stanie wykonać co najmniej kilku praktycznych ćwiczeń.

Po drugie, przygotuj się do przeczytania tej książki od początku do końca, z zachowaniem kolejności rozdziałów. I znów jest to coś, co dokładniej wyjaśnimy w rozdziale 1.. Chodzi o to, że każdy rozdział wprowadza kilka nowych rzeczy, których będziesz potrzebować w kolejnych rozdziałach. Nie powinieneś próbować czytać tej książki za jednym podejściem — pozostań przy założonej przez nas częstotliwości czytania po jednym rozdziale dziennie. Ludzki mózg może zaabsorbować tylko pewną określoną ilość informacji naraz, toteż dawkując wiedzę o pracy z powłoką PowerShell małymi porcjami, dużo szybciej nauczysz się z niej efektywnie korzystać.

Po trzecie, w tej książce znajdziesz wiele fragmentów kodu. Większość z nich jest krótka, więc nie powinieneś mieć żadnych problemów z ich przepisaniem. Właściwie to zalecamy takie postępowanie, ponieważ dzięki temu będziesz miał okazję udoskonalić umiejętność niezbędną do pracy z powłoką PowerShell: dokładne wpisywanie poleceń! Dłuższe fragmenty kodu zostały zamieszczone na listingach, a ich gotowe kody źródłowe możesz pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/wipom3.zip>.

Oprócz tego powinieneś również poznać pewne konwencje zapisu, które przyjęliśmy w tej książce. Polecenia i kod źródłowy skryptów będą zawsze wyróżnione specjalną czcionką, tak jak w tym przykładzie:

```
Get-WmiObject -class Win32_OperatingSystem  
  ↪ -computerName SERVER-R2
```

W przedstawionym przykładzie widoczny jest również znak kontynuacji wiersza, którego będziemy używać w tej książce. Wskazuje on, że te dwa wiersze powinny być wpisane jako pojedynczy wiersz polecenia powłoki PowerShell. Innymi słowy, przepisując takie polecenie, nie naciskaj klawisza *Enter* ani *Return* po nazwie klasy `Win32_OperatingSystem` — po prostu pisz dalej. Powłoka PowerShell pozwala na wpisywanie bardzo długich wierszy poleceń, ale strona książki ma niestety swoje fizyczne ograniczenia.

Czasami zobaczysz także podobną czcionkę w samym tekście, na przykład gdy piszemy `Get-Command`. Użycie takiej czcionki w tekście po prostu informuje, że patrzysz na polecenie, parametr lub inny element, który możesz wpisać w powłoce.

Czwarta sprawa to trudny temat, o którym będziemy jeszcze mówić w kilku rozdziałach: znak odwróconego apostrofu, czyli tzw. *gravis* (```). Oto przykład:

```
Invoke-Command -scriptblock {Dir} `   
-computerName SERVER-R2, localhost
```

Znak na końcu pierwszego wiersza kodu nie jest przypadkowym kleksem — to prawdziwy znak, który możesz wpisać. Na standardowej klawiaturze odwrócony apostrof znajdziesz zwykle w pobliżu lewego górnego rogu, pod klawiszem *Esc*, na tym samym klawiszu co znak tyldy (`~`). Jeżeli zobaczysz taki znak w poleceniu lub na listingu, po prostu wpisz go dokładnie w tym samym miejscu. Co więcej, jeżeli taki znak pojawia się na końcu wiersza — tak jak w poprzednim przykładzie — upewnij się, że jest to ostatni znak w tym wierszu. Jeżeli pozwolisz, aby pojawiły się po nim spacje lub tabulacje, odwrócony apostrof nie będzie działał poprawnie, podobnie jak zawierający go przykładowy kod.

Wreszcie od czasu do czasu pokierujemy Cię do zasobów internetowych. Tam, gdzie adresy URL są szczególnie długie i trudne do wpisania, zastąpiliśmy je skrótami opartymi na systemie wydawnictwa Manning, które wyglądają jak *<http://mng.bz/S085>* (zobaczysz je na przykład w rozdziale 1.).

O autorach

DON JONES jest wielokrotnym laureatem prestiżowej nagrody Microsoft Most Valuable Professional (MVP) za pracę z powłoką Windows PowerShell. Przez pięć lat pisał artykuły na temat powłoki Windows PowerShell dla „Microsoft TechNet Magazine”. Obecnie prowadzi blog na stronie <http://PowerShell.org> i jest autorem kolumny *Decision Maker* oraz bloga w portalu „Redmond Magazine”. Don jest bardzo płodnym autorem, bowiem od 2001 roku wydał już ponad tuzin swoich książek. Obecnie jest dyrektorem programowym IT Ops na internetowej platformie szkoleniowej Pluralsight. Jego pierwszym językiem skryptowym w systemie Windows był KiXtart, sięgający aż do połowy lat dziewięćdziesiątych. W 1995 roku Don przerzucił się na VBScript i był jednym z pierwszych profesjonalistów IT, którzy zaczęli używać wczesnych wersji nowego produktu firmy Microsoft o nazwie kodowej Monad. Później produkt ten przeobraził się w powłokę Windows PowerShell. Don mieszka w Las Vegas, a kiedy robi się tam za gorąco, przenosi się do domu niedaleko Duck Creek Village w Utah.

JEFFERY HICKS to weteran branży IT z ponad 25-letnim doświadczeniem, na które składa się w znacznym stopniu praca w charakterze konsultanta ds. infrastruktury IT, specjalizującego się w technologiach serwerowych firmy Microsoft, z naciskiem na automatyzację i optymalizację wydajności. Jest wielokrotnym laureatem nagrody Microsoft MVP Award w kategorii Windows PowerShell. Obecnie pracuje jako niezależny autor, trener i konsultant. Prowadził szkolenia z powłoki PowerShell i prezentacje o korzyściach płynących z automatyzacji zadań dla specjalistów IT na całym świecie. Jeff napisał artykuły dla wielu stron internetowych i wydawnictw drukowanych, jest autorem licznych artykułów w portalu Petri IT Knowledgebase, autorem wielu szkoleń dostępnych na platformie Pluralsight oraz częstym prelegentem na konferencjach technologicznych i spotkaniach grup użytkowników. Jeffa możesz śledzić na jego blogu, dostępnym pod adresem <http://jdhitsolutions.com/blog>, oraz na Twitterze (@JeffHicks).

Zanim zaczniesz



Posługiwania się powłoką Windows PowerShell uczymy począwszy od jej wersji 1.0, która ukazała się w 2006 roku. W tamtych czasach większość użytkowników korzystających z powłoki wykazywała się doświadczeniem w zakresie stosowania języka VBScript i chętnie wykorzystywała swoje umiejętności do poznawania powłoki PowerShell. Z tego powodu zarówno my, jak i inni specjaliści piszący książki i artykuły opieraliśmy się na założeniu, że użytkownik będzie wykorzystywał swoje nabyte wcześniej umiejętności programowania i pisania skryptów.

W drugiej połowie 2009 roku nastąpiła jednak poważna zmiana. Z powłoki PowerShell zaczęło korzystać coraz więcej administratorów, którzy *nie* mieli wcześniejszych doświadczeń z VBScript. Nagle okazało się, że nasze stare schematy nauczania nie działają już tak dobrze jak kiedyś, ponieważ skupialiśmy się przede wszystkim na pisaniu skryptów i programowaniu. Wtedy zaczęliśmy sobie zdawać sprawę z tego, że PowerShell nie jest językiem skryptowym. Jest to pełnowymiarowa powłoka systemu, w której uruchamiane są różne narzędzia wiersza polecenia. Podobnie jak wszystkie dobre powłoki, PowerShell ma możliwości tworzenia i uruchamiania skryptów, ale nie musisz ich używać, a już na pewno nie musisz rozpoczynać pracy z powłoką od ich tworzenia. Zaczęliśmy więc zmieniać nasze schematy nauczania i sposoby prowadzenia prezentacji na licznych konferencjach, na których występujemy co roku, a Don zastosował te zmiany również w szkoleniach prowadzonych przez naszych instruktorów.

Książka, którą trzymasz w ręku, jest wynikiem tego procesu i możemy śmiało powiedzieć, że jest to nasza najlepsza książka przeznaczona dla użytkowników powłoki PowerShell, którzy nie mają doświadczenia w pisaniu skryptów (choć z pewnością nie zaszkodzi, jeżeli będą je mieli). Ale zanim przejdziemy do nauki, musimy jeszcze omówić kilka ważnych zagadnień.

1.1. Dlaczego nie możesz sobie pozwolić na ignorowanie powłoki PowerShell

Skrypty wsadowe. KiXtart. VBScript. Spójrzmy prawdzie w oczy, tak naprawdę Windows PowerShell nie jest pierwszą próbą zapewnienia możliwości automatyzacji zadań dla administratorów systemu Windows, jaką przeprowadziła firma Microsoft i inne. Naszym skromnym zdaniem naprawdę warto poznać powłokę PowerShell, ponieważ kiedy to zrobisz, poczujesz, że czas poświęcony na naukę zaczyna naprawdę procentować. Najpierw zastanowimy się, jak wyglądało życie, zanim pojawił się PowerShell, i przyjrzymy się niektórym zaletom korzystania z tej powłoki.

1.1.1. Życie bez powłoki PowerShell

Wielu administratorów systemu Windows bardzo chętnie używa narzędzi wyposażonych w graficzny interfejs użytkownika (GUI — ang. *Graphical User Interface*) do wykonywania swoich codziennych obowiązków. W końcu przecież system Windows opiera się na interfejsie graficznym i dlatego nie nazywamy tego systemu „tekstowym”. Aplikacje i narzędzia wyposażone w interfejsy GUI są bardzo wygodne, ponieważ pozwalają w dużej mierze samodzielnie odkrywać, co można przy ich pomocy zrobić. Don pamięta, kiedy po raz pierwszy uruchomił przystawkę *Użytkownicy i komputery usługi Active Directory* (ang. *Active Directory Users and Computers*). Ustawiał wskaźnik myszy nad poszczególnymi ikonami poleceń, czytał wyświetlane podpowiedzi, otwierał kolejne menu i klikał prawym klawiszem, aby zobaczyć, jakie polecenia są dostępne w menu podręcznym. Graficzne interfejsy użytkownika znakomicie ułatwiają poznawanie nowych narzędzi. Niestety najczęściej narzędzia wyposażone w interfejsy GUI mają zerowy zwrot z inwestycji. Jeżeli utworzenie nowego użytkownika w usłudze Active Directory zajmuje z wykorzystaniem takiego narzędzia około pięciu minut (przy założeniu, że wypełniasz danymi większość dostępnych pól, to całkiem ostrożny szacunek), to raczej szansa na to, że uda Ci się znacząco przyspieszyć taki proces, jest dosyć mizerna. Utworzenie kont dla stu użytkowników zajmie w takim przypadku około pięciuset minut — i w zasadzie nic (no może poza nauką szybkiego pisania i klikania) nie może tego procesu przyspieszyć.

Firma Microsoft próbowała na różne sposoby rozwiązać ten problem, a powstanie języka VBScript było chyba tego najbardziej udaną próbą. Napisanie skryptu VBScript, który mógłby zaimportować nowych użytkowników z pliku CSV, zajęłoby Ci zapewne jakąś godzinę, ale po zainwestowaniu tej godziny w napisanie skryptu kolejne operacje tworzenia użytkowników w przyszłości zajmowałyby tylko kilka czy kilkanaście sekund. Jednak problem z językiem VBScript polega na tym, że firma Microsoft nigdy nie przykładała się zbytnio do jego wsparcia. Deweloperzy tej firmy musieli nieustannie pamiętać o tym, aby udostępniać w języku VBScript nowe funkcje i mechanizmy systemu Windows, a jeżeli o tym zapominali, to nie można było z tym nic zrobić. Chcesz zmienić adres IP karty sieciowej za pomocą języka VBScript? Dobrze, możesz to oczywiście zrobić. Chcesz sprawdzić prędkość łącza? Aaa, tego już nie możesz zrobić, ponieważ nikt z deweloperów nie pofatygował się, aby udostępnić taką możliwość z poziomu języka VBScript. Koniec, kropka. Jeffrey Snover, architekt powłoki Windows PowerShell, nazywa to *fenomenem ostatniej mili* — przy użyciu skryptów języka VBScript (i innych

podobnych technologii) możesz zrobić naprawdę wiele, ale często w pewnym momencie, gdy już wydaje Ci się, że wszystko jest na dobrej drodze, napotykasz taką przeszkodę i nie jesteś w stanie pokonać tej przysłowiowej „ostatniej mili” prowadzącej do osiągnięcia celu.

Windows PowerShell to ze strony firmy Microsoft wyraźna próba ułatwienia życia wszystkim użytkownikom i pokonania tej „ostatniej mili”. Co ważniejsze, jak do tej pory była to próba bardzo udana. Powłoka PowerShell i jej możliwości są wykorzystywane przez wiele produktów zarówno firmy Microsoft, jak i firm trzecich, a jej popularność jest dodatkowo podsycona przez gwałtownie rosnącą globalną społeczność ekspertów i użytkowników.

1.1.2. Życie z powłoką PowerShell

Celem, jaki postawiła sobie firma Microsoft, tworząc Windows PowerShell, było zapewnienie stuprocentowej funkcjonalności administracyjnej tej powłoki. Oczywiście firma ta nadal tworzy i rozwija nowe narzędzia wyposażone w interfejsy GUI, ale bardzo często takie narzędzia gdzieś w tle wykonują polecenia powłoki PowerShell. Takie podejście wręcz zmusiło firmę Microsoft do zapewnienia, że każdą operację, która jest dostępna za pośrednictwem danego narzędzia, można również wykonać z poziomu powłoki PowerShell. Dzięki temu, jeżeli chcesz zautomatyzować często wykonywane zadania lub utworzyć proces, nad którym narzędzia wyposażone w interfejs graficzny nie dają pełnej kontroli, możesz po prostu uruchomić powłokę PowerShell i wykonać takie zadanie z jej poziomu.

Takie podejście zostało już przyjęte w co najmniej kilku rodzinach produktów firmy Microsoft, takich jak Exchange Server 2007 i jego nowsze wersje, SharePoint Server 2010 i późniejsze wersje, w wielu produktach System Center, Office 365 oraz wielu komponentach samego systemu Windows. Firma Microsoft idzie za ciosem i coraz więcej produktów i nowych wersji składników systemu Windows będzie podążać tą drogą. Przykładowo, Windows Server 2012, w którym zaimplementowana została powłoka PowerShell v3.0, może być niemal w stu procentach zarządzany z jej poziomu lub z poziomu interfejsu GUI będącego nakładką na powłokę PowerShell. Dlatego właśnie nie możesz ignorować powłoki PowerShell — w ciągu najbliższych kilku lat stanie się ona bowiem podstawowym narzędziem do zarządzania systemem Windows i jego aplikacjami. Co więcej, powłoka PowerShell stała się już bazą dla wielu technologii wyższego poziomu, takich jak Desired State Configuration (DSC), PowerShell Workflow i wielu innych. PowerShell jest wszędzie!

Zadaj sobie to pytanie: „Jeżeli byłbym odpowiedzialny za cały zespół administratorów IT, kogo bym chciał zatrudniać na bardziej odpowiedzialnych i lepiej opłacanych stanowiskach? Administratorów, którzy za każdym razem na wykonanie prostej operacji potrzebują co najmniej kilku minut i narzędzia z interfejsem graficznym, czy takich, którzy potrafią szybko zautomatyzować często powtarzające się zadania i następnie wykonywać je w dosłownie kilka sekund?”. Odpowiedź na takie pytanie będzie z pewnością taka sama niezależnie od tego, w jakiej części świata IT byśmy je zadali. Chcesz się o tym przekonać? Zapytaj dowolnie wybranego administratora urządzeń Cisco, operatora systemów AS/400 czy administratora systemu UNIX. W zdecydowanej większości przypadków

odpowiedź zapewne będzie brzmiała mniej więcej tak: „To chyba jasne, że wolałbym wybrać osobę, która może efektywnie działać, korzystając z poleceń powłoki!”. W przyszłości świat Windows zacznie dzielić się na dwie grupy: administratorów, którzy mogą korzystać z powłoki PowerShell, i tych, którzy jej nie znają.

Bardzo się cieszymy, że zdecydowałeś się poznać możliwości powłoki PowerShell i nauczyć się z niej korzystać!

1.2. Od teraz już tylko PowerShell!

W połowie 2016 roku firma Microsoft podjęła trudną do wyobrażenia sobie wcześniej decyzję o udostępnieniu kodu powłoki Windows PowerShell jako projektu *open source*. W tym samym czasie firma ta wypuściła również nowe wersje powłoki PowerShell (już bez słowa „Windows” w nazwie) działające na platformach macOS i licznych dystrybucjach systemu Linux. Niesamowicie! Dzięki temu ta ukierunkowana obiektowo powłoka jest dostępna w wielu systemach operacyjnych i może być rozwijana i ulepszana przez ogółnoświatową społeczność użytkowników. Z tego względu, pracując nad tą książką, postanowiliśmy również pokazać, że powłoka PowerShell działa także na innych platformach niż system Windows. Oczywiście nadal uważamy, że największymi odbiorcami powłoki PowerShell będą użytkownicy wspomnianego systemu, ale po prostu chcieliśmy się również upewnić, że będziesz wiedział, jak ta powłoka działa w innych systemach operacyjnych.

1.3. Czy ta książka jest dla Ciebie?

Nasza książka z założenia nie miała być uniwersalnym kompendium wiedzy dla wszystkich użytkowników. Zespół firmy Microsoft, który pracuje nad zastosowaniami i rozwojem powłoki PowerShell, w nieco swobodny sposób definiuje trzy rodzaje użytkowników korzystających z tej powłoki:

- Administratorzy, którzy najczęściej tylko uruchamiają różne dostępne polecenia i korzystają z narzędzi napisanych przez innych.
- Administratorzy, którzy łączą polecenia i narzędzia w bardziej złożone procesy oraz tworzą z nich proste narzędzia, których mogą używać inni, mniej doświadczeni administratorzy.
- Doświadczeni administratorzy i programiści tworzący złożone narzędzia i aplikacje wielokrotnego użytku.

Ta książka jest przeznaczona przede wszystkim dla tej pierwszej grupy użytkowników. Wychodzimy tutaj z założenia, że każdy użytkownik, a nawet programista, powinien dobrze zrozumieć, w jaki sposób można używać powłoki do wykonywania poleceń i realizacji różnych zadań. W końcu, jeżeli naprawdę zamierzasz tworzyć swoje własne polecenia i narzędzia, powinieneś dobrze poznać funkcjonalność powłoki PowerShell oraz używać sprawdzonych wzorców pozwalających na efektywne korzystanie z jej możliwości.

Jeśli jesteś zainteresowany tworzeniem skryptów do automatyzacji złożonych procesów, takich jak zarządzanie kontami użytkowników, będziesz mógł to robić po przeczytaniu tej książki. Dowiesz się nawet, jak możesz tworzyć własne polecenia, z których będą mogli korzystać inni administratorzy. Pamiętaj jednak, że z oczywistych względów w naszej książce nie będziemy w stanie dokładnie omówić wszystkich możliwości powłoki PowerShell. Naszym celem jest po prostu sprawienie, abyś mógł efektywnie korzystać z tej powłoki w swoim środowisku produkcyjnym.

Oprócz tego przedstawimy także kilka sposobów używania powłoki PowerShell do korzystania z zewnętrznych technologii zarządzania i innych elementów — WMI czy wyrażenia regularne to dwa przykładowe zagadnienia, które jako pierwsze przychodzą nam na myśl. W kolejnych rozdziałach przedstawimy pokrótce takie technologie i pokażemy, jak można z nich korzystać z poziomu powłoki PowerShell. Oczywiście takie tematy zdecydowanie zasługują na swoje własne książki (których już jest całe mnóstwo), więc skoncentrujemy się wyłącznie na ich zagadnieniach dotyczących powłoki PowerShell. Przekażemy Ci sugestie związane z dalszymi poszukiwaniami które to sugestie będziesz mógł wykorzystywać podczas samodzielnego korzystania z tych technologii. Krótko mówiąc, nie mieliśmy zamiaru, abyś ucząc się korzystać z powłoki PowerShell, nie musiał sięgać już po żadną inną książkę — zamiast tego nasza książka ma być po prostu znakomitym pierwszym krokiem prowadzącym Cię do osiągnięcia celu.

1.4. Jak korzystać z tej książki

Założeniem naszej książki było to, że będziesz czytał jeden rozdział każdego dnia. Nie musisz oczywiście tego robić podczas lunchu, ale całość materiału została rozplanowana tak, aby przeczytanie każdego rozdziału zajęło Ci około 40 minut, dzięki czemu pozostanie Ci dodatkowe 20 minut na dokończenie kanapki, wypicie kawy i przeciwieźnienie materiału omawianego w danym rozdziale.

1.4.1. Główne rozdziały

Zdecydowana większość materiału została umieszczona w rozdziałach od 2. do 25., co przekłada się na zestaw 24 lunchów, podczas których możesz poznawać tajniki powłoki PowerShell. Możesz oczekiwać, że przy takim założeniu będziesz w stanie przyswoić omawiane zagadnienia mniej więcej w ciągu miesiąca. Staraj się trzymać tego harmonogramu na tyle, na ile to możliwe, i nie zmuszaj się do czytania dodatkowych rozdziałów w danym dniu. Ważniejsze jest to, abyś poświęcał trochę czasu na ćwiczenia z materiałem omawianym w danym rozdziale, ponieważ pomoże Ci to ugruntować to, czego się nauczyłeś. Nie każdy rozdział wymaga pełnej godziny, więc czasami przed powrotem do pracy będziesz mógł poświęcić nieco dodatkowego czasu na ćwiczenia z powłoką (i być może zjedzenie obiadu). Uważamy, że wiele osób uczy się szybciej, gdy trzyma się tylko jednego rozdziału dziennie, ponieważ daje to czas na zastanowienie się nad nowymi pomysłami i na samodzielne ćwiczenia. Nie spiesz się, a być może okaże się, że poruszasz się szybciej, niż myślałeś, że to możliwe.

1.4.2. Ćwiczenia praktyczne

W większości głównych rozdziałów znajdziesz krótkie ćwiczenia praktyczne i zadania do samodzielnego wykonania. Oczywiście otrzymasz do nich szczegółowe instrukcje, a być może i jakąś podpowiedź lub nawet dwie. Odpowiedzi do ćwiczeń zamieszczone są na końcu każdego rozdziału, ale postaraj się wykonywać poszczególne ćwiczenia bez patrzenia na odpowiedzi.

1.4.3. Przykładowe kody

W książce znajdziesz cały szereg listingów zawierających kody nieco dłuższych, przykładowych skryptów powłoki PowerShell. Nie musisz jednak ich mozolnie przepisywać — gotowe pliki z kodami skryptów możesz pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/wipom3.zip>.

1.4.4. Materiały uzupełniające

Na kanale YouTube Dona, dostępnym pod adresem <https://www.youtube.com/PowerShellDon>, znajdziesz szereg ciekawych filmów, które przygotował dla pierwszego wydania tej książki, a które nadal są w stu procentach aktualne. To świetny sposób na szybką i ciekawą prezentację wybranych zagadnień. Na jego kanale znajdziesz również kilka klipów wideo nagranych podczas różnych warsztatów konferencyjnych i nie tylko. Zapewniamy, że wszystkie jego filmy są warte obejrzenia. Z pewnością warto również zajrzeć na kanał *PowerShell.org*, dostępny pod adresem <https://www.youtube.com/powershellorg>, na którym to kanale znajdziesz mnóstwo interesujących klipów wideo, takich jak nagrane sesje z konferencji PowerShell + DevOps Global Summit, webinaria społeczności użytkowników i wiele innych materiałów. Wszystko za darmo!

Z kolei Jeff publikuje wiele artykułów w serwisie Petri IT Knowledgebase (<https://www.petri.com/>), gdzie znajdziesz ogromną kolekcję materiałów obejmujących wszystkie zagadnienia związane z powłoką PowerShell. Od czasu do czasu możesz również sprawdzić, czy Jeff nie udostępnił czegoś nowego na swoim kanale na YouTube, dostępnym pod adresem <https://www.youtube.com/jdhitsolutions>¹.

1.4.5. Dalsze pogłębianie swojej wiedzy

W niektórych rozdziałach tej książki z oczywistych powodów byliśmy w stanie tylko bardzo pobieżnie omówić niektóre fajne technologie, ale w takich sytuacjach zawsze kończyliśmy te rozdziały sugestiami dotyczącymi dalszego, samodzielnego ich odkrywania. Wskazujemy tam wybrane dodatkowe źródła i zasoby wiedzy, które w razie potrzeby będziesz mógł wykorzystać do zdobycia nowych umiejętności i poszerzenia swojej wiedzy na temat tych technologii.

¹ Wszystkie opisane w tym punkcie materiały dostępne są w języku angielskim — *przyp. red.*

1.4.6. Dla zainteresowanych

Kiedy uczyliśmy się pracy z powłoką PowerShell, bardzo często chcieliśmy spojrzeć na dane zagadnienie z innej perspektywy i sprawdzić, dlaczego coś działa właśnie tak, jak działa. W taki sposób nie nabyliśmy co prawda zbyt wielu dodatkowych, praktycznych umiejętności, ale zdobyliśmy bardzo cenną wiedzę na temat tego, czym tak naprawdę jest i jak działa powłoka PowerShell. Wiele z tych informacji zamieściliśmy w tej książce w sekcjach „Dla zainteresowanych”. Przeczytanie żadnej z tych sekcji nie zajmie Ci więcej niż kilka minut i może Ci dostarczyć wielu ciekawych informacji, zwłaszcza jeżeli jesteś osobą, która lubi dokładnie wiedzieć, dlaczego coś działa tak, a nie inaczej. Jeżeli jednak uważasz, że takie sekcje mogą odciągnąć Cię od zdobywania praktycznych umiejętności pracy z powłoką, możesz śmiało zignorować je podczas pierwszego czytania — zawsze możesz do nich wrócić później, po opanowaniu głównego materiału rozdziału.

1.5. Konfigurowanie środowiska testowego

Pracując z tą książką, będziesz wykonywać wiele ćwiczeń z powłoką Windows PowerShell, zatem potrzebne Ci będzie odpowiednie środowisko testowe — zdecydowanie nie powinieneś eksperymentować w produkcyjnym środowisku swojego miejsca pracy!

Do uruchomienia zdecydowanej większości przykładów omawianych w tej książce i ukończenia wszystkich ćwiczeń potrzebujesz jedynie kopii systemu Windows z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej. Sugerujemy użycie systemu Windows 8.1 lub nowszego, ewentualnie systemu Windows Server 2012 R2 lub nowszego, które dostarczane są z powłoką PowerShell v4. Pamiętaj, że powłoka PowerShell może nie być dostępna w niektórych wersjach systemu Windows, takich jak wersje Starter. Jeśli naprawdę chcesz poznać możliwości powłoki PowerShell i nauczyć się nią posługiwać, musisz zainwestować w taką wersję systemu Windows, która ją posiada. Powinieneś również pamiętać, że niektóre ćwiczenia opierają się na nowych funkcjach i mechanizmach systemów Windows 8 lub Windows Server 2012, więc jeśli używasz starszej wersji systemu Windows, to niektóre elementy mogą działać nieco inaczej (lub nie działać wcale). Na początku każdego ćwiczenia informujemy, jakiego systemu operacyjnego będziesz potrzebować do jego ukończenia.

Pamiętaj, że w tej książce zakładamy, że pracujesz na 64-bitowym systemie operacyjnym, oznaczanym często jako system operacyjny *x64*. W takich systemach znajdziesz dwie kopie powłoki Windows PowerShell oraz graficznie zorientowane środowisko skryptowe Windows PowerShell, znane jako Integrated Scripting Environment (ISE). W menu *Start* systemu Windows 7 (lub na ekranie *Start* systemu Windows 8) 64-bitowe wersje powłoki są wymienione jako *Windows PowerShell* i *Windows PowerShell ISE*. Wersje 32-bitowe są wyróżnione przez dodanie identyfikatora (*x86*) w nazwie skrótu, a ciąg znaków (*x86*) pojawia się również na pasku tytułu okna po uruchomieniu tych wersji. Jeśli korzystasz z 32-bitowego systemu operacyjnego, będziesz mieć tylko 32-bitową wersję Powershella, która nie będzie posiadała oznaczenia (*x86*).

Przykłady w tej książce oparte są na wykorzystaniu 64-bitowych wersji powłoki PowerShell i środowiska ISE. Jeśli korzystasz z innych wersji powłoki, podczas wykonywania

ćwiczeń i przykładów możesz czasami uzyskać nieco inne wyniki, a w niektórych sytuacjach opisywane polecenia mogą nawet nie działać poprawnie. 32-bitowe wersje powłoki zostały udostępnione przede wszystkim dla zachowania kompatybilności wstecznej. Na przykład niektóre rozszerzenia powłoki są dostępne tylko w wersjach 32-bitowych i mogą być wykorzystywane wyłącznie w 32-bitowej wersji powłoki. Jeżeli nie musisz korzystać z takich rozszerzeń, zdecydowanie zalecamy korzystanie z 64-bitowej wersji powłoki w 64-bitowym systemie operacyjnym. Firma Microsoft rozwija i w przyszłości będzie rozwijała przede wszystkim rozwiązania 64-bitowe; jeśli utkniesz na 32-bitowym systemie operacyjnym, to wcześniej czy później to Cię niestety powstrzyma.

WSKAZÓWKA W praktyce do wykonania wszystkich omawianych w naszej książce ćwiczeń i przykładów powinien Ci w zupełności wystarczyć jeden komputer wyposażony w powłokę PowerShell, chociaż niektóre rzeczy stają się zdecydowanie bardziej interesujące, jeżeli masz do dyspozycji dwa lub trzy komputery w jednej sieci. W naszym przypadku użyliśmy usługi CloudShare (<https://www.cloudshare.com>) jako taniego sposobu na uruchomienie kilku wirtualnych maszyn w chmurze. Jeśli interesuje Cię taki scenariusz, sprawdź tę usługę lub skorzystaj z podobnego rozwiązania. Pamiętaj również, że usługa CloudShare w niektórych krajach może być niedostępna. Jeżeli korzystasz z systemu Windows 8 lub nowszego, możesz również skorzystać z funkcji Hyper-V i uruchomić na jednym komputerze fizycznym kilka maszyn wirtualnych.

Jeśli chcesz korzystać z powłoki PowerShell na platformie innej niż Windows, to nie musisz się martwić. Po prostu pobierz odpowiednią kompilację dla swojej wersji systemu macOS, Linux lub innych ze strony <https://github.com/PowerShell/PowerShell>, zainstaluj ją w swoim systemie i gotowe. Powinieneś jednak pamiętać, że wiele funkcji, których będziemy używać w naszych przykładach, dotyczy wyłącznie systemu Windows. Przykładowo, w systemie Linux nie można wyświetlić listy usług, ponieważ nie ma w nim usług (co prawda działają tam odgrywające podobną rolę daemony, ale to jednak nie to samo).

1.6. Instalowanie Windows PowerShell

Powłoka Windows PowerShell v3 była dostępna dla większości wersji systemu Windows począwszy od pojawienia się systemów Windows Server 2008, Windows Server 2008 R2, Windows 7 i nowszych wersji. W systemie Windows Vista powłoka PowerShell v3 nie jest obsługiwana, ale nadal może działać w wersji 2. Powłoka PowerShell jest preinstalowana tylko w najnowszych wersjach systemu Windows; w starszych wersjach należy ją zainstalować ręcznie. Powłoka PowerShell v4 jest dostępna dla systemu Windows 7 i nowszych wersji oraz systemu Windows Server 2008 R2 lub nowszego. Warto jednak zauważyć, że w tych wersjach systemu niektóre jego komponenty nie są jeszcze „podłączone” do powłoki PowerShell, dlatego jako minimalną wersję zalecamy system Windows 8 lub Windows Server 2012 — chociaż PowerShell v4 nie jest najnowszą wersją tej powłoki, to jednak w zupełności wystarczy ona do wykonania wszystkich ćwiczeń i przykładów z tej książki.

WSKAZÓWKA Jeżeli chcesz sprawdzić swoją wersję powłoki PowerShell, powinieś uruchomić konsolę tej powłoki, a następnie wpisać polecenie `$PSVersionTable` i nacisnąć klawisz *Enter*. Jeżeli wynikiem działania tego polecenia będzie błąd lub właściwość `PSVersion` będzie miała wartość inną niż 4.0, to będzie to oznaczało, że używasz innej wersji powłoki niż PowerShell v4.

Jeśli chcesz sprawdzić lub pobrać najnowszą dostępną wersję powłoki PowerShell, przejdź na stronę <http://msdn.microsoft.com/powershell>. Jest to oficjalna strona powłoki PowerShell, na której znajdziesz łącza do najnowszego instalatora WMF (ang. *Windows Management Framework*), pozwalającego na zainstalowanie wspomnianej powłoki i powiązanych z nią technologii. Ze względu na fakt, że w naszej książce omawiamy tylko podstawowe zagadnienia powłoki PowerShell, przekonasz się, że w porównaniu z v3 nie zmieniło się zbyt wiele, ale zawsze fajnie jest mieć najnowszą wersję.

Powłoka PowerShell ma dwa komponenty aplikacji: standardową konsolę tekstową (*PowerShell.exe*) i wyposażone w graficzny interfejs użytkownika środowisko ISE (*PowerShell ISE.exe*). W zdecydowanej większości przykładów i ćwiczeń będziemy korzystać z konsoli tekstowej, ale jeżeli chcesz, możesz równie dobrze używać środowiska ISE.

UWAGA Środowisko *PowerShell ISE* nie jest preinstalowane w serwerowych systemach operacyjnych. Jeśli chcesz z niego skorzystać, musisz uruchomić program *Server Manager* (menedżer serwera), przejść do opcji *Windows Features* (funkcje systemu Windows) i ręcznie dodać funkcję ISE. Zamiast tego możesz również uruchomić tekstową konsolę powłoki PowerShell i wykonać polecenie `Add-WindowsFeature powershell-ise`. Środowisko ISE w ogóle nie jest dostępne w takich instalacjach serwerowych, które nie mają pełnego interfejsu GUI (na przykład *Server Core* lub *Nano Server*).

Zanim przejdziemy dalej, powinieś poświęcić kilka minut na dostosowanie powłoki do własnych potrzeb. Jeśli używasz konsoli tekstowej, zdecydowanie zalecamy zmianę domyślnej czcionki konsoli na czcionkę o stałej szerokości (na przykład *Lucida*). Domyślna czcionka bardzo utrudnia odróżnienie niektórych znaków interpunkcyjnych, których używa powłoka PowerShell. Aby dostosować czcionkę, powinieś wykonać następujące operacje:

1. Kliknij ikonę powłoki PowerShell, znajdującą się w lewym górnym rogu okna konsoli, i z menu podręcznego, które pojawi się na ekranie, wybierz polecenie *Właściwości*.
2. Na ekranie ukaże się okno dialogowe właściwości powłoki, w którym na poszczególnych kartach możesz zmieniać czcionki, kolory, rozmiar okna, jego położenie i inne elementy.

WSKAZÓWKA Zdecydowanie zalecamy, abyś się upewnił, że opcje *Rozmiar okna* oraz *Rozmiar bufora ekranu* mają ustawioną taką samą wartość właściwości *Szerokość*.

Wprowadzone zmiany zostaną zastosowane do konsoli domyślnej, co oznacza, że pozostaną aktywne po otwarciu nowych okien. Oczywiście wszystko to dotyczy tylko systemu Windows — w innych systemach operacyjnych zwykle instalujesz powłokę PowerShell, uruchamiasz wiersz poleceń systemu operacyjnego (na przykład powłokę Bash) i wykonujesz polecenie `powershell`. Wygląd konsoli PowerShell, jej kolory i układ ekranu będzie determinowany przez ustawienia konsoli systemu, które możesz zmienić tak, aby dostosować je do własnych preferencji.

1.7. Kontakt z nami

Pomaganie ludziom takim jak Ty w nauce posługiwania się konsolą Windows PowerShell stało się naszą pasją i dlatego staramy się dostarczyć Ci jak najwięcej zasobów i źródeł wiedzy. Doceniamy również Twoje opinie, ponieważ dzięki nim możemy opracować nowe pomysły i elementy, które możemy dodać do naszej witryny oraz wykorzystać do ulepszenia przyszłych wydań tej książki. Z Donem możesz skontaktować się na Twitterze pod adresem [@concentratedDon](https://twitter.com/concentratedDon), a z Jeffem — pod adresem [@JeffHicks](https://twitter.com/JeffHicks). Obaj spędzają również wolny czas na forach <http://PowerShell.org>, gdzie możesz zadawać im pytania dotyczące powłoki PowerShell i nie tylko. Witryna <http://PowerShell.org> to wspaniałe miejsce, gdzie możesz znaleźć wiele ciekawych zasobów, w tym darmowe e-booki, informacje o dorocznych konferencjach, bezpłatne seminaria internetowe i wiele innych. Staramy się pomagać w prowadzeniu tej witryny i serdecznie zachęcamy do jej odwiedzania po ukończeniu lektury tej książki.

1.8. Natychmiastowe działanie z powłoką PowerShell

Natychmiastowe działanie to fraza, która stała się naszym hasłem przewodnim dla całej książki. W miarę możliwości każdy rozdział skupia się na czymś, co można od razu wykorzystać w rzeczywistym środowisku produkcyjnym. Oznacza to, że od czasu do czasu ględzimy co prawda trochę o niektórych detalach, ale zawsze, kiedy to możliwe, zabieramy się do rzeczy i obiecujemy, że powrócimy do omawiania szczegółów danego zagadnienia we właściwym czasie. W wielu sytuacjach musieliśmy wybierać między zasypaniem czytelnika dwudziestoma stronami teorii a pójściem na skróty i wykonaniem zadania bez wyjaśniania wszystkich niuansów, zastrzeżeń i szczegółów. Kiedy pojawiały się takie dylematy, prawie zawsze decydowaliśmy się na drogę na skróty, tak abyś *natychmiast mógł przejść do skutecznego działania*. Oczywiście nie zmieniło to jednak w niczym faktu, że wszystkie te ważne szczegóły i niuanse zostały dokładnie objaśnione we właściwym czasie w innych miejscach książki.

No cóż, zatem wystarczy już tego gadania — przyszła pora na natychmiastowe działanie. Jeżeli masz gdzieś pod ręką swój lunch, to najwyższy już czas, abyś rozpoczął swoją pierwszą lekcję.

Poznaj powłokę PowerShell

W tym rozdziale chodzi o to, abyś zaprzyjaźnił się z powłoką PowerShell i zdecydował, którego interfejsu tej powłoki będziesz używał (tak, masz wybór!). Jeśli korzystałeś już z powłoki PowerShell, ten materiał może wydać Ci się zbędny, choć mimo to warto przejrzeć ten rozdział — możesz znaleźć tu kilka ciekawostek, które pomogą Ci w pracy z powłoką.

Niniejszy rozdział dotyczy wyłącznie powłoki PowerShell w systemie Windows. Wersje działające na innych platformach niż Windows nie mają tylu opcji i możliwości, więc jeżeli jesteś w takiej sytuacji, możesz spokojnie pominąć ten rozdział.

2.1. Wybierz swoją broń

W systemie Windows firma Microsoft oferuje dwa sposoby pracy z powłoką PowerShell (a w zasadzie nawet cztery, jeżeli jesteś bardzo wybredny). Na rysunku 2.1 pokazano stronę aplikacji ekranu *Start* z czterema ikonami powłoki PowerShell (wyróżniliśmy je obramowaniem, tak aby łatwiej można je było dostrzec).

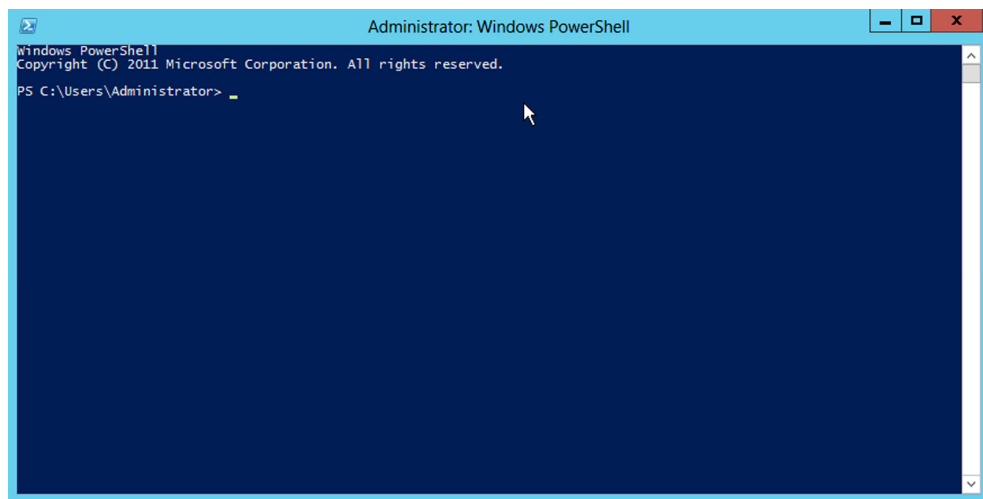
WSKAZÓWKA W starszych wersjach systemu Windows wspomniane ikony znajdują się w menu *Start*. Aby je odnaleźć, powinieneś wybrać polecenie *Wszystkie programy/Akcesoria/Windows PowerShell*. Zamiast tego możesz również wybrać z menu *Start* polecenie *Uruchom*, wpisać `powershell.exe` i nacisnąć klawisz *Enter*, co spowoduje uruchomienie konsoli powłoki PowerShell. W systemach Windows 8 i Windows Server 2012 lub nowszych przytrzymaj klawisz *Windows* na klawiaturze, naciśnij klawisz *R*, a na ekranie pojawi się okno dialogowe *Uruchom*. Aby szybko dostać się do ikon powłoki PowerShell, naciśnij i zwolnij klawisz *Windows*, a następnie w polu wyszukiwania zacznij pisać `powershell`.

WSKAZÓWKA W systemach 64-bitowych można bardzo łatwo przypadkowo uruchomić niewłaściwą wersję aplikacji. Powinieneś wyrobić w sobie nawyk spoglądania na pasek tytułu okna aplikacji — jeżeli w nazwie okna znajdziesz ciąg znaków x86, oznacza to, że uruchamiasz aplikację 32-bitową. Rozszerzenia 64-bitowe nie będą działać w 32-bitowej wersji powłoki (a zdecydowana większość nowych rozszerzeń to wersje 64-bitowe). Dobrym, rekomendowanym przez nas rozwiązaniem jest przypięcie skrótu do wybranej wersji powłoki bezpośrednio do menu *Start*.

2.1.1. Okno konsoli

Na rysunku 2.2 przedstawiono okno konsoli, które jest miejscem, gdzie większość użytkowników po raz pierwszy styka się z powłoką PowerShell. Nieco przewrotnie rozpoczniemy tę sekcję od przedstawienia kilku argumentów przemawiających przeciwko używaniu tekstowej konsoli powłoki PowerShell:

- Konsola tekstowa nie obsługuje zestawów znaków dwubajtowych, a zatem wiele tekstów napisanych w językach innych niż angielski nie będzie wyświetlanych poprawnie.
- Operacje z użyciem schowka systemowego (kopiowanie i wklejanie) wykorzystują niestandardowe kombinacje klawiszy, do których trudno się przyzwyczaić.
- Konsola tekstowa zapewnia tylko niewielką pomoc przy wpisywaniu poleceń z klawiatury (w porównaniu z konsolą ISE, którą omówimy w dalszej części tego rozdziału), chociaż w powłoce PowerShell v5 wygląda to znacznie lepiej. Warto również zauważyć, że w systemie Windows 10 firma Microsoft wprowadziła do powłoki PowerShell wiele poprawek naprawiających niektóre z istniejących od dawna problemów, więc Twoje wrażenia mogą być nieco inne.

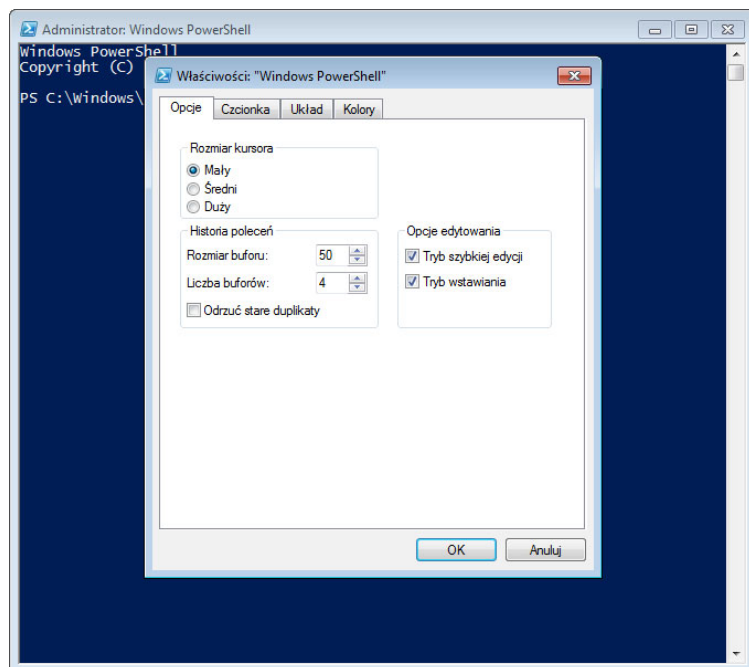


Rysunek 2.2. Standardowe okno tekstowej konsoli powłoki PowerShell — PowerShell.exe

Wymienione wyżej niedogodności nie zmieniają w niczym faktu, że konsola tekstowa jest jedynym rozwiązaniem, gdy chcesz uruchomić powłokę PowerShell na serwerach, które nie posiadają zainstalowanego graficznego interfejsu użytkownika (takich jak przykładowo serwery Server Core, Nano Server lub dowolna instancja serwera Windows Server, w której interfejs GUI został usunięty lub nie został zainstalowany). Po stronie plusów tekstowej konsoli powłoki PowerShell możemy wymienić:

- Aplikacja konsoli tekstowej ma niewielkie rozmiary, dzięki czemu szybko się ładuje i nie zużywa dużo pamięci.
- Nie wymaga żadnych innych komponentów z platformy .NET Framework oprócz tych, których potrzebuje sama powłoka PowerShell.
- W razie potrzeby możesz zmienić ustawienia kolorów na przykład na zielony tekst na czarnym tle i udawać, że pracujesz na komputerze klasy mainframe z lat 70. ubiegłego stulecia.

Jeśli zdecydujesz się na korzystanie z tekstowej konsoli powłoki PowerShell, zapoznaj się z kilkoma naszymi sugestiami dotyczącymi jej konfiguracji. Aby wprowadzić opisane niżej zmiany w ustawieniach, powinieneś kliknąć ikonę powłoki znajdującą się w lewym górnym rogu okna konsoli i z menu podręcznego wybrać polecenie *Właściwości*. Na ekranie pojawi się okno dialogowe przedstawione na rysunku 2.3. W systemie Windows 10 okno to wygląda nieco inaczej (znajdziesz tam kilka nowych opcji), ale ogólnie sposób postępowania jest taki sam.



Rysunek 2.3. Zmiana ustawień konfiguracyjnych okna konsoli

Na karcie *Opcje* możesz zwiększyć rozmiar bufora historii poleceń (opcja *Historia poleceń/Rozmiar buforu*), który pozwala konsoli zapamiętywać wpisane polecenia i umożliwia ich przywoływanie z poziomu klawiatury za pomocą strzałek w górę i w dół. Listę wcześniej wykonanych poleceń możesz także wyświetlić, naciskając klawisz *F7*.

Na karcie *Czcionka* zmien rozmiar czcionki na nieco większy niż domyślne 12 pkt. Prosimy. Naprawdę nie obchodzi nas, czy masz sokoli wzrok — po prostu zaufaj nam i zwiększ nieco rozmiar czcionki. Pracując z powłoką PowerShell, będziesz musiał często szybko odróżniać od siebie wiele podobnych znaków, takich jak `'` (apostrof) czy ``` (odwrócony apostrof), a mała czcionka wcale w tym nie pomaga.

Na karcie *Układ* ustaw obie opcje *Szerokość* na tę samą wartość i upewnij się, że otrzymane okno mieści się na ekranie. Jeżeli tego nie zrobisz, może się okazać, że w dolnej części okna pojawi się poziomy pasek przewijania, a niektóre, „szerokie” wyniki działania poleceń powłoki będą się pojawiały poza prawą krawędzią okna i nie będą widoczne. W czasie prowadzonych przez nas szkoleń zdarzało się, że nasi studenci spędzali sporo czasu na uporczywych próbach zmuszania do działania jakiegoś pozornie niedziałającego polecenia, które w rzeczywistości działało jak należy, wyświetlając wyniki poza krawędzią ekranu. To może być irytujące.

Na koniec przejdź na kartę *Kolory* i postaraj się zachować umiar. Pozostaw w miarę wysoki kontrast i zachowaj łatwość czytania. Jeżeli nie podoba Ci się domyślne ustawienie kolorów (biała czcionka na niebieskim tle), to pamiętaj, że na przykład czarna czcionka na średnioszarym tle też będzie wyglądać całkiem nieźle.

Powinieneś pamiętać o jeszcze jednej sprawie: aplikacja, którą uruchamiasz w konsoli tekstowej, to nie jest powłoka PowerShell — to tylko narzędzie, za pomocą którego wchodzisz w interakcję z powłoką PowerShell. Sama aplikacja konsolowa ma swoje korzenie w połowie lat 80. ubiegłego stulecia, a co za tym idzie, jest dosyć prymitywna i nie powinieneś oczekiwać, że zrobi na Tobie ogromne wrażenie.

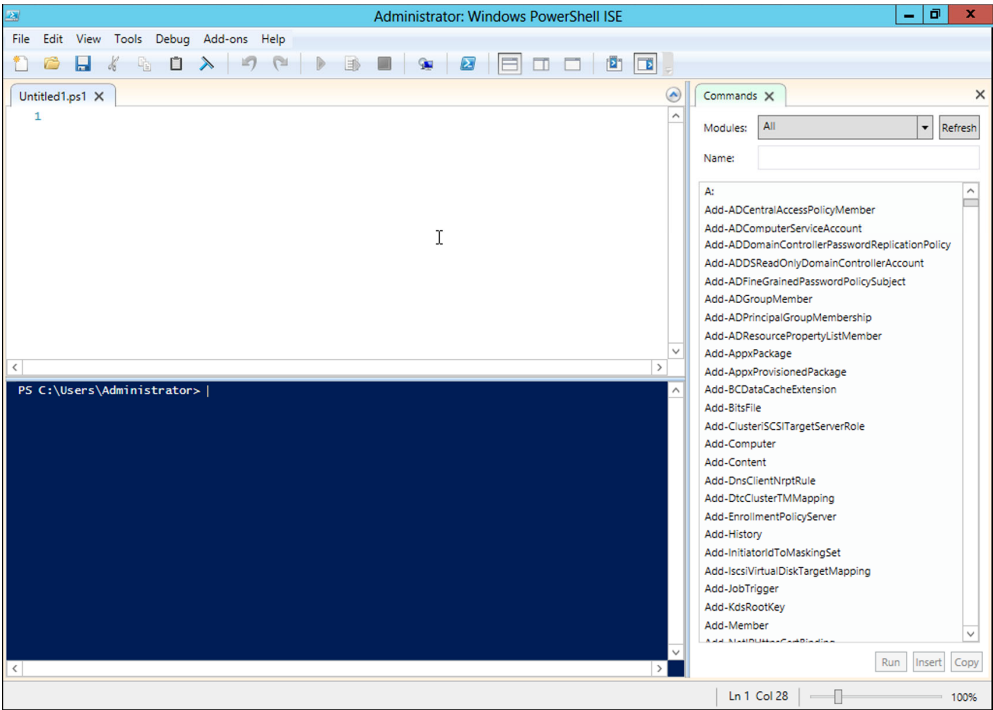
2.1.2. Zintegrowane środowisko skryptowe

Na rysunku 2.4 przedstawiono wygląd środowiska *PowerShell Integrated Scripting Environment* (ISE).

WSKAZÓWKA Jeżeli przypadkowo uruchomisz standardową aplikację konsolową, to możesz uruchomić środowisko ISE, wpisując polecenie `ise` i naciskając klawisz *Enter*.

Istnieje wiele zagadnień związanych ze środowiskiem PowerShell ISE, o których warto wspomnieć, ale rozpoczniemy od przedstawienia tabeli 2.1, gdzie zamieszczamy krótkie zestawienie jego najważniejszych zalet i wad.

Zacznijmy od podstaw. Na rysunku 2.5 wyróżniono trzy główne obszary okna środowiska ISE i zaznaczono obszar paska narzędzi ISE, za pomocą którego można kontrolować te obszary.

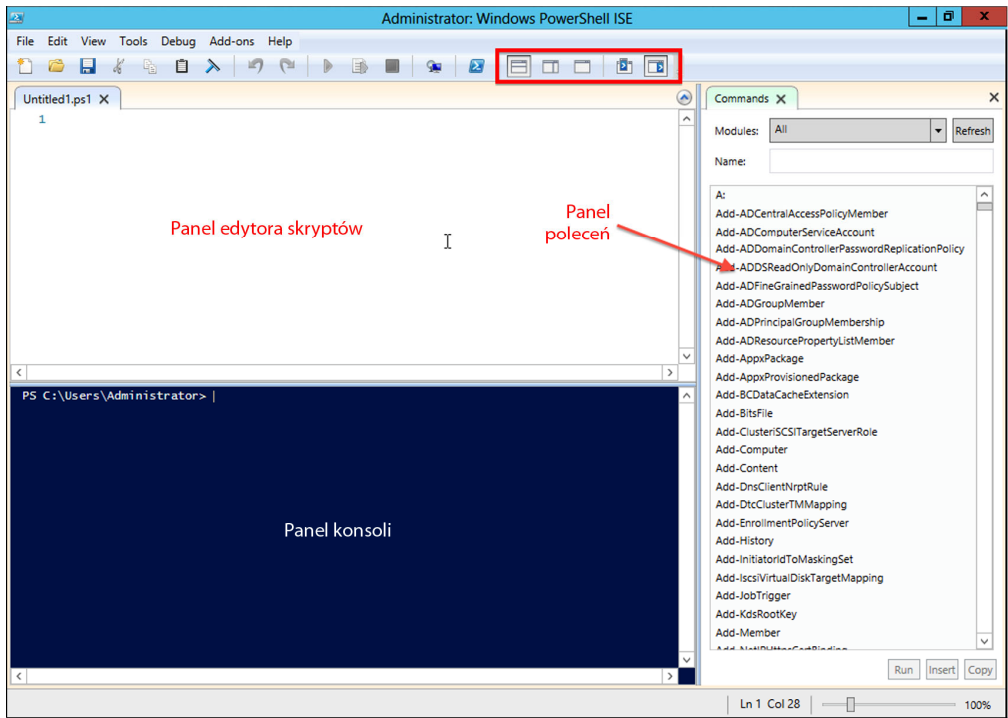


Rysunek 2.4. Środowisko PowerShell ISE (powershell_ise.exe)

Tabela 2.1. Zalety i wady środowiska PowerShell ISE

Zalety	Wady
Środowisko ISE jest znacznie ładniejsze od aplikacji konsolowej i obsługuje zestawy znaków dwubajtowych.	Wymaga obecności zainstalowanego pakietu Windows Presentation Foundation (WPF), co oznacza, że nie może działać na serwerze, na którym odinstalowano GUI (choć może działać w trybie GUI Minimal Server, który obsługuje aplikacje WPF).
Wspomaga tworzenie poleceń i skryptów powłoki PowerShell, o czym przekonasz się w dalszej części tego rozdziału.	Uruchamianie środowiska ISE zajmuje nieco więcej czasu, niż to ma miejsce w przypadku aplikacji konsolowej (aczkolwiek jest to różnica zaledwie kilku sekund).
Środowisko ISE wykorzystuje standardowe kombinacje klawiszy operacji kopiowania i wklejania.	Wersje wcześniejsze niż 5.0 nie obsługują mechanizmu transkrypcji.

Na rysunku 2.5 górny obszar to okienko edytora skryptów, którego w tej książce nie będziemy używać. W prawym górnym rogu tego panelu zauważysz małą niebieską strzałkę; kliknij ją, aby ukryć edytor skryptów i zmaksymalizować panel konsoli, czyli obszar, z którego będziemy korzystać. Po prawej stronie okna znajduje się panel poleceń, który możesz pozostawić otwarty lub zamknąć, klikając mały przycisk X w prawym górnym rogu. „Pływające” okno poleceń możesz przywołać, klikając przycisk *Show Command Window* (drugi od prawej przycisk na pasku narzędzi). Jeśli zamkniesz panel



Rysunek 2.5. Pasek narzędzi i trzy główne obszary okna środowiska ISE

poleceń i zechcesz go przywrócić, powinieneś nacisnąć przycisk *Show Command Add-on* (pierwszy od prawej na pasku narzędzi). Pierwsze trzy przyciski, które wyróżniliśmy na pasku narzędzi, sterują układem edytora skryptów i panelu konsoli. Możesz ustawić te panele jeden nad drugim, obok siebie lub otworzyć pełnoekranowy panel edytora skryptów.

W prawym dolnym rogu okna środowiska ISE znajdziesz suwak, za pomocą którego możesz zmieniać rozmiar czcionki. W menu *Tools* (narzędzia) znajdziesz polecenie *Options* (opcje), za pomocą którego można definiować niestandardowe schematy kolorów i inne ustawienia wyglądu; możesz śmiało eksperymentować z tymi ustawieniami.

ZRÓB TO SAM Przyjmujemy tutaj założenie, że podczas pracy z tą książką do tworzenia i analizowania skryptów będziesz używał wbudowanego edytora skryptów środowiska ISE. Na razie jednak ukryj okienko edytora skryptów i (jeśli chcesz) panel poleceń. Rozmiar czcionki możesz oczywiście ustawić według własnego uznania. Jeżeli domyślny schemat kolorów nie odpowiada Twoim preferencjom, również możesz go dowolnie zmodyfikować i dostosować do swoich potrzeb. Jeżeli jednak zamiast tego wolisz korzystać z aplikacji konsolowej, to oczywiście nic złego się nie wydarzy — wszystkie przykłady w książce będą nadal działać poprawnie. W kilku przypadkach będziemy jednak omawiać mechanizmy i funkcje specyficzne dla ISE, ale w takich sytuacjach zawsze Cię o tym wcześniej poinformujemy, żebyś miał szansę uruchomić środowisko ISE.

2.2. Wracamy do wpisywania poleceń z klawiatury

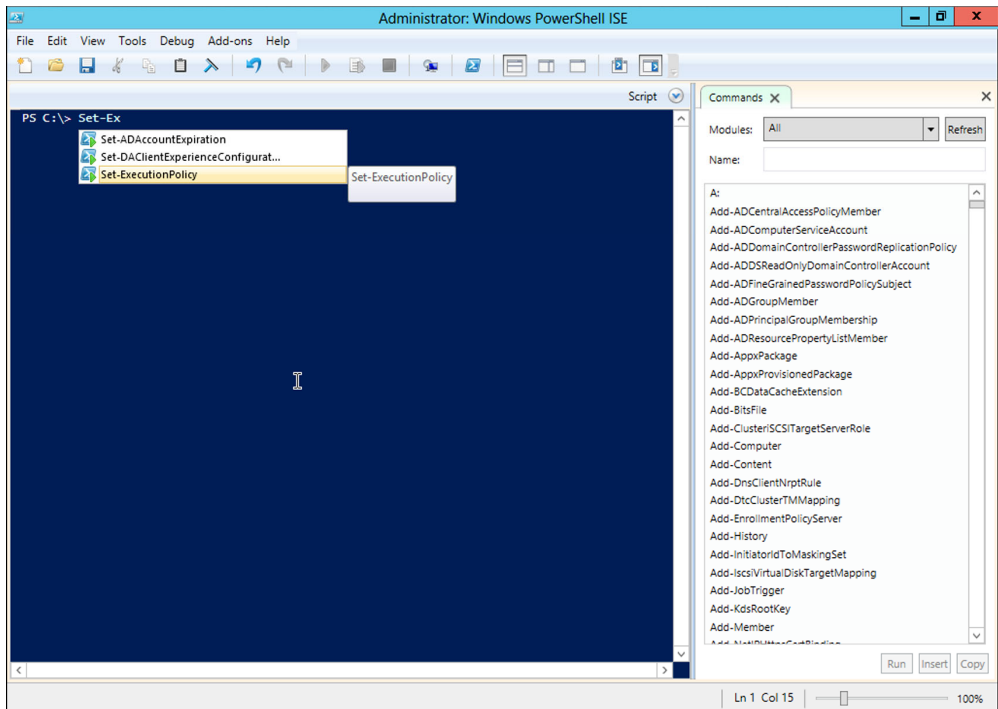
Powłoka PowerShell to interfejs wiersza poleceń, co oznacza, że będziesz wpisywać z klawiatury wiele poleceń. Podczas takiej pracy oczywiście nie sposób uniknąć prostych literówek, ale na szczęście obie aplikacje PowerShell posiadają mechanizmy wspomagające wpisywanie poleceń i minimalizowanie liczby literówek.

ZRÓB TO SAM Przykładów opisanych poniżej z oczywistych przyczyn nie możemy zilustrować w książce, ale fajnie będzie je zobaczyć w działaniu. Spróbuj je wykonać w powłoce PowerShell na swoim komputerze.

Aplikacja konsolowa powłoki PowerShell posiada mechanizm dopełniania poleceń działający w czterech obszarach:

- Wpisz `Get-S` i kilka razy naciśnij klawisz `Tab`, a następnie spróbuj nacisnąć kombinację klawiszy `Shift-Tab`. Powłoka PowerShell będzie przechodziła w obie strony przez wszystkie polecenia potencjalnie pasujące do podanego wzorca. Kontynuuj naciskanie tych klawiszy aż do momentu, kiedy znajdziesz interesujące Cię polecenie.
- Wpisz `Dir`, dodaj spację, a następnie wpisz ciąg znaków `C:\` i naciśnij klawisz `Tab`. Powłoka PowerShell rozpocznie cykliczne przechodzenie do kolejnych nazw plików i podfolderów z bieżącego folderu.
- Wpisz `Set-Execu`, naciśnij klawisz `Tab`, a następnie dodaj spację i myślnik (`-`). Zaczynaj naciskać klawisz `Tab`, aby zobaczyć, jak powłoka PowerShell przechodzi przez kolejne dostępne parametry wybranego polecenia. Można również wpisać fragment nazwy parametru (na przykład `-E`) i nacisnąć klawisz `Tab`, aby rozpocząć dopasowywanie parametrów, których nazwy rozpoczynają się od podanego ciągu znaków. Aby wyczyścić wiersz polecenia, naciśnij klawisz `Esc`.
- Ponownie wpisz `Set-Execu` i naciśnij klawisz `Tab`. Wpisz spację, a następnie `-E` i naciśnij znów klawisz `Tab`. Wpisz kolejną spację i ponownie naciśnij klawisz `Tab`. PowerShell będzie wyświetlał kolejne dozwolone wartości tego parametru, ale działa to tylko dla parametrów, które mają predefiniowany zestaw dozwolonych wartości (taki sposób dopełniania nazywamy **wyliczaniem parametrów**). Ponownie naciśnij klawisz `Esc`, aby usunąć wpisaną komendę; nie powinieneś jeszcze teraz uruchamiać tego polecenia.

Środowisko PowerShell ISE oferuje bardzo podobny mechanizm dopełniania, nazywany IntelliSense. Mechanizm ten będzie działał we wszystkich opisanych wyżej sytuacjach z użyciem klawisza `Tab`, z tym że teraz na ekranie będzie pojawiało się fajne zgrabne podręczne menu podpowiedzi, takie jak pokazane na rysunku 2.6. Zawartość menu możesz przewijać w górę lub w dół za pomocą klawiszy strzałek. Kiedy znajdziesz żądany element, możesz go wybrać, naciskając klawisz `Tab` lub `Enter`, a następnie możesz pisać dalej.



Rysunek 2.6. Mechanizm IntelliSense w środowisku ISE odgrywa tę samą rolę co klawisz Tab w konsoli

OSTRZEŻENIE To *bardzo, bardzo, bardzo, bardzo, bardzo* ważne, abyś podczas pracy z powłoką PowerShell *bardzo, bardzo, bardzo, bardzo* uważnie wpisywał wszystkie polecenia. W niektórych przypadkach pojedyncza zagubiona spacja, pominięty cudzysłów, apostrof czy inny znak mogą spowodować, że próba wykonania danego polecenia zakończy się niepowodzeniem. Jeżeli po uruchomieniu polecenia na ekranie pojawiają się komunikaty o wystąpieniu błędu, najpierw zawsze dwukrotnie sprawdź, co wpisałeś w wierszu polecenia.

Mechanizm IntelliSense działa w panelu konsoli ISE oraz w okienku edytora skryptów.

2.3. Najczęściej spotykane problemy

Omówimy teraz pokrótce kilka elementów, które w zupełnie niezamierzony sposób mogą Ci skomplikować życie podczas pracy z powłoką PowerShell.

- *Poziome paski przewijania w aplikacji konsolowej* — ten pozornie nieszkodliwy i bardzo przydatny mechanizm może czasami wprawić w konsternację nawet całkiem zaawansowanych użytkowników, o czym w ciągu wielu lat prowadzenia szkoleń przekonaliśmy się już niejednokrotnie. Skonfiguruj konsolę tak, aby nie miała poziomego paska przewijania u dołu okna. Nieco wcześniej w tym rozdziale opisywaliśmy już, jak możesz to zrobić.

- *32-bitowa i 64-bitowa wersja powłoki PowerShell* — powinieneś korzystać z 64-bitowej wersji systemu Windows i używać 64-bitowej wersji powłoki PowerShell (które nie mają w nazwie ciągu znaków (x86)). Wiemy, że dla niektórych użytkowników konieczność kupienia 64-bitowego komputera działającego pod kontrolą 64-bitowej wersji systemu Windows może nie być taka oczywista, ale jest to inwestycja, którą musisz przeprowadzić, jeżeli chcesz efektywnie korzystać z powłoki PowerShell. Większość ćwiczeń i przykładów omawianych w tej książce będzie co prawda poprawnie działać w 32-bitowej wersji powłoki PowerShell, ale szybko się przekonasz, że w środowisku produkcyjnym używanie 64-bitowej wersji tej powłoki robi poważną różnicę.
- Upewnij się, że na pasku tytułu okna powłoki PowerShell znajduje się słowo *Administrator*. Jeżeli go tam nie ma, zamknij to okno, ponownie kliknij prawym przyciskiem myszy ikonę powłoki PowerShell i z menu podręcznego wybierz polecenie *Uruchom jako administrator*. Pamiętaj, że w środowisku produkcyjnym nie zawsze będziesz mógł to zrobić, stąd w dalszej części tej książki pokażemy, jak podawać poświadczenia logowania podczas uruchamiania poleceń. Na razie musisz po prostu mieć pewność, że na pasku tytułu okna powłoki PowerShell znajduje się słowo *Administrator* — w przeciwnym razie możesz napotkać później różne problemy.

2.4. Jaka to wersja?

Sprawdzenie, której wersji powłoki PowerShell używasz, wbrew pozorom może wcale nie być takie proste, ponieważ każda kolejna wersja jest instalowana w katalogu o nazwie *1.0* (odnosi się to do wersji silnika języka skryptowego powłoki, co oznacza, że każda wersja jest zgodna z wydaniem *v1*). W powłoce PowerShell v3 i nowszych istnieje jednak łatwy sposób sprawdzenia wersji. Aby to zrobić, wpisz polecenie `$PSVersionTable` i naciśnij klawisz *Enter*:

```
PS C:\> $PSVersionTable
Name                           Value
----                           -
PSVersion                      3.0
WSManStackVersion              3.0
SerializationVersion           1.1.0.1
CLRVersion                     4.0.30319.17379
BuildVersion                   6.2.8250.0
PSCompatibleVersions           {1.0, 2.0, 3.0}
PSRemotingProtocolVersion      2.2
```

Po wykonaniu tego polecenia na ekranie wyświetlone zostaną zarówno numery wersji poszczególnych komponentów związanych z powłoką PowerShell, jak i wersja samej powłoki. Jeżeli opisane polecenie nie działa lub jeżeli parametr `PSVersion` nie pokazuje wersji *3.0* lub nowszej, oznacza to, że używasz powłoki PowerShell w wersji, która nie jest odpowiednia dla tej książki. Informacje o tym, jak pobrać i zainstalować najnowszą wersję powłoki PowerShell znajdziesz w rozdziale 1.

ZRÓB TO SAM Nie czekaj dłużej, aby zacząć korzystać z powłoki PowerShell. Zacznij od sprawdzenia numeru wersji zainstalowanej w Twoim systemie i upewnij się, że korzystasz z wersji 3.0 lub nowszej. Jeżeli tak nie jest, zanim zaczniesz coś robić, powinieneś zainstalować powłokę PowerShell w wersji co najmniej v3.

Powłokę PowerShell v3 (i jej nowsze wersje) można zainstalować równolegle obok wersji v2. Aby później uruchomić wersję v2, możesz wykonać polecenie `PowerShell.exe -version 2.0`. Możesz tak skonfigurować powłokę PowerShell, aby automatycznie używała wersji v2, jeżeli próbujesz uruchomić coś, co nie jest kompatybilne z wersją v3 (choć to bardzo rzadki przypadek). Instalator v3 nie instaluje wersji v2, zatem powłokę v2 będziesz mógł uruchomić tylko wtedy, jeżeli została wcześniej zainstalowana jako pierwsza. Instalatory wersji v2 i v3 będą nadpisywać wersję v1 (o ile oczywiście jest już zainstalowana), ponieważ te wersje nie mogą istnieć obok siebie. Nowsze wersje, takie jak v4, mogą działać w trybie v2, ale nie mają możliwości działania w innych trybach — inaczej mówiąc, powłoka PowerShell v4 nie może działać jako v3.

WSKAZÓWKA W nowych wersjach systemu Windows domyślnie instalowana jest najnowsza wersja powłoki PowerShell, choć mogą one posiadać również silnik powłoki PowerShell w wersji v2. W razie potrzeby możesz zainstalować silnik v2, wykonując z poziomu powłoki PowerShell polecenie `Add-WindowsFeature powershell-v2`. Jeżeli opcja `powershell-v2` nie jest dostępna w Twojej wersji systemu Windows, to masz pecha, ale na szczęście w tym momencie najprawdopodobniej nie będziesz jej potrzebować.

2.5. Ćwiczenia

Ponieważ są to pierwsze ćwiczenia w tej książce, poświęcimy chwilę na opisanie, w jaki sposób powinieneś je wykonywać. W każdym zestawie ćwiczeń zamieszczamy kilka zadań, które możesz wykonać samodzielnie. Czasami dodamy również jedną lub nawet kilka podpowiedzi, tak abyś mógł pójść we właściwym kierunku.

Dajemy Ci pełną gwarancję, że wszystko, co musisz wiedzieć, aby wykonać dany zestaw ćwiczeń, znajduje się w tym samym lub w poprzednim rozdziale (a jeżeli będziesz musiał skorzystać z informacji omawianych w poprzednim rozdziale, to zazwyczaj poinformujemy Cię o tym w podpowiedziach). Nikt jednak nie twierdzi, że rozwiązania poszczególnych zadań zawsze będą proste i oczywiste. Zazwyczaj zagadnienia omawiane w danym rozdziale będą pokazywały, w jaki sposób znajdować rozwiązania różnych problemów, i przez podobny proces analizy i odkrywania będziesz musiał samodzielnie przejść podczas wykonywania ćwiczeń. Początkowo może się to wydawać nieco frustrujące, ale zmuszanie się do takiej pracy w dłuższej perspektywie z całą pewnością sprawi, że odniesiesz sukces w pracy z powłoką PowerShell. Możemy Ci to obiecać.

Pamiętaj, że na końcu każdego rozdziału znajdziesz przykładowe odpowiedzi i rozwiązania zadań. Nasze propozycje nie zawsze muszą być podobne do Twoich, a będzie się tak zapewne zdarzało coraz częściej, w miarę jak będziemy przechodzić do bardziej złożonych zagadnień. Nieraz przekonasz się, że powłoka PowerShell oferuje z pół tuzina

lub nawet więcej sposobów na wykonanie niemal każdego zadania. Pokażemy Ci metody działania, z których korzystamy najczęściej, ale jeżeli wymyślisz jakieś inne rozwiązanie, to też będzie dobrze. Każde rozwiązanie prowadzące do wykonania zadania jest poprawne.

UWAGA Do wykonania tego zestawu ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Zacniemy od czegoś łatwego: chcielibyśmy tylko, abyś skonfigurował konsolę i środowisko ISE do swoich potrzeb. Wykonaj następujących pięć kroków:

1. Wybierz odpowiednie czcionki i kolory.
2. Upewnij się, że aplikacja konsoli nie ma poziomego paska przewijania u dołu. W tym rozdziale wspominaliśmy o tym co najmniej trzykrotnie, więc prawdopodobnie jest to naprawdę ważne.
3. W środowisku ISE zmaksymalizuj panel konsoli; panel poleceń możesz pozostawić lub wyłączyć według własnego uznania.
4. W obu aplikacjach wpisz pojedynczy cudzysłów (') oraz odwrócony apostrof (^) i upewnij się, że możesz je łatwo od siebie odróżnić. Na standardowych klawiaturach apostrof odwrócony znajduje się poniżej klawisza *Esc*, na tym samym klawiszu co znak tyldy (~).
5. Wpisz również znaki reprezentujące nawiasy okrągłe (), nawiasy kwadratowe [], znaki mniejszości i większości <> oraz nawiasy klamrowe {} i upewnij się, że wybrana czcionka i jej rozmiar pozwalają na łatwe i jednoznaczne rozróżnianie wpisanych znaków. Jeżeli wygląd niektórych znaków nie jest oczywisty, zmień krój czcionki lub zwiększ jej rozmiar.

W tym rozdziale pokazywaliśmy już, jak wykonać wszystkie opisane wyżej operacje, zatem nie ma potrzeby zamieszczania żadnych odpowiedzi — sam będziesz dobrze wiedział, czy wszystko wykonałeś poprawnie.

Korzystanie z systemu pomocy

W pierwszym rozdziale tej książki wspomnieliśmy, że kluczową cechą interfejsów graficznych jest to, że pozwalają użytkownikowi na samodzielną eksplorację swoich możliwości i są znacznie łatwiejsze do opanowania niż interfejsy wiersza poleceń (CLI — ang. *Command Line Interface*), takie jak konsolowa wersja powłoki PowerShell. W aplikacjach działających w konsoli tekstowej użytkownikowi jest o wiele trudniej poznać funkcje i możliwości. W rzeczywistości interfejs tekstowy powłoki PowerShell posiada naprawdę imponujące funkcje ułatwiające użytkownikowi pracę, choć na pierwszy rzut oka nie są one tak oczywiste. Jednym z takich głównych mechanizmów jest system pomocy.

3.1. System pomocy — jak poznawać polecenia powłoki PowerShell?

Teraz musimy chwilę pogłędzić, ale nie potrwa to zbyt długo, więc mimo wszystko spróbuj z nami wytrzymać.

Pracujemy w branży, w której nie kładzie się zbyt dużego nacisku na czytanie. Co prawda zawsze gdzieś pod ręką mamy słynny akronim *RTFM* (ang. *Read The Friendly Manual* — przeczytaj przyjazny podręcznik), który w czasie szkoleń od czasu do czasu staramy się podtykać naszym studentom i użytkownikom, kiedy chcemy, aby dokładniej przeczytali jakąś instrukcję. Większość administratorów ma jednak tendencję do samodzielnego odkrywania możliwości systemów i aplikacji, wykorzystując do tego celu takie komponenty jak podpowiedzi ekranowe, menu kontekstowe i inne mechanizmy spotykane powszechnie w graficznych interfejsach użytkownika. My również bardzo często tak pracujemy i wyobrażamy sobie, że zapewne robisz tak samo. Ale musimy tutaj sobie wyjaśnić jedną ważną kwestię:

Jeśli nie masz zamiaru korzystać z plików pomocy powłoki PowerShell, to praktycznie nie będziesz w stanie jej stosować w efektywny sposób. Nie nauczysz się poprawnie używać powłoki PowerShell i nie dowiesz się, jak w produktywny sposób zarządzać systemem Windows i jego komponentami. W takiej sytuacji równie dobrze możesz pozostać przy narzędziach wyposażonych w interfejs GUI.

Mamy nadzieję, że wszystko jest jasne. Oczywiście jest to bardzo mocne i jednoznaczne stanowisko, ale absolutnie prawdziwe. Wyobraź sobie, że próbujesz się dowiedzieć, jak działa konsola *Użytkownicy i komputery usługi Active Directory* lub dowolna inna konsola zarządzająca, ale robisz to bez pomocy etykiet narzędzi, odpowiedzi ekranowych, menu kontekstowego czy systemu pomocy. Próba nauczania się korzystania z powłoki PowerShell bez poświęcania czasu na przeczytanie i zrozumienie plików pomocy jest dokładnie tym samym. To trochę tak, jakbyś próbował poskładać duży zestaw mebli do samodzielnego montażu bez czytania instrukcji. Przy odrobinie wiedzy, doświadczenia i szczęścia zapewne może Ci się to udać, ale z pewnością taka próba będzie bardzo frustrująca, popełnisz wiele błędów i będziesz działał bardzo nieefektywnie. Dlaczego?

- Jeżeli chcesz wykonać jakieś zadanie i nie wiesz, jakiego polecenia użyć, to z pewnością szybko znajdziesz je dzięki systemowi pomocy. Nie dzięki Google ani Bing, ale po prostu — wbudowanemu systemowi pomocy powłoki PowerShell.
- Jeżeli podczas próby wykonania danego polecenia pojawi się jakiś błąd, system pomocy pokaże Ci, jak poprawnie uruchomić takie polecenie.
- Jeżeli chcesz połączyć wiele poleceń w celu wykonania zadania złożonego, system pomocy pokaże Ci, w jaki sposób każde z poleceń może łączyć się z innymi. Nie musisz szukać przykładów w wyszukiwarkach Google czy Bing; zamiast tego musisz po prostu nauczyć się, jak korzystać z poleceń i systemu pomocy powłoki PowerShell.

Zdajemy sobie sprawę, że nasze kazania mogą wydawać Ci się nudne, ale z doświadczenia wiemy, że co najmniej 90% problemów, z jakimi zgłaszali się nasi użytkownicy podczas szkoleń czy w pracy, mogłoby zostać szybko rozwiązanych, gdyby tylko przyszło użytkownikom na myśl odłożyć dumę na bok, wziąć głęboki oddech i przeczytać plik pomocy. Z tego właśnie powodu również Ty powinieneś uważnie przeczytać ten rozdział, tak aby nauczyć się korzystać z systemu pomocy, jaki oferuje powłoka PowerShell.

Warto to zrobić z co najmniej kilku powodów:

- Choć w naszych przykładach pokazujemy wiele poleceń, prawie nigdy nie omawiamy pełnej funkcjonalności ani wszystkich opcji i możliwości każdego polecenia. Aby efektywnie korzystać z powłoki PowerShell, powinieneś samodzielnie zapoznać się z pełnym opisem każdego polecenia, które prezentujemy w przykładach, tak aby wiedzieć, jak ono działa i do czego jeszcze można je wykorzystać.

- W ćwiczeniach możemy Ci dawać wskazówki, których poleceń powinieneś użyć do wykonania tego czy innego zadania, ale nie będziemy udzielać żadnych wskazówek dotyczących ich składni. W takich sytuacjach będziesz musiał użyć systemu pomocy i samodzielnie odkrywać składnię poszczególnych poleceń.

Możemy Ci zagwarantować, że dobre opanowanie systemu pomocy jest kluczem do zostania ekspertem w zakresie powłoki PowerShell. Oczywiście musisz sobie zdawać sprawę z tego, że nie zawsze znajdziesz tam absolutnie wszystko, co będzie Ci potrzebne, ponieważ w funkcjonowaniu wielu poleceń są liczne niuanse i opcje, które nie są całkowicie udokumentowane w systemie pomocy. Jeżeli jednak chcesz w efektywny sposób używać powłoki PowerShell do wykonywania codziennych zadań administratora systemu, to po prostu nie masz innego wyjścia i musisz do perfekcji opanować korzystanie z systemu pomocy. Mamy nadzieję, że nasza książka uczyni ten proces łatwiejszym.

Dość jednak tego ględzenia.

Polecenia i cmdlety

Powłoka PowerShell posiada wiele rodzajów poleceń wykonywalnych. Niektóre z nich nazywane są **poleceniami cmdlet** (lub po prostu **cmdletami**), niektóre są nazywane **funkcjami**, inne zaś — **przepływami pracy** (ang. *workflows*) i tak dalej. Wszystkie wymienione elementy łącznie tworzą zbiór *poleceń* powłoki PowerShell, a w systemie pomocy znajdziesz informacje praktycznie o każdym z nich. Polecenia *cmdlet* są czymś unikatowym dla powłoki PowerShell i jak się sam przekonasz, bardzo wiele uruchomianych poleceń powłoki jest właśnie poleceniami *cmdlet*. Aby jednak uniknąć zamieszania i niepotrzebnie nie komplikować sobie życia, spróbujemy konsekwentnie używać określenia *polecenie* za każdym razem, kiedy będziemy mówić ogólnie o wykonywanych komendach.

3.2. Aktualizowanie systemu pomocy

Kiedy po raz pierwszy uruchomisz system pomocy powłoki PowerShell, możesz być nieco zaskoczony, ponieważ, no cóż, tej pomocy będzie tam jak na lekarstwo. Ale poczekaj, możemy to wyjaśnić.

Począwszy od powłoki PowerShell v3 firma Microsoft wprowadziła nowy mechanizm, nazywany *updatable help* (aktualizowalny system pomocy), dzięki któremu możemy pobierać zaktualizowaną, poprawioną i rozszerzoną zawartość systemu pomocy bezpośrednio z sieci Internet. Niestety, wadą tego rozwiązania jest to, że na dzień dobry otrzymujemy jedynie mocno ograniczony zestaw plików pomocy. Jeżeli przed aktualizacją spróbujesz uzyskać pomoc dla jakiegoś polecenia, otrzymasz skrócony, automatycznie wygenerowany opis składni polecenia wraz z informacją o tym, jak pobrać i zainstalować pełną wersję systemu pomocy, co może wyglądać na przykład tak:

```
PS C:\> help Get-Service
NAME
    Get-Service
SYNTAX
    Get-Service [[-Name] <string[]>] [-ComputerName <string[]>] [-DependentServices]
    <[-RequiredServices] [-Include <string[]>] [-Exclude <string[]>] [<CommonParameters>]
```

```
Get-Service -DisplayName <string[]> [-ComputerName <string[]>] [-DependentServices]
↳[-RequiredServices] [-Include <string[]>] [-Exclude <string[]>] [<CommonParameters>]
Get-Service [-ComputerName <string[]>] [-DependentServices] [-RequiredServices] [-Include
↳<string[]>] [-Exclude <string[]>] [-InputObject <ServiceController[]>] [<CommonParameters>]
```

ALIASES

```
gsv
```

REMARKS

```
Get-Help cannot find the Help files for this cmdlet on this computer. It is displaying
↳only partial help.
-- To download and install Help files for the module that includes this cmdlet, use
↳Update-Help.
-- To view the Help topic for this cmdlet online, type: "Get-Help Get-Service
↳-Online" or go to http://go.microsoft.com/fwlink/?LinkID=113332.
```

WSKAZÓWKA Jak widać, w praktyce nie można nie zauważyć tego, że lokalny system pomocy nie jest zainstalowany. Za każdym razem, kiedy spróbujesz wyświetlić pomoc dla jakiegoś polecenia, powłoka PowerShell poprosi Cię o zaktualizowanie systemu pomocy.

Aktualizacja systemu pomocy powłoki PowerShell powinna być jednym z Twoich pierwszych zadań. Pliki pomocy są przechowywane w katalogu *System32*, co oznacza, że powłoka musi działać z podniesionymi uprawnieniami. Jeżeli na pasku tytułowym powłoki PowerShell nie widnieje ciąg znaków *Administrator*, to najprawdopodobniej próba zaktualizowania systemu pomocy spowoduje pojawienie się komunikatu o wystąpieniu błędu:

```
PS C:\> update-help
Update-Help : Failed to update Help for the module(s)
'Microsoft.PowerShell.Management, Microsoft.PowerShell.Utility, Microsoft.PowerShell.Diagnostics,
↳Microsoft.PowerShell.Core, Microsoft.PowerShell.Host, Microsoft.PowerShell.Security,
↳Microsoft.WSMan.Management' : This command did not update help topics for the Windows
↳PowerShell core commands or for any modules in the $pshome\Modules directory. To update
↳these help topics, start Windows PowerShell with the "Run as Administrator" option and try
↳the command again.
At line:1 char:1
+ update-help
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [Update-Help], Exception
+ FullyQualifiedErrorId : UpdatableHelpSystemRequiresElevation, Microsoft.PowerShell.
↳Commands.UpdateHelpCommand
```

Najważniejszą część komunikatu o błędzie wyróżniliśmy pogrubioną czcionką — ten fragment informuje o tym, gdzie leży problem i jak go rozwiązać. Teraz uruchom powłokę PowerShell na prawach administratora systemu i ponownie wykonaj polecenie *Update-Help* — cały proces aktualizacji systemu pomocy powinien zająć tylko kilka minut.

Bardzo istotną sprawą jest to, abyś wyrobił w sobie nawyk aktualizowania pomocy raz na miesiąc albo przynajmniej raz na kwartał. W systemie pomocy PowerShell mogą się również znajdować opisy innych poleceń, utworzonych przez inne firmy niż Microsoft, pod warunkiem oczywiście, że moduły poleceń znajdują się w odpowiedniej lokalizacji oraz że zostały przygotowane do korzystania z systemu pomocy powłoki PowerShell.

Czy masz komputery, które nie są podłączone do sieci Internet? Nie ma problemu: przejdź do komputera podłączonego do sieci i za pomocą polecenia `Save-Help` utwórz lokalną kopię systemu pomocy. Następnie umieść nowo utworzoną kopię na swoim serwerze plików lub w innym miejscu dostępnym dla reszty sieci i uruchom polecenie `Update-Help` z parametrem `-Source`, wskazującym na lokalizację pobranej kopii systemu pomocy. W taki sposób komputery w Twojej sieci będą pobierały zaktualizowaną zawartość systemu pomocy z tego centralnego repozytorium, a nie z sieci Internet.

System pomocy typu *open source*

Pliki systemu pomocy powłoki PowerShell firmy Microsoft są materiałami typu *open source*, które są dostępne na stronie <http://github.com/powershell>. Od czasu do czasu warto zaglądać na tę stronę; można tam znaleźć najnowsze informacje, które często jeszcze nie zostały skompilowane do oficjalnych plików pomocy pobieranych i wyświetlanych przez powłokę PowerShell.

3.3. Korzystanie z systemu pomocy

Powłoka PowerShell udostępnia polecenie o nazwie `Get-Help`, które pozwala na wyszukiwanie tematów pomocy. W praktyce możesz znaleźć wiele przykładów (zwłaszcza w sieci Internet), w których użytkownicy do uzyskania pomocy używają polecenia `Help` czy nawet wywodzącego się z systemu UNIX polecenia `Man` (ang. *Manual*; podręcznik). Polecenia `Man` i `Help` nie są macierzystymi cmdletami powłoki PowerShell; są to funkcje opakowujące (ang. *wrappers*) głównego cmdletu `Get-Help`.

Pomoc w systemach macOS/Linux

W systemach macOS i Linux tematy pomocy są wyświetlane przy użyciu tradycyjnego polecenia `man`, które zazwyczaj „przejmuje” ekran na czas wyświetlania pomocy i powraca do powłoki PowerShell, kiedy zakończysz czytanie.

Polecenie `Help` działa podobnie do bazowego cmdletu `Get-Help`, z tym że tekst pomocy kierowany jest od razu do polecenia `More`, dzięki czemu tekst wyświetlany jest strona po stronie w przyjazny dla użytkownika sposób, zdecydowanie ułatwiający czytanie. Przykładowo, wykonanie poleceń `Help Get-Content` i `Get-Help Get-Content` daje takie same rezultaty, ale w przypadku tego pierwszego polecenia tekst jest wyświetlany strona po stronie. Oczywiście zawsze możesz uruchomić polecenie `Get-Help Get-Content | More` i uzyskać w ten sposób stronicowanie wyświetlanego tekstu, ale jak widać, wymaga to nieco więcej pisania. W praktyce najczęściej korzystamy z polecenia `Help`, ale chcielibyśmy po prostu zwrócić uwagę na to, że w powłoce PowerShell każda operacja może być wykonywana na wiele sposobów.

UWAGA Z technicznego punktu widzenia polecenie `Help` jest funkcją, a polecenie `Man` jest aliasem dla funkcji `Help`, ale niezależnie od tego, którego z nich użyjesz, uzyskasz takie same wyniki. O aliasach będziemy mówić w kolejnym rozdziale.

Nawiasem mówiąc, czasami wyświetlanie treści pomocy strona po stronie może być nieco irytujące, zwłaszcza kiedy znalazłeś już szukane informacje, a do wyświetlenia pozostało jeszcze kilka kolejnych stron tekstu. W takiej sytuacji możesz po prostu nacisnąć kombinację klawiszy *Ctrl+C*, co spowoduje przerwanie działania polecenia wyświetlającego tekst pomocy i powrót wiersza poleceń powłoki. Pamiętaj, że kiedy pracujesz w oknie konsoli tekstowej powłoki PowerShell, kombinacja klawiszy *Ctrl+C* zawsze powoduje *przerwanie* działania danego polecenia, a nie *kopiowanie do schowka*. Jeżeli jednak używasz graficznie zorientowanego środowiska Windows PowerShell ISE, naciśnięcie kombinacji klawiszy *Ctrl+C* spowoduje skopiowanie danych do schowka. Aby w środowisku ISE zatrzymać działanie polecenia, powinieneś nacisnąć czerwony przycisk *Stop Operation* (zatrzymaj operację), znajdujący się na pasku narzędzi.

UWAGA Polecenie *More* nie działa w środowisku ISE, stąd nawet jeżeli użyjesz polecenia *Help* lub *Man*, tekst pomocy zostanie od razu wyświetlony w całości, bez podziału na strony.

System pomocy powłoki PowerShell ma dwa główne zadania: wspomaganie użytkownika w wyszukiwaniu poleceń pozwalających na wykonywanie określonych zadań oraz pomoc w nauce korzystania z tych poleceń.

3.4. Zastosowanie systemu pomocy do wyszukiwania poleceń

Technicznie rzecz biorąc, system pomocy nie ma pojęcia, jakie polecenia są dostępne w powłoce PowerShell. Jego „wiedza” sprowadza się do tego, jakie tematy pomocy są dostępne. Jeżeli jakieś polecenie nie będzie miało swojego pliku pomocy (a tak może się zdarzyć), to w takim przypadku system pomocy nie będzie wiedział, że takie polecenie istnieje. Na szczęście firma Microsoft stara się dostarczać pliki pomocy dla prawie każdego cmdletu. Co więcej, system pomocy może korzystać z informacji, które nie są bezpośrednio związane z określonym poleceniem *cmdlet*, takich jak informacje kontekstowe czy inne informacje ogólne.

Podobnie jak większość poleceń, *Get-Help* (a zatem również i *Help*) ma kilka parametrów. Jednym z nich, być może tym najważniejszym, jest *-Name*. Ten parametr określa nazwę tematu pomocy, do którego chcesz uzyskać dostęp. Jest to parametr pozycyjny, stąd w wierszu poleceń nie musisz wpisywać słowa kluczowego *-Name*; zamiast tego wystarczy po prostu podać nazwę, której szukasz. W nazwach można używać symboli wieloznacznych, dzięki czemu system pomocy znakomicie radzi sobie z wyszukiwaniem poleceń.

Załóżmy na przykład, że chcesz wykonać jakieś operacje związane z dziennikami zdarzeń, ale nie wiesz, jakie polecenia są dostępne, i stąd decydujesz się poszukać w systemie pomocy tematów związanych z dziennikami zdarzeń. Możesz to zrobić, uruchamiając jedno z poleceń przedstawionych poniżej:

```
Help *log*
Help *event*
```

Przykładowe wyniki działania pierwszego z tych poleceń zostały przedstawione poniżej:

Name	Category	Module
----	-----	-----
Clear-EventLog	Cmdlet	Microsoft.PowerShell.M...
Get-EventLog	Cmdlet	Microsoft.PowerShell.M...
Limit-EventLog	Cmdlet	Microsoft.PowerShell.M...
New-EventLog	Cmdlet	Microsoft.PowerShell.M...
Remove-EventLog	Cmdlet	Microsoft.PowerShell.M...
Show-EventLog	Cmdlet	Microsoft.PowerShell.M...
Write-EventLog	Cmdlet	Microsoft.PowerShell.M...
Get-AppxLog	Function	Appx
Get-DtcLog	Function	MsDtc
Reset-DtcLog	Function	MsDtc
Set-DtcLog	Function	MsDtc
Get-LogProperties	Function	PSDiagnostics
Set-LogProperties	Function	PSDiagnostics
about_Eventlogs	HelpFile	
about_Logical_Operators	HelpFile	

UWAGA Z pewnością zauważyłeś, że na przedstawionej wyżej liście znajdują się polecenia (i funkcje) z modułów takich jak Appx i MsDtc. System pomocy wyświetla o nich informacje, mimo że jeszcze nie załadowałeś tych rozszerzeń do pamięci. Takie rozwiązanie pomaga w wyszukiwaniu poleceń, które inaczej mogłyby zostać łatwo przeoczone. System pomocy powłoki PowerShell pozwala na wyszukiwanie informacji o poleceniach z dowolnych rozszerzeń zainstalowanych we właściwej lokalizacji; więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 7.

Wiele poleceń i funkcji z przedstawionej listy wydaje się mieć coś wspólnego z dziennikami zdarzeń, a na podstawie ich nazw możemy się domyślić, że wszystkie — z wyjątkiem dwóch ostatnich — są tematami pomocy związanymi z określonymi poleceniami *cmdlet*. Dwa ostatnie tematy pomocy obejmują informacje ogólne, z czego ten ostatni nie wydaje się mieć nic wspólnego z dziennikami zdarzeń — pojawił się na liście, ponieważ zawiera ciąg znaków *log* — (fragment słowa *logical*). Pamiętaj, że jeżeli to możliwe, powinieneś zawsze starać się używać jak najbardziej ogólnego wzorca wyszukiwania (na przykład **event** czy **log** zamiast **event-log**), ponieważ dzięki temu otrzymasz najwięcej wyników.

Kiedy znajdziesz *cmdlet*, który Twoim zdaniem nadaje się do wykonania danego zadania (w naszym przypadku dobrym kandydatem wydaje się polecenie *Get-EventLog*), możesz wyświetlić jego bardziej szczegółowy opis, wykonując następujące polecenie:

```
Help Get-EventLog
```

Nie zapomnij o mechanizmie dopełniania poleceń przy użyciu klawisza *Tab*! Przypominamy, że wystarczy wpisać część nazwy polecenia i nacisnąć klawisz *Tab*, aby powłoka dopełniła nazwę najbardziej pasującym ciągiem znaków. Jeżeli pasujących dopełnień jest więcej, możesz je wyświetlać kolejno, naciskając klawisz *Tab* aż do momentu znalezienia odpowiedniego polecenia.

ZRÓB TO SAM W wierszu poleceń konsoli powłoki PowerShell wpisz ciąg znaków `Help Get-Ev` i naciśnij klawisz `Tab`. Pierwsze wyświetlone dopasowanie to `Get-Event`, które nie jest tym, czego potrzebujesz. Dopiero ponowne naciśnięcie klawisza `Tab` wyświetla poszukiwaną funkcję `Get-EventLog`. Aby zaakceptować wybrane polecenie i wyświetlić dla niego tekst pomocy, powinieneś nacisnąć klawisz `Enter`. Jeżeli pracujesz w środowisku ISE, nie musisz nawet naciskać klawisza `Tab` — po wpisaniu fragmentu nazwy lista pasujących poleceń pojawia w postaci menu podręcznego, z którego możesz wybrać pasującą nazwę i zakończyć pisanie naciśnięciem klawisza `Enter`.

Korzystając z polecenia `Help`, możesz używać symboli wieloznacznych; dotyczy to szczególnie symbolu gwiazdki (*), która zastępuje dowolny ciąg znaków. Jeżeli system pomocy znajdzie tylko jeden temat pomocy pasujący do podanego wzorca, to nie będzie wyświetlał listy znalezionych poleceń, a zamiast tego od razu wyświetli zawartość tego elementu.

ZRÓB TO SAM Uruchom polecenie `Help Get-EventL*` — zamiast listy pasujących tematów na ekranie powinieneś od razu zobaczyć zawartość pliku pomocy dla polecenia `Get-EventLog`.

Jeżeli wykonałeś polecenie przedstawione powyżej, na ekranie powinieneś mieć teraz wyświetloną zawartość pliku pomocy dla polecenia `Get-EventLog`. Ten plik zawiera krótki opis polecenia i jego składni. Takie informacje są bardzo przydatne, gdy chcesz sobie szybko odświeżyć pamięć o tym, jak używać danego polecenia.

Dla zainteresowanych

Czasami chcemy podzielić się z Tobą informacjami, które chociaż bardzo przydatne, to jednak nie są niezbędne do zrozumienia sposobu działania powłoki. Z tego względu będziemy umieszczać takie informacje w sekcjach „Dla zainteresowanych”, takich jak ta. Jeżeli je pominiesz, nic się nie stanie, jeżeli jednak je przeczytasz, poznasz alternatywne sposoby wykonywania różnych zadań, dowiesz się, do czego jeszcze można wykorzystać różne polecenia, i poznasz wiele innych tajemnic powłoki PowerShell.

Wspominaliśmy już, że polecenie `Help` nie szuka samych poleceń *cmdlet* — zamiast tego poszukuje odpowiednich tematów pomocy. Ponieważ niemal każdy *cmdlet* posiada swój plik pomocy, możemy jednak powiedzieć, że wyszukiwanie tematów pomocy daje takie same wyniki jak wyszukiwanie poleceń. Powinieneś jednak pamiętać, że możesz również bezpośrednio szukać *cmdletów* za pomocą polecenia `Get-Command` (lub jego aliasu `Gcm`).

Podobnie jak polecenie `Help`, `Get-Command` akceptuje symbole wieloznaczne, dzięki czemu możesz na przykład użyć polecenia `Gcm *event*` do wyświetlenia wszystkich poleceń zawierających w nazwie ciąg znaków *event*. Warto jednak zauważyć, że wyniki działania takiego polecenia będą zawierać nie tylko polecenia *cmdlet*, ale również polecenia zewnętrzne, które nie zawsze muszą być przydatne w danej sytuacji.

Znacznie lepszym rozwiązaniem może być użycie parametrów `-Noun` („rzeczownik”) lub `-Verb` („czasownik”). Ponieważ tylko nazwy poleceń *cmdlet* składają się z rzeczowników i czasowników, wyniki wyszukiwania będą ograniczone do *cmdletów*. Przykładowo, wykonanie

połączenia `Get-Command -noun *event*` zwraca listę cmdletów dotyczących zdarzeń, a polecenie `Get-Command -verb Get` zwraca wszystkie cmdlety umożliwiające pobieranie różnych elementów. Można również użyć parametru `-CommandType`, określając typ poszukiwanego polecenia *cmdlet*. Przykładowo, polecenie `Get-Command *log* -CommandType cmdlet` wyświetla listę wszystkich cmdletów, które w nazwie zawierają ciąg znaków *log*, a dzięki określeniu typu lista nie zawiera żadnych aplikacji lub poleceń zewnętrznych.

3.5. Interpretacja treści plików pomocy

Pliki pomocy cmdletów powłoki PowerShell zostały przygotowane z zachowaniem określonych reguł. Poznanie tych konwencji jest kluczem do pełnego zrozumienia zawartości plików pomocy, wydobycia z nich maksymalnej ilości informacji i korzystania z cmdletów w najbardziej efektywny sposób.

3.5.1. Zestawy parametrów i parametry wspólne

Większość poleceń może działać na różne sposoby, w zależności od tego, co chcemy uzyskać. Dla przykładu zamieszczamy poniżej fragment pliku pomocy polecenia `Get-EventLog`, opisujący jego składnię:

SYNTAX

```
Get-EventLog [-AsString] [-ComputerName <string[]>] [-List] [<CommonParameters>]
Get-EventLog [-LogName] <string> [[-InstanceId] <Int64[]>] [-After <DateTime>]
➤ [-AsBaseObject] [-Before <DateTime>] [-ComputerName <string[]>] [-EntryType <string[]>]
➤ [-Index <Int32[]>] [-Message <string>] [-Newest <int>] [-Source <string[]>] [-UserName
➤ <string[]>] [<CommonParameters>]
```

Zauważ, że w przedstawionym przykładzie składni polecenie `GetEventLog` jest wymienione dwa razy, co oznacza, że obsługuje ono dwa *zestawy parametrów* i możesz użyć tego polecenia na dwa różne sposoby. Niektóre parametry wywołania są wspólne dla obu zestawów. Przykładem takiego parametru może być `-ComputerName`. Zawsze jednak w każdym z zestawów będzie znajdował się co najmniej jeden unikatowy parametr, który istnieje tylko w tym zestawie. W naszym przykładzie w pierwszym zestawie mamy parametry `-AsString` i `-List`, których nie ma w drugim zestawie; z kolei drugi zestaw zawiera wiele parametrów, których próżno by szukać w pierwszym.

Jak to działa? Jeżeli użyjesz parametru, który znajduje się tylko w jednym zestawie, zostajesz niejako „zablokowany” w tym zestawie i możesz używać tylko takich parametrów, które pojawiają się w tym właśnie zestawie. Przykładowo, jeżeli wybierzesz opcję `-List`, jedynymi innymi parametrami, których możesz użyć, będą `-AsString` i `-ComputerName`, ponieważ są to jedyne dwa pozostałe parametry zawarte w tym zestawie. Nie możesz dodać na przykład opcji `-LogName`, ponieważ nie występuje ona w pierwszym zestawie parametrów. Oznacza to, że opcje `-List` i `-LogName` *wzajemnie się wykluczają* — nie możesz użyć ich obu w tym samym poleceniu, ponieważ znajdują się w różnych zestawach parametrów.

Czasami można uruchomić polecenie tylko z opcjami, które są wspólne dla wielu zestawów. W takich przypadkach powłoka zwykle wybiera domyślnie pierwszy zestaw.

Ponieważ każdy zestaw opcji implikuje inne zachowanie polecenia, ważne jest, aby zrozumieć, który z nich zostanie w takiej sytuacji wybrany.

Zwróć uwagę, że wszystkie zestawy parametrów dla cmdletów powłoki PowerShell kończą się opcją [`<CommonParameters>`]. Odnosi się ona do zestawu ośmiu parametrów dostępnych w każdym cmdlecie, bez względu na to, jak go używasz. Nie będziemy teraz o nich mówić, zrobimy to w stosunku do niektórych z nich w dalszej części książki, kiedy będziemy ich używać do wykonania określonych zadań. Oprócz tego w dalszej części tego rozdziału powiemy, gdzie możesz dowiedzieć się czegoś więcej o tych parametrach.

UWAGA Uważni czytelnicy będą już w stanie rozpoznać różnice widoczne w niektórych naszych listingach. Przykładowo, niektórzy czytelnicy z pewnością zauważyli nieco inną zawartość pliku pomocy dla polecenia `Get-EventLog`, która zmienia się w zależności od używanej wersji powłoki PowerShell. W niektórych wersjach możesz nawet znaleźć kilka nowych parametrów, ale mimo to podstawowe, fundamentalne zagadnienia i pojęcia, które objaśniamy, nie uległy zmianie. Krótko mówiąc, nie powinieneś przejmować się tym, że to, co widzisz na swoim ekranie, może nieco różnić się od tego, co pokazujemy w naszej książce.

3.5.2. Parametry obligatoryjne i opcjonalne

Aby uruchomić wybrane polecenie *cmdlet*, nie musisz oczywiście używać wszystkich dostępnych parametrów. W systemie pomocy powłoki PowerShell parametry opcjonalne wyświetlane są w nawiasach kwadratowych. Na przykład zapis [`-ComputerName <string[]>`] wskazuje, że cały parametr `-ComputerName` jest opcjonalny. W praktyce w ogóle nie musisz go używać; jeżeli ten parametr zostanie pominięty, to polecenie *cmdlet* domyślnie przyjmie, że dotyczy komputera lokalnego. Z tego względu w opisie składni opcja [`<CommonParameters>`] została umieszczona w nawiasach kwadratowych — takie polecenia możesz uruchomić bez użycia żadnego ze wspólnych parametrów.

Prawie każde polecenie *cmdlet* posiada co najmniej jeden parametr opcjonalny. Niektórych z nich będziesz zapewne używał niemal codziennie, a innych być może nie użyjesz nigdy. Pamiętaj, że stosując dany parametr, wystarczy w wierszu poleceń wpisać taki fragment jego nazwy, który w jednoznaczny sposób pozwoli powłocie na jego identyfikację. Przykładowo, zapis `-L` nie będzie w wystarczający sposób identyfikował parametru `-List`, ponieważ może również oznaczać `-LogName`, ale już zapis `-Li` będzie jednoznacznym skrótem dla parametru `-List`, ponieważ nazwa żadnego innego parametru nie rozpoczyna się od ciągu znaków `-Li`.

A co się stanie, jeżeli spróbujesz uruchomić polecenie i zapomnisz umieścić w wierszu wywołania jednego z parametrów obligatoryjnych? Rzuć okiem na zawartość pliku pomocy polecenia `Get-EventLog`, a przekonasz się, że parametr `-LogName` jest obligatoryjny (parametr nie jest ujęty w nawiasy kwadratowe). Spróbujmy zatem uruchomić polecenie `Get-EventLog` bez określania nazwy dziennika zdarzeń.

ZRÓB TO SAM Przekonaj się, co się stanie, kiedy uruchomisz polecenie `Get-EventLog` bez żadnych parametrów.

W takiej sytuacji powłoka PowerShell powinna wyświetlić monit o podanie brakującego obligatoryjnego parametru `LogName`. Jeśli wpiszesz nazwę wybranego dziennika zdarzeń, takiego jak `System` lub `Application`, i naciśniesz klawisz `Enter`, polecenie będzie działać poprawnie. Jeżeli chcesz przerwać działanie polecenia, powinieneś nacisnąć kombinację klawiszy `Ctrl+C`.

3.5.3. Parametry pozycyjne

Projektanci powłoki PowerShell doskonale zdawali sobie sprawę z tego, że niektóre parametry będą używane tak często, iż wpisywanie za każdym razem ich nazw w wierszu poleceń będzie dla użytkownika po prostu irytujące. Takie powszechnie używane parametry są często *pozycyjne*, co oznacza, że możesz podawać ich wartości bez konieczności wpisywania nazwy parametru, pod warunkiem jednak, że umieścisz tę wartość na właściwej pozycji w wierszu wywołania polecenia.

Parametry pozycyjne możesz zidentyfikować na dwa sposoby: w opisie składni polecenia lub w pełnym opisie polecenia w jego pliku pomocy.

WYSZUKIWANIE PARAMETRÓW POZYCYJNYCH W OPISIE SKŁADNI POLECENIA

Parametrów pozycyjnych możesz poszukać w podsumowaniu składni polecenia, gdzie nazwa parametru (ale tylko nazwa) jest ujęta w nawiasy kwadratowe. Na przykład spójrz na pierwsze dwie opcje w drugim zestawie parametrów polecenia `Get-EventLog`:

```
[ -LogName ] <string> [[ -InstanceId ] <Int64[> ]]
```

Pierwszy parametr, `-LogName`, nie jest opcjonalny. Możemy to wywnioskować z tego, że cały parametr (czyli jego nazwa oraz wartość) nie jest otoczony nawiasami kwadratowymi. Widać jednak, że sama nazwa parametru jest ujęta w nawiasy kwadratowe, co oznacza, że jest to parametr pozycyjny — w wierszu polecenia możesz podać nazwę dziennika bez konieczności wpisywania samej nazwy opcji `-LogName`. Ponieważ w opisie składni parametr ten pojawia się na pierwszej pozycji po nazwie polecenia, wiemy, że nazwa dziennika zdarzeń jest pierwszym parametrem wywołania, który musimy podać.

Drugi parametr, `-InstanceId`, jest opcjonalny, ponieważ zarówno jego nazwa, jak i wartość są ujęte w nawiasy kwadratowe. Nazwa parametru `-InstanceId` sama w sobie jest również ujęta w nawiasy kwadratowe, co wskazuje, że jest to także parametr pozycyjny. W opisie składni pojawia się on na drugiej pozycji, stąd jeżeli zdecydujesz się na pominięcie jego nazwy, w wierszu wywołania polecenia musisz podać jego wartość na drugiej pozycji, zaraz po nazwie dziennika zdarzeń.

Parametr `-Before` (który w opisie składni pojawia się nieco dalej; możesz się o tym przekonać, wykonując polecenie `Help Get-EventLog`) jest opcjonalny, ponieważ jest w całości zamknięty w nawiasach kwadratowych. Nazwa parametru nie została osobno ujęta w nawiasy kwadratowe, tak więc, jeśli zdecydujesz się użyć tego parametru, musisz wpisać jego nazwę (lub przynajmniej jej jednoznacznie identyfikujący fragment).

Używając parametrów pozycyjnych, powinieneś pamiętać o kilku wskazówkach:

- Parametrów pozycyjnych można używać łącznie z parametrami wymagającymi użycia nazw, z tym że parametry pozycyjne muszą zawsze znajdować się we właściwych pozycjach wiersza wywołania polecenia. Na przykład składnia polecenia `Get-EventLog System -Newest 20` jest najzupełniej prawidłowa: nazwa `System` zostanie przekazana do polecenia jako wartość parametru `-LogName`, ponieważ znajduje się na pierwszej pozycji po nazwie polecenia, natomiast liczba `20` zostanie przekazana jako wartość parametru `-Newest`, ponieważ w tym przypadku została użyta nazwa parametru.
- Podawanie nazwy dla każdego z używanych parametrów (łącznie z nazwami parametrów pozycyjnych) jest zawsze dozwolone. Jeżeli zdecydujesz się na takie rozwiązanie, to kolejność podawania parametrów wywołania polecenia nie będzie miała żadnego znaczenia. Przykładowo, wywołanie polecenia `Get-EventLog -Newest 20 -Log Application` jest całkowicie poprawne, ponieważ użyliśmy nazw parametrów (a w przypadku opcji `-LogName` jej nazwa została skrócona do `-Log`, co w zupełności wystarczy do jej jednoznacznej identyfikacji).
- Jeśli używasz wielu parametrów pozycyjnych, pamiętaj o zachowaniu ich odpowiedniej kolejności. Przykładowo, polecenie `Get-EventLog Application 0` będzie działać poprawnie, ponieważ nazwa `Application` zostanie przekazana do polecenia jako nazwa parametru `-LogName`, a `0` — jako wartość parametru `-InstanceId`. Jednak już próba wykonania polecenia `Get-EventLog 0 Application` zakończy się niepowodzeniem, ponieważ wartość `0` zostanie przekazana do polecenia jako wartość parametru `-LogName`, a przecież żaden dziennik zdarzeń nie nosi nazwy `0`.

Aby uniknąć niepotrzebnych błędów i związanej z nimi frustracji, powinieneś po prostu zawsze używać nazw parametrów dopóty, dopóki nie poczujesz się w pełni komfortowo z ich składnią i nie dojdiesz do wniosku, że to ciągle wpisywanie nazw parametrów po prostu zaczęło Cię męczyć. Następnie zacznij w składni uruchamianych poleceń używać parametrów pozycyjnych, co oszczędzi Ci pisanie. Kiedy zaczniesz zapisywać bardziej złożone polecenia w plikach tekstowych (co ułatwi później ich używanie), zawsze stosuj pełne nazw cmdletów i podawaj pełne nazwy parametrów wywołania, bez używania parametrów pozycyjnych i bez skróconych nazw pozostałych parametrów. Dzięki temu zawartość pliku będzie bardziej czytelna i łatwiejsza do przeanalizowania w przyszłości, a samo zapisanie złożonego polecenia w pliku zaoszczędzi Ci również niepotrzebnego klikania.

WYSZUKIWANIE PARAMETRÓW POZYCYJNYCH W PEŁNYM OPISIE POLECENIA

Jak już wspominaliśmy wcześniej, parametry pozycyjne danego polecenia możemy znaleźć na dwa sposoby. Pierwszy w nich przedstawiliśmy przed chwilą, a drugi wymaga wyświetlenia pełnej zawartości pliku pomocy, co możemy zrobić, wywołując polecenie `Help` z opcją `-full`.

ZRÓB TO SAM Wykonaj polecenie `Help Get-EventLog -full`. Pamiętaj, że zawartość pliku pomocy jest wyświetlana po jednym ekranie na raz, a do kolejnych stron możesz przechodzić, naciskając klawisz spacji. Jeżeli chcesz przerwać wyświetlanie pomocy, możesz w dowolnym momencie nacisnąć kombinację klawiszy `Ctrl+C`. Na razie jednak przejrzyj całą zawartość pliku pomocy. Następnie zamiast parametru `-full` spróbuj użyć parametru `-ShowWindow`, który powinien działać na dowolnym komputerze lub serwerze wyposażonym w graficzny interfejs użytkownika. Warto zauważyć, że powodzenie w korzystaniu z opcji `-ShowWindow` zależy od zawartości bazowego pliku pomocy XML. Jeśli zawartość tego pliku nie jest zapisana w odpowiedni sposób, niektóre fragmenty pomocy mogą nie być wyświetlane. Pamiętaj, że opcja `-ShowWindow` nie działa w systemach operacyjnych innych niż Windows.

Po wyświetleniu pełnego opisu polecenia przewijaj kolejne strony, aż zobaczysz opis parametru `-LogName`, który powinien wyglądać mniej więcej tak:

```
-LogName <string>
  Specifies the event log. Enter the log name (the value of the Log property; not the
  ↳LogDisplayName) of one event log. Wildcard characters are not permitted. This parameter
  ↳is required.
  Required?                true
  Position?                1
  Default value
  Accept pipeline input?   false
  Accept wildcard characters? False
```

W opisie możemy łatwo znaleźć informację, że jest to parametr obligatoryjny (właściwość `Required` ma wartość `true`) i pozycyjny, który w wierszu wywołania polecenia występuje na pierwszej pozycji zaraz po nazwie cmdletu.

Zawsze zachęcamy naszych studentów, aby rozpoczynając pracę z nowym poleceniem *cmdlet*, skupili się na przeczytaniu pełnego pliku pomocy, a nie tylko na skróconym opisie składni. Z oczywistych powodów w pełnym opisie polecenia znajduje się znacznie więcej szczegółów. Przykładowo, z pełnego opisu parametru `-LogName` możesz się dowiedzieć, że parametr ten pozwala na używanie symboli wieloznacznych, tak więc w wierszu wywołania nie możesz podać wartości takiej jak `App*`, a zamiast tego musisz wpisać pełną nazwę dziennika zdarzeń, na przykład `Application`.

3.5.4. Wartości parametrów

Pliki pomocy zawierają również wskazówki dotyczące rodzaju danych wejściowych akceptowanych przez poszczególne parametry. Niektóre parametry, zwane **przełącznikami**, w ogóle nie wymagają podawania żadnych wartości wejściowych. W skróconym opisie składni taki parametr może wyglądać na przykład tak:

```
[-AsString]
```

W pełnym pliku pomocy znajdziemy znacznie bardziej rozbudowany opis tego parametru:

```
-AsString [<SwitchParameter>]
  Returns the output as strings, instead of objects.
  Required?                false
  Position?                named
  Default value
  Accept pipeline input?    false
  Accept wildcard characters? False
```

Zapis [<SwitchParameter>] potwierdza, że jest to przełącznik i że nie wymaga podawania żadnej wartości wejściowej. Przełączniki nigdy nie mają charakteru parametrów pozycyjnych, stąd zawsze musisz podawać ich nazwy (lub przynajmniej ich skrócone wersje). Przełączniki są zawsze opcjonalne, dzięki czemu w zależności od potrzeb możesz ich używać bądź nie.

Inne parametry wymagają podawania takich czy innych wartości wejściowych, które zawsze umieszczamy po nazwie danego parametru i oddzielamy spacją (a nie znakiem dwukropka, znakiem równości lub innym znakiem, chociaż od czasu do czasu możesz spotkać wyjątki od tej reguły). W skróconym opisie składni polecenia typ oczekiwanego wejścia jest definiowany w nawiasach ostrych, tak jak to zostało pokazane poniżej:

```
[-LogName] <string>
```

W pełnym opisie polecenia wygląda to tak:

```
-Message <string>
  Gets events that have the specified string in their messages. You can use this property
  ↳ to search for messages that contain certain words or phrases. Wildcards are permitted.
  Required?                false
  Position?                named
  Default value
  Accept pipeline input?    false
  Accept wildcard characters? True
```

Przyjrzyjmy się teraz niektórym typowym rodzajom danych wejściowych:

- String — ciągi znaków alfanumerycznych. Czasami mogą zawierać spację, ale w takich sytuacjach cały ciąg musi zostać ujęty w znaki apostrofu. Na przykład ścieżka *C:\Windows* nie musi być podawana w apostrofach, ale już *C:\Program Files* tak, ponieważ w środku tego ciągu znaków znajduje się spacja. W obecnej wersji powłoki można zamiennie używać apostrofów lub znaków cudzysłowu, ale najlepiej pozostać przy stosowaniu apostrofów.
- Int, Int32 *lub* Int64 — liczby całkowite o różnych rozmiarach (bez części dziesiętnej).
- DateTime — ogólnie rzecz biorąc, jest to ciąg znaków, który na podstawie ustawień regionalnych komputera można interpretować jako datę. Dla Polski będzie to na przykład 2010-10-10 (odpowiednio: rok, miesiąc, dzień).

Inne, bardziej wyspecjalizowane typy parametrów będziemy omawiać na bieżąco w miarę potrzeb.

Z pewnością zauważyłeś, że w zapisie niektórych typów wartości występują pary nawiasów kwadratowych:

```
[-ComputerName <string[]>]
```

Użycie takiej dodatkowej pary nawiasów kwadratowych bezpośrednio po typie wartości parametru wcale nie oznacza, że jest on opcjonalny. Zamiast tego zapis `string[]` wskazuje, że wartościami danego parametru mogą być *tablice*, *kolekcje* bądź *listy* elementów typu `String`. W omawianych sytuacjach oczywiście zawsze możemy dla takiego parametru podać pojedynczą wartość:

```
Get-EventLog Security -computer Server-R2
```

Oczywiście w przypadku takich parametrów dopuszczalne jest również podawanie wielu wartości. Prostim sposobem na to jest podanie listy wartości oddzielonych od siebie przecinkami. Powłoka PowerShell traktuje wszystkie listy wartości oddzielonych od siebie przecinkami jako tablice wartości:

```
Get-EventLog Security -computer Server-R2, DC4, Files02
```

Jak już wspominaliśmy wcześniej, każdy ciąg znaków zawierający spację musi być ujęty w znaki apostrofu. Pamiętaj jednak, że nie chodzi tutaj o całą listę, ale tylko o pojedyncze wartości. Poniżej pokazujemy przykład takiej poprawnie zapisanej listy:

```
Get-EventLog Security -Computer 'Server-R2', 'Files02'
```

Choć żadna z użytych wartości nie musi być w apostrofach, to jednak taki zapis jest całkowicie poprawny. W kolejnym przykładzie prezentujemy niepoprawny zapis listy wartości:

```
Get-EventLog Security -Computer 'Server-R2, Files01'
```

W takim przypadku nasz cmdlet będzie poszukiwał jednego komputera o nazwie *Server-R2, Files01*, co najprawdopodobniej nie będzie tym, czego oczekiwałeś.

Innym sposobem podania listy wartości jest umieszczenie ich w pliku tekstowym, po jednym elemencie w każdym wierszu. Oto przykład zawartości takiego pliku:

```
Server-R2  
Files02  
Files03  
DC04  
DC03
```

Następnie można użyć polecenia `Get-Content`, aby odczytać zawartość tego pliku i przekazać ją jako wartość parametru `-computerName`. Możemy to zrobić, zmuszając powłokę PowerShell do wykonania polecenia `Get-Content` w pierwszej kolejności, dzięki czemu odczytane dane będą mogły zostać przekazane dalej jako wartość parametru `-computerName`.

Pamiętasz z matematyki, w jaki sposób nawiasy `()` mogą być używane do określania porządku wykonywania operacji w działaniach matematycznych? Podobnie działa to w powłoce PowerShell: umieszczając polecenie w nawiasach, wymuszasz jego wykonanie w pierwszej kolejności:

```
Get-EventLog Application -computer (Get-Content names.txt)
```

Powyższy przykład demonstruje bardzo przydatną sztuczkę. Załóżmy, że mamy pliki tekstowe z nazwami komputerów różnych klas — serwerów sieciowych, kontrolerów domen, serwerów baz danych i tak dalej. Dzięki zastosowaniu tego prostego rozwiązania możemy szybko uruchomić potrzebne polecenia na wszystkich komputerach należących do wybranej klasy.

Istnieje jeszcze kilka innych sposobów przekazywania list wartości do parametrów wywołania polecenia, włącznie z odczytaniem nazw komputerów z Active Directory. Takie techniki są jednak nieco bardziej skomplikowane, więc powrócimy do nich w późniejszych rozdziałach, już po tym, jak nauczysz się korzystać z innych, przydatnych poleceń.

Innym sposobem przekazywania wielu wartości dla wybranego parametru (pod warunkiem że jest to parametr obligatoryjny) jest umieszczenie tego parametru w wierszu wywołania bez podawania wartości. W takiej sytuacji powłoka PowerShell po prostu poprosi sama o podanie wartości tego parametru. W przypadku parametrów akceptujących wiele wartości możesz wpisać pierwszą wartość i nacisnąć klawisz *Enter*. PowerShell poprosi o podanie drugiej wartości, którą możesz wpisać i ponownie nacisnąć klawisz *Enter*. Powtarzaj taką operację aż do podania wszystkich wartości i po pojawieniu się kolejnego monitu naciśnij po prostu klawisz *Enter*, aby powiadomić powłokę PowerShell, że już skończyłeś. Oczywiście jeżeli chcesz przerwać działanie polecenia, możesz w dowolnym momencie nacisnąć kombinację klawiszy *Ctrl+C*.

3.5.5. Wyszukiwanie przykładów poleceń

Wszyscy wiemy, że najlepiej uczyć się na przykładach, dlatego pisząc tę książkę, staraliśmy się umieścić w niej jak najwięcej przykładów. Projektanci powłoki PowerShell również doskonale zdają sobie sprawę z tego, że większość administratorów lubi korzystać z przykładów, więc w plikach pomocy przygotowali ich bardzo wiele. Jeżeli przewinąłeś do końca zawartość pliku pomocy dla polecenia `Get-EventLog`, najprawdopodobniej znalazłeś tam prawie tuzin przykładów użycia tego cmdletu.

Istnieje jednak znacznie łatwiejszy sposób wyświetlenia przykładów zastosowania danego polecenia (przy założeniu oczywiście, że to wszystko, co chcesz zobaczyć). Aby to zrobić, zamiast parametru `-full` polecenia `Help` powinieneś użyć parametru `-example`, tak jak to zostało pokazane poniżej:

```
Help Get-EventLog -example
```

ZRÓB TO SAM Spróbuj samodzielnie wyświetlić przykłady użycie tego i innych poleceń powłoki PowerShell, używając tego nowego parametru polecenia `Help`.

Przykłady użycia poleceń powłoki PowerShell to naprawdę świetna sprawa, chociaż niektóre z nich mogą być dosyć złożone. Jeżeli dany przykład wygląda na zbyt skomplikowany, zignoruj go i sprawdź pozostałe przykłady albo po prostu samodzielnie poeksperymentuj, aby sprawdzić, czy potrafisz zrozumieć, jak to działa i dlaczego. Pamiętaj, aby takie eksperymenty zawsze przeprowadzać w środowisku testowym, a nie produkcyjnym!

3.6. Dostęp do ogólnych tematów pomocy

Wcześniej w tym rozdziale wspominaliśmy, że w systemie pomocy powłoki PowerShell znajdują się zarówno pliki opisujące tematy ogólne, jak i szczegółowe opisy poszczególnych poleceń *cmdlet*. Ogólne tematy pomocy są często nazywane tematami *About*, ponieważ ich nazwy plików zaczynają się od ciągu znaków *about_*. Jak zapewne pamiętasz, pisaliśmy też, że wszystkie cmdlety obsługują zestaw wspólnych parametrów (ang. *common parameters*). Jak myślisz, w jaki sposób możesz dowiedzieć się czegoś więcej o tych wspólnych parametrach?

ZRÓB TO SAM Zanim zaczniesz czytać dalej, sprawdź, czy korzystając z systemu pomocy powłoki PowerShell, możesz wyświetlić listę wspólnych parametrów.

Zacniemy od użycia symboli wieloznacznych. Ponieważ w języku angielskim wspólne parametry cmdletów to *common parameters*, słowo *common* będzie na początek dobrym słowem kluczowym do wyszukiwania:

```
Help *common*
```

Szczerze mówiąc, jest to tak dobre słowo kluczowe, że pasuje do niego tylko jeden temat pomocy: *about_common_parameters*. Temat ten wyświetla się automatycznie, ponieważ jest jedynym pasującym plikiem pomocy. Przeglądając jego zawartość, szybko znajdziesz poniższą listę ośmiu wspólnych parametrów:

```
-Verbose  
-Debug  
-WarningAction  
-WarningVariable  
-ErrorAction  
-ErrorVariable  
-OutVariable  
-OutBuffer
```

W pliku pomocy można znaleźć informację, że powłoka PowerShell posiada również dwa dodatkowe parametry wspomagające testowanie skryptów i pozwalające na ograniczanie ryzyka uruchamiania poleceń, ale nie każde polecenie *cmdlet* je obsługuje.

Tematy *About* dostępne w systemie pomocy powłoki PowerShell są niezwykle ważnym źródłem informacji, ale ponieważ nie są powiązane z żadnym konkretnym cmdletem, mogą być łatwo przeoczone. Jeśli uruchomisz polecenie *help about**, wyświetlające wszystkie tematy pomocy zawierające w nazwie słowo *about*, możesz być zaskoczony, jak wiele dodatkowej dokumentacji jest ukryte wewnątrz systemu pomocy powłoki.

3.7. Dostęp do pomocy online

Zawartość plików pomocy powłoki PowerShell została przygotowana przez zwykłych ludzi, a zatem nie są one wolne od błędów. Oprócz aktualizacji plików pomocy (którą można przeprowadzić, uruchamiając polecenie `Update-Help`) firma Microsoft udostępnia pliki pomocy na swojej stronie internetowej. Użycie w wywołaniu polecenia `Help` parametru `-online` spowoduje próbę wyświetlenia danego tematu pomocy bezpośrednio z sieci, co powinno działać nawet w systemie macOS lub Linux! Przykładowe wywołanie takiego sieciowego systemu pomocy może wyglądać na przykład tak:

```
Help Get-EventLog -online
```

Sieciowe pliki pomocy powłoki PowerShell są obsługiwane przez witrynę TechNet firmy Microsoft i często są bardziej aktualne niż te, które zostały zainstalowane w Twoim systemie. Jeśli uważasz, że zauważyłeś błąd w jakimś przykładzie lub w składni polecenia, spróbuj wyświetlić wersję online tego tematu pomocy. Niestety nie każdy cmdlet we wszechświecie ma dostępną pomoc online — przygotowanie odpowiedniego pliku należy do deweloperów poszczególnych produktów (takich jak zespół Exchange, zespół SQL Server czy zespół programu SharePoint). Nie zawsze taki plik zostaje opracowany, ale kiedy jest dostępny, stanowi znakomite uzupełnienie zawartości wbudowanego systemu pomocy.

Bardzo lubimy system pomocy dostępny online, ponieważ korzystając z niego, możemy wygodnie czytać tekst pomocy w oknie przeglądarki internetowej. Don ma szczególnie korzystać z systemu dwumonitorowego, co zapewnia bardzo wygodną pracę. Pamiętaj, że jeżeli pracując w konsoli tekstowej powłoki PowerShell, chcesz wyświetlić określony temat pomocy w osobnym oknie, możesz użyć przełącznika `-ShowWindow`, o którym mówiliśmy nieco wcześniej.

Pamiętaj także, że zespół firmy Microsoft zajmujący się powłoką PowerShell udostępnia od kwietnia 2016 roku wszystkie pliki pomocy jako projekt *open source*, dzięki czemu praktycznie każdy może dodawać nowe przykłady, poprawiać błędy i ogólnie pomagać w ulepszaniu plików pomocy. Z projektem *open source* powłoki PowerShell można się zapoznać na stronie <https://github.com/PowerShell/>. Zazwyczaj udostępniana jest tylko dokumentacja posiadana przez sam zespół PowerShella, stąd nie zawsze dostępna jest tam dokumentacja innych grup tworzących polecenia powłoki PowerShell. Zawsze możesz jednak poprosić takie zespoły o „uwolnienie” dokumentacji ich produktów i udostępnienie ich na licencji *open source*.

3.8. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Mamy nadzieję, że po przeczytaniu tego rozdziału będziesz już wiedział, jak ogromne znaczenie ma umiejętność sprawnego posługiwania się systemem pomocy powłoki PowerShell. Teraz nadszedł czas, aby powtórzyć sobie nabytą wiedzę i utrwalić zdobyte

umiejętności, wykonując zadania opisane poniżej. Pamiętaj, że na końcu rozdziału znajdziesz przykładowe odpowiedzi i rozwiązania zadań. Zwróć uwagę na słowa zapisane w treści zadań *kursywą* i użyj ich jako wskazówek do wykonania tychże ćwiczeń. Niektóre zadania dotyczą wyłącznie systemu Windows, ale w takich przypadkach na początku treści zadania zamieszczamy odpowiednią informację.

1. Uruchom polecenie `Update-Help` i upewnij się, że jego działanie zakończyło się bez żadnych błędów. Po wykonaniu tego polecenia system pomocy powłoki PowerShell zainstalowany na Twoim komputerze lokalnym zostanie zaktualizowany do najnowszej wersji. Pamiętaj, że do wykonania tego zadania potrzebne Ci będzie połączenie z siecią Internet, a powłoka musi działać z podniesionymi uprawnieniami (co oznacza, że na pasku tytułowym okna powłoki musi być widoczne słowo *Administrator*).
2. Wyłącznie dla systemu Windows: czy uda Ci się znaleźć jakieś polecenia *cmdlet* umożliwiające przekształcenie wyników działania innych *cmdlet*ów do formatu *HTML*?
3. Częściowo dla systemu Windows: czy istnieją jakieś polecenia *cmdlet*, które mogą przekierowywać wyniki działania innych poleceń bezpośrednio do *pliku* (ang. *file*) lub na *drukarkę* (ang. *printer*)?
4. Ile *cmdlet*ów jest przeznaczonych do pracy z *procesami* (ang. *process*)? (Podpowiedź: pamiętaj, że w nazwach poleceń *cmdlet* stosowane są angielskie rzeczowniki w liczbie pojedynczej).
5. Jakiego polecenia *cmdlet* można użyć do *zapisania* (ang. *write*) nowego zdarzenia w *dzienniku* zdarzeń (ang. *log*)? (Taka operacja jest również możliwa w systemach operacyjnych innych niż Windows, ale wtedy otrzymasz inną odpowiedź).
6. Wspominaliśmy, że aliasy są swego rodzaju skróconymi wersjami nazw poleceń *cmdlet* — jakie *cmdlety* pozwalają na tworzenie, modyfikowanie, eksportowanie lub importowanie *aliasów*?
7. Czy istnieje sposób na zachowanie historii wszystkiego, co *wpisujesz* (ang. *transcript*) w powłoce, i zapisanie jej w pliku tekstowym?
8. Wyłącznie dla systemu Windows: wyświetlenie wszystkich wpisów z wybranego *dziennika zdarzeń* (ang. *event log*) może zająć dużo czasu. Jak wyświetlić tylko 100 najnowszych wpisów?
9. Wyłącznie dla systemu Windows: czy istnieje sposób pobrania listy *usług* (ang. *services*) zainstalowanych na komputerze zdalnym?
10. Czy istnieje możliwość sprawdzenia, jakie *procesy* są uruchomione na komputerze zdalnym? (Odpowiedz na to pytanie możesz znaleźć również dla systemów operacyjnych innych niż Windows, ale samo polecenie może nie działać poprawnie).
11. Sprawdź plik pomocy dla polecenia `Out-File`. Jaka jest domyślna liczba znaków w wierszu dla plików tworzonych za pomocą tego *cmdletu*? Czy istnieje parametr, który umożliwia zmianę domyślnej szerokości wiersza (liczby znaków w wierszu)?

12. Domyślnie cmdlet Out-File nadpisuje istniejący plik o takiej samej nazwie. Czy istnieje parametr, za pomocą którego możesz uniemożliwić nadpisywanie istniejących plików?
13. Jak można wyświetlić listę wszystkich *aliasów* zdefiniowanych w powłoce PowerShell?
14. Używając zarówno aliasów, jak i skróconych nazw parametrów, spróbuj utworzyć najkrótsze możliwe polecenie, które będzie wyświetlało listę procesów działających na komputerze o nazwie Server1.
15. Ile cmdletów jest przeznaczonych do pracy z *obiektami*? (Podpowiedź: pamiętaj, aby używać pojedynczego rzeczownika *object*, a nie liczby mnogiej *objects*).
16. W tym rozdziale krótko wspominaliśmy o *tablicach* (ang. *array*). Jak możesz się o nich dowiedzieć czegoś więcej?

3.9. Odpowiedzi

1. Update-Help
Jeżeli uruchamiasz aktualizację częściej niż raz dziennie, możesz użyć polecenia Update-Help -force.
2. help html
Zamiast tego możesz również użyć polecenia Get-Command:
get-command -noun html
3. get-command -noun file.printer
4. get-command -noun process
lub
Help *Process
5. get-command -verb write -noun eventlog
Jeżeli nie jesteś pewny, jakiego rzeczownika (-noun) użyć, zastosuj symbol wieloznaczny:
help *log
6. help *alias
lub
get-command -noun alias
7. help transcript
8. help Get-Eventlog -parameter Newest
9. help Get-Service -parameter computername
10. help Get-Process -parameter computername
11. Help Out-File -full
lub
Help Out-File -parameter Width
Wykonanie tego polecenia powinno pokazać, że domyślna szerokość wiersza to 80 znaków. Ten sam parametr możesz stosować do zmiany domyślnej liczby znaków w wierszu.
12. Jeżeli wyświetlisz pełny plik pomocy cmdletu Out-File przy użyciu polecenia Help Out-File -full, na liście opcji znajdziesz parametr -NoClobber.

- 13. `Get-Alias`
- 14. `ps -c server1`
- 15. `get-command -noun object`
- 16. `help about_arrays`
Zamiast tego możesz również użyć symboli wieloznacznych:
`help *array*`

Uruchamianie poleceń

4

Kiedy zaczyna się przeglądać przykłady zastosowania powłoki PowerShell dostępne w sieci Internet, łatwo można odnieść wrażenie, że PowerShell to rodzaj skryptów opartych na .NET Framework lub po prostu że jest to jakiś rodzaj języka programowania. Warto jednak zauważyć, że zarówno wielu naszych współpracowników, laureatów nagrody Microsoft Most Valuable Professional (MVP), jak i setki innych, zaawansowanych użytkowników powłoki PowerShell zaczynało swoją przygodę od tego, gdzie zaczyna się ten rozdział: od uruchamiania prostych i bardziej złożonych poleceń. I to właśnie będziemy robić w tym rozdziale: nie będziemy się zajmować tworzeniem skryptów, nie będziemy się bawić w programowanie, ale po prostu zajmiemy się uruchamianiem poleceń i korzystaniem z narzędzi wiersza poleceń.

4.1. Nie tworzymy skryptów, ale uruchamiamy polecenia

PowerShell, jak sama nazwa wskazuje, jest **powłoką** systemu (ang. *shell* — powłoka). Powłoka PowerShell jest nieco podobna do konsoli `cmd.exe`, której prawdopodobnie używałeś wcześniej, a przy odrobinie dobrych chęci możemy znaleźć nawet pewne podobieństwo do starego dobrego systemu MS-DOS, który był dostarczany z pierwszymi komputerami w latach 80. Powłoka PowerShell wykazuje również silne podobieństwo do powłok systemu UNIX, takich jak Bash, z końca lat 80. czy nawet do oryginalnej powłoki Bourne Shell systemu UNIX, wprowadzonej pod koniec lat 70. Powłoka PowerShell jest oczywiście znacznie bardziej nowoczesna, ale w końcu PowerShell nie jest przecież tylko językiem skryptowym takim jak VBScript czy KiXtart.

W językach skryptowych, podobnie jak w większości innych języków programowania, siadasz przed edytorem tekstu (nawet jeśli jest to Notatnik systemu Windows) i wpisujesz serię poleceń i słów kluczowych tworzących skrypt. Następnie zapisujesz taki plik i być może jeszcze dodatkowo go testujesz, zanim zaczniesz używać na dobre. Powłoka

PowerShell może oczywiście być używana w taki sposób, ale niekoniecznie jest to jej podstawowy wzorzec działania, szczególnie gdy jesteś początkującym użytkownikiem. Pracując z powłoką PowerShell, wpisujesz polecenie, dodajesz kilka parametrów, aby dostosować jego zachowanie do konkretnego zadania, naciskasz klawisz *Enter* i natychmiast otrzymujesz wyniki działania.

Po upływie pewnego czasu będziesz miał w końcu dość ciągłego wpisywania w kółko tego samego polecenia (i jego parametrów), więc pewnie skopiujesz je i wkleisz do pliku tekstowego. Potem nadasz temu plikowi rozszerzenie *.PS1* i nagle okaże się, że otrzymałeś **skrypt PowerShella**. Teraz, zamiast ponownie wpisywać to samo polecenie i cały ciąg jego parametrów, uruchamiasz po prostu ten skrypt, który wykonuje wszystkie polecenia znajdujące się wewnątrz. Jest to taki sam wzorzec zachowania, który mógł być używany w plikach wsadowych w powłoce *Cmd.exe*, ale zazwyczaj jest to proces znacznie mniej złożony niż samo pisanie skryptów lub programowanie. W rzeczywistości schemat postępowania jest podobny do tego stosowanego od dziesięcioleci przez administratorów systemów UNIX/Linux używających powłok takich jak Bash: uruchamiasz polecenia „z ręki”, dopóki nie uzyskasz pewności, że działają poprawnie, a następnie wklejasz je do pliku tekstowego i nazywasz **skryptem** (ang. *script*).

Nie zrozum nas źle: korzystając z powłoki PowerShell, możesz tworzyć tak złożone skrypty i narzędzia, jak będzie Ci to potrzebne. Powłoka obsługuje takie same mechanizmy jak VBScript i inne języki skryptowe czy języki programowania. PowerShell daje użytkownikowi dostęp do wszystkich mechanizmów środowiska .NET Framework (choć w systemach operacyjnych innych niż Windows jej możliwości w tym zakresie mogą być nieco mniejsze) i niejednokrotnie widzieliśmy „skrypty” powłoki PowerShell, których praktycznie nie dawało się odróżnić od programów napisanych w Visual Studio. Powłoka PowerShell jest bardzo elastyczna i obsługuje różne wzorce użycia, ponieważ z założenia ma być przydatna bardzo szerokiemu gronu odbiorców. Chodziło nam po prostu o to, że choć powłoka *obsługuje* bardzo złożone mechanizmy systemowe i może z nich korzystać, nie oznacza to, że Ty *musisz* z nich korzystać ani że nie możesz pracować bardzo efektywnie przy użyciu prostych, mniej złożonych rozwiązań.

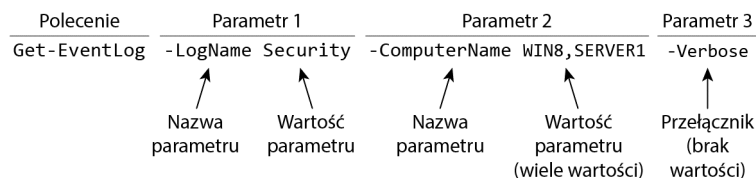
Spróbujemy się tutaj podeprzeć małą analogią: z dużą dozą prawdopodobieństwa możemy chyba założyć, że posiadasz prawo jazdy i prowadzisz samochód. Jeżeli jesteś choć trochę podobny do nas, to wymiana oleju jest prawdopodobnie najbardziej skomplikowanym zadaniem, jakie kiedykolwiek wykonałeś w swoim samochodzie. My też nie jesteśmy mechanikami samochodowymi ani maniakami motoryzacji i z całą pewnością nie potrafimy ani samodzielnie rozłożyć silnika na części, ani go ponownie złożyć. Nie potrafimy również jeździć tak agresywnie ani wykonywać takich fajnych, efektownych manewrów, jakie można zobaczyć w filmach. Możemy się śmiało założyć o każde pieniądze, że nigdy nie zobaczysz nas, jak prowadzimy samochód na zamkniętym torze wyścigowym w reklamie samochodowej (choć Jeff bardzo o tym marzy, ale on zdecydowanie za dużo ogląda programu *Top Gear*). Z drugiej jednak strony sam fakt, że nie jesteśmy profesjonalnymi kierowcami, w niczym nie zmienia tego, że jesteśmy całkiem dobrymi użytkownikami samochodów na nieco mniej zaawansowanym, normalnym poziomie. Ba, któregoś dnia być może zdecydujemy się nawet na udział w kursach jazdy

dla kaskaderów i będziemy to traktować jako przyjemne i podniecające hobby (a nasze towarzystwa ubezpieczeniowe będą z pewnością tym zachwycone...). Oczywiście po podjęciu takiej decyzji będziemy musieli zapewne dowiedzieć się nieco więcej o tym, jak działają nasze samochody, opanować nowe umiejętności i tak dalej, ale taka opcja jest dla nas zawsze dostępna. Na razie jednak jesteśmy w pełni zadowoleni z tego, co możemy osiągnąć jako normalni kierowcy.

Z tych samych powodów na razie pozostaniemy zwykłymi użytkownikami powłoki PowerShell, korzystającymi z niej na normalnym, przeciętnym poziomie. Wierz lub nie, ale to właśnie tacy zwykli użytkownicy są głównym, docelowym segmentem odbiorców powłoki PowerShell i wkrótce przekonasz się, że nawet na tym poziomie możesz wykonać wiele niesamowitych rzeczy. Wszystko, co musisz na początek zrobić, to nauczyć się uruchamiać różne polecenia w powłoce, i już będziesz na dobrej drodze do osiągnięcia celu.

4.2. Anatomia polecenia

Rysunek 4.1 pokazuje podstawową budowę złożonego polecenia powłoki PowerShell. Nazywamy to **pełną składnią** polecenia (ang. *full-form syntax*). Pokazujemy tutaj nieco bardziej złożone polecenie, dzięki czemu możesz zobaczyć wszystkie rzeczy, które mogą się pojawić.



Rysunek 4.1. Anatomia polecenia powłoki PowerShell

Aby upewnić się, że znasz reguły tworzenia poleceń powłoki PowerShell, przyjrzyj się bliżej każdemu z elementów pokazanych na powyższym rysunku:

- Nazwa polecenia to Get-EventLog. Nazwy poleceń *cmdlet* powłoki PowerShell zawsze mają taki sam format: czasownik-rzeczownik (ang. *verb-noun*). Więcej szczegółowych informacji na temat poleceń *cmdlet* znajdziesz w następnej sekcji.
- Pierwszym parametrem naszego polecenia jest -LogName, który otrzymuje wartość Security. Ponieważ taki ciąg znaków nie zawiera żadnych spacji ani innych znaków specjalnych, nie musimy umieszczać go w apostrofach.
- Drugi parametr polecenia to -ComputerName, który otrzymuje dwie wartości: WIN8 i SERVER1. Zostały one zapisane w postaci listy wartości rozdzielanych od siebie przecinkami. Podobnie jak w poprzednim przypadku, żadna z tych wartości nie zawiera spacji ani innych znaków specjalnych, zatem nie musimy umieszczać ich w apostrofach.
- Ostatni parametr, -Verbose, jest tak zwanym przełącznikiem. Oznacza to, że nie musimy mu nadawać żadnych wartości, a znaczenie ma sama jego obecność w wierszu wywołania polecenia.

- Zwróć uwagę, że pomiędzy nazwą polecenia a jego pierwszym parametrem znajduje się obowiązkowa spacja.
- Nazwy parametrów zawsze rozpoczynają się od myślnika (-).
- Po nazwie parametru oraz między wartością parametru i następną nazwą parametru występują obowiązkowe spacje.
- Nie wstawiamy spacji między myślnikiem (-), który poprzedza nazwę parametru, a samą nazwą parametru.
- Nie zwracamy tutaj uwagi na pisownię małych i wielkich liter.

Przyzwyczaj się do tych zasad. Staraj się precyzyjnie i bezbłędnie wpisywać nazwy poleceń i ich parametry. Zwracanie szczególnej uwagi na spacje, kreski oraz inne reguły tworzenia poleceń znacząco może przyczynić się do zmniejszenia ilości głupich, niepotrzebnych błędów, z którymi z całą pewnością będziesz się niejednokrotnie borykał podczas pracy z powłoką PowerShell.

4.3. Konwencja tworzenia nazw poleceń *cmdlet*

Na początek skoncentrujemy się na omówieniu kilku podstawowych pojęć. O ile nam wiadomo, jesteśmy jedynymi, którzy używają tej terminologii w codziennych rozmowach, ale robimy to konsekwentnie, więc równie dobrze możemy spróbować Ci to wyjaśnić:

- Polecenie *cmdlet* to natywne narzędzie wiersza polecenia powłoki PowerShell. Cmdlety funkcjonują tylko wewnątrz powłoki PowerShell i są napisane w języku środowiska .NET Framework, takim jak C#. Samo określenie *cmdlet* jest unikatowe i charakterystyczne dla powłoki PowerShell, stąd jeżeli dodasz je do słów kluczowych w wyszukiwarce Google lub Bing, wyniki, które otrzymasz, będą związane głównie z powłoką PowerShell. Słowo *cmdlet* wymawiamy zwyczajowo jako *komandlet*.
- **Funkcje** powłoki PowerShell (ang. *functions*) są podobne do poleceń *cmdlet*, z tym że są napisane we własnym języku skryptowym powłoki PowerShell, a nie w języku środowiska .NET Framework.
- **Przepływ pracy** (ang. *workflow*) to specjalny rodzaj funkcji, który jest powiązany z systemem zarządzania przepływem pracy PowerShella.
- **Aplikacja** (ang. *application*) to dowolny zewnętrzny plik wykonywalny, włącznie z narzędziami uruchamianymi z poziomu wiersza polecenia, takimi jak `ping` czy `ipconfig`.
- **Polecenie** (ang. *command*) jest ogólnym terminem używanym w odniesieniu do dowolnego lub wszystkich poprzednich elementów.

Firma Microsoft przyjęła pewną konwencję nazewnictwa dla poleceń *cmdlet*. Ta sama konwencja nazewnictwa *powinna* być stosowana również w odniesieniu do funkcji i przepływów pracy, chociaż Microsoft nie może zmusić nikogo oprócz swoich własnych programistów do przestrzegania tych reguł.

Wspomniana konwencja jest następująca: nazwy poleceń zaczynają się od standardowego czasownika, takiego jak Get, Set, New czy Pause. Aby wyświetlić listę dopuszczalnych czasowników, możesz wykonać polecenie Get-Verb (wszystkich czasowników jest około setki, ale tylko trochę ponad dwa tuziny należy do grupy Common). Po czasowniku znajduje się myślnik, a po nim rzeczownik w liczbie pojedynczej, taki jak Service, Process czy EventLog. Programiści używają swoich własnych rzeczowników w miarę potrzeb, więc nie istnieje polecenie *cmdlet* takie jak Get-Noun, które by je wyświetlało.

Zatem co takiego nadzwyczajnego jest w tej regule? Załóżmy, że istnieją polecenia *cmdlet* takie jak New-Service, Get-Service, Get-Process czy Set-Service. Czy bazując na nazwach wymienionych poleceń, możesz zgadnąć, za pomocą jakiego cmdletu możesz utworzyć nową skrzynkę pocztową Exchange? Czy możesz zgadnąć, jakie polecenie zmodyfikowałoby ustawienia konta użytkownika w Active Directory? Jeżeli na pierwsze pytanie odpowiedziałeś Get-Mailbox, to miałeś rację. Jeżeli na drugie pytanie odpowiedziałeś Set-User, to byłeś bardzo blisko, bo poprawna odpowiedź brzmi Set-ADUser (znajdziesz takie polecenie na kontrolerach domeny w module Active-Directory). Chodzi więc o to, że mając taką spójną konwencję nazewnictwa z ograniczonym zbiorem czasowników, możesz przewidywać nazwy nowych poleceń, a następnie użyć polecenia Help lub Get-Command wraz z symbolami wieloznacznymi do potwierdzenia swoich przypuszczeń. Dzięki przyjęciu takiej konwencji wymyślanie nazw potrzebnych poleceń stało się znacznie łatwiejsze i minimalizuje konieczność korzystania za każdym razem z wyszukiwarek Google lub Bing.

WSKAZÓWKA Nie wszystkie tak zwane czasowniki są naprawdę czasownikami. Chociaż Microsoft oficjalnie używa określenia *konwencja czasownik-rzeczownik*, w zestawieniach poleceń zobaczysz takie „czasowniki” jak New (nowy) czy Where (gdzie). Przyzwyczaj się.

4.4. Aliasy — skrócone nazwy poleceń

Chociaż nazwy poleceń powłoki PowerShell mogą być ładne i spójne, to jednak mogą również być długie. Nazwy niektórych poleceń, takich jak Set-WinDefaultInputMethodOverride, wymagają naprawdę sporej liczby kliknięć, nawet przy korzystaniu z mechanizmu dopełniania za pomocą klawisza *Tab*. Dzięki takiej opisowej nazwie możesz z dużą dozą prawdopodobieństwa odgadnąć przeznaczenie takiego polecenia, ale jej wpisywanie może być dosyć *uciążliwe*.

Właśnie tutaj z pomocą przychodzą nam aliasy poleceń powłoki PowerShell. Alias jest niczym więcej niż skróconą nazwą czy nawet swego rodzaju pseudonimem polecenia. Masz już dość wpisywania Get-Service? Spróbuj tego:

```
PS C:\> get-alias -Definition "Get-Service"
Capability      Name
-----
Cmdlet          gsv -> Get-Service
```

Teraz wiesz, że gsv jest aliasem polecenia Get-Service.

Jeżeli używasz aliasu, polecenie działa dokładnie w ten sam sposób. Parametry są takie same, wszystko jest takie samo — jedynie nazwa polecenia jest krótsza. Jeżeli jesteś przyzwyczajony do pracy z powłoką systemów UNIX lub Linux, gdzie aliasy mogą również zawierać pewne parametry, to pamiętaj, że powłoka PowerShell nie działa w ten sposób.

Jeśli patrzysz na jakiś alias (użytkownicy w sieci Internet często używają ich tak, jak byśmy wszyscy pamiętali setki wbudowanych aliasów) i nie masz pojęcia, co to jest, skorzystaj z systemu pomocy:

```
PS C:\> help gsv
NAME
    Get-Service
SYNOPSIS
    Gets the services on a local or remote computer.
SYNTAX
    Get-Service [[-Name] <String[]>] [-ComputerName <String[]>]
    [-DependentServices [<SwitchParameter>]] [-Exclude <String[]>]
    [-Include <String[]>] [-RequiredServices [<SwitchParameter>]]
    [<CommonParameters>]
    Get-Service [-ComputerName <String[]>] [-DependentServices
    [<SwitchParameter>]] [-Exclude <String[]>] [-Include <String[]>]
    [-RequiredServices [<SwitchParameter>]] -DisplayName <String[]>
    [<CommonParameters>]
    Get-Service [-ComputerName <String[]>] [-DependentServices
    [<SwitchParameter>]] [-Exclude <String[]>] [-Include <String[]>]
    [-InputObject <ServiceController[]>] [-RequiredServices
    [<SwitchParameter>]] [<CommonParameters>]
```

Jeżeli poprosisz system pomocy o wyświetlenie informacji na temat danego aliasu, system ten zawsze wyświetli treść pomocy dla pełnego polecenia, do którego odnosi się ten alias.

Dla zainteresowanych

Własne aliasy możesz tworzyć za pomocą polecenia `New-Alias`, możesz eksportować listę aliasów przy użyciu polecenia `Export-Alias`, a nawet importować listę wcześniej utworzonych aliasów za pomocą polecenia `Import-Alias`. Jeżeli utworzysz własny alias, będzie on dostępny tylko w bieżącej sesji powłoki. Gdy zamkniesz okno, alias zniknie i dlatego być może będziesz chciał wyeksportować aliasy, tak aby móc je w razie potrzeby później ponownie zaimportować do powłoki.

Starajmy się jednak unikać tworzenia niestandardowych, własnych aliasów, ponieważ są one dostępne tylko dla nas. Przykładowo, jeżeli jakiś użytkownik z zewnątrz nie będzie w stanie sprawdzić, co robi polecenie `xtd`, którego używamy w naszym skrypcie, to możemy w taki sposób spowodować niepotrzebne zamieszanie i niekompatybilność.

A swoją drogą polecenie `xtd` nie robi nic. To po prostu fałszywy alias, który wymyśliliśmy na potrzeby tego przykładu.

Warto zauważyć, że ze względu na to, iż powłoka PowerShell jest teraz dostępna w systemach operacyjnych innych niż Windows, jej koncepcja *aliasu* w systemie Windows różni się nieco od aliasu w systemie, powiedzmy, Linux. W systemie Linux alias może być rodzajem skrótu do uruchamiania polecenia i *zawierać szereg „zaszytych” parametrów*.

PowerShell nie zachowuje się w ten sposób. Alias jest *tylko* skróconą wersją nazwy polecenia i nie może zawierać żadnych z góry określonych parametrów.

4.5. Tworzenie skrótów

I tutaj zaczynają się schody. Naprawdę bardzo chcielibyśmy powiedzieć, że wszystko, co do tej pory Ci pokazaliśmy, jest jedynym sposobem wykonywania takich operacji, ale niestety nie byłaby to prawda. Korzystając z różnych przykładów i gotowych skryptów powłoki PowerShell dostępnych w sieci Internet, będziesz musiał samodzielnie analizować ich kod i wiedzieć, na co patrzysz i jak to interpretować.

Oprócz aliasów, które są krótszymi wersjami nazw poleceń, można również korzystać ze *skróconych* wersji parametrów. Istnieją trzy sposoby tworzenia takich skrótów, z których każdy jest potencjalnie jeszcze bardziej zagmatwany niż poprzedni.

4.5.1. Używanie skróconych nazw parametrów

Uruchamiając polecenia powłoki PowerShell, nie musisz wpisywać pełnych nazw parametrów. Przykładowo, zamiast pisać `-computerName`, możesz użyć opcji `-comp`. Cała sztuczka polega na tym, że musisz wpisać wystarczająco długi fragment nazwy parametru, aby powłoka PowerShell mogła ją jednoznacznie zidentyfikować. Jeżeli w wierszu polecenia występują parametry `-computerName`, `-common` i `-composite`, to chcąc skorzystać z nazw skróconych, powinieneś wpisać co najmniej następujące ciągi znaków: `-compu`, `-commo` i `-compo`, co w jednoznaczny sposób pozwoli na identyfikację każdego z parametrów.

W praktyce oczywiście nie ma nic złego w korzystaniu ze skróconych nazw parametrów, ale mimo to powinieneś w sobie wyrobić nawyk, że po wpisaniu skróconej nazwy parametru naciskasz klawisz *Tab*, tak aby powłoka PowerShell mogła automatycznie uzupełnić brakującą resztę wyrażenia za Ciebie.

4.5.2. Używanie aliasów nazw parametrów

Parametry również mogą mieć własne aliasy, aczkolwiek ze względu na to, że takie aliasy nie są wyświetlane w plikach pomocy, mogą one być bardzo trudne do znalezienia. Na przykład polecenie `Get-EventLog` ma parametr o nazwie `-computerName`. Aby znaleźć alias takiego parametru, powinieneś wykonać następujące polecenie:

```
PS C:\> (get-command get-eventlog | select -ExpandProperty parameters).computername.alias
```

W powyższym przykładzie nazwę polecenia i parametru wyróżniliśmy dodatkowo pochyloną czcionką, dzięki czemu możesz łatwo zamienić je na dowolne inne polecenie i parametr, który Cię interesuje. W naszym przypadku wyniki działania pokazują, że aliasem parametru `-ComputerName` jest `-Cn`, stąd możesz uruchomić na przykład takie polecenie:

```
PS C:\> Get-EventLog -LogName Security -Cn SERVER2 -Newest 10
```

Mechanizm dopełniania nazw przy użyciu klawisza *Tab* również powoduje wyświetlanie aliasów parametrów; przykładowo, jeżeli wpiszesz polecenie `Get-EventLog -C` i zaczniesz

naciskać klawisz *Tab*, w pewnym momencie pojawi się alias `-Cn`. Pamiętaj jednak, że system pomocy dla tego polecenia w ogóle nie wyświetla aliasu `-Cn`, a samo dopełnianie przy użyciu klawisza *Tab* wcale nie musi oznaczać, że `-Cn` i `-ComputerName` to jest ten sam parametr.

4.5.3. Używanie parametrów pozycyjnych

Przeglądając składnię polecenia w pliku pomocy, można łatwo odszukać jego parametry pozycyjne:

SYNTAX

```
Get-ChildItem [[-Path] <String[]>] [[-Filter] <String>] [-Exclude <String[]>]
[-Force [<SwitchParameter>]] [-Include <String[]>] [-Name [<SwitchParameter>]]
[-Recurse [<SwitchParameter>]] [-UseTransaction [<SwitchParameter>]] [<CommonParameters>]
```

Jak łatwo zauważyć, parametry `-Path` oraz `-Filter` są pozycyjne i wiemy o tym, ponieważ nazwa parametru została ujęta w nawiasy kwadratowe. Znacznie dokładniejsze objaśnienia możemy znaleźć w pełnym opisie tego cmdletu, który możemy wyświetlić za pomocą polecenia `Help Get-ChildItem -full`, tak jak to zostało pokazane poniżej:

```
-Path <String[]>
    Specifies a path to one or more locations. Wildcards are permitted. The default location
    ↪ is the current directory (.).
    Required?                false
    Position?                1
    Default value             Current directory
    Accept pipeline input?    true (ByValue, ByPropertyName)
    Accept wildcard characters? True
```

W opisie polecenia jest informacja, że w wierszu wywołania polecenia parametr `-Path` znajduje się na pozycji numer 1. Jak pamiętasz, w przypadku parametrów pozycyjnych nie musimy wpisywać nazwy parametru; zamiast tego możemy podać jego wartość na właściwej pozycji. Na przykład:

```
PS C:\> Get-ChildItem c:\users
    Directory: C:\users
Mode                LastWriteTime         Length      Name
----                -
d----      3/27/2016 11:20 AM
d-r--      2/18/2016  2:06 AM      donjones
Public
```

Wyniki działania powyższego polecenia są identyczne jak wyniki działania jego pełnej wersji zawierającej nazwę parametru:

```
PS C:\> Get-ChildItem -path c:\users
    Directory: C:\users
Mode                LastWriteTime         Length      Name
----                -
d----      3/27/2016 11:20 AM
d-r--      2/18/2016  2:06 AM      donjones
Public
```

Problem z parametrami pozycyjnymi polega na tym, że bierzesz na siebie odpowiedzialność za to, co się dzieje w wierszu polecenia. Najpierw zawsze musisz wpisać w poprawnej kolejności wszystkie parametry pozycyjne, a dopiero później możesz dodać jakiegokolwiek pozostałe parametry (wymagające podania nazw bądź ich skrótów lub aliasów). Jeżeli pomylisz kolejność parametrów pozycyjnych, próba wykonania polecenia zakończy się niepowodzeniem lub nawet przyniesie nieprzewidziane rezultaty. W przypadku prostych poleceń, takich jak `Dir`, którego prawdopodobnie używasz od lat, gdybyś dopisał nazwę parametru `-Path`, z pewnością czułbyś się dziwnie i w praktyce chyba nikt tego nie robi. Ale w przypadku bardziej złożonych poleceń, które mogą mieć trzy lub cztery parametry pozycyjne z rzędu, może być trudno zapamiętać ich znaczenie i właściwą kolejność.

Na przykład takie polecenie jest nieco trudne do odczytania i zinterpretowania:

```
PS C:\> move file.txt users\donjones\
```

Jeżeli jednak używamy nazw parametrów, całość staje się znacznie łatwiejsza do przeanalizowania:

```
PS C:\> move -Path c:\file.txt -Destination \users\donjones\
```

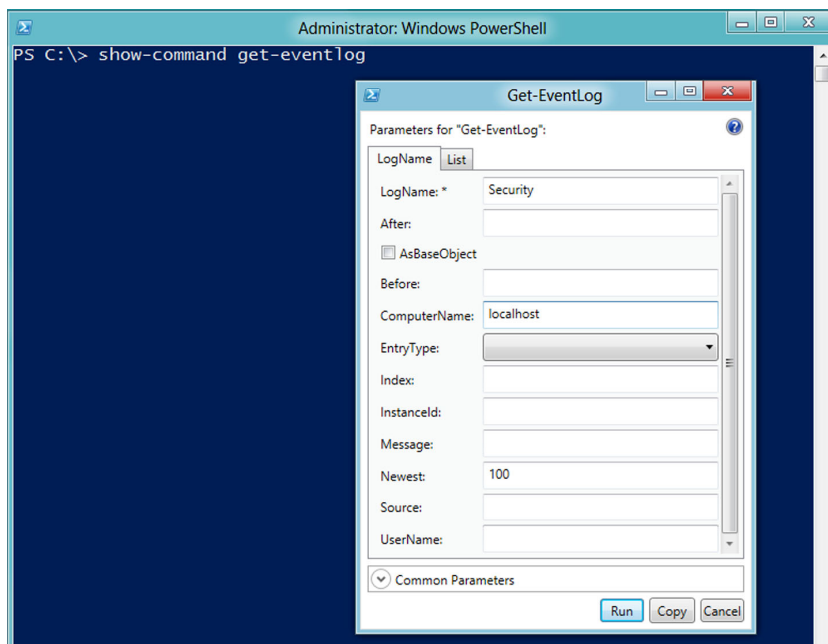
Co więcej, jeżeli używamy nazw parametrów, w wierszu wywołania polecenia możemy umieszczać parametry w innej kolejności:

```
PS C:\> move -Destination \users\donjones\ -Path c:\file.txt
```

Zwykle zalecamy, aby nie stosować parametrów pozycyjnych (bez podawania ich nazwy), chyba że akurat musisz na szybko wykonać jakieś proste polecenie. We wszystkich innych przypadkach, a zwłaszcza kiedy tworzysz skrypt, plik wsadowy czy chociażby wpis na blogu, powinieneś w poleceniach zawsze umieszczać wszystkie nazwy używanych parametrów. W naszej książce również staramy się to robić tak często, jak to tylko możliwe, z wyjątkiem kilku przypadków, w których musieliśmy skrócić wiersz poleceń, żeby pasował do formatu drukowanych stron.

4.6. Trochę oszukiwania — polecenie Show-Command

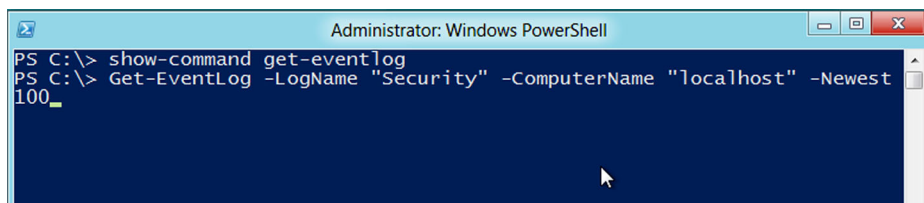
Pomimo naszego długiego doświadczenia w korzystaniu z powłoki PowerShell, złożoność składni niektórych poleceń czasami doprowadza nas do szału. Jednym z nowych fajnych poleceń powłoki PowerShell v3 (i nowszych wersji, choć nie w systemach operacyjnych innych niż Windows) jest cmdlet `Show-Command`. Jeżeli masz kłopoty z przypomnieniem sobie poprawnej składni jakiegoś polecenia, z tymi wszystkimi spacjami, myślnikami, przecinkami, apostrofami i innymi znakami, to z pewnością szybko zaprzyjaźnisz się z poleceniem `Show-Command`. Parametrem wywołania tego cmdletu jest nazwa polecenia, z którym się zmagasz, a po jego uruchomieniu na ekranie pojawia się graficzne okno dialogowe z prostym formularzem pozwalającym na wygodne wprowadzanie wszystkich parametrów wywoływanego polecenia. Jak pokazano na rysunku 4.2, poszczególne zestawy parametrów (mówiliśmy o nich w poprzednim rozdziale) znajdują się



Rysunek 4.2. Show-Command wykorzystuje graficzne okno dialogowe do wprowadzania parametrów wywoływanego polecenia

na oddzielnych kartach, więc nie ma możliwości pomieszczenia i niedopasowania parametrów z różnych zestawów. Wartości parametrów powinnyś podawać tylko na jednej, wybranej karcie. Pamiętaj, że polecenie Show-Command nie będzie działać w systemie, w którym graficzny interfejs użytkownika nie został zainstalowany.

Po zakończeniu wpisywania wartości niezbędnych parametrów możesz nacisnąć przycisk *Run* (uruchom), aby uruchomić polecenie, lub *Copy* (kopiuj), aby skopiować utworzone polecenie do schowka systemowego. Po skopiowaniu polecenia możesz powrócić do powłoki i wkleić polecenie w wierszu poleceń (aby to zrobić, kliknij prawym przyciskiem myszy w konsoli lub naciśnij kombinację klawiszy *Ctrl+V* w środowisku ISE). Po wklejeniu skopiowanego polecenia do wiersza poleceń możesz go spokojnie obejrzeć i przeanalizować. Jest to naprawdę świetny sposób na nauczanie się korzystania z poszczególnych poleceń, ponieważ za każdym razem otrzymujesz poprawną, pełną składnię polecenia (zobacz rysunek 4.3).



Rysunek 4.3. Show-Command na podstawie podanych parametrów generuje poprawną składnię polecenia

Kiedy wykonasz polecenie w ten sposób, zawsze otrzymasz jego pełną, formalną składnię: pełną nazwę polecenia, pełne nazwy wszystkich parametrów (również dla parametrów pozycyjnych) i tak dalej. To naprawdę świetny sposób, aby w doskonały, preferowany i sprawdzony sposób poznać parametry i poprawną składnię poleceń powłoki PowerShell.

Niestety Show-Command działa tylko z pojedynczymi komendami. Jeżeli chcesz łączyć ze sobą wiele poleceń, to może Ci ono pomóc tylko z jednym poleceniem jednocześnie.

4.7. Obsługa poleceń zewnętrznych

Do tej pory wszystkie polecenia, które uruchamialiśmy w powłoce (a przynajmniej te, które sugerowaliśmy, aby uruchomić), były wbudowanymi poleceniami *cmdlet*. W desktopowych wersjach systemu Windows możemy znaleźć ponad 400 wbudowanych cmdletów, a w serwerowych systemach Windows mamy ich do dyspozycji dosłownie tysiące, a może ich być jeszcze więcej — produkty takie jak Exchange Server, SharePoint Server i SQL Server są instalowane z pakietami zawierającymi setki dodatkowych poleceń *cmdlet*.

Ale nie jesteś ograniczony tylko do cmdletów, które są dostępne w samej powłoce PowerShell. Możesz także używać tych samych zewnętrznych narzędzi wiersza poleceń, których prawdopodobnie używasz od lat, takich jak ping, nslookup, ipconfig czy net. Ponieważ nie są one natywnymi cmdletami powłoki PowerShell, używasz ich w taki sam sposób jak zawsze. PowerShell wywoła w tle konsolę cmd.exe, ponieważ „wie”, jak uruchomić takie komendy zewnętrzne, a jednocześnie wszelkie wyniki ich działania zostaną wyświetlone w oknie powłoki PowerShell. Śmiało, nie czekaj i wypróbuj kilka swoich starych ulubionych poleceń już teraz. Często nasi studenci pytają nas, w jaki sposób można użyć powłoki PowerShell do zmapowania dysku sieciowego, tak aby można go było zobaczyć z poziomu Eksploratora Windows. Zawsze odpowiadamy, że używamy do tego celu polecenia net use i że w powłoce PowerShell sprawdza się ono znakomicie.

To samo dotyczy systemów operacyjnych innych niż Windows: możesz używać takich poleceń jak grep, bash, sed, awk, ping i wszystkich innych istniejących narzędzi wiersza poleceń. Będą funkcjonować normalnie, a PowerShell wyświetli wyniki ich działania w taki sam sposób, jak zrobiłaby to Twoja stara powłoka (na przykład Bash).

ZRÓB TO SAM Spróbuj uruchomić wybrane zewnętrzne narzędzia wiersza poleceń, z których korzystałeś już wcześniej. Czy działają tak samo? Czy próba uruchomienia któregoś z nich zakończyła się niepowodzeniem?

Przykład z poleceniem net use ilustruje ważną lekcję: wprowadzając powłokę PowerShell, Microsoft (być może po raz pierwszy w historii) nie powiedział: „Musisz zacząć od początku i uczyć się wszystkiego od nowa”. Zamiast tego Microsoft mówi: „Jeśli już wiesz, jak coś zrobić, rób to dalej w ten sam sposób. Postaramy się udostępnić Ci jeszcze lepsze i bardziej kompletne narzędzia, ale to, czego nauczyłeś się do tej pory, będzie nadal działało”. Jednym z powodów, dla których w powłoce PowerShell nie ma cmdletu Map-Drive, jest to, że polecenie net use po prostu działa dobrze, dlatego więc nie używać go dalej?

UWAGA Z przykładu z użyciem polecenia `net use` korzystamy od czasu, kiedy na rynku pojawiła się powłoka PowerShell w wersji v1. Mimo upływu lat jest to nadal dobra historia, ale już powłoka PowerShell v3 udowodniła, że firma Microsoft zaczyna poświęcać czas na opracowywanie „powershellowych” sposobów wykonywania starych zadań. Przykładowo, począwszy od wersji 3 polecenie `New-PSDrive` ma parametr `-Persist`, który w połączeniu z dostawcą `FileSystem` pozwala na tworzenie dysków widocznych w Eksploratorze Windows.

W pewnych przypadkach Microsoft dostarcza obecnie znacznie lepsze narzędzia niż niektóre wciąż powszechnie używane, ale nieuchronnie starzejące się komendy. Na przykład nowy cmdlet `Test-Connection` udostępnia znacznie więcej opcji i bardziej elastyczny wynik niż stare zewnętrzne polecenie `ping`. Ale jeżeli wiesz, jak korzystać z polecenia `ping`, i otrzymujesz to, czego potrzebujesz, to korzystaj z niego — będzie ono nadal dobrze działać z poziomu powłoki PowerShell.

To wszystko, o czym do tej pory powiedzieliśmy, nie zmienia jednak w niczym twardej, szorstkiej prawdy: nie każde polecenie zewnętrzne będzie działało bezbłędnie z poziomu powłoki PowerShell, a przynajmniej nie bez odrobiny nacisków z Twojej strony. Dzieje się tak dlatego, że parser powłoki PowerShell — czyli mechanizm powłoki, który odczytuje to, co wpisałeś, i próbuje zrozumieć, co chcesz zrobić — nie zawsze poprawnie odgaduje Twoje intencje. Krótko mówiąc, czasami zdarza się i tak, że wpisujesz polecenie zewnętrzne, a powłoka PowerShell wyświetla jakieś dziwne błędy i generalnie nie działa tak, jak tego oczekiwałeś.

Przykładowo, może się tak zdarzyć, kiedy polecenie zewnętrzne ma wiele parametrów — właśnie w takich sytuacjach problemy pojawiają się najczęściej. Nie będziemy teraz zbyt mocno wnikać w szczegóły, ale poniżej przedstawiamy sposób na uruchomienie polecenia, który zapewnia, że jego wszystkie parametry będą działać poprawnie:

```
$exe = "C:\Vmware\vcbMounter.exe"
$host = "server"
$user = "joe"
$password = "password"
$machine = "somepc"
$location = "someLocation"
$backupType = "incremental"
& $exe -h $host -u $user -p $password -s "name:$machine" -r $location -t $backupType
```

W podanym przykładzie zakładamy, że masz zewnętrzne polecenie o nazwie `vcbMounter.exe`. (Jest to prawdziwe narzędzie wiersza poleceń, dostarczane z niektórymi produktami wirtualizacyjnymi VMware. Jeżeli nigdy nie używałeś tego narzędzia, to nic nie szkodzi — zdecydowana większość „staroświeckich” narzędzi wiersza poleceń będzie działać w taki sposób, więc nadal jest to bardzo dobry przykład). Nasze polecenie przyjmuje kilka parametrów wywołania, którymi są:

- `-h` — nazwa hosta,
- `-u` — nazwa użytkownika,
- `-p` — hasło,

- -s — nazwa serwera,
- -r — lokalizacja,
- -t — typ kopii zapasowej.

W przykładzie umieściliśmy wszystkie elementy składowe naszego polecenia — ścieżkę i nazwę pliku wykonywalnego oraz wartości wszystkich parametrów — w elementach zastępczych (zmiennych) rozpoczynających się od znaku \$. Zmusza to powłokę PowerShell do traktowania tych wartości jako pojedynczych elementów i zapobiega ich analizowaniu pod kątem występowania jakichkolwiek znaków specjalnych. Następnie używamy operatora wywołania (&), przekazując mu nazwę pliku wykonywalnego oraz wszystkie parametry i ich wartości. Taki sposób będzie działał dla większości narzędzi wiersza poleceń, które mają problemy z normalnym uruchamianiem z poziomu powłoki PowerShell.

Brzmi skomplikowanie? No cóż, mamy też i dobrą wiadomość: w powłoce PowerShell v3 i nowszych wersjach nie musisz już tak dużo robić. Zamiast tego wystarczy dodać dwie kreski i znak procentu przed dowolnym poleceniem zewnętrznym. Kiedy to zrobisz, powłoka PowerShell nawet nie będzie próbowała przeanalizować takiego polecenia i po prostu przekaże go do konsoli cmd.exe. Zasadniczo możesz uruchomić w ten sposób praktycznie wszystko, używając składni dla konsoli cmd.exe i nie martwiąc się o to, jak zareaguje powłoka PowerShell! Aby jednak wszystko było absolutnie jasne — oznacza to również, że w ten sposób nie można przekazywać zmiennych jako wartości parametrów.

A oto krótki przykład wywołania polecenia, które się nie powiedzie:

```
PS C:\> $n = "bits"
PS C:\> C:\windows\system32\sc.exe --% qc $n
[SC] OpenService FAILED 1060:
The specified service does not exist as an installed service.
```

W tym przykładzie próbowaliśmy uruchomić program narzędziowy o nazwie sc.exe, za pomocą którego można zarządzać usługami. Jeżeli jednak jednoznacznie określimy, co chcemy zrobić, PowerShell przekaże wszystkie parametry do podstawowego polecenia, nie próbując nic z nimi zrobić, i wszystko będzie działać tak, jak powinno.

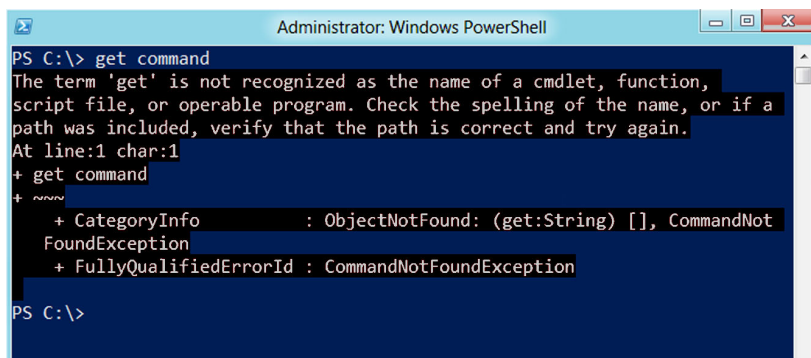
```
PS C:\> C:\windows\system32\sc.exe --% qc bits
[SC] QueryServiceConfig SUCCESS
SERVICE_NAME: bits
        TYPE               : 20  WIN32_SHARE_PROCESS
        START_TYPE          : 2   AUTO_START (DELAYED)
        ERROR_CONTROL       : 1   NORMAL
        BINARY_PATH_NAME    : C:\windows\System32\svchost.exe -k netsvcs
        LOAD_ORDER_GROUP    :
        TAG                 : 0
        DISPLAY_NAME        : Background Intelligent Transfer Service
        DEPENDENCIES        : RpcSs
                           : EventSystem
        SERVICE_START_NAME  : LocalSystem
PS C:\>
```

Na szczęście nie jest to coś, z czego będziesz musiał korzystać zbyt często.

4.8. Jak radzić sobie z błędami

Niestety jest to nieuniknione, że kiedy zaczniesz pracować z powłoką PowerShell, wcześniej czy później zobaczysz na ekranie czerwony komunikat o wystąpieniu błędu — i to nawet jeżeli masz ogromne doświadczenie w pracy z tą powłoką. Każdy może przecież popełnić błąd — zdarza się to nam wszystkim. Ale niech ten czerwony tekst Cię nie stresuje (szczerze mówiąc, kojarzy nam się to trochę z czasami szkolnymi i wypracowaniami upstrzonymi czerwonym długopisem przez naszą polonistkę...).

Jeżeli jednak uda Ci się nie zwracać uwagi na stresujący czerwony kolor, dostrzeżesz, że takie komunikaty o błędach powłoki PowerShell mogą być bardzo pomocne. Przykładowo, komunikat o błędzie pokazany na rysunku 4.4 dokładnie wskazuje miejsce, gdzie powłoka PowerShell wpadła w kłopoty.



```
Administrator: Windows PowerShell
PS C:\> get command
The term 'get' is not recognized as the name of a cmdlet, function,
script file, or operable program. Check the spelling of the name, or if a
path was included, verify that the path is correct and try again.
At line:1 char:1
+ get command
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (get:String) [], CommandNot
  FoundException
+ FullyQualifiedErrorId : CommandNotFoundException

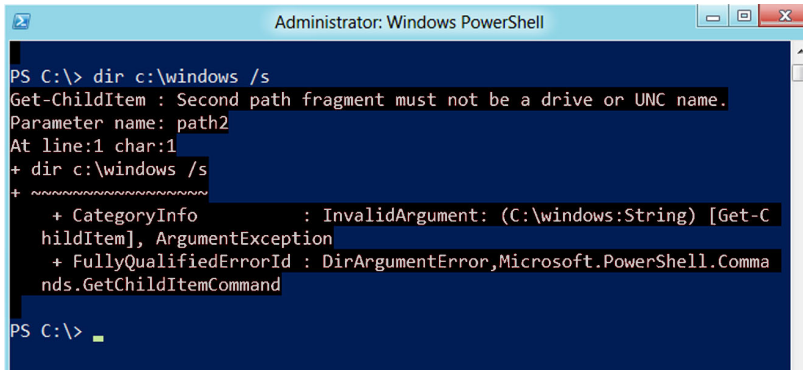
PS C:\>
```

Rysunek 4.4. Przykładowy komunikat o błędzie powłoki PowerShell

Komunikaty o błędach prawie zawsze zawierają numer wiersza i numer znaku w tym wierszu, gdzie powłoka PowerShell wpadła w kłopoty. Na rysunku 4.4 jest to pierwszy znak pierwszego wiersza (At line:1 char: 1) — czyli inaczej mówiąc, problem pojawił się już na początku pierwszego wiersza. Powłoka PowerShell w taki sposób stara się powiedzieć mniej więcej tyle: „Napisałeś coś w tym wierszu, ale nie mam zielonego pojęcia, co to oznacza”. W tym przypadku przyczyna jest dosyć oczywista, bo źle wpisaliśmy nazwę polecenia — zamiast polecenia `Get-Command` wpisaliśmy `get command`. Ups. A co wydarzyło się w sytuacji przedstawionej na rysunku 4.5?

Komunikat o błędzie, przedstawiony na rysunku 4.5: „Drugi fragment ścieżki nie może być dyskiem lub nazwą UNC”, jest nieco mylący. Jaka druga ścieżka? Nie wpisaliśmy przecież żadnej drugiej ścieżki — wprowadziliśmy jedną ścieżkę, `c:\windows`, i parametr wiersza poleceń `/s`, zgadza się?

No cóż, nie do końca. Jednym z najprostszych sposobów rozwiązania tego problemu jest przeczytanie odpowiedniego tematu pomocy i wpisanie polecenia w poprawny sposób. Gdybyśmy wpisali polecenie `Get-ChildItem -path C:\Windows`, zrozumielibyśmy, że dodanie parametru `/s` nie jest poprawną składnią i zamiast tego powinien zostać użyty parametr `-recurse`. Czasami komunikat o błędzie może nie wydawać się pomocny, ale kiedy masz wrażenie, że Ty i PowerShell mówicie różnymi językami, to zazwyczaj jednak



```
Administrator: Windows PowerShell
PS C:\> dir c:\windows /s
Get-ChildItem : Second path fragment must not be a drive or UNC name.
Parameter name: path2
At line:1 char:1
+ dir c:\windows /s
+ ~~~~~
+ CategoryInfo          : InvalidArgument: (C:\windows:String) [Get-ChildItem], ArgumentException
+ FullyQualifiedErrorId : DirArgumentError,Microsoft.PowerShell.Commands.GetChildItemCommand

PS C:\> 
```

Rysunek 4.5. Co to jest „drugi fragment ścieżki” (ang. second path fragment)?

to Ty jesteś w błędzie. Powłoka PowerShell oczywiście nie zmieni swojego języka komunikatów, więc prawdopodobnie najlepszym i często najszybszym sposobem rozwiązania problemu będzie skonsultowanie pomocy i napisanie całego polecenia, parametrów i ich wartości od nowa. Pamiętaj, że istnieje polecenie `Show-Command`, które zawsze pomoże Ci znaleźć właściwą składnię.

4.9. Najczęściej spotykane problemy

Za każdym razem, kiedy wydaje się to właściwe, podsumowujemy rozdział krótkim podrozdziałem omawiającym niektóre z typowych problemów i błędów, z którymi często borykają się studenci podczas naszych szkoleń. Chodzi nam o to, abyś dowiedział się, co najczęściej sprawia problemy innym administratorom i abyś podczas pracy z powłoką PowerShell mógł dzięki temu uniknąć takich sytuacji w przyszłości lub przynajmniej szybko znaleźć dla nich rozwiązanie.

4.9.1. Wpisywanie nazw poleceń cmdlet

Pierwszym z często spotykanych problemów jest niepoprawne wpisywanie nazw poleceń *cmdlet*. Jak już wspominaliśmy, nazwy takich poleceń składają się zawsze z pary czasownik-rzeczownik, oddzielonych od siebie znakiem myślnika, na przykład `Get-Content`. Poniżej przedstawiamy kilka przykładów błędnie wpisywanych nazw poleceń, które choć podobne do oryginału, nie będą oczywiście działać:

- `Get Content,`
- `GetContent,`
- `Get=Content,`
- `Get_Content.`

Główną przyczyną powstawania takich problemów są literówki (na przykład użycie `=` zamiast `-`) oraz pomijanie znaku myślnika. W praktyce wszyscy wymawiamy nazwę takiego polecenia jako *Get Content*, werbalnie pomijając myślnik, którego jednak w wierszu poleceń nie możemy pominąć.

4.9.2. Wpisywanie parametrów

Nazwy i wartości parametrów poleceń również muszą być wpisywane konsekwentnie i zgodnie ze składnią. Parametr, który nie przyjmuje żadnej wartości, na przykład `-recurse`, również musi być poprzedzony znakiem myślnika. Nazwa polecenia *cmdlet* i nazwy poszczególnych parametrów oraz ich wartości muszą być od siebie oddzielone spacjami. Poniżej podajemy dwa przykłady poprawnie zapisanych poleceń:

- `Dir -rec` (skrótowa nazwa parametru jest zapisana poprawnie),
- `New-PSDrive -name DEMO -psprovider FileSystem -root \\Server\Share`.

Przykłady poleceń zamieszczone poniżej nie są jednak poprawne:

- `Dir-rec` (brak spacji między aliasem polecenia a nazwą parametru),
- `New-PSDrive -nameDEMO` (brak spacji między nazwą parametru a jego wartością),
- `New-PSDrive -name DEMO-psprovider FileSystem` (brak spacji między wartością pierwszego parametru a nazwą drugiego parametru).

W normalnych warunkach powłoka PowerShell nie jest wybredna w stosunku do małych i wielkich liter, co oznacza, że polecenia `dir` oraz `DIR` oznaczają to samo, podobnie jak nazwy parametrów `-RECURSE`, `-recurse` czy `-Recurse`. Nie zmienia to jednak w niczym faktu, że w kwestii składni polecenia i odpowiedniego stosowania spacji i myślników powłoka PowerShell z pewnością jest bardzo wybredna.

4.10. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie komputer z zainstalowanym systemem Windows 8, Windows Server 2012 lub nowszym oraz powłoką PowerShell w wersji 3 lub nowszej.

Korzystając z tego, czego dowiedziałeś się w tym oraz w poprzednim rozdziale, dotyczącym korzystania z systemu pomocy, wykonaj następujące zadania z użyciem powłoki Windows PowerShell:

1. Wyświetl listę uruchomionych procesów.
2. Wyświetl 100 najnowszych wpisów z dziennika zdarzeń Application (nie używaj do tego celu polecenia `Get-WinEvent`. Pokazaliśmy Ci inne polecenie, przy użyciu którego możesz wykonać to zadanie) — ćwiczenie dotyczy tylko systemów operacyjnych Windows.
3. Wyświetl listę wszystkich poleceń *cmdlet* (to może być dosyć trudne — pokazaliśmy Ci już polecenie `Get-Command`, ale będziesz musiał dokładnie zapoznać się z treścią pomocy, aby dowiedzieć się, jak zawęzić wyniki działania polecenia do listy, o którą prosiłeś).
4. Wyświetl listę wszystkich aliasów.

5. Utwórz nowy alias o nazwie np, tak aby przy jego użyciu można było uruchomić program Notatnik z poziomu wiersza poleceń powłoki PowerShell. Dotyczy tylko systemu Windows, chyba że w swoim systemie Linux zainstalowałeś pakiet Wine.
6. Wyświetl listę usług o nazwach rozpoczynających się od litery M. Zanim przystąpisz do pisania polecenia, zapoznaj się z treścią pomocy dotyczącą niezbędnego polecenia. Nie zapominaj, że gwiazdka (*) jest niemal uniwersalnym symbolem wieloznacznym w powłoce PowerShell. Pamiętaj, że ćwiczenie to jest przeznaczone tylko dla systemu operacyjnego Windows.
7. Wyświetl listę wszystkich reguł zapory sieciowej systemu Windows (ang. *Windows Firewall*). Aby odkryć niezbędne polecenie, będziesz musiał skorzystać z systemu pomocy lub użyć polecenia *Get-Command*. Podobnie jak poprzednio, ćwiczenie to przeznaczone jest wyłącznie dla systemu Windows.
8. Wyświetl listę reguł zapory sieciowej systemu Windows dla ruchu przychodzącego. Możesz użyć tego samego polecenia *cmdlet* co w poprzednim zadaniu, ale musisz dokładnie zapoznać się z treścią jego tematu pomocy, aby znaleźć niezbędny parametr i jego dopuszczalne wartości. I znów, ćwiczenie to przeznaczone jest wyłącznie dla systemu Windows.

Mamy nadzieję, że przedstawione wyżej zadania wydają Ci się proste. Jeśli tak, to doskonale. Podczas ich wykonywania możesz wykorzystać nabyte wcześniej umiejętności pracy z powłoką PowerShell i nauczyć się wykonywać nowe zadania. Jeżeli jesteś nowicjuszem w pracy z wierszem poleceń powłoki, to przedstawione wyżej zadania z pewnością będą dobrym wprowadzeniem do tego, co będziesz robić w dalszej części tej książki.

5

Praca z dostawcami

Jednym z trudniejszych aspektów pracy z powłoką PowerShell jest korzystanie z **dostawców** (ang. *providers*). Ostrzegamy, że niektóre z zagadnień omawianych w tym rozdziale mogą wydawać Ci się nieco nadmiarowe. Zakładamy na przykład, że znasz system plików Windows i prawdopodobnie znasz wszystkie polecenia potrzebne do zarządzania systemem plików z poziomu wiersza poleceń. Chodzi nam jednak o to, że chcieliśmy pokazać, w jaki sposób możesz wykorzystać swoją znajomość systemu plików, aby tym łatwiej zrozumieć koncepcję dostawców. Powinieneś również pamiętać, że powłoka PowerShell to nie jest konsola `cmd.exe`. W tym rozdziale możesz zobaczyć pewne rzeczy, które na pozór będą wyglądały znajomo, ale zapewniamy Cię, że robią coś zupełnie innego niż to, do czego jesteś przyzwyczajony.

5.1. Czym są dostawcy?

Dostawca powłoki PowerShell lub inaczej `PSProvider` jest adapterem, który został zaprojektowany do operowania na określonych magazynach danych i sprawiania, aby wyglądały one jak dyski. Listę zainstalowanych dostawców powłoki PowerShell możesz wyświetlić bezpośrednio z poziomu wiersza polecenia:

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

Name	Capabilities	Drives
----	-----	-----
Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess, Credentials	{C, A, D}
Function	ShouldProcess	{Function}
Registry	ShouldProcess, Transactions	{HKLM, HKCU}
Variable	ShouldProcess	{Variable}

Oczywiście do powłoki PowerShell mogą być dodawani nowi dostawcy, zazwyczaj dzieje się tak wraz z zainstalowaniem nowego modułu lub przystawki, które są dwoma sposobami rozszerzania funkcjonalności powłoki PowerShell (o rozszerzeniach powłoki PowerShell będziemy mówić w rozdziale 7.). Czasami włączenie pewnych funkcji powłoki PowerShell może spowodować utworzenie nowego dostawcy PSProvider. Na przykład po włączeniu dostępu zdalnego (o którym będziemy mówić w rozdziale 13.) na liście pojawi się nowy dostawca o nazwie WSMAN, tak jak to zostało pokazane poniżej:

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

Name	Capabilities	Drives
----	-----	-----
Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess, Credentials	{C, A, D}
Function	ShouldProcess	{Function}
Registry	ShouldProcess, Transactions	{HKLM, HKCU}
Variable	ShouldProcess	{Variable}
WSMan	Credentials	{WSMan}

Zauważ, że poszczególni dostawcy mają różne zestawy możliwości (kolumna Capabilities). Jest to bardzo ważne, ponieważ bezpośrednio wpływa na sposoby korzystania z danego dostawcy. Oto niektóre typowe możliwości:

- ShouldProcess — dostawca obsługuje parametry -WhatIf oraz -Confirm, które umożliwiają „przetestowanie” określonych działań przed ich rzeczywistym wykonaniem.
- Filter — dostawca obsługuje parametr -Filter w cmdletach, które operują na treściach dostarczanych przez dostawców.
- Credentials — dostawca umożliwia określenie alternatywnych poświadczeń podczas łączenia się z magazynami danych; do wykonania takiej operacji używany jest parametr -credential.
- Transactions — dostawca zapewnia obsługę transakcji; dzięki temu dostawcy możemy przeprowadzić serię określonych operacji, a następnie wyczołnąć je lub zatwierdzić w jednym bloku transakcji.

Z użyciem dostawców możemy również tworzyć dyski PSDrive, które wykorzystują takiego dostawcę do połączenia z magazynem danych. Wygląda to jak mapowanie dysków, które są później widoczne w Eksploratorze Windows, z tym że dyski PSDrive, dzięki dostawcom, można łączyć nie tylko z systemami plików, ale również z innymi magazynami danych. Aby wyświetlić listę aktualnie podłączonych dysków, możesz użyć polecenia przedstawionego poniżej:

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

Name	Used (GB)	Free (GB)	Provider	Root
----	-----	-----	-----	-----
A			FileSystem	A:\
Alias			Alias	
C	9.88	54.12	FileSystem	C:\
D	3.34		FileSystem	D:\

Env	Environment	
Function	Function	
HKCU	Registry	HKEY_CURRENT_USER
HKLM	Registry	HKEY_LOCAL_MACHINE
Variable	Variable	

W wynikach działania powyższego polecenia możemy zobaczyć, że mamy trzy dyski korzystające z dostawcy FileSystem, dwa korzystające z dostawcy Registry i tak dalej. Dostawcy PSProvider powłoki PowerShell dostosowują magazyny danych, które są następnie udostępniane w postaci dysków PSDrive, dzięki czemu użytkownik może używać odpowiedniego zestawu poleceń *cmdlet* do przeglądania danych udostępnianych na poszczególnych dyskach PSDrive i operowania na nich. W większości przypadków polecenia *cmdlet* używane do pracy z dyskami PSDrive zawierają w nazwie rzeczownik Item:

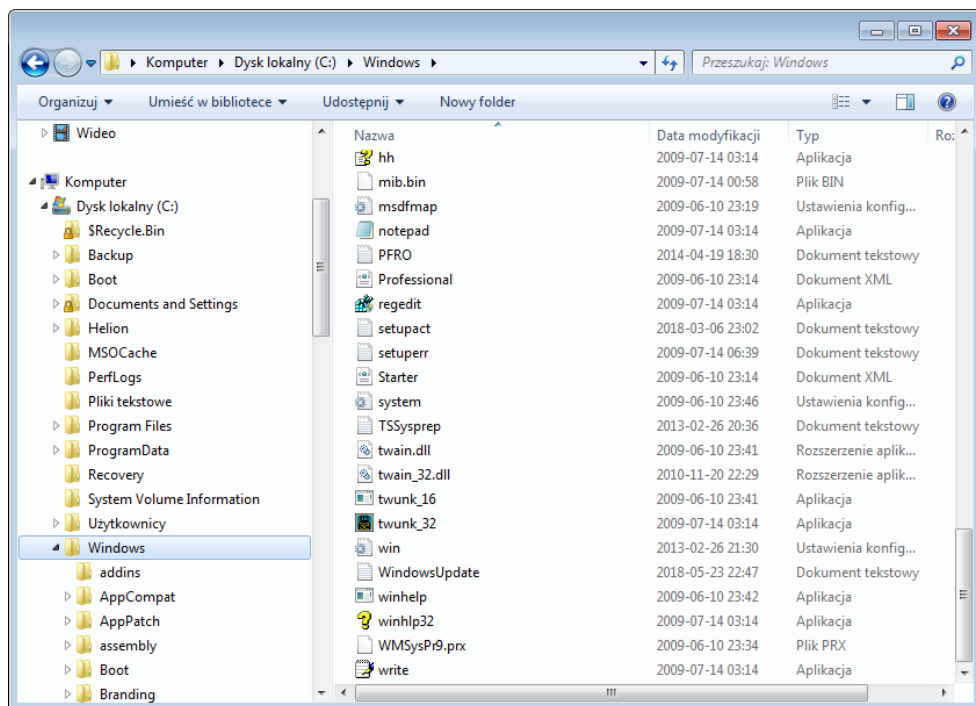
```
PS C:\> get-command -noun *item*
Capability      Name
-----
Cmdlet          Clear-Item
Cmdlet          Clear-ItemProperty
Cmdlet          Copy-Item
Cmdlet          Copy-ItemProperty
Cmdlet          Get-ChildItem
Cmdlet          Get-Item
Cmdlet          Get-ItemProperty
Cmdlet          Invoke-Item
Cmdlet          Move-Item
Cmdlet          Move-ItemProperty
Cmdlet          New-Item
Cmdlet          New-ItemProperty
Cmdlet          Remove-Item
Cmdlet          Remove-ItemProperty
Cmdlet          Rename-Item
Cmdlet          Rename-ItemProperty
Cmdlet          Set-Item
Cmdlet          Set-ItemProperty
```

Przedstawionych wyżej cmdletów i ich aliasów będziemy używać do pracy z dostawcami w naszym systemie. Naszą przygodę z dostawcami zaczniemy od prawdopodobnie najbardziej znanego i powszechnie używanego dostawcy PSProvider, czyli od systemu plików — FileSystem.

5.2. Jak zorganizowany jest system plików

System plików Windows (podobnie jak systemy plików w systemach macOS i Linux) jest zorganizowany w oparciu o trzy główne typy obiektów: dyski, foldery i pliki. **Dyski** (ang. *drives*), czyli obiekty najwyższego poziomu, zawierają zarówno foldery, jak i pliki. **Foldery** (ang. *folders*) są także rodzajem kontenera, który może zawierać tak pliki, jak i inne foldery. **Pliki** (ang. *files*) nie są rodzajem kontenera; w systemie plików odgrywają rolę swego rodzaju obiektu końcowego.

Większość użytkowników najczęściej poznaje system plików za pomocą Eksploratora Windows (zobacz rysunek 5.1), gdzie hierarchia dysków, folderów i plików jest oczywista i widoczna na pierwszy rzut oka.



Rysunek 5.1. Przeglądanie plików, folderów i dysków w oknie Eksploratora Windows

Terminologia używana w powłoce PowerShell różni się nieco od terminologii używanej w systemie plików. Ze względu na to, że dyski PSDrive nie muszą wskazywać wyłącznie na system plików — na przykład za pomocą dysków PSDrive można zmapować rejestr systemu Windows, który oczywiście nie jest systemem plików — w powłoce PowerShell nie używa się terminów takich jak *plik* i *folder*. Zamiast tego odwołujemy się do takich obiektów za pomocą bardziej ogólnego określenia *element* (ang. *item*). Zarówno pliki, jak i foldery są uważane za elementy, chociaż są to oczywiście elementy różnego typu. Z tego właśnie względu w wyświetlanych wcześniej nazwach cmdletów znajdujemy rzeczownik *Item* (element).

Elementy mogą i często mają swoje właściwości. Przykładowo, element reprezentujący plik może mieć wiele różnych właściwości, takich jak data i czas ostatniego zapisu, atrybut określający, czy element jest przeznaczony tylko do odczytu. Niektóre elementy, na przykład foldery, mogą zawierać **elementy podrzędne** (ang. *child items*), które są innymi elementami zawartymi w elemencie nadrzędnym. Znajomość tych zagadnień pomoże Ci zrozumieć czasowniki i rzeczowniki w nazwach poleceń, których listę pokazaliśmy wcześniej:

- Czasowniki takie jak Clear (wyczyść), Copy (kopiuj), Get (pobierz), Move (przenieś), New (nowy), Remove (usuń), Rename (zmień nazwę) i Set (ustaw) mogą odnosić się tak do elementów (na przykład plików i folderów), jak i do ich właściwości (na przykład do daty ostatniego zapisu, atrybutu określającego, czy jest to element tylko do odczytu).
- Rzeczownik Item (element) odnosi się do pojedynczych obiektów, takich jak pliki i foldery.
- Rzeczownik ItemProperty (właściwość elementu) odnosi się do atrybutów przedmiotu, takich jak czas utworzenia, rozmiar czy przeznaczenie tylko do odczytu.
- Rzeczownik ChildItem odnosi się do elementów podrzędnych (takich jak pliki i podfoldery) zawartych w elemencie nadrzędnym (na przykład folderze).

Pamiętaj, że nazwy takich poleceń *cmdlet* są celowo bardzo ogólne, ponieważ są przeznaczone do pracy z wieloma różnymi magazynami danych. Niektóre funkcje *cmdlet*ów nie mają zastosowania w pewnych określonych sytuacjach. Przykładowo, ze względu na fakt, że dostawca FileSystem nie obsługuje transakcji (nie posiada zdolności Transactions), żaden z parametrów *-UseTransaction* poleceń *cmdlet* nie będzie działał z dyskami reprezentującymi systemy plików. Podobnie, ponieważ dostawca rejestru (Registry) nie obsługuje zdolności Filter, parametry *-Filter* poleceń *cmdlet* nie będą działały z dyskami reprezentującymi zawartość rejestru.

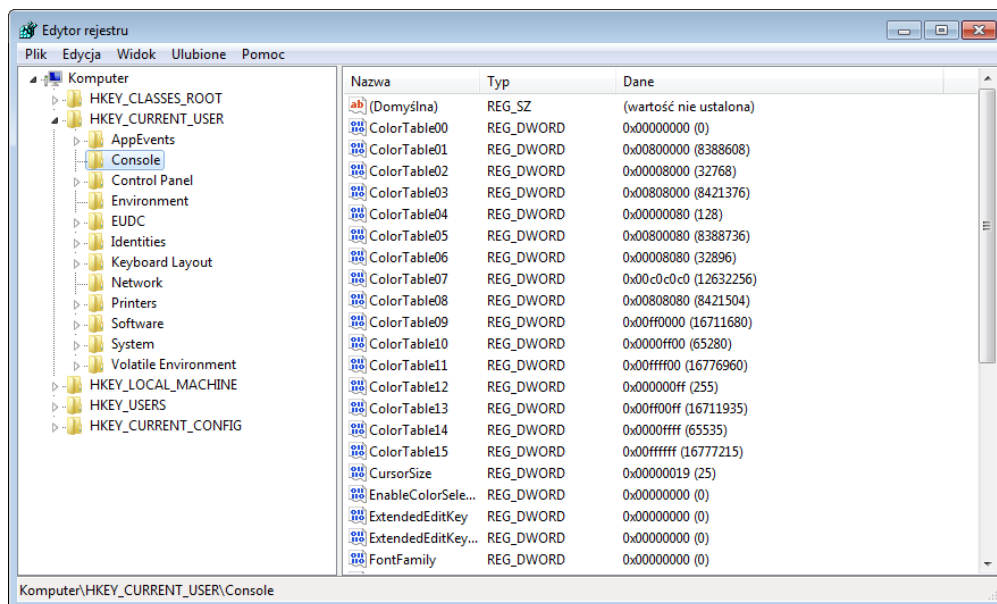
Niektórzy dostawcy PSProviders nie obsługują właściwości elementów. Przykładowo, dostawca Environment jest używany do utworzenia dysku ENV: dostępnego z poziomu powłoki PowerShell. Taki dysk zapewnia dostęp do zmiennych środowiskowych systemu Windows, ale jak pokazuje poniższy przykład, nie posiada żadnych właściwości:

```
PS C:\> Get-ItemProperty -Path Env:\PSModulePath
Get-ItemProperty : Cannot use interface. The IPropertyCmdletProvider interface is not
↳supported by this provider.
At line:1 char:1
+ Get-ItemProperty -Path Env:\PSModulePath
+ ~~~~~
+ CategoryInfo          : NotImplemented: (:) [Get-ItemProperty], PSNotSupportedException
+ FullyQualifiedErrorId : NotSupported,Microsoft.PowerShell.Commands.GetItemPropertyCommand
```

To, że nie każdy dostawca PSProvider jest taki sam, prawdopodobnie sprawia największą trudność początkującym użytkownikom powłoki PowerShell. Musisz zawsze pamiętać o tym, do których elementów daje dostęp dany dostawca, i zrozumieć, że nawet jeżeli polecenie *cmdlet* ma określone możliwości, to jeszcze nie oznacza, że dostawca, z którym pracujesz, będzie obsługiwał taką operację.

5.3. W czym system plików jest podobny do innych magazynów danych?

System plików jest modelem dla innych rodzajów magazynów danych. Przykładowo, na rysunku 5.2 pokazano edytor rejestru systemu Windows.



Rysunek 5.2. Rejestr systemu Windows oraz system plików mają podobną, hierarchiczną strukturę

Rejestr systemu Windows jest podobny do systemu plików z folderami (klucze rejestru), plikami (wartości kluczy rejestru) i tak dalej. To duże podobieństwo sprawia, że system plików jest doskonałym modelem magazynu danych, dlatego PowerShell traktuje magazyny danych jako dyski (ang. *drives*) przechowujące elementy (ang. *items*), które posiadają swoje właściwości (ang. *item properties*). Ale podobieństwo na tym się kończy: kiedy zagłębisz się w szczegóły, zauważysz, że poszczególne magazyny danych mogą się znacząco od siebie różnić. Z tego właśnie względu polecenia *cmdlet* mają tak szeroki zakres możliwości i z tego samego powodu nie każda ich funkcjonalność będzie działać ze wszystkimi magazynami danych.

5.4. Poruszanie się w systemie plików

Kolejnym poleceniem *cmdlet*, które musisz znać podczas pracy z dostawcami, jest *Set-Location*. Używa się go z poziomu wiersza poleceń powłoki do zmiany bieżącej lokalizacji na inny element typu kontener, taki jak folder:

```
PS C:\> Set-Location -Path C:\Windows
PS C:\Windows>
```

Prawdopodobnie znacznie lepiej znasz alias tego polecenia, *cd*, który odpowiada poleceniu *cd* (ang. *change directory*; zmień katalog) konsoli *cmd.exe*:

```
PS C:\Windows> cd 'C:\Program Files'
PS C:\Program Files>
```

W przedstawionym przykładzie używamy aliasu polecenia *cmdlet* i przekazujemy do niego żadaną ścieżkę docelową jako parametr pozycyjny.

Dyski w systemach operacyjnych innych niż Windows

Systemy macOS i Linux nie używają *dysków* do odwoływania się do poszczególnych napędów pamięci masowej. Zamiast tego cały system operacyjny posiada jeden katalog główny (*root*), reprezentowany przez prawy ukośnik (w PowerShellu akceptowany jest również lewy ukośnik). Powłoka PowerShell nadal jednak udostępnia użytkownikowi dyski *PSDrives* w systemach operacyjnych innych niż Windows. Spróbuj uruchomić polecenie *Get-PSDrive*, aby zobaczyć, jakie dyski są dostępne w Twoim systemie.

Jednym z trudniejszych zadań w powłoce PowerShell jest tworzenie nowych elementów. Na przykład jak możesz utworzyć nowy katalog? Spróbuj użyć chociażby polecenia *New-Item*, a na ekranie pojawi się nieoczekiwany monit:

```
PS C:\users\donjones\Documents> new-item testFolder
Type:
```

Pamiętaj, że polecenie *cmdlet* *New-Item* jest bardzo uniwersalne i „nie wie” jeszcze, że chcesz utworzyć folder. Za pomocą tego polecenia możesz tworzyć foldery, pliki, klucze rejestru i wiele innych elementów, ale zawsze musisz podać typ elementu, który chcesz utworzyć:

```
PS C:\users\donjones\Documents> new-item testFolder
Type: directory
Directory: C:\users\donjones\Documents
Mode                LastWriteTime         Length      Name
----                -
d-----          3/29/2016 10:43 AM                testFolder
```

Powłoka PowerShell posiada polecenie *mkdir*, które wielu użytkowników uważa za alias polecenia *New-Item*, ale powinieneś pamiętać, że polecenie *mkdir* nie wymaga podawania typu tworzonego elementu:

```
PS C:\users\donjones\Documents> mkdir test2
Directory: C:\users\donjones\Documents
Mode                LastWriteTime         Length      Name
----                -
d-----          3/29/2016 10:44 AM                test2
```

Co nam to daje? Okazuje się, że *mkdir* jest funkcją, a nie aliasem. Wewnętrznie nadal używa *cmdletu* *New-Item*, ale funkcja od razu dodaje parametr *-Type Directory*, dzięki czemu polecenie *mkdir* zachowuje się bardziej jak jego poprzednik z konsoli *cmd.exe*. Pamiętanie o tym i wielu innych drobiazgach znacząco może Ci ułatwić pracę z dostawcami, ponieważ musisz sobie zdawać sprawę z tego, że nie każdy dostawca jest taki sam, a poszczególne *cmdlety* są dosyć uniwersalne i czasami potrzebują doprecyzowania nieco większej ilości informacji, niż mogłoby się to na pierwszy rzut oka wydawać.

5.5. Używanie symboli wieloznacznych i dokładnych ścieżek

Większość poleceń *cmdlet* typu *Item* posiada parametr *-Path*, który domyślnie akceptuje symbole wieloznaczne. Przykładowo, po wyświetleniu pełnej treści pomocy dotyczącej polecenia *Get-ChildItem* znajdziemy tam następującą informację:

```
-Path <String[]>
  Specifies a path to one or more locations. Wildcards are permitted. The default location
  ↪ is the current directory (.).
  Required? false
  Position? 1
  Default value Current directory
  Accept pipeline input? true (ByValue, ByPropertyName)
  Accept wildcard characters? True
```

Symbol *** reprezentuje brak lub dowolną ilość innych znaków, podczas gdy symbol wieloznaczny *?* reprezentuje jeden dowolny znak. Nie mamy żadnych wątpliwości, że już wielokrotnie używałeś takich symboli, najprawdopodobniej podczas korzystania z aliasu *Dir* polecenia *Get-ChildItem*:

```
PS C:\Windows> dir *.exe
Directory: C:\Windows

Mode                LastWriteTime         Length      Name
----                -
-a---      2/17/2012 9:17 PM          75264    bfsvc.exe
-a---      2/17/2012 11:21 PM        2355208    explorer.exe
-a---      2/17/2012 9:18 PM        899072    HelpPane.exe
-a---      2/17/2012 9:18 PM         16896    hh.exe
-a---      2/17/2012 9:18 PM        233472    notepad.exe
-a---      2/17/2012 9:18 PM        159744    regedit.exe
-a---      2/17/2012 9:18 PM        125440    splwow64.exe
-a---      2/17/2012 10:09 PM          9728    winhlp32.exe
-a---      2/17/2012 9:18 PM         10240    write.exe
```

Symbol wieloznaczny wymienione w poprzednim przykładzie są tymi samymi znakami, z których zawsze korzystały systemy plików firmy Microsoft począwszy już od starego dobrego systemu MS-DOS. Ponieważ znaki takie odgrywają specjalną rolę, to ich używanie nie jest dozwolone w nazwach plików i folderów. Jednak w przypadku powłoki PowerShell system plików nie jest jedynym dostępnym rodzajem pamięci masowej. W większości innych magazynów danych znaki *** i *?* są całkowicie poprawnymi znakami dla nazw elementów. Na przykład w rejestrze systemu Windows znajdziesz całkiem sporo kluczy o nazwach, które zawierają znaki *?*. Pojawia się zatem problem: kiedy używasz znaku *** lub *?* w ścieżce, czy powłoka PowerShell powinna traktować je jako symbole wieloznaczne, czy dosłownie, jak zwykłe znaki? Jeśli szukasz elementów o nazwie *Windows?*, to czy chcesz znaleźć elementy o nazwie *Windows?*, czy też chcesz, aby znak *?* był traktowany jako symbol wieloznaczny, pozwalający na znalezienie elementów takich jak *Windows7* i *Windows8*?

Rozwiązaniem stosowanym w powłoce PowerShell jest użycie dodatkowego parametru, *-LiteralPath*, który nie akceptuje symboli wieloznacznych:

Specifies a path to one or more locations. Unlike the Path parameter, the value of the `LiteralPath` parameter is used exactly as it is typed. **No characters are interpreted as wildcards.** If the path includes escape characters, enclose it in single quotation marks. Single quotation marks tell Windows PowerShell not to interpret any characters as escape sequences.

Required?	true
Position?	named
Default value	
Accept pipeline input?	true (ByValue, ByPropertyName)
Accept wildcard characters?	False

5.6. Praca z innymi dostawcami

Rozpocznij od przejścia do klucza HKEY_CURRENT_USER, udostępnianego za pośrednictwem dysku HKCU:

Następnie przejdź do odpowiedniej części rejestru:

[illegible]


```

Roaming
Shell
TabletPC
Windows Error Reporting
AlwaysHibernateThumbnails : 0
ColorizationColor : 3226847725
ColorizationColorBalance : 72
ColorizationAfterglow : 3226847725
ColorizationAfterglowBalance : 0
ColorizationBlurBalance : 28
ColorizationGlassReflectionIntensity : 50
ColorizationOpaqueBlend : 0
ColorizationGlassAttribute : 0
Disabled : 0
MaxQueueCount : 50
DisableQueue : 0
LoggingDisabled : 0
DontSendAdditionalData : 0
ForceQueue : 0
DontShowUI : 0
ConfigureArchive : 1
MaxArchiveCount : 500
DisableArchive : 0
LastQueuePesterTime : 129773462733828600

```

Zwróć uwagę na klucz EnableAeroPeek. Zmienimy jego wartość na 0:

```
PS HKCU:\software\microsoft\Windows> Set-ItemProperty -Path dwm -PSProperty EnableAeroPeek -Value 0
```

Zamiast parametru -PSProperty możemy tutaj również użyć parametru -Name. Sprawdzimy teraz ponownie zawartość tej gałęzi rejestru, żeby upewnić się, że nasza zmiana została wprowadzona poprawnie:

```

PS HKCU:\software\microsoft\Windows> Get-ChildItem
Hive: HKEY_CURRENT_USER\software\microsoft\Windows
Name                                     Property
----                                     -
CurrentVersion
DWM
Composition : 1
EnableAeroPeek : 0
AlwaysHibernateThumbnails : 0
ColorizationColor : 3226847725
ColorizationColorBalance : 72
ColorizationAfterglow : 3226847725
ColorizationAfterglowBalance : 0
ColorizationBlurBalance : 28
ColorizationGlassReflectionIntensity : 50
ColorizationOpaqueBlend : 0
ColorizationGlassAttribute : 0
Roaming
Shell
TabletPC
Windows Error Reporting
Disabled : 0
MaxQueueCount : 50
DisableQueue : 0
LoggingDisabled : 0

```

```

DontSendAdditionalData : 0
ForceQueue : 0
DontShowUI : 0
ConfigureArchive : 1
MaxArchiveCount : 500
DisableArchive : 0
LastQueuePesterTime : 129773462733828600

```

Zadanie wykonane! Korzystając z przedstawionej techniki, możesz pracować z dowolnym dostawcą, którego spotkasz na swojej drodze.

5.7. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej. Niniejsze ćwiczenia przeznaczone są tylko dla systemów operacyjnych Windows.

Z poziomu wiersza polecenia powłoki PowerShell wykonaj następujące zadania:

1. W rejestrze przejdź do klucza HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer. Zlokalizuj klucz Advanced i ustaw jego właściwość DontPrettyPath na wartość 1.
2. Utwórz nowy katalog o nazwie C:\Labs.
3. Utwórz plik C:\Labs\Test.txt o zerowym rozmiarze (pusty plik). Użyj do tego celu polecenia New-Item.
4. Czy możliwe jest użycie polecenia Set-Item do zmiany zawartości pliku C:\Labs\Test.txt na TESTOWANIE? Czy podczas próby wykonania tego polecenia otrzymasz błąd? Jeżeli pojawi się błąd, to dlaczego?
5. Za pomocą dostawcy Environment wyświetl wartość zmiennej systemowej %TEMP%.
6. Jakie są różnice między parametrami -Filter, -Include i -Exclude polecenia Get-ChildItem?

5.8. Co dalej?

Szybko przekonasz się, że różne pakiety oprogramowania, takie jak Internet Information Server (IIS), SQL Server, a nawet Active Directory, również posiadają swoich dostawców. W większości przypadków deweloperzy tych produktów zdecydowali się na użycie dostawców, ponieważ funkcjonalność tych pakietów oprogramowania można rozszerzać w dynamiczny sposób. Deweloperzy nie mogą przewidzieć z góry, jakie funkcje zostaną zainstalowane w tych produktach w przyszłości, zatem nie mogą również przygotować dla nich statycznego zestawu poleceń. Użycie dostawców umożliwia programistom eksponowanie dynamicznych struktur danych w spójny sposób, co pozwala zespołom pracującym z nowymi wersjami oprogramowania (szczególnie dotyczy to IIS i SQL Server) na efektywne stosowanie kombinacji poleceń *cmdlet* i dostawców.

Jeśli masz dostęp do tych produktów (w przypadku serwera IIS wymagana jest wersja 7.5 lub nowsza, w przypadku serwera SQL Server zalecamy używanie wersji

SQL Server 2012 lub nowszej), zdecydowanie powinieneś spędzić trochę czasu na badaniu modeli tych dostawców. Zobacz, jak zespoły produktowe uporządkowały strukturę swoich „dysków” i jak możesz użyć poleceń *cmdlet* omawianych w tym rozdziale do przeglądania i zmieniania ustawień konfiguracyjnych oraz innych właściwości.

5.9. Odpowiedzi

1. Wykonaj podaną niżej sekwencję poleceń:
`cd HKCU:\software\microsoft\Windows\currentversion\explorer`
`cd advanced`
`Set-ItemProperty -Path . -Name DontPrettyPath -Value 1`
2. Możesz użyć funkcji `mkdir`:
`mkdir c:\ labs`
lub polecenia `New-Item`:
`New-Item -Path C:\Labs -ItemType Directory`
3. `New-Item -path c:\labs -Name test.txt -ItemType file`
4. Dostawca systemu plików nie obsługuje tej akcji.
5. Możesz to zrobić za pomocą dowolnego z tych poleceń:
`Get-item env:temp`
`Dir env:temp`
6. Parametry `-Include` i `-Exclude` należy stosować z opcją `-Recurse` lub w przypadku wysyłania zapytań do kontenera. Parametr `-Filter` korzysta z funkcji filtrowania, która nie jest obsługiwana przez wszystkich dostawców. Na przykład pracując z systemem plików, możesz użyć polecenia `DIR -filter`, ale już w przypadku dostawcy rejestru takie polecenie nie będzie działać — choć dla dostawcy rejestru bardzo podobne rezultaty filtrowania możesz uzyskać za pomocą polecenia `DIR ↪-Include`.

Potoki — łączenie poleceń

W rozdziale 4. dowiedziałeś się, że uruchamianie poleceń w PowerShellu przebiega tak samo jak wykonywanie poleceń w dowolnej innej powłoce: wpisujesz nazwę polecenia, podajesz parametry i naciskasz klawisz *Enter*. Tym, co sprawia, że powłoka PowerShell jest wyjątkowa, nie jest uruchamianie polecenia, ale sposób, w jaki pozwala ona na łączenie wielu poleceń w potężne, jednowierszowe sekwencje.

6.1. Łączenie poleceń ze sobą — mniej pracy dla Ciebie

W powłoce PowerShell poszczególne polecenia łączymy ze sobą za pomocą **potoku** (ang. *pipeline*). Potoki zapewniają efektywny sposób przekazywania wyników działania jednego polecenia na wejście innego polecenia, dzięki czemu to drugie polecenie otrzymuje zestaw danych wejściowych, na których może pracować.

Widzieliście już takie rozwiązania w poleceniach takich jak `Dir | More`. W takim przykładzie przekazujemy wyniki działania polecenia `Dir` na wejście polecenia `More`, które pobiera wyświetlaną przez `Dir` listę katalogów i wyświetla ją po jednym ekranie na raz. Powłoka PowerShell przyjmuje tę samą koncepcję potoków i znacząco zwiększa jej funkcjonalność.

Korzystanie z potoków w powłoce PowerShell może początkowo wyglądać podobnie jak w przypadku powłok systemów UNIX i Linux. Nie daj się jednak zwieść. Jak się niebawem sam przekonasz w najbliższych kilku rozdziałach, implementacja potoków w powłoce PowerShell jest znacznie bogatsza i bardziej nowoczesna.

6.2. Eksportowanie wyników działania polecenia do pliku CSV lub XML

Uruchom jakieś proste polecenie. Oto kilka propozycji:

- Get-Process (lub gps; to powinno działać również w systemach macOS i Linux);
- Get-Service (lub gsv; tylko w systemie Windows);
- Get-EventLog Security -newest 100 (tylko w systemie Windows).

Wybraliśmy te polecenia, ponieważ są to łatwe, proste w użyciu cmdlety. W nawiasach podaliśmy również aliasy dla poleceń Get-Process i Get-Service. W przypadku polecenia Get-EventLog, które działa tylko w systemie Windows, określiliśmy jego parametr obowiązkowy oraz parametr -newest (dzięki któremu wykonanie polecenia nie zajmie zbyt dużo czasu).

ZRÓB TO SAM Oczywiście możesz niemal dowolnie wybrać polecenia, z którymi chcesz pracować. My w kolejnych przykładach użyjemy polecenia Get-Process, a Ty możesz trzymać się jednego z trzech wymienionych poleceń lub przełączać się między nimi, aby zobaczyć różnice w wynikach działania.

Jak wyglądają wyniki działania? Kiedy wykonamy polecenie Get-Process, na ekranie pojawi się tabela zawierająca różne informacje pogrupowane w kilku kolumnach (zobacz rysunek 6.1).

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
45	5	564	2076	18	0.00	1352	coherence
29	5	612	1876	38	0.02	1436	coherence
33	6	756	1028	39	0.02	1444	coherence
100	10	2660	10848	94	2.61	1220	conhost
154	10	1620	2948	46	0.13	396	csrss
196	13	1840	3608	47	0.89	460	csrss
81	7	1084	3808	53	0.02	2056	dllhost
105	9	1616	5336	40	0.03	2820	dllhost
172	18	49016	26712	150	1.44	760	dwm
1511	95	29212	39916	425	8.20	1288	explorer
0	0	0	20	0	0.00	0	Idle
631	17	2900	5796	35	0.58	556	lsass
446	30	56320	15380	181	22.33	1596	MsMpEng
520	38	104620	111024	699	9.09	1776	powershell
276	26	3792	8368	105	0.41	3008	prl_cc
121	11	1612	4332	76	0.08	1476	prl_tools
90	11	1228	3344	51	0.05	1424	prl_tools_ser...
83	10	3868	7892	91	0.31	812	regedit
491	29	14480	8180	615	0.20	2500	SearchIndexer
195	11	3452	5348	32	0.98	548	services
36	2	280	788	4	0.05	288	smss
328	16	3048	5820	47	0.09	1080	spoolsv
583	37	13512	14056	1386	2.13	404	svchost
295	12	2116	6240	36	0.13	632	svchost
313	14	2708	5372	34	0.55	676	svchost
635	26	14036	12976	118	0.53	736	svchost
319	23	13244	9668	93	5.05	856	svchost
574	28	7736	8748	133	0.89	892	svchost
1071	44	11628	13988	134	3.17	932	svchost

Rysunek 6.1. Wynikiem działania polecenia Get-Process jest tabela składająca się z kilku kolumn danych

Oczywiście to wspaniale, że możemy wyświetlić takie informacje na ekranie, ale to nie wszystko, co można z nimi zrobić. Na przykład jeśli chcesz tworzyć wykresy ilustrujące wykorzystanie pamięci i zasobów procesora, możesz wyeksportować takie dane do pliku CSV, który można odczytać w aplikacjach takich jak Microsoft Excel.

6.2.1. Eksportowanie wyników działania do pliku CSV

Eksportowanie wyników działania do pliku to zadanie, do którego wykonania musimy użyć potoku i drugiego polecenia:

```
Get-Process | Export-CSV procs.csv
```

Podobnie jak w przypadku potokowania wyników działania polecenia `Dir` do polecenia `More`, w tym przykładzie wyniki działania polecenia `Get-Process` przekazujemy za pomocą potoku na wejście polecenia `Export-CSV`. Ten drugi cmdlet posiada obowiązkowy parametr pozycyjny, który wykorzystaliśmy do określenia nazwy pliku wyjściowego. Ponieważ polecenie `Export-CSV` jest natywnym cmdletem powłoki PowerShell, doskonale „wie”, jak dokonać konwersji tabeli generowanej przez polecenie `Get-Process` na format standardowego pliku CSV.

Kiedy polecenie zakończy działanie, otwórz nowo utworzony plik CSV w programie Notepad (Notatnik) systemu Windows, aby zobaczyć jego zawartość, tak jak to zostało pokazane na rysunku 6.2.

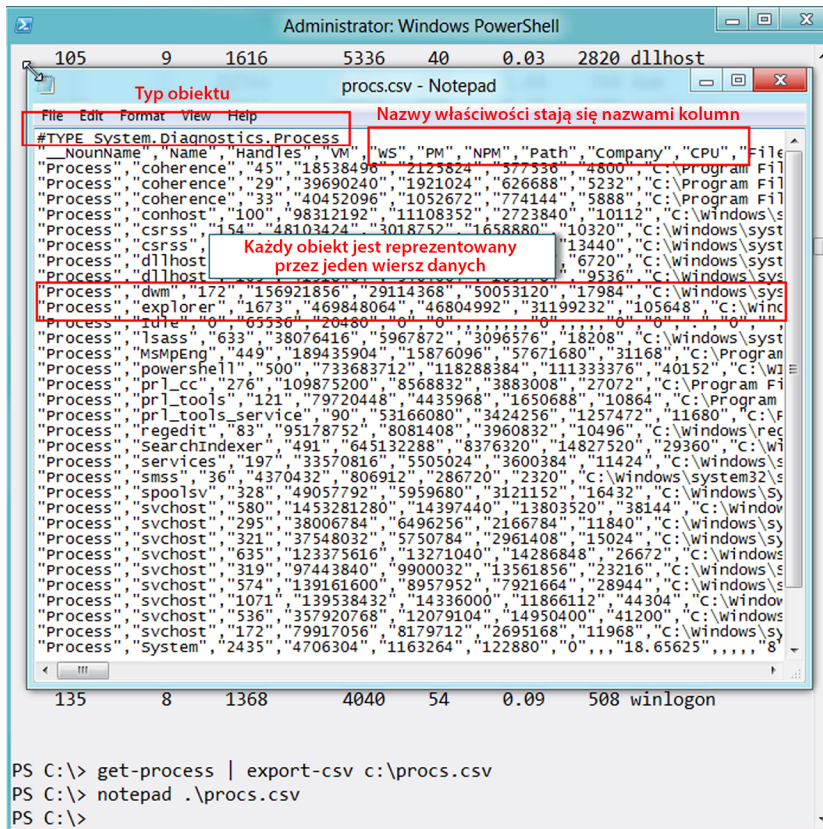
```
Notepad procs.csv
```

Pierwszy wiersz pliku to komentarz poprzedzony znakiem `#`, który identyfikuje rodzaj informacji zawartych w pliku. Na rysunku 6.2 widać, że jest to `System.Diagnostics.Process`, czyli nazwa, której system Windows używa do identyfikowania informacji związanych z uruchomionym procesem. Drugi wiersz zawiera nagłówki kolumn, a kolejne wiersze zawierają listę informacji dotyczących różnych procesów uruchomionych na komputerze.

Za pomocą potoków możesz przysyłać do polecenia `Export-CSV` wyniki działania prawie wszystkich cmdletów z rodziny `Get-` i za każdym razem uzyskiwać doskonałe wyniki. Możesz również zauważyć, że wynikowy plik CSV zawiera znacznie więcej informacji niż to, co zwykle widać na ekranie. Jest to celowe. Powłoka „wie”, że nie może zmieścić wszystkich informacji na ekranie, więc używa pliku konfiguracyjnego dostarczonego przez firmę Microsoft, aby do wyświetlenia na ekranie wybrać tylko najważniejsze informacje. W kolejnych rozdziałach pokażemy, jak zmodyfikować takie ustawienia konfiguracyjne, żeby wyświetlać na ekranie to, co zechcesz.

Po zapisaniu informacji w pliku CSV można łatwo wysłać go w postaci załącznika wiadomości poczty elektronicznej do współpracowników i poprosić o wyświetlenie jego zawartości z poziomu powłoki PowerShell. Aby to zrobić, Twój kolega powinien zaimportować taki plik za pomocą następującego polecenia:

```
Import-CSV procs.csv
```



Rysunek 6.2. Przeglądanie zawartości pliku CSV w programie Notepad

Powłoka PowerShell odczyta zawartość pliku CSV i wyświetli informacje o procesach. Oczywiście nie będą to informacje przekazywane w czasie rzeczywistym, ale będą obrazowały stan systemu w momencie utworzenia pliku CSV.

6.2.2. Eksportowanie wyników działania do pliku XML

A co, jeżeli pliki w formacie CSV nie są tym, czego potrzebujesz? Powłoka PowerShell posiada również polecenie `Export-CliXML`, które tworzy plik w formacie CliXML (ang. *Command Line Interface Extensible Markup Language*). Format CliXML jest unikatowy dla powłoki PowerShell, ale może go odczytać każdy program zdolny do odczytywania plików w formacie XML. Jak łatwo się domyślić, mamy również do dyspozycji analogiczne polecenie `Import-CliXML`, pozwalające na odczytywanie i importowanie takich plików. Polecenia *cmdlet* umożliwiające importowanie i eksportowanie danych (takie jak `Import-CSV` i `Export-CSV`) wymagają podania parametru wywołania reprezentującego cego nazwę pliku.

ZRÓB TO SAM Spróbuj wyeksportować do plików CliXML takie informacje jak lista usług, procesów lub wpisów dziennika zdarzeń. Upewnij się, że możesz ponownie zaimportować takie pliki, i spróbuj otworzyć wynikowy plik w Notatniku oraz Internet Explorerze, tak aby zobaczyć, jak każda z tych aplikacji wyświetla zapisane w pliku informacje.

Czy powłoka PowerShell posiada jakiekolwiek inne polecenia pozwalające na importowanie lub eksportowanie danych? Możesz się tego dowiedzieć, używając polecenia `Get-Command` z parametrem `-verb` mającym wartość, odpowiednio, `Import` lub `Export`.

ZRÓB TO SAM Sprawdź, czy powłoka PowerShell posiada inne cmdlety pozwalające na importowanie lub eksportowanie danych. Możesz powtórzyć tę operację po załadowaniu nowych poleceń do powłoki, o czym będziemy mówić w kolejnym rozdziale.

6.2.3. Porównywanie plików

Zarówno pliki CSV, jak i CliXML mogą być przydatne do utrwalania migawek informacji, udostępniania ich innym osobom i przeglądania ich w późniejszym czasie. Powłoka PowerShell posiada bardzo wygodne polecenie `Compare-Object`, którego możesz użyć do tego celu. Polecenie to posiada alias `Diff`, z którego będziemy korzystać.

Najpierw uruchom polecenie `help diff` i przeczytaj zawartość systemu pomocy dla polecenia `Diff`. Chcielibyśmy, abyś w szczególności zwrócił uwagę na trzy parametry tego polecenia: `-ReferenceObject`, `-DifferenceObject` i `-Property`.

Polecenie `Diff` zostało zaprojektowane do pobierania dwóch zestawów informacji i porównywania ich ze sobą. Na przykład wyobraź sobie, że uruchamiasz polecenie `Get-Process` na dwóch komputerach stojących obok siebie. Komputer skonfigurowany dokładnie tak, jak chcesz, znajduje się po lewej stronie i odgrywa rolę **komputera odniesienia** (ang. *reference computer*). Komputer po prawej może być taki sam lub może być nieco inny; to **komputer badany** (ang. *difference computer*). Po uruchomieniu polecenia `Get-Process` na każdym z nich otrzymasz dwa zestawy informacji o działających procesach, a Twoim zadaniem będzie ustalenie, czy między nimi istnieją jakieś różnice.

Ponieważ badanymi elementami są procesy, to zawsze będziesz widział jakieś różnice w takich właściwościach jak wykorzystanie CPU czy zajętość pamięci, a więc takie kolumny danych można spokojnie zignorować. W rzeczywistości musimy skupić się na kolumnie `Name`, ponieważ chcemy sprawdzić, czy na *badanym komputerze* działają jakieś dodatkowe procesy bądź czy nie działają procesy, które tam powinny być uruchomione. Ręczne porównywanie ze sobą wszystkich nazw procesów z obu tabel mogłoby zająć trochę czasu, ale na szczęście nie musisz tego robić — robi to za Ciebie polecenie `Diff`.

Załóżmy, że siadłeś teraz przed konsolą komputera odniesienia i uruchomiłeś następujące polecenie:

```
Get-Process | Export-CliXML reference.xml
```

Do takiego porównywania wyników działania poleceń wolimy używać plików w formacie CliXML niż CSV, ponieważ pliki w formacie CliXML mogą przechowywać więcej informacji niż zwykłe pliki CSV. Po utworzeniu wynikowego pliku XML musisz przenieść go do komputera badanego i uruchomić następujące polecenie:

```
Diff -reference (Import-CliXML reference.xml) -difference (Get-Process) -property Name
```

Ponieważ polecenie przedstawione powyżej jest dosyć złożone, spróbujemy pokrótce wyjaśnić, co tam się dzieje:

- Podobnie jak w matematyce, nawiasy w powłoce PowerShell kontrolują kolejność wykonywania poleceń. W naszym przykładzie wymuszają uruchomienie poleceń `Import-CliXML` i `Get-Process` przed uruchomieniem polecenia `Diff`. Wyniki działania polecenia `Import-CliXML` są przekazywane jako wartość parametru `-reference`, a wyniki działania polecenia `Get-Process` są podawane jako wartość parametru `-difference`.
- Pełne nazwy parametrów użytych w wywołaniu polecenia `Diff` to `-reference` ↪ `Object` i `-differenceObject`; jak jednak zapewne pamiętasz, możesz je skracać, wpisując takie fragmenty ich nazw, aby powłoka mogła w jednoznaczny sposób określić, którego parametru chcesz użyć. W tym przypadku nazwy `-reference` i `-difference` są oczywiście więcej niż wystarczające do jednoznacznej identyfikacji parametrów i prawdopodobnie mógłbyś skrócić je jeszcze bardziej do czegoś takiego jak `-ref` i `-diff`, a polecenie nadal by działało poprawnie.
- Zamiast porównywania dwóch pełnych tabel polecenie `Diff` koncentruje się na właściwości `Name`, ponieważ zdefiniowaliśmy ją za pomocą parametru `-property`. Gdybyśmy tego nie zrobili, mógłbyś odnieść wrażenie, że każdy proces jest inny, ponieważ wartości kolumn takich jak `VM`, `CPU` i `PM` zawsze będą się od siebie różnić.
- Wynik działania polecenia `Diff` ma postać tabeli informującej, jakie różnice występują między podanymi zestawami danych. Każdy proces, który znajduje się w zestawie odniesienia (`-reference`), ale którego nie ma w zestawie badanym (`-difference`), będzie wyróżniony wskaźnikiem `<=` (oznaczającym, że proces jest obecny tylko po lewej stronie). Jeżeli dany proces będzie obecny na komputerze badanym, ale nie będzie go na komputerze odniesienia, zamiast tego pojawi się wskaźnik `=>`. Procesy, które pasują do obu zestawów, nie są uwzględniane w danych wyjściowych polecenia `Diff`.

ZRÓB TO SAM Powinieneś teraz samodzielnie wypróbować polecenie opisane w powyższym przykładzie. Jeżeli nie masz dwóch komputerów, zacznij od eksportowania bieżących procesów do pliku CliXML, jak pokazano w poprzednim przykładzie. Następnie uruchom dodatkowe procesy, takie jak Notatnik, Windows Paint czy gra Solitaire. Po ich uruchomieniu Twój komputer stanie się komputerem badanym (znajdującym się po prawej stronie), podczas gdy początkowy plik CliXML nadal będzie zbiorem odniesienia (po lewej).

Przykładowe wyniki działania naszego polecenia są następujące:

```
PS C:\> diff -reference (import-xml ref.xml) -difference (get-process) -property name
name                               SideIndicator
----                               -
calc                               =>
mspaint                            =>
notepad                            =>
conhost                            <=
powershell_ise                     <=
```

Jest to sztuczka bardzo przydatna dla każdego administratora. Jeśli potraktujesz referencyjne pliki CliXML jako bazę podstawowych konfiguracji systemu, możesz porównać dowolny bieżący komputer z konfiguracją podstawową i uzyskać raport różnicowy. W naszej książce znajdziesz znacznie więcej poleceń *cmdlet* pobierających informacje o systemie. Ich wyniki działania, po przekierowaniu do plików CliXML, mogą zostać użyte jako pliki bazowe dla usług, procesów, konfiguracji systemu operacyjnego, użytkowników, grup i wielu innych elementów oraz być wykorzystywane później do porównywania bieżącego stanu systemu z konfiguracją odniesienia.

ZRÓB TO SAM Dla sprawdzenia spróbuj ponownie uruchomić opisane wyżej polecenie *Diff*, ale tym razem całkowicie pomiń parametr *-property*. Jakie są wyniki działania takiego polecenia? Łatwo zauważyć, że na liście różnic wymienione zostały wszystkie procesy, ponieważ wartości ich właściwości takich jak *PM*, *VM* i wielu innych uległy zmianie, nawet jeśli są to te same procesy, działające na tym samym komputerze. Wyniki działania samego polecenia *Diff* również nie są tak użyteczne, ponieważ dla każdego elementu wyświetlane są tylko typ i nazwa procesu.

A tak przy okazji, powinieneś wiedzieć, że polecenie *Diff* zasadniczo nie radzi sobie najlepiej przy porównywaniu plików tekstowych. Choć inne systemy operacyjne i powłoki posiadają zazwyczaj komendę *Diff*, która jest wyraźnie przeznaczona do porównywania plików tekstowych, polecenie *Diff* powłoki PowerShell działa zupełnie inaczej. Po wykonaniu ćwiczeń na końcu tego rozdziału przekonasz się, na czym polegają różnice w działaniu między poleceniem *Diff* a innymi poleceniami tego typu.

UWAGA Jeśli wydaje Ci się, że często korzystamy w naszych przykładach z poleceń *Get-Process*, *Get-Service* i *Get-EventLog*, to spieszmy powiedzieć, że robimy to celowo. Chodzi o to, że możemy zagwarantować, że na pewno masz dostęp do tych poleceń, ponieważ są one macierzystymi *cmdletami* powłoki PowerShell i nie wymagają zainstalowania żadnych dodatkowych pakietów, takich jak *Exchange* czy *SharePoint*. Nie zmienia to jednak w niczym faktu, że umiejętności, które zdobywasz, odnoszą się do każdego innego polecenia *cmdlet*, które kiedykolwiek będziesz musiał uruchomić, w tym i dostarczanych z pakietami *Exchange*, *SharePoint*, *SQL Server* i innymi produktami serwerowymi. Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 26., ale na razie skup się na tym, *jak* używać *cmdletów*, a nie na tym, jakie działania one wykonują. Inne polecenia *cmdlet* będziemy poznawać nieco później, w bardziej odpowiednim czasie.

6.3. Przesyłanie wyników działania polecenia do pliku lub na drukarkę

Zawsze, gdy otrzymujesz dobrze sformatowane dane wyjściowe, takie jak tabele generowane przez polecenia `Get-Service` lub `Get-Process`, możesz zachować je w pliku lub nawet na papierze. Domyślnie wyniki działania poleceń powłoki PowerShell są wyświetlane na ekranie, ale oczywiście w razie potrzeby można to zmienić. Nieco wcześniej pokazaliśmy już jeden ze sposobów, w jaki można to zrobić:

```
Dir > DirectoryList.txt
```

Znak `>` jest skrótem dodanym do powłoki PowerShell, zapewniającym zgodność składni ze starszą powłoką `cmd.exe`. W rzeczywistości po uruchomieniu tego polecenia powłoka PowerShell „pod maską” wykonuje następujące polecenie:

```
Dir | Out-File DirectoryList.txt
```

Oczywiście zamiast używać składni ze znakiem `>`, możesz od razu uruchomić polecenie przedstawione powyżej. Dlaczego miałbyś jednak tak robić? Poważnym argumentem jest to, że polecenie `Out-File` posiada szereg dodatkowych parametrów, które umożliwiają określenie alternatywnego kodowania znaków (takiego jak UTF-8 czy Unicode), dołączanie nowych danych do istniejącego pliku i tak dalej. Domyślnie pliki tworzone przez polecenie `Out-File` mają szerokość 80 kolumn, co oznacza, że czasami powłoka PowerShell może zmienić formatowanie wyników polecenia tak, aby mieściły się w granicach 80 znaków. Taka zmiana może spowodować, że zawartość pliku będzie wyglądać nieco inaczej niż po uruchomieniu tego samego polecenia i wyświetleniu jego wyników działania na ekranie. Zajrzyj do systemu pomocy dla polecenia `Out-File` i sprawdź, czy możesz znaleźć parametr, który pozwoliłby zmienić szerokość wiersza pliku wyjściowego na więcej (lub mniej) niż domyślne 80 znaków.

ZRÓB TO SAM Nie szukaj tutaj odpowiedzi — otwórz plik pomocy i zobacz, co możesz tam znaleźć. Gwarantujemy, że już po krótkiej chwili znajdziesz odpowiedni parametr.

Powłoka PowerShell posiada wiele poleceń *cmdlet*. Jedno z nich nazywa się `Out-Default` i jest tym, którego używa powłoka, gdy w wierszu wywołania nie podasz innego cmdletu z serii `Out-`. Przykładowo, jeżeli wykonasz polecenie:

```
Dir
```

to, technicznie rzecz biorąc, uruchamiasz następujące polecenie:

```
Dir | Out-Default
```

nawet jeżeli nie miałeś o tym pojęcia. Polecenie `Out-Default` nie robi nic więcej poza bezpośrednim przekierowaniem otrzymywanych danych do polecenia `Out-Host`, co oznacza, że tak naprawdę uruchamiasz takie polecenie:

```
Dir | Out-Default | Out-Host
```

nadal nie zdając sobie z tego sprawy. Polecenie `Out-Host` po prostu wyświetla informacje na ekranie. Jakie inne polecenia *cmdlet* z serii `Out-` możesz znaleźć?

ZRÓB TO SAM Czas zbadać inne polecenia z serii `Out-`. Aby rozpocząć, spróbuj użyć polecenia `Help` i symboli wieloznacznych, na przykład w taki sposób: `Help Out *`. Innym rozwiązaniem jest użycie polecenia `Get-Command` w podobny sposób, na przykład `Get-Command Out *`. Zamiast tego możesz również użyć parametru `-verb`, na przykład `Get-Command -verb Out`. Czego się dowiedziałeś?

`Out-Printer` jest prawdopodobnie jednym z najbardziej użytecznych *cmdletów* z serii `Out-`, choć dostępny jest tylko w systemie Windows. Polecenie `Out-GridView` jest również bardzo przydatne, ale wymaga zainstalowania pakietu Microsoft .NET Framework v3.5 oraz środowiska ISE Windows PowerShell, co nie zawsze jest możliwe do zrealizowania w serwerowych systemach operacyjnych oraz na platformach innych niż Windows. Jeżeli jednak w Twoim systemie takie pakiety są zainstalowane, spróbuj uruchomić polecenie `Get-Service | Out-GridView` i zobacz, co się stanie. Polecenia `Out-Null` i `Out-String` mają swoje określone zastosowania, o których nie będziemy teraz mówić, ale zachęcamy do zapoznania się z ich plikami pomocy i zamieszczonymi w nich przykładami.

6.4. Konwersja na pliki w formacie HTML

Chcesz tworzyć raporty w formacie HTML? Taka opcja nie jest dostępna w innych systemach operacyjnych (a przynajmniej nie była w czasie pisania tej książki). Ale w systemie Windows jest to łatwe — wystarczy przekazać wyniki działania polecenia do *cmdletu* `ConvertTo-HTML` generującego dobrze sformatowany plik w formacie HTML, który może być wyświetlany w dowolnej przeglądarce internetowej. Już domyślnie taki raport wygląda całkiem nieźle, ale jeżeli chcesz utworzyć jeszcze bardziej atrakcyjne formatowanie, możesz zmodyfikować odpowiednie ustawienia za pomocą pliku CSS. Zauważ, że polecenie `ConvertTo-HTML` nie wymaga podawania nazwy pliku wyjściowego: `Get-Service | ConvertTo-HTML`

ZRÓB TO SAM Zanim przejdziesz dalej, spróbuj samodzielnie wykonać takie przykładowe polecenie — chcemy, abyś na własne oczy zobaczył wyniki jego działania.

W świecie powłoki PowerShell czasownik `Export` oznacza, że pobierasz dane, konwertujesz je na inny format i zapisujesz w wybranym magazynie danych, takim jak plik. Czasownik `ConvertTo` implikuje tylko część tego procesu — konwersję na inny format, ale już bez zapisywania go w pliku. Po uruchomieniu powyższego polecenia na ekranie został wyświetlony kod HTML, co prawdopodobnie nie było zgodne z Twoimi oczekiwaniami. Pomyśl przez chwilę: jak możesz zapisać taki kod HTML w pliku tekstowym na dysku?

ZRÓB TO SAM Jeżeli potrafisz wymyślić jakiś sposób wykonania takiej operacji, wypróbuj go, zanim przejdiesz dalej.

Aby zapisać wygenerowany kod HTML w pliku na dysku, możesz wykonać następujące polecenie:

```
Get-Service | ConvertTo-HTML | Out-File services.html
```

Zauważyłeś, w jaki sposób łączenie ze sobą kilku poleceń pozwala wykonywać coraz bardziej złożone operacje? Każde polecenie realizuje jeden krok z całego procesu, a wiersz polecenia jako całość pozwala na wykonanie użytecznego zadania.

Powłoka PowerShell posiada szereg innych poleceń *cmdlet* z serii *ConvertTo-*, takich jak *ConvertTo-CSV* i *ConvertTo-XML*. Podobnie jak w przypadku polecenia *ConvertTo-HTML*, nie zapisują one pliku wynikowego na dysku; zamiast tego dokonują konwersji danych otrzymanych na wejściu na format CSV lub XML. Oczywiście możesz przekazać przekonwertowane dane z wyjścia do polecenia *Out-File*, które zapisze je w postaci pliku na dysku (choć znacznie bardziej efektywnym rozwiązaniem byłoby użycie jednego z poleceń *Export-CSV* lub *Export-CliXML*, które dokonują konwersji danych i od razu zapisują je w pliku na dysku).

Dla zainteresowanych

Czas na kilka dodatkowych informacji, które mogą wydawać się nieco bezużyteczne, niemniej jednak odpowiadają na pytanie, które często zadają nam nasi studenci — dlaczego firma Microsoft zdecydowała się na zaimplementowanie spełniających niemal identyczne role poleceń *Export-CSV* i *ConvertTo-CSV* oraz dwóch analogicznych *cmdletów* dla formatu XML?

W niektórych zaawansowanych scenariuszach przetwarzania danych zapisywanie wyników w postaci pliku na dysku nie zawsze jest potrzebne. Na przykład możesz dokonać konwersji danych na format XML, a następnie przesłać je do usługi internetowej lub innego miejsca docelowego. Dzięki różnym poleceniom z serii *ConvertTo-*, które nie zapisują danych w pliku, masz dużą elastyczność i możesz realizować złożone zadania w najbardziej wygodny dla siebie sposób.

6.5. Używanie poleceń *cmdlet* do modyfikowania systemu — zakończenie działania procesów i zatrzymywanie usług

Eksportowanie i konwertowanie danych to nie jedyne powody, dla których warto połączyć ze sobą dwa polecenia. Na przykład zastanów się, jak działa następujące polecenie (*ale nie uruchamiaj go*):

```
Get-Process | Stop-Process
```

Czy możesz sobie wyobrazić, co zrobi to polecenie? Podpowiemy Ci: spowoduje spektakularną awarię Twojego systemu. Polecenie to pobiera informacje o poszczególnych procesach, a następnie próbuje zakończyć ich działanie. Kiedy nasze polecenie natrafi na krytyczny proces, taki jak *lsass.exe* (ang. *Local Security Authority Subsystem*

Service), Twój komputer najprawdopodobniej zakończy działanie ze słynnym błękitnym ekranem śmierci na monitorze (ang. *Blue Screen of Death* — BSOD). Jeżeli pracujesz z powłoką PowerShell na maszynie wirtualnej i chcesz się trochę zabawić, możesz spróbować uruchomić tę komendę.

Cała sztuczka polega na tym, że polecenia *cmdlet* z tym samym rzeczownikiem (w tym przypadku *Process*) często mogą przekazywać informacje między sobą. W typowym scenariuszu zazwyczaj podajesz nazwę określonego procesu i tym samym nie próbujesz zatrzymać wszystkich pozostałych:

```
Get-Process -name Notepad | Stop-Process
```

Usługi oferują podobną funkcjonalność — dane wyjściowe z polecenia *Get-Service* można przesyłać do cmdletów takich jak *Stop-Service*, *Start-Service* czy *Set-Service*.

Jak można się spodziewać, istnieją określone zasady regulujące to, które polecenia mogą się ze sobą łączyć. Na przykład jeżeli spojrzymy na następującą sekwencję poleceń: *Get-ADUser* | *New-SQLDatabase*, to raczej nie powinniśmy oczekiwać, że jej wykonanie przyniesie jakieś rozsądne rezultaty (choćczasami w takich przypadkach wyniki mogą być mocno zaskakujące i zupełnie pozbawione sensu). W rozdziale 7. nieco bardziej zagłębimy się w reguły określające sposób, w jaki poszczególne polecenia mogą się ze sobą łączyć.

Chcielibyśmy, abyś wiedział jeszcze jedną rzecz o cmdletach takich jak *Stop-Service* i *Stop-Process*. Te polecenia w pewnym stopniu modyfikują system, a wszystkie cmdlety modyfikujące system mają zdefiniowaną wewnętrznie właściwość określającą **poziom wpływu** danego cmdletu na system (ang. *impact level*). Autor polecenia definiuje ten poziom i nie możesz go zmienić. Powłoka PowerShell posiada odpowiednią zmienną o nazwie *\$ConfirmPreference*, która domyślnie jest ustawiona na wartość *High* (wysoki). Aby wyświetlić bieżącą wartość tej zmiennej, powinieneś w wierszu polecenia wpisać jej nazwę, tak jak to zostało pokazane poniżej:

```
PS C:\> $confirmPreference
High
```

Działa to w następujący sposób. Kiedy wewnętrzne ustawienia cmdletu są równe lub wyższe niż wartość zmiennej *\$ConfirmPreference* powłoki, przy próbie uruchomienia takiego polecenia PowerShell automatycznie zapyta, czy na pewno chcesz to zrobić (z ang. *Are you sure?*). Jeżeli użyłeś maszyny wirtualnej do wypróbowania polecenia zatrzymującego usługi i w efekcie powodującego awarię systemu, o której wspomnieliśmy wcześniej, to prawdopodobnie takie pytanie pojawiało się na ekranie przed zatrzymaniem każdego z procesów. Kiedy wewnętrzne ustawienia cmdletu są mniejsze niż wartość zmiennej *\$ConfirmPreference*, to taka prośba o potwierdzenie nie będzie się pojawiała.

W razie potrzeby możesz jednak zmusić powłokę, aby poprosiła o potwierdzenie zamiaru wykonania danego polecenia:

```
Get-Service | Stop-Service -confirm
```

Jak widać, aby wymusić potwierdzenie, wystarczy po prostu dodać parametr `-confirm` do wiersza wywołania `cmdletu`. Parametr ten powinien być obsługiwany przez dowolne polecenie, które wprowadza jakąś zmianę do systemu, a informacja o tym powinna być zamieszczona w pliku pomocy dla takiego polecenia.

Podobnym parametrem jest `-whatif`, który jest obsługiwany przez wszystkie polecenia `cmdlet` obsługujące parametr `-confirm`. Parametr `-whatif` nie jest domyślnie włączony, ale można go dodać w dowolnym momencie:

```
PS C:\> get-process | stop-process -whatif
What if: Performing operation "Stop-Process" on Target "conhost (1920)".
What if: Performing operation "Stop-Process" on Target "conhost (1960)".
What if: Performing operation "Stop-Process" on Target "conhost (2460)".
What if: Performing operation "Stop-Process" on Target "csrss (316)".
```

Wykonanie takiego polecenia pokaże Ci, co zrobiłoby dane polecenie, jednak bez konieczności samego wykonywania takiej operacji. Jest to bardzo użyteczna metoda pozwalająca na sprawdzenie, co potencjalnie niebezpieczne polecenie zrobiłoby w systemie, dzięki czemu możesz dodatkowo upewnić się, czy na pewno chcesz je uruchomić.

6.6. Najczęściej spotykane problemy

Jednym z elementów pracy z powłoką PowerShell, który może wprowadzać użytkowników w zakłopotanie, są polecenia `Export-CSV` i `Export-CliXML`. Technicznie rzecz biorąc, oba polecenia tworzą pliki tekstowe. Zawartość plików będących wynikiem działania obu poleceń można wyświetlić w zwykłym Notatniku, jak to wcześniej pokazywaliśmy na rysunku 6.2. Musisz jednak przyznać, że wygląd tego tekstu jest dosyć specyficzny, niezależnie od tego, czy mówimy o formacie CSV, czy XML.

Problem zaczyna się w momencie, kiedy użytkownik zostanie poproszony o załadowanie tych plików z powrotem do powłoki. Czy używałeś już polecenia `Get-Content` (lub jego aliasów `Type` lub `Cat`)? Załóżmy, że robisz tak:

```
PS C:\> get-eventlog -LogName security -newest 5 | export-csv events.csv
```

Teraz spróbuj odczytać zawartość wyeksportowanego pliku, używając do tego celu polecenia `Get-Content`:

```
PS C:\> Get-Content .\events.csv
#TYPE System.Diagnostics.EventLogEntry#security/Microsoft-Windows-Security-Auditing/4797
"EventID","MachineName","Data","Index","Category","CategoryNumber","EntryType","Message",
↳"Source","ReplacementStrings","InstanceId","TimeGenerated","TimeWritten","UserName","Site",
↳"Container"
"4797","DONJONES1D96","System.Byte[]","263","(13824)","13824","SuccessAudit","An attempt was
↳made to query the existence of a blank password for an account.
Subject:
  Security ID:          S-1-5-21-87969579-3210054174-450162487-100
  Account Name:         donjones
  Account Domain:       DONJONES1D96
  Logon ID:             0x10526
Additional Information:
  Caller Workstation:    DONJONES1D96
```



```

Target Account Name: Guest
Target Account Domain: DONJONES1D96", "Microsoft-Windows-Security-Auditing", "System.
↳String[]", "4797", "3/29/2012 9:43:36 AM", "3/29/2012 9:43:36 AM", ...
"4616", "DONJONES1D96", "System.Byte[]", "262", "(12288)", "12288", "SuccessAudit", "The system
↳time was changed.

```

Prezentujemy tutaj tylko niewielki fragment zawartości tego pliku, ale cała reszta jest mniej więcej podobna. Jest to niemal nieczytelne, prawda? Tak właśnie wyglądają „surowe”, nieprzetworzone dane w formacie CSV. Nasze polecenie nie próbowało w żaden sposób interpretować lub inaczej mówiąc, *parsować* tych danych. Dla kontrastu porównajmy teraz wyniki działania polecenia Import-CSV:

```
PS C:\> import-csv .\events.csv
```

```

EventID           : 4797
MachineName       : DONJONES1D96
Data              : System.Byte[]
Index             : 263
Category          : (13824)
CategoryNumber    : 13824
EntryType         : SuccessAudit
Message           : An attempt was made to query the existence of a
                   blank password for an account.
                   Subject:
                       Security ID: S-1-5-21-87969579-3210054174-450162487-1001
                       Account Name: donjones
                       Account Domain: DONJONES1D96
                       Logon ID: 0x10526
                   Additional Information:
                       Caller Workstation: DONJONES1D96
                       Target Account Name: Guest
                       Target Account Domain: DONJONES1D96
Source            : Microsoft-Windows-Security-Auditing
ReplacementStrings : System.String[]
InstanceId        : 4797
TimeGenerated     : 3/29/2012 9:43:36 AM
TimeWritten       : 3/29/2012 9:43:36 AM
UserName          :

```

Wygląda to o wiele lepiej, prawda? Polecenia z serii Import- zwracają uwagę na zawartość pliku, próbują ją zinterpretować i następnie wyświetlić w sposób, który przypomina wyniki działania oryginalnego polecenia (w tym przypadku Get-EventLog). Zazwyczaj, jeśli stworzysz plik za pomocą polecenia Export-CSV, odczytujesz go przy użyciu polecenia Import-CSV. Jeżeli stworzysz plik za pomocą polecenia Export-CliXML, zazwyczaj odczytujesz go, używając polecenia Import-CliXML. Stosując te polecenia parami, uzyskujesz znacznie lepsze wyniki. Z polecenia Get-Content powinieneś korzystać, tylko kiedy odczytujesz dane z pliku tekstowego i nie chcesz, aby PowerShell próbował analizować dane — czyli inaczej mówiąc, tylko wtedy, kiedy chcesz pracować z nieprzetworzonym tekstem.

6.7. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Rozdział ten jest nieco krótszy niż pozostałe, ponieważ wykonanie niektórych przykładów prawdopodobnie zabrało Ci trochę więcej czasu, a także dlatego, że chcemy, abyś poświęcił więcej czasu na wykonywanie opisanych niżej ćwiczeń praktycznych. Jeśli nie wykonałeś jeszcze wszystkich zadań zamieszczonych w tym rozdziale w sekcjach „Zrób to sam”, zdecydowanie zalecamy, żebyś to zrobił przed przystąpieniem do wykonywania tych ćwiczeń:

1. Utwórz dwa podobne, ale nieco różniące się od siebie pliki tekstowe. Spróbuj porównać je za pomocą polecenia `Diff`. Wypróbuj następujące polecenie:
`Diff -reference (Get-Content File1.txt) -difference (Get-Content File2.txt).`
Jeżeli pliki różnią się tylko jednym wierszem tekstu, polecenie powinno działać poprawnie.
2. Co się stanie (w systemie Windows), jeżeli z poziomu konsoli uruchomisz następujące polecenie:
`Get-Service | Export-CSV services.csv | Out-File`
Dlaczego tak się dzieje?
3. Czy oprócz pobierania jednej lub więcej usług i przesyłania ich na wejście polecenie `Stop-Service` pozwala w inny sposób określić usługę lub usługi, które chcesz zatrzymać? Czy można zatrzymać wybraną usługę bez użycia polecenia `Get-Service`?
4. Co powinieneś zrobić, jeżeli chcesz utworzyć plik CSV, w którym poszczególne elementy danych zamiast przecinkami będą rozdzielane znakami potoku (`|`)? Nadal powinieneś skorzystać z polecenia `Export-CSV`, ale jakich parametrów wywołania użyjesz?
5. Czy istnieje sposób na wyeliminowanie wiersza komentarza rozpoczynającego się od znaku `#`, który znajduje się na początku wyeksportowanego pliku CSV? Wiersz ten zazwyczaj zawiera informacje o rodzaju danych zamieszczonych w pliku, ale co powinieneś zrobić, jeżeli chcesz pominąć taki wiersz w konkretnym pliku?
6. Polecenia `Export-CLIXML` i `Export-CSV` modyfikują system, ponieważ mogą tworzyć nowe i nadpisywać istniejące pliki. Użycie którego parametru uniemożliwiłoby nadpisanie istniejącego pliku? Który parametr spowoduje, że przed rozpoczęciem zapisywania danych w pliku polecenie zapyta, czy na pewno chcesz rozpocząć taką operację?
7. W ustawieniach regionalnych systemu Windows możesz wybrać znak, który będzie używany jako domyślny separator list. W systemach z amerykańskimi ustawieniami separatorem listy jest przecinek. Jak spowodować, aby polecenie `Export-CSV` zamiast przecinka używało domyślnych ustawień systemowych?

ZRÓB TO SAM Po ukończeniu tego laboratorium spróbuj wykonać ćwiczenia podsumowujące dla rozdziałów 1. – 6., które znajdziesz w dodatku „Ćwiczenia podsumowujące”.

6.8. Odpowiedzi

1. Wykonaj sekwencję poleceń przedstawioną poniżej:

```
PS C:\> "Jestem wielorybem" | out-file file1.txt
PS C:\> "Jestem pingwinem" | out-file file2.txt
PS C:\> $f1=get-content .\file1.txt
PS C:\> $f2=Get-Content .\file2.txt
PS C:\> diff $f1 $f2
InputObject                               SideIndicator
-----
Jestem wielorybem                         =>
Jestem pingwinem                          <=
```

2. Jeżeli wywołując polecenie Out-File, nie podasz nazwy pliku, pojawi się błąd. Ale nawet jeżeli podasz tę nazwę, polecenie Out-File nie zrobi niczego, ponieważ plik zostanie zapisany na dysku przez polecenie Export-CSV.
3. Polecenie Stop-Service może pobierać jedną lub więcej nazw usług jako wartości parametru -Name. Na przykład możesz uruchomić takie polecenie:
Stop-Service spooler
4. Get-Service | Export-Csv services.csv -Delimiter "|"
5. Użyj parametru -NoTypeInfo polecenia Export-CSV.
6. Get-Service | Export-Csv services.csv -noclobber
Get-Service | Export-Csv services.csv -confirm
7. Get-Service | Export-Csv services.csv -UseCulture

7

Dodawanie poleceń

Jedną z głównych zalet powłoki PowerShell jest jej elastyczność i łatwość, z jaką można zwiększać jej funkcjonalność. Ponieważ firma Microsoft coraz bardziej rozwija powłokę PowerShell, opracowuje dla niej coraz więcej poleceń powiązanych z konkretnymi produktami, takimi jak Exchange Server, SharePoint Server, rodzina System Center czy SQL Server. Zazwyczaj po zainstalowaniu pakietu narzędzi do zarządzania danym produktem otrzymujesz graficzną konsolę zarządzania oraz jedno lub więcej rozszerzeń dla powłoki Windows PowerShell.

7.1. Jak jedna powłoka może zrobić wszystko

Zakładamy, że masz już pewne doświadczenia w pracy z graficzną konsolą MMC (ang. *Microsoft Management Console*), dlatego użyjemy jej jako przykładu do zilustrowania sposobu działania powłoki PowerShell. Jeżeli chodzi o rozszerzalność, oba produkty działają podobnie, co nie powinno być zbyt wielkim zaskoczeniem, jeżeli weźmiemy pod uwagę fakt, że nad tworzeniem i rozwojem zarówno konsoli MMC, jak i powłoki PowerShell pracuje ten sam zespół deweloperów firmy Microsoft.

Kiedy otwierasz nową, pustą konsolę MMC, to jest ona trochę bezużyteczna. W zasadzie nie możesz nic zrobić, ponieważ MMC ma bardzo mało swoich wbudowanych funkcji. Aby to narzędzie stało się naprawdę przydatne, przejdź do menu *Plik* i wybierz polecenie *Dodaj/Usuń przystawkę...*. W świecie konsoli MMC **przystawkami** są takie narzędzia jak *Użytkownicy i komputery usługi Active Directory*, *Zarządzanie DNS* czy *Zarządzanie DHCP*. W zależności od potrzeb możesz dodawać do konsoli MMC niemal dowolną liczbę przystawek, a następnie zapisać utworzoną konfigurację konsoli, tak aby ułatwić ponowne otwarcie tego samego zestawu przystawek w przyszłości.

Skąd pochodzą przystawki konsoli MMC? Po zainstalowaniu narzędzi do zarządzania, powiązanych z takimi produktami jak Exchange Server, Forefront czy System Center,

przystawki tych produktów stają się dostępne w oknie dialogowym *Dodawanie lub usuwanie przystawek* konsoli MMC. Większość takich produktów instaluje również swoje wstępnie skonfigurowane pliki konsoli MMC, które pozwalają na szybkie uruchomienie konsoli MMC i wstępne załadowanie do niej danej przystawki (lub zestawu przystawek). Oczywiście jeżeli nie chcesz, nie musisz używać tych prekonfigurowanych konsoli, ponieważ zawsze możesz otworzyć pustą konsolę MMC i ręcznie załadować odpowiednie przystawki. Przykładowo, wstępnie skonfigurowana konsola MMC programu Exchange Server nie zawiera przystawki *Lokalizacje i usługi Active Directory*, ale możesz łatwo utworzyć swoją konfigurację konsoli MMC, która będzie obejmowała zarówno program Exchange, jak i inne potrzebne Ci przystawki.

Powłoka PowerShell działa niemal dokładnie w taki sam sposób — wystarczy zainstalować narzędzia do zarządzania danym produktem (opcja instalacji narzędzi do zarządzania jest zazwyczaj zawarta w menu programu instalacyjnego — kiedy instalujesz produkt, taki jak Exchange Server w systemie Windows 7, narzędzia do zarządzania często są jedyną dostępną opcją instalatora). Po zainstalowaniu takich narzędzi uzyskasz dostęp do wszystkich powiązanych z nimi rozszerzeń powłoki PowerShell, a nawet będziesz mógł utworzyć powłokę przeznaczoną do zarządzania danym produktem.

7.2. Powłoki przeznaczone do zarządzania danym produktem

Powłoki przeznaczone do zarządzania danym produktem niemal od początku były źródłem ogromnych nieporozumień. W tym miejscu musimy wyraźnie powiedzieć, że istnieje tylko jedna powłoka Windows PowerShell — nie ma oddzielnej wersji powłoki PowerShell przeznaczonej do zarządzania serwerami Exchange czy usługą Active Directory; to wszystko ciągle odbywa się w jednej i tej samej powłoce PowerShell.

Weźmy przykład Active Directory. W menu *Start* kontrolera domeny działającego pod kontrolą systemu Windows Server 2008 R2, w katalogu *Narzędzia administracyjne* znajduje się ikona modułu *Active Directory dla Windows PowerShell*. Jeśli klikniesz ten element prawym przyciskiem myszy i z menu podręcznego wybierzesz polecenie *Właściwości*, pierwszą rzeczą, którą powinieneś zobaczyć, jest pole *Element docelowy*, które będzie miało następującą wartość:

```
%windir%\system32\WindowsPowerShell\v1.0\powershell.exe -noexit -command import-module  
↳ActiveDirectory
```

Ta komenda uruchamia standardową aplikację powłoki PowerShell.exe i za pomocą odpowiedniego parametru wiersza polecenia przekazuje jej polecenie uruchomienia konkretnej komendy: `Import-Module ActiveDirectory`. Wynikiem takiej operacji jest kopia procesu powłoki, która ma wstępnie załadowany moduł `ActiveDirectory`. Z drugiej jednak strony nie potrafimy wymyślić żadnego rozsądnego powodu, dla którego nie można by uruchomić „normalnej” powłoki PowerShell, wykonać takie polecenie ręcznie i w efekcie uzyskać tę samą funkcjonalność.

To samo dotyczy prawie każdej powłoki do zarządzania takimi produktami jak Exchange czy SharePoint. Sprawdź w menu *Start* właściwości skrótów tych produktów, a przekonasz się, że uruchamiają one normalną powłokę PowerShell.exe i przekazują odpowiedni parametr wiersza polecenia, aby zaimportować wybrany moduł, dodać przystawkę lub załadować wstępnie skonfigurowany plik konsoli (a plik konsoli to po prostu lista przystawek do automatycznego załadowania).

SQL Server 2008 i SQL Server 2008 R2 są tutaj wyjątkami. Ich „specyficzna dla produktu” powłoka, *Sqllps*, jest specjalnie skompilowaną wersją powłoki PowerShell, która uruchamia tylko rozszerzenia SQL Server — właściwie takie rozwiązanie nazywany **minipowłoką** (ang. *mini-shell*). Firma Microsoft wypróbowała to podejście po raz pierwszy w serwerze SQL Server, było to jednak rozwiązanie bardzo niepopularne i szybko z niego zrezygnowano — w SQL Server 2012 użyto już powłoki PowerShell.

Nie jesteś ograniczony do pracy tylko z określonymi wcześniej rozszerzeniami. Przykładowo, po uruchomieniu powłoki do zarządzania serwerem Exchange możesz wykonać polecenie `Import-Module ActiveDirectory` i w ten sposób, zakładając oczywiście, że moduł `ActiveDirectory` był wcześniej zainstalowany na Twoim komputerze, dodać do powłoki Exchange funkcje zarządzania usługą `Active Directory`. Zamiast tego możesz też po prostu uruchomić standardową konsolę PowerShell i ręcznie dodać dowolne rozszerzenia.

Jak już wspominaliśmy nieco wcześniej, konsole przeznaczone do zarządzania określonymi produktami były ogromnym źródłem problemów dla użytkowników, zwłaszcza tych wierzących, że istnieje wiele wersji powłoki PowerShell, które wzajemnie nie mogą korzystać ze swoich funkcjonalności. Kilka lat temu Don wdał się nawet w poważną dyskusję na ten temat na swoim blogu i w końcu musiał poprosić członków zespołu deweloperów PowerShell, aby wkroczyli do akcji i wsparli go, jednoznacznie wyjaśniając całą sytuację. Krótko mówiąc, powinieneś nam zaufać: w pojedynczej instancji powłoki PowerShell możesz mieć zarówno pełną funkcjonalność samej powłoki PowerShell, jak i jej wszystkich rozszerzeń, a specyficzne dla wybranych produktów skróty do powłok „zarządzania” tworzone w menu *Start* w żaden sposób nie ograniczają Twoich możliwości ani nie sugerują, że istnieją specjalne wersje powłoki PowerShell dla tych produktów.

7.3. Rozszerzenia — wyszukiwanie i dodawanie przystawek

Powłoka PowerShell ma dwa rodzaje rozszerzeń: moduły (ang. *modules*) i przystawki (ang. *snap-ins*). Najpierw przyjrzymy się przystawkom.

Prawidłową, właściwą nazwą przystawki dla powłoki PowerShell jest *PSSnapin*, co odróżnia ją od innych przystawek, przeznaczonych dla graficznej konsoli MMC. Przystawki *PSSnapin* zostały po raz pierwszy zastosowane dla powłoki PowerShell v1. Przystawka *PSSnapin* zazwyczaj składa się z jednego lub więcej plików DLL, wraz z dodatkowymi plikami XML, które zawierają ustawienia konfiguracyjne i tekst pomocy. Zanim będzie można użyć przystawek *PSSnapin*, muszą one zostać odpowiednio zainstalowane i zarejestrowane, tak aby powłoka PowerShell wiedziała, że istnieją i są dostępne.

UWAGA Firma Microsoft wydaje się stopniowo coraz bardziej odchodzić od koncepcji przystawek PSSnapin, stąd w przyszłości liczba takich produktów będzie się sukcesywnie zmniejszać. Zgodnie z informacjami płynącymi od deweloperów nowe rozszerzenia funkcjonalności powłoki PowerShell w przyszłości będą raczej dostarczane w postaci modułów.

Aby wyświetlić na ekranie listę dostępnych przystawek, powinieneś z poziomu konsoli powłoki PowerShell uruchomić polecenie `Get-PSSnapin -registered`. Na naszym komputerze, który jest kontrolerem domeny i posiada pakiet SQL Server 2008, wyniki działania tego polecenia są następujące:

```
PS C:\> get-pssnapin -registered
Name                : SqlServerCmdletSnapin100
PSVersion           : 2.0
Description          : This is a PowerShell snap-in that includes various SQL
                      Server cmdlets.
Name                : SqlServerProviderSnapin100
PSVersion           : 2.0
Description          : SQL Server Provider
```

Wyniki działania mówią, że mamy dwie przystawki zainstalowane i dostępne, które jednak nie są załadowane. Aby wyświetlić listę załadowanych przystawek, powinieneś uruchomić polecenie `Get-PSSnapin`. Wykonanie tego polecenia powoduje wyświetlenie listy zawierającej wszystkie podstawowe, automatycznie ładowane przystawki, zapewniające natywną funkcjonalność powłoki PowerShell.

Aby załadować wybraną przystawkę, powinieneś uruchomić polecenie `Add-PSSnapin` i jako parametr wywołania podać nazwę przystawki:

```
PS C:\> add-pssnapin sqlservercmdletsnapin100
```

Jak zwykle nie musisz się martwić o poprawne pisanie wielkich i małych liter — powłoka PowerShell się tym nie przejmuje.

Po załadowaniu przystawki zapewne będziesz chciał się dowiedzieć, co zostało dodane do powłoki. Przystawki PSSnapin mogą dodawać cmdlety oraz dostawców PSDrive. Aby dowiedzieć się, jakie cmdlety zostały dodane, powinieneś użyć polecenia `Get-Command` (lub jego aliasu `Gcm`):

```
PS C:\> gcm -pssnapin sqlservercmdletsnapin100
CommandType      Name                Definition
-----
Cmdlet           Invoke-PolicyEvaluation Invoke-PolicyEvaluation...
Cmdlet           Invoke-Sqlcmd       Invoke-Sqlcmd [[-Query]...
```

W naszym przykładowym poleceniu podajemy, że wyświetlone powinny zostać tylko polecenia z przystawki `SqlServerCmdletSnapin100`, i w wynikach działania pojawiają się tylko dwa polecenia. Tak, to wszystko SQL Server dodaje w tej przystawce, ale warto zauważyć, że jedno z tych poleceń jest w stanie wykonywać zapytania Transact-SQL (T-SQL). Ponieważ przy pracy z bazami danych SQL Server za pomocą zapytań T-SQL

można wykonać prawie wszystko, tak naprawdę polecenie `cmdlet Invoke-Sqlcmd` umożliwia wykonanie praktycznie każdej operacji, jaka może nam być potrzebna.

Aby zobaczyć, czy przystawka dodała nowych dostawców `PSDrive`, powinniśmy uruchomić polecenie `Get-PSProvider`. Za pomocą tego polecenia `cmdlet` nie można wybrać określonej przystawki, zatem aby wykryć coś nowego, musisz wiedzieć, którzy dostawcy byli już dostępni w systemie przed zainstalowaniem przystawki. W naszym przypadku wyniki działania tego polecenia były następujące:

```
PS C:\> get-psprovider
```

Name	Capabilities	Drives
----	-----	-----
WSMan	Credentials	{WSMan}
Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess	{C, A, D}
Function	ShouldProcess	{Function}
Registry	ShouldProcess, Transa...	{HKLM, HKCU}
Variable	ShouldProcess	{Variable}
Certificate	ShouldProcess	{cert}

Nie widać tutaj niczego nowego. Nie jesteśmy tym zaskoczeni, ponieważ nowo zainstalowana przystawka nazywa się `SqlServerCmdletSnapin100`. Przypominamy, że na liście dostępnych przystawek znajdował się również `PSSnapin` o nazwie `SqlServerProviderSnapin100`, co sugeruje, że deweloperzy SQL Server z jakiegoś powodu osobno zapakowali `cmdlety` i dostawcę `PSDrive`. Spróbujmy zatem dodać również tę drugą przystawkę:

```
PS C:\> add-pssnapin sqlserverprovidersnapin100
PS C:\> get-psprovider
```

Name	Capabilities	Drives
----	-----	-----
WSMan	Credentials	{WSMan}
Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess	{C, A, D}
Function	ShouldProcess	{Function}
Registry	ShouldProcess, Transa...	{HKLM, HKCU}
Variable	ShouldProcess	{Variable}
Certificate	ShouldProcess	{cert}
SqlServer	Credentials	{SQLSERVER}

Przeglądając wyniki działania, widzimy, że do naszej powłoki został dodany dysk `PSDrive` o nazwie `SQLSERVER`: udostępniany przez dostawcę `SqlServer`. Dodanie tego nowego dysku oznacza, że możemy przejść na nowy dysk za pomocą polecenia `cd sqlserver:` i prawdopodobnie rozpocząć eksplorowanie bazy danych.

7.4. Rozszerzenia — wyszukiwanie i dodawanie modułów

Powłoka PowerShell v3 (a także v2) obsługuje drugi typ rozszerzeń, nazywanych **modułami**. Moduły są zaprojektowane tak, aby były nieco bardziej samodzielne i nieco łatwiejsze do dystrybucji. Pomimo tego, że moduły działają podobnie do przystawek

PSSnapin, aby je odnaleźć i efektywnie z nich korzystać, musisz się dowiedzieć o nich czegoś więcej.

Moduły nie wymagają jakiegoś złożonego procesu rejestracji. Zamiast tego program PowerShell automatycznie przeszukuje określony zestaw ścieżek w celu znalezienia dostępnych modułów. Zmienna środowiskowa `PSModulePath` definiuje ścieżki, w których PowerShell oczekuje obecności modułów:

```
PS C:\> get-content env:psmodulepath
C:\Users\Administrator\Documents\WindowsPowerShell\Modules;C:\Windows\system32\
WindowsPowerShell\v1.0\Modules\
```

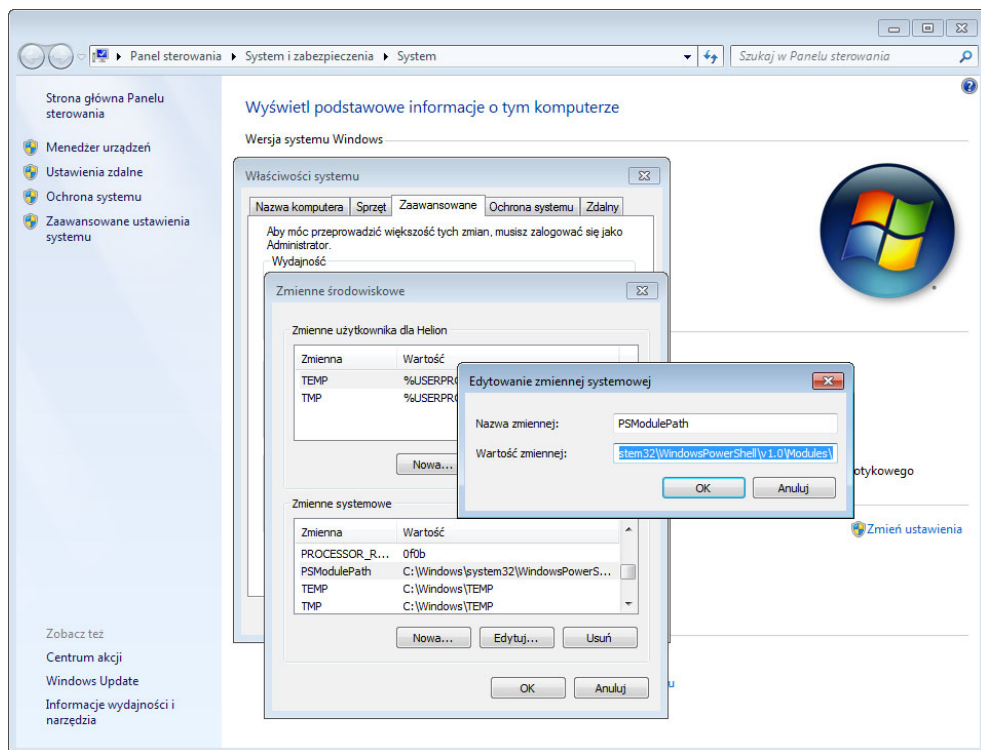
Jak widać w tym przykładzie, istnieją dwie domyślne lokalizacje dla modułów powłoki PowerShell: jedna — w podkatalogu systemu operacyjnego, gdzie znajdują się moduły systemowe, a druga — w folderze *Dokumenty*, gdzie możesz umieszczać dowolne własne moduły. Jeśli używasz nowszych wersji powłoki PowerShell, możesz znaleźć tutaj również inne lokalizacje, które obecnie wykorzystuje Microsoft. W razie potrzeby możesz także dodawać moduły z dowolnych innych lokalizacji, pod warunkiem, że znasz ich pełne ścieżki.

UWAGA Zmiennej `PSModulePath` nie możesz modyfikować w samej powłoce PowerShell; jest to część środowiska Twojego systemu Windows. Zawartość tej zmiennej możesz zmienić z poziomu panelu sterowania systemem lub ustawić za pomocą *Group Policy* (zasady grupy). Niektóre produkty firmy Microsoft lub innych dostawców oprogramowania również mogą modyfikować zawartość tej zmiennej.

Ścieżki zapisane w zmiennej `PSModulePath` są bardzo ważne dla powłoki PowerShell v3. Jeżeli masz jakieś moduły, które z takiego czy innego powodu zostały umieszczone w innym miejscu, powinieneś dodać ich ścieżki do zmiennej środowiskowej `PSModulePath`. Rysunek 7.1 pokazuje, jak to zrobić z poziomu panelu sterowania systemem Windows, a nie z poziomu powłoki PowerShell.

Dlaczego zmienna `PSModulePath` jest tak ważna? Dzięki niej powłoka PowerShell może automatycznie zlokalizować wszystkie moduły zainstalowane w Twoim systemie. Po przeszukaniu ścieżek powłoka PowerShell automatycznie rozpoznaje dostępne moduły. Będzie wyglądało tak, jakby wszystkie dostępne moduły były przez cały czas załadowane. Jeżeli spróbujesz wyświetlić zawartość pomocy dla jakiegoś wybranego modułu, to takie polecenie zostanie wykonane natychmiast, bez konieczności ładowania tego modułu. Jeżeli uruchomisz jakieś znalezione tam polecenie, to powłoka PowerShell automatycznie załaduje odpowiedni moduł je zawierający. Polecenie `Update-Help` powłoki PowerShell również wykorzystuje zmienną `PSModulePath` do wykrywania zainstalowanych modułów, a następnie dla każdego z nich poszukuje zaktualizowanych plików pomocy.

Na przykład spróbuj uruchomić polecenie `Get-Module | Remove-Module`, aby usunąć wszystkie załadowane moduły. Następnie uruchom polecenie przedstawione poniżej (Twoje wyniki mogą nieznacznie różnić się w zależności od tego, której wersji systemu Windows używasz na swoim komputerze):



Rysunek 7.1. Zmiana zawartości zmiennej PSModulePath

```
PS C:\> help *network*
```

Name	Category	Module
Get-BCNetworkConfiguration	Function	BranchCache
Get-DtcNetworkSetting	Function	MsDtc
Set-DtcNetworkSetting	Function	MsDtc
Get-SmbServerNetworkInterface	Function	SmbShare
Get-SmbClientNetworkInterface	Function	SmbShare

Jak widać, PowerShell wyświetla kilka poleceń (funkcji), które w swoich nazwach mają słowo network. Znając nazwę polecenia, możesz poprosić o wyświetlenie treści pomocy dla dowolnego z nich, nawet jeżeli odpowiedni moduł nie został załadowany:

```
PS C:\> help Get-SmbServerNetworkInterface
```

NAME

Get-SmbServerNetworkInterface

SYNTAX

```
Get-SmbServerNetworkInterface [-CimSession <CimSession[]>]
[-ThrottleLimit <int>] [-AsJob] [<CommonParameters>]
```

Jeżeli chcesz, możesz nawet uruchomić takie polecenie, a powłoka PowerShell automatycznie upewni się, że odpowiedni moduł został załadowany. Ta funkcja automatycznego

wykrywania i ładowania modułów jest bardzo przydatna i pomaga w wyszukiwaniu i używaniu poleceń, które nie są natywnie dostępne w powłoce podczas jej uruchamiania.

WSKAZÓWKA Polecenia `Get-Module` możesz także użyć do pobrania listy modułów dostępnych na komputerze zdalnym, a następnie użyć polecenia `Import-Module` do załadowania wybranego modułu zdalnego do bieżącej sesji powłoki PowerShell. W rozdziale 13. dowiesz się, jak to zrobić.

Mechanizm automatycznego wykrywania modułów pozwala powłoce PowerShell na dopełnianie nazw poleceń (przy użyciu klawisza *Tab* w konsoli lub mechanizmu IntelliSense w środowisku ISE), wyświetlanie pomocy i uruchamianie poleceń, nawet tych znajdujących się w modułach, które nie zostały jawnie załadowane do pamięci. Jak widać, z tych i wielu innych powodów warto się upewnić, że zawartość zmiennej `PSModulePath` jest kompletna i aktualna.

A co powinieneś zrobić, jeżeli dany moduł nie znajduje się w jednej ze ścieżek zapisanych w zmiennej `PSModulePath`? W takiej sytuacji możesz użyć polecenia `Import-Module` i jako parametr wywołania podać pełną ścieżkę do modułu, na przykład `C:\MojeProgramy\Narzędzia\MójModuł`.

Jeżeli w menu *Start* znajduje się skrót prowadzący do powłoki przeznaczonej dla konkretnego produktu, na przykład SharePoint Server, i nie wiesz, w którym miejscu taki produkt zainstalował swój moduł rozszerzający funkcjonalność powłoki PowerShell, powinieneś otworzyć właściwości tego skrótu z menu *Start*. Jak już pokazaliśmy wcześniej w tym rozdziale, właściwość *Element docelowy* takiego skrótu będzie zawierać polecenie `Import-Module` użyte do załadowania modułu i właśnie tam znajdziesz nazwę modułu i ścieżkę.

Moduły mogą również dodawać dostawców `PSDrive`. Do identyfikacji nowych dostawców możesz użyć tej samej techniki, którą zastosowałeś w pracy z przystawkami: uruchom polecenie `Get-PSProvider`.

7.5. Konflikty poleceń i usuwanie rozszerzeń

Przyjrzyj się poleceniom, które dodaliśmy dla SQL Server i Active Directory. Zauważyłeś coś szczególnego w nazwach poleceń?

Większość rozszerzeń powłoki PowerShell — serwer Exchange jest tutaj godnym uwagi wyjątkiem — dodaje do rzeczownika w nazwie polecenia krótki przedrostek, na przykład `Get-ADUser` lub `Invoke-SqlCmd`. Takie przedrostki mogą wydawać się nieco niezręczne, ale mają na celu zapobieganie konfliktom poleceń.

Żalóźmy przykładowo, że mamy dwa moduły, z których każdy zawiera polecenie `Get-User`. Skoro po ich załadowaniu mamy dwa polecenia o tej samej nazwie, to który cmdlet zostanie wykonany po uruchomieniu polecenia `Get-User`? Otóż okazuje się, że ten, który został załadowany jako ostatni. Warto jednak zauważyć, że to drugie polecenie o tej samej nazwie nie jest całkowicie niedostępne. Aby konkretnie uruchomić takie polecenie, musisz użyć nieco niezręcznej konwencji nazewnictwa, która wymaga

podania zarówno nazwy przystawki, jak i nazwy samego polecenia. Jeżeli takie polecenie `Get-User` pochodzi z przystawki o nazwie `MyCoolPowerShellSnapin`, możesz je uruchomić za pomocą następującego wywołania:

```
MyCoolPowerShellSnapin\Get-User
```

To dużo pisania i dlatego firma Microsoft sugeruje rozwiązanie polegające na dodawaniu prefiksów specyficznych dla określonych produktów, takich jak `AD` lub `SQL`, do rzeczowników w nazwach poszczególnych poleceń. Zastosowanie odpowiednich prefiksów pomaga zapobiegać konfliktom i ułatwia identyfikację poleceń oraz ich stosowanie.

Jeżeli jednak znajdziesz się w takiej kłopotliwej sytuacji, zawsze możesz usunąć jedno z kolidujących rozszerzeń. Aby usunąć wybrane rozszerzenie, powinieneś uruchomić polecenie `Remove-PSSnapin` lub `Remove-Module` i jako parametr wywołania podać nazwę przystawki lub modułu, który powinien zostać usunięty.

7.6. Jak to wygląda w systemach operacyjnych innych niż Windows

Jeśli korzystasz z systemu `Linux` lub `macOS`, musisz wziąć poprawkę na kilka zastrzeżeń dotyczących powyższej dyskusji. Po pierwsze, zmienna środowiskowa `PSModulePath`, mimo że istnieje, będzie wskazywać zupełnie gdzie indziej. Pamiętaj, aby przed rozpoczęciem pracy z powłoką sprawdzić, gdzie dany system szuka modułów.

Po drugie, wiele istniejących modułów rozszerzeń najzwyczajniej nie będzie działać na systemach innych niż `Windows`. Dzieje się tak, ponieważ takie moduły mogą polegać na funkcjonalności (na przykład `WMI`), która nie działa w innych systemach operacyjnych, lub mogą wymagać dostępności usług i innych zależności (takich jak połączenie z usługą `Active Directory`), które nie są dostępne dla systemu operacyjnego Twojego komputera. Ale to działa w obie strony — istnieje duże prawdopodobieństwo, że znajdziesz jakiś moduł rozszerzenia powłoki `PowerShell`, który nie będzie działał w systemie `Windows`, ale za to znakomicie będzie współpracować z wybranymi mechanizmami systemów `Linux` lub `macOS`.

7.7. Używanie nowego modułu

Spróbujemy teraz wykorzystać Twoją nowo odkrytą wiedzę. Zakładamy, że używasz jednej z najnowszych wersji systemu `Windows` (nie jest to `macOS` lub `Linux`), i chcemy, abyś postępował zgodnie z poleceniami, które przedstawimy w tym podrozdziale. Co ważniejsze, chcemy, abyś postępował zgodnie z procesem, który zaprezentujemy, ponieważ w ten sposób nauczysz się korzystać z nowych poleceń bez konieczności biegania do księgarni i kupowania nowych książek dla każdego z produktów, z którymi przyjdzie Ci pracować. W ćwiczeniach zamieszczonych na końcu tego rozdziału będziemy korzystać z tego samego procesu, by poznać zupełnie nowy zestaw poleceń.

Naszym zadaniem jest wyczyszczenie pamięci podręcznej mechanizmu `DNS` w naszym komputerze. Nie mamy pojęcia, czy powłoka `PowerShell` może to zrobić, więc zaczynamy od zapytania systemu pomocy o wskazówkę:

```
PS C:\> help *dns*
Name                                Category    Module
-----
dnsn                                Alias
Resolve-DnsName                    Cmdlet      DnsClient
Clear-DnsClientCache               Function    DnsClient
Get-DnsClient                      Function    DnsClient
Get-DnsClientCache                 Function    DnsClient
Get-DnsClientGlobalSetting         Function    DnsClient
Get-DnsClientServerAddress         Function    DnsClient
Register-DnsClient                 Function    DnsClient
Set-DnsClient                      Function    DnsClient
Set-DnsClientGlobalSetting         Function    DnsClient
Set-DnsClientServerAddress         Function    DnsClient
Add-DnsClientNrptRule              Function    DnsClient
Get-DnsClientNrptPolicy            Function    DnsClient
Get-DnsClientNrptGlobal            Function    DnsClient
Get-DnsClientNrptRule             Function    DnsClient
Remove-DnsClientNrptRule           Function    DnsClient
Set-DnsClientNrptGlobal            Function    DnsClient
Set-DnsClientNrptRule             Function    DnsClient
```

Aha! Jak widać, mamy do dyspozycji cały moduł o nazwie DnsClient. W wynikach działania możemy nawet znaleźć polecenie Clear-DnsClientCache, ale jesteśmy również ciekawi, jakie inne polecenia są dostępne. Aby się tego dowiedzieć, ładujemy moduł ręcznie i wyświetlamy dostępne w nim polecenia.

ZRÓB TO SAM Postępuj zgodnie z opisywanymi przez nas poleceniami. Jeżeli na Twoim komputerze moduł DnsClient nie jest dostępny, najprawdopodobniej używasz starszej wersji systemu Windows. Zastanów się nad migracją do nowszej wersji lub nawet zdobyciem wersji próbnej, którą będziesz mógł uruchomić w maszynie wirtualnej, co pozwoli Ci na kontynuowanie pracy z książką.

```
PS C:\> import-module -Name DnsClient
PS C:\> get-command -Module DnsClient
Capability    Name
-----
CIM           Add-DnsClientNrptRule
CIM           Clear-DnsClientCache
CIM           Get-DnsClient
CIM           Get-DnsClientCache
CIM           Get-DnsClientGlobalSetting
CIM           Get-DnsClientNrptGlobal
CIM           Get-DnsClientNrptPolicy
CIM           Get-DnsClientNrptRule
CIM           Get-DnsClientServerAddress
CIM           Register-DnsClient
CIM           Remove-DnsClientNrptRule
CIM           Set-DnsClient
CIM           Set-DnsClientGlobalSetting
CIM           Set-DnsClientNrptGlobal
CIM           Set-DnsClientNrptRule
CIM           Set-DnsClientServerAddress
Cmdlet       Resolve-DnsName
```

UWAGA Moglibyśmy tutaj od razu wyświetlić treść pomocy dla polecenia `Clear-DnsClientCache` lub nawet uruchomić to polecenie bezpośrednio. Powłoka PowerShell automatycznie załadowałaby dla nas moduł `DnsClient` w tle. Ze względu jednak na to, że poruszamy się na nieznanym gruncie, takie podejście pozwala nam zobaczyć pełną listę poleceń nowego modułu.

Lista dostępnych poleceń wygląda mniej więcej tak samo jak lista wyświetlona wcześniej. No dobrze, w takim razie zobaczmy, jak wygląda polecenie `Clear-DnsClientCache`:

```
PS C:\> help Clear-DnsClientCache
NAME
    Clear-DnsClientCache
SYNTAX
    Clear-DnsClientCache [-CimSession <CimSession[]>] [-ThrottleLimit
    <int>] [-AsJob] [-WhatIf] [-Confirm] [<CommonParameters>]
```

Składnia polecenia wydaje się prosta, a jego wywołanie nie wymaga podawania żadnych obligatoryjnych parametrów. Spróbujmy uruchomić polecenie:

```
PS C:\> Clear-DnsClientCache
```

Brak jakichkolwiek wiadomości jest zazwyczaj sam w sobie dobrą wiadomością. Mimo to z pewnością byłoby miło, gdybyśmy mogli zobaczyć, że nasze polecenie coś zrobiło. Spróbujmy zatem wywołać je z użyciem dodatkowego parametru `-verbose`:

```
PS C:\> Clear-DnsClientCache -verbose
VERBOSE: The specified name resolution records cached on this machine will
be removed.
Subsequent name resolutions may return up-to-date information.
```

Przełącznik `-verbose` jest dostępny dla wszystkich poleceń, chociaż jego dodanie w wierszu wywołania nie zawsze przynosi jakieś efekty. W naszym przypadku otrzymujemy komunikat informujący o tym, co się wydarzyło, a to nam potwierdza, że uruchomienie polecenia przyniosło jakieś rezultaty.

7.8. Skrypty profili powłoki PowerShell — wstępne ładowanie rozszerzeń podczas uruchamiania powłoki

Żałujemy, że uruchomiłeś powłokę PowerShell i załadowałeś kilka ulubionych przystawek i modułów. Wykonanie takiego zadania wymaga uruchomienia osobnego polecenia dla każdej przystawki lub modułu, który chcesz załadować, co przy kilku różnych rozszerzeniach może zająć całkiem ładną chwilę. Kiedy zakończysz pracę z powłoką, zazwyczaj zamykasz jej okno. Przy następnym uruchomieniu powłoka znowu będzie znajdowała się w „dziewiczym” stanie i ponowne załadowanie przystawek i modułów będzie wymagało powtórnego wykonania wspomnianej wcześniej sekwencji poleceń. Okropne. Musi istnieć lepszy sposób.

W rzeczywistości pokażemy Ci trzy lepsze sposoby. Pierwszy z nich polega na utworzeniu **pliku konsoli** (ang. *console file*), który zapamiętuje tylko załadowane przy-

stawki PSSnapin i nie będzie działać z żadnymi modułami. Rozpocznij od załadowania wszystkich pożądaných przystawek, a potem uruchom następujące polecenie:

```
Export-Console c:\myshell.psc
```

Wykonanie tego polecenia spowoduje utworzenie małego pliku w formacie XML, zawierającego listę przystawek załadowanych do powłoki.

Następnie powinienesz gdzieś utworzyć skrót do nowej konfiguracji powłoki PowerShell. Element docelowy tego skrótu powinien mieć następującą postać:

```
%windir%\system32\WindowsPowerShell\v1.0\powershell.exe -noexit -psconsolefile c:\myshell.psc
```

Kiedy użyjesz tego skrótu do otwarcia nowej sesji powłoki PowerShell, po uruchomieniu powłoka automatycznie załaduje wszystkie przystawki wymienione w pliku konsoli. Pamiętaj, że moduły nie są tutaj uwzględniane. A co powinienesz zrobić, jeżeli korzystasz zarówno z przystawek, jak i z modułów lub z samych modułów?

WSKAZÓWKA Powinienesz pamiętać, że powłoka PowerShell automatycznie ładuje moduły znajdujące się w lokalizacjach wskazywanych przez zmienną `PSModulePath`. Procedura opisana poniżej może być przydatna tylko wtedy, kiedy chcesz wstępnie załadować moduły, które znajdują się w lokalizacjach innych niż wskazywane przez zmienną `PSModulePath`.

Rozwiązaniem takiego problemu jest użycie **skryptu profilu** (ang. *profile script*). O skryptach profilów wspominaliśmy już wcześniej, bardziej szczegółowo omówimy je w rozdziale 25., a na razie, jeżeli chcesz dowiedzieć się, jak z nich korzystać w systemie Windows, powinienesz po prostu wykonać kroki opisane poniżej:

1. W folderze *Dokumenty* utwórz nowy folder o nazwie *WindowsPowerShell* (bez spacji w nazwie folderu; w systemach operacyjnych innych niż Windows ta ścieżka może być inna, ale możesz ją wyświetlić, uruchamiając z poziomu konsoli powłoki polecenie `$profile`).
2. Użyj programu Notatnik do utworzenia w tym folderze pliku o nazwie *profile.ps1*. Gdy będziesz zapisywał plik z poziomu Notatnika, pamiętaj o umieszczeniu nazwy pliku w znakach cudzysłowu ("*profil.ps1*"). Użycie cudzysłowu zapobiega dodawaniu przez Notatnik rozszerzenia pliku *.txt*. Jest to o tyle ważne, że jeżeli rozszerzenie *.txt* zostanie dodane, opisana niżej sztuczka po prostu nie zadziała.
3. W nowo utworzonym pliku tekstowym wpisz polecenia `Add-PSSnapin` oraz `Import-Module`, których zadaniem będzie załadowanie odpowiednich przystawek i modułów. Każde z poleceń powinienesz umieścić w osobnym wierszu.
4. Powróć do konsoli powłoki PowerShell, gdzie musisz włączyć wykonywanie skryptów (domyślnie jest wyłączone). W rozdziale 17. dowiesz się więcej na temat tego, jakie to może mieć konsekwencje związane z bezpieczeństwem systemu, ale na razie zakładamy po prostu, że wykonujesz tę operację na samodzielnej maszynie wirtualnej lub na samodzielnym komputerze testowym, gdzie względy bezpieczeństwa mają nieco mniejsze znaczenie. Z poziomu wiersza

polecen powłoki uruchom polecenie `Set-ExecutionPolicy RemoteSigned`. Zwróć uwagę, że polecenie będzie działać tylko wtedy, kiedy pracujesz z systemem Windows i uruchomiłeś powłokę na prawach użytkownika *Administrator*. Ustawienia tej właściwości możesz również zmieniać za pośrednictwem zasad grupy — jeśli tak zrobisz, otrzymasz odpowiedni komunikat ostrzegawczy.

5. Przy założeniu, że jak dotąd nie natknąłeś się na żadne błędy ani ostrzeżenie, możesz zamknąć i ponownie uruchomić powłokę PowerShell, która automatycznie odczyta zawartość pliku *profile.ps1* i wykona umieszczone w nim polecenia, ładując Twoje ulubione przystawki i moduły.

ZRÓB TO SAM Nawet jeżeli nie masz jeszcze swojej ulubionej przystawki lub modułu, utworzenie takiego prostego profilu jest dobrą praktyką. Jeżeli nie masz nic więcej, możesz po prostu umieścić w tym pliku polecenie `cd \`, tak aby powłoka po uruchomieniu zawsze automatycznie przechodziła do głównego katalogu dysku systemowego. Nie powinieneś jeszcze tego robić na komputerze będącym częścią produkcyjnej sieci Twojej firmy, ponieważ nie omówiliśmy jeszcze wszystkich niezbędnych implikacji bezpieczeństwa.

7.9. Pobieranie modułów z Internetu

Firma Microsoft wprowadziła nowy moduł o nazwie *PowerShellGet*, który ułatwia wyszukiwanie, pobieranie, instalowanie i aktualizowanie modułów z repozytoriów sieciowych. Moduł *PowerShellGet* jest bardzo podobny do menedżerów pakietów, które tak bardzo kochają administratorzy systemów Linux, takich jak *rpm*, *yum* czy *apt-get*. Firma Microsoft prowadzi nawet swego rodzaju galerię sieciową lub inaczej mówiąc, repozytorium zasobów powłoki PowerShell, o nazwie *PowerShell Gallery* (zobacz <http://powershellgallery.com>).

OSTRZEŻENIE To, że firma Microsoft prowadzi takie repozytorium, nie oznacza wcale, że to firma Microsoft tworzy, weryfikuje i wspiera udostępniane w nim pakiety. Galeria PowerShell zawiera moduły dodawane przez szeroką społeczność użytkowników, stąd zawsze, uruchamiając kod napisany przez kogoś innego, powinieneś zachować ostrożność.

Moduł *PowerShellGet* jest dostarczany wraz z powłoką PowerShell v5 i nowszymi wersjami (choć domyślnie może nie być obecny w systemach operacyjnych innych niż Windows), więc jeżeli pracujesz z taką wersją powłoki PowerShell (co możesz sprawdzić, wyświetlając `$PSVersionTable`), powinieneś mieć do niego dostęp. Na stronie <http://powershellgallery.com> znajdziesz łącze pozwalające na pobranie modułu *PowerShellGet*, który można zainstalować w systemach Windows 7 Service Pack 1 i nowszych lub Windows Server 2008 R2 z dodatkiem Service Pack 1 i nowszych. Aby moduł działał poprawnie, w Twoim systemie musi być również zainstalowana odpowiednia wersja środowiska .NET Framework; szczegółowe wymagania systemowe znajdziesz na stronie repozytorium.

Używanie modułu PowerShellGet jest łatwe i może być nawet zabawne:

- Uruchom polecenie `Register-PSRepository`, aby ustawić adres URL repozytorium. Strona <http://powershellgallery.com> jest zwykle skonfigurowana domyślnie, ale w razie potrzeby możesz nawet utworzyć swoje własne, prywatne repozytorium i wskazać je za pomocą odpowiedniego polecenia `Register-PSRepository`.
- Użyj polecenia `Find-Module` do wyszukiwania modułów w repozytoriach. W nazwach modułów możesz stosować symbole wieloznaczne, definiować znaczniki i zawęzać wyniki wyszukiwania na wiele innych sposobów.
- Użyj polecenia `Install-Module`, aby pobrać i zainstalować wybrany, znaleziony moduł.
- Użyj polecenia `Update-Module`, aby upewnić się, że posiadasz najnowszą dostępną wersję modułu. Jeżeli tak nie jest, to za pomocą tego polecenia możesz pobrać i zainstalować jego najnowszą wersję.

Moduł PowerShellGet posiada również szereg innych poleceń (na stronie <http://powershellgallery.com> znajdziesz ich szczegółową dokumentację), ale używanie tego modułu zazwyczaj rozpoczynasz od poleceń wymienionych powyżej. Na przykład zainstaluj moduł PowerShellGet (jeśli nie używasz powłoki PowerShell v5) i następnie zainstaluj moduł EnhancedHTML2, którego autorem jest Don (na stronie <http://PowerShell.org> znajdziesz bezpłatną książkę *Creating HTML Reports in PowerShell*, opisującą ten moduł), lub moduł ISEScriptingGeek, którego autorem jest Jeff.

7.10. Najczęściej spotykane problemy

Początkujący użytkownicy, którzy nie mają zbyt wielu doświadczeń z powłoką PowerShell, zaczynając pracę z modułami i przystawkami, często popełniają jeden kardynalny błąd — nie korzystają z systemu pomocy bądź przeglądając zawartość pomocy dla określonego polecenia, nie używają parametrów `-example` lub `-full`.

Szczerze mówiąc, najlepszym sposobem nauczania się korzystania z danego polecenia jest przeglądanie i analizowanie przykładów dostępnych w systemie pomocy. Oczywiście doskonale zdajemy sobie sprawę z tego, że przewijanie listy zawierającej setki poleceń może być nieco nużącym i zniechęcającym zajęciem (na przykład Exchange Server dodaje ponad 400 nowych poleceń), ale pamiętaj, że użycie poleceń `Help` i `Get-Command` z odpowiednimi symbolami wieloznacznymi powinno znacznie ułatwić Ci zawężenie wyświetlanej listy poleceń do takich, które dotyczą interesujących Cię zagadnień. Krótko mówiąc, od tej chwili *korzystaj z systemu pomocy!*

7.11. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie komputer z zainstalowanym systemem Windows 7, Windows Server 2008 R2 lub nowszym oraz powłoką PowerShell w wersji 3 lub nowszej.

Jak zwykle przyjmujemy założenie, że Twój komputer testowy (lub maszyna wirtualna) działa pod kontrolą najnowszej wersji systemu Windows (klient lub serwer).

W tym ćwiczeniu masz przed sobą tylko jedno zadanie: uruchomić pakiet rozwiązywania problemów z siecią (ang. *network troubleshooting pack*). Gdy to zrobisz, zostaniesz poproszony o podanie InstanceID. Naciśnij klawisz *Enter*, uruchom kontrolę połączenia internetowego i poproś o pomoc w nawiązaniu połączenia z określoną stroną internetową. Jako adresu testowego użyj strony <https://www.pluralsight.com/browse/it-ops>. Mamy nadzieję, że po wykonaniu tego ćwiczenia otrzymasz komunikat *No problems were detected* (nie wykryto żadnych problemów), co będzie oznaczało, że udało Ci się pomyślnie przetestować połączenie sieciowe.

Aby wykonać to zadanie, musisz samodzielnie odszukać polecenie, za pomocą którego możesz pobrać pakiet do rozwiązywania problemów sieciowych, oraz kolejne polecenie, które pozwoli Ci uruchomić pobrany pakiet. Musisz również dowiedzieć się, gdzie znajdują się takie pakiety i jak się nazywają. Wszystkim, czego potrzebujesz do wykonania tego zadania, jest powłoka PowerShell oraz jej system pomocy. To już wszystko, co możemy Ci podpowiedzieć!

7.12. Odpowiedzi

Poniżej przedstawiamy przykładową sekwencję poleceń, za pomocą której możesz wykonać nasze zadanie:

- `get-module *trouble* -list,`
- `import-module TroubleshootingPack,`
- `get-command -Module TroubleshootingPack,`
- `help get-troubleshootingpack -full,`
- `help Invoke-TroubleshootingPack -full,`
- `dir C:\windows\diagnostics\system,`
- `$pack=get-troubleshootingpack C:\windows\diagnostics\system\Networking,`
- `Invoke-TroubleshootingPack $pack,`
- *Enter*,
- `1,`
- `2,`
- <https://www.pluralsight.com/browse/it-ops>.

Obiekty — dane pod inną nazwą

W tym rozdziale zrobimy coś innego. Używanie obiektów powłoki PowerShell może być jednym z najbardziej zagmatwanych zagadnień, które będziemy omawiać, ale jednocześnie jest jednym z najbardziej krytycznych pojęć, wpływającym na wszystko, co robisz w tej powłoce. Przez lata próbowaliśmy objaśniać to zagadnienie na wiele różnych sposobów, aż wreszcie zdecydowaliśmy się na kilka z nich, które wyjątkowo dobrze sprawdzają się dla szerokiego grona odbiorców. Jeśli masz jakieś doświadczenie w programowaniu i czujesz się komfortowo z pojęciem obiektów, możesz od razu przejść do podrozdziału 8.2. Jeśli nie masz takiego doświadczenia i wcześniej nigdy nie pracowałeś z obiektami, zacznij od podrozdziału 8.1. i po prostu przeczytaj cały rozdział.

8.1. Czym są obiekty?

Spróbuj z poziomu powłoki PowerShell uruchomić polecenie `Get-Process`. Powinieneś zobaczyć tabelę składającą się z kilku kolumn danych, ale tak naprawdę wyświetlane kolumny to ledwo muśnięcie bogactwa informacji o procesach, które są dla Ciebie dostępne. Każdy proces posiada całe mnóstwo przypisanych właściwości, takich jak nazwa komputera, uchwyt głównego okna, maksymalny rozmiar zestawu roboczego, kod zakończenia czy czas rozpoczęcia działania. W praktyce znajdziesz dla każdego procesu ponad 60 powiązanych z nim informacji. Dlaczego więc powłoka PowerShell wyświetla ich tak mało?

Odpowiedź jest prosta — *zdecydowana większość* komponentów, do których powłoka PowerShell ma dostęp, posiada znacznie więcej właściwości i powiązanych z nimi informacji, niż można w wygodny sposób wyświetlić na ekranie. Po wykonaniu dowolnego

polecenia, takiego jak `Get-Process`, `Get-Service` czy `Get-EventLog`, powłoka PowerShell tworzy w pamięci operacyjnej odpowiednią tabelę zawierającą wszystkie informacje o takich elementach. Przykładowo, dla polecenia `Get-Process` taka wynikowa tabela składa się z około 67 kolumn, po jednym wierszu dla każdego procesu działającego na Twoim komputerze. W każdej z kolumn przechowywany jest dany element informacji o procesie, taki jak wykorzystanie pamięci wirtualnej, wykorzystanie procesora, nazwa procesu czy identyfikator procesu. Następnie powłoka PowerShell sprawdza, czy określiłeś, które kolumny powinny zostać wyświetlone. Jeżeli tego nie zrobiłeś (a przecież nie pokazaliśmy Ci jeszcze, jak to działa), powłoka sprawdza plik konfiguracyjny dostarczony przez firmę Microsoft i wyświetla tylko te kolumny tabeli, które jej zdaniem chcesz domyślnie zobaczyć.

Jednym ze sposobów zobaczenia wszystkich kolumn jest użycie polecenia `Convert-To-HTML`:

```
Get-Process | ConvertTo-HTML | Out-File processes.html
```

Ten cmdlet nie bawi się w żadne filtrowanie kolumn. Zamiast tego tworzy plik HTML, w którym zamieszcza wszystkie dostępne kolumny danych. To jeden ze sposobów pozwalających zobaczyć całą zawartość tabeli.

Oprócz kolumn danych każdy wiersz tabeli zawiera powiązane z nim akcje, które obejmują to wszystko, co system operacyjny może zrobić z procesem opisanym w danym wierszu tabeli. Przykładowo, system operacyjny może między innymi zamknąć proces, zabić go, odświeżyć jego informacje lub czekać, aż proces zakończy działanie.

Za każdym razem, gdy uruchomisz polecenie, które tworzy dane wyjściowe, wyniki działania przyjmują postać tabeli przechowywanej w pamięci komputera. Gdy przekazujesz wyniki działania z jednego polecenia do drugiego, tak jak to zostało pokazane poniżej:

```
Get-Process | ConvertTo-HTML
```

cała tabela danych jest przesyłana za pomocą potoku z wyjścia pierwszego polecenia na wejście drugiego. Tabela nie jest filtrowana do mniejszej liczby kolumn, dopóki nie zostaną wykonane wszystkie polecenia.

Teraz przyszedł czas na małą zmianę terminologii. Powłoka PowerShell nie odwołuje się do takiej tabeli w pamięci jako do *tabeli*. Zamiast tego używa następujących określeń:

- **Obiekt** (ang. *object*) — to właśnie do tej pory nazywaliśmy wierszem tabeli (ang. *table row*). Każdy obiekt reprezentuje jedną rzecz, na przykład pojedynczy proces lub pojedynczą usługę.
- **Właściwość** (ang. *property*) — to jest to, co nazywaliśmy kolumną tabeli. Właściwość reprezentuje jedną informację o danym obiekcie, taką jak nazwa procesu, identyfikator procesu lub stan usługi.
- **Metoda** (ang. *method*) — tak będziemy nazywać akcje. Każda metoda jest powiązana z pojedynczym obiektem i sprawia, że obiekt coś robi — na przykład zabija proces lub uruchamia usługę.
- **Kolekcja** (ang. *collection*) — kolekcja to cały zestaw obiektów lub inaczej mówiąc, to, co nazywaliśmy tabelą.

Jeżeli uważasz, że dyskusja na temat obiektów, którą zamieszczamy w kolejnych podrozdziałach, jest nieco myląca, wróć do tej czteropunktowej listy. Spróbuj zawsze wyobrażać sobie *zbiór* obiektów jako dużą tablicę informacji w pamięci, gdzie poszczególne kolumny tabeli są *właścivościami* obiektów, a poszczególne *obiekty* są reprezentowane przez kolejne wiersze tabeli.

8.2. Dlaczego powłoka PowerShell używa obiektów?

Jednym z powodów, dla których powłoka PowerShell używa obiektów do reprezentowania danych, jest to, że takie dane muszą być *w jakiś sposób* reprezentowane, prawda? Powłoka PowerShell mogłaby przecież przechowywać dane w formacie takim jak XML, a równie dobrze jej twórcy mogli przecież zdecydować się na użycie zwykłych tabel tekstowych. Najwyraźniej jednak mieli jakieś konkretne powody, aby nie podążać tą drogą.

Pierwszym powodem jest to, że sam system Windows jest systemem operacyjnym zorientowanym obiektowo — a przynajmniej większość oprogramowania działającego w tym systemie jest zorientowana obiektowo. W takiej sytuacji wybór obiektów jako struktur do przechowywania danych wydaje się zupełnie oczywisty, ponieważ cały system operacyjny jest praktycznie na nich oparty.

Innym powodem używania obiektów jest to, że w ostatecznym rozrachunku znakomicie ułatwiają pracę i zapewniają większą elastyczność i możliwość przetwarzania danych. Na chwilę spróbujmy udawać, że wynikiem działania poleceń powłoki PowerShell nie są obiekty, a zamiast nich powłoka tworzy proste tabele tekstowe (prawdopodobnie właśnie tak na początku wyobrażałeś sobie jej sposób działania). Po uruchomieniu polecenia takiego jak `Get-Process` otrzymywalibyśmy zatem wyniki w postaci odpowiednio sformatowanego tekstu:

```
PS C:\> get-process
Handles  NPM(K)  PM(K)  WS(K)  VM(M)  CPU(s)  Id  ProcessName
-----  -
39       5       1876   4340   52      11.33   1920 conhost
31       4       792    2260   22      0.00    2460 conhost
29       4       828    2284   41      0.25    3192 conhost
574      12      1864   3896   43      1.30    316  csrss
181      13      5892   6348   59      9.14    356  csrss
306      29     13936  18312  139      4.36   1300 dfsrs
125      15      2528   6048   37      0.17   1756 dfssvc
5159    7329   85052  86436  118      1.80   1356 dns
```

A co, jeżeli chciałbyś zrobić coś innego z tymi informacjami? Na przykład chciałbyś zmodyfikować wszystkie procesy korzystające z okna konsoli `conhost`. Aby to zrobić, musisz trochę przefiltrować listę wyników. W powłoce systemów takich jak UNIX czy Linux możesz użyć do tego celu polecenia `grep`, przekazując mu mniej więcej takie żądanie: „Spójrz na tę tabelę. Zachowaj tylko te wiersze, w których kolumny od 58. do 64. zawierają ciąg znaków `conhost`. Usuń wszystkie pozostałe wiersze”. Wyniki działania takiego polecenia będą zawierały tylko procesy, które spełniają określony przez Ciebie warunek:

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
39	5	1876	4340	52	11.33	1920	conhost
31	4	792	2260	22	0.00	2460	conhost
29	4	828	2284	41	0.25	3192	conhost

Następnie przesyłasz otrzymane wyniki do innego polecenia i nakazujesz mu na przykład wyodrębnić z listy identyfikatorów procesów. Polecenie może brzmieć następująco: „Przejdź przez tabelę i pobierz dane z kolumn od 52. do 56., ale usuń pierwsze dwa wiersze (nagłówki tabeli)”. Wynik takiej operacji może być następujący:

```
1920
2460
3192
```

Na koniec przesyłasz otrzymane wyniki do *kolejnego* polecenia, nakazując mu zabicie procesów (lub cokolwiek innego, co próbujesz zrobić) reprezentowanych przez te identyfikatory.

Dokładnie w taki sposób działają najczęściej administratorzy systemów UNIX i Linux. Spędzają masę czasu, doskonaląc sposoby przetwarzania tekstu za pomocą narzędzi takich jak `grep`, `awk` i `sed`, a także biegle posługując się wyrażeniami regularnymi (ang. *regular expressions*). Nabyte doświadczenia ułatwiają im definiowanie wzorców tekstowych, których mają szukać ich komputery. Użytkownicy systemów UNIX i Linux lubią języki programowania takie jak Perl, ponieważ te języki zawierają rozbudowane funkcje analizy tekstu i manipulacji nim.

Jednak to podejście oparte na przetwarzaniu tekstu stwarza też pewne problemy:

- Możesz poświęcić znacznie więcej czasu przetwarzaniu tekstu niż wykonywaniu swoich prawdziwych zadań.
- Jeżeli wyniki działania danego polecenia zmieniają się (na przykład kolumna `ProcessName` zostanie przeniesiona na początek tabeli), musisz zmodyfikować wszystkie swoje polecenia i skrypty, ponieważ w prosty sposób zależą one od takich elementów jak pozycje kolumn.
- Musisz być biegły w korzystaniu z języków programowania i narzędzi, które analizują tekst — nie dlatego, że Twoja praca wymaga analizowania tekstu, ale dlatego, że przetwarzanie tekstu jest środkiem do osiągnięcia zamierzonego celu.

Zastosowanie obiektów w powłoce PowerShell pozwoliło usunąć cały narzut związany z koniecznością przetwarzania tekstu. Ponieważ obiekty działają jak tablice w pamięci, nie musisz podawać powłoce PowerShell numeru kolumny, w której znajduje się dana informacja. Zamiast tego podajesz po prostu nazwę kolumny, a PowerShell dokładnie „wie”, dokąd się udać, aby pozyskać odpowiednie dane. Bez względu na to, jak ułożysz wyniki końcowe na ekranie lub w pliku, tablica danych w pamięci jest zawsze taka sama, więc nigdy nie musisz modyfikować poleceń z powodu zmiany położenia tej czy innej kolumny. Dzięki takiemu rozwiązaniu poświęcasz znacznie mniej czasu „dodatkowej obsłudze” zadania i możesz bardziej skupić się na tym, co chcesz osiągnąć.

To prawda, że aby efektywnie pracować z powłoką PowerShell, musisz nauczyć się pewnych elementów składni poleceń, ale jest tego *dużo mniej* niż wtedy, gdy pracujesz w innej, czysto tekstowej powłoce systemu Windows.

Nie daj się zwariować

Chcieliśmy nadmienić, że żaden z powyższych argumentów nie ma na celu atakowania ani krytykowania systemów takich jak Linux czy UNIX. Są to systemy operacyjne oparte na przetwarzaniu ciągów tekstu, zatem praca z tekstem ma dla nich sens. System Windows nie jest jednak systemem tekstowym. Jest to system operacyjny bazujący na modelach obiektowych i wykorzystujący interfejsy API, stąd powłoka PowerShell działa w systemie Windows w bardziej natywny sposób.

8.3. Odkrywanie obiektów — polecenie *Get-Member*

Jeśli obiekty są jak gigantyczny stół przechowywany w pamięci, a powłoka PowerShell pokazuje na ekranie tylko niewielki fragment tego stołu, to jak możesz sprawdzić, z czym jeszcze masz do czynienia? Jeżeli pomyślałeś, że powinieneś użyć polecenia *Help*, to bardzo się z tego powodu cieszymy, ponieważ z pewnością zauważyłeś, że w ostatnich kilku rozdziałach notorycznie próbowaliśmy Ci to wbić do głowy. Ale niestety tym razem się mylisz...

System pomocy opisuje jedynie podstawowe zagadnienia (w postaci tematów *about*) oraz dokumentuje składnię poleceń. Aby dowiedzieć się czegoś więcej o danym obiekcie, powinieneś użyć innego polecenia: *Get-Member*. Musisz nabrać dużej wprawy w posługiwaniu się tym poleceniem — tak dużej, że powinieneś nawet zacząć szukać krótszego sposobu jego wpisywania. Damy Ci zatem gotową odpowiedź: alias *Gm*.

Polecenia *Gm* możesz używać po dowolnym cmdlecie, który normalnie generuje wyniki działania. Przykładowo, wiesz już, że wykonanie polecenia *Get-Process* powoduje wyświetlenie wyników działania na ekranie, zatem możesz spróbować przekierować je za pomocą potoku do polecenia *Gm*, tak jak to zostało pokazane poniżej:

```
Get-Process | Gm
```

Za każdym razem, kiedy dany cmdlet tworzy kolekcję obiektów, tak jak robi to polecenie *Get-Process*, cała kolekcja pozostaje dostępna aż do końca działania potoku. Dopiero po wykonaniu wszystkich poleceń powłoka PowerShell filtruje kolumny danych i tworzy ostateczną postać wyników działania, które mają zostać wyświetlone na ekranie. Właśnie dlatego w poprzednim przykładzie polecenie *Gm* ma pełny dostęp do wszystkich właściwości i metod obiektów procesów, że nie zostały one jeszcze przefiltrowane do wyświetlenia. Polecenie *Gm* sprawdza każdy obiekt i tworzy listę właściwości i metod obiektów. Wygląda to mniej więcej tak:

```
PS C:\> get-process | gm
      TypeName: System.Diagnostics.Process

Name      MemberType      Definition
-----
Handles   AliasProperty   Handles = Handlecount
```

Name	AliasProperty	Name = ProcessName
NPM	AliasProperty	NPM = NonpagedSystemMemo...
PM	AliasProperty	PM = PagedMemorySize
VM	AliasProperty	VM = VirtualMemorySize
WS	AliasProperty	WS = WorkingSet
Disposed	Event	System.EventHandler Disp...
ErrorDataReceived	Event	System.Diagnostics.DataR...
Exited	Event	System.EventHandler Exit...
OutputDataReceived	Event	System.Diagnostics.DataR...
BeginErrorReadLine	Method	System.Void BeginErrorRe...
BeginOutputReadLine	Method	System.Void BeginOutputR...
CancelErrorRead	Method	System.Void CancelErrorR...
CancelOutputRead	Method	System.Void CancelOutput...

Oczywiście celowo nieco obcięliśmy powyższą listę, ponieważ jest zdecydowanie zbyt długa, aby ją prezentować w całości, ale mamy nadzieję, że wpadłeś już na to, o co tutaj chodzi.

ZRÓB TO SAM Nie wierz nam na słowo. To naprawdę idealny moment, abyś samodzielnie uruchamiał polecenia, które tutaj omawiamy, i analizował pełne wyniki ich działania.

A tak przy okazji, może Cię zainteresować fakt, że wszystkie właściwości, metody i inne rzeczy związane z obiektem są zbiorczo nazywane jego **elementami składowymi** (ang. *members*). To właśnie stąd polecenie `Get-Member` bierze swoją nazwę — wyświetla elementy składowe danego obiektu. Pamiętaj jednak: ponieważ konwencja nazewnictwa poleceń powłoki PowerShell wymaga stosowania rzeczowników w liczbie pojedynczej, nazwa tego cmdletu to *Get-Member*, a nie *Get-Members*.

WAŻNE Pamiętaj, aby zawsze zwracać uwagę na pierwszy wiersz wyników działania polecenia `Get-Member`, w którym to wierszu znajduje się wartość właściwości `TypeName`, czyli unikatowa nazwa przypisana do konkretnego typu obiektu. Co prawda wydaje się to niezbyt ważne — bo w końcu kogo to obchodzi — ale jak się sam przekonasz, taka informacja będzie miała kluczowe znaczenie już w kolejnym rozdziale.

8.4. Używanie atrybutów i właściwości obiektów

Przeglądając wyniki działania polecenia `Gm`, możesz zauważyć kilka rodzajów właściwości:

- `ScriptProperty`,
- `Property`,
- `NoteProperty`,
- `AliasProperty`.

Dla zainteresowanych

Normalnie obiekty w środowisku .NET Framework — z którego pochodzą wszystkie obiekty programu PowerShell — posiadają tylko *właściwości*. Powłoka PowerShell dynamicznie dodaje inne elementy składowe, takie jak *ScriptProperty*, *NoteProperty* czy *AliasProperty*. Jeżeli poszukasz informacji na temat obiektu danego typu w dokumentacji MSDN Microsoftu (możesz to zrobić, wpisując nazwę typu obiektu reprezentowaną przez *TypeName* do ulubionej wyszukiwarki sieciowej), to nie zobaczysz tam tych dodatkowych właściwości.

Powłoka PowerShell posiada mechanizm *Extensible Type System* (ETS), który jest odpowiedzialny za dynamiczne dodawanie takich właściwości. Dlaczego tak się dzieje? W niektórych sytuacjach ma to na celu zapewnienie większej spójności obiektów, na przykład poprzez dodanie właściwości *Name* do obiektów, które natywnie posiadają tylko właściwość taką jak *ProcessName* (właśnie do tego służy *AliasProperty*). W innych sytuacjach ma to na celu udostępnienie informacji, które są głęboko ukryte w czeluściach obiektu (obiekty procesów mają kilka właściwości *ScriptProperties*, które właśnie to robią).

Dzięki takiemu rozwiązaniu, kiedy pracujesz z powłoką PowerShell, wszystkie właściwości zachowują się w ten sam, spójny sposób. Nie powinieneś jednak się dziwić, jeżeli takie dodatkowe właściwości nie pojawiają się na oficjalnych stronach dokumentacji — powłoka dynamicznie dodaje takie właściwości, aby po prostu ułatwić Ci życie.

Z Twojego punktu widzenia wszystkie właściwości są takie same. Jedyna różnica między nimi polega na tym, jak zostały pierwotnie utworzone, ale z pewnością nie jest to coś, czym musiałbyś się przejmować. Najważniejsze jest to, że dla Ciebie wszystkie natywne i dodatkowe właściwości są po prostu właściwościami obiektu i używasz ich w taki sam sposób.

Właściwości zawsze posiadają przypisane wartości. Na przykład właściwość *ID* obiektu procesu może mieć wartość 1234, a właściwość *Name* tego samego obiektu może mieć wartość *Notepad*. Właściwości opisują określone charakterystyki obiektu: jego status, identyfikator, nazwę i tak dalej. W powłoce PowerShell właściwości są często przeznaczone tylko do odczytu, co oznacza chociażby, że nie możesz zmienić nazwy usługi, przypisując jej właściwości *Name* nową, arbitralnie wybraną wartość. Zamiast tego możesz pobrać nazwę tej usługi, odczytując jej właściwość *Name*. Bazując na naszych doświadczeniach, możemy oszacować, że około 90% tego, co robisz w powłoce PowerShell, będzie wymagało korzystania z takich czy innych właściwości.

8.5. Używanie akcji i metod obiektu

Wiele obiektów obsługuje jedną lub więcej metod. Jak wspominaliśmy już wcześniej, są one działaniami, które może wykonać dany obiekt. Przykładowo, obiekty procesów mają metodę *Kill*, która kończy działanie procesu. Niektóre metody wymagają podania jednego lub więcej argumentów wejściowych, które dostarczają dodatkowych informacji niezbędnych do wykonania konkretnej akcji, ale w początkowym etapie swojej przygody z powłoką PowerShell z pewnością nie będziesz miał z nimi do czynienia. Może się zdarzyć i tak, że spędzisz całe miesiące lub nawet lata, pracując z powłoką PowerShell, i nigdy nie będziesz musiał wywoływać żadnych metod obiektów. Jest to spowodowane tym, że wiele z takich działań jest również dostępnych za pośrednictwem cmdletów.

Na przykład jeżeli chcesz zakończyć działanie jakiegoś procesu, możesz to zrobić na trzy sposoby. Jednym z nich jest pobranie obiektu, a następnie wywołanie w jakiś sposób jego metody `Kill`. Innym sposobem może być użycie następującej sekwencji poleceń:

```
Get-Process -Name Notepad | Stop-Process
```

Taki sam rezultat możesz osiągnąć za pomocą jednego polecenia:

```
Stop-Process -name Notepad
```

W tej książce skupiamy się wyłącznie na wykonywaniu zadań przy użyciu poleceń *cmdlet* powłoki PowerShell. Takie podejście zapewnia najłatwiejszy i najbardziej przyjazny dla administratorów sposób postępowania, pozwalający się maksymalnie skupić na osiągnięciu zaplanowanego celu. Bezpośrednie korzystanie z metod zaczyna już zahaczać o programowanie w środowisku .NET Framework, co może być znacznie bardziej skomplikowane i może wymagać od użytkownika posiadania odpowiedniej wiedzy i doświadczenia w tej materii. Z tego powodu w tej książce bardzo rzadko — o ile w ogóle — będziesz miał okazję zobaczyć bezpośrednie wywołanie metody obiektu. Nasze podejście do tego zagadnienia można określić w następujący sposób: „Jeżeli nie możesz czegoś zrobić za pomocą polecenia *cmdlet*, wróć i skorzystaj z interfejsu GUI”. Zapewne nie będziesz musiał tak postępować przez całą swoją karierę, ale na razie jest to dobry sposób, aby skupić się na „powershellowym” sposobie wykonywania zadań.

Dla zainteresowanych

Na tym etapie edukacji nie będzie Ci to jeszcze potrzebne, ale mimo to warto wspomnieć, że oprócz właściwości i metod obiekty powłoki PowerShell mogą również posiadać przypisane zdarzenia. Zdarzenie to sposób, w jaki obiekt powiadamia Cię, że coś związanego z nim się wydarzyło. Przykładowo, obiekt procesu może po zakończeniu działania generować zdarzenie `Exited`. Do zdarzeń możesz dołączać własne polecenia, tak aby na przykład po zakończeniu działania określonego procesu automatycznie była wysyłana przygotowana wcześniej wiadomość e-mail. Taki sposób pracy ze zdarzeniami jest jednak bardzo zaawansowanym zagadnieniem, które daleko wykracza poza ramy tej książki.

8.6. Sortowanie obiektów

Większość poleceń *cmdlet* powłoki PowerShell tworzy obiekty w sposób deterministyczny, co oznacza, że przy każdym kolejnym uruchomieniu danego polecenia powłoka ma tendencję do tworzenia obiektów w tej samej kolejności. Przykładowo, zarówno usługi, jak i procesy są wymieniane w kolejności alfabetycznej według nazwy. Wpisy w dzienniku zdarzeń zwykle są wyświetlane w kolejności chronologicznej. A co powinieneś zrobić, jeżeli chcesz to zmienić?

Załóżmy, że chcesz wyświetlić listę procesów posortowaną tak, aby procesy będące największymi użytkownikami pamięci wirtualnej VM (ang. *virtual memory*) wyświetlały się na górze listy, a procesy zużywające jej najmniej — na dole listy. Musimy zatem w jakiś sposób na podstawie właściwości maszyny wirtualnej zmienić domyślną kolej-

ność listy obiektów. Powłoka PowerShell udostępnia prosty cmdlet, `Sort-Object`, który robi dokładnie to, o co nam chodziło:

```
Get-Process | Sort-Object -property VM
```

ZRÓB TO SAM Mamy nadzieję, że nadążasz za nami i uruchamiasz te same polecenia w swoim systemie. Nie zamieszczamy tutaj wyników działania tego polecenia, ponieważ wynikowe tabele są zbyt długie, ale jeżeli samodzielnie wykonasz takie polecenie, zobaczysz mniej więcej to samo na swoim ekranie.

Jest niezłe, ale wyniki działania powyższego polecenia nie są dokładnie tym, co chcieliśmy osiągnąć. Polecenie sortuje co prawda procesy według zużycia pamięci wirtualnej (właściwość `VM`), ale robi to w porządku rosnącym, z największymi wartościami na dole listy. Przywołując plik pomocy dla polecenia `Sort-Object`, możemy jednak zobaczyć, że ma ono parametr `-descending`, który powinien odwrócić kolejność sortowania. Zauważamy również, że parametr `-property` jest pozycjonowany, więc w wierszu polecenia nie musimy wpisywać samej nazwy tego parametru. W tym miejscu zdradzimy Ci również, że polecenie `Sort-Object` ma alias o nazwie `Sort`, a zatem w kolejnym przykładzie możesz zaoszczędzić sobie trochę pisania:

```
Get-Process | Sort VM -desc
```

Jak widać, skróciliśmy `-descending` do `-desc` i wreszcie mamy wyniki, które chcieliśmy osiągnąć. Parametr `-property` akceptuje wiele wartości (które na pewno widziałeś w pliku pomocy, o ile oczywiście tam zaglądałeś).

W przypadku gdy dwa procesy zużywają tę samą ilość pamięci wirtualnej, chcielibyśmy dodatkowo posortować je według identyfikatora procesu, co możemy zrobić przy użyciu następującego polecenia:

```
Get-Process | Sort VM, ID -desc
```

Jak zawsze, lista wartości oddzielonych od siebie przecinkami jest uniwersalnym sposobem przekazywania wielu wartości do dowolnego parametru wywołania polecenia.

8.7. Wybieranie żądanych właściwości

Kolejnym przydatnym poleceniem jest cmdlet `Select-Object`, który akceptuje obiekty z potoku i umożliwia określenie właściwości, które mają być wyświetlane. Pozwala to na uzyskanie dostępu do właściwości, które są domyślnie odfiltrowane przez reguły konfiguracji powłoki PowerShell, lub na przycięcie listy do kilku wybranych, interesujących Cię właściwości. Takie rozwiązanie może być bardzo przydatne podczas przekazywania obiektów do polecenia `ConvertTo-HTML`, ponieważ to polecenie domyślnie tworzy tabelę zawierającą wszystkie właściwości.

Porównaj wyniki działania tych dwóch poleceń:

```
Get-Process | ConvertTo-HTML | Out-File test1.html
```

```
Get-Process | Select-Object -property Name, ID, VM, PM | Convert-ToHTML | Out-File test2.html
```

ZRÓB TO SAM Wykonaj każde z tych poleceń osobno, a następnie otwórz wynikowe pliki HTML w przeglądarce sieciowej, żeby zobaczyć różnicę.

Rzuć okiem na plik pomocy dla polecenia `Select-Object` (zamiast nazwy polecenia możesz użyć aliasu `Select`). Parametr `-property` wydaje się pozycjonowany, co oznacza, że możemy skrócić nasze ostatnie polecenie do następującej postaci:

```
Get-Process | Select Name,ID,VM,PM | ConvertTo-HTML | Out-File test3.html
```

Spróbuj samodzielnie trochę poeksperymentować z poleceniem `Select-Object`. Wypróbuj różne warianty następującego polecenia, które pozwala wyświetlać dane wyjściowe na ekranie:

```
Get-Process | Select Name,ID,VM,PM
```

Spróbuj dodawać do tej listy i usuwać z niej różne właściwości obiektu procesu i sprawdź, jaki to ma wpływ na wyniki działania polecenia. Ile właściwości możesz dodać, tak aby wyniki działania nadal miały postać tabeli? Ile właściwości wymusza na powłoce PowerShell formatowanie wyników działania polecenia jako listy, a nie tabeli?

Dla zainteresowanych

Polecenie `Select-Object` posiada również parametry `-First` i `-Last`, które pozwalają wybierać podzbiór obiektów w potoku. Na przykład polecenie `Get-Process | Select -First 10` powoduje wybranie pierwszych 10 obiektów przekazanych z poprzedniego polecenia. Nie ma tutaj żadnych kryteriów pozwalających na selektywne wybieranie obiektów — zamiast tego polecenie wybiera po prostu pierwszych 10 (lub na przykład ostatnich 10) obiektów.

OSTRZEŻENIE Użytkownicy często mylą ze sobą dwa polecenia powłoki PowerShell: `Select-Object` i `Where-Object` (o tym drugim poleceniu jeszcze nie mówiliśmy). Polecenie `Select-Object` służy do wybierania właściwości obiektów (lub inaczej mówiąc, kolumn tabeli), które chcesz zobaczyć, a oprócz tego umożliwia wybieranie dowolnego podzbioru wierszy wyjściowych (przy użyciu opcji `-First` i `-Last`). Polecenie `Where-Object` usuwa lub filtruje obiekty z potoku na podstawie kryteriów zdefiniowanych przez użytkownika.

8.8. Obiekty aż do końca

Potok wyników poleceń powłoki PowerShell zawsze zawiera obiekty dopóty, dopóki nie zostanie wykonane ostatnie polecenie danej sekwencji. W tym momencie powłoka PowerShell sprawdza, jakie obiekty znajdują się w potoku, a następnie przegląda różne pliki konfiguracyjne, aby zobaczyć, których właściwości użyć do utworzenia wyników wyświetlanych na ekranie. W tym momencie powłoka na podstawie zestawu odpowiednich reguł i ustawień podejmuje również decyzję, czy wyniki działania będą wyświetlane na ekranie w postaci tabeli, czy listy (więcej szczegółowych informacji na temat tych reguł i ustawień oraz o tym, jak możesz je modyfikować, znajdziesz w rozdziale 10.).

Bardzo istotną sprawą jest to, że w czasie realizacji jednego wiersza poleceń potok może zawierać wiele rodzajów obiektów. W kilku następnych przykładach będziemy wypisywać po jednym poleceniu w każdym wierszu, dzięki czemu łatwiej będzie wyjaśnić, o czym mówimy.

Oto pierwszy przykład:

```
Get-Process |  
Sort-Object VM -descending |  
Out-File c:\procs.txt
```

W tym przykładzie rozpoczynamy od uruchomienia polecenia `Get-Process`, które umieszcza obiekty procesu w potoku. Następnym poleceniem jest `Sort-Object`. Uruchomienie tego polecenia nie zmienia samej zawartości potoku, a tylko kolejność znajdujących się w nim obiektów, zatem na wyjściu polecenia `Sort-Object` nadal znajdują się obiekty procesów. Ostatnie polecenie to `Out-File`. W tym przypadku powłoka PowerShell musi utworzyć jakieś dane wyjściowe, więc bierze wszystko, co znajduje się w potoku (czyli obiekty procesów), i formatuje to zgodnie z wewnętrznym zestawem reguł. Wyniki takiej operacji są zapisywane w określonym pliku.

Kolejny przykład jest nieco bardziej skomplikowany:

```
Get-Process |  
Sort-Object VM -descending |  
Select-Object Name,ID,VM
```

Działanie tego wiersza poleceń zaczyna się w ten sam sposób co w poprzednim przykładzie. Polecenie `Get-Process` umieszcza obiekty procesów w potoku, następnie są one przekazywane do polecenia `Sort-Object`, które je sortuje i przesyła do potoku na wyjściu. Ale polecenie `Select-Object` działa już nieco inaczej. Każdy obiekt procesów zawsze ma dokładnie takie same elementy składowe. Aby przyciąć listę właściwości dożądanego poziomu, polecenie `Select-Object` nie może usuwać niepotrzebnych właściwości, ponieważ wynik takiej operacji nie byłby już obiektem procesu. Zamiast tego polecenie `Select-Object` tworzy rodzaj nowego, niestandardowego obiektu o nazwie `PSObject`, do którego kopiuje żądane właściwości z obiektu procesu i w efekcie taki niestandardowy obiekt jest umieszczany w potoku.

ZRÓB TO SAM Spróbuj samodzielnie uruchomić powyższe polecenie, pamiętając oczywiście, żeby wpisać wszystkie polecenia składowe w jednym wierszu. Zastanów się, czym wyniki działania tego polecenia różnią się od normalnych wyników działania polecenia `Get-Process`?

Kiedy powłoka PowerShell dotrze do końca wiersza polecenia, musi zdecydować, w jaki sposób розміścić finalne wyniki działania. Ponieważ w potoku danych wyjściowych nie ma już żadnych obiektów procesów, powłoka PowerShell nie użyje domyślnych reguł i ustawień, które normalnie mają zastosowanie do przetwarzania obiektów. Zamiast tego powłoka szuka reguł dla obiektu `PSObject`, który obecnie znajduje się w potoku. Niestety firma Microsoft nie dostarcza żadnych reguł ani ustawień dla obiektów `PSObject`,

ponieważ z definicji mają one być używane do prezentowania niestandardowych wyników. W takiej sytuacji powłoka PowerShell przyjmuje swoje założenia domyślne i prezentuje wyniki w postaci tabeli, zgodnie z teorią, że trzy wspomniane wcześniej właściwości będą mogły zostać wyświetlone w tej postaci na ekranie. Wynikowa tabela nie jest tak ładnie rozplanowana jak normalne wyniki działania polecenia `Get-Process`, ponieważ powłoce brakuje dodatkowych informacji konfiguracyjnych potrzebnych do utworzenia niestandardowej tabeli.

Aby zobaczyć obiekty znajdujące się w danej w chwili w potoku, możesz użyć polecenia `Gm`. Pamiętaj, że polecenie to możesz dodać po każdym poleceniu `cmdlet`, które tworzy dane na wyjściu:

```
Get-Process | Sort VM -descending | gm
Get-Process | Sort VM -descending | Select Name, ID, VM | gm
```

ZRÓB TO SAM Spróbuj samodzielnie uruchomić te dwa polecenia i zwróć uwagę na różnicę w wynikach ich działania.

Zauważ, że w wynikach działania polecenia `Gm` powłoka PowerShell wyświetla nazwę typu obiektu, który widzi w potoku. W pierwszym przypadku jest to obiekt `System.Diagnostics.Process`, ale w przypadku drugiego polecenia potok zawiera inny rodzaj obiektów, które posiadają tylko trzy właściwości: `Name`, `ID` i `VM` oraz kilka dodatkowych elementów wygenerowanych przez system.

Nawet polecenie `Gm` tworzy obiekty i umieszcza je w potoku. Po uruchomieniu tego polecenia potok nie zawiera już obiektów procesów ani *wybranych* obiektów tymczasowych; zamiast tego zawiera obiekty utworzone przez polecenie `Gm`: `Microsoft.PowerShell.Commands.MemberDefinition`. Aby się o tym przekonać, powinieneś przekazać wyniki działania polecenia `Gm` do kolejnego polecenia `Gm`:

```
Get-Process | Gm | Gm
```

ZRÓB TO SAM Z pewnością będziesz chciał samodzielnie wypróbować takie polecenie, ale najpierw przemyśl dokładnie jego sposób działania, aby upewnić się, że ma to sens. Rozpoczynamy od uruchomienia polecenia `Get-Process`, które umieszcza w potoku obiekty procesów. Te trafiają do polecenia `Gm`, które je analizuje i tworzy własne obiekty typu `MemberDefinition`. Następnie są one przesyłane do kolejnego polecenia `Gm`, które je analizuje i generuje dane wyjściowe zawierające listę elementów składowych obiektów typu `MemberDefinition`.

Kluczem do opanowania powłoki PowerShell jest nauczenie się, jak śledzić obiekty znajdujące się w potoku w dowolnym miejscu wiersza polecenia. Polecenie `Gm` może Ci w tym pomóc, ale analizowanie „na sucho” zawartości wiersza polecenia to również dobre ćwiczenie, które może Ci ułatwić zrozumienie, co tam się dzieje.

8.9. Najczęściej spotykane problemy

Nasi studenci, rozpoczynając pracę z powłoką PowerShell, bardzo często popełniają kilka typowych błędów. Zdecydowana większość z nich jest spowodowana po prostu brakiem doświadczenia i po pewnym czasie już się nie pojawia, ale mimo to zdecydowaliśmy się umieścić je na poniższej liście, tak abyś mógł się szybciej zorientować, gdzie leży błąd, kiedy go popełnisz.

- Pamiętaj, że pliki pomocy powłoki PowerShell nie zawierają informacji o właściwościach obiektów. Aby wyświetlić listę właściwości, musisz za pomocą potoku przekazać takie obiekty do polecenia `Gm` (`Get-Member`).
- Pamiętaj, że polecenie `Gm` możesz umieścić na końcu każdego polecenia, które generuje wyniki działania. Przykładowo, polecenie takie jak `Get-Process -name Notepad | Stop-Process` zazwyczaj nie wyświetla żadnych rezultatów, więc umieszczanie sekwencji `| Gm` na jego końcu też niczego tutaj nie utworzy.
- Zwróć uwagę na staranne wpisywanie poleceń. Pamiętaj, aby po obu stronach znaku potoku umieszczać spacje, ponieważ przykładowy wiersz polecenia powinien być odczytywany jako `Get-Process | Gm`, a nie `Get-Process|Gm`. Klawisz spacji jest taki duży nie bez powodu — używaj go.
- Pamiętaj, że w zależności od miejsca w wierszu polecenia w potoku mogą znajdować się różne typy obiektów. Zastanów się, jaki typ obiektu znajduje się w danym miejscu, i skoncentruj się na tym, co następne polecenie zrobi z takim *typem* obiektu.

8.10. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

W tym rozdziale omawialiśmy wiele różnych zagadnień, które prawdopodobnie były znacznie bardziej rozbudowane i skomplikowane niż w jakimkolwiek wcześniejszym rozdziale. Mamy jednak nadzieję, że zrozumiałeś wszystko, co staraliśmy się przekazać, i że poniższe ćwiczenia pomogą Ci ugruntować nowo nabytą wiedzę. Niektóre z zadań opierają się na umiejętnościach, które zdobyłeś w poprzednich rozdziałach, co również pozwoli Ci odświeżyć pamięć i zachować czujność.

1. Znajdź polecenie, które tworzy liczbę losową.
2. Znajdź polecenie, które wyświetla bieżącą datę i godzinę.
3. Jakiego typu obiekt tworzy polecenie z zadania 2.? Jaka jest nazwa typu obiektu utworzonego przez to polecenie?
4. Używając cmdletu, który znalazłeś w zadaniu 2., oraz cmdletu `Select-Object` napisz polecenie, które będzie wyświetlało na ekranie bieżący dzień tygodnia w postaci tabeli zaprezentowanej poniżej. (Uwaga: wyniki działania będą wyrównane do prawej strony, a zatem upewnij się, że okno powłoki PowerShell nie ma poziomego paska przewijania):

```
DayOfWeek
```

```
-----
```

```
Monday
```

5. Znajdź polecenie wyświetlające informacje o poprawkach zainstalowanych w systemie Windows.
6. Za pomocą polecenia z zadania 5. wyświetl listę zainstalowanych poprawek. Następnie rozbuduj całe polecenie tak, aby posortować listę według daty instalacji i wyświetlić tylko datę instalacji, nazwę użytkownika, który zainstalował poprawkę, oraz identyfikator poprawki. Pamiętaj, że nagłówki kolumn wyświetlane w domyślnych wynikach działania polecenia niekoniecznie są prawdziwymi nazwami poszczególnych właściwości — aby mieć pewność, musisz samodzielnie sprawdzić nazwy wszystkich właściwości odpowiedniego obiektu.
7. Powtórz zadanie 6., ale tym razem posortuj wyniki według opisów poprawek i dołącz do nich opis, identyfikator poprawki oraz datę instalacji. Umieść wyniki w pliku HTML.
8. Wyświetl listę 50 najnowszych wpisów z dziennika zdarzeń Security (jeżeli dziennik Security jest pusty, możesz użyć innego dziennika zdarzeń, na przykład System lub Application). Posortuj listę tak, aby najstarsze wpisy pojawiały się jako pierwsze, a wpisy dokonane w tym samym czasie były dodatkowo posortowane według ich numerów indeksu. Wyświetl indeks, czas i źródło każdego wpisu. Zapisz te informacje w pliku tekstowym (pamiętaj: nie w pliku HTML, ale w pliku tekstowym). Być może będzie Cię kusiło, aby użyć polecenia `Select-Object` i jego parametrów `-First` albo `-Last`, ale nie rób tego. Istnieje lepszy sposób. Oprócz tego nie używaj również polecenia `Get-WinEvent`; z pewnością uda Ci się znaleźć znacznie lepsze polecenie do wykonania tego konkretnego zadania.

8.11. Odpowiedzi

1. `Get-Random`
2. `Get-Date`
3. `System.DateTime`
4. `Get-Date | select DayOfWeek`
5. `Get-Hotfix`
6. `Get-HotFix | Sort InstalledOn | Select InstalledOn,InstalledBy,HotFixID`
7. `Get-HotFix | Sort Description | Select Description,InstalledOn,InstalledBy,HotFixID | -ConvertTo-Html -Title "HotFix Report" | Out-File HotFixReport.htm`
8. `Get-EventLog -LogName System -Newest 50 | Sort TimeGenerated,Index | Select Index, TimeGenerated, -Source | Out-File elogs.txt`

Potoki — zaglądamy nieco głębiej

Do tej pory dzięki potokom powłoki PowerShell nauczyłeś się już całkiem efektywnie wykonywać nawet złożone operacje. Potokowanie poleceń (na przykład `Get-Process | Sort VM -desc | ConvertTo-HTML | Out-File procs.html`) to naprawdę potężny mechanizm, pozwalający na osiągnięcie w jednym wierszu polecenia tego, co zwykle wymagało napisania przynajmniej kilku wierszy kodu w skrypcie. Ale możesz to zrobić jeszcze lepiej. W tym rozdziale jeszcze głębiej zajrzemy do potoków i odkryjemy niektóre z ich najpotężniejszych możliwości.

9.1. Potoki — większe możliwości przy mniejszej liczbie poleceń

Jednym z powodów, dla których tak bardzo lubimy pracować z powłoką PowerShell, jest to, że pozwala nam być jeszcze bardziej efektywnymi administratorami systemów bez konieczności pisania skomplikowanych skryptów, jak to miało miejsce w przypadku języka VBScript. Klucz do ogromnych możliwości poleceń jednowierszowych leży jednak w sposobie działania potoków powłoki PowerShell.

Powiedzmy sobie jasno: możesz spokojnie pominąć ten rozdział i nadal pracować z powłoką PowerShell, ale w większości przypadków będziesz musiał uciekać się do pisania skryptów i programów w stylu języka VBScript. Choć praca z potokami powłoki PowerShell może być dosyć wymagająca, prawdopodobnie znacznie łatwiej się tego nauczyć niż umiejętności pisania bardziej skomplikowanych programów. Ucząc się operowania potokami, możesz być o wiele bardziej efektywny bez potrzeby pisania skryptów.

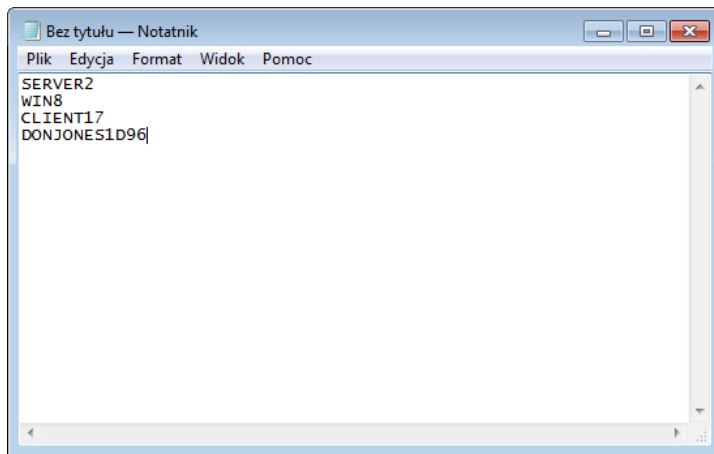
Cała idea polega na tym, aby spowodować, że powłoka wykona za Ciebie jeszcze więcej pracy przy użyciu jak najmniejszej liczby wpisywanych poleceń. Uważamy, że będziesz naprawdę zaskoczony, jak znakomicie powłoka PowerShell może sobie z tym poradzić!

9.2. Jak powłoka PowerShell przekazuje dane za pomocą potoku

Za każdym razem, kiedy łączysz dwa polecenia, PowerShell musi wiedzieć, jak przekazywać dane z wyjścia pierwszego polecenia na wejście drugiego polecenia. W nadchodzących przykładach pierwsze polecenie będziemy określali mianem *PolecenieA*. Będzie ono odpowiedzialne za tworzenie określonych wyników działania. Drugie polecenie to *PolecenieB*, które musi przyjąć wyniki działania polecenia A, a następnie wykonać na nich własne działania.

```
PS C:\> PolecenieA | PolecenieB
```

Załóżmy, że mamy plik tekstowy zawierający listę nazw komputerów, po jednej w każdym wierszu, tak jak to zostało pokazane na rysunku 9.1:



Rysunek 9.1. Tworzenie pliku tekstowego zawierającego listę nazw komputerów, po jednej nazwie w każdym wierszu

Nazw komputerów możemy użyć jako danych wejściowych określających, na których komputerach ma zostać uruchomione nasze polecenie. Rozważmy ten przykład:

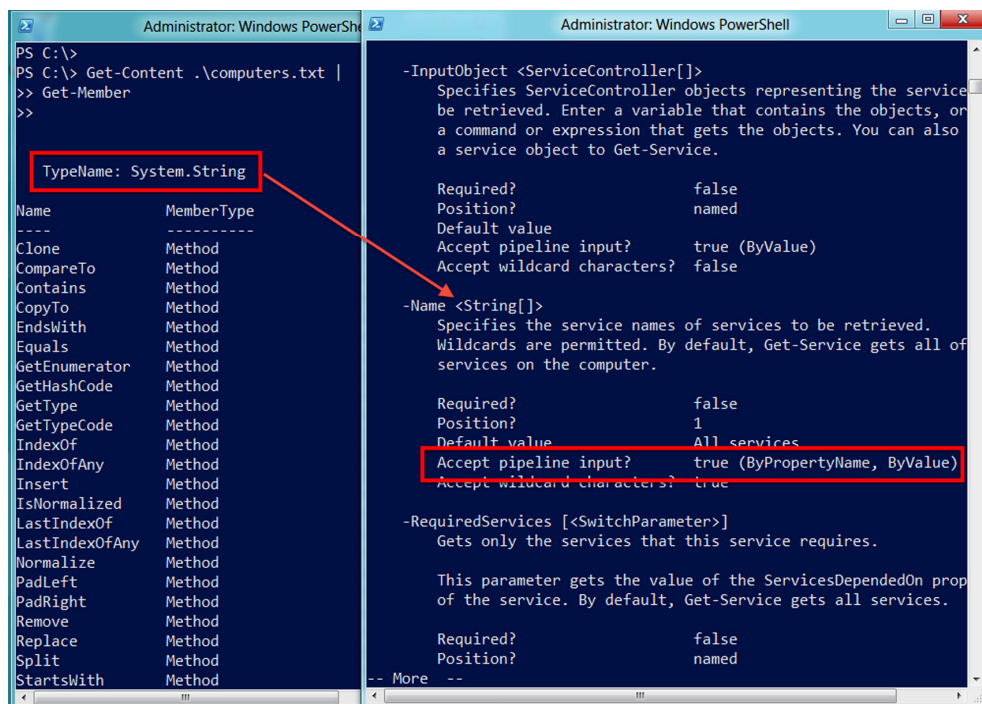
```
PS C:\> Get-Content .\computers.txt | Get-Service
```

Po uruchomieniu polecenia `Get-Content` nazwy komputerów zostają umieszczone w potoku. Następnie powłoka PowerShell musi zdecydować, w jaki sposób przekazać takie dane do polecenia `Get-Service`. Cała sztuczka polega na tym, że polecenia mogą akceptować dane wejściowe tylko dla swoich parametrów. Powłoka PowerShell musi

sprawdzić, który parametr polecenia Get-Service zaakceptuje wyniki działania polecenia Get-Content. Taki proces określania sposobu przekazywania danych między poleceniami jest nazywany **potokowym wiązaniem parametrów** (ang. *pipeline parameter binding*) i właśnie o tym będziemy mówić w tym rozdziale. Powłoka PowerShell posiada dwie metody przekazywania wyników działania polecenia Get-Content do parametrów polecenia Get-Service. Pierwszy sposób wiązania, którego spróbuje użyć powłoka, nazywa się ByValue (poprzez wartość), a jeśli to nie zadziała, powłoka użyje drugiego sposobu, o nazwie ByPropertyName (poprzez nazwę właściwości).

9.3. Plan A — przekazywanie danych ByValue

Korzystając z tego sposobu wiązania parametrów potoku, powłoka PowerShell sprawdzi typ obiektu tworzonego przez polecenie A i próbuje ustalić, czy dowolny parametr polecenia B może zaakceptować taki typ obiektu z potoku. Możesz to również sprawdzić samodzielnie. Najpierw przekaż wyniki działania polecenia A do polecenia Get-Member, aby zobaczyć, jaki typ obiektów jest tworzony przez polecenie A. Następnie wyświetl pełną zawartość pliku pomocy polecenia B (na przykład Help Get-Service -full) i sprawdź, czy jakiegokolwiek parametr tego polecenia może zaakceptować ByValue taki typ danych z potoku. Rysunek 9.2 pokazuje, co możesz odkryć.



Rysunek 9.2. Porównywanie wyników działania polecenia Get-Content z listą parametrów wejściowych polecenia Get-Service

Zobaczysz, że polecenie `Get-Content` tworzy obiekty typu `System.String` (lub po prostu `String`). Przekonasz się również, że polecenie `Get-Service` ma parametr, który akceptuje `ByValue` dane typu `String` z potoku. Problem polega na tym, że jest to parametr `-Name`, a on zgodnie z informacjami w pliku pomocy określa nazwy usług, które mają zostać pobrane (ang. *specifies the service names of services to be retrieved*). Nie tego chcemy — nasz plik tekstowy, a zatem nasze obiekty `String` to nazwy komputerów, a nie nazwy usług. Gdybyśmy uruchomili następujące polecenie:

```
PS C:\> Get-Content .\computers.txt | Get-Service
```

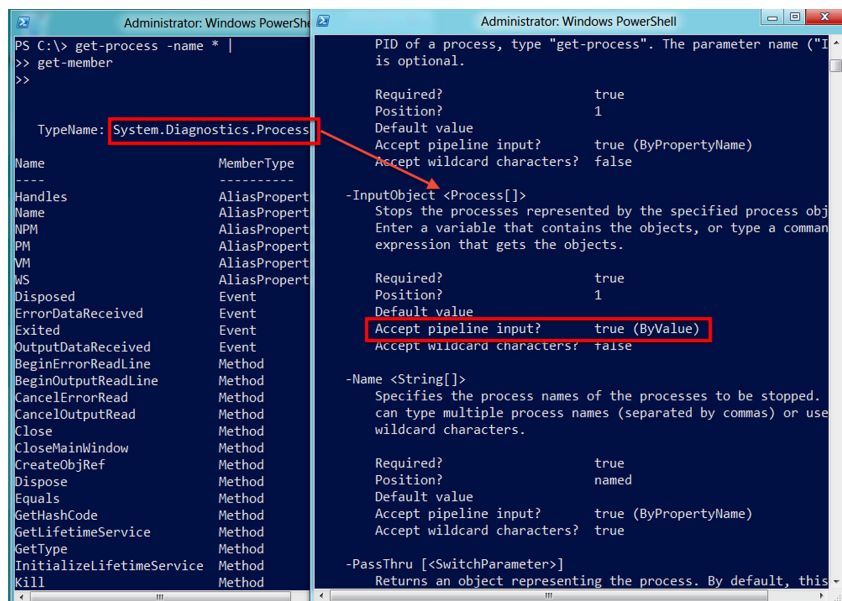
to powłoka próbowałaby pobierać usługi o nazwach `SERVER2`, `WIN8` i tak dalej, co prawdopodobnie nie zadziałałoby tak, jak tego oczekiwaliśmy.

Powłoka PowerShell pozwala tylko jednemu parametrowi zaakceptować `ByValue` dany typ obiektu z potoku. Ponieważ parametr `-Name` akceptuje `ByValue` dane typu `String` z potoku, żaden inny parametr nie może tego zrobić. To niestety niweczy nasze nadzieje na dokonanie próby przekazywania nazw komputerów z naszego pliku tekstowego do polecenia `Get-Service`.

Oczywiście w tym przypadku przekazywanie danych za pomocą potoku na wejście polecenia, technicznie rzecz biorąc, działa, ale po prostu nie przynosi oczekiwanych przez nas wyników. Rozważmy inny przykład, w którym osiągamy żądane rezultaty. Nasz wiersz poleceń wygląda tak:

```
PS C:\> get-process -name note* | Stop-Process
```

Przełącz wyniki działania polecenia `A` do polecenia `Get-Member` i sprawdź zawartość pliku pomocy dla polecenia `B`. Rysunek 9.3 pokazuje, co tam znajdziesz.



Rysunek 9.3. Wiązanie wyników działania polecenia `Get-Process` z parametrem polecenia `Stop-Process`

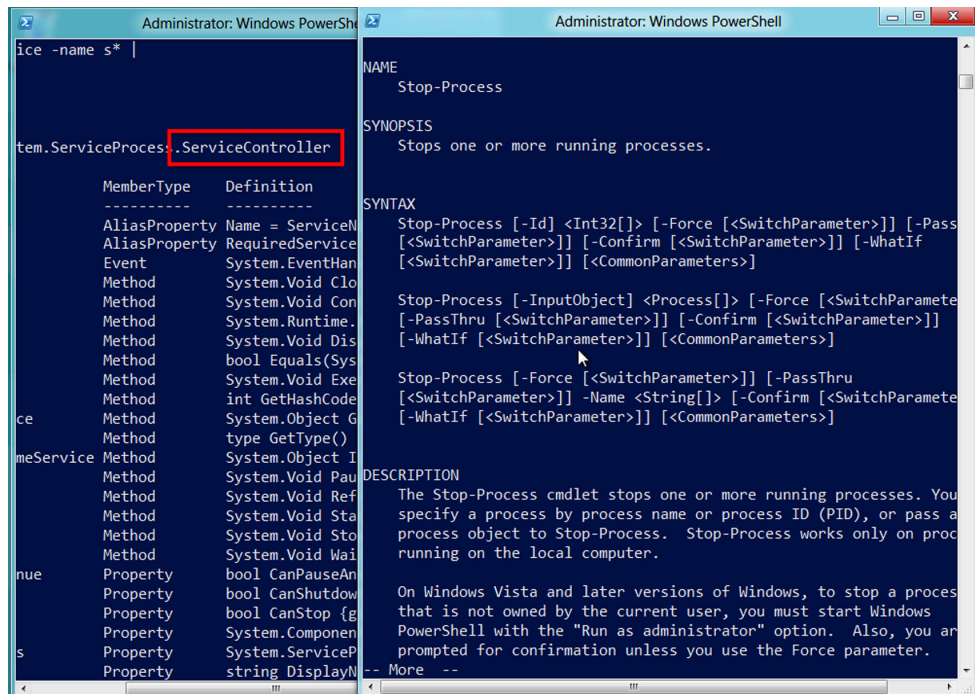
Polecenie `Get-Process` tworzy obiekty typu `System.Diagnostics.Process` (pamiętaj, że ograniczamy wyniki działania tego polecenia tylko do procesów, których nazwy zaczynają się od ciągu znaków `note*`; oprócz tego musimy się upewnić, że w naszym systemie została uruchomiona co najmniej jedna instancja programu Notepad, tak aby wykonanie polecenia wygenerowało jakieś wyniki działania). Polecenie `Stop-Process` może zaakceptować `ByValue` obiekty `Process` z potoku i przypisać je do swojego parametru `-InputObject`. Zgodnie z informacjami w pliku pomocy ten parametr pozwala na określenie procesów, które zostaną zatrzymane. Innymi słowy, polecenie A pobiera jeden lub więcej obiektów procesów, a polecenie B je zatrzymuje (lub *zabija*).

Jest to dobry przykład działania potokowego wiązania parametrów, który ilustruje także ważną właściwość powłoki PowerShell — w zdecydowanej większości przypadków polecenia posiadające w nazwie ten sam rzeczownik (jak na przykład `Get-Process` i `Stop-Process`) mogą za pomocą potoku przekazywać sobie wyniki działania `ByValue`.

Przedstawimy jeszcze jeden przykład:

```
PS C:\> get-service -name s* | stop-process
```

Na pierwszy rzut oka wydaje się to nie mieć żadnego sensu. Ale spróbujmy przeanalizować taką konstrukcję poprzez przekazanie wyników działania polecenia A do polecenia `Get-Member` i dokładną analizę zawartości pliku pomocy dla polecenia B. Na rysunku 9.4 pokazujemy, co powinniśmy znaleźć.

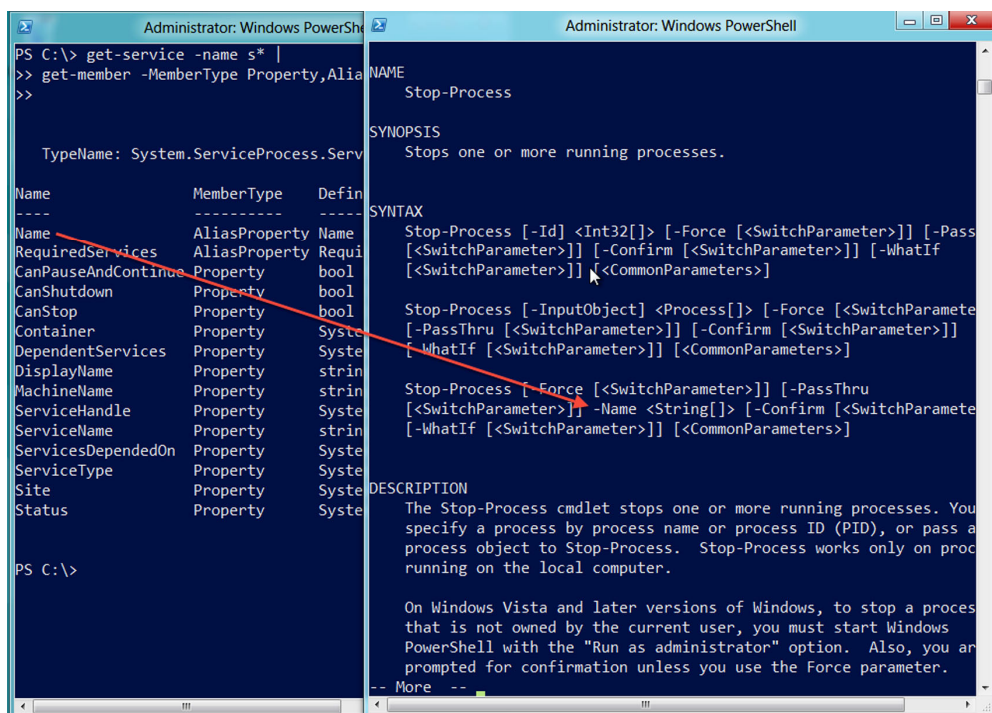


Rysunek 9.4. Analiza wyników działania polecenia `Get-Service` oraz parametrów wejściowych polecenia `Stop-Process`

Polecenie `Get-Service` tworzy obiekty typu `ServiceController` (technicznie rzecz biorąc, pełna nazwa takich obiektów to `System.ServiceProcess.ServiceController`, ale zazwyczaj można używać ostatniego elementu składowego jako skrótu nazwy typu obiektu). Niestety polecenie `Stop-Process` nie posiada ani jednego parametru, który może pobierać obiekty typu `ServiceController`. Jak widać, próba powiązania parametrów sposobem `ByValue` nie powiodła się, zatem powłoka PowerShell przechodzi do wypróbowania drugiego rozwiązania — wiązania parametrów `ByPropertyName`.

9.4. Plan B — przekazywanie danych `ByPropertyName`

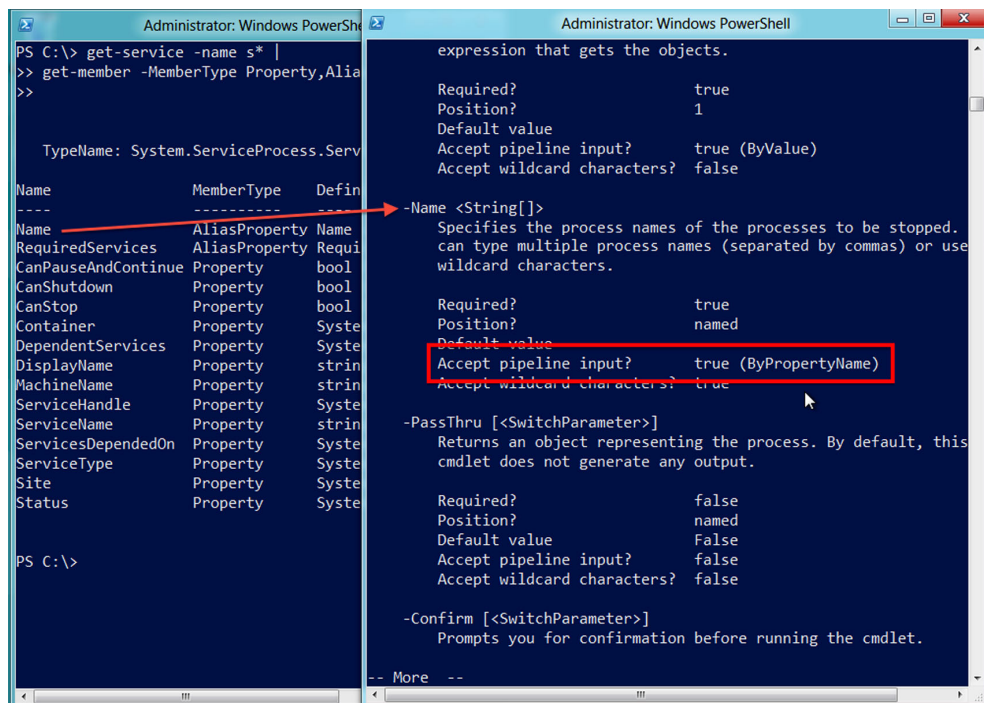
Podobnie jak w poprzednim przypadku, wiązanie `ByPropertyName` pozwala na przekazywanie wyników działania polecenia A bezpośrednio do parametrów wywołania polecenia B. Jednak wiązanie `ByPropertyName` jest nieco inne niż wiązanie `ByValue`. Dzięki tej metodzie możemy wiązać wiele parametrów polecenia B. Aby się o tym przekonać, ponownie uruchom polecenie A i prześlij wyniki jego działania do polecenia `Get-Member`, a następnie sprawdź w pliku pomocy składnię polecenia B. Rysunek 9.5 pokazuje, co powinieneś znaleźć — wyniki działania polecenia A mają jedną właściwość, której nazwa odpowiada parametrowi polecenia B.



Rysunek 9.5. Mapowanie właściwości do parametrów

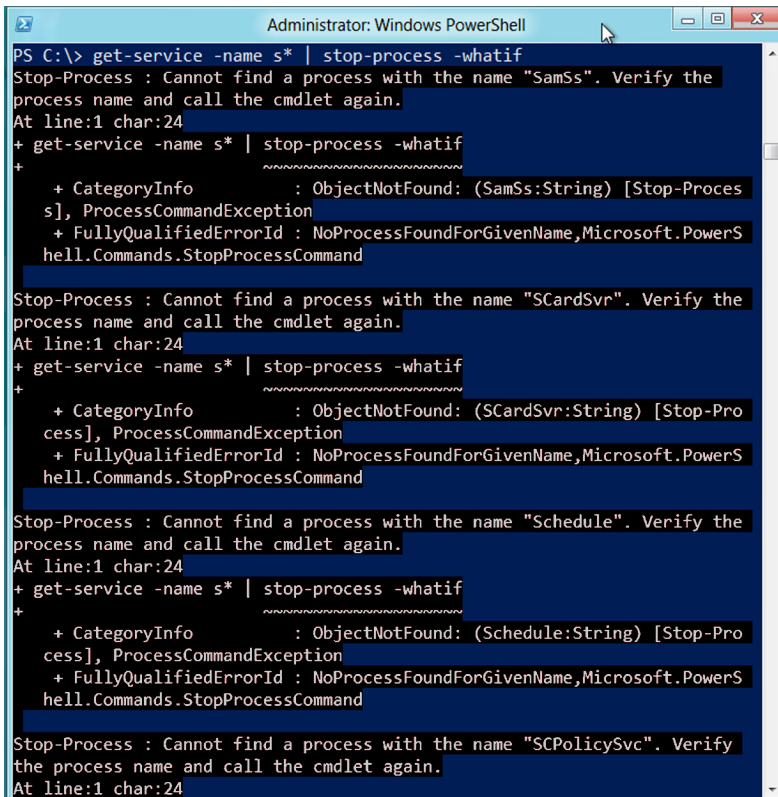
Wielu użytkowników zastanawiało się nad tym, co tak naprawdę się tutaj dzieje, więc spieszmy wyjaśnić, w jak prosty sposób działa powłoka — najzwyczajniej w świecie sprawdza, które właściwości wyników działania polecenia A mają nazwy pasujące do nazw parametrów polecenia B. To wszystko. W naszym przypadku, ponieważ właściwość `Name` ma taką samą nazwę jak parametr `-Name`, powłoka próbuje je ze sobą połączyć.

Oczywiście powłoka nie może tego zrobić od razu; najpierw musi sprawdzić, czy parametr `-Name` będzie akceptował *ByPropertyName* dane wejściowe z potoku. Aby to sprawdzić, wystarczy zajrzeć do pliku pomocy, tak jak to zostało pokazane na rysunku 9.6.



Rysunek 9.6. Sprawdzenie, czy parametr `-Name` polecenia `Stop-Object` może pobierać z potoku wartości parametrów na podstawie nazw właściwości (*ByPropertyName*)

W tym przypadku widzimy, że parametr `-Name` akceptuje wartości *ByPropertyName*, więc to połączenie działa. Cała sztuczka polega na tym, że w przeciwieństwie do wiązania *ByValue*, które pozwalało na wiązanie tylko jednego parametru, wiązanie *ByPropertyName* łączy wszystkie pasujące do siebie właściwości i parametry (oczywiście pod warunkiem, że określony parametr danego polecenia został zaprojektowany tak, aby akceptować wartości z potoku za pomocą wiązania *ByPropertyName*). W naszym obecnym przykładzie w taki sposób dopasowane mogą być tylko parametry `Name` oraz `-Name`. Rezultat? Sprawdź rysunek 9.7.



```
PS C:\> get-service -name s* | stop-process -whatif
Stop-Process : Cannot find a process with the name "SamSs". Verify the
process name and call the cmdlet again.
At line:1 char:24
+ get-service -name s* | stop-process -whatif
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (SamSs:String) [Stop-Proces
s], ProcessCommandException
+ FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerS
hell.Commands.StopProcessCommand

Stop-Process : Cannot find a process with the name "SCardSvr". Verify the
process name and call the cmdlet again.
At line:1 char:24
+ get-service -name s* | stop-process -whatif
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (SCardSvr:String) [Stop-Pro
cess], ProcessCommandException
+ FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerS
hell.Commands.StopProcessCommand

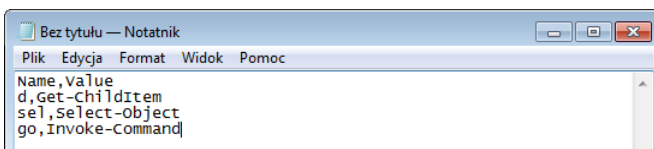
Stop-Process : Cannot find a process with the name "Schedule". Verify the
process name and call the cmdlet again.
At line:1 char:24
+ get-service -name s* | stop-process -whatif
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (Schedule:String) [Stop-Pro
cess], ProcessCommandException
+ FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerS
hell.Commands.StopProcessCommand

Stop-Process : Cannot find a process with the name "SCPolicySvc". Verify
the process name and call the cmdlet again.
At line:1 char:24
```

Rysunek 9.7. Próba przekazania za pomocą potoku wyników działania polecenia `Get-Service` do polecenia `Stop-Process`

Próba wykonania takiego polecenia kończy się wyświetleniem całego szeregu komunikatów o błędach. Problem polega na tym, że usługi mają zwykle nazwy takie jak `ShellHwDetection` czy `SessionEnv`, podczas gdy pliki wykonywalne usług mają nazwy zupełnie inne, na przykład `svchost.exe`. Polecenie `Stop-Process` zajmuje się tylko nazwami plików wykonywalnych, stąd nawet jeżeli właściwość `Name` łączy się z parametrem `-Name` za pomocą potoku, wartości właściwości `Name` nie mają żadnego sensu dla parametru `-Name`, co prowadzi do powstawania błędów.

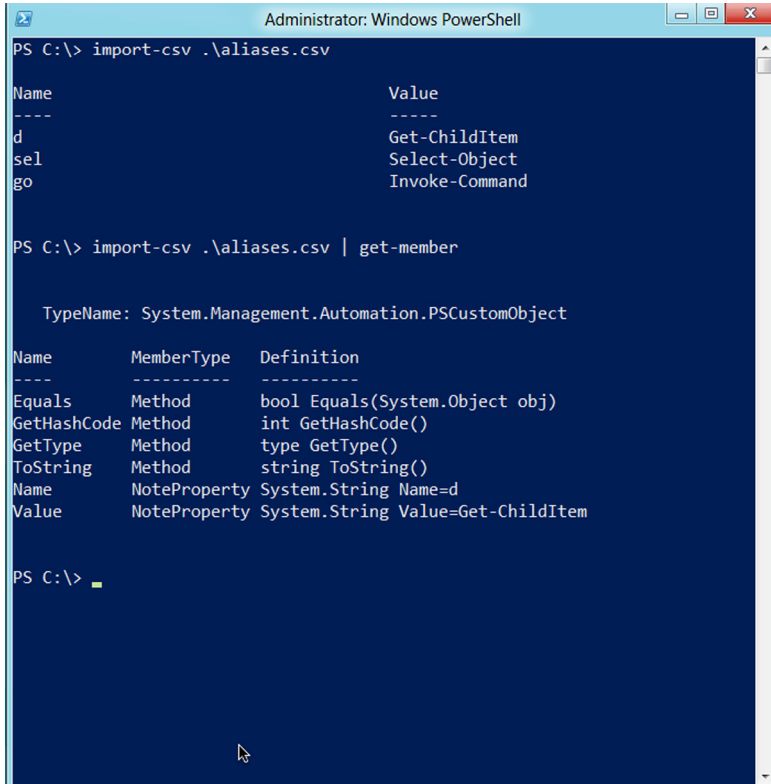
Spójrzmy na bardziej udany przykład. Utwórz w Notatniku prosty plik CSV, w którym poszczególne wartości są poroździelane przecinkami, tak jak to zostało pokazane na rysunku 9.8:



```
Bez tytułu — Notatnik
Plik  Edycja  Format  Widok  Pomoc
Name,Value
d,Get-Childitem
sel,Select-object
go,Invoke-Command
```

Rysunek 9.8. Tworzenie pliku CSV w programie Notatnik

Zapisz plik pod nazwą *Aliases.csv*. Teraz przejdź z powrotem do powłoki i spróbuj zaimportować ten plik, tak jak to zostało zilustrowane na rysunku 9.9. Powinieneś także spróbować przesłać wyniki działania polecenia *Import-CSV* do polecenia *Get-Member*, żeby sprawdzić elementy wyjściowe.



```
Administrator: Windows PowerShell
PS C:\> import-csv .\aliases.csv

Name                                     Value
----                                     -
d                                         Get-ChildItem
sel                                       Select-Object
go                                        Invoke-Command

PS C:\> import-csv .\aliases.csv | get-member

      TypeName: System.Management.Automation.PSCustomObject

Name      MemberType Definition
-----
Equals    Method      bool Equals(System.Object obj)
GetHashCode Method      int GetHashCode()
GetType   Method      type GetType()
ToString  Method      string ToString()
Name       NoteProperty System.String Name=d
Value      NoteProperty System.String Value=Get-ChildItem

PS C:\> _
```

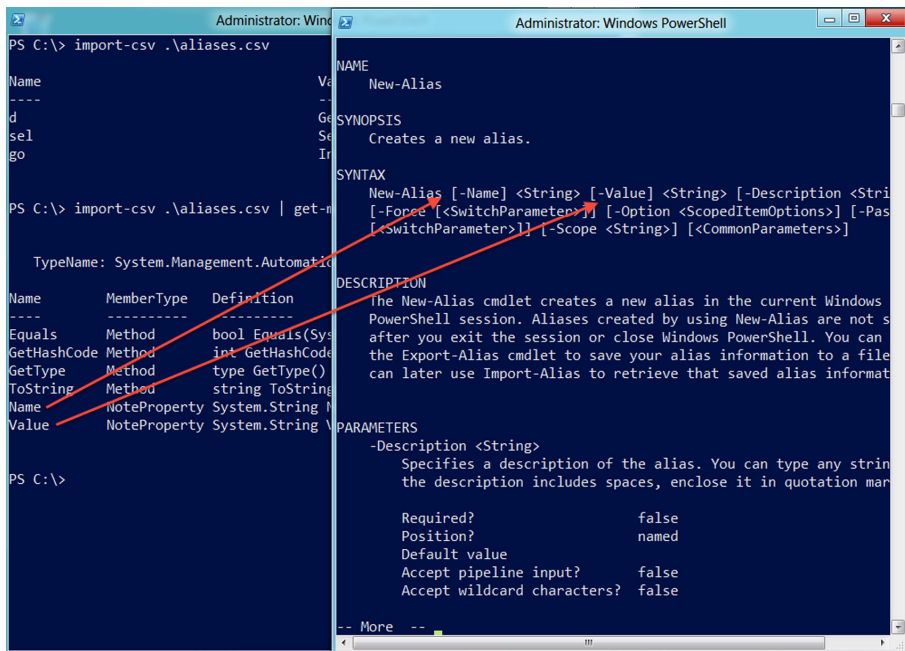
Rysunek 9.9. Importowanie pliku CSV i sprawdzanie jego elementów składowych

Wyraźnie widać, że kolumny z pliku CSV stają się właściwościami, a poszczególne wiersze danych w tym pliku stają się obiektami. Teraz sprawdź zawartość pliku pomocy polecenia *New-Alias*, jak to pokazano na rysunku 9.10.

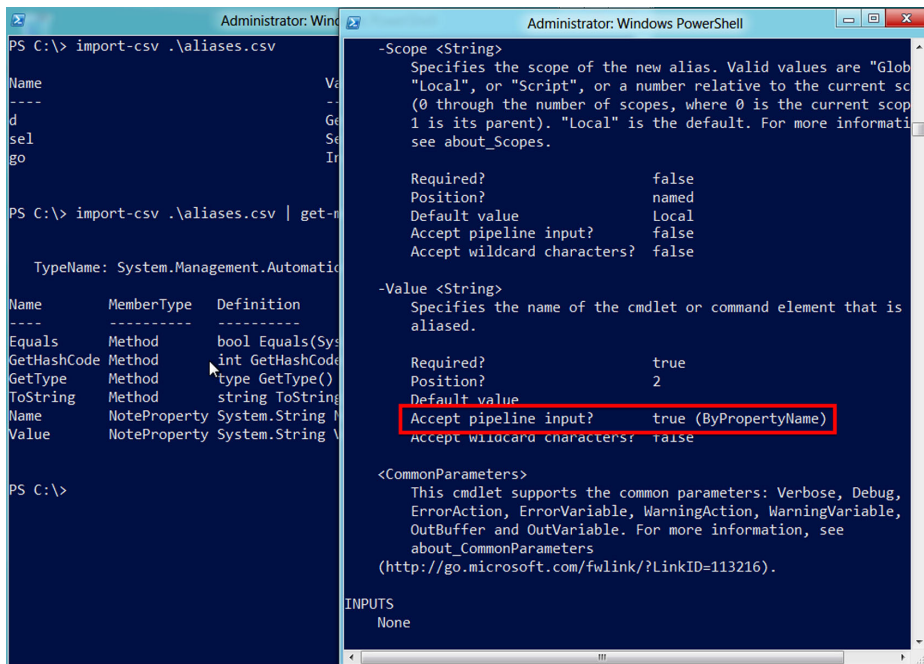
Obie właściwości (*Name* i *Value*) odpowiadają nazwom parametrów polecenia *New-Alias*. Oczywiście zostało to zrobione celowo — po utworzeniu pliku CSV możesz nadać tym kolumnom dowolne nazwy. Teraz sprawdź — parametry *-Name* i *-Value* akceptują wartości przekazywane z potoku za pomocą wiązania *ByPropertyName*, jak to widać na rysunku 9.11.

Oba parametry akceptują takie wartości, co oznacza, że ta sztuczka działa. Spróbuj teraz uruchomić następujące polecenie:

```
PS C:\> import-csv .\aliases.csv | new-alias
```



Rysunek 9.10. Dopasowywanie właściwości do nazw parametrów



Rysunek 9.11. Wyszukiwanie w pliku pomocy parametrów, które akceptują wartości z potoku za pomocą wiązania ByPropertyName

Wynikiem działania tego polecenia są trzy nowe aliasy o nazwach `d`, `sel` i `go`, które reprezentują, odpowiednio, polecenia `Get-ChildItem`, `Select-Object` oraz `Invoke-Command`. Jest to potężna technika przekazywania danych z jednego polecenia do drugiego, pozwalająca na wykonywanie bardzo złożonych zadań przy użyciu minimalnej liczby poleceń.

9.5. Gdy dane do siebie nie pasują — właściwości niestandardowe

Przykład z plikiem CSV jest fajny, ale łatwo jest nadawać nazwy właściwości i parametrów, kiedy tworzysz dane wejściowe od zera. Sytuacja staje się trudniejsza, kiedy jesteś zmuszony zajmować się obiektami lub danymi, które utworzył ktoś inny.

W tym przykładzie wprowadzamy nowe polecenie, do którego w swoim systemie możesz nie mieć dostępu: `New-ADUser`. Polecenie to jest częścią modułu `ActiveDirectory`, który można znaleźć na dowolnym kontrolerze domeny działającym pod kontrolą systemu `Windows Server 2008 R2` (lub nowszego). Modułu tego można również używać na komputerze klienckim, instalując pakiet `Microsoft Remote Server Administration Tools (RSAT)`. Na razie jednak nie martw się o uruchamianie polecenia — po prostu postaraj się uważnie prześledzić omawiany przykład.

Polecenie `New-ADUser` posiada parametry pozwalające na pobieranie informacji o nowym użytkowniku usługi `Active Directory`. Oto kilka przykładów takich parametrów:

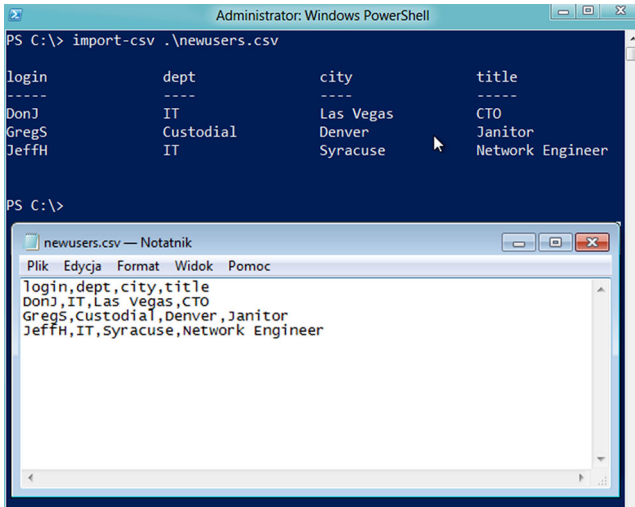
- `-Name` (obligatoryjny),
- `-samAccountName` (technicznie rzecz biorąc, podawanie tego parametru nie jest obowiązkowe, ale w praktyce musisz go podać, aby konto było możliwe do użycia),
- `-Department`,
- `-City`,
- `-Title`.

Moglibyśmy tutaj wymieniać również inne parametry, ale pozostaniemy przy podanych. Wszystkie wymienione parametry akceptują wartości z potoku `ByPropertyName`.

W tym przykładzie ponownie przyjmujemy założenie, że pracujesz z plikiem CSV, który otrzymałeś z działu kadr swojej firmy. Sęk w tym, że choć co najmniej kilkanaście razy wysyłałeś im pożądaną specyfikację formatu takiego pliku, otrzymywane od nich dane ciągle nieco od niego odbiegają, co pokazuje rysunek 9.12:

Jak widać na rysunku 9.12, powłoka może bez problemów zaimportować taki plik CSV, tworząc trzy obiekty, z których każdy będzie miał cztery właściwości. Problem polega na tym, że właściwość `dept` nie jest zgodna z parametrem `-Department` polecenia `New-ADUser`, właściwość `login` jest bez znaczenia, a Ty nie masz właściwości `samAccountName` ani `Name`, a obie są niezbędne do uruchomienia przedstawionego niżej polecenia tworzącego nowe konta użytkowników:

```
PS C:\> import-csv .\newusers.csv | new-aduser
```



Rysunek 9.12. Zawartość pliku CSV dostarczonego przez dział kadr

Jak możesz to naprawić? Oczywiście zawsze możesz otworzyć plik CSV i odpowiednio zmodyfikować jego zawartość, ale wymaga to sporo mozolnej, ręcznej pracy, a przecież jednym z celów korzystania z powłoki PowerShell jest ograniczenie pracy ręcznej. Dlaczego zatem nie spowodować, aby powłoka sama to naprawiła? Spójrz na następujący przykład:

```
PS C:\> import-csv .\newusers.csv |
>> select-object -property *,
>> @{name='samAccountName';expression={$_.login}},
>> @{label='Name';expression={$_.login}},
>> @{n='Department';e={$_.Dept}}
>>
```

login	: DonJ
dept	: IT
city	: Las Vegas
title	: CTO
samAccountName	: DonJ
Name	: DonJ
Department	: IT
login	: GregS
dept	: Custodial
city	: Denver
title	: Janitor
samAccountName	: GregS
Name	: GregS
Department	: Custodial
login	: JeffH
dept	: IT
city	: Syracuse
title	: Network Engineer
samAccountName	: JeffH
Name	: JeffH
Department	: IT

To całkiem ciekawa składnia polecenia, zatem spróbujmy podzielić ją na elementy składowe:

- Używamy polecenia `Select-Object` i jego parametru `-Property`. Zaczynamy od określenia właściwości `*`, co można wyrazić jako „wszystkie istniejące właściwości”. Zauważ, że po symbolu `*` występuje przecinek, co oznacza, że kontynuujemy listę właściwości.
- Następnie tworzymy tabelę mieszającą, która jest konstrukcją zaczynającą się od ciągu znaków `@{` i kończącą się znakiem `}`. Tabele mieszające składają się z jednej lub więcej par klucz-wartość, a polecenie `Select-Object` pozwala na wyszukiwanie określonych kluczy pasujących do podanego wzorca.
- Pierwszym kluczem, który chce wybrać polecenie `Select-Object`, jest `Name`, `N`, `Label` lub `L`, a wartość tego klucza jest nazwą właściwości, którą chcemy utworzyć. W pierwszej tabeli mieszania podajemy `SamAccountName`, w drugim — `Name`, a w trzecim — `Department`. Odpowiadają one nazwom parametrów polecenia `New-ADUser`.
- Drugim kluczem, którego potrzebuje polecenie `Select-Object`, może być `Expression` lub `E`. Wartość tego klucza to blok skryptu zawarty w nawiasach klamrowych `{}`. W obrębie tego bloku używasz specjalnego znacznika `$_` do odwoływania się do istniejącego obiektu przekazywanego za pomocą potoku (oryginalny wiersz danych z pliku CSV), po którym to znaczniku następuje kropka. Znacznik `$_` umożliwia dostęp do wybranej właściwości potokowanego obiektu lub inaczej mówiąc, do jednej kolumny pliku CSV, która reprezentuje wartości nowych właściwości.

ZRÓB TO SAM Utwórz plik CSV pokazany na rysunku 9.12. Następnie spróbuj samodzielnie wykonać polecenie, które pokazaliśmy w naszym przykładzie — możesz wpisać je dokładnie tak, jak to przedstawiliśmy.

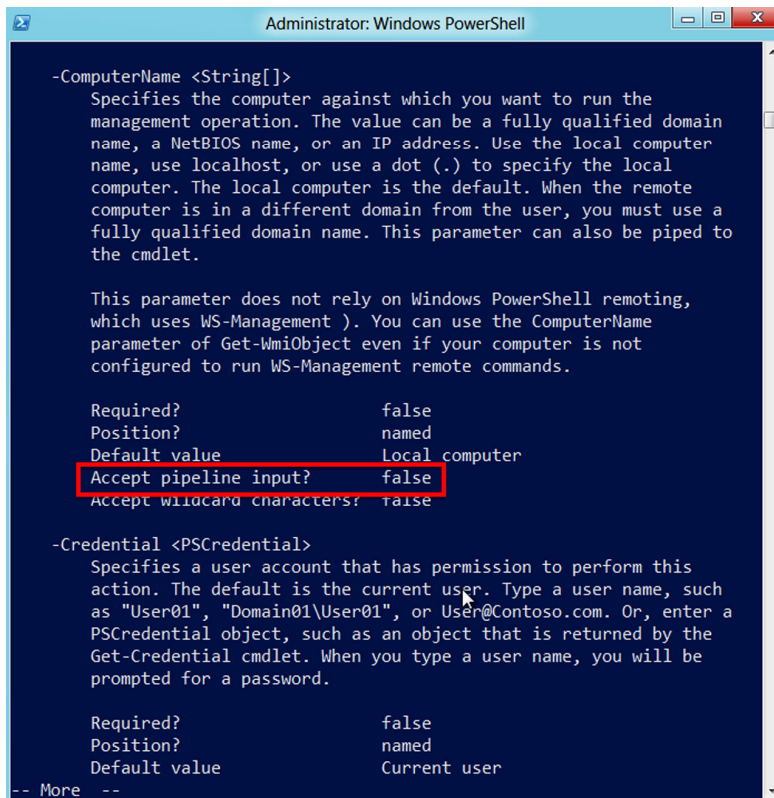
Nasze polecenie pobrało zawartość pliku CSV — czyli inaczej mówiąc, wyniki działania polecenia `Import-Csv` — i dynamicznie zmodyfikowało je w potoku. Nasze nowe wyniki działania odpowiadają temu, co na wejściu chce otrzymać polecenie `New-ADUser`, dzięki czemu możemy teraz tworzyć nowych użytkowników, uruchamiając następujące polecenie:

```
PS C:\> import-csv .\newusers.csv |
>> select-object -property *,
>> @{name='SamAccountName';expression={$_.login}},
>> @{label='Name';expression={$_.login}},
>> @{n='Department';e={$_.Dept}} |
>> new-aduser
>>
```

Składnia może być nieco toporna, ale sama technika ma niesamowicie wielki potencjał. Mechanizm ten jest bardzo użyteczny w przypadku wielu innych aspektów pracy z powłoką PowerShell, a z pewnością zobaczysz go ponownie w kolejnych rozdziałach. Takie i podobne rozwiązania znajdziesz nawet w przykładach zawartych w plikach pomocy poleceń powłoki PowerShell — wykonaj polecenie `Help Select -Example` i przekonaj się o tym na własne oczy.

9.6. Polecenia w nawiasach

Czasami, bez względu na to, jak bardzo się starasz, nie udaje Ci się w poprawny sposób za pomocą potoku przesyłać danych z jednego polecenia do drugiego. Rozważmy na przykład polecenie `Get-WmiObject`. Więcej szczegółowych informacji na jego temat zdobędziesz w kolejnym rozdziale, a na razie zobacz, co możesz znaleźć w pliku pomocy dla jego właściwości o nazwie `-ComputerName`, tak jak widać to na rysunku 9.13.



Rysunek 9.13. Przeglądanie zawartości pliku pomocy dla polecenia `Get-WmiObject`

Jak łatwo zauważyć, ten parametr nie akceptuje wartości (nazw komputerów) z potoku. W jaki sposób możemy zatem pobierać nazwy z jakiegoś miejsca — na przykład pliku tekstowego, w którym nazwy komputerów są zapisane po jednej w każdym wierszu — i przekazać je do polecenia? Następujące rozwiązanie nie będzie działać poprawnie:

```
PS C:\> get-content .\computers.txt | get-wmiobject -class win32_bios
```

Obiekty typu `String` utworzone przez polecenie `Get-Content` nie będą pasowały do parametru `-ComputerName` polecenia `Get-WmiObject`. Co zatem możesz zrobić? Spróbuj użyć nawiasów, tak jak to zostało pokazane poniżej:

```
PS C:\> Get-WmiObject -class Win32_BIOS -ComputerName (Get-Content .\computers.txt)
```


Jak zapewne pamiętasz z lekcji algebry w szkole średniej, umieszczenie działania w nawiasach oznacza, że zostanie ono wykonane jako pierwsze. Dokładnie tak samo postępuje powłoka PowerShell: najpierw wykonuje polecenia umieszczone w nawiasach. Wyniki działania tego polecenia — w tym przypadku grupa obiektów typu `String` — są przekazywane do parametru `-ComputerName`, a ponieważ parametr ten wymaga podania grupy obiektów typu `String`, takie polecenie działa poprawnie.

ZRÓB TO SAM Jeśli masz pod ręką kilka komputerów, na których mógłbyś wypróbować działanie takiego polecenia, to zrób to teraz. Umieść odpowiednie nazwy komputerów lub adresy IP w pliku *computers.txt*. Najlepsze rezultaty osiągniesz w przypadku komputerów znajdujących się w tej samej domenie, ponieważ w takim środowisku łatwiej uniknąć kłopotów z odpowiednimi uprawnieniami dostępu.

Sztuczka z użyciem nawiasów daje ogromne możliwości, ponieważ jest zupełnie niezależna od mechanizmu potokowego wiązania parametrów — po prostu pobieramy obiekty utworzone przez polecenia umieszczone w nawiasach i wstawiamy je bezpośrednio do odpowiedniego parametru. Powinieneś jednak pamiętać, że taka technika nie zadziała, jeżeli polecenia umieszczone w nawiasach nie będą generowały dokładnie takiego typu obiektu, jaki jest oczekiwany przez dany parametr. Stąd od czasu do czasu będziesz musiał trochę pokombinować. Zobaczmy zatem, co można tu zrobić.

9.7. Wyodrębnianie wartości z jednej właściwości

Nieco wcześniej w tym rozdziale pokazywaliśmy przykład użycia nawiasów do wykonania polecenia `Get-Content`, którego wyniki działania były przesyłane do parametru wywołania innego polecenia:

```
Get-Service -ComputerName (Get-Content names.txt)
```

Zamiast pobierać nazwy komputerów ze statycznego pliku tekstowego, możesz je pobierać bezpośrednio z usługi Active Directory. Dzięki modułowi `ActiveDirectory` (dostępnemu na kontrolerach domeny działających pod kontrolą systemu Windows Server 2008 R2 lub nowszego; na swoim komputerze możesz go zainstalować razem z pakietem RSAT) możesz wykonywać zapytania na wszystkich kontrolerach domeny:

```
get-adcomputer -filter * -searchbase "ou=domain controllers, dc=company, dc=pri"
```

Czy możemy zatem użyć sztuczki z nawiasami do przekazywania nazw komputerów bezpośrednio do polecenia `Get-Service`? Na przykład czy polecenie przedstawione poniżej będzie działać poprawnie?

```
Get-Service -computerName (Get-ADComputer -filter * -searchBase "ou=domain  
↪controllers,dc=company,dc=pri")
```

Dla zainteresowanych

Jeśli nie masz pod ręką kontrolera domeny, nie przejmuj się — zaraz Ci wyjaśnimy, co powinieneś wiedzieć o poleceniu `Get-ADComputer`.

Po pierwsze, polecenie to jest zawarte w module o nazwie `ActiveDirectory`. Jak już wspominaliśmy, moduł ten jest dostępny na kontrolerach domeny działających pod kontrolą systemu Windows Server 2008 R2 lub nowszego, a na swoim komputerze domowym możesz go zainstalować razem z pakietem RSAT.

Po drugie, polecenie to — jak można się spodziewać — pobiera z domeny obiekty reprezentujące poszczególne komputery.

Po trzecie, posiada ono dwa przydatne parametry. Pierwszy parametr, `-Filter *`, pobiera wszystkie komputery, ale w razie potrzeby możesz określić inne kryteria filtrowania, aby ograniczyć wyniki na przykład tylko do jednego, wybranego komputera. Z kolei parametr `-SearchBase` informuje polecenie o tym, w którym miejscu rozpocząć wyszukiwanie komputerów; w tym przykładzie uruchamiamy go w jednostce OU=Domain Controllers domeny `Company.com`:

```
get-adcomputer -filter * -searchbase "ou=domain controllers, dc=company, dc=pri"
```

Po czwarte, obiekty reprezentujące komputery mają właściwość `Name`, która zawiera nazwę komputera.

Zdajemy sobie sprawę z tego, że użycie polecenia, do którego możesz nie mieć dostępu (w zależności od tego, jakim dysponujesz środowiskiem testowym), jest trochę nie fair. Nie zmienia to jednak w niczym faktu, że w wielu sytuacjach jest to niezwykle przydatne rozwiązanie i zdecydowanie warto go używać w środowisku produkcyjnym.

Niestety takie polecenie nie będzie działać. Jeżeli zajrzysz do pliku pomocy polecenia `Get-Service`, przekonasz się, że parametr `-ComputerName` oczekuje wartości typu `String`. Spróbujmy zatem wykonać następujące polecenie:

```
get-adcomputer -filter * -searchbase "ou=domain controllers, dc=company,dc=pri" | gm
```

Wyniki działania polecenia `Get-Member` ujawniają, że polecenie `Get-ADComputer` tworzy obiekty typu `ADComputer`. Nie są to obiekty typu `String`, więc parametr `-ComputerName` nie będzie wiedział, co z nimi zrobić. Ale obiekty `ADComputer` również mają właściwość `Name`. Należy zatem wyodrębnić wartości właściwości `Name` takiego obiektu i przekazać wartości reprezentujące nazwy komputerów do parametru `-ComputerName`.

WSKAZÓWKA Jest to ważne zagadnienie dotyczące powłoki PowerShell, więc jeżeli trochę się w tym wszystkim pogubiłeś, *zatrzymaj się* i jeszcze raz uważnie przeczytaj poprzednie akapity. Polecenie `Get-ADComputer` tworzy obiekty typu `ADComputer`, co potwierdzają wyniki działania polecenia `Get-Member`. Parametr `-ComputerName` polecenia `Get-Service` nie może pobierać obiektów typu `ADComputer`; akceptuje tylko obiekty typu `String`, tak jak to zostało opisane w pliku pomocy, stąd polecenie, które umieściliśmy w nawiasach, nie zadziała tak, jak mogłeś tego oczekiwać.

I znów w takiej sytuacji uratować nas może polecenie `Select-Object`, ponieważ posiada ono parametr `-expandProperty`, który pozwala na określenie nazwy interesującej nas właściwości. Polecenie pobiera tę właściwość, wyodrębnia jej wartości i zwraca je jako wyniki działania. Rozważmy poniższy przykład:

```
Get-ADComputer -filter * -searchbase "ou=domain controllers, dc=company, dc=pri"  
↪ | Select-Object -expand name
```

Wynikiem działania powyższego polecenia powinna być prosta lista nazw komputerów. Możemy ją przekazać do parametru `-ComputerName` polecenia `Get-Service` (lub dowolnego innego polecenia posiadającego parametr `-ComputerName`):

```
Get-Service -computerName (Get-ADComputer -filter * -searchbase "ou=domain  
↪ controllers, dc=company, dc=pri" | Select-Object -expand name)
```

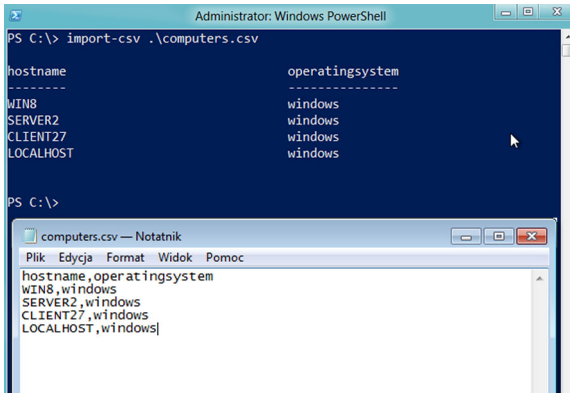
WSKAZÓWKA I znów jest to bardzo ważna koncepcja. Zwykle polecenie takie jak `Select-Object -Property Name` tworzy obiekty, które mają tylko właściwość `Name`, ponieważ jest to wszystko, co wymieniliśmy w wierszu wywołania polecenia. Parametr `-ComputerName` nie akceptuje jakiegoś losowego obiektu posiadającego właściwość `Name`; zamiast tego oczekuje wartości typu `String`, co jest znacznie prostszym rozwiązaniem. Użycie parametru `-ExpandName` powoduje, że polecenie `Select-Object` przechodzi do właściwości `Name`, wyodrębnia jej wartości i zwraca je w postaci wartości typu `String`.

Jak widać, jest to fajna sztuczka, która pozwala łączyć ze sobą jeszcze szerszą gamę poleceń. Oszczędza Ci pisanie i sprawia, że powłoka PowerShell może wykonywać jeszcze bardziej złożone zadania.

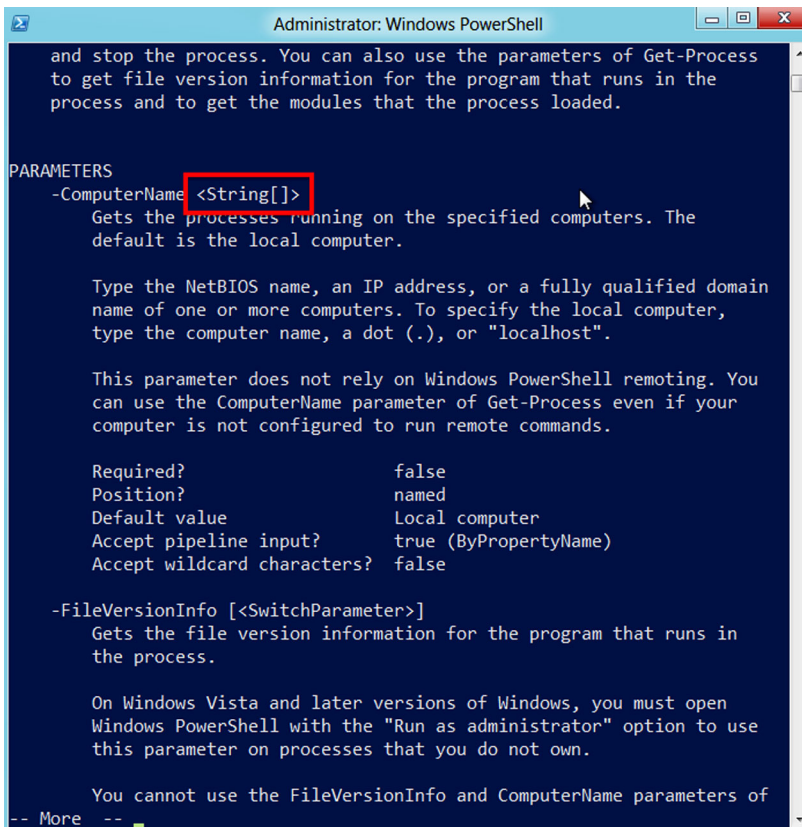
Teraz, gdy widziałeś już, jak wygodne może być polecenie `Get-ADComputer`, spójrzmy na podobny przykład z wykorzystaniem poleceń, do których na pewno powinieneś mieć dostęp. Zakładamy jednak, że używasz najnowszej wersji systemu Windows, choć w tym przykładzie nie musisz mieć komputera w domenie, dostępu do kontrolera domeny ani nawet dostępu do systemu operacyjnego serwera. Zamiast tego będziemy ogólnie mówić o „pobieraniu nazw komputerów”, ponieważ jest to powszechna potrzeba każdego administratora w środowisku produkcyjnym.

Rozpocznij od utworzenia przy użyciu Notatnika pliku CSV, który będzie wyglądał tak, jak widać to na rysunku 9.14. Jeżeli umieścisz tam nazwy komputerów działających w Twojej sieci, będziesz mógł uruchomić na nich nasze przykładowe polecenia. Jeżeli masz do dyspozycji tylko jeden komputer, jako nazwy hosta użyj `localhost` i wpisz ją w pliku trzy lub cztery razy. Wszystko nadal będzie działać poprawnie.

Załóżmy teraz, że chcesz wyświetlić listę procesów działających na każdym z tych komputerów. Jeżeli przeanalizujesz zawartość pliku pomocy dla polecenia `Get-Process` (zobacz rysunek 9.15), przekonasz się, że parametr `-ComputerName` akceptuje wejście potoku `ByPropertyName` i oczekuje, że dane wejściowe będą obiektami typu `String`. Nie będziemy się jednak trudzić modyfikowaniem danych przekazywanych za pomocą potoku, a zamiast tego skupimy się na wyodrębnieniu z nich odpowiedniej właściwości. Istotną informacją z pliku pomocy jest dla nas to, że parametr `-ComputerName` potrzebuje jednego lub więcej obiektów typu `String`.

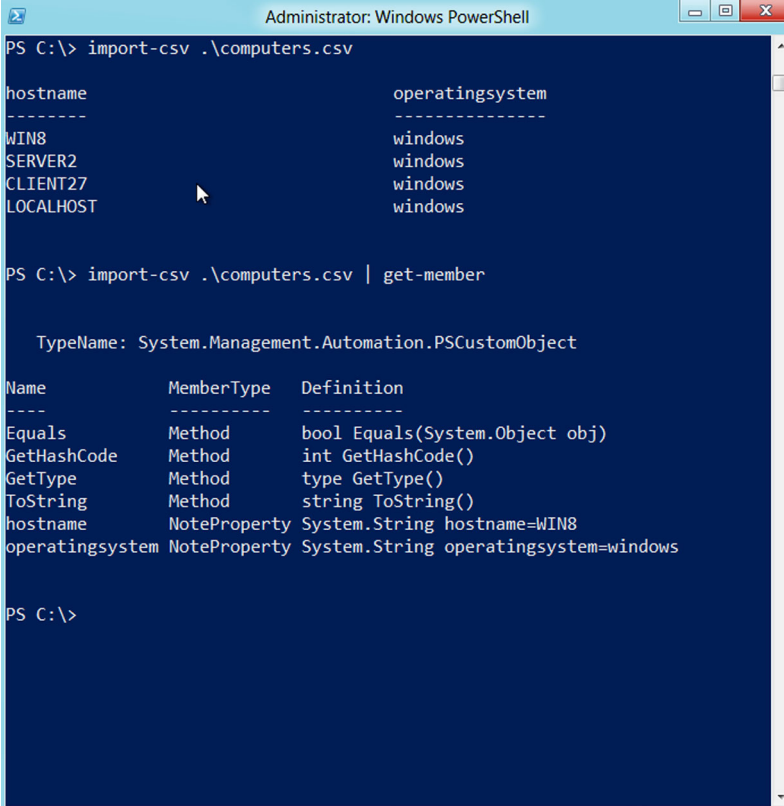


Rysunek 9.14. Upewnij się, że możesz poprawnie zaimportować ten plik CSV — użyj polecenia `Import-CSV` i sprawdź, czy otrzymałeś wyniki podobne do przedstawionych na rysunku



Rysunek 9.15. Weryfikowanie typu danych akceptowanych przez parametr `-ComputerName`

Wracamy do podstaw: zacznij od sprawdzenia, jakie dane generuje polecenie A. Aby to zrobić, prześlij wyniki jego działania do polecenia `Get-Member`. Wyniki takiej operacji przedstawiono na rysunku 9.16:



```
PS C:\> import-csv .\computers.csv

hostname                                operatingsystem
-----
WIN8                                    windows
SERVER2                                windows
CLIENT27                               windows
LOCALHOST                              windows

PS C:\> import-csv .\computers.csv | get-member

TypeName: System.Management.Automation.PSCustomObject

Name      MemberType Definition
-----
Equals     Method      bool Equals(System.Object obj)
GetHashCode Method      int GetHashCode()
GetType    Method      type GetType()
ToString   Method      string ToString()
hostname   NoteProperty System.String hostname=WIN8
operatingsystem NoteProperty System.String operatingsystem=windows

PS C:\>
```

Rysunek 9.16. Polecenie `Import-CSV` tworzy obiekty typu `PSCustomObject`

Obiekty `PSCustomObject`, będące wynikami działania polecenia `Import-CSV`, to nie wartości typu `String`, zatem następujące polecenie nie będzie działać:

```
PS C:\> Get-Process -ComputerName (import-csv .\computers.csv)
```

Spróbujmy z pliku CSV wybrać pole `HostName` i zobaczymy, co nam to przyniesie. Wynik takiej operacji został pokazany na rysunku 9.17.

Jak widać, wynikiem działania polecenia wciąż jest obiekt typu `PSCustomObject`, ale ma on mniej właściwości niż w poprzednim przypadku. To bardzo ważna cecha polecenia `Select-Object` i jego parametru `-Property` — wynikiem działania tego polecenia nadal jest obiekt.

Parametr `-ComputerName` nie może pobierać obiektów typu `PSCustomObject`, więc następne polecenie, przedstawione poniżej, nadal nie będzie działać:

```
PS C:\> Get-Process -ComputerName (import-csv .\computers.csv | select -property hostname)
```

```

Administrator: Windows PowerShell

PS C:\> import-csv .\computers.csv | select -property hostname | get-member

TypeName: Selected.System.Management.Automation.PSCustomObject

Name      MemberType Definition
-----
Equals     Method      bool Equals(System.Object obj)
GetHashCode Method      int GetHashCode()
GetType    Method      type GetType()
ToString   Method      string ToString()
hostname   NoteProperty System.String hostname=WIN8

PS C:\>

```

Rysunek 9.17. Wybranie jednej właściwości nadal daje obiekt typu PSCustomObject

Właśnie w takiej sytuacji powinniśmy użyć parametru `-ExpandProperty`. Spróbuj samodzielnie przeprowadzić taki eksperyment i porównaj otrzymane wyniki z naszymi wynikami, przedstawionymi na rysunku 9.18.

```

Administrator: Windows PowerShell

PS C:\> import-csv .\computers.csv | select -expand hostname | get-member

TypeName: System.String

Name      MemberType Definition
-----
Clone     Method      System.Object Clone()
CompareTo Method      int CompareTo(System.Object valu...
Contains  Method      bool Contains(string value)
CopyTo    Method      System.Void CopyTo(int sourceInd...
EndsWith  Method      bool EndsWith(string value), boo...
Equals    Method      bool Equals(System.Object obj), ...
GetEnumerator Method    System.CharEnumerator GetEnumera...
GetHashCode Method    int GetHashCode()
GetType    Method      type GetType()
GetTypeCode Method    System.TypeCode GetTypeCode()
IndexOf   Method      int IndexOf(char value), int Ind...
IndexOfAny Method    int IndexOfAny(char[] anyOf), in...
Insert    Method      string Insert(int startIndex, st...
IsNormalized Method    bool IsNormalized(), bool IsNorm...
LastIndexOf Method    int LastIndexOf(char value), int...
LastIndexOfAny Method    int LastIndexOfAny(char[] anyOf)...
Normalize Method      string Normalize(), string Norma...
PadLeft   Method      string PadLeft(int totalWidth), ...
PadRight  Method      string PadRight(int totalWidth),...
Remove    Method      string Remove(int startIndex, in...
Replace   Method      string Replace(char oldChar, cha...
Split     Method      string[] Split(Params char[] sep...
StartsWith Method    bool StartsWith(string value), b...
Substring Method      string Substring(int startIndex)...
ToBoolean Method      bool ToBoolean(System.IFormatPro...
ToByte    Method      byte ToByte(System.IFormatProvid...

```

Rysunek 9.18. Nareszcie w wynikach działania mamy obiekt typu String!

Ponieważ wartości właściwości `Hostname` są ciągami tekstu, parametr `-ExpandProperty` jest w stanie zamienić je na obiekty typu `String`, czyli na wartości, jakich oczekuje parametr `-ComputerName`. Oznacza to, że kolejna wersja naszego polecenia będzie wreszcie działać poprawnie:

```
PS C:\> Get-Process -ComputerName (import-csv .\Computers.csv | select -expand hostname)
```

Taka technika daje naprawdę ogromne możliwości. Na początku możesz mieć pewne problemy z jej opanowaniem, ale kiedy uświadomisz sobie, że właściwość jest czymś w rodzaju pudełka, wszystko stanie się bardziej zrozumiałe. Używając polecenia `Select -Property`, decydujesz co prawda, które pudełka chcesz otrzymać, ale nadal są to pudełka. Dopiero za pomocą polecenia `Select -ExpandProperty` wyodrębniasz zawartość pudełka i całkowicie się go pozbywasz, pozostawiając sobie tylko jego zawartość.

9.8. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Po raz kolejny w stosunkowo krótkim czasie omówiliśmy bardzo wiele ważnych koncepcji. Najlepszym sposobem na ugruntowanie nowej wiedzy jest jej natychmiastowe wykorzystanie w praktyce. Sugerujemy, abyś poniższe ćwiczenia wykonywał w przedstawionej kolejności, ponieważ każde kolejne zadanie opiera się na wynikach poprzednich zadań lub nawiązuje do nich, a wszystko po to, aby Ci przypominać, czego się do tej pory nauczyłeś, i pomagać w znalezieniu praktycznych sposobów wykorzystania tej wiedzy.

Aby było to jednak nieco trudniejsze, spróbujemy przekonać Cię do rozważenia polecenia `Get-ADComputer`. Polecenie to znajdziesz na każdym kontrolerze domeny działającym pod kontrolą systemu Windows Server 2008 R2 lub nowszego, ale tak naprawdę nie będzie Ci ono potrzebne do wykonania zadania. Musisz wiedzieć tylko trzy rzeczy:

- Polecenie `Get-ADComputer` ma parametr o nazwie `-filter`; uruchomienie polecenia `Get-ADComputer -filter *` pobiera z domeny wszystkie obiekty reprezentujące komputery.
- Obiekty reprezentujące komputery w domenie mają właściwość `Name`, która zawiera nazwę komputera.
- Obiekty reprezentujące komputery w domenie są typu `ADComputer`, co oznacza, że polecenie `Get-ADComputer` tworzy obiekty typu `ADComputer`.

To wszystko, co musisz wiedzieć. Mając to na uwadze, spróbuj wykonać zadania przedstawione poniżej.

UWAGA Nie musisz uruchamiać tych poleceń. Jest to ćwiczenie bardziej umysłowe. Zamiast uruchamiać samo polecenie, powinieneś raczej odpowiedzieć na pytanie, czy te polecenia będą działać i dlaczego. Dowiedziałeś się, jak działa polecenie `Get-ADComputer` i jakie są jego wyniki; w podobny sposób możesz zapoznać się z plikami pomocy innych poleceń i sprawdzić, jakich danych wejściowych oczekują.

1. Czy polecenie przedstawione poniżej pozwoli na pobranie listy poprawek zainstalowanych na wszystkich komputerach w określonej domenie? Dlaczego tak lub dlaczego nie? Przedstaw wyjaśnienie podobne do tych, które prezentowaliśmy wcześniej w tym rozdziale.

```
Get-Hotfix -ComputerName (Get-ADComputer -filter * | Select-Object -expand Name)
```

2. Czy przedstawiona niżej alternatywna wersja poprzedniego polecenia będzie działać prawidłowo i pozwoli na pobranie listy poprawek zainstalowanych na tych samych komputerach co w poprzednim przykładzie? Dlaczego tak lub dlaczego nie? Przedstaw wyjaśnienie podobne do tych, które prezentowaliśmy wcześniej w tym rozdziale.

```
Get-ADComputer -filter * | Get-HotFix
```

3. Czy przedstawiona niżej trzecia wersja tego polecenia będzie działać prawidłowo i pozwoli na pobranie listy poprawek zainstalowanych na tych samych komputerach co w poprzednich przykładach? Dlaczego tak lub dlaczego nie? Przedstaw wyjaśnienie podobne do tych, które prezentowaliśmy wcześniej w tym rozdziale.

```
Get-ADComputer -filter * | Select-Object @{l = 'computername'; e = {$_.name}} | Get-Hotfix
```

4. Napisz polecenie korzystające z potokowego wiązania parametrów, które będzie pobierało listę procesów działających na poszczególnych komputerach w domenie Active Directory (AD). Nie używaj nawiasów.
5. Napisz polecenie, które będzie pobierało listę zainstalowanych usług z poszczególnych komputerów w domenie AD. Nie korzystaj z parametrów w potoku; zamiast tego użyj odpowiednich poleceń w nawiasach.
6. W przypadku niektórych poleceń Microsoft zapominał o dodaniu możliwości potokowego wiązania parametrów. Sprawdź, czy następujące polecenie będzie działało poprawnie i pobierało informacje ze wszystkich komputerów w domenie? Przedstaw wyjaśnienie podobne do tych, które prezentowaliśmy wcześniej w tym rozdziale.

```
Get-ADComputer -filter * | Select-Object @{l='computername'; e={$_.name}} |  
Get-WmiObject -class Win32_BIOS
```

9.9. Co dalej?

Z naszych doświadczeń wynika, że wielu użytkowników ma pewne trudności z opanowaniem takiej koncepcji działania potoków głównie dlatego, że jest ona nieco abstrakcyjna. Niestety, opanowanie tego materiału ma również kluczowe znaczenie dla zrozumienia sposobu działania powłoki PowerShell. W razie potrzeby przeczytaj ten rozdział ponownie, jeszcze raz samodzielnie uruchom przykładowe polecenia i uważnie przyjrzyj się ich wynikom działania. Spróbuj na przykład odpowiedzieć na pytanie, dlaczego wyniki działania tego polecenia:

```
Get-Date | Select -Property DayOfWeek
```

nieco różnią się od wyników działania następującego polecenia:

```
Get-Date | Select -ExpandProperty DayOfWeek
```


Jeżeli nadal nie jesteś pewien, zajrzyj na forum dyskusyjne na stronie <http://PowerShell.org>.

9.10. Odpowiedzi

1. Polecenie powinno działać poprawnie, ponieważ zagnieżdżone wyrażenie `Get-ADComputer` zwróci kolekcję nazw komputerów, a parametr `-ComputerName` może przyjmować tablicę wartości.
2. Takie polecenie nie zadziała, ponieważ polecenie `Get-Hotfix` nie przyjmuje żadnych parametrów wiązanych `ByValue`. Ten cmdlet może przyjmować wartości parametru `-ComputerName` według nazwy właściwości, ale nie używamy tego w tym poleceniu.
3. Polecenie powinno działać poprawnie. Pierwsza część wyrażenia przesyła niestandardowy obiekt do potoku, z którego wyodrębniana jest właściwość `ComputerName`. Ta właściwość może być powiązana z parametrem `ComputerName` polecenia `Get-Hotfix`, ponieważ polecenie to może przyjmować wartości parametrów z potoku według nazw właściwości (`ByPropertyName`).
4. `Get-Service -Computername (get-adcomputer -filter * | Select-Object -expand >property name)`
5. `Get-ADComputer -filter * | Select-Object @{l='computername';e={$_.name}} | >Get-WmiObject -class -Win32_BIOS`
6. Takie polecenie nie będzie działać. Parametr `ComputerName` polecenia `Get-WMIObject` nie przyjmuje żadnych parametrów wiązanych potokowo.

10

Formatowanie wyników działania

Dla przypomnienia — wiesz już, że wynikiem działania poleceń powłoki PowerShell są obiekty i że obiekty te często posiadają znacznie więcej właściwości, niż powłoka PowerShell domyślnie wyświetla na ekranie. Wiesz również, jak używać polecenia `Gm`, aby uzyskać listę wszystkich właściwości obiektu, a także jak używać polecenia `Select-Object` do wybierania właściwości, które chcesz zobaczyć. Do tego momentu w książce polegaliśmy na domyślnej konfiguracji i regułach powłoki PowerShell, które określały, w jaki sposób wyniki działania poleceń będą wyświetlane na ekranie (lub w pliku czy na drukarce). W tym rozdziale nauczysz się nadpisywać ustawienia domyślne i tworzyć własne formaty wyświetlania wyników działania poleceń.

10.1. Formatowanie — upiększanie tego, co widzisz

Nie chcielibyśmy, abyś odniósł wrażenie, że powłoka PowerShell jest pełnoprawnym narzędziem do zarządzania systemem i tworzenia raportów, ponieważ tak nie jest. Ale powłoka PowerShell ma naprawdę ogromne możliwości zbierania informacji o komputerach, a dysponując określonym zestawem danych, można z pewnością tworzyć efektywne raporty wykorzystujące takie informacje. Cała sztuczka polega na uzyskaniu odpowiednich efektów, a tutaj właśnie kłania się formatowanie wyników.

Na pierwszy rzut oka mechanizm formatowania wyników działania poleceń powłoki PowerShell może wydawać się łatwy w użyciu — i w większości przypadków jest to prawda. Z drugiej strony jednak można się tam natknąć na kilka naprawdę złośliwych „pułapek”, dlatego chcemy się upewnić, że rozumiesz, jak on działa i dlaczego robi to, co robi. Nie mamy jednak zamiaru tutaj poprzestać na zaprezentowaniu kilku nowych poleceń — zamiast tego postaramy się raczej wyjaśnić, jak działa cały system, jak go używać i jakie ograniczenia możesz napotkać.

10.2. Praca z formatowaniem domyślnym

Rozpocznij od uruchomienia naszego starego dobrego polecenia `Get-Process` i uważnie przyjrzyj się nagłówkom kolumn. Z pewnością zauważysz, że często nie odpowiadają dokładnie nazwom reprezentowanych właściwości. Poszczególne nagłówki mają różne szerokości, odpowiednie wyrównanie i tak dalej. Cała ta konfiguracja musi pochodzić z jakiegoś miejsca, prawda? Znajdziesz ją w jednym z plików, *format.ps1xml*, instalowanych razem z powłoką PowerShell. W szczególności reguły formatowania obiektów procesów znajdują się w pliku *DotNetTypes.format.ps1xml*.

ZRÓB TO SAM Powinieneś mieć pod ręką uruchomioną powłokę PowerShell, tak abyś mógł podążać za tym, co zaraz będziemy pokazywać. Pomoże Ci to zrozumieć, jak działa system formatowania i co ma „pod maską”.

Zacniemy od przejścia do folderu instalacyjnego powłoki PowerShell i otwarcia pliku *DotNetTypes.format.ps1xml*. Pamiętaj, aby nie zapisywać żadnych zmian w tym pliku! Jest podpisany cyfrowo, a wszelkie modyfikacje jego zawartości, nawet jeżeli będzie to pojedyncza spacja czy wstawiony nowy, pusty wiersz, spowodują unieważnienie podpisu cyfrowego i uniemożliwią powłoce PowerShell korzystanie z tego pliku.

```
PS C:\>cd $pshome
PS C:\>notepad dotnettypes.format.ps1xml
```

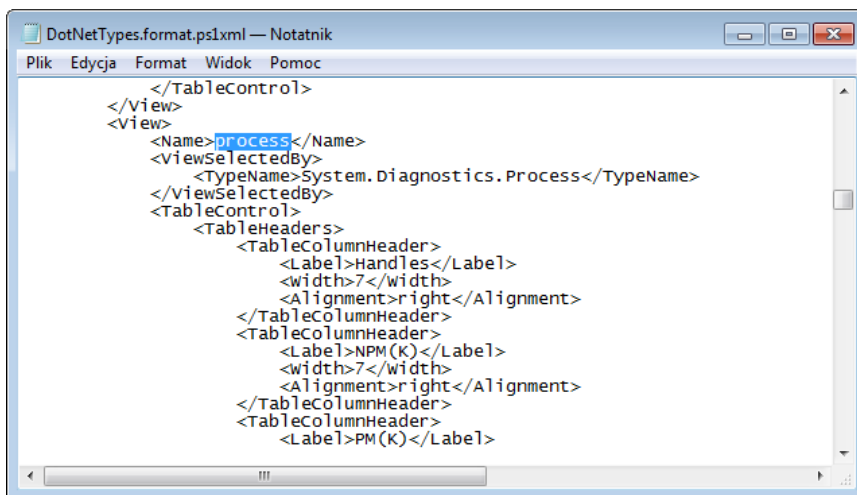
Następnie sprawdź, jakie obiekty zwraca polecenie `Get-Process`:

```
PS C:\>get-process | gm
```

Teraz wykonaj następujące kroki:

1. Skopiuj i wklej do schowka pełną nazwę typu obiektu: `System.Diagnostics.Process`. Aby to zrobić, w wynikach działania powyższego polecenia zaznacz nazwę typu i naciśnij klawisz *Enter*.
2. Przejdź do Notatnika, w którym otworzyłeś plik reguł, i naciśnij kombinację klawiszy *Ctrl+F*, aby wywołać okno wyszukiwania.
3. W oknie wyszukiwania wklej nazwę typu skopiowaną wcześniej do schowka. Naciśnij przycisk *Znajdź następny*.
4. Pierwszym elementem, który znajdziesz, będzie prawdopodobnie obiekt `Process` ➔ `Module`, a nie obiekt `Process`, zatem ponownie naciśnij przycisk *Znajdź następny*, aż znajdziesz obiekt `System.Diagnostics.Process`. Na rysunku 10.1 pokazujemy, co powinieneś znaleźć.

To, co teraz widzisz w swoim Notatniku, to zestaw wskazówek, które określają domyślny sposób wyświetlania obiektów procesów. Przewiń w dół, a zobaczysz definicję **widoku tabeli**, którego powinieneś się spodziewać, ponieważ wiesz już, że informacje o procesach są wyświetlane w postaci tabeli z wieloma kolumnami. Zobaczysz tutaj znane Ci nazwy kolumn, a jeśli przewiniesz nieco dalej, znajdziesz miejsce, w którym definiowane



Rysunek 10.1. Wyszukiwanie widoku o nazwie process w Notatniku

są właściwości wyświetlane w poszczególnych kolumnach. Zobaczysz również definicje szerokości kolumn i linii trasowania. Po zakończeniu przeglądania zamknij Notatnik, uważając, aby nie zapisać żadnych zmian, które mogłeś przypadkowo wprowadzić, i powrót do powłoki PowerShell.

A oto co się dzieje w powłoce PowerShell po uruchomieniu polecenia `Get-Process`:

1. Polecenie `Get-Process` przekazuje obiekty typu `System.Diagnostics.Process` do potoku.
2. Na „końcu” potoku znajduje się niewidoczny cmdlet o nazwie `Out-Default`. Jest tam zawsze, a jego zadaniem jest zbieranie obiektów znajdujących się w potoku po wykonaniu wszystkich poleceń.
3. Cmdlet `Out-Default` przekazuje obiekty do cmdletu `Out-Host`, ponieważ konsola PowerShell została zaprojektowana tak, że domyślnym urządzeniem wyjściowym jest ekran (nazywany *hostem*). Teoretycznie ktoś mógłby napisać powłokę, która jako domyślnego wyjścia będzie używała drukarki lub nawet systemu plików, ale z tego, co wiemy, to jeszcze nikt tego nie zrobił.
4. Większość poleceń z rodziny `Out-` nie jest w stanie pracować ze standardowymi obiektami. Zamiast tego są zaprojektowane do pracy ze specjalnymi instrukcjami formatowania, zatem kiedy polecenie `Out-Host` „widzi”, że otrzymało standardowe obiekty, przekazuje je do systemu formatowania.
5. System formatowania analizuje typ obiektu i postępuje zgodnie z wewnętrznym zestawem reguł formatowania (omówimy je już za chwilę), których używa do generowania instrukcji formatowania przekazywanych do `Out-Host`.
6. Kiedy `Out-Host` zauważy, że otrzymał instrukcje formatowania, postępuje zgodnie z nimi, przygotowując dane do wyświetlenia na ekranie.

Wszystko to dzieje się również w sytuacji, kiedy ręcznie użyjesz jakiegoś polecenia z rodziny Out-. Na przykład spróbuj uruchomić polecenie `Get-Process | Out-File procs.txt`. Polecenie Out-File „zorientuje się”, że wysłałeś do niego kilka normalnych obiektów. Przekaze je do systemu formatowania, który tworzy instrukcje formatowania i odsyła je z powrotem do cmdletu Out-File, a cmdlet ten na ich podstawie tworzy plik tekstowy. Jak widać, system formatowania jest angażowany za każdym razem, gdy obiekty muszą zostać przekształcone w czytelne dla użytkownika wyniki tekstowe.

Jakich reguł używa system formatowania w kroku 5.? W przypadku pierwszej reguły formatowania system sprawdza, czy typ obiektu, z którym ma do czynienia, ma predefiniowany widok. To właśnie widziałeś w pliku *DotNetTypes.format.ps1xml*: predefiniowany widok dla obiektu `-Process`. Z powłoką PowerShell instalowanych jest kilka innych plików *.format.ps1xml*, które są ładowane domyślnie po uruchomieniu powłoki. W razie potrzeby możesz również tworzyć własne predefiniowane widoki, ale takie zagadnienia wykraczają już daleko poza zakres tej książki.

System formatowania szuka predefiniowanego widoku przeznaczonego dla obiektów, z którymi ma do czynienia. W naszym przykładzie system szuka widoku, który obsługuje obiekty typu `System.Diagnostics.Process`.

A co się dzieje, jeżeli odpowiedni predefiniowany obiekt nie zostanie znaleziony? Na przykład spróbuj uruchomić następujące polecenie:

```
Get-WmiObject Win32_OperatingSystem | Gm
```

Zapamiętaj typ obiektu (lub przynajmniej część jej nazwy, `Win32_OperatingSystem`) i spróbuj znaleźć go w jednym z plików *.format.ps1xml*. Stop. Zaoszczędzimy Ci trochę czasu, z góry mówiąc, że go tam nie znajdziesz.

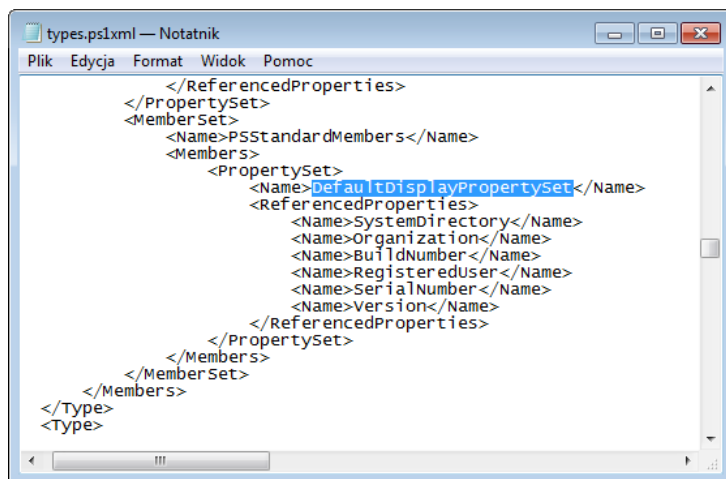
W tym miejscu system formatujący przechodzi do następnego kroku lub — inaczej mówiąc — do tego, co nazywamy **drugą regułą formatowania**: sprawdza, czy ktokolwiek zadeklarował domyślne właściwości wyświetlania dla obiektów tego typu. Znajdziesz je w pliku konfiguracyjnym o nazwie *Types.ps1xml*. Otwórz go teraz w Notatniku (tak jak poprzednio, uważaj, aby nie zapisać żadnych zmian w tym pliku) i użyj funkcji *Znajdź*, aby zlokalizować sekcję `Win32_OperatingSystem`. Przewiń w dół, aż zobaczysz `DefaultDisplayPropertySet`, tak jak to zostało pokazane na rysunku 10.2. Zanutuj sekcję wymienionych tam właściwości.

Teraz wróć do powłoki PowerShell i uruchom następujące polecenie:

```
Get-WmiObject Win32_OperatingSystem
```

Czy wyniki działania tego polecenia wyglądają znajomo? Powinny: wyświetlone właściwości są widoczne tylko dlatego, że zostały wymienione jako domyślne w pliku *Types.ps1xml*. Jeżeli system formatowania znajdzie predefiniowany zestaw domyślnych właściwości do wyświetlania, użyj go. Jeżeli nie znajdzie takiego zestawu, następna reguła formatowania będzie brała pod uwagę wszystkie właściwości obiektu.

Wspomniana następna reguła, czyli inaczej mówiąc **trzecia reguła formatowania**, decyduje o tym, w jaki sposób będą wyświetlane wyniki działania. Jeżeli mechanizm



Rysunek 10.2. Właściwość DefaultDisplayPropertySet wyświetlona w programie Notatnik

formatowania wyświetla cztery lub mniej właściwości, wyniki zostaną przedstawione w postaci tabeli. Jeżeli wyświetlanych będzie pięć lub więcej właściwości, wyniki będą przedstawiane w postaci listy. Z tego właśnie względu obiekt Win32 _OperatingSystem nie był wyświetlany jako tabela: jego sześć właściwości spowodowało wybranie listy. Teoria „mówi”, że przy wyświetlaniu wyników w postaci tabel tworzonych ad hoc więcej niż cztery właściwości mogą się po prostu nie zmieścić bez ograniczania ilości wyświetlanych informacji.

Teraz już wiesz, jak działa formatowanie domyślne. Wiesz również, że większość poleceń z rodziny Out- automatycznie uruchamia mechanizm formatowania, dzięki czemu może uzyskać odpowiednie instrukcje formatowania niezbędne do wyświetlenia wyników działania. Teraz przyjrzyjmy się, jak możesz samodzielnie kontrolować działanie mechanizmu formatowania i modyfikować jego ustawienia domyślne.

A tak przy okazji, to właśnie mechanizm formatowania jest powodem, dla którego PowerShell czasami wydaje się „kłamać”. Na przykład uruchom polecenie Get-Process i spójrz na nagłówki kolumn. Zauważyłeś kolumnę oznaczoną jako PM(K)? Cóż, to właśnie jest owo wspomniane „klamstwo”, ponieważ w rzeczywistości nie ma właściwości o nazwie PM(K). Istnieje za to właściwość o nazwie PM. Wniosek z tego jest taki, że sformatowane nagłówki kolumn są wyłącznie nagłówkami kolumn i nie muszą być takie same jak pełne nazwy odpowiadających im właściwości. Jedynym bezpiecznym sposobem na wyświetlenie prawdziwych, pełnych nazw właściwości jest użycie polecenia Get-Member.

10.3. Formatowanie tabel

PowerShell ma cztery polecenia przeznaczone do formatowania wyników działania, a my będziemy tutaj korzystać z trzech, które są najczęściej wykorzystywane podczas codziennej pracy (czwarte polecenie zostało krótko omówione w sekcji „Dla zainteresowanych” pod

koniec tego rozdziału). Rozpocznijmy od przedstawienia polecenia `Format-Table`, które posiada alias o nazwie `Ft`.

Po zapoznaniu się z treścią pliku pomocy polecenia `Format-Table` zauważysz, że posiada ono cały szereg parametrów wywołania. Poniżej przedstawiamy zestawienie kilku najbardziej użytecznych parametrów wraz z krótkimi przykładami ich użycia:

- `-autoSize` — domyślnie powłoka PowerShell próbuje dopasować szerokość wyświetlanej tabeli do szerokości okna (z wyjątkiem sytuacji, gdy domyślne szerokości kolumn są określone w predefiniowanym widoku wyników działania, takim jak przy wyświetlaniu procesów). Jednak w przypadku niektórych tabel z niewielką liczbą kolumn może to powodować powstawanie dużej ilości wolnego miejsca między kolejnymi kolumnami, co nie zawsze będzie dobrze wyglądać. Dodając parametr `-autosize`, zmuszasz powłokę, aby próbowała dopasować rozmiary poszczególnych kolumn do ich zawartości, i nic więcej. Powoduje to, że wyświetlana tabela jest „bardziej zwarta” (aczkolwiek przygotowanie takiej tabeli zajmuje nieco więcej czasu). Dzieje się tak, ponieważ powłoka musi sprawdzić wszystkie obiekty w wynikach działania danego polecenia i znaleźć najdłuższe wartości dla każdej kolumny. Spróbuj wykonać przedstawione poniżej polecenie najpierw z parametrem `-autosize`, a następnie bez niego:

```
Get-WmiObject Win32_BIOS | Format-Table -autoSize
```

- `-property` — wartością tego parametru jest rozdzielana przecinkami lista właściwości, które powinny znaleźć się w wyświetlanej tabeli. W nazwach tych właściwości wielkość liter nie jest rozróżniana, ale powłoka będzie używać podanych nazw jako nagłówków wyświetlanych kolumn, stąd stosując odpowiednią pisownię, możesz uzyskać lepiej wyglądające wyniki działania (na przykład wpisując `CPU` zamiast `cpu`). Parametr `-property` akceptuje symbole wieloznaczne, co oznacza, że możesz użyć znaku `*`, aby uwzględnić w tabeli wszystkie właściwości, lub na przykład `c*`, aby uwzględnić wszystkie właściwości, których nazwy rozpoczynają się od litery `c`. Zauważ, że powłoka nadal będzie wyświetlać tylko właściwości, które zmieszczą się w tabeli, więc może się zdarzyć, że nie każda wybrana właściwość zostanie wyświetlona. Parametr `-property` jest pozycjonowany, więc nie trzeba wpisywać jego nazwy, o ile oczywiście lista właściwości znajduje się na pierwszej pozycji w wierszu wywołania polecenia. Wypróbuj przykładowe polecenia przedstawione poniżej (wyniki działania ostatniego z nich zostały pokazane na rysunku 10.3):

```
Get-Process | Format-Table -property *
```

```
Get-Process | Format-Table -property ID,Name,Responding -autoSize
```

```
Get-Process | Format-Table * -autoSize
```

- `-groupBy` — ten parametr generuje nowy zestaw nagłówków kolumn za każdym razem, gdy zmienia się wartość określonej właściwości. Takie rozwiązanie będzie poprawnie działać tylko wtedy, kiedy najpierw posortujemy obiekty według grupującej właściwości. Aby przekonać się, jak to działa w praktyce, spróbuj samodzielnie wykonać następujące polecenie:

```
Get-Service | Sort-Object Status | Format-Table -groupBy Status
```


Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
38	6	1984	4552	57	43.88	2196	conhost
31	4	796	2272	22	0.14	2508	conhost
29	4	832	2300	41	2.27	2888	conhost
32	5	976	2904	46	0.11	3236	conhost
506	13	1900	4064	48	3.78	320	csrss
213	12	7928	5892	53	54.17	372	csrss
296	30	14576	19516	143	20.83	1300	dfsrs
122	15	2584	6188	41	0.63	1760	dfssvc
5157	7329	85720	87088	122	4.48	1356	dns
65	7	1824	4684	53	0.25	324	dwm
669	40	27176	41668	174	8.28	2100	explorer
129	9	3032	5056	38	8.95	2500	fdhost
48	6	1020	3244	25	0.02	2432	fdlauncher
0	0	0	24	0		0	Idle
134	14	5760	11684	68	0.08	1420	inetinfo
100	14	2988	4876	39	0.13	1464	ismserv
1332	111	34368	30308	164	67.03	484	lsass
194	11	2820	5676	30	7.55	492	lsm
308	42	52244	52348	559	11.03	1236	Microsoft.ActiveDirectory...
146	18	3228	7180	60	0.09	2576	msdtc
1793	44	473460	492088	1010	53.44	3028	powershell
545	54	128788	133564	766	224.25	3760	powershell_ise

Rysunek 10.3. Wyświetlanie listy procesów z dopasowaniem szerokości kolumn do ich zawartości

- -wrap — powoduje, że jeżeli powłoka przycina wartość właściwości do szerokości kolumny, umieszcza na jej końcu ciąg znaków (...), wskazując w ten sposób, że dane w tym miejscu zostały obcięte. Użycie tego parametru umożliwia powłoce zawijanie informacji, co powoduje, że tabela jest dłuższa, ale zachowuje wszystkie informacje, które chcesz wyświetlić. Oto przykład:

```
Get-Service | Format-Table Name,Status,DisplayName -autoSize -wrap
```

ZRÓB TO SAM Wykonaj samodzielnie wszystkie przykłady prezentowane w tym rozdziale i uważnie przyjrzyj się otrzymywanym wynikom działania. Pamiętaj, że możesz łączyć ze sobą poszczególne techniki. Spróbuj poeksperymentować z różnymi parametrami, aby zobaczyć, jak to działa i jakie wyniki możesz otrzymać.

10.4. Formatowanie list

Czasami jednak musimy wyświetlić znacznie więcej informacji, niż może się zmieścić w poziomej tabeli. W takich sytuacjach bardzo przydatnym rozwiązaniem staje się formatowanie wyników działania w postaci listy, co możesz zrobić przy użyciu polecenia `Format-List` lub jego aliasu `Fl`.

Polecenie `Format-List` obsługuje podobną listę parametrów co polecenie `Format-Table`, włącznie z parametrem `-property`. W praktyce polecenie `Fl` jest po prostu innym sposobem wyświetlania właściwości obiektów. `Fl`, w odróżnieniu od polecenia `Gm`, wyświetla również wartości poszczególnych właściwości, dzięki czemu możemy zobaczyć zawarte w nich informacje:

```
Get-Service | Fl *
```

Na rysunku 10.4 przedstawiono przykładowe wyniki działania tego polecenia. Warto zauważyć, że polecenia `Fl` często używamy jako alternatywnej metody sprawdzania właściwości obiektu.

```

Administrator: Windows PowerShell
ServiceName       : NlaSvc
ServicesDependedOn : {RpcSs, TcpIp, NSI}
ServiceHandle     : SafeServiceHandle
Status            : Running
ServiceType       : Win32ShareProcess
Site              :
Container         :

Name              : nsi
RequiredServices  : {nsiproxy}
CanPauseAndContinue : False
CanShutdown       : False
CanStop           : True
DisplayName       : Network Store Interface Service
DependentServices : {netprofm, NlaSvc, SharedAccess, Netman...}
MachineName       : .
ServiceName       : nsi
ServicesDependedOn : {nsiproxy}
ServiceHandle     : SafeServiceHandle
Status            : Running
ServiceType       : Win32ShareProcess
Site              :
Container         :

Name              : NTDS
PS C:\>

```

Rysunek 10.4. Wyświetlanie działających usług w postaci listy

ZRÓB TO SAM Zapoznaj się z zawartością pliku pomocy dla polecenia `Format-`

`List` i spróbuj samodzielnie poeksperymentować z jego różnymi parametrami wywołania.

10.5. Formatowanie szerokich list

Ostatnie polecenie, `Format-Wide` (lub jego alias `Fw`), wyświetla wyniki w postaci szerokiej, wielokolumnowej listy. Polecenie to może wyświetlać wartości tylko jednej właściwości jednocześnie, więc jej parametr `-property` akceptuje tylko jedną nazwę właściwości, a nie ich listę, i nie może akceptować symboli wieloznacznych.

Domyślnie polecenie `Format-Wide` poszukuje właściwości `Name` obiektu, ponieważ `Name` jest powszechnie używaną właściwością i zwykle zawiera bardzo użyteczne informacje. Polecenie to wyświetla domyślnie dwie kolumny, ale w razie potrzeby możemy użyć parametru `-columns` do określenia większej liczby kolumn:

```
Get-Process | Format-Wide name -col 4
```

Na rysunku 10.5 przedstawiono przykładowe wyniki działania tego polecenia:

```

PS C:\> get-process | format-wide name -col 4

conhost      conhost      conhost      conhost
csrss        csrss        dfsrc        dfsrc
dns          dwm          explorer     fdsrc
fdlauncher   Idle         inetinfo     ismserv
lsass        lsm          Microsoft.Activ... msdtc
powershell   powershell_ise PresentationFon... services
smss         spoolsv      sqlservr     sqlwriter
svchost      svchost      svchost      svchost
svchost      svchost      svchost      svchost
svchost      svchost      svchost      svchost
svchost      System       taskhost     TPMAutoConnect
TPMAutoConnSvc vds         vmtoolsd     VMUpgradeHelper
VMWareTray   VMwareUser   wininit      winlogon
WmiPrvSE

PS C:\>

```

Rysunek 10.5. Wyświetlanie nazw procesów w postaci szerokiej listy

ZRÓB TO SAM Zapoznaj się z zawartością pliku pomocy polecenia `Format-Wide` i spróbuj samodzielnie poeksperymentować z jego parametrami wywołania.

10.6. Tworzenie niestandardowych kolumn i elementów list

Wróć na chwilę do poprzedniego rozdziału i zajrzyj do podrozdziału 9.5. Pokazywaliśmy tam, jak użyć konstrukcji tablicy mieszającej, aby dodawać nowe, niestandardowe właściwości do obiektu. Zarówno polecenie `Format-Table`, jak i `Format-List` mogą używać tych samych konstrukcji do tworzenia niestandardowych kolumn tabeli lub niestandardowych wpisów na liście.

Takich rozwiązań możesz na przykład użyć do utworzenia nagłówka kolumny różniącego się od nazwy wyświetlanej właściwości:

```

Get-Service |
Format-Table @{name='ServiceName';expression={$_.Name}},Status,DisplayName

```

Zamiast tego możesz na przykład utworzyć bardziej złożone wyrażenie matematyczne:

```

Get-Process |
Format-Table Name,@{name='VM(MB)';expression={$_.VM / 1MB -as [int]}} -autosize

```

Na rysunku 10.6 przedstawiono wyniki działania poprzedniego polecenia. Musimy jednak przyznać, że w powyższym poleceniu użyliśmy kilku elementów, o których jeszcze nie mówiliśmy, możemy zatem zrobić to teraz:

```

Administrator: Windows PowerShell
PS C:\> get-process | format-table name,@{l='VM(MB)';e={$_.VM / 1MB -as [int]}} -autosize

```

Name	VM(MB)
----	-----
conhost	57
conhost	22
conhost	41
conhost	46
csrss	48
csrss	53
dfsr	143
dfssvc	41
dns	122
dwm	53
explorer	173
fdhost	38
fdlauncher	25
Idle	0
inetinfo	67
ismserv	38
lsass	164
lsim	30
Microsoft.ActiveDirectory.WebServices	559
msdtc	60
powershell	1010

Rysunek 10.6. Tworzenie własnej niestandardowej kolumny obliczeniowej

- Rozpoczynamy oczywiście od polecenia `Get-Process`, które dobrze już znasz. Jeżeli uruchomisz polecenie `Get-Process | FL *`, zobaczysz, że wartość właściwości `VM` wyrażana jest w bajtach, choć w domyślnym widoku tabeli tak nie jest.
- Informujemy polecenie `Format-Table`, aby wyświetlanie wyników działania rozpoczęło od właściwości `Name` procesów.
- Następnie używamy specjalnej tablicy mieszającej do utworzenia niestandardowej kolumny, która będzie oznaczona jako `VM(MB)`. Odpowiada za to pierwsza część polecenia, aż do średnika, który jest separatorem. Druga część definiuje wartość lub inaczej mówiąc, wyrażenie dla tej kolumny, biorąc normalną wartość właściwości `VM` obiektu i dzieląc ją przez 1 MB. W powłoce PowerShell prawy ukośnik jest operatorem dzielenia, a powłoka automatycznie rozpoznaje skróty KB, MB, GB, TB i PB jako oznaczające, odpowiednio, kilobajt, megabajt, gigabajt, terabajt i petabajt.
- Wynik tej operacji podziału będzie zapewne zawierał część dziesiętną, której nie chcemy wyświetlać. Operator `-as` pozwala nam zmienić typ danych wyniku z wartości zmiennoprzecinkowej na, w tym przypadku, wartość całkowitą (określoną przez `[int]`). Powłoka podczas dokonywania konwersji automatycznie zaokrągli tę wartość, odpowiednio, w górę lub w dół. Wynikiem operacji jest liczba całkowita bez składowej ułamkowej.

Pokazujemy tę małą sztuczkę z dzieleniem i zmianą typu danych, ponieważ może być przydatna w tworzeniu efektownie wyglądających wyników działania poleceń. W tej książce nie poświęcimy już więcej czasu na takie operacje (choćbyśmy Ci, że w powłoce PowerShell znak `*` jest operatorem mnożenia i — jak można się było spodziewać — znaki `+` i `-` są, odpowiednio, operatorami dodawania i odejmowania).

Dla zainteresowanych

Spróbuj samodzielnie wykonać następujące polecenie:

```
Get-Process |
```

```
Format-Table Name,@{name='VM(MB)';expression={$_.VM / 1MB -as [int]}} -autosize
```

Tym razem jednak nie wpisuj wszystkiego w jednym wierszu. Zamiast tego wpisz polecenie dokładnie tak, jak pokazano tutaj w książce, w trzech wierszach. Zwróć uwagę, że po wpisaniu pierwszego wiersza, który kończy się znakiem potoku, powłoka PowerShell zmienia znak zachęty. Dzieje się tak dlatego, że zakończyłeś bieżący wiersz znakiem potoku, co dla powłoki jest sygnałem, że dalej powinna oczekiwać kolejnych poleceń. Powłoka wchodzi w ten sam tryb „oczekiwania na zakończenie polecenia”, kiedy naciśniesz klawisz *Enter* bez odpowiedniego zamknięcia wszystkich nawiasów klamrowych, cudzysłowów i nawiasów okrągłych.

Jeżeli nie chcesz wchodzić w rozszerzony tryb wprowadzania poleceń, naciśnij kombinację klawiszy *Ctrl+C*, aby przerwać wprowadzanie, i zacznij od nowa. W naszym przypadku po zakończeniu wpisywania pierwszego wiersza i pojawieniu się nowego znaku zachęty wpisz drugi wiersz, naciśnij klawisz *Enter*, a następnie wpisz trzeci wiersz i ponownie naciśnij klawisz *Enter*. Będąc w tym trybie, musisz po zakończeniu wpisywania nacisnąć klawisz *Enter* jeszcze raz, w pustym wierszu, co jest dla powłoki sygnałem zakończenia wprowadzania polecenia. Kiedy to zrobisz, powłoka wykona całe polecenie tak, jakby zostało wpisane w pojedynczym długim wierszu.

W przeciwieństwie do polecenia *Select-Object*, którego tablice mieszające mogą przyjmować tylko klucze *Name* i *Expression* (choć akceptują również klucze *N*, *L* i *Label* dla *Name* oraz *E* dla *Expression*), polecenia z rodziny *Format-* mogą obsługiwać dodatkowe klucze, które są przeznaczone do sterowania wyświetlaniem na ekranie. Takie dodatkowe klucze są najbardziej przydatne w przypadku polecenia *Format-Table*:

- *FormatString* pozwala na zdefiniowanie ciągu formatującego, powodującego wyświetlanie danych w określonym formacie. Jest to szczególnie przydatne w przypadku danych numerycznych i daty. Więcej szczegółowych informacji na temat ciągów formatujących dane liczbowe i daty oraz pozwalających na niestandardowe formatowanie liczb i dat znajdziesz na stronie MSDN pod adresem [https://msdn.microsoft.com/en-us/library/fbxf59x\(v=vs.95\).aspx](https://msdn.microsoft.com/en-us/library/fbxf59x(v=vs.95).aspx).
- *Width* określa żądaną szerokość kolumny.
- *Alignment* określa żądane wyrównanie kolumn, do lewej (*Left*) lub do prawej (*Right*).

Używanie tych dodatkowych kluczy ułatwia osiągnięcie wyników przedstawionych w poprzednim przykładzie, a nawet ich poprawienie:

```
Get-Process |
```

```
Format-Table Name,@{name='VM(MB)';expression={$_.VM};formatstring='F2';align='right'} -autosize
```

Teraz nie musimy już wykonywać operacji dzielenia, ponieważ powłoka PowerShell automatycznie sformatuje liczbę jako wartość stałoprzecinkową z dwoma miejscami dziesiętnymi, a sam wynik zostanie wyrównany do prawej.

10.7. Przesyłanie danych na wyjście: do pliku, drukarki lub na ekran

Po sformatowaniu wyników działania musisz zdecydować, dokąd zostaną one przesłane.

Jeżeli wiersz polecenia kończy się cmdletem z rodziny `Format-`, wygenerowane przez niego instrukcje formatowania przesyłane są do polecenia `Out-Default`, które przekazuje je do polecenia `Out-Host` wyświetlającego wyniki na ekranie:

```
Get-Service | Format-Wide
```

Instrukcje formatowania możesz również ręcznie przesłać do polecenia `Out-Host`, co przyniesie dokładnie takie same rezultaty jak w poprzednim przykładzie:

```
Get-Service | Format-Wide | Out-Host
```

Alternatywnym rozwiązaniem jest przekazywanie instrukcji formatowania polecenia `Out-File` lub `Out-Printer`, co spowoduje przesłanie sformatowanych danych wyjściowych do pliku lub na drukarkę. Jak się przekonasz w podrozdziale 10.9, w dowolnym wierszu polecenia po poleceniu z rodziny `Format-` powinno występować tylko jedno z trzech poleceń z rodziny `Out-`.

Pamiętaj, że zarówno polecenie `Out-Printer`, jak i `Out-File` mają określoną domyślną szerokość wydruku (wyrażoną w znakach), co oznacza, że na wydruku lub w pliku tekstowym wyniki działania mogą wyglądać inaczej niż na ekranie. Wymienione polecenia mają parametr `-width`, który umożliwia zmianę szerokości danych na wyjściu, co w razie potrzeby pozwala na dostosowanie formatowania do szerszych tabel.

10.8. Jeszcze jeden typ wyjścia: *GridViews*

Polecenie `Out-GridView` udostępnia jeszcze jeden, bardzo przydatny rodzaj wyjścia danych. Zwróć uwagę, że — technicznie rzecz biorąc — nie chodzi tutaj o formatowanie danych wyjściowych; w rzeczywistości polecenie `Out-GridView` całkowicie pomija podsystem formatowania. Nie są tutaj wywoływane żadne polecenia z rodziny `Format-`, nie są tworzone instrukcje formatowania i w oknie konsoli nie jest wyświetlany żaden tekst. Polecenie `Out-GridView` nie może odbierać danych wyjściowych z cmdletów `Format-`; zamiast tego może odbierać tylko zwykłe obiekty przesyłane przez inne polecenia. Polecenie `Out-GridView` może nie działać w systemach innych niż Windows.

Na rysunku 10.7 przedstawiono przykładowe wyniki działania tego polecenia:

10.9. Najczęściej spotykane problemy

Jak wspomnieliśmy na początku tego rozdziału, w podsystemie formatowania jest wiele pułapek czyhających na początkujących, niedoświadczonych użytkowników powłoki PowerShell. Nasi studenci podczas szkoleń najczęściej borykają się z dwoma problemami, więc postaramy Ci się je przybliżyć, tak abyś mógł ich uniknąć.

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
38	6	1,094	4,552	57	46.94	2,195	conhost
31	4	795	2,372	22	0.14	2,508	conhost
29	4	832	2,300	41	2.27	2,888	conhost
32	5	976	2,904	46	0.11	3,236	conhost
492	13	1,900	4,060	48	3.80	320	csrss
209	12	7,928	5,892	53	55.63	372	csrss
296	30	14,572	19,512	143	20.86	1,300	dfsvc
133	15	2,636	6,204	42	0.64	1,760	dfsvc
5,158	7,329	85,444	87,068	121	4.50	1,356	dns
65	7	1,824	4,684	53	0.25	324	dwm
669	40	27,100	41,652	173	8.30	2,100	explorer
129	9	3,032	5,056	38	9.02	2,500	fdhost
48	6	1,020	3,244	25	0.02	2,432	fdlauncher
0	0	0	0	0	0	0	Idle
134	14	5,708	11,668	67	0.08	1,420	inetinfo
98	13	2,904	4,860	38	0.13	1,464	iiserv
1,304	108	34,300	30,292	163	67.31	484	lsass
194	11	2,820	5,676	30	7.58	492	lsmon
308	42	52,240	52,348	559	11.06	1,236	Microsoft.ActiveDirectory.WebServices
146	18	3,228	7,180	60	0.09	2,576	msdtc
1,730	46	475,944	494,092	1,028	54.67	3,028	powershell
545	54	128,788	133,564	766	224.25	3,760	powershell_ise
147	24	26,068	17,928	505	0.17	2,812	PresentationFontCache
305	20	11,532	11,944	116	3.14	476	services
29	2	368	960	5	0.08	216	smss
326	26	9,296	16,824	106	1,335.00	1,204	spoolsv
365	141	137,200	75,840	758	14.42	1,536	sqlservr
77	9	1,684	5,884	42	0.06	1,656	sqlwriter
279	32	10,464	13,112	54	7.11	332	svchost
343	14	3,584	8,552	45	1.70	636	svchost
278	19	3,640	7,916	39	2.94	720	svchost
326	16	9,276	12,104	48	8.48	804	svchost
898	38	19,364	33,412	150	11.84	852	svchost
276	21	5,500	10,432	44	2.27	900	svchost

Rysunek 10.7. Przykładowe wyniki działania polecenia Out-GridView

10.9.1. Formatuj dane na końcu

Niezwykle ważną sprawą jest to, abyś z tego rozdziału zapamiętał jedną, fundamentalną zasadę — *formatuj dane na końcu*. Polecenie z rodziny Format- powinno być ostatnim elementem wiersza poleceń (jedynymi wyjątkami od tej reguły mogą być polecenia Out-File lub Out-Printer). Wynika to stąd, że polecenia z rodziny Format- generują instrukcje formatowania, które mogą być właściwie interpretowane tylko przez polecenia z rodziny Out-. Jeżeli polecenie Format- jest ostatnim elementem wiersza poleceń, instrukcje formatowania są przekazywane do polecenia Out-Default (które zawsze domyślnie i niejawnie znajduje się na końcu potoku), a z jego wyjścia są przesyłane do polecenia Out-Host, które z instrukcjami formatowania radzi sobie znakomicie.

Aby przekonać się, jak wspomniana wyżej reguła formatowania działa w praktyce, powinieneś wykonać następujące polecenie:

```
Get-Service | Format-Table | Gm
```

Zwróć uwagę, że w tym przypadku polecenie Gm nie wyświetla informacji o obiektach usług, ponieważ polecenie Format-Table nie generuje takich obiektów. Zamiast tego polecenie Format-Table pobiera obiekty usług i wyprowadza instrukcje formatowania, które zostają uwidocznione w wynikach działania polecenia Gm, tak jak to zostało pokazane na rysunku 10.8:

```

Administrator: Windows PowerShell

TypeName: Microsoft.PowerShell.Commands.Internal.Format.GroupEndData

Name                                     MemberType Definition
----
Equals                                  Method      bool Equals(System.Object)
GetHashCode                             Method      int GetHashCode()
GetType                                Method      type GetType()
ToString                               Method      string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property    System.String ClassId
groupingEntry                           Property    Microsoft.PowerShell.Commands.Internal.Format.GroupEndData

TypeName: Microsoft.PowerShell.Commands.Internal.Format.FormatEndData

Name                                     MemberType Definition
----
Equals                                  Method      bool Equals(System.Object)
GetHashCode                             Method      int GetHashCode()
GetType                                Method      type GetType()
ToString                               Method      string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property    System.String ClassId
groupingEntry                           Property    Microsoft.PowerShell.Commands.Internal.Format.FormatEndData

PS C:\>

```

Rysunek 10.8. Polecenia formatujące generują specjalne instrukcje formatowania, które nie mają żadnego znaczenia dla użytkownika

Teraz spróbuj wykonać następujące polecenie:

```
Get-Service | Select Name,DisplayName,Status | Format-Table | ConvertTo-HTML | Out-File
services.html
```

Otwórz nowo utworzony plik `Services.html` w przeglądarce Internet Explorer, a przekonasz się, że zawiera on wyniki zupełnie na pozór pozbawione sensu. Stało się tak dlatego, że do polecenia `ConvertTo-HTML` nie przekazywaliśmy obiektów reprezentujących usługi — zamiast tego przesłane zostały instrukcje formatowania, które następnie zostały zapisane w pliku HTML. Taka sytuacja doskonale ilustruje, dlaczego polecenia z rodziny `Format-` (jeżeli ich używasz) muszą być ostatnim elementem w wierszu polecenia (lub przedostatnim, jeżeli ostatnim poleceniem jest `Out-File` lub `Out-Printer`).

Wiemy również, że polecenie `Out-GridView` jest dosyć nietypowe (przynajmniej dla rodziny poleceń `Out-`), ponieważ nie akceptuje instrukcji formatowania i pobiera tylko standardowe obiekty. Wypróbuj dwa polecenia przedstawione poniżej, aby przekonać się, na czym polega różnica:

```
PS C:\> Get-Process | Out-GridView
PS C:\> Get-Process | Format-Table | Out-GridView
```

Właśnie dlatego wyraźnie wymieniliśmy polecenia `Out-File` i `Out-Printer` jako jedyne cmdlety, które w wierszu polecenia mogą znajdować się za poleceniami z rodziny `Format-` (pod względem technicznym cmdlet `Out-Host` może również znajdować się za poleceniami `Format-`, ale nie ma takiej potrzeby, ponieważ zakończenie wiersza poleceń cmdletem z rodziny `Format-` i tak spowoduje przekazanie jego wyników działania do cmdletu `Out-Host`).

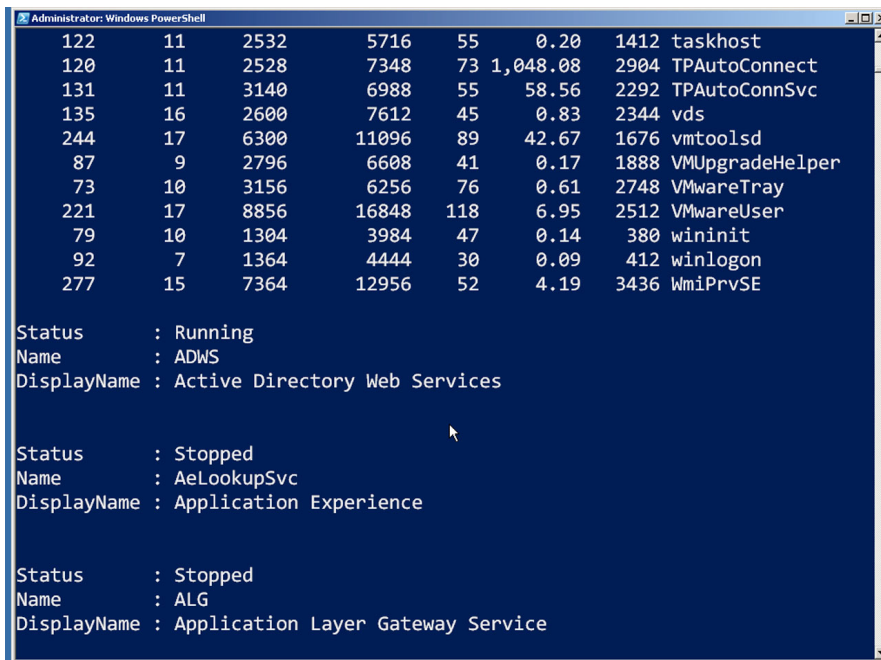
10.9.2. Jeden typ obiektu jednocześnie

Następną rzeczą, której należy unikać, jest jednoczesne umieszczanie w potoku różnych rodzajów obiektów. Podsystem formatowania analizuje pierwszy obiekt w potoku i używa jego typu do określenia sposobu formatowania. Jeżeli potok zawiera dwa typy obiektów lub ich większą liczbę, dane wyjściowe nie zawsze będą kompletne lub użyteczne.

Przykładowo, spróbuj uruchomić następujące polecenie:

```
Get-Process; Get-Service
```

Użycie średnika pozwala na umieszczenie dwóch poleceń w jednym wierszu poleceń bez wyprowadzania wyjścia pierwszego polecenia do drugiego. Oznacza to, że oba polecenia *cmdlet* działają niezależnie, ale przesyłają dane wyjściowe do tego samego potoku. Jeżeli samodzielnie spróbujesz wykonać taki wiersz polecenia lub po prostu spojrzysz na rysunek 10.9, przekonasz się, że początkowo wyniki działania wydają się poprawne i reprezentują obiekty procesów. Ale wyniki działania raptownie zmieniają się, gdy nadejdzie czas wyświetlania listy działających usług. Zamiast wyświetlania tabeli, do której jesteś przyzwyczajony, powłoka PowerShell powraca do wyświetlania wyników w postaci listy. System formatowania nie jest przystosowany do obsługi wielu rodzajów obiektów jednocześnie i próbuje sprawić, aby wyniki działania wyglądały tak atrakcyjnie, jak to tylko możliwe.



Rysunek 10.9. Umieszczenie dwóch rodzajów obiektów w potoku jednocześnie może zdezorientować mechanizm formatowania powłoki PowerShell

A co powinieneś zrobić, jeżeli chcesz połączyć informacje pochodzące z dwóch (lub więcej) źródeł w jeden spójny wynik? Jest to jak najbardziej możliwe, a co więcej, możesz to zrobić tak, że system formatowania znakomicie sobie z tym poradzi. Niestety jest to dosyć zaawansowane zagadnienie, które wykracza już poza ramy tej książki.

10.10. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Dla zainteresowanych

Z technicznego punktu widzenia podsystem formatowania może obsługiwać wiele typów obiektów — pod warunkiem, że powiesz mu, jak to zrobić. Przykładowo, kiedy wykonasz polecenie `Dir | Gm`, zauważysz, że potok zawiera zarówno obiekty typu `DirectoryInfo`, jak i `FileInfo` (polecenie `Gm` nie ma problemu z pracą z potokami, które zawierają wiele rodzajów obiektów, i poprawnie wyświetla informacje o elementach składowych wszystkich typów obiektów znajdujących się w potoku). Kiedy uruchamiasz samo polecenie `Dir`, jego wyniki działania są bardzo czytelne. Dzieje się tak dlatego, że firma Microsoft udostępniła predefiniowany niestandardowy widok formatowania dla obiektów typu `DirectoryInfo` i `FileInfo`, który jest obsługiwany przez polecenie `Format-Custom`.

Polecenie `Format-Custom` służy głównie do wyświetlania różnych predefiniowanych widoków niestandardowych. Teoretycznie możesz tworzyć swoje własne predefiniowane widoki niestandardowe, ale niezbędna składnia opisu w formacie XML jest bardzo skomplikowana i obecnie nie jest publicznie udokumentowana, zatem, póki co, widoki niestandardowe są ograniczone do takich, które udostępniła firma Microsoft.

Nie zmienia to jednak w niczym faktu, że niestandardowe widoki Microsoftu są bardzo użyteczne. Przykładowo, informacje pomocy powłoki PowerShell są przechowywane w postaci obiektów, a sformatowane pliki pomocy wyświetlane na ekranie są wynikiem przekazywania tych obiektów do odpowiedniego niestandardowego widoku.

Spróbuj samodzielnie wykonać następujące zadania:

1. Wyświetl tabelę działających procesów, która będzie zawierała tylko nazwy procesów, ich identyfikatory oraz to, czy reagują na żądania z systemu Windows (odpowiednia informacja jest przechowywana przez właściwość `Responding`). Sformatuj wyniki działania tak, aby tabela wynikowa zajmowała jak najmniej miejsca w poziomie, ale nie pozwól, aby jakiegokolwiek informacje zostały obcięte.
2. Wyświetl tabelę procesów, która zawiera nazwy działających procesów i ich identyfikatory. Uwzględnij także kolumny zawierające informacje o wykorzystaniu pamięci wirtualnej i fizycznej, wyrażając te wartości w megabajtach (MB).
3. Użyj polecenia `Get-EventLog` (tylko dla systemu Windows), aby wyświetlić listę dostępnych dzienników zdarzeń. (Wskazówka: zapoznaj się z zawartością pliku pomocy dla tego polecenia, aby znaleźć odpowiedni parametr wywołania polecenia). Sformatuj dane wyjściowe jako tabelę, która zawiera kolejno nazwę dziennika i okres przechowywania. Nagłówki kolumn powinny nosić nazwy `LogName` i `RetDays`.

4. Wyświetl listę usług w taki sposób, aby uruchomione usługi były wyświetlane w jednej tabeli, a zatrzymane usługi w drugiej. Uruchomione usługi powinny zostać wyświetlone jako pierwsze. (Wskazówka: użyj parametru `-groupBy`).
5. Wyświetl listę nazw wszystkich podkatalogów znajdujących się w katalogu głównym dysku C:; lista powinna zostać wyświetlona w czterech kolumnach.
6. Utwórz sformatowaną listę wszystkich plików `.exe` znajdujących się w katalogu `C:\Windows`, zawierającą ich nazwy, informacje o wersji oraz rozmiar pliku. Półka PowerShell używa właściwości o nazwie `Length`, ale aby zwiększyć przejrzystość prezentowanych wyników, w Twoim przypadku odpowiednia kolumna powinna nosić nazwę `Size`.

10.11. Co dalej?

Teraz nadszedł doskonały czas na samodzielne eksperymentowanie z systemem formatowania. Spróbuj użyć trzech głównych poleceń rodziny `Format-` do tworzenia danych wyjściowych w różnych formatach. Ćwiczenia w kolejnych rozdziałach będą często wymagały użycia określonego formatowania, więc powinieneś przez cały czas doskonalić swoje umiejętności i nabierać doświadczenia w stosowaniu tych poleceń oraz ich parametrów.

10.12. Odpowiedzi

1. `get-process | format-table Name,ID,Responding -autosize -Wrap`
2. `get-process | format-table Name,ID,
@{1='VirtualMB';e={$_.vm/1mb}},
@{1='PhysicalMB';e={$_.workingset/1MB}} -autosize`
3. `Get-EventLog -List | Format-Table
@{1='LogName';e={$_.LogDisplayname}},
@{1='RetDays';e={$_.MinimumRetentionDays}} -autosize`
4. `Get-Service | sort Status -descending | format-table -GroupBy Status`
5. `Dir c:\ -directory | format-wide -column 4`
6. `dir c:\windows*.exe |
Format-list Name,VersionInfo,@{Name="Size";Expression={$_.length}}`

11

Filtrowanie i porównywanie danych

Do tej pory pracowaliśmy z pełnymi wynikami działania otrzymywanymi od powłoki: wszystkimi procesami, wszystkimi usługami, wszystkimi wpisami dziennika zdarzeń, wszystkimi poprawkami i tak dalej. Taki rodzaj wyników jednak nie zawsze będzie tym, czego oczekujesz. Bardzo często będziesz chciał zawęzić wyniki do kilku elementów, które Cię szczególnie interesują. Właśnie tego nauczysz się w tym rozdziale.

11.1. Jak sprawić, aby powłoka dała Ci to, czego potrzebujesz

Powłoka oferuje dwie metody zawężania wyników działania poleceń i obie metody nazywane są **filtrowaniem**. W pierwszej metodzie próbujemy zmusić polecenie do pobierania tylko takich informacji, które określiliśmy. W drugiej metodzie, która przyjmuje podejście iteracyjne, bierzemy wszystko, co daje polecenie pobierające dane, i używamy drugiego polecenia do odfiltrowania niepotrzebnych elementów.

Optymalnym rozwiązaniem jest korzystanie z pierwszej metody, którą nazywamy **wczesnym filtrowaniem** (ang. *early filtering*), tak często, jak to tylko możliwe. Może się to sprowadzać do prostego poinformowania polecenia, o jakie informacje Ci chodzi. Przykładowo, korzystając z polecenia `Get-Service`, możesz określić nazwy usług, o których chcesz wyświetlić informacje:

```
Get-Service -name e*,*s*
```

Ale jeżeli chcesz, aby usługa `Get-Service` zwróciła tylko działające usługi, bez względu na ich nazwy, nie możesz tego zrobić bezpośrednio za pomocą tego polecenia, ponieważ nie oferuje ono żadnych parametrów umożliwiających wykonanie takiej operacji.

Podobnie, jeżeli korzystasz z modułu ActiveDirectory firmy Microsoft, to powinieneś wiedzieć, że wszystkie jego polecenia obsługują parametr `-filter`. Mimo że możesz użyć parametru `-filter *` do wyświetlenia wszystkich obiektów, nie zalecamy takiego rozwiązania z powodu obciążenia kontrolera domeny, jakie w dużych domenach może spowodować próba wykonania takiego polecenia. Zamiast tego powinieneś zatem określić kryteria, które dokładnie precyzują, co chcesz uzyskać, tak jak to zostało pokazane w przykładzie poniżej:

```
Get-ADComputer -filter "Name -like '*DC'"
```

Jest to dobre rozwiązanie, ponieważ wykonywane polecenie musi pobierać tylko obiekty pasujące do podanego wzorca. Takie rozwiązanie nazywamy **filtrowaniem z lewej strony** (ang. *filter left technique*).

11.2. Filtrowanie z lewej

Filtrowanie z lewej strony oznacza umieszczenie kryteriów filtrowania tak daleko w lewo, w kierunku początku wiersza polecenia, jak to tylko możliwe. Im wcześniej możesz odfiltrować niechciane obiekty, tym mniej pracy będą miały do wykonania kolejne polecenia w wierszu poleceń, a do Twojego komputera będzie przesyłanych mniej niepotrzebnych informacji.

Minusem techniki filtrowania z lewej strony jest to, że każde polecenie może implementować własne sposoby określania filtrowania i że każde polecenie posiada różne możliwości filtrowania. W przypadku polecenia `Get-Service` możemy filtrować wyniki działania tylko na podstawie właściwości `Name`. Jednak już w przypadku polecenia `Get-ADComputer` możemy filtrować wyniki według dowolnych atrybutów usługi Active Directory, jakie może mieć obiekt typu `Computer`. Efektywność techniki filtrowania z lewej jest w dużej mierze uzależniona od doświadczenia użytkownika i jego wiedzy na temat sposobu działania poszczególnych poleceń, ale ta technika pozwala znakomicie poprawić wydajność działania polecenia.

Jeśli nie możesz uzyskać żądanych wyników przy użyciu danego polecenia, możesz skorzystać z jednego z podstawowych poleceń powłoki PowerShell, o nazwie `Where-Object` (alias `Where`). Polecenie to wykorzystuje składnię ogólną i można go użyć do filtrowania dowolnego rodzaju obiektów po ich pobraniu i umieszczeniu w potoku.

Aby użyć polecenia `Where-Object`, musisz nauczyć się, jak poinformować powłokę o tym, co chcesz filtrować, a taka operacja wymaga użycia operatorów porównania powłoki. Interesujące jest to, że niektóre techniki filtrowania, takie jak parametr `-filter` poleceń z rodziny `Get-` z modułu Active Directory, używają tych samych operatorów porównania, więc poznanie ich będzie naprawdę przydatne. Warto jednak zauważyć, że niektóre polecenia (myślimy tutaj o `Get-WmiObject`, o którym będziemy pisać nieco później w tym rozdziale) używają zupełnie innego języka filtrowania i porównywania.

11.3. Korzystanie z operatorów porównania

Operacja **porównania** zawsze bierze dwa obiekty lub wartości i sprawdza ich wzajemne relacje. Możesz sprawdzać, czy są równe, czy jedna wartość jest większa od drugiej lub czy któraś z tych wartości pasuje do podanego wzorca. Korzystając z **operatora porównania** (ang. *comparison operator*), wskazujesz rodzaj relacji, którą chcesz przetestować. Wynik porównania jest zawsze wartością typu Boolean: True lub False. Innymi słowy, albo testowany związek jest taki, jak określiłeś, albo nie jest.

Powłoka PowerShell używa wielu operatorów porównania, których zestawienie zamieszczamy poniżej. Zwróć uwagę, że przy porównywaniu ciągów tekstowych nie uwzględniają one wielkości liter — inaczej mówiąc, wielkie i małe litery nie są rozróżniane.

- `-eq` — *równy*; na przykład `5 -eq 5` (prawda; True) lub `"hello" -eq "help"` (fałsz; False);
- `-ne` — *różny*; na przykład `10 -ne 5` (prawda) lub `"help" -ne "help"` (fałsz, ponieważ oba argumenty są identyczne);
- `-ge` i `-le` — *większy lub równy* i *mniej lub równy*; na przykład `10 -ge 5` (prawda) lub `Get-Date -le "2012-12-02"` (wynik tej operacji zależy od tego, kiedy uruchomisz takie polecenie; jest to dobry przykład na to, jak można porównywać daty);
- `-gt` i `-lt` — *większy niż* i *mniej niż*; na przykład `10 -lt 10` (fałsz) lub `100 -gt 10` (prawda).

W przypadku porównywania ciągów znaków możesz również używać osobnego zestawu operatorów uwzględniających wielkość liter (jeżeli jest to konieczne): `-ceq`, `-cne`, `-cgt`, `-clt`, `-cge`, `-cle`.

Jeśli chcesz porównać więcej niż jeden element naraz, możesz używać operatorów logicznych (ang. *boolean operators*) `-and` i `-or`. Są to operatory dwuargumentowe, stąd wymagają umieszczenia odpowiednich podwyrażeń po obu stronach operatora. Złożone wyrażenia z operatorami logicznymi zwykle umieszczamy w nawiasach, tak aby całość wyrażenia była bardziej przejrzysta i łatwiejsza do przeanalizowania:

- `(5 -gt 10) -and (10 -gt 100)` to fałsz, ponieważ co najmniej jedno z podwyrażeń jest fałszywe;
- `(5 -gt 10) -or (10 -lt 100)` to prawda, ponieważ co najmniej jedno z podwyrażeń jest prawdziwe.

Istnieje również operator negacji, `-not`, który zamienia wartości True i False. Taki operator może być przydatny, gdy mamy do czynienia ze zmienną lub właściwością, która ma określoną wartość logiczną i chcemy przetestować ją pod kątem przeciwnej wartości. Na przykład jeżeli chcesz sprawdzić, czy dany proces nie odpowiada, możesz wykonać następującą operację (wyrażenia `$__` użyjemy jako symbolu zastępczego dla wybranego obiektu procesu):

```
$_.Responding -eq $Fałsz
```

Windows PowerShell posiada predefiniowane stałe `$False` i `$True`, reprezentujące wartości logiczne `False` i `True`. Innym sposobem wykonania powyższej operacji może być użycie następującego wyrażenia:

```
-not $_.Responding
```

Ponieważ właściwość `Responding` w normalnych warunkach przyjmuje wartość `True` lub `False`, użycie operatora `-not` odwraca `False` na `True`. Jeżeli proces nie odpowiada (czyli właściwość `Responding` ma wartość `False`), takie porównanie zwróci wartość `True`, wskazując tym samym, że proces „nie odpowiada” (ang. *not responding*). Preferujemy tę drugą technikę. Czasami możesz się również spotkać z innym zapisem, gdzie zamiast operatora `-not` używany jest wykrzyknik (!).

Istnieje również kilka innych operatorów porównania, które są szczególnie użyteczne, gdy zachodzi potrzeba porównywania ciągów tekstu:

- `-like` — pozwala na używanie symbolu `*` zastępującego dowolny ciąg znaków, dzięki czemu możesz wykonywać na przykład takie operacje porównania, jak `"Hello" -like "*ll*"` (prawda). Operatorem przeciwnym jest `-notlike`. Oba operatory nie uwzględniają wielkości liter; jeżeli podczas porównywania musisz brać małe i wielkie litery, powinieneś użyć operatorów `-clike` i `-cnotlike`.
- `-match` — dokonuje porównania ciągów tekstu ze wzorcem w postaci wyrażenia regularnego. Logicznym przeciwieństwem jest operator `-notmatch`, a jak można się było spodziewać, oba operatory są dopełnione operatorami `-cmatch` i `-cnotmatch`, zapewniającymi rozróżnianie wielkości liter. Niestety zagadnienia związane z tworzeniem i używaniem wyrażeń regularnych wykraczają już daleko poza zakres tej książki.

Ciekawą i użyteczną funkcją powłoki jest to, że prawie wszystkie testy można uruchomić bezpośrednio z poziomu wiersza poleceń (wyjątkiem jest ten, w którym używamy symbolu `$_` — takie polecenie nie będzie działać samodzielnie, ale w kolejnym podrozdziale zobaczysz, jak należy go używać).

ZRÓB TO SAM Spróbuj poeksperymentować i samodzielnie wypróbować wybrane lub nawet wszystkie omówione wyżej operatory porównania. Możesz je wpisywać bezpośrednio w wierszu polecenia, na przykład wpisz wyrażenie `5 -eq 5`, naciśnij klawisz *Enter* i zobacz, jaki rezultat otrzymałeś.

Informacje o innych operatorach porównania możesz znaleźć w pliku pomocy `about_>comparison operators`, a jeszcze kilka innych operatorów poznasz w rozdziale 25.

Dla zainteresowanych

Jeżeli jakieś polecenie nie używa operatorów porównania powłoki PowerShell, omawianych w podrozdziale 11.3, prawdopodobnie będzie korzystało z bardziej tradycyjnych operatorów porównania używanych w wielu językach programowania (operatory te możesz znać jeszcze ze szkoły średniej bądź nawet z codziennej pracy):

- = równy,
- <> różny,
- <= mniejszy lub równy,
- >= większy lub równy,
- > większy niż,
- < mniejszy niż.

Jeżeli polecenie obsługuje operatory logiczne, zwykle są to operatory AND i OR; niektóre polecenia mogą również obsługiwać operatory takie jak LIKE. Przykładowo, wszystkie wymienione operatory są obsługiwane przez parametr `-filter` polecenia `Get-WmiObject`; pełną listę obsługiwanych operatorów przypomnimy podczas omawiania tego polecenia w rozdziale 14.

Projektanci poszczególnych poleceń muszą samodzielnie określić, czy i w jaki sposób dane polecenie będzie obsługiwało filtrowanie; więcej szczegółowych informacji na temat działania poszczególnych poleceń oraz przykłady ich wywołania możesz znaleźć w odpowiednich plikach pomocy.

11.4. Filtrowanie obiektów poza potokiem

Po odpowiednim zapisaniu wyrażenia porównania gdzie możesz go używać? Jednym z najczęstszych zastosowań jest parametr `-filter` niektórych poleceń, a zwłaszcza z cmdletami rodziny `Get-` modułu `ActiveDirectory`. Możesz go także używać w poleceniu `Where-Object` odgrywającym rolę ogólnego polecenia filtrującego powłoki PowerShell.

Na przykład aby wyświetlić listę działających usług, powinieneś z wyników działania polecenia `Get-Service` odfiltrować wszystkie usługi, które nie zostały uruchomione:

```
Get-Service | Where-Object -filter { $_.Status -eq 'Running' }
```

Parametr `-filter` jest pozycjonowany, co oznacza, że często spotykamy polecenia wpisywane bez podawania jego nazwy oraz z aliasem `Where` zamiast pełnej nazwy polecenia:

```
Get-Service | Where { $_.Status -eq 'Running' }
```

Jeżeli głośno przeczytasz takie polecenie (w języku angielskim), to zabrzmi ono całkiem sensownie: *gdzie status usługi jest równy 'uruchomiona'*. Oto jak to działa: po przekazaniu obiektów za pomocą potoku do polecenia `Where-Object` każdy z nich jest sprawdzany przy użyciu filtra. Polecenie umieszcza po jednym obiekcie w miejscu symbolu zastępczego `$_`, a następnie sprawdza, czy powstałe wyrażenie jest prawdziwe. Jeżeli wynikiem porównania jest `False`, obiekt jest usuwany z potoku. Jeżeli wynik porównania ma wartość `True`, obiekt jest wyprowadzany z obiektu `Where` i przekazywany do następnego polecenia w potoku. W tym przypadku następnym poleceniem jest cmdlet `Out-Default`, który zawsze znajduje się na końcu potoku (jak to opisywaliśmy w rozdziale 8.) i który rozpoczyna proces formatowania danych wyjściowych w celu ich wyświetlenia na ekranie.

Symbol zastępczy `$_` jest specjalnym elementem: spotkałeś go już wcześniej w rozdziale 10., a zobaczysz go jeszcze w jednym lub dwóch innych zastosowaniach. Tego

symbolu zastępczego można używać tylko w określonych miejscach, w których powłoka PowerShell go oczekuje, a nasz przykład ilustruje jedno z takich miejsc. Jak dowiedziałeś się w rozdziale 10., znak kropki „mówi” powłoce, że nie chcesz porównywać całego obiektu, a tylko jedną z jego właściwości, Status.

Mamy nadzieję, że zaczynasz już się orientować, gdzie przydaje się polecenie Gm. Jego wyniki działania są szybkim i łatwym sposobem na odkrycie właściwości obiektu, dzięki czemu możesz użyć ich w operacjach porównania, takich jak operacja pokazana w naszym przykładzie. Powinieneś zawsze pamiętać, że nagłówki kolumn wyświetlane w wynikach działania poleceń powłoki PowerShell nie zawsze odzwierciedlają rzeczywiste nazwy poszczególnych właściwości obiektów. Na przykład po uruchomieniu polecenia Get-Process w jego wynikach działania znajdziesz kolumnę o nazwie PM(MB). Następnie uruchom polecenie Get-Process | Gm, a przekonasz się, że w rzeczywistości ta właściwość nosi nazwę PM. To bardzo ważna uwaga: zawsze powinieneś sprawdzać nazwy właściwości za pomocą polecenia Gm, a nie za pomocą poleceń z rodziny Format-.

Dla zainteresowanych

W powłoce PowerShell v3 została wprowadzona nowa, „uproszczona” składnia polecenia Where-Object. Możesz jej używać tylko wtedy, gdy robisz jedno porównanie; jeżeli chcesz porównywać wiele elementów, musisz nadal używać oryginalnej składni, którą omawialiśmy w tym podrozdziale.

Użytkownicy często zastanawiają się, czy ta uproszczona składnia jest do czegoś przydatna. Wygląda to mniej więcej tak:

```
Get-Service | Where Status -eq "Running"
```

Oczywiście taka forma zapisu jest nieco bardziej przejrzysta i bardziej czytelna: rezygnujemy tutaj z nawiasów klamrowych {} i nie musimy używać nieco dziwnego symbolu zastępczego \$_. Ale pojawienie się nowej składni nie oznacza wcale, że możesz zapomnieć o starej składni, wciąż bowiem musisz jej używać do przeprowadzania bardziej złożonych porównań:

```
get-service | where-object {$_.status -eq "running" -AND $_.StartType -eq "Manual"}
```

Co więcej, w sieci Internet możesz znaleźć całe mnóstwo przykładów wykorzystujących starą składnię, a zatem musisz ją znać, aby z nich korzystać. Nie zmienia to jednak faktu, że musisz także znać nową składnię tego polecenia, ponieważ coraz częściej zaczyna się ona pojawiać w nowych przykładach. Konieczność poznania dwóch zestawów składni nie musi bynajmniej wydawać się „uproszczeniem”, ale przynajmniej już wiesz, o co tutaj chodzi.

11.5. Używanie iteracyjnego modelu wiersza poleceń

Chcielibyśmy teraz krótko omówić coś, co nazywamy iteracyjnym modelem wiersza poleceń powłoki PowerShell (PSICLM — ang. *PowerShell Iterative Command-Line Model*; w zasadzie nie ma powodu, aby tworzyć taki akronim, ale brzmi on całkiem fajnie). Idea modelu PSICLM polega na tym, że nie musisz budować dużych, złożonych wierszy poleceń i tworzyć ich całkowicie od zera. Zamiast tego możesz zacząć od czegoś mniejszego.

Powiedzmy, że chcesz sprawdzić, ile pamięci wirtualnej jest używane przez 10 najbardziej „pamięciożernych” procesów. Ponieważ jednak sama powłoka PowerShell jest jednym z takich procesów, nie chcesz, aby była ona uwzględniona w obliczeniach. Przygotujmy zatem szybką listę tego, co musisz zrobić:

1. Pobierz listę procesów.
2. Pozbądź się wszystkiego, co jest związane z powłoką PowerShell.
3. Posortuj listę procesów według zużycia pamięci wirtualnej.
4. Zachowaj tylko pierwszych 10 procesów o największym lub najmniejszym zużyciu pamięci wirtualnej (w zależności od tego, w jakiej kolejności je sortujesz).
5. Zsumuj ilości pamięci wirtualnej zużywanej przez poszczególne procesy.

Wierzmy, że wiesz już, jak wykonać pierwsze trzy kroki. Czwarty możemy zrealizować za pomocą naszego starego dobrego polecenia `Select-Object`.

ZRÓB TO SAM Zapoznaj się z zawartością pliku pomocy dla polecenia `Select-Object`. Czy potrafisz znaleźć parametry, które umożliwią zachowanie określonej liczby obiektów, licząc od początku lub do końca kolekcji?

Mamy nadzieję, że znalazłeś odpowiedź.

Na koniec musimy zsumować rozmiary obszarów pamięci wirtualnej zużywanej przez poszczególne procesy. Będzie nam do tego potrzebne nowe polecenie, które możemy znaleźć, korzystając z cmdletu `Get-Command` i symboli wieloznacznych lub systemu pomocy. We wzorcu wyszukiwania możesz użyć słów kluczowych `Add`, `Sum` lub nawet słowa kluczowego `Measure`.

ZRÓB TO SAM Zobacz, czy uda Ci się znaleźć polecenie, które zwraca całkowitą wartość określonej właściwości numerycznej, takiej jak zużycie pamięci wirtualnej. Użyj polecenia `Help` lub `Get-Command` z symbolem wieloznacznym `*`.

Jeżeli samodzielnie wykonujesz poszczególne małe zadania składowe (i nie zaglądasz z wyprzedzeniem na koniec, aby znaleźć gotową odpowiedź), nabierasz niezbędnego doświadczenia, które pozwoli Ci zostać prawdziwym ekspertem w pracy z powłoką PowerShell. Kiedy dojdiesz do wniosku, że masz już gotowe rozwiązanie, możesz spróbować iteracyjnego podejścia do wykonania takiego zadania.

Aby rozpocząć, musimy uzyskać listę działających procesów. To oczywiście jest bardzo proste:

```
Get-Process
```

ZRÓB TO SAM Spróbuj samodzielnie wykonywać kolejne przedstawiane tutaj polecenia. Uważnie przyglądaj się wynikom działania każdego z nich i zastanów się, co powinieneś zmienić dla następnej iteracji polecenia.

W dalszej kolejności musimy odfiltrować to, co nie jest nam potrzebne. Pamiętaj, że *filtrowanie z lewej strony* oznacza, że chcemy, aby filtr był jak najbliżej początku wiersza poleceń. W naszym przypadku do filtrowania użyjemy polecenia `Where-Object`, które będzie kolejnym cmdletem w naszym wierszu poleceń.

Nie jest to co prawda tak dobre rozwiązanie jak filtrowanie już w pierwszym poleceniu, ale zawsze jest lepsze niż filtrowanie w dalszej części potoku.

Będąc w konsoli powłoki, naciśnij klawisz ze strzałką w górę, tak aby przywołać ostatnio wykonane polecenie, po czym zmodyfikuj je w następujący sposób:

```
Get-Process | Where-Object -filter {$_.Name -notlike 'powershell*'}
```

Nie jesteśmy pewni, czy proces powłoki to powershell, czy powershell.exe, więc używamy symbolu wieloznacznego, tak aby porównanie objęło wszystkie możliwości. Po wykonaniu tego polecenia w potoku pozostaną tylko takie obiekty procesów, które nie spełniają warunku filtrowania.

Uruchom przygotowane polecenie, aby je przetestować, a następnie naciśnij ponownie klawisz ze strzałką w górę, aby do naszego wiersza polecenia dodać następny element:

```
Get-Process | Where-Object -filter {$_.Name -notlike 'powershell*' } |  
Sort VM -descending
```

Naciśnij klawisz *Enter*, aby sprawdzić, czy polecenie działa poprawnie, a następnie kolejny raz naciśnij klawisz ze strzałką w górę, aby dodać kolejny element układanki:

```
Get-Process | Where-Object -filter {$_.Name -notlike 'powershell*' } |  
Sort VM -descending | Select -first 10
```

Jeżeli posortowalibyśmy listę w domyślnej kolejności rosnącej, należałoby użyć parametru `-last 10`. Ostatni element wiersza polecenia ma taką postać:

```
Get-Process | Where-Object -filter {$_.Name -notlike 'powershell*' } |  
Sort VM -descending | Select -first 10 |  
Measure-Object -property VM -sum
```

Mamy nadzieję, że udało Ci się wcześniej znaleźć przynajmniej nazwę tego ostatniego cmdletu (`Measure-Object`), a może nawet użył ją w naszym przykładzie składni.

Taki model działania, polegający na uruchamianiu polecenia, badaniu jego wyników działania, a następnie przywoływaniu wykonanego polecenia i odpowiednim modyfikowaniu go w celu wykonania kolejnego kroku operacji, odróżnia powłokę PowerShell od bardziej tradycyjnych języków skryptowych. Ponieważ PowerShell jest powłoką wiersza polecenia, wyniki działania poleceń otrzymujesz natychmiast, dzięki czemu masz możliwość szybkiego i łatwego modyfikowania polecenia w sytuacji, kiedy wyniki nie są zgodne z Twoimi oczekiwaniami. Szybko przekonasz się, jak ogromne możliwości daje łączenie ze sobą nawet kilku prostych cmdletów, które poznałeś do tej pory.

11.6. Najczęściej spotykane problemy

Za każdym razem, kiedy podczas naszych szkoleń przechodzimy do omawiania polecenia `Where-Object`, nasi studenci borykają się z dwoma głównymi zagadnieniami. Staraliśmy się je dokładnie omówić w dyskusji, ale jeśli masz jakiegokolwiek wątpliwości, spróbujemy je teraz rozwiązać.

11.6.1. Filtr z lewej, proszę

Zawsze powinieneś się starać, aby kryteria filtrowania znajdowały się *tak blisko początku wiersza poleceń, jak to możliwe*. Jeżeli możesz wykonać niezbędne filtrowanie w pierwszym cmdlecie, zrób to; jeżeli nie, spróbuj filtrować w drugim cmdlecie, tak aby kolejne polecenia miały jak najmniej pracy do wykonania.

Spróbuj również przeprowadzać filtrowanie tak blisko źródła danych, jak to tylko możliwe. Przykładowo, jeżeli sprawdzasz, jakie usługi działają na komputerze zdalnym i chcesz użyć polecenia `Where-Object`, tak jak to robiliśmy w jednym z przykładów tego rozdziału, rozważ możliwość zdalnego wykonywania poleceń powłoki PowerShell, żeby filtrowanie odbywało się na komputerze zdalnym, dzięki czemu nie będziesz musiał przysyłać wszystkich obiektów przez sieć i filtrować ich dopiero na swoim komputerze. W rozdziale 13. będziemy zajmować się zagadnieniami związanymi z dostępem zdalnym i powrócimy tam do idei filtrowania danych u źródła.

11.6.2. Kiedy używanie symbolu `$_` jest dozwolone?

Specjalny symbol zastępczy `$_` może być używany tylko w takich miejscach, w których powłoka PowerShell go oczekuje. Jeżeli lokalizacja jest poprawna, w jego miejscu pojawia się jeden obiekt jednocześnie spośród tych, które zostały przekazane za pomocą potoku. Pamiętaj, że zawartość potoku może i z pewnością będzie się zmieniać, w miarę jak kolejne polecenia będą wykonywane i będą przekazywać do potoku wyniki swojego działania.

Powinieneś również uważać na zagnieżdżone potoki, czyli takie, które występują w nawiasach. Na przykład na pierwszy rzut oka wyniki działania poniższego polecenia mogą być trudne do przewidzenia:

```
Get-Service -computername (Get-Content c:\names.txt |  
Where-Object -filter { $_ -notlike '*dc' }) |  
Where-Object -filter { $_.Status -eq 'Running' }
```

Spróbujmy zatem przyrzeć się temu bliżej:

1. Wiersz polecenia rozpoczyna się od polecenia `Get-Service`, ale to nie jest pierwsze polecenie, które zostanie wykonane. Ze względu na umieszczenie w nawiasach jako pierwsze zostanie wykonane polecenie `Get-Content`.
2. Polecenie `Get-Content` przekazuje wyniki swojego działania, składające się z prostych obiektów typu `String`, na wejście polecenia `Where-Object`. Polecenie `Where-Object` również znajduje się wewnątrz nawiasów, a symbol `$_` umieszczony w jego filtrze reprezentuje obiekty `String` przesyłane z polecenia `Get-Content`. Wynikiem działania polecenia `Where-Object` są tylko takie ciągi znaków, które nie kończą się znakami `dc`.
3. Wynik działania polecenia `Where-Object` staje się wynikiem działania całego polecenia w nawiasie, ponieważ `Where-Object` jest ostatnim cmdletem w nawiasach, zatem wszystkie nazwy komputerów, które nie kończą się ciągiem znaków `dc`, będą przekazywane do parametru `-computername` polecenia `Get-Service`.

4. Dopiero *teraz* wykonywane jest polecenie `Get-Service`, a obiekty typu `Service` ➔ `Controller`, które będą wynikiem jego działania, zostaną przekazane do drugiego polecenia `Where-Object`, gdzie będą kolejno umieszczane w miejscu symbolu `$_`. Wynikiem działania będzie lista usług uruchomionych na komputerach zdalnych, których właściwość `Status` ma wartość `Running`.

Czasami mamy wrażenie, że te wszystkie nawiasy klamrowe, kropki i nawiasy okrągłe powodują, że całe polecenie jest trochę niezrozumiałe, ale w ten sposób właśnie działa powłoka PowerShell. Jeżeli będziesz w stanie przeanalizować przebieg procesu wykonania polecenia, będziesz w stanie zrozumieć, co ono robi i jakie mogą być wyniki jego działania.

11.7. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie komputer działający pod kontrolą systemu Windows 8, Windows Server 2012 lub nowszego oraz z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Pamiętaj, że polecenie `Where-Object` nie jest jedynym sposobem filtrowania, co więcej, nie jest to nawet sposób, z którego w pierwszej kolejności powinieneś korzystać. Starał się, aby ten rozdział był stosunkowo krótki, tak aby dać Ci więcej czasu na pracę z przykładami praktycznymi. Zatem pamiętając o zasadzie *filtrowania z lewej strony*, spróbuj samodzielnie wykonać następujące zadania:

1. Zaimportuj moduł `NetAdapter` (dostępny w najnowszych wersjach systemu Windows zarówno dla komputerów, jak i serwerów). Za pomocą polecenia `Get-NetAdapter` wyświetl listę kart sieciowych, których właściwość `Virtual` ma wartość `False` (powłoka PowerShell reprezentuje ją za pomocą specjalnej stałej `$False`).
2. Zaimportuj moduł `DnsClient` (dostępny w najnowszych wersjach systemu Windows zarówno dla komputerów, jak i serwerów). Za pomocą polecenia `Get-DnsClientCache` wyświetl listę rekordów `A` i `AAAA` z pamięci podręcznej. Wskazówka: jeżeli pamięć podręczna jest pusta, spróbuj odwiedzić kilka stron internetowych, aby wymusić zapisanie niektórych elementów w pamięci podręcznej.
3. Wyświetl wszystkie pliki typu `EXE` znajdujące się w folderze `C:\Windows\System32`, które mają rozmiar większy niż 5 MB.
4. Wyświetl listę wszystkich poprawek, które są aktualizacjami zabezpieczeń.
5. Wyświetl listę poprawek, które zostały zainstalowane przez użytkownika `Administrator` i które są aktualizacjami. Jeżeli nie znajdziesz żadnych, spróbuj wyszukać poprawki zainstalowane w kontekście użytkownika `SYSTEM`. Zauważ, że w przypadku niektórych poprawek właściwość `InstalledBy` nie ma przypisanej żadnej wartości — jest to najzupełniej normalne.
6. Wyświetl listę wszystkich działających procesów o nazwach `conhost` lub `svchost`.

11.8. Co dalej?

Praktyka czyni mistrza, zatem spróbuj samodzielnie na różne sposoby filtrować wybrane wyniki działania znanych Ci poleceń, takich jak `Get-Hotfix`, `Get-EventLog`, `Get-Process`, `Get-Service`, a nawet `Get-Command`. Możesz chociażby spróbować filtrować wyniki działania polecenia `Get-Command`, tak aby wyświetlić tylko cmdlety. Możesz również użyć polecenia `Test-Connection` do pingowania szeregu komputerów i wyświetlać wyniki tylko dla komputerów, które nie odpowiedziały. Nie sugerujemy, że w każdym przypadku musisz używać polecenia `Where-Object`, aczkolwiek powinieneś nabierać praktyki w używaniu go, gdy jest to właściwe.

11.9. Odpowiedzi

1. `import-module NetAdapter`
`get-netadapter -physical`
2. `Import-Module DnsClient`
`Get-DnsClientCache -type AAAA.A`
3. `Dir c:\windows\system32*.exe | where {$_.length -gt 5MB}`
4. `Get-Hotfix -Description 'Security Update'`
5. `get-hotfix -Description Update | where {$_.InstalledBy -match "administrator"}`
lub dowolne z poniższych:
`get-hotfix -Description Update | where {$_.InstalledBy -match "system"}`
`get-hotfix -Description Update | where {$_.InstalledBy -eq "NT Authority\↵System"}`
6. `get-process -name svchost,conhost`

12

Trochę praktyki

Najwyższy czas zacząć wykorzystywać nowo nabytą wiedzę w praktyce. W tym rozdziale nie będziemy nawet próbować uczyć Cię czegoś nowego. Zamiast tego spróbujemy szczegółowo omówić przykładowe zadanie, które możesz wykonać, korzystając z tego, czego się do tej pory nauczyłeś. Będzie to przykład wzięty z codziennego życia administratora systemu, gdzie postawimy sobie określone zadanie, a następnie pokażemy, jak można je wykonać w praktyce. Ten rozdział jest dobrym przykładem idei, jaka przyświecała nam podczas pisania tej książki, ponieważ zamiast po prostu przedstawiać Ci gotowe rozwiązania, wskazujemy Ci właściwą drogę postępowania, dzięki czemu *możesz je wykonać samodzielnie*.

12.1. Definiowanie zadania

Przyjmujemy założenie, że używasz komputera działającego pod kontrolą systemu Windows 10, Windows Server 2012 R2 lub nowszego i masz zainstalowaną powłokę PowerShell v5 lub nowszą. Przykład, który będziemy omawiać, może dobrze działać z wcześniejszymi wersjami systemu Windows i powłoki PowerShell, ale wszystkie testy przeprowadzaliśmy w systemie Windows 10 z powłoką PowerShell v5. Wiemy, że to nie będzie działać w systemie Linux lub macOS, ponieważ te systemy operacyjne używają innego modelu uprawnień użytkownika niż system Windows.

Naszym zadaniem jest użycie powłoki PowerShell do modyfikacji niektórych domyślnych uprawnień użytkownika w systemie lokalnym. Tak naprawdę to nie są uprawnienia, ale raczej ogólne zadania systemowe, które może wykonywać użytkownik lub grupa.

12.2. Wyszukiwanie odpowiednich poleceń

Pierwszym krokiem do rozwiązania dowolnego zadania jest ustalenie listy poleceń, za pomocą których można takie zadanie wykonać. W zależności od używanego środowiska wyniki Twoich poszukiwań mogą się nieco różnić od naszych, ale ważny jest przebieg samego procesu wyszukiwania poleceń. Ponieważ wiemy, że chcemy modyfikować uprawnienia użytkownika, poszukiwania zaczniemy od słowa kluczowego `privilege`:

```
PS C:\> help *privilege*
Name                                Category    Module
-----
Update-Help                        Cmdlet      Microsoft.PowerShell.Core
ConvertTo-Csv                      Cmdlet      Microsoft.PowerShell.U...
Import-Counter                     Cmdlet      Microsoft.PowerShell.D...
about_Remote_Requirements          HelpFile
about_Remote_Troubleshooting       HelpFile
```

Hm. Zbyt wiele nam to nie pomogło. Nic tutaj nie wygląda tak, jakby miało coś wspólnego z uprawnieniami. No dobrze, spróbujmy innego podejścia — tym razem skoncentrujemy się na poleceniach, a nie na plikach pomocy, i spróbujemy użyć nieco mniej dokładnego wzorca wyszukiwania:

```
PS C:\> get-command -noun *priv*
PS C:\>
```

OK, zatem nie ma żadnych poleceń, których nazwy zawierają ciąg znaków *priv*. Co za rozczarowanie! Teraz musimy zobaczyć, co można znaleźć na stronie internetowej PowerShell Gallery. Zauważ, że ten krok opiera się na założeniu, że masz zainstalowanego menedżera pakietów PowerShell Package Manager. Jest to część powłoki PowerShell v5, ale możesz też zainstalować ją w starszych wersjach powłoki. Na stronie <http://powershellgallery.com> znajdziesz łącza do pobierania odpowiednich plików.

```
PS C:\> find-module *privilege* | format-table -auto
Version Name      Repository      Description
-----
0.3.0.0 PoshPrivilege  PSGallery      Module designed to use allow easi...
```

O, to wygląda znacznie bardziej zachęcająco! Spróbujmy zatem zainstalować ten moduł:

```
PS C:\> install-module poshprivilege
You are installing the module(s) from an untrusted repository. If you trust this repository,
↳ change its InstallationPolicy value by running the Set-PSRepository cmdlet.
Are you sure you want to install software from
'https://go.microsoft.com/fwlink/?LinkID=397631&clcid=0x409'?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help(default is "N"): y
```

Teraz musimy zachować ostrożność. Chociaż to firma Microsoft udostępnia i prowadzi stronę PowerShell Gallery, nie sprawdza poprawności kodu modułów publikowanych przez strony trzecie. Zrobiliśmy więc przerwę w realizacji zadania, poświęciliśmy trochę czasu na dokładne sprawdzenie kodu nowo zainstalowanego modułu i upewniliśmy się, że jest bezpieczny. W tym przedsięwzięciu pomogło nam także to, że autorem tego modułu jest inny MVP, Boe Prox, którego znamy i któremu ufamy.

Teraz zobaczymy polecenia, które właśnie otrzymaliśmy po zainstalowaniu modułu poshprivilege:

```
PS C:\> get-command -module PoshPrivilege | format-table -auto
CommandType      Name                Version            Source
-----
Function          Add-Privilege       0.3.0.0            PoshPrivilege
Function          Disable-Privilege   0.3.0.0            PoshPrivilege
Function          Enable-Privilege    0.3.0.0            PoshPrivilege
Function          Get-Privilege       0.3.0.0            PoshPrivilege
Function          Remove-Privilege    0.3.0.0            PoshPrivilege
```

No dobrze, poszczególne polecenia wydają się dosyć proste i oczywiste. W naszym przypadku będziemy chcieli dodawać bądź włączać uprawnienia, zatem pozostaje nam odpowiedzieć na pytanie, jakie uprawnienia będą nam potrzebne.

12.3. Uczymy się, jak korzystać z nowych poleceń

Na szczęście Boe zatroszczył się o przygotowanie odpowiednich plików pomocy do swojego modułu. Programiści, którzy piszą moduły, ale nie dołączają plików pomocy, są naprawdę paskudami. W książce *Learn PowerShell Toolmaking in a Month of Lunches* (wyd. Manning, 2012) omawiamy sposoby tworzenia modułów i dołączania do nich plików pomocy — a tworzenie plików pomocy jest ze wszech miar zalecane i słuszne. Zobaczmy zatem, czy Boe postąpił, jak nakazują dobre obyczaje programistów:

```
PS C:\> help Add-Privilege
NAME
    Add-Privilege
SYNOPSIS
    Adds a specified privilege for a user or group
SYNTAX
    Add-Privilege [[-AccountName] <String>] [-Privilege]
    <Privileges[]> [-WhatIf] [-Confirm] [<CommonParameters>]
```

Jak widać, Boe zrobił co trzeba! Oczywiście nie zamieszczamy tutaj całej treści pliku pomocy, ale zdecydowanie przeczytaliśmy to wszystko. Wygląda na to, że możemy podać nazwę użytkownika lub grupy jako wartość parametru `-AccountName`, a następnie musimy podać nazwę lub nazwy żądanych uprawnień. Chodzi jednak o to, że nie wiemy, jakie one są. W module dostępne jest jednak polecenie `Get`, zatem spróbujemy go użyć:

```
PS C:\> Get-Privilege
ComputersName      Privilege
-----
DESKTOP-GDI89IG    SeAssignPrimaryTokenPrivilege    {NT AUTHORITY\SYSTEM}
DESKTOP-GDI89IG    SeAuditPrivilege                  {NT AUTHORITY\SYSTEM}
DESKTOP-GDI89IG    SeBackupPrivilege                 {BUILTIN\Bac...}
DESKTOP-GDI89IG    SeBatchLogonRight                 {BUILTIN\Per...}
DESKTOP-GDI89IG    SeChangeNotifyPrivilege          {BUILTIN\Bac...}
DESKTOP-GDI89IG    SeCreateGlobalPrivilege          {NT AUTHORITY\SYSTEM}
DESKTOP-GDI89IG    SeCreatePagefilePrivilege        {BUILTIN\Adm...}
```

Pełne wyniki działania zajmują kilka ekranów, ale w zamian otrzymujemy listę wszystkich dostępnych uprawnień oraz — zupełnie nieprzypadkowo — listę wszystkich użytkowników, którzy posiadają nadane uprawnienia. No dobrze. Spróbujmy zatem wykonać następujące polecenie:

```
PS C:\> Add-Privilege -AccountName Administrators -Privilege SeDenyBatchLogonRight
```

Świetnie, to było proste. Sprawdźmy teraz, czy przyniosło oczekiwany efekt:

```
PS C:\> Get-Privilege -Privilege SeDenyBatchLogonRight
Computersname      Privilege
-----
DESKTOP-GDI89IG    SeDenyBatchLogonRight
Accounts
-----
{BUILTIN\Adm...
```

Działa!

W tym miejscu musimy jednak przyznać, że to nie jest zbyt skomplikowane zadanie. Ale samo zadanie nie było tutaj najważniejsze. Chodziło nam bardziej o to, *w jaki sposób doszliśmy do tego, jak je wykonać*. Podsumujmy zatem, jak to zrobiliśmy.

1. Zaczęliśmy od przeszukiwania lokalnych plików pomocy pod kątem występowania określonego słowa kluczowego. Kiedy okazało się, że podane słowo kluczowe nie pasuje do nazwy żadnego polecenia, powłoka PowerShell rozpoczęła przeszukiwanie pełnej zawartości wszystkich plików pomocy. Jest to bardzo przydatne, ponieważ pozwala na znalezienie poszukiwanych słów kluczowych również w opisach poleceń.
2. Następnie przeszliśmy do poszukiwania konkretnych nazw poleceń. Dzięki takiemu postępowaniu możemy znaleźć polecenia, *dla których nie zainstalowano żadnych plików pomocy*. Teoretycznie wszystkie polecenia powinny mieć zainstalowane pliki pomocy, ale nie żyjemy w idealnym świecie, więc zawsze robimy ten dodatkowy krok.
3. Nie znaleźliśmy żadnego pasującego polecenia, zatem przeszukaliśmy zasoby strony PowerShell Gallery i znaleźliśmy moduł, który wyglądał bardzo obiecująco. Następnie zainstalowaliśmy ten moduł i sprawdziliśmy, jakie udostępnia polecenia.
4. Ponieważ autor nowo zainstalowanego modułu był dobrym człowiekiem i przygotował do niego odpowiednie pliki pomocy, mogliśmy sprawdzić, jakiego polecenia należy użyć do wyświetlenia listy dostępnych uprawnień. Dzięki systemowi pomocy mogliśmy również zobaczyć, jak uporządkowane są poszczególne polecenia i jakich parametrów wywołania oczekują. Takie poszukiwania często zaczynamy od polecenia `Get`, które (jeśli oczywiście jest dostępne) pozwala zobaczyć, jak wygląda sytuacja.
5. Wykorzystując zebrane do tej chwili informacje, byliśmy już w stanie wprowadzić żądane zmiany i tym samym wykonać zadanie.

12.4. Wskazówki dotyczące samodzielnej nauki

Jak już wspominaliśmy, prawdziwym celem tej książki jest nauczyć Cię, że możesz uczyć się samodzielnie — i takie właśnie było założenie tego rozdziału. Oto kilka dodatkowych wskazówek:

- Nie bój się plików pomocy i zawsze zapoznawaj się z ich treścią oraz zamieszczonymi w nich przykładami. Powtarzamy to uparcie w kółko, a i tak nikt nam nie wierzy. Na naszych kursach ciągle widzimy studentów, którzy na naszych oczach poszukują opisów poleceń czy przykładów ich zastosowania w wyszukiwarkach sieciowych takich jak Google. Co jest tak przerażającego w plikach pomocy? Jeżeli masz odwagę przeglądać posty na czyimś blogu, dlaczego nie chcesz najpierw zajrzeć do przykładów zamieszczonych w plikach pomocy?
- Zwracaj uwagę na szczegóły. Każda informacja wyświetlana na ekranie jest potencjalnie ważna — spróbuj machinalnie nie pomijać rzeczy, które nie są bezpośrednio powiązane z tym, czego szukasz. To nie takie proste, ale spróbuj tak nie robić. Zamiast tego zwracaj uwagę na wszystko, co pojawia się na ekranie, i spróbuj dowiedzieć się, do czego to służy i jakie informacje możesz z tego uzyskać.
- Nie bój się, że coś Ci się nie uda. Mamy nadzieję, że dysponujesz odpowiednią maszyną wirtualną, której możesz używać do testowania — więc korzystaj z niej. Studenci stale zadają nam pytania typu „Hej, jeśli zrobię coś takiego, to co się stanie?”. Często na takie pytania odpowiadamy po prostu: „Nie mam pojęcia, spróbuj”. Eksperymentowanie jest dobre. W końcu jeżeli coś pójdzie mocno nie tak, to w maszynie wirtualnej zawsze możesz się cofnąć do ostatniej migawki, prawda? Zatem nie przejmuj się i idź śmiało do przodu niezależnie od tego, nad czym pracujesz.
- Jeżeli coś nie działa, nie wal głową w ścianę — spróbuj czegoś innego.

Po pewnym czasie i przy odpowiedniej dozie cierpliwości popartej praktyką wszystko staje się łatwe i oczywiste przy założeniu, że zawsze *myślisz* nad tym, co robisz.

12.5. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie komputer działający pod kontrolą systemu Windows 8, Windows Server 2012 lub nowszego oraz z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Teraz Twoja kolej. Zakładamy, że pracujesz na maszynie wirtualnej lub innym komputerze, który możesz trochę zepsuć w imię nauki. Nie rób tego w środowisku produkcyjnym na żadnym komputerze o krytycznym znaczeniu dla jego funkcjonowania!

W systemach Windows 8 i Windows Server 2012 (oraz nowszych) znajduje się moduł do pracy z udziałami sieciowymi. Twoim zadaniem jest utworzenie na swoim komputerze katalogu o nazwie *LABS* i udostępnienie go poprzez sieć innym użytkownikom. W tym ćwiczeniu można założyć, że zarówno sam folder, jak i udział sieciowy

jeszcze nie istnieją. Nie martw się o uprawnienia NTFS, ale upewnij się, że uprawnienia na poziomie udziału są ustawione tak, że każdy użytkownik ma dostęp do odczytu i zapisu plików, a lokalni administratorzy mają nad nim pełną kontrolę. Ponieważ w udostępnianym katalogu będą się znajdować pliki, możesz ustawić tryb buforowania udziału na Documents. Twój skrypt powinien wyświetlić obiekt reprezentujący nowy udział i jego uprawnienia.

12.6. Odpowiedzi

#Tworzenie folderu

```
New-item -Path C:\Labs -Type Directory | Out-Null
```

#Tworzenie udziału sieciowego

```
$myShare = New-SmbShare -Name Labs -Path C:\Labs \  
-Description "MoL Lab Share" -ChangeAccess Everyone \  
-FullAccess Administrators -CachingMode Documents
```

#Pobieranie i wyświetlanie uprawnień udziału sieciowego

```
$myShare | Get-SmbShareAccess
```

Komunikacja zdalna — jeden-do-jednego i jeden-do-wielu

Kiedy po raz pierwszy, dawno temu, zaczynaliśmy przygodę z powłoką PowerShell (w wersji 1), podczas pracy z poleceniem `Get-Service` zauważyliśmy, że ma ono parametr `-computerName`. Hm... zaczęliśmy się zastanawiać, czy to oznacza, że możemy również uzyskać listę usług działających na innych komputerach. Po kilku eksperymentach okazało się, że działa to dokładnie tak, jak nam się wydawało. Byliśmy tym bardzo podniekcytowani i zaczęliśmy szukać parametru `-computerName` w innych poleceniach. Spotkało nas jednak spore rozczarowanie, kiedy się okazało, że tylko kilka innych poleceń posiada taki parametr. W wersji v2 pojawiło się co prawda kilka kolejnych, ale nadal liczba poleceń, które mają ten parametr, jest znacznie mniejsza niż poleceń, które go nie mają.

Po pewnym czasie zdaliśmy sobie sprawę z tego, że twórcy powłoki PowerShell są nieco leniwi — i że wyszło to wszystkim na dobre. Ponieważ nie chcieli dodawać parametru `-computerName` dla każdego cmdletu, stworzyli dla całej powłoki specjalny mechanizm o nazwie *remoting*, który umożliwia uruchamianie dowolnego polecenia na komputerze zdalnym. W rzeczywistości można nawet uruchamiać polecenia dostępne na komputerze zdalnym, których nie ma na komputerze lokalnym, co oznacza, że nie zawsze musimy instalować wszystkie polecenia na lokalnej stacji roboczej. Taki mechanizm zdalnego dostępu jest naprawdę potężny i daje administratorom ogromne możliwości zdalnego zarządzania komputerami.

UWAGA Mechanizm zdalnego dostępu to ogromna, złożona technologia. W tym rozdziale spróbujemy Ci ją nieco przybliżyć i omówić najczęściej spotykane scenariusze użycia, które stanowią jakieś 80% do 90% przypadków zastosowań. Z oczywistych powodów nie jesteśmy jednak w stanie omówić wszystkich możliwych zastosowań, stąd w sekcji „Co dalej”, znajdującej się na końcu tego rozdziału, zamieszczamy kilka wskazówek i źródeł, w których znajdziesz bardziej szczegółowe informacje na temat mechanizmu komunikacji zdalnej w powłoce PowerShell.

13.1. Koncepcja komunikacji zdalnej w powłoce PowerShell

Zdalna sesja powłoki PowerShell działa podobnie jak Telnet i inne tradycyjne technologie komunikacji zdalnej. Po wywołaniu danego polecenia jest ono uruchamiane *na komputerze zdalnym*, a do Twojego komputera są przesyłane tylko wyniki jego działania. Jednak zamiast używać protokołu Telnet lub Secure Shell (SSH), powłoka PowerShell stosuje nowy protokół komunikacyjny o nazwie *Web Services for Management* (WS-MAN).

Protokół WS-MAN działa całkowicie za pośrednictwem protokołu HTTP lub HTTPS, dzięki czemu w razie potrzeby można łatwo przepuścić go przez zaporę sieciową (ponieważ każdy z tych protokołów wykorzystuje do komunikacji tylko jeden port). Implementacja protokołu WS-MAN została dokonana przez Microsoft w postaci usługi Windows Remote Management (WinRM) działającej w tle. Usługa WinRM jest instalowana wraz z powłoką PowerShell v2 i jest domyślnie uruchamiana w serwerowych systemach operacyjnych, takich jak Windows Server 2008 R2. Jest ona także domyślnie zainstalowana w systemie Windows 7, ale sama usługa jest wyłączona. Usługa WinRM jest również dołączona do powłoki PowerShell w wersji 3 oraz nowszych i jest domyślnie włączona w systemie Windows Server 2012 i nowszych wersjach (ogólnie rzecz biorąc, usługa WinRM jest domyślnie włączona na serwerach i domyślnie wyłączona na klientach).

Komunikacja zdalna za pośrednictwem protokołu SSH

W czasie kiedy oryginalne wydanie tej książki było przygotowywane do druku, firma Microsoft ogłosiła, że mechanizm komunikacji zdalnej powłoki PowerShell oprócz protokołu WS-MAN będzie mógł działać również z wykorzystaniem protokołu SSH. To dobra wiadomość dla firm, które mają już doświadczenie w pracy z protokołem SSH, ale nie mają pewności co do protokołu WS-MAN i usługi WinRM. Z perspektywy użytkownika nie powinno to jednak wpływać na sposób korzystania z mechanizmów zdalnego dostępu; różnice w używanych protokołach komunikacyjnych powinny być dla Ciebie czymś zupełnie niezauważalnym.

Zachęcamy Cię do zapoznania się z dokumentacją poleceń `Invoke-Command` i `New-PSSessionOption` zaraz po tym, jak wsparcie dla protokołu SSH zostanie oficjalnie uruchomione. Pamiętaj, że w tym rozdziale skupimy się wyłącznie na pracy zdalnej z wykorzystaniem protokołu WS-MAN i usługi WinRM.

Dowiedziałeś się już, że wynikiem działania poleceń powłoki Windows PowerShell są obiekty. Po uruchomieniu zdalnego polecenia obiekty będące wynikiem jego działania muszą zostać zamienione na postać, w której mogą być łatwo przesłane przez sieć przy

użyciu protokołu HTTP (lub HTTPS). Okazuje się, że świetnym rozwiązaniem jest zastosowanie języka XML, stąd powłoka PowerShell automatycznie dokonuje *serializacji* obiektów wyjściowych do postaci danych w formacie XML. Dane XML są przesyłane przez sieć do Twojego komputera, a następnie zostają automatycznie *zdeserializowane* do postaci obiektów, z którymi możesz pracować z poziomu powłoki PowerShell. Serializacja i deserializacja to tak naprawdę tylko forma konwersji formatu danych: z postaci obiektów na format XML (serializacja) i z formatu XML na postać obiektów (deserializacja).

Dlaczego powinieneś uważać na to, w jaki sposób przekazywane są dane wyjściowe? Ponieważ obiekty serializowane, a następnie deserializowane są tylko swego rodzaju migawkami, które nie podlegają ciągłej aktualizacji. Przykładowo, jeżeli w wyniku działania jakiegoś polecenia otrzymasz serię obiektów reprezentujących procesy działające na komputerze zdalnym, to takie informacje będą aktualne tylko w chwili, w której te obiekty zostały wygenerowane. Wartości takie jak zużycie pamięci i wykorzystanie CPU nie będą aktualizowane i nie będą tym samym automatycznie odzwierciedlać zmieniających się warunków. Powinieneś również pamiętać, że nie możesz zmusić deserializowanego obiektu do wykonania jakiejś operacji — na przykład nie możesz nakazać zatrzymania takiego procesu.

Są to podstawowe ograniczenia zdalnego dostępu, które jednak nie powstrzymują Cię od robienia naprawdę niesamowitych rzeczy. W rzeczywistości możesz nakazać zatrzymanie działania zdalnego procesu, ale musisz wiedzieć, co robisz. Nieco później w tym rozdziale pokażemy, jak to wykonać.

Aby pracować z wykorzystaniem komunikacji zdalnej, musisz spełnić dwa podstawowe wymagania:

- Zarówno Twój komputer, jak i komputer, na którym chcesz zdalnie wykonywać polecenia, muszą mieć zainstalowaną powłokę Windows PowerShell v2 lub nowszą. Windows XP jest najstarszą wersją systemu Windows, w której można zainstalować powłokę PowerShell v2, więc tym samym Windows XP jest najstarszą wersją systemu Windows, której możemy używać do zdalnego wykonywania poleceń.
- W idealnej sytuacji oba komputery powinny być członkami tej samej domeny lub zaufanych domen. Istnieje co prawda możliwość uzyskania zdalnego dostępu do komputerów znajdujących poza domeną, ale jest to znacznie trudniejsze i nie będziemy o tym mówić w tym rozdziale. Więcej szczegółowych informacji na ten temat znajdziesz w pliku pomocy po wykonaniu polecenia `help about_remote_trouble-shooting`.

ZRÓB TO SAM Mamy nadzieję, że będziesz w stanie samodzielnie wykonywać przynajmniej niektóre przykłady spośród omawianych w tym rozdziale. Aby mieć taką możliwość, powinieneś przygotować drugi komputer testowy lub maszynę wirtualną, które będą się znajdować w tej samej domenie usługi Active Directory co komputer testowy, którego używasz do tej pory. Na tym drugim komputerze możesz zainstalować dowolną wersję systemu Windows, pod warunkiem jednak, że będzie obsługiwała powłokę PowerShell v2 lub jej nowszą wersję. Jeżeli nie możesz użyć dodatkowego komputera lub maszyny wirtualnej, użyj hosta lokalnego

(localhost), aby utworzyć połączenie „zdalne” z Twoim własnym komputerem. W ten sposób nadal będziesz używał „zdalnego” dostępu, aczkolwiek z pewnością takie „zdalne” sterowanie swoim własnym komputerem, przy którym siedzisz, nie jest tak ekscytujące.

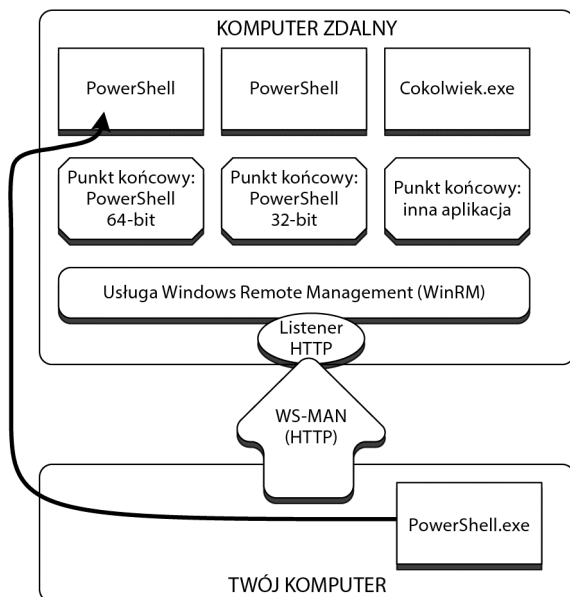
13.2. Usługa WinRM

Porozmawiajmy teraz o usłudze WinRM, ponieważ będziesz musiał ją odpowiednio skonfigurować, aby używać połączenia zdalnego. Pamiętaj jednak, że usługę WinRM oraz powłokę PowerShell musisz skonfigurować tylko na tych komputerach, które będą *odbierać* przychodzące polecenia. W większości środowisk, w których mieliśmy okazję pracować, administratorzy włączali możliwość komunikacji zdalnej na każdym komputerze z systemem Windows (należy pamiętać, że komunikacja zdalna w powłoce PowerShell jest obsługiwana na wszystkich systemach Windows począwszy od wersji XP). Dzięki takiemu rozwiązaniu możesz zdalnie wykonywać operacje w tle na różnych komputerach, a ich użytkownicy nie będą wiedzieli, że to robisz, co może być niezwykle przydatne.

Usługa WinRM nie jest przeznaczona wyłącznie dla powłoki PowerShell. Firma Microsoft zaczyna ją wykorzystywać do wykonywania coraz większej ilości zadań związanych ze zdalnym zarządzaniem systemami — nawet do takich, które obecnie używają innych protokołów komunikacyjnych. Mając to na uwadze, firma Microsoft zaimplementowała w usłudze WinRM możliwość kierowania ruchu do wielu aplikacji związanych z zarządzaniem komputerem, a nie tylko do powłoki PowerShell. Usługa WinRM działa jako dyspozytor: kiedy nadchodzi takie połączenie, WinRM decyduje, która aplikacja musi je obsługiwać. Cały ruch związany z WinRM jest oznaczony nazwą aplikacji odbiorcy. Wszystkie takie aplikacje muszą się wcześniej zarejestrować w usłudze WinRM jako **punkty końcowe** (ang. *endpoints*), tak aby mogła ona nasłuchiwać, odbierać i odpowiednio kierować przeznaczony dla nich ruch przychodzący. W praktyce oznacza to, że na stacji końcowej musisz nie tylko włączyć usługę WinRM, ale również nakazać powłoce PowerShell zarejestrować się jako *punkt końcowy* dla tej usługi. Na rysunku 13.1 pokazujemy, w jaki sposób poszczególne elementy są ze sobą połączone.

Jak widać, w swoim systemie możesz mieć dziesiątki lub nawet setki punktów końcowych WinRM (powłoka PowerShell nazywa je **konfiguracjami sesji**). Każdy punkt końcowy może wskazywać na inną aplikację, ale możesz również mieć punkty końcowe, które wskazują tę samą aplikację, ale zapewniają różne uprawnienia i funkcjonalności. Na przykład można utworzyć punkt końcowy powłoki PowerShell, który pozwala tylko na wykonywanie jednego lub dwóch poleceń, i udostępnić go konkretnym użytkownikom w danym środowisku. W tym rozdziale nie będziemy się jednak zbyt głęboko zagłębiać w takie zagadnienia, ale nadrobimy to w rozdziale 23.

Na rysunku 13.1 możesz zobaczyć również listenera usługi WinRM, który w tym przypadku wykorzystuje protokół HTTP. Listener oczekuje na przychodzący ruch sieciowy przeznaczony dla usługi WinRM — co przypomina nieco rolę serwera internetowego oczekującego na nadejście żądania. Listener nasłuchuje na określonym porcie



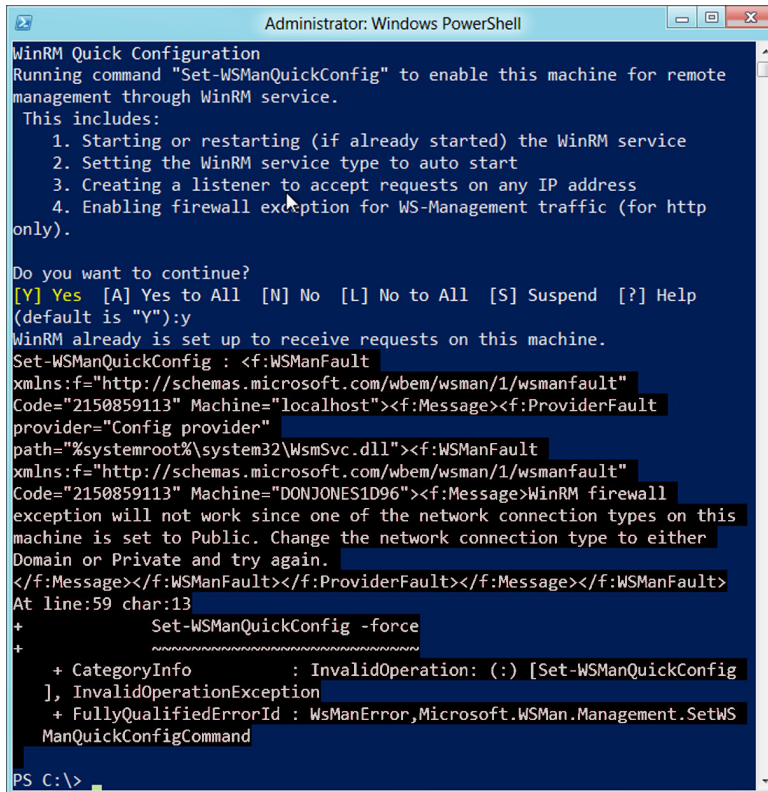
Rysunek 13.1. Powiązania pomiędzy WinRM, WS-MAN, punktami końcowymi i powłoką PowerShell

i na określonym adresie IP, chociaż listener domyślny utworzony przez polecenie `Enable-PSRemoting` nasłuchuje na *wszystkich* lokalnych adresach IP.

Listener łączy się z określonym punktem końcowym. Jednym ze sposobów utworzenia punktu końcowego jest uruchomienie z poziomu administratora systemu osobnej sesji powłoki PowerShell i uruchomienie polecenia `Enable-PSRemoting`. Czasami możesz spotkać się w tym kontekście z innym poleceniem, o nazwie `Set-WSManQuickConfig`. Nie musisz jednak jego uruchamiać; cmdlet `Enable-PSRemoting` zrobi to za Ciebie i dodatkowo wykona kilka innych kroków, które są niezbędne do uruchomienia komunikacji zdalnej. Krótko mówiąc, polecenie `Enable-PSRemoting` uruchomi usługę WinRM, skonfiguruje ją tak, aby uruchamiała się automatycznie po załadowaniu systemu, zarejestruje powłokę PowerShell jako punkt końcowy, a nawet w razie potrzeby skonfiguruje odpowiedni wyjątek dla zapory sieciowej systemu Windows, żeby zezwolić na przychodzący ruch WinRM.

ZRÓB TO SAM Spróbuj samodzielnie włączyć mechanizm komunikacji zdalnej na drugim komputerze (lub na pierwszym, jeżeli pracujesz z jedną maszyną). Upewnij się, że uruchomiłeś powłokę PowerShell jako administrator (na pasku tytułu okna widnieje napis *Administrator*). Jeżeli tak nie jest, zamknij okno powłoki, kliknij prawym przyciskiem myszy ikonę powłoki PowerShell w menu *Start*, a następnie z menu podręcznego wybierz opcję *Uruchom jako administrator*. Jeżeli po włączeniu zdalnego dostępu pojawi się komunikat o błędzie, zatrzymaj się i sprawdź jego przyczynę. Nie będziesz mógł kontynuować, dopóki polecenie `Enable-PSRemoting` nie zostanie wykonane bez błędu.

Na rysunku 13.2 przedstawiono jeden z najczęstszych błędów, które mogą się pojawić po uruchomieniu polecenia `Enable-PSRemoting`.



```

Administrator: Windows PowerShell

WinRM Quick Configuration
Running command "Set-WSManQuickConfig" to enable this machine for remote
management through WinRM service.
This includes:
  1. Starting or restarting (if already started) the WinRM service
  2. Setting the WinRM service type to auto start
  3. Creating a listener to accept requests on any IP address
  4. Enabling firewall exception for WS-Management traffic (for http
only).

Do you want to continue?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):y
WinRM already is set up to receive requests on this machine.
Set-WSManQuickConfig : <f:WSManFault
xmlns:f="http://schemas.microsoft.com/wbem/wsman/1/wsmanfault"
Code="2150859113" Machine="localhost"><f:Message><f:ProviderFault
provider="Config provider"
path="%systemroot%\system32\WsmSvc.dll"><f:WSManFault
xmlns:f="http://schemas.microsoft.com/wbem/wsman/1/wsmanfault"
Code="2150859113" Machine="DONJONES1D96"><f:Message>WinRM firewall
exception will not work since one of the network connection types on this
machine is set to Public. Change the network connection type to either
Domain or Private and try again.
</f:Message></f:WSManFault></f:ProviderFault></f:Message></f:WSManFault>
At line:59 char:13
+ ~~~~~
+ Set-WSManQuickConfig -force
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [Set-WSManQuickConfig
], InvalidOperationException
+ FullyQualifiedErrorId : WsManError,Microsoft.WSMan.Management.SetWS
ManQuickConfigCommand

PS C:\>

```

Rysunek 13.2. Najczęstszy błąd pojawiający się po włączeniu komunikacji zdalnej na komputerze klienckim

Ten błąd zwykle występuje tylko na komputerach klienckich, a sam komunikat o błędzie dokładnie wskazuje, na czym polega problem. W naszym przypadku mamy co najmniej jeden interfejs sieciowy skonfigurowany do sieci typu *Public*. Należy pamiętać, że w systemie Windows Vista i nowszych wersjach do każdego interfejsu sieciowego przypisywany jest typ sieci — *Praca* (*Work*), *Dom* (*Home*) lub *Publiczna* (*Public*). Dla sieci zdefiniowanych jako publiczne nie możemy tworzyć wyjątków zapory sieciowej systemu Windows, zatem kiedy uruchamiamy polecenie `Enable-PSRemoting`, próba utworzenia wyjątków kończy się niepowodzeniem. Jedynym rozwiązaniem jest przejście do systemu Windows i zmodyfikowanie ustawień interfejsu sieciowego tak, aby dowolna sieć, z której korzystamy, miała status *Praca* lub *Dom*. Pamiętaj jednak, że nie powinieneś tego robić, jeżeli masz połączenie z siecią publiczną (na przykład w publicznym punkcie dostępowym sieci bezprzewodowej), ponieważ w taki sposób wyłączasz ważne zabezpieczenia Twojego komputera.

UWAGA W przypadku włączania komunikacji zdalnej dla powłoki PowerShell na serwerowych systemach operacyjnych nie musisz się przejmować ustawieniami sieci publicznych, ponieważ takie systemy nie mają tego typu ograniczeń.

Jeżeli nie uśmiecha Ci się konieczność chodzenia od komputera do komputera i ręcznego włączania obsługi zdalnego dostępu, nie martw się: możesz to zrobić także za pomocą odpowiednio ustawionego obiektu reguł grupy (GPO — ang. *Group Policy Object*). Niezbędne ustawienia GPO są wbudowane w kontrolery domeny działające pod kontrolą systemu Windows Server 2008 R2 i nowszych (aby dodać takie ustawienia GPO do kontrolerów domeny działających pod kontrolą starszych systemów, możesz ze strony <http://download.microsoft.com> pobrać odpowiedni szablon ADM). Uruchom edytor GPO i zajrzyj do sekcji *Konfiguracja komputera/Szablony administracyjne/Składniki systemu Windows*. U dołu listy znajdziesz zarówno opcję *Powłoka zdalna systemu Windows*, jak i *Zdalne zarządzanie systemem Windows (WinRM)*. Na razie zakładamy jednak, że po prostu uruchomisz polecenie `Enable-PSRemoting` na tych komputerach, które chcesz skonfigurować do komunikacji zdalnej, ponieważ w tym momencie prawdopodobnie korzystasz tylko z jednej czy dwóch maszyn wirtualnych.

UWAGA W pliku pomocy `about_remote_troubleshooting` powłoki PowerShell znajdziesz wiele szczegółowych informacji na temat używania obiektów GPO. Poszukaj tam sekcji zatytułowanych *How to enable remoting in an enterprise* (jak włączyć obsługę zdalnego dostępu w przedsiębiorstwie) oraz *How to enable listeners by using a Group Policy* (jak uruchomić listenery przy użyciu zasad grupy).

Usługa WinRM v2 (z której korzysta powłoka PowerShell v2 i nowsze wersje) domyślnie używa portu TCP 5985 dla połączeń HTTP i portu 5986 dla połączeń HTTPS. Takie numery portów zapewniają, że listener nie będzie w konflikcie z lokalnie zainstalowanymi serwerami WWW, które używają portów 80 i 443. Domyślna konfiguracja zdalnego połączenia włączana przez polecenie `Enable-PSRemoting` uruchamia tylko listenera nasłuchującego nieszyfrowanej komunikacji HTTP na porcie 5985. Oczywiście w razie potrzeby możesz skonfigurować usługę WinRM tak, aby używała innych numerów portów, ale nie zalecamy takiego postępowania. Jeżeli pozostawisz porty domyślne, masz pewność, że wszystkie polecenia komunikacji zdalnej powłoki PowerShell będą działać normalnie. Jeśli zmienisz numery portów, zawsze podczas uruchamiania zdalnego polecenia będziesz musiał podawać odpowiedni numer portu, co może być uciążliwe i będzie wymagało dodatkowego klikania.

Jeżeli jednak z takiego czy innego powodu staniesz przed koniecznością zmiany domyślnych numerów portów, możesz to zrobić, uruchamiając następujące polecenie:

```
Winrm set winrm/config/listener?Address=**+Transport=HTTP@{Port="1234"}
```

W tym przykładzie zmieniamy domyślny numer portu usługi WinRM na 1234. Teraz samodzielnie spróbuj zmodyfikować to polecenie tak, aby zamiast nieszyfrowanego protokołu HTTP korzystać z protokołu HTTPS.

Nie próbuj tego teraz

Choć w środowisku produkcyjnym zmiana domyślnego portu usługi WinRM może się okazać niezbędna, nie zmieniaj go na swoim komputerze testowym. Jeżeli pozostawisz ustawienia usługi WinRM w domyślnej konfiguracji, będziesz miał pewność, że pozostałe przykłady z tej książki będą działały bez żadnych modyfikacji.

Musimy tutaj przyznać, że w praktyce istnieje również możliwość takiego skonfigurowania usługi WinRM na komputerach klienckich, aby domyślnie używała innego numeru portu, dzięki czemu nie będziesz musiał przy każdym wywołaniu polecenia zdalnego podawać odpowiedniego numeru portu komunikacyjnego. Na razie jednak trzymajmy się domyślnych ustaleń firmy Microsoft. Istnieje również możliwość tworzenia wielu listenerów dla usługi WinRM — na przykład jednego dla ruchu HTTP i jednego dla zaszyfrowanego ruchu HTTPS lub kolejnych listenerów dla różnych adresów IP. Wszystkie utworzone w taki sposób listenery będą dostarczały ruch do odpowiednich punktów końcowych skonfigurowanych na komputerze.

UWAGA Jeżeli przyglądałeś się ustawieniom obiektu zasad grupy dla powłoki zdalnej, z pewnością zauważyłeś, że możesz określić, jak długo sesja zdalna może pozostać bezczynna, zanim serwer ją zakończy, ilu równoczesnych użytkowników może zdalnie połączyć się z serwerem, ile pamięci oraz ile procesów może wykorzystywać każda powłoka zdalna, ile powłok zdalnych może otworzyć jednocześnie dany użytkownik i tak dalej. Są to świetne ustawienia pozwalające zapewnić, że Twoje serwery nie będą zbyt obciążone przez zapominałskich administratorów. Domyślnie jednak, aby używać połączeń zdalnych, musisz *być* administratorem systemu, a więc nie musisz się martwić, że zwykli użytkownicy będą zapychać Twoje serwery.

13.3. Używanie poleceń *Enter-PSSession* i *Exit-PSSession* do połączeń zdalnych typu jeden-do-jednego

Powłoka PowerShell używa dwóch różnych modeli połączeń zdalnych. Pierwszy z nich nazywa się **jeden-do-jednego** lub **1:1** (ang. *one-to-one*). Drugi model to **jeden-do-wielu** lub **1:N** (ang. *one-to-many*) i będziemy o nim pisać w kolejnym podrozdziale. Przy połączeniach typu jeden-do-jednego uzyskujesz dostęp do powłoki na jednym komputerze zdalnym. Wszelkie wykonywane polecenia będą uruchamiane bezpośrednio na tym komputerze zdalnym, a wyniki będą wyświetlane w Twoim oknie powłoki. Jest to trochę podobne do korzystania z usługi pulpitu zdalnego (ang. *Remote Desktop Connection*), z tym że ogranicza się tylko do środowiska wiersza poleceń powłoki Windows PowerShell. Z drugiej jednak strony taki rodzaj połączenia zdalnego wykorzystuje tylko *niewielką część* zasobów, których wymaga uruchomienie pełnego pulpitu zdalnego, zatem narzut na obciążenie serwera jest znacznie mniejszy.

Aby nawiązać połączenie typu jeden-do-jednego z komputerem zdalnym, uruchom następujące polecenie:

```
Enter-PSSession -computerName Server-R2
```

(Oczywiście zamiast *Server-R2* musisz podać nazwę odpowiedniego komputera).

Przy założeniu, że włączona usługa dostępu zdalnego została uruchomiona na komputerze zdalnym, oba komputery znajdują się w tej samej domenie, a sieć działa poprawnie, połączenie powinno zostać nawiązane prawidłowo. Powłoka PowerShell poinformuje Cię o tym, odpowiednio zmieniając znak zachęty:

```
[Server-r2] PS C:\>
```

Znak zachęty powłoki wskazuje teraz, że wszystko, co robisz, będzie wykonywane na maszynie zdalnej o nazwie *Server-R2* (lub dowolnym innym komputerze, z którym się połączyłeś). Możesz teraz uruchamiać dowolne polecenia, a nawet importować dowolne moduły lub dodawać przystawki PSSnapin, które są zainstalowane na tym komputerze zdalnym.

ZRÓB TO SAM Spróbuj samodzielnie utworzyć połączenie zdalne z drugim komputerem lub maszyną wirtualną. Jeżeli jeszcze tego nie zrobiłeś do tej pory, przed dokonaniem próby połączenia powinieneś włączyć usługę dostępu zdalnego na tym komputerze. Pamiętaj, że musisz znać prawdziwą nazwę komputera zdalnego; usługa WinRM domyślnie nie zezwala na łączenie się przy użyciu adresu IP lub aliasu DNS.

Twoje uprawnienia są przenoszone przez połączenie zdalne. Twoja powłoka PowerShell będzie przekazywać każdy token bezpieczeństwa, z którym została uruchomiona (będzie to robić za pomocą usługi Kerberos, więc nie przekaże nazwy Twojego konta użytkownika ani hasła poprzez sieć). Każde polecenie uruchomione na komputerze zdalnym będzie działało w kontekście Twojego konta użytkownika, dzięki czemu będziesz mógł wykonywać wszystkie operacje, do których masz uprawnienia. To tak samo, jakbyś zalogował się do konsoli tego komputera i bezpośrednio używał na nim swojej konsoli PowerShell.

No prawie tak samo. Przyjrzyjmy się zatem, jakie są różnice:

- Nawet jeżeli na komputerze zdalnym masz swój skrypt profilu powłoki PowerShell, to po nawiązaniu połączenia za pomocą dostępu zdalnego nie będzie on działał. Nie opisywaliśmy jeszcze szczegółowo skryptów profili (będziemy o tym mówić w rozdziale 25.), ale wystarczy tutaj powiedzieć, że skrypt profilu to zestaw poleceń uruchamianych automatycznie przy każdym uruchomieniu powłoki. Użytkownicy wykorzystują takie skrypty do automatycznego ładowania rozszerzeń i modułów powłoki oraz wykonywania wielu innych operacji. Jeżeli jednak nawiązujesz połączenie zdalne z takim komputerem, skrypt profilu nie jest uruchamiany. Pamiętaj o tym.

- Po nawiązaniu połączenia nadal podlegasz ograniczeniom wprowadzanym na komputerze zdalnym przez reguły polityki wykonywania skryptów (ang. *execution policy*). Załóżmy, że polityka wykonywania skryptów na Twoim komputerze lokalnym jest ustawiona na opcję `RemoteSigned`, co oznacza, że możesz uruchamiać lokalne skrypty bez podpisu cyfrowego. To świetnie, ale jeżeli na komputerze zdalnym polityka wykonywania skryptów jest ustawiona na domyślną opcję `Restricted`, to nie będziesz mógł uruchomić na nim żadnych skryptów.

Oprócz tych dwóch dość niewielkich zastrzeżeń cała reszta powinna działać normalnie. Ale czekaj — co zrobisz, kiedy wykonasz już wszystkie niezbędne polecenia na komputerze zdalnym? Bardzo wiele poleceń powłoki PowerShell występuje w parach, gdzie jedno polecenie wykonuje określoną operację, a drugie, komplementarne polecenie realizuje operację odwrotną. No dobrze, w takim przypadku, skoro polecenie `Enter-PSSession` pozwala na *nawiązanie połączenia* z komputerem zdalnym, czy możesz odgadnąć, za pomocą jakiego polecenia możesz taką sesję *zakończyć*? Jeżeli odpowiedziałeś, że będzie to polecenie `Exit-PSSession`, to zasłużyłeś na nagrodę. Polecenie `Exit-PSSession` nie wymaga podawania żadnych parametrów wywołania; wystarczy je po prostu uruchomić i połączenie zdalne zostanie zakończone, a znak zachęty powłoki powróci do normalnego stanu.

ZRÓB TO SAM Jeżeli wcześniej utworzyłeś połączenie zdalne, spróbuj je teraz zakończyć. Na razie to wszystko na ten temat.

A co powinieneś zrobić, jeżeli zapomniałeś wykonać polecenie `Exit-PSSession` i zamiast tego po prostu zamknąłeś okno powłoki PowerShell? Nie martw się. Powłoka PowerShell i usługa WinRM są wystarczająco inteligentne, aby zorientować się, co się stało, i połączenie zdalne zostanie zamknięte automatycznie.

Mamy dla Ciebie jedną, ważną uwagę: jeżeli nawiązałeś połączenie zdalne, nie próbuj uruchamiać kolejnego polecenia `Enter-PSSession` z *poziomu komputera zdalnego*, chyba że naprawdę wiesz, co robisz. Załóżmy, że pracujesz na komputerze A, na którym działa system Windows 7, i nawiązałeś połączenie zdalne z maszyną o nazwie `Server-R2`. Następnie, pracując na zdalnej konsoli PowerShell, uruchamiasz poniższe polecenie:

```
[server-r2] PS C:\>enter-pssession server-dc4
```

Wykonanie takiego polecenia spowoduje, że `Server-R2` utworzy połączenie zdalne z maszyną o nazwie `Server-DC4`, co może rozpocząć tworzenie trudnego do prześledzenia *łańcucha zdalnych połączeń*, który w dodatku spowoduje niepotrzebne obciążenie serwerów. Czasami jednak można to zrobić — myślimy tutaj głównie o sytuacjach, w których komputer taki jak `Server-DC4` znajduje się za zaporą sieciową i nie możesz uzyskać do niego bezpośredniego dostępu. W takiej sytuacji możesz użyć komputera `Server-R2` jako swego rodzaju pośrednika, za pomocą którego nawiązujesz połączenie z `Server-DC4`. Zasadniczo jednak powinieneś starać się unikać tworzenia takiego łańcucha połączeń zdalnych.

OSTRZEŻENIE Niektórzy użytkownicy określają taką sekwencję połączeń zdalnych jako „drugi przeskok”, jednak utworzenie takiego połączenia może być dla użytkownika nieco mylące. Nasza recepta na uniknięcie nieporozumień jest prosta: jeżeli w znaku zachęty powłoki PowerShell wyświetlana jest nazwa komputera, to nie powinieneś tworzyć kolejnego połączenia zdalnego dopóty, dopóki nie zakończysz tej sesji i „nie powrócisz” do swojego komputera. Włączanie możliwości tworzenia połączeń zdalnych z wykorzystaniem wielu hostów pośrednich (ang. *multihop remoting*) zostanie omówione w rozdziale 23.

Jeżeli korzystasz z połączeń zdalnych typu jeden-do-jednego, nie musisz się martwić o serializację i deserializację obiektów. Z punktu widzenia użytkownika po nawiązaniu połączenia po prostu piszesz bezpośrednio na konsoli komputera zdalnego. Jeżeli pobierzesz obiekt procesu i przekażesz go za pomocą potoku do polecenia `Stop-Process`, to taki proces zostanie zatrzymany, tak jak mógłbyś tego oczekiwać.

13.4. Zastosowanie polecenia *Invoke-Command* do komunikacji zdalnej typu *jeden-do-wielu*

Kolejną sztuczką — i szczerze mówiąc, jest to jedna z najfajniejszych rzeczy, jakie możesz zrobić przy użyciu powłoki Windows PowerShell — jest *wysyłanie polecenia do wielu komputerów zdalnych w tym samym czasie*. Zgadza się, możemy to już nazwać przetwarzaniem rozproszonym na pełną skalę. Każdy z komputerów zdalnych samodzielnie wykona otrzymane polecenie i odeśle jego wyniki działania z powrotem do Twojego komputera. Wszystko odbywa się za pomocą polecenia `Invoke-Command`; taki rodzaj komunikacji zdalnej nazywamy połączeniami typu *jeden-do-wielu* lub po prostu 1:N.

Samo polecenie ma taką oto postać:

```
Invoke-Command -computerName Server-R2,Server-DC4,Server12  
-command { Get-EventLog Security -newest 200 |  
Where { $_.EventID -eq 1212 }}
```

ZRÓB TO SAM Spróbuj samodzielnie uruchomić takie polecenie. W miejsce naszych przykładowych maszyn wstaw odpowiednie nazwy swoich komputerów.

Wszystko, co znajduje się w najbardziej zewnętrznych nawiasach klamrowych {}, zostaje przekazane do komputerów zdalnych — wszystkich trzech. Domyślnie powłoka PowerShell może komunikować się maksymalnie z 32 komputerami naraz; jeżeli w wierszu wywołania polecenia podasz ich więcej, to powłoka automatycznie ustawi je w kolejce i będzie sukcesywnie wysyłała to polecenie do następnych maszyn, w miarę jak poprzednie komputery będą kończyły jego wykonywanie. Jeżeli jednak masz szybką sieć i mocne komputery, możesz zwiększyć maksymalną liczbę uruchamianych jednocześnie maszyn, zmieniając wartość parametru `-throttleLimit` polecenia `Invoke-Command`; więcej szczegółowych informacji na ten temat znajdziesz w pliku pomocy dla tego polecenia.

Uważaj na składnię poleceń

Chcielibyśmy nieco bardziej szczegółowo omówić przykład realizujący zdalne połączenie typu jeden-do-wielu, ponieważ w takich sytuacjach składnia poleceń powłoki PowerShell może być nieco myląca, co w efekcie może prowadzić do nieporozumień i pomyłek podczas samodzielnego tworzenia takich poleceń.

Rozważmy następujący przykład:

```
Invoke-Command -computerName Server-R2,Server-DC4,Server12  
-command { Get-EventLog Security -newest 200 |  
Where { $_.EventID -eq 1212 } }
```

Dwa polecenia w tym przykładzie używają nawiasów klamrowych: `Invoke-Command` i `Where` (alias dla polecenia `Where-Object`). Polecenie `Where` jest całkowicie zagnieżdżone w zewnętrznym zestawie nawiasów klamrowych, który zamyka wszystko, co jest wysyłane do zdalnych komputerów w celu wykonania:

```
Get-EventLog Security -newest 200 | Gdzie { $_. EventID -eq 1212 }
```

Analizowanie zagnieżdżonych poleceń może być trudne, szczególnie w książce takiej jak ta, gdzie ograniczenia związane z fizyczną szerokością strony powodują konieczność podzielenia polecenia i rozpisania go w kilku wierszach tekstu.

Zawsze powinieneś się upewnić, że potrafisz dokładnie zidentyfikować polecenie wysyłane do zdalnego komputera i że rozumiesz, do czego służy każdy z użytych zestawów nawiasów klamrowych.

W tym miejscu musimy powiedzieć, że parametru `-command` nie znajdziesz w pliku pomocy dla polecenia `Invoke-Command`, ale mimo to polecenie, które Ci pokazaliśmy wcześniej, działa jak najbardziej prawidłowo. Parametr `-command` jest *aliasem* parametru `-scriptblock`, który oczywiście zobaczysz w pliku pomocy. Szczerze mówiąc, alias `-command` jest chyba łatwiejszy do zapamiętania, więc zdecydowanie chętniej używamy go niż parametru `-scriptblock`, ale oba działają w ten sam sposób.

Jeśli zapoznałeś się uważnie z zawartością pliku pomocy polecenia `Invoke-Command` (jak widać, nadal konsekwentnie naciskamy na korzystanie z rodzimego systemu pomocy powłoki PowerShell), z pewnością zauważyłeś także parametr, który pozwala na wskazanie skryptu zapisanego w pliku, a nie tylko pojedynczego polecenia. Parametr ten umożliwia wysłanie całego skryptu z komputera lokalnego na komputery zdalne, co znacząco ułatwia automatyzację wielu złożonych zadań i wykonywanie ich na wielu komputerach jednocześnie.

ZRÓB TO SAM Zapoznaj się z zawartością pliku pomocy dla polecenia `Invoke-Command`, uważnie przeczytaj opis parametru `-scriptblock` oraz poszukaj parametru, który umożliwiłby podanie ścieżki i nazwy skryptu zamiast pojedynczego polecenia.

Teraz chcielibyśmy powrócić do parametru `-computerName`, o którym wspomnieliśmy na początku rozdziału. Kiedy po raz pierwszy używaliśmy polecenia `Invoke-Command`, wpisywaliśmy listę nazw komputerów, oddzielając je od siebie przecinkami, tak jak w poprzednim przykładzie. Ale kiedy często pracujemy z wieloma komputerami, wpisywanie wszystkich nazw za każdym razem może być bardzo uciążliwe i z pewnością nie chcemy tego robić. Dobrym rozwiązaniem takiego problemu jest przechowywanie odpowiednich zestawów

nazw komputerów w osobnych plikach tekstowych, na przykład listy kontrolerów domen w jednym pliku, listy serwerów WWW w kolejnym i tak dalej. W pliku tekstowym umieszczamy po jednej nazwie komputera w każdym wierszu, i to wszystko — bez przecinków, cudzysłówów i żadnych innych dodatkowych elementów. Powłoka PowerShell znakomicie ułatwia i wspomaga korzystanie z takich plików:

```
Invoke-Command -command {dir}
               -computerName (Get-Content webservers.txt)
```

Nawiasy użyte w tym poleceniu powodują, że powłoka PowerShell najpierw wykonuje polecenie `Get-Content` — w podobny sposób nawiasy działają w matematyce. Wyniki działania polecenia `Get-Content` są następnie przekazywane do parametru `-computerName`, dzięki czemu główne polecenie jest wykonywane na każdym z komputerów wymienionych w pliku.

Istnieje również możliwość pobierania nazw komputerów bezpośrednio z usługi Active Directory, ale jest to nieco trudniejsze. Możemy do tego celu użyć polecenia `Get-ADComputer` (dostępnego na kontrolerach domeny działających pod kontrolą systemu Windows Server 2008 R2 i nowszych), jednak nie możemy go umieszczać w nawiasach, tak jak to robiliśmy w poprzednim przykładzie z poleceniem `Get-Content`. Wynikiem działania polecenia `Get-Content` są proste łańcuchy tekstu, których oczekuje parametr `-computerName`, natomiast wynikiem działania polecenia `Get-ADComputer` są złożone obiekty, z którymi parametr `-computerName` po prostu nie wie, co ma zrobić.

Jeżeli chcemy korzystać z polecenia `Get-ADComputer`, musimy znaleźć sposób na pobieranie z obiektów wynikowych tylko wartości właściwości reprezentujących nazwy komputerów. Poniższy przykład pokazuje, jak możemy to zrobić:

```
Invoke-Command -command { dir } -computerName (
Get-ADComputer -filter * -searchBase "ou=Sales,dc=company,dc=pri" |
Select-Object -expand Name )
```

ZRÓB TO SAM Jeżeli używasz powłoki PowerShell na kontrolerze domeny działającym pod kontrolą systemu Windows Server 2008 R2 (lub nowszym) albo na komputerze z systemem Windows 7 (lub nowszym), na których został zainstalowany pakiet Remote Server Administration Tools (RSAT), możesz wykonać polecenie `Import-Module ActiveDirectory` i następnie spróbować uruchomić poprzednie polecenie. Jeżeli w Twojej domenie nie ma jednostki organizacyjnej o nazwie Sales, zmień `ou=Sales` na `ou=Domain Controllers`. Pamiętaj również, aby odpowiednio zaktualizować pozostałe informacje o domenie (na przykład jeżeli Twoja domena to `moja_domena.org`, powinieneś zamiast `company` wstawić `moja_domena`, a zamiast `pri` — wstawić `org`).

W nawiasach przekazujemy za pomocą potoku obiekty do polecenia `Select-Object`, w którym użyliśmy parametru `-expand`. Powoduje to, że polecenie wyodrębnia wartość właściwości `Name` obiektów, które pojawiają się na jego wejściu, czyli w tym przypadku — obiektów reprezentujących komputery. Wynikiem całego wyrażenia znajdującego

się w nawiasach będzie grupa nazw komputerów, a nie obiektów komputerowych — a nazwy komputerów są przecież dokładnie tym, czego oczekuje parametr `-computerName`.

UWAGA Mamy nadzieję, że powyższa dyskusja na temat parametru `-Expand` wywołała u Ciebie odczucie pewnego *déjà vu* — tak, z parametrem `-Expand` spotkałeś się już wcześniej, w rozdziale 9. W razie potrzeby możesz powrócić do tego rozdziału, aby sobie odświeżyć omawiane tam zagadnienia.

Nie sposób omawiać polecenia `Get-ADComputer`, nie wspominając o jego niezmiernie istotnym parametrze `-filter`, który w naszym przykładzie określa, że w wynikach działania tego polecenia powinny zostać zawarte wszystkie znalezione komputery. Parametr `-searchBase` wskazuje miejsce w domenie, od którego należy rozpocząć wyszukiwanie komputerów — w tym przypadku OU o nazwie `Sales` domeny `company.pri`. Jeszcze raz przypominamy, że polecenie `Get-ADComputer` jest dostępne tylko na kontrolerach domeny działających pod kontrolą systemu Windows Server 2008 R2 (i nowszych) oraz na komputerach klienckich z systemem Windows 7 (i nowszymi), na których zainstalowano pakiet narzędzi RSAT (ang. *Remote Server Administration Tools*).

13.5. Różnice między poleceniami zdalnymi i lokalnymi

Teraz chcielibyśmy wyjaśnić, na czym polega różnica między uruchamianiem poleceń za pomocą polecenia `Invoke-Command` a uruchamianiem tego samego polecenia lokalnie. Przy okazji wspomnimy również o różnicach między zdalnym wykonywaniem poleceń a innymi formami komunikacji zdalnej. Jako przykładu bazowego w naszej dyskusji użyjemy następującego polecenia:

```
Invoke-Command -computerName Server-R2,Server-DC4,Server12
-command { Get-EventLog Security -newest 200 |
Where { $_.EventID -eq 1212 } }
```

Przyjrzyjmy się wybranym alternatywom tego polecenia i zobaczmy, czym się różnią.

13.5.1. Polecenie `Invoke-Command` kontra parametr `-computerName`

Oto inny sposób wykonania naszego przykładowego zadania bazowego:

```
Get-EventLog Security -newest 200
-computerName Server-R2,Server-DC4,Server12
| Where { $_.EventID -eq 1212 }
```

W powyższym przykładzie zamiast zdalnego wywoływania całego polecenia używaliśmy parametru `-computerName` polecenia `Get-EventLog`. Otrzymane wyniki będą mniej więcej takie same, ale istnieją pewne istotne różnice w sposobie wykonywania tej wersji polecenia:

- Wywołania dla poszczególnych komputerów są realizowane sekwencyjnie, a nie równoległe, co oznacza, że wykonanie polecenia może potrwać znacznie dłużej.

- W wynikach działania nie znajdziemy właściwości `PSComputerName`, co może mocno utrudnić stwierdzenie, które wyniki pochodzą z którego komputera.
- Połączenie nie jest nawiązywane za pomocą usługi WinRM, ale zamiast tego wykorzystywany jest dowolny protokół bazowy, z którego korzysta .NET Framework. Nie wiemy, jaki to będzie protokół, i dlatego uzyskanie połączenia przez zaporę sieciową z komputerami zdalnymi może być znacznie trudniejsze.
- Pobieramy po 200 rekordów z każdego z trzech wymienionych komputerów i dopiero wtedy filtrujemy je tak, aby wyszukać zdarzenia o identyfikatorze EventID 1212. Oznacza to, że najprawdopodobniej przesyłamy z komputerów zdalnych bardzo wiele rekordów, które są nam zupełnie niepotrzebne.
- Wyniki działania zawierają obiekty dziennika zdarzeń, które są w pełni funkcjonalne.

Powyższe różnice dotyczą dowolnego polecenia z parametrem `-computerName`. Ogólnie rzecz biorąc, użycie polecenia `Invoke-Command` może być znacznie bardziej wydajne i efektywne, niż miałyby to miejsce w przypadku zastosowania parametru `-computerName`.

A oto co się stanie, jeżeli użyjemy naszego bazowego polecenia `Invoke-Command`:

- Wywołania dla poszczególnych komputerów są realizowane równolegle, a zatem wykonanie polecenia może potrwać znacznie krócej.
- W wynikach działania znajdziemy właściwość `PSComputerName`, co umożliwia łatwiejsze odróżnienie danych pochodzących z poszczególnych komputerów.
- Połączenie jest nawiązywane za pośrednictwem usługi WinRM, która wykorzystuje jeden predefiniowany port sieciowy, co może znacząco ułatwić uzyskiwanie połączenia poprzez zaporę sieciową.
- Każdy komputer zdalny pobiera z dziennika 200 rekordów i filtruje je lokalnie, dzięki temu jedynymi danymi wyjściowymi przesyłanymi przez sieć są wyniki tego filtrowania, w wyniku czego przesyłane są tylko te rekordy, które są nam potrzebne.
- Przed transmisją wyników każdy komputer serializuje swoje dane wyjściowe do formatu XML. Nasz komputer otrzymuje dane w formacie XML i deserializuje je z powrotem do postaci czegoś, co wygląda jak obiekty. W rzeczywistości jednak nie są to pełnoprawne obiekty dziennika zdarzeń, co może nieco ograniczać nasze możliwości ich przetwarzania po przesłaniu ich na nasz komputer lokalny.

Ten ostatni punkt to bardzo ważna różnica między używaniem parametru `-computerName` a polecenia `Invoke-Command`. Przyjrzyjmy się temu nieco bliżej.

13.5.2. Przetwarzanie lokalne kontra zdalne

Powtórzymy jeszcze raz nasz oryginalny przykład bazowy:

```
Invoke-Command -computerName Server-R2,Server-DC4,Server12
-command { Get-EventLog Security -newest 200 |
Where { $_.EventID -eq 1212 }}
```

Teraz porównajmy go z alternatywną wersją tego polecenia:

```
Invoke-Command -computerName Server-R2,Server-DC4,Server12  
-command { Get-EventLog Security -newest 200 } |  
Where { $_.EventID -eq 1212 }
```

Jak widać, różnice są bardzo subtelne. W zasadzie możemy tutaj znaleźć tylko jedną różnicę: jedna para nawiasów klamrowych została przeniesiona w inne miejsce.

W drugiej wersji zdalnie wywoływane jest tylko polecenie `Get-EventLog`. Wszystkie wyniki generowane przez `Get-EventLog` są serializowane i przekazywane do naszego komputera, gdzie są ponownie przekształcane (deserializowane) do postaci obiektów, a następnie przesyłane do polecenia `Where`, przez które są odpowiednio filtrowane. Druga wersja naszego polecenia jest znacznie mniej wydajna, ponieważ wiele niepotrzebnych danych jest przesyłanych przez sieć do naszego komputera, który dodatkowo musi filtrować wyniki z wszystkich trzech komputerów zdalnych. Znacznie lepszym rozwiązaniem byłaby sytuacja, w której komputery zdalne same filtrują wyniki działania polecenia `Get-EventLog` i wysyłają tylko odpowiednie obiekty, pasujące do podanego wzorca. Ja widać, ta druga wersja to nie jest dobry pomysł.

Przyjrzyjmy się dwóm wersjom innego polecenia:

```
Invoke-Command -computerName Server-R2  
-command { Get-Process -name Notepad } |  
Stop-Process
```

Zobaczmy teraz drugą wersję tego polecenia:

```
Invoke-Command -computerName Server-R2  
-command { Get-Process -name Notepad |  
Stop-Process }
```

Po raz kolejny jedyną różnicą między tymi dwoma poleceniami jest miejsce umieszczenie nawiasów klamrowych. Jednak w tym przypadku pierwsza wersja polecenia po prostu nie będzie działać.

Przyjrzyj się uważnie: przekazujemy polecenie `Get-Process -name Notepad` do komputera zdalnego, który pobiera określony proces, przekształca go do formatu XML i z powrotem wysyła do nas przez sieć. Nasz komputer odbiera dane w formacie XML, deserializuje je z powrotem do postaci obiektu i przekazuje go do polecenia `Stop-Process`. Problem polega na tym, że zdeserializowany obiekt nie zawiera wystarczającej ilości danych pozwalających naszemu komputerowi zorientować się, że ten proces pochodzi z komputera zdalnego. Zamiast tego nasz komputer będzie próbował zatrzymać lokalny proces Notatnika, a przecież zupełnie nie o to nam tutaj chodziło.

Morał tej historii jest taki, aby zawsze wykonywać jak najwięcej przetwarzania danych na komputerze zdalnym. Jedynymi operacjami, jakie w większości przypadków powinieneś wykonywać z wynikami polecenia `Invoke-Command`, jest wyświetlanie ich na ekranie lub zapisywanie w postaci raportów lub plików danych. Druga wersja naszego polecenia jest zgodna z tymi zaleceniami: do komputera zdalnego przesyłane jest polecenie `Get-Process -name Notepad | Stop-Process`, zatem całe wyrażenie — zarówno

Kiedy używamy polecenia `Invoke-Command`, zawsze musimy zwracać szczególną uwagę na polecenia, które występują po nim. Jeżeli widzimy polecenia, których zadaniem jest formatowanie lub eksport danych, wszystko jest w porządku, ponieważ możemy wykonywać takie operacje z wynikami działania poleceń wywoływanych za pomocą polecenia `Invoke-Command`. Jeżeli jednak w wierszu wywołania po poleceniu `Invoke-Command` następują polecenia wykonania określonych akcji — czyli takie, które coś uruchamiają, zatrzymują, ustawiają, zmieniają lub wykonują inne operacje o podobnym charakterze — wówczas musimy dokładnie się zastanowić nad tym, co chcemy osiągnąć. W idealnym scenariuszu wszystkie te działania powinny być wykonywane na komputerze zdalnym, a nie na naszym komputerze lokalnym.

Przykładowo, jeżeli uruchomisz następujące polecenie na swoim komputerze lokalnym, zauważysz, że obiekt `Service-Controller` posiada wiele powiązanych z nim metod:

Name	MemberType	Definition
----	-----	-----
Name	AliasProperty	Name = ServiceName
RequiredServices	AliasProperty	RequiredServices = ServicesDep
Disposed	Event	System.EventHandler Disposed(S
Close	Method	System.Void Close()
Continue	Method	System.Void Continue()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef
Dispose	Method	System.Void Dispose()
Equals	Method	bool Equals(System.Object obj)
ExecuteCommand	Method	System.Void ExecuteCommand(int
GetHashCode	Method	int GetHashCode()
GetLifetimeService	Method	System.Object GetLifetimeServi
GetType	Method	type GetType()
InitializeLifetimeService	Method	System.Object InitializeLifeti
Pause	Method	System.Void Pause()
Refresh	Method	System.Void Refresh()
Start	Method	System.Void Start(), System.Vo
Stop	Method	System.Void Stop()
WaitForStatus	Method	System.Void WaitForStatus(Syst

CanPauseAndContinue	Property	bool CanPauseAndContinue {get;
CanShutdown	Property	bool CanShutdown {get;}
CanStop	Property	bool CanStop {get;}
Container	Property	System.ComponentModel.IContainer
DependentServices	Property	System.ServiceProcess.ServiceC

Teraz spróbujmy pozyskać te same obiekty z komputera zdalnego:

```
PS C:\> Invoke-Command -ScriptBlock { Get-Service } -ComputerName DONJONESE408 | Get-Member
    TypeName: Deserialized.System.ServiceProcess.ServiceController

Name      MemberType Definition
-----
ToString  Method      string ToString(), string ToString(string
                        format, System.I

Name      NoteProperty System.String Name=AeLookupSvc
PSComputerName NoteProperty System.String PSComputerName=DONJONESE408
PSShowComputerName NoteProperty System.Boolean PSShowComputerName=True
RequiredServices NoteProperty Deserialized.System.ServiceProcess
                        .ServiceController[] Req

RunspaceId NoteProperty System.Guid RunspaceId=6dc9e130-f7b2-4db4-
                        8b0d-3863033d7df

CanPauseAndContinue Property System.Boolean {get;set;}
CanShutdown      Property System.Boolean {get;set;}
CanStop          Property System.Boolean {get;set;}
Container        Property {get;set;}
DependentServices Property Deserialized.System.ServiceProcess
                        .ServiceController[] {get

DisplayName      Property System.String {get;set;}
MachineName      Property System.String {get;set;}
ServiceHandle    Property System.String {get;set;}
ServiceName      Property System.String {get;set;}
ServicesDependedOn Property Deserialized.System.ServiceProcess
                        .ServiceController[] {get

ServiceType      Property System.String {get;set;}
Site            Property {get;set;}
Status          Property System.String {get;set;}
```

Jak łatwo zauważyć, praktycznie wszystkie metody obiektów zniknęły — z wyjątkiem uniwersalnej metody `ToString()`, wspólnej dla wszystkich obiektów. Element otrzymany w wyniku działania polecenia zdalnego to przeznaczona tylko do odczytu kopia obiektu, stąd nie możesz jej „poprosić”, aby wykonywała takie operacje jak zatrzymywanie, wstrzymywanie czy wznowianie. Wynika stąd jasno, że wszelkie działania, które chcesz przeprowadzić za pomocą wykonywanego zdalnie polecenia, powinny być zawarte w bloku skryptu wysyłanym do komputera zdalnego; w ten sposób możesz pracować na obiektach, które są „żywe” i nadal posiadają wszystkie swoje metody.

13.6. Ale poczekaj, jest jeszcze coś więcej

We wszystkich poprzednich przykładach używaliśmy połączeń zdalnych tworzonych ad hoc, co oznacza, że w wierszu polecenia podawaliśmy nazwy komputerów. Jeżeli jednak zamierzasz ponownie łączyć się z tym samym komputerem (lub komputerami)

kilka razy w krótkim czasie, możesz utworzyć trwałe połączenia przeznaczone do wielokrotnego użytku. Omawiamy tę technikę w rozdziale 20.

Musimy tutaj jednak zauważyć, że nie każda firma zezwala na uruchamianie komunikacji zdalnej z użyciem powłoki PowerShell — a przynajmniej nie nastąpi to tak szybko. Firmy, które wdrożyły bardzo restrykcyjne zasady zabezpieczeń, mogą na przykład mieć zapory sieciowe działające na wszystkich komputerach klienckich i serwerach, które to zapory blokują połączenia zdalne. Jeżeli Twoja firma do nich należy, powinieneś sprawdzić, czy w ustawieniach zapory sieciowej istnieje wyjątek dotyczący protokołu RDP (ang. *Remote Desktop Protocol*). Z naszych doświadczeń wynika, że jest to dosyć powszechnie tworzony wyjątek, ponieważ administratorzy systemów z oczywistych powodów potrzebują zdalnej łączności z serwerami. Jeżeli połączenia RDP są dozwolone, spróbuj porozmawiać z administratorami o zezwoleniu na komunikację zdalną przy użyciu powłoki PowerShell. Ponieważ połączenia zdalne mogą być kontrolowane (takie połączenia, podobnie jak dostęp do udziałów sieciowych, mogą się pojawiać w dziennikach zdarzeń), są zazwyczaj domyślnie udostępniane tylko administratorom. Takie połączenia pod względem bezpieczeństwa nie różnią się wiele od połączeń RDP, a jednocześnie wnoszą znacznie mniejszy narzut na obciążenie komputerów zdalnych niż połączenia RDP.

13.7. Opcje komunikacji zdalnej

Jeżeli uważnie zapoznasz się z plikami pomocy, z pewnością zauważysz, że zarówno polecenie `Invoke-Command`, jak i `Enter-PSSession` mają parametr `-SessionOption`, który akceptuje obiekt `<PSSessionOption>`. O co w tym wszystkim chodzi?

Jak już wcześniej wspominaliśmy, po uruchomieniu każde z tych poleceń tworzy nowe połączenie powłoki PowerShell lub — inaczej mówiąc — nową *sesję połączenia*, a następnie po wykonaniu zadania automatycznie zamyka taką sesję. Jeżeli chcesz zmodyfikować sposób tworzenia sesji, możesz to zrobić przy użyciu odpowiedniego zestawu opcji, definiowanego za pomocą polecenia `New-PSSessionOption`. Możesz go użyć do określenia między innymi następujących elementów:

- definiowania limitów czasu na otwarcie, anulowania i okresu bezczynności połączenia,
- włączania i wyłączania kompresji i (lub) szyfrowania strumienia danych,
- definiowania różnych opcji związanych z przesyłaniem ruchu przez serwery proxy,
- pomijania certyfikatów SSL komputera zdalnego, nazw i innych mechanizmów bezpieczeństwa.

Przykładowo, oto jak możesz otworzyć sesję zdalną i pominąć sprawdzanie nazwy komputera:

```
PS C:\> Enter-PSSession -ComputerName DONJONESE408 -SessionOption (New-PSSessionOption -SkipCNCHECK)
[DONJONESE408]: PS C:\Users\donjones\Documents>
```

Uważnie zapoznaj się z zawartością plików pomocy polecenia `New-PSSessionOption`, aby przekonać się, do czego jeszcze możesz go używać. W rozdziale 20. zastosujemy kilka z tych opcji do wykonywania różnych zaawansowanych zadań wykorzystujących połączenia zdalne.

13.8. Najczęściej spotykane problemy

Niemal zawsze, kiedy w trakcie naszych szkoleń rozpoczynamy omawianie zagadnień związanych z komunikacją zdalną, pojawiają się niektóre typowe, związane z tym problemy:

- Domyślnie połączenia zdalne działają tylko z rzeczywistymi nazwami komputerów zdalnych, co oznacza, że w takich poleceniach nie możemy używać aliasów DNS ani adresów IP. W rozdziale 23. omawiamy niektóre z przyczyn tego ograniczenia i pokazujemy, jak je obejść.
- Mechanizm tworzenia połączeń zdalnych powłoki PowerShell został zaprojektowany tak, aby takie połączenia mogły być mniej lub bardziej automatycznie konfigurowane w domenie. Jeżeli wszystkie zaangażowane komputery i konta użytkowników należą do tej samej domeny (lub domen powiązanych relacjami zaufania), połączenia między nimi zazwyczaj będą działać świetnie. Jeżeli tak nie jest, powinieneś zapoznać się z zawartością pliku pomocy `about_remote_troubleshooting` i uważnie zagłębić się w opisywane tam szczegóły. Jednym ze scenariuszy, w którym być może będziesz musiał tak zrobić, jest tworzenie połączeń zdalnych między komputerami znajdującymi się w różnych domenach. Aby takie połączenia działały poprawnie, może być konieczna niewielka modyfikacja ustawień konfiguracyjnych, ale na szczęście w pliku pomocy znajdziesz wszystkie niezbędne szczegóły.
- Kiedy wywołujesz polecenie zdalne, nakazujesz komputerowi zdalnemu uruchomienie powłoki PowerShell, wykonanie odpowiedniego polecenia, a następnie zamknięcie powłoki PowerShell. Kolejne polecenie, które zostanie po pewnym czasie uruchomione na tym samym komputerze zdalnym, rozpocznie całą procedurę od zera — nic, co zostało uruchomione w pierwszym wywołaniu, nie będzie już działać. Jeżeli musisz wykonać całą serię powiązanych ze sobą poleceń, powinieneś umieścić je wszystkie w tym samym wywołaniu.
- Upewnij się, że używasz powłoki PowerShell jako administrator systemu, szczególnie jeżeli Twój komputer ma włączony mechanizm kontroli konta użytkownika (UAC — ang. *User Account Control*). Jeżeli konto, z którego korzystasz, nie ma uprawnień administratora na komputerze zdalnym, powinieneś użyć parametru `-credential` polecenia `Enter-PSSession` lub polecenia `Invoke-Command` i określić alternatywne konto użytkownika, które posiada uprawnienia administratora.
- Jeśli używasz lokalnej zapory sieciowej, innej niż wbudowana zaporą systemu Windows, polecenie `Enable-PSRemoting` nie będzie w stanie automatycznie skonfigurować wymaganych wyjątków zapory. W takiej sytuacji musisz to zrobić ręcznie. Jeżeli połączenie zdalne będzie musiało po drodze przejść przez inną zaporę

sieciową, na przykład zaimplementowaną na routerze lub serwerze proxy, to również najprawdopodobniej nie obejdziesz się bez ręcznego utworzenia odpowiedniego wyjątku dla takiego połączenia.

- Nie zapominać, że wszelkie ustawienia lokalne mogą być odgórnie zastępowane przez odpowiednie zasady grupy. Nieraz na własne oczy widzieliśmy, jak mniej rozgarnięci administratorzy przez wiele godzin zmagają się z konfiguracją systemu, usiłując włączyć połączenia zdalne tylko po to, aby w końcu odkryć, że ich ustawienia zostały nadpisane przez GPO. Zdarzało się i tak, że takie czy inne ustawienia GPO zostały utworzone dawno temu i wszyscy już o nich zapomnieli. Nigdy nie zakładaj, że ustawienia GPO Ciebie nie dotyczą; zawsze najpierw sprawdź, czy tak jest na pewno.

13.9. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej. W idealnej sytuacji powinieneś mieć do dyspozycji co najmniej dwa takie komputery, które należą do tej samej domeny Active Directory, ale jeżeli posiadasz tylko jeden komputer, to też nie będzie problemu.

Nadszedł czas, aby połączyć to, czego nauczyłeś się w tym rozdziale na temat połączeń zdalnych, z wiedzą i umiejętnościami, które nabyłeś w poprzednich rozdziałach. Spróbuj samodzielnie wykonać następujące zadania:

1. Nawiąż połączenie typu jeden-do-jednego z komputerem zdalnym (jeżeli masz do dyspozycji tylko jeden komputer, w roli komputera zdalnego możesz użyć adresu `localhost`). Spróbuj zdalnie uruchomić program `Notepad.exe`. Co się wydarzyło?
2. Za pomocą polecenia `Invoke-Command` pobierz z jednego lub dwóch komputerów zdalnych listę usług, które nie zostały uruchomione (jeżeli masz tylko jeden komputer, możesz dwa razy użyć adresu `localhost`). Sformatuj wyniki działania polecenia jako szeroką listę. (Wskazówka: możesz najpierw pobierać wyniki i formatować je na swoim komputerze lokalnym — nie powinieneś umieszczać poleceń z rodziny `Format-` w komendach wywoływanych zdalnie).
3. Użyj polecenia `Invoke-Command`, aby uzyskać listę 10 procesów, które zużywają najwięcej zasobów pamięci wirtualnej (VM). Jeżeli masz taką możliwość, pobierz dane z jednego lub dwóch komputerów zdalnych; jeżeli dysponujesz tylko jednym komputerem, dwukrotnie użyj hosta `localhost`.
4. Utwórz plik tekstowy zawierający nazwy trzech komputerów, po jednej nazwie w każdym wierszu. Jeżeli masz dostęp tylko do jednego komputera, możesz trzykrotnie użyć tej samej nazwy komputera bądź nawet trzy razy użyć adresu `localhost`. Następnie zastosuj polecenie `Invoke-Command`, aby pobrać 100 nowych wpisów dziennika zdarzeń aplikacji z komputerów, których nazwy zostały wymienione w tym pliku.

5. Za pomocą polecenia `Invoke-Command` spróbuj wyświetlić wartości właściwości `ProductName`, `EditionID` oraz `CurrentVersion` z klucza rejestru `HKEY_Local_Machine\SOFTWARE\Microsoft\Windows NT\CurrentVersion\` jednego lub kilku komputerów zdalnych. (Wskazówka: wykonanie zadania będzie wymagało pobrania właściwości wybranego elementu).

13.10. Co dalej?

Oczywiście moglibyśmy napisać dużo więcej na temat tworzenia połączeń zdalnych z użyciem powłoki PowerShell — wystarczy wspomnieć, że w *kolejnym* rozdziale nadal będziemy zajmować się tą tematyką. Niestety niektóre z trudniejszych zagadnień związanych z komunikacją zdalną nie są dobrze udokumentowane, dlatego gorąco zachęcamy Cię do odwiedzenia witryny *PowerShell.org*, a dokładniej sekcji e-booków, gdzie Don wraz z dr. Tobiasem Weltnerem (MVP) przygotowali dla Ciebie obszerną (i darmową!) publikację zatytułowaną *Secrets of PowerShell Remoting*. Znajdziesz tam trochę podstawowych informacji zawartych również w tym rozdziale, ale przewodnik ten skupia się przede wszystkim na bardzo szczegółowych instrukcjach krok po kroku (łącznie z kolorowymi zrzutami ekranu), które pokazują, jak skonfigurować różne scenariusze zdalnego działania. Przewodnik omawia również niektóre bardziej szczegółowe informacje na temat protokołów komunikacyjnych i rozwiązywania problemów, a nawet zawiera krótką sekcję o tym, jak zapytać administratora systemu o możliwość włączenia zdalnego dostępu. Przewodnik jest okresowo aktualizowany, więc sprawdzaj go co kilka miesięcy, aby upewnić się, że masz jego najnowszą wersję. Nie zapominaj również, że na forach witryny *PowerShell.org* możesz dotrzeć ze swoimi pytaniami bezpośrednio do Dona.

13.11. Odpowiedzi

1. `Enter-PSSession Server01`
`[Server01] PS C:\Users\Administrator\Documents> Notepad`
 Proces Notatnika rozpocznie działanie, nie będziesz miał z nim żadnej możliwości interakcji (ani lokalnie, ani zdalnie). W rzeczywistości po uruchomieniu w ten sposób Notatnika znak zachęty powłoki PowerShell nie pojawi się ponownie dopóty, dopóki proces Notatnika nie zakończy działania; alternatywnym poleceniem pozwalającym na uruchomienie Notatnika jest `Start-Process Notepad`.
2. `Invoke-Command -scriptblock {get-service | where {$_.status -eq "stopped"}} -computername Server01,Server02 | format-wide -Column 4`
3. `Invoke-Command -scriptblock {get-process | sort VM -Descending | Select-first 10} -computername Server01,Server02`
4. `Invoke-Command -scriptblock {get-eventlog -LogName Application -Newest 100} -ComputerName (Get-Content computers.txt)`
5. `invoke-command -scriptblock{get-itemproperty 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion\' | Select ProductName,EditionID,CurrentVersion} -computername Server01,Server02`

14

Zastosowanie mechanizmu WMI oraz standardu CIM

Z niecierpliwością oczekiwaliśmy dobrej okazji do napisania tego rozdziału i jednocześnie trochę się tego obawialiśmy. Mechanizm Windows Management Instrumentation (WMI) jest prawdopodobnie jedną z najlepszych rzeczy, jakie Microsoft zaoferował administratorom. Z drugiej strony to także jedna z najgorszych rzeczy, którą Microsoft mógł nam zrobić. Mechanizm WMI umożliwia łatwe zbieranie niesamowitych ilości informacji systemowych z komputera, ale również czasami jest bardzo tajemniczy i nieprzenikniony, a jego dokumentacja z pewnością nie jest przyjazna dla użytkownika. W tym rozdziale pokażemy, w jaki sposób możesz korzystać z mechanizmu WMI z poziomu powłoki PowerShell, objaśnimy, jak to działa, i wskażemy niektóre z jego mniej pięknych aspektów, tak abyś miał pełny obraz tego, z czym masz do czynienia.

Chcemy podkreślić, że WMI jest technologią zewnętrzną; powłoka PowerShell jedynie z nią współpracuje. W niniejszym rozdziale skupimy się na tym, jak to robi powłoka PowerShell, a nie będziemy omawiać wewnętrznych tajemnic samego mechanizmu WMI. Jeżeli chcesz dalej badać WMI, na końcu tego rozdziału znajdziesz kilka naszych sugestii. Pamiętaj, że powłoka PowerShell v3 poczyniła niesamowite postępy w minimalizowaniu ilości pracy, którą musisz włożyć w to, aby korzystać z WMI.

Należy również podkreślić, że WMI to narzędzie do zarządzania systemem Windows; jest to mechanizm specyficzny dla tego systemu i nie istnieje w systemach takich jak Linux czy macOS. Co prawda w niektórych wersjach Linuksa jest dostępne nieco podobne rozwiązanie o nazwie OMI (ang. *Open Management Instrumentation*), ale w chwili kiedy powstawała ta książka, powłoka PowerShell dla systemów Linux i macOS jeszcze nie współdziałała z OMI.

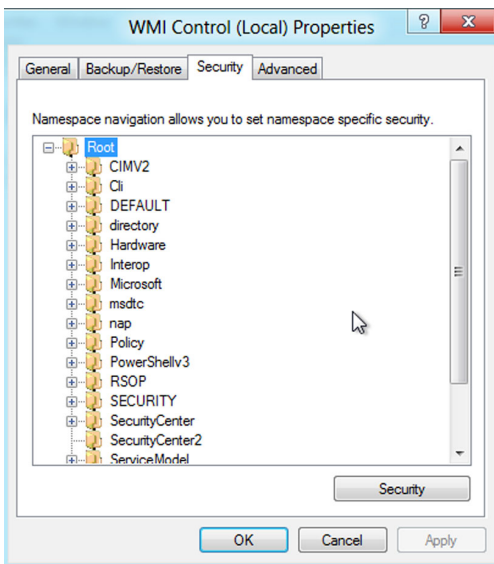
14.1. Podstawowe elementy WMI

W typowym komputerze działającym pod kontrolą systemu Windows możemy znaleźć dziesiątki tysięcy informacji związanych z zarządzaniem, a mechanizm WMI dąży do zorganizowania tego pozornego chaosu w coś, co jest bardziej przystępne dla użytkownika.

Na najwyższym poziomie hierarchii WMI jest podzielony na tzw. **przestrzeń nazw** (ang. *namespaces*). Przestrzeń nazw możesz sobie wyobrazić jako swego rodzaju folder powiązany z konkretnym produktem lub technologią. Na przykład przestrzeń nazw `root\CIMv2` obejmuje cały system operacyjny Windows oraz informacje o sprzęcie komputerowym, a przestrzeń nazw `root\MicrosoftDNS` zawiera wszystkie informacje o serwerze DNS (przy założeniu, że zainstalowałeś tę rolę w swoim systemie). W komputerach klienckich przestrzeń nazw `root\SecurityCenter` zawiera informacje na temat zapór sieciowych, programów antywirusowych i programów antyszpiegowskich.

UWAGA Zawartość przestrzeni nazw `root\SecurityCenter` różni się w zależności od tego, co jest zainstalowane w danym komputerze. Co więcej, nowsze wersje systemu Windows używają przestrzeni `root\SecurityCenter2`, co jest dobrym przykładem tego, jak mylący może być mechanizm WMI.

Na rysunku 14.1 pokazano niektóre przestrzenie nazw dostępne w naszym komputerze. Informację o tym wygenerowaliśmy za pomocą przystawki *WMI Control* konsoli MMC (ang. *Microsoft Management Console*).



Rysunek 14.1. Przeglądanie przestrzeni nazw WMI

W obrębie przestrzeni nazw WMI dzieli się na serię klas. *Klasa* reprezentuje komponent zarządzania, który WMI potrafi odpytać. Przykładowo, klasa `AntivirusProduct` z przestrzeni `root\SecurityCenter` została zaprojektowana w celu przechowywania informacji

o produktach antyspyware, a klasa `Win32_LogicalDisk` w przestrzeni `root\CIMv2` jest przeznaczona do przechowywania informacji o dyskach logicznych. Nawet jednak jeśli dana klasa istnieje w komputerze, nie oznacza to jeszcze, że musi on posiadać taki komponent; na przykład klasa `Win32_TapeDrive` jest obecna we wszystkich wersjach systemu Windows, niezależnie od tego, czy napęd taśmowy jest zainstalowany, czy nie.

UWAGA Jeszcze raz przypominamy, że nie każdy komputer zawiera te same przestrzenie nazw lub klasy WMI. Na przykład komputery z nowszym systemem Windows oprócz przestrzeni nazw `root\SecurityCenter` mają również przestrzeń nazw `root\SecurityCenter2`, gdzie znajdują się wszystkie użyteczne informacje o produktach związanych z bezpieczeństwem systemu.

Poniżej przedstawiamy prosty przykład zapytania klasy `AntiSpywareProduct` z przestrzeni nazw `root\SecurityCenter2`; uważnie przyjrzyj się wynikom działania tego polecenia:

```
PS C:\> Get-CimInstance -Namespace root\securitycenter2 -ClassName antispywareproduct
```

UWAGA Aby wykonać powyższe polecenie, musisz korzystać z powłoki PowerShell v3 lub nowszej; o poleceniu `Get-CimInstance` dowiesz się czegoś więcej już za chwilę.

Jeżeli masz jeden lub więcej zarządzalnych komponentów, będziesz mieć odpowiadającą im liczbę instancji danej klasy. **Instancja** to rzeczywiste wystąpienie czegoś reprezentowanego przez daną klasę. Jeżeli Twój komputer ma jeden BIOS (jak wszystkie komputery), będziesz mieć jedną instancję klasy `Win32_BIOS` w przestrzeni nazw `root\CIMv2`; jeżeli Twój komputer ma zainstalowane 100 usług działających w tle, będziesz miał 100 instancji klasy `Win32_Service`. Zauważ, że nazwy klas w przestrzeni `root\CIMv2` zwykle zaczynają się od ciągu znaków `Win32_` (nawet na maszynach 64-bitowych) lub `CIM_` (ang. *Common Information Model*; standard, na bazie którego zbudowany jest WMI). W innych przestrzeniach nazw przedrostki nazw klas nie są zwykle używane. W różnych przestrzeniach nazw mogą się znajdować klasy o takich samych nazwach. Jest to co prawda rzadko spotykane, ale WMI na to pozwala, ponieważ każda przestrzeń nazw działa jako rodzaj osobnego kontenera. Kiedy odwołujesz się do danej klasy, musisz używać odpowiedniej nazwy przestrzeni nazw, aby WMI wiedział, gdzie szukać danej klasy — dzięki temu nie będą mylone dwie klasy o takich samych nazwach, ale znajdujące się w różnych przestrzeniach nazw.

Wszystkie te instancje, klasy i inne komponenty egzystują w czymś, co nosi nazwę **repozytorium WMI** (ang. *WMI repository*). W starszych wersjach systemu Windows zdarzało się, że repozytorium mogło czasami zostać uszkodzone i nie nadawało się do użytku, a użytkownik musiał je przebudować; na szczęście od czasu pojawienia się systemu Windows 7 nie jest to zjawisko zbyt częste.

Na pierwszy rzut oka korzystanie z mechanizmu WMI wydaje się dość proste: użytkownik sprawdza, która klasa WMI zawiera żądane informacje, wysyła zapytanie do instancji tej klasy, a następnie wyświetla właściwości otrzymanych obiektów klasy.

W niektórych przypadkach możesz uruchamiać wybrane metody danej instancji, pozwalające na wykonanie określonych akcji lub dokonanie zmian w konfiguracji.

14.2. Złe wieści na temat usługi WMI

Niestety przez większość czasu (ostatnio sytuacja nieco się zmieniła) firma Microsoft utrzymywała mechanizm WMI pod ścisłą kontrolą. Microsoft ustanowił zestaw standardów programowania, ale poszczególne grupy produktów zostały mniej lub bardziej pozostawione własnemu losowi, w wyniku czego pozwolono różnym producentom na określanie sposobu wdrażania nowych klas i ich dokumentowania, tak więc mechanizm WMI może być dla użytkownika nieco zagmatwany i dezorientujący.

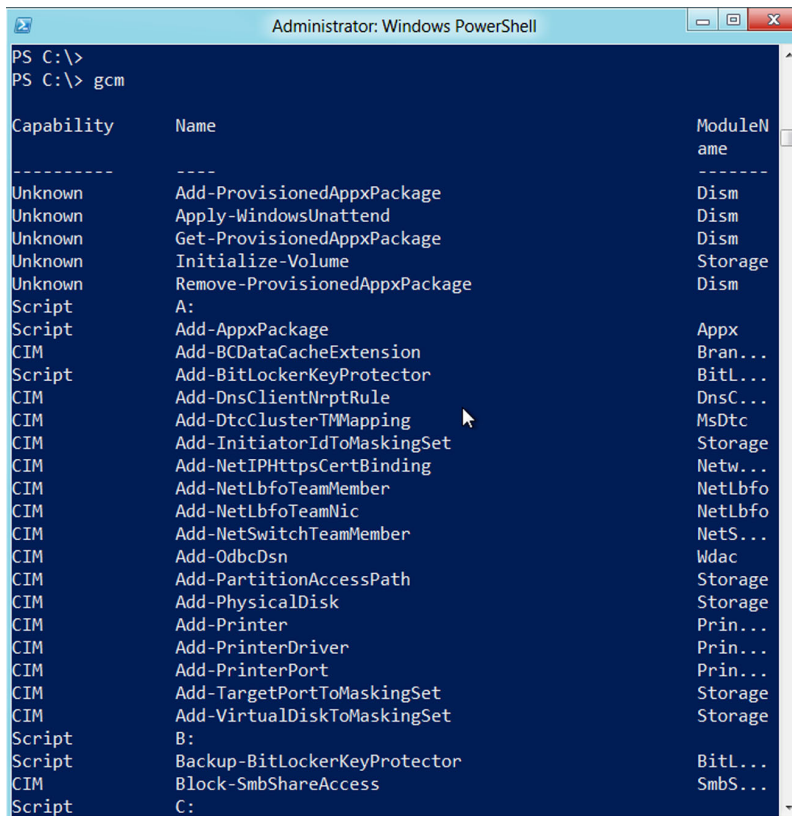
Przykładowo, w przestrzeni nazw `root\CIMv2` tylko kilka klas posiada jakieś metody umożliwiające zmianę ustawień konfiguracyjnych. Właściwości obiektów możemy bezpośrednio tylko odczytywać, co oznacza, że do wprowadzania zmian musimy mieć odpowiednią metodę; jeżeli taka metoda nie istnieje, do wprowadzania zmian nie możemy użyć mechanizmu WMI. Kiedy zespół serwera IIS zaimplementował mechanizm WMI (dla IIS wersji 6), jego deweloperzy wdrożyli równoległe klasy dla wielu elementów. Podajmy przykład — strona internetowa może być reprezentowana przez jedną klasę, która ma typowe właściwości przeznaczone tylko do odczytu, ale także przez drugą klasę, posiadającą właściwości do odczytu i zapisu, których wartości można zmienić, co dla użytkownika może być bardzo mylące. Całe zamieszanie pogłębia się jeszcze z powodu braku dobrej dokumentacji na temat korzystania z tych klas, ponieważ zgodnie z początkowymi zamierzeniami zespołu deweloperów serwera IIS takie klasy miały być używane przez ich własne narzędzia, a nie bezpośrednio przez administratorów. Po pewnym czasie jednak deweloperzy serwera IIS całkowicie wycofali się z używania WMI jako interfejsu zarządzania, a począwszy od wersji v7.5 skupili się na cmdletach powłoki PowerShell i dostawcach PSProvider.

Firma Microsoft nie ma reguły, która mówi, że każdy produkt *musi* korzystać z mechanizmu WMI ani że jeśli korzysta z usługi WMI, musi udostępniać każdy możliwy komponent za pośrednictwem WMI. Przykładowo, serwer DHCP firmy Microsoft jest całkowicie niedostępny z poziomu WMI, podobnie było ze starym serwerem WINS. Choć w praktyce możemy zapytać na przykład o konfigurację karty sieciowej, to nie otrzymamy już informacji o prędkości łącza, ponieważ takie dane nie są dostarczane. Chociaż większość klas `Win32_` jest dobrze udokumentowana, to już w przypadku klas znajdujących się w innych przestrzeniach nazw jest jak na lekarstwo dokumentacji. Repozytorium WMI nie można przeszukiwać, więc proces wyszukiwania potrzebnej klasy może być czasochłonny i frustrujący (choć w następnej sekcji spróbujemy Ci nieco w tym pomóc).

Dobrą wiadomością jest to, że firma Microsoft stara się dostarczyć odpowiednie cmdlety powłoki PowerShell dla jak największej liczby zadań związanych z zarządzaniem. Jeszcze nie tak dawno temu mechanizm WMI był jedynym praktycznym sposobem umożliwiającym przeprowadzenie programowego restartu komputera zdalnego; można było tego dokonać przy użyciu odpowiedniej metody klasy `Win32_OperatingSystem`.

Obecnie powłoka PowerShell udostępnia polecenie `Restart-Computer`, pozwalające na wykonanie takiego zadania. W niektórych przypadkach polecenia *cmdlet* wewnętrznie korzystają z usług WMI, ale w takich sytuacjach użytkownik nie ma żadnej potrzeby bezpośredniej interakcji z WMI. Polecenia *cmdlet* zapewniają bardziej spójny interfejs dla użytkownika i są prawie zawsze znacznie lepiej udokumentowane. Oczywiście mechanizm WMI jeszcze nie zamierza zniknąć, ale z czasem prawdopodobnie będziesz coraz rzadziej musiał z niego korzystać.

W rzeczywistości poczynawszy od powłoki PowerShell w wersji v3 (szczególnie w najnowszych wersjach systemu Windows, takich jak Windows 8, Windows Server 2012 i nowszych) zauważysz wiele poleceń CIM, tak jak to pokazano na rysunku 14.2, na którym zamieszczono fragment wyników działania polecenia `Get-Command`. W większości przypadków są to swego rodzaju „polecenia opakowujące” dla odpowiednich klas WMI, dzięki którym możesz pracować z mechanizmem WMI w bardziej „powershellowy” sposób. Takich *cmdlet*ów używa się tak, jak każdego innego polecenia, łącznie z wyświetlaniem dla nich zawartości plików pomocy, co czyni je bardziej spójnymi z resztą poleceń powłoki PowerShell i pomaga ukryć niektóre z najbardziej irytujących ekscentryzmów mechanizmu WMI.



Rysunek 14.2. Polecenia CIM są „opakowaniami” dla odpowiednich klas WMI

14.3. Eksplorowanie WMI

Być może najłatwiejszym sposobem rozpoczęcia korzystania z WMI jest odstawienie na bok powłoki PowerShell i samodzielna eksploracja mechanizmu WMI. Użyjemy do tego celu darmowego narzędzia o nazwie WMI Explorer. Niestety często narzędzia takie pojawiają się i przemijają jak pory roku, więc nie chcemy wskazywać żadnego z nich w szczególności. Po prostu uruchom swoją ulubioną wyszukiwarkę sieciową, w polu wyszukiwania wpisz `WMI Explorer` i zobacz, co się pojawi. Możesz również zajrzeć na stronę <https://powershell.org/2013/03/08/wmi-explorer/>. Korzystając z takich narzędzi, możemy łatwo znaleźć większość tego, czego potrzebujemy z WMI. Wymaga to jednak dużej ilości przeglądania i sporej dozy cierpliwości — nie będziemy więc udawać, że jest to doskonały proces, ale w ostatecznym rozrachunku pozwala osiągnąć zamierzony cel.

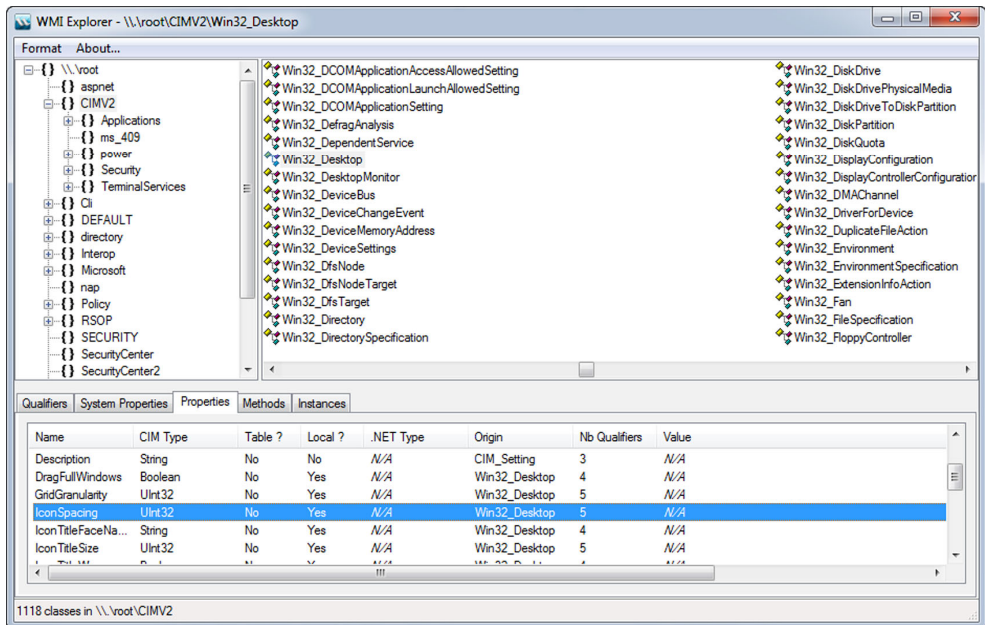
Ponieważ w poszczególnych komputerach mogą występować różne przestrzenie nazw i klasy WMI, powinieneś uruchomić to narzędzie bezpośrednio na tym komputerze, do którego chcesz wysłać zapytanie, tak aby można było sprawdzić zawartość repozytorium WMI tego komputera.

Załóżmy, że musimy przesłać zapytanie do kilku komputerów klienckich, aby sprawdzić, jak są ustawione domyślne odstępy między ikonami. Jest to zadanie związane z pulpitem systemu Windows, który jest jednym z podstawowych części systemu operacyjnego, więc zaczynamy od klasy `root\CIMV2`, pokazanej w widoku drzewa z lewej strony okna naszego programu WMI Explorer (patrz rysunek 14.3). Kliknięcie wybranej przestrzeni nazw powoduje wyświetlenie listy jej klas w panelu w prawej części okna. Możemy śmiało założyć, że wyraz *Desktop* jest tutaj słowem kluczowym. Przewijając listę dostępnych klas, w końcu odnajdujemy klasę `Win32_Desktop` i klikamy jej nazwę. W ten sposób uaktywniamy okienko szczegółów w dolnej części okna i przechodzimy na kartę *Properties* (właściwości), aby zobaczyć, co jest dostępne. Po krótkim sprawdzeniu odnajdujemy właściwość *IconSpacing*, której wartość jest wyrażana w postaci liczby całkowitej.

Oczywiście użycie wyszukiwarki sieciowej to kolejny dobry sposób na znalezienie klasy WMI, która Cię interesuje. Tworząc takie zapytania, zazwyczaj poprzedzamy je prefiksem *wmi*, na przykład *wmi icon spacing*, co bardzo często daje dobre rezultaty i wskazuje nam właściwy kierunek postępowania. Znalezione przykłady często są powiązane z takimi językami jak VBScript, .NET, a nawet C# czy Visual Basic, ale to jest w porządku, ponieważ jesteśmy zainteresowani nazwą odpowiedniej klasy WMI. Przykładowo, szukając klasy WMI zawierającej informacje o odstępach między ikonami, wpisaliśmy w wyszukiwarce sieciowej frazę *wmi icon spacing* i pierwszym wynikiem wyszukiwania była strona <https://stackoverflow.com/questions/202971/-formula-or-api-for-calulating-desktop-icon-spacing-on-windows-xp>, na której znaleźliśmy następujący kod w języku C#:

```
ManagementObjectSearcher searcher = new ManagementObjectSearcher("root\\CIMV2", "SELECT * FROM Win32_Desktop");
```

Nie mamy zielonego pojęcia, co to oznacza, ale `Win32_Desktop` wygląda jak poprawna nazwa klasy WMI. Nasze następne wyszukiwanie dotyczyło oczywiście tej nazwy klasy,



Rysunek 14.3. WMI Explorer

ponieważ bardzo często możemy się w ten sposób przekonać, czy taka klasa posiada odpowiednią dokumentację (o dokumentacjach klas WMI będziemy mówić w nieco dalszej części tego rozdziału).

Innym podejściem jest użycie samej powłoki PowerShell. Załóżmy, że chcemy wykonać jakąś operację związaną z dyskami. Zaczniemy od znalezienia odpowiedniej przestrzeni nazw. Wiemy jednak, że `root\CIMv2` zawiera klasy reprezentujące wszystkie podstawowe elementy systemu operacyjnego i sprzętu, zatem uruchamiamy następujące polecenie:

```
PS C:\> Get-WmiObject -Namespace root\CIMv2 -list | where name -like '*dis*'
```

```
NameSpace: ROOT\CIMv2
```

Name	Methods	Properties
----	-----	-----
CIM_LogicalDisk	{SetPowerState, R...	{Access, Avail...
Win32_LogicalDisk	{SetPowerState, R...	{Access, Avail...
Win32_MappedLogicalDisk	{SetPowerState, R...	{Access, Avail...
CIM_DiskPartition	{SetPowerState, R...	{Access, Avail...
Win32_DiskPartition	{SetPowerState, R...	{Access, Avail...
CIM_DiskDrive	{SetPowerState, R...	{Availability, ...
Win32_DiskDrive	{SetPowerState, R...	{Availability, ...
CIM_DisketteDrive	{SetPowerState, R...	{Availability, ...
CIM_DiskSpaceCheck	{Invoke}	{AvailableDisk...
Win32_LogicalDiskRootDirectory	{}	{GroupComponent...
Win32_DiskQuota	{}	{DiskSpaceUsed...
Win32_LogonSessionMappedDisk	{}	{Antecedent, D...
CIM_LogicalDiskBasedOnPartition	{}	{Antecedent, D...
Win32_LogicalDiskToPartition	{}	{Antecedent, D...
CIM_LogicalDiskBasedOnVolumeSet	{}	{Antecedent, D...

Win32_DiskDrivePhysicalMedia	{}	{Antecedent, D...
CIM_RealizesDiskPartition	{}	{Antecedent, D...
Win32_DiskDriveToDiskPartition	{}	{Antecedent, D...
Win32_OfflineFilesDiskSpaceLimit	{}	{AutoCacheSize...
Win32_PerfFormattedData_Counters...	{}	{Caption, Desc...
Win32_PerfRawData_Counters_FileS...	{}	{Caption, Desc...
Win32_PerfFormattedData_PerfDisk...	{}	{AvgDiskBytesP...

W wynikach działania tego polecenia znajdujemy klasę o nazwie Win32_LogicalDisk.

UWAGA Klasy, których nazwy zaczynają się od ciągu znaków CIM_, są często *klasami bazowymi* i nie używamy ich bezpośrednio. Klasy z prefiksem Win32_ są specyficzne dla systemu Windows. Zwróć uwagę, że prefiks Win32_ jest używany tylko w tej konkretnej przestrzeni nazw — w innych przestrzeniach nazw takie prefiksy nie są zwykle używane.

14.4. Wybierz swoją broń: WMI lub CIM

W powłoce PowerShell w wersji 3 i nowszych masz dwa sposoby interakcji z WMI:

- *Cmdlety WMI*, takie jak `Get-WmiObject` i `Invoke-WmiMethod` — są historycznie starszymi poleceniami, co oznacza, że nadal działają, ale firma Microsoft przestała już inwestować w ich dalszy rozwój. Polecenia WMI komunikują się przez zdalne wywołania procedur (RPC), które mogą przechodzić przez zaporę sieciową tylko wtedy, gdy jest odpowiednio skonfigurowana i obsługuje pełnostanową inspekcję pakietów (innymi słowy, jest to trudne).
- *Cmdlety CIM*, takie jak `Get-CimInstance` i `Invoke-CimMethod` — to nowe wersje poleceń powłoki PowerShell, które są mniej więcej równoważne starym cmdletom WMI, mimo że zamiast RPC komunikują się za pomocą protokołu WS-MAN (implementowanego przez usługę Windows Remote Management lub WinRM). Tą drogą obecnie podąża Microsoft, a wykonanie polecenia `Get-Command -noun CIM*` jasno pokazuje, że takich cmdletów jest coraz więcej i oferują coraz więcej funkcjonalności.

Nie powinieneś jednak wyciągać z tego mylnych wniosków: wszystkie te polecenia używają tego samego zaplecza WMI (*repozytorium WMI*). Główna różnica polega na tym, w jaki sposób się komunikują i jak z nich korzystasz. W starszych systemach, które nie mają zainstalowanej powłoki PowerShell lub które nie mają włączonej usługi WinRM, cmdlety WMI często nadal działają (technologia ta została wprowadzona już w systemie Windows NT 4.0 SP3 i nowszych). W przypadku nowszych systemów z zainstalowaną powłoką PowerShell i włączoną usługą WinRM cmdlety CIM zapewniają najlepszą jakość obsługi, a firma Microsoft wciąż będzie zwiększała ich możliwości funkcjonalne i poprawiała wydajność. W rzeczywistości w systemach Windows Server 2012 R2 i nowszych „stare” cmdlety WMI są domyślnie wyłączone, więc powinieneś się trzymać wersji CIM, zawsze kiedy tylko możesz. Ponadto cmdlety CIM mogą komunikować się również przy użyciu starszego protokołu RPC (lub DCOM), a zatem można ich używać, nawet jeżeli pracujemy ze starszymi maszynami.

14.5. Używanie polecenia *Get-WmiObject*

Za pomocą polecenia *Get-WmiObject* można określić przestrzeń nazw, nazwę klasy, a nawet nazwę komputera zdalnego, a także — w razie potrzeby — alternatywne poświadczenia logowania, tak aby pobrać wszystkie wystąpienia tej klasy z określonego komputera.

Jeżeli chcesz uzyskać tylko określone wystąpienia danej klasy, w wywołaniu polecenia możesz podawać odpowiednie kryteria filtrowania. Polecenie pozwala również na wyświetlenie listy klas z wybranej przestrzeni nazw. Aby to zrobić, powinieneś użyć następującej składni:

```
Get-WmiObject -namespace root\cimv2 -list
```

Zwróć uwagę, że przy podawaniu nazwy przestrzeni nazw używamy lewego, a nie prawego ukośnika.

Aby pobrać wybraną klasę, należy podać nazwę przestrzeni nazw i nazwę klasy, na przykład:

```
Get-WmiObject -namespace root\cimv2 -class win32_desktop
```

Przestrzeń nazw *root\CIMv2* jest domyślną przestrzenią nazw w systemach Windows XP z dodatkiem Service Pack 2 i nowszych, więc jeżeli dana klasa znajduje się w tej przestrzeni nazw, nie trzeba jej dodatkowo określać. Ponadto parametr *-class* jest pozycjonowany, zatem jeżeli podasz nazwę klasy na pierwszej pozycji, to całe polecenie zadziała tak samo.

Oto dwa przykłady; w drugim z nich zamiast pełnej nazwy polecenia używamy jego aliasu *gwmi*:

```
PS C:\> Get-WmiObject win32_desktop
PS C:\> gwmi antispywareproduct -namespace root\securitycenter2
```

ZRÓB TO SAM Od tej chwili powinieneś próbować samodzielnie wykonywać wszystkie polecenia, które omawiamy w tym rozdziale. W przypadku poleceń, które zawierają nazwę komputera zdalnego, możesz ją zastąpić adresem *localhost* (o ile nie masz innego komputera zdalnego, który możesz wykorzystać do testowania).

W przypadku wielu klas WMI powłoka PowerShell ma domyślne ustawienia określające, które właściwości będą wyświetlane. Klasa *Win32_OperatingSystem* jest tutaj dobrym przykładem, ponieważ domyślnie wyświetlanych jest tylko sześć jej właściwości. Pamiętaj, że zawsze możesz przekazać obiekty WMI do polecenia *Gm* lub *Format-List **, tak aby wyświetlić wszystkie dostępne właściwości; polecenie *Gm* poda Ci również listę dostępnych metod. Oto przykład:

```
PS C:\> Get-WmiObject win32_operatingsystem | gm
      TypeName: System.Management.ManagementObject#root\cimv2\Win32_OperatingSystem
Name      MemberType Definition
-----
Reboot     Method      System.Managemen...
SetDateTime Method      System.Managemen...
Shutdown   Method      System.Managemen...
```

Win32Shutdown	Method	System.Managemen...
Win32ShutdownTracker	Method	System.Managemen...
BootDevice	Property	System.String Bo...
BuildNumber	Property	System.String Bu...
BuildType	Property	System.String Bu...
Caption	Property	System.String Ca...
CodeSet	Property	System.String Co...
CountryCode	Property	System.String Co...
CreationClassName	Property	System.String Cr...

Prezentujemy tutaj tylko fragment wyników działania tego polecenia, pełne wyniki zobaczysz po jego uruchomieniu na swoim komputerze.

Parametr `-filter` pozwala ustalić kryteria pobierania określonych instancji. Może być nieco trudny w użyciu, zatem przedstawiamy trochę bardziej złożony przykład jego zastosowania:

```
PS C:\> gwmi -class win32_desktop -filter "name='COMPANY\\Administrator'"
```

```
__GENUS           : 2
__CLASS           : Win32_Desktop
__SUPERCLASS      : CIM_Setting
__DYNASTY         : CIM_Setting
__RELPATH         : Win32_Desktop.Name="COMPANY\\Administrator"
__PROPERTY_COUNT  : 21
__DERIVATION      : {CIM_Setting}
__SERVER          : SERVER-R2
__NAMESPACE      : root\cimv2
__PATH            : \\SERVER-R2\root\cimv2:Win32_Desktop.Name="COMPANY\\Administrator"
BorderWidth       : 1
Caption           :
CoolSwitch        :
CursorBlinkRate   : 530
Description       :
DragFullWindows   : False
GridGranularity   :
IconSpacing       : 43
IconTitleFaceName : Tahoma
IconTitleSize     : 8
IconTitleWrap     : True
Name              : COMPANY\\Administrator
Pattern           : 0
ScreenSaverActive  : False
ScreenSaverExecutable :
ScreenSaverSecure  :
ScreenSaverTimeout :
SettingID         :
Wallpaper          :
WallpaperStretched : True
WallpaperTiled    : False
```

W wywołaniu polecenia oraz wynikach jego działania powinieneś zwrócić uwagę na kilka szczegółów:

- Kryteria filtrowania są zwykle ujęte w podwójny cudzysłów.
- Operatory porównania filtrów nie są standardowymi operatorami powłoki PowerShell, takimi jak `-eq` czy `-like`. Zamiast tego WMI wykorzystuje bardziej tradycyjne operatory, podobne do tych stosowanych w językach programowania, takie jak `=`, `>`, `<`, `<=`, `>=` i `<>`. Możesz również użyć słowa kluczowego `LIKE` jako operatora, stosując jednocześnie znak `%` jako symbol wieloznaczny, na przykład `"NAME LIKE '%administrator%'"`. Zauważ, że nie możesz używać znaku `*` jako symbolu wieloznacznego, tak jak to robiliśmy w innych poleceniach powłoki PowerShell.
- Ciągi znaków używane w porównaniach są ujęte w apostrofy, dlatego całe wyrażenie filtra musi być ujęte w cudzysłów.
- Lewe ukośniki są traktowane przez WMI jako znaki ucieczki, więc jeżeli musisz użyć samego znaku lewego ukośnika (jak w przykładzie powyżej), musisz zastosować dwa lewe ukośniki (pierwszy reprezentuje znak ucieczki, a drugi — właściwy znak lewego ukośnika).
- Wyniki działania polecenia `Gwmi` zawsze zawierają wiele właściwości systemowych. W domyślnej konfiguracji wyświetlania powłoki PowerShell są często pomijane, ale można je zobaczyć, jeżeli wyświetlimy pełną listę właściwości lub jeżeli dana klasa nie ma zdefiniowanej domyślnej konfiguracji wyświetlania. Nazwy właściwości systemowych zaczynają się od dwóch znaków podkreślenia. Oto dwie szczególnie przydatne właściwości z tej kategorii:
 - `__SERVER` zawiera nazwę komputera, z którego została pobrana instancja. Może to być bardzo przydatne podczas pobierania danych WMI z wielu komputerów jednocześnie. Wartość tej właściwości jest powielana w łatwiejszej do zapamiętania właściwości `PSComputerName`.
 - `__PATH` jest bezwzględny odwołaniem do samej instancji klasy i może służyć do ponownego wykonania zapytania.

Polecenie może pobierać dane nie tylko z jednego, ale także z wielu komputerów zdalnych, których listę możemy podawać przy użyciu dowolnej techniki tworzącej kolekcję ciągów znaków zawierających nazwy komputerów lub adresy IP. Oto przykład:

```
PS C:\> Gwmi Win32_BIOS -comp server-r2,server3,dc4
```

Połączenia z poszczególnymi komputerami są nawiązywane po kolei, a jeżeli któryś z komputerów nie jest dostępny, polecenie wygeneruje błąd, pominie ten komputer i przejdzie do następnego. Zanim dany komputer zostanie uznany za niedostępny, zwykle musi zostać przekroczony maksymalny dozwolony czas nawiązania połączenia, co oznacza, że polecenie `cmdlet` zanim zrezygnuje, zatrzyma się na około 30 – 45 sekund, wygeneruje odpowiedni komunikat o błędzie i przejdzie dalej.

Po pobraniu zestawu instancji WMI możemy je przesłać do dowolnego polecenia z rodziny `-Object-`, do dowolnego polecenia z rodziny `Format-` lub do dowolnego polecenia z rodziny `Out-`, `Export-` lub `ConvertTo-`. Poniżej pokazujemy przykład polecenia, które tworzy niestandardową tabelę właściwości klasy `Win32_BIOS`:

```
PS C:\> Gwmi Win32_BIOS | Format-Table SerialNumber,Version -auto
```

W rozdziale 10. pokazywaliśmy technikę, w której używane jest polecenie `Format-Table` do tworzenia niestandardowych kolumn tabeli. Ta technika może się przydać, kiedy chcesz wykonać zapytanie na kilku klasach WMI z danego komputera i połączyć wyniki w jedną tabelę. Aby to zrobić, powinieneś utworzyć niestandardową kolumnę dla tabeli i przypisać do niej odpowiednie zapytanie WMI. Składnia polecenia może być nieco myląca, ale za to wyniki działania są naprawdę imponujące:

```
PS C:\> gwmi -class win32_bios -computer server-r2,localhost | format-table
➤ @{label='ComputerName';expression={$_.__SERVER}}, @{label='BIOSSerial';expression=
➤ {$_.SerialNumber}}, @{label='OSBuild';expression= {gwmi -class \win32_operatingsystem-
➤ computer $_.__SERVER | select-object -expand BuildNumber}} -autosize
```

ComputerName	BIOSSerial	OSBuild
SERVER-R2	VMware-56 4d 45 fc 13 92 de c3-93 5c 40 6b 47 bb 5b 86	7600

Składnia powyższego polecenia może zyskać na przejrzystości, jeżeli skopiujesz je do środowiska PowerShell ISE i trochę przeformatujesz:

```
gwmi -class win32_bios -computer server-r2,localhost |
format-table @{label='ComputerName';expression={$_.__SERVER}},
@{label='BIOSSerial';expression={$_.SerialNumber}},
@{label='OSBuild';expression={
    gwmi -class win32_operatingsystem -comp $_.__SERVER |
    select-object -expand BuildNumber}
} -autosize
```

Oto co się tutaj dzieje:

- Polecenie `Get-WmiObject` wykonuje zapytanie na klasie `Win32_BIOS` z dwóch komputerów.
- Wyniki są przesyłane do polecenia `Format-Table`, które otrzymuje instrukcje utworzenia trzech niestandardowych kolumn:
 - Pierwsza kolumna nosi nazwę `ComputerName` i używa właściwości systemowej `__SERVER` instancji klasy `Win32_BIOS`.
 - Druga kolumna nosi nazwę `BIOSSerial` i używa właściwości `SerialNumber` instancji klasy `Win32_BIOS`.
 - Trzecia kolumna nosi nazwę `OSBuild`. Wyrażenie dla tej kolumny wykonuje całkowicie nowe zapytanie `Get-WmiObject`, pobierając klasę `Win32_OperatingSystem` z komputera reprezentowanego przez wartość właściwości `__SERVER` instancji `Win32_BIOS` (czyli z tego samego komputera zdalnego). Wynik działania jest przesyłany do polecenia `Select-Object`, które pobiera wartość właściwości `BuildNumber` instancji klasy `Win32_OperatingSystem` i wstawia ją jako wartość kolumny `OSBuild`.

Składnia tego polecenia jest nieco złożona, ale oferuje potężne możliwości. Jest to również doskonały przykład tego, ile możesz osiągnąć, łącząc ze sobą kilka starannie wyselekcjonowanych poleceń powłoki PowerShell.

Jak już wspominaliśmy, niektóre klasy WMI posiadają swoje metody. W rozdziale 16. opowiemy o tym, jak ich używać — może to być nieco skomplikowane, stąd cały temat zasługuje na własny, osobny rozdział.

14.6. Używanie polecenia *Get-CimInstance*

Polecenie *Get-CimInstance* pojawiło się jako nowość w powłoce PowerShell v3 i działa podobnie jak polecenie *Get-WmiObject*, ale z kilkoma głównymi różnicami w składni:

- Zamiast parametru *-Class* używamy parametru *-ClassName* (choć w praktyce wystarczy wpisać tylko *-Class*, więc jeśli o tym będziesz pamiętać, to będzie w porządku).
- Nie mamy parametru *-List* do wyświetlania wszystkich klas w przestrzeni nazw. Zamiast tego możesz skorzystać z polecenia *Get-CimClass* i do wyświetlenia listy klas użyć parametru *-Namespace*.
- Nie mamy parametru *-Credential*; jeżeli zamierzasz wysłać zapytania do komputera zdalnego i musisz podać alternatywne poświadczenia logowania, powinieneś wysłać polecenie *Get-CimInstance* za pomocą polecenia *Invoke-Command* (o którym mówiliśmy w poprzednim rozdziale).

Na przykład:

```
PS C:\> Get-CimInstance -ClassName Win32_LogicalDisk
```

DeviceID	DriveType	ProviderName	VolumeName	Size	FreeSpace
A:	2				
C:	3			687173...	580806...
D:	5		HB1_CCPA_X64F...	358370...	0

Jeśli chcesz wysłać zapytanie do komputera zdalnego przy użyciu alternatywnych poświadczeń logowania, możesz użyć następującego polecenia:

```
invoke-command -ScriptBlock { Get-CimInstance -ClassName win32_process } -ComputerName WIN8
↪-Credential DOMAIN\Administrator
```

14.7. Dokumentacja WMI

Wspominaliśmy wcześniej, że wyszukiwarka sieciowa jest często najlepszym sposobem na znalezienie dokumentacji WMI. Klasy *Win32_* są dobrze udokumentowane w witrynie MSDN Library firmy Microsoft, ale wyszukiwarka nadal pozostaje najprostszym sposobem na to, by wylądować na właściwej stronie. Wpisz nazwę poszukiwanej klasy w wyszukiwarce Google lub Bing, a pierwszym trafieniem będzie prawdopodobnie strona w witrynie <http://msdn.microsoft.com>.

14.8. Najczęściej spotykane problemy

Ponieważ w ostatnich dziesięciu rozdziałach uporczywie przypominaliśmy Ci o zaletach korzystania z wbudowanego systemu pomocy powłoki PowerShell, być może szukając

opisów klas WMI, instynktownie będziesz chciał uruchomić polecenie w stylu `help win32_service` bezpośrednio z poziomu powłoki PowerShell. Niestety tym razem to nie zadziała. W systemie operacyjnym nie ma żadnej dokumentacji WMI, więc funkcja pomocy PowerShell nie będzie miała gdzie jej szukać. W tej sytuacji pozostaje Ci tylko korzystanie z pomocy dostępnej online — a bardzo wiele przydatnych informacji będzie pochodzić od innych użytkowników, administratorów i programistów, a nie od firmy Microsoft. Przykładowo, spróbuj poszukać jakichkolwiek informacji na temat przestrzeni nazw `root\SecurityCenter`, a najprawdopodobniej nie znajdziesz w wynikach ani jednej strony dokumentacji pochodzącej od firmy Microsoft, co jest bardzo zasmucające.

Kolejnym zagadnieniem, które często bywa bardzo dezorientujące dla użytkowników, jest odmienna składnia kryteriów filtrowania wykorzystywanych przez WMI. Jeżeli chcesz, aby w wyniku zapytania były wyświetlane wyłącznie elementy pasujące do podanego wzorca, zawsze powinieneś używać odpowiedniego filtra. Problem w tym, że będziesz musiał zapamiętać różne składnie filtrów. Kryteria filtrowania przekazywane są do WMI i nie są przetwarzane przez powłokę PowerShell, dlatego w wywołaniu polecenia należy używać składni wymaganej przez WMI zamiast natywnych operatorów powłoki PowerShell.

Do pewnego stopnia powodem tego, że WMI sprawia takie kłopoty niektórym naszym studentom, jest fakt, że chociaż powłoka PowerShell zapewnia łatwy sposób wyszukiwania informacji z WMI, to WMI nie jest zintegrowany z powłoką PowerShell. Mechanizm WMI jest technologią zewnętrzną i ma swoje własne zasady oraz własny sposób działania. Chociaż można uzyskać do niego dostęp z poziomu powłoki PowerShell, nigdy nie będzie jednak zachowywać się dokładnie tak samo jak inne cmdlety i techniki, które są całkowicie zintegrowane z powłoką PowerShell. Miej to na względzie i uważaj na te małe pułapki i różnice wynikające z odrębności mechanizmu WMI.

14.9. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Spróbuj poświęcić trochę czasu na samodzielne wykonanie opisanych niżej zadań praktycznych. Duża trudność w korzystaniu z WMI polega na znalezieniu odpowiedniej klasy, która da Ci potrzebne informacje, stąd zapewne dużo czasu przy rozwiązywaniu zadań poświęcisz na szukanie właściwej klasy. Staraj się myśleć w kategoriach słów kluczowych (podamy Ci kilka wskazówek) i użyj przeglądarki WMI do szybkiego przeszukiwania klas (WMI Explorer wyświetla listy klas w porządku alfabetycznym, co znacząco może ułatwić Ci weryfikację Twoich przypuszczeń). Pamiętaj również o tym, że system pomocy powłoki PowerShell nie może pomóc Ci w wyszukiwaniu klas WMI.

1. Jakiej klasy można użyć do wyświetlenia bieżącego adresu IP karty sieciowej? Czy ta klasa ma jakieś metody, które można zastosować do zwolnienia adresu przydzielonego przez serwer DHCP? (Podpowiedź: *network* (sieć) to dobre słowo kluczowe).

2. Utwórz tabelę zawierającą nazwę komputera, numer kompilacji systemu operacyjnego, opis systemu operacyjnego (ang. *caption*) i numer seryjny BIOS. (Podpowiedź: widziałeś już tę technikę, ale musisz ją odwrócić, aby najpierw zapytać o klasę systemu operacyjnego, a dopiero potem zapytać o BIOS).
3. Przygotuj zapytanie wykorzystujące WMI, które będzie wyświetlało listę zainstalowanych poprawek. (Podpowiedź: firma Microsoft formalnie określa takie poprawki jako *quick-fix engineering*). Czy wygenerowana lista różni się od tej zwróconej przez polecenie `Get-Hotfix`?
4. Wyświetl listę zainstalowanych usług obejmującą również ich bieżące statusy, tryby uruchamiania i nazwy kont używane do logowania.
5. Korzystając z poleceń CIM, wyświetl w przestrzeni nazw `SecurityCenter2` wszystkie dostępne klasy, które w swoich nazwach mają słowo `Product`.
6. Po znalezieniu odpowiedniej klasy użyj poleceń CIM, aby wyświetlić informacje o dowolnej zainstalowanej aplikacji antyspyware. Możesz również sprawdzić produkty antywirusowe.

ZRÓB TO SAM Po zakończeniu wszystkich ćwiczeń spróbuj wykonać drugi zestaw dodatkowych ćwiczeń sprawdzających, które znajdziesz w dodatku.

14.10. Co dalej?

WMI jest ogromną, złożoną technologią i ktoś mógłby z łatwością napisać o niej całą książkę. W rzeczywistości niejedna taka książka już powstała, czego przykładem może być *PowerShell and WMI* Richarda Siddawaya (wyd. Manning, 2012). Książka ta zawiera mnóstwo przykładów i omawia niektóre nowe możliwości poleceń CIM wprowadzonych w powłocę PowerShell v3. Serdecznie polecamy tę książkę wszystkim, którzy chcą dowiedzieć się więcej na temat usługi WMI.

Jeżeli stwierdzisz, że korzystanie z WMI jest mocno frustrujące i dezorientujące, nie martw się. To powszechna reakcja. Ale mamy dobrą wiadomość: w powłocę PowerShell v3 i późniejszych bardzo często można już korzystać z WMI bez konieczności jego bezpośredniego „dotykania”. Dzieje się tak, ponieważ firma Microsoft udostępnia już setki cmdletów, które „opakowują” WMI. Te polecenia *cmdlet* zapewniają pomoc, umożliwiają odkrywanie potrzebnych klas, zawierają przykłady i przydatne opisy, ale wewnętrznie wykorzystują WMI. Ułatwia to korzystanie z możliwości WMI bez konieczności radzenia sobie z jego frustrującymi przypadłościami.

14.11. Odpowiedzi

1. Możesz użyć klasy `Win32_NetworkAdapterConfiguration`. Jeśli uruchomisz polecenie `Get-Wmiobject` dla tej klasy i za pomocą potoku przekażesz wyniki działania do polecenia `Get-Member`, powinieneś zobaczyć wiele metod powiązanych z DHCP. To samo możesz również znaleźć za pomocą polecenia CIM:

```
Get-CimClass win32_networkadapterconfiguration | select -expand methods |
  ↪where Name -match "dhcp"
```

2. `get-wmiobject win32_operatingsystem | Select BuildNumber,Caption,
@{1='Computername';e={$_.__SERVER}}},
@{1='BIOSSerialNumber';e={(gwmi win32_bios).serialnumber }} | ft -auto`
lub używając polecenia CIM:
`get-ciminstance win32_operatingsystem | Select BuildNumber,Caption,
@{1='Computername';e={$_.CSName}}},
@{1='BIOSSerialNumber';e={(get-ciminstance win32_bios).serialnumber}} |
ft -auto`
3. `get-wmiobject win32_quickfixengineering`
4. **Otrzymane wyniki powinny być bardzo podobne.**
5. `get-wmiobject win32_service | Select Name,State,StartMode,StartName`
lub
`get-ciminstance win32_service | Select Name,State,StartMode,StartName`
6. `get-cimclass -namespace root/SecurityCenter2 -ClassName *product`
`get-ciminstance -namespace root/SecurityCenter2 -ClassName AntiSpywareProduct`

15

Wielozadaniowość z zadaniami działającymi w tle

Wszyscy zawsze mówią Ci, żebyś działał *wielozadaniowo*, prawda? Dlaczego więc powłoka PowerShell nie mogłaby Ci w tym pomóc, robiąc więcej niż jedną rzecz jednocześnie? Okazuje się, że powłoka PowerShell może tak działać, szczególnie w przypadku dłuższych zadań, które mogą obejmować wiele komputerów docelowych. Zanim jednak zagłębisz się w lekturze tego rozdziału, upewnij się, że uważnie przeczytałeś rozdziały 13. i 14., ponieważ będziemy tutaj wykorzystywać i dalej rozwijać omawiane tam koncepcje komunikacji zdalnej oraz WMI.

15.1. Uruchamianie wielu zadań powłoki PowerShell w tym samym czasie

Powłokę PowerShell powinieneś traktować jako aplikację jednowątkową, co oznacza, że może ona wykonywać tylko jedno zadanie naraz. Wpisujesz polecenie, naciskasz klawisz *Enter* i powłoka rozpoczyna jego wykonywanie. Nie możesz uruchomić drugiego polecenia, dopóki pierwsze polecenie nie zakończy działania.

Jednak dzięki możliwości uruchamiania zadań do działania w tle, powłoka PowerShell może przenieść wykonywanie danego polecenia do osobnego wątku tła (czyli inaczej mówiąc, do osobnego procesu powłoki PowerShell działającego w tle). Dzięki temu takie polecenie może działać na drugim planie, a Ty możesz użyć powłoki do wykonania innego zadania. Pamiętaj jednak, że taką decyzję musisz podjąć przed uruchomieniem

polecenia; po naciśnięciu klawisza *Enter* nie można przesuwać uruchomionego polecenia do działania w tle.

Po uruchomieniu zadania w tle powłoka PowerShell zapewnia mechanizmy sprawdzania ich statusu, pobierania wyników i tak dalej.

UWAGA Wczesne wydania powłoki PowerShell dla systemu Linux posiadają niektóre mechanizmy pozwalające na korzystanie z zadań działających w tle, ale nie są one jeszcze w pełni funkcjonalne. Oczekujemy, że powłoka PowerShell dla systemu Linux w końcu kiedyś będzie w pełni obsługiwać zadania w tle tej powłoki, ale nie możemy zagwarantować, że będzie zachowywać się w stu procentach tak samo jak jej odpowiednik w systemie Windows. Na szczęście wiesz już jednak doskonale, jak korzystać z systemu pomocy powłoki PowerShell, a w tym rozdziale objaśnimy wszystkie niezbędne, dodatkowe zagadnienia.

15.2. Zadania synchroniczne i asynchroniczne

Najpierw musimy wprowadzić nieco nowej terminologii. Powłoka PowerShell uruchamia normalne polecenia *synchronicznie*, co oznacza, że naciskamy klawisz *Enter*, a następnie czekamy na zakończenie działania polecenia. Przeniesienie zadania do działania w tle umożliwia jego pracę *asynchroniczną*, czyli po uruchomieniu takiego polecenia w tle możemy dalej używać powłoki do wykonywania innych zadań.

Przyjrzyjmy się teraz kilku ważnym różnicom między uruchamianiem zadań synchronicznych i asynchronicznych:

- Po uruchomieniu polecenia synchronicznie możesz odpowiadać na żądania wprowadzenia danych wejściowych. Kiedy uruchamiasz polecenia do działania w tle, nie ma możliwości wyświetlenia i zobaczenia żądania wprowadzenia danych wejściowych — w rzeczywistości próba „wyświetlenia” takiego żądania spowoduje zatrzymanie wykonywania polecenia.
- Jeżeli coś pójdzie nie tak, jak powinno, polecenia synchroniczne generują i od razu wyświetlają odpowiednie komunikaty o błędach. Polecenia działające w tle również generują komunikaty o błędach, ale nie zobaczysz ich od razu. Aby je wyświetlić, będziesz musiał podjąć odpowiednie kroki (więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 22.).
- Jeżeli podczas wywoływania polecenia synchronicznego pominiesz jakiś wymagany parametr, powłoka PowerShell może poprosić Cię o brakujące informacje. W przypadku polecenia uruchomionego do pracy w tle powłoka nie może tego zrobić, więc taka próba wykonania polecenia zakończy się niepowodzeniem.
- Wyniki działania polecenia synchronicznego są wyświetlane na ekranie od razu, gdy staną się dostępne. W przypadku poleceń działających w tle musisz poczekać, aż polecenie zakończy działanie, i dopiero wtedy możesz pobrać wyniki działania z pamięci podręcznej.

Zazwyczaj najpierw uruchamiamy polecenia synchronicznie, tak aby sprawdzić, czy działają poprawnie, a uruchamiamy je do działania w tle dopiero wtedy, gdy wiemy, że są w pełni przetestowane i funkcjonują zgodnie z naszymi oczekiwaniami. Postępujemy w ten sposób po to, aby zagwarantować, że polecenie uruchomione do pracy w tle będzie działać bez żadnych problemów i że będzie miało maksymalnie dużą szansę na poprawne zakończenie działania.

Jak już wspominaliśmy, w powłoce PowerShell polecenia działające w tle są określane mianem **zadań** (ang. *jobs*). Zadania takie możesz tworzyć na kilka sposobów, a do zarządzania nimi możesz użyć kilku ciekawych poleceń.

Dla zainteresowanych

Technicznie rzecz biorąc, zadania omawiane w tym rozdziale są tylko jednym z rodzajów zadań, z którymi możesz się spotkać w codziennej pracy. Zadania są po prostu mechanizmem rozszerzającym funkcjonalność powłoki PowerShell, co oznacza, że istnieje możliwość, aby ktoś inny (niezależnie od tego, czy to będzie deweloper firmy Microsoft, czy programista innej firmy) utworzył inne komponenty nazywane zadaniami, które będą wyglądały i działały nieco inaczej niż to, co opisujemy w tym rozdziale. Przykładowo, zadania zaplanowane, które będziemy omawiać w dalszej części tego rozdziału, również działają nieco inaczej niż „normalne” zadania, którymi będziemy się zajmować w pierwszej kolejności, a instalując różne rozszerzenia powłoki PowerShell, możesz napotkać jeszcze inne rodzaje zadań. Chcielibyśmy, abyś dobrze to zrozumiał i pamiętał, że to, czego się tutaj uczysz, odnosi się tylko do rodzimych, standardowych zadań powłoki PowerShell.

15.3. Tworzenie zadań lokalnych

Najprostszym rodzajem zadań, które to weźmiemy na pierwszy ogień, są zadania lokalne. Są to polecenia uruchamiane niemal wyłącznie na komputerze lokalnym (z wyjątkami, które omówimy już za moment) i działające w tle.

Aby uruchomić jedno z takich zadań, powinieneś użyć polecenia `Start-Job`. Parametr `-scriptblock` umożliwia określenie polecenia (lub poleceń), które powinny zostać wykonane. Powłoka PowerShell tworzy domyślną nazwę zadania (`Job1`, `Job2` i tak dalej), ale w razie potrzeby za pomocą parametru `-Name` możesz określić swoją własną, niestandardową nazwę zadania. Jeżeli chcesz, aby zadanie działało z alternatywnymi poświadczeniami logowania, możesz użyć parametru `-credential`, który akceptuje nazwę konta użytkownika w formacie `Domena\uzytkownik` i wyświetla monit o podanie hasła. Zamiast bezpośrednio definiować polecenia za pomocą parametru `-scriptblock`, możesz użyć parametru `-FilePath` do wskazania odpowiedniego skryptu zawierającego cały zestaw odpowiednich poleceń.

Oto przykład uruchomienia prostego zadania do działania w tle:

```
PS C:\> start-job -scriptblock { dir }
Id      Name      State      HasMoreData      Location
--      -
1       Job1      Running    True              localhost
```

Nasze polecenie tworzy obiekt zadania i jak widać w przykładzie, zadanie zaczyna działać natychmiast. Nowo utworzone zadanie ma również przypisany kolejny numer identyfikacyjny, który jest pokazany w tabeli.

Powiedzieliśmy, że takie zadania działają całkowicie na komputerze lokalnym, i jest to prawda. Jeżeli jednak polecenia uruchamiane w zadaniu lokalnym obsługują parametr `-computerName`, to takie zadanie może mieć również dostęp do komputerów zdalnych. Oto przykład:

```
PS C:\> start-job -scriptblock {
get-eventlog security -computer server-r2
}
```

Id	Name	State	HasMoreData	Location
--	----	-----	-----	-----
3	Job3	Running	True	localhost

ZRÓB TO SAM Mamy nadzieję, że podążasz za nimi i samodzielnie uruchamiasz wszystkie opisywane tutaj polecenia. Jeżeli masz do dyspozycji tylko jeden komputer, odwołuj się do jego nazwy i używaj adresu `localhost` jako alternatywy, dzięki czemu powłoka PowerShell będzie działała tak, jakby miała do czynienia z dwoma komputerami.

Przetwarzanie takiego zadania odbywa się na komputerze lokalnym. Komunikuje się ono z określonym komputerem zdalnym (w tym przykładzie `SERVER-R2`), więc takie zadanie jest w pewnym sensie „zadaniem zdalnym”. Ze względu jednak na fakt, że samo polecenie działa lokalnie, nadal nazywamy to zadaniem lokalnym.

Uważni czytelnicy z pewnością zauważyli, że pierwsze utworzone przez nas zadanie nosi nazwę `Job1` i otrzymało identyfikator 1, ale drugie zadanie to `Job3` z identyfikatorem 3. Okazuje się, że każde zadanie ma co najmniej jedno **zadanie podrzędne** (ang. *child job*), stąd nasze pierwsze zadanie podrzędne (dla zadania `Job1`) otrzymało nazwę `Job2` oraz identyfikator 2. Zadania podrzędne będziemy bardziej szczegółowo omawiać w dalszej części tego rozdziału.

A oto coś, o czym zawsze powinniśmy pamiętać: chociaż zadania lokalne, jak sama nazwa wskazuje, są uruchamiane lokalnie, to jednak wymagają całej infrastruktury systemu komunikacji zdalnej powłoki PowerShell, o której mówiliśmy w rozdziale 13. Jeżeli mechanizm komunikacji zdalnej nie został włączony, uruchamianie zadań lokalnych nie będzie możliwe.

15.4. Zapytania WMI jako zadania działające w tle

Innym sposobem na uruchomienie zadania w tle jest użycie polecenia `Get-WmiObject`. Jak wyjaśniliśmy w poprzednim rozdziale, polecenie `Get-WmiObject` może komunikować się z jednym lub większą liczbą komputerów zdalnych, ale robi to sekwencyjnie. Oznacza to, że w przypadku długiej listy nazw komputerów wykonanie polecenia może zająć dużo czasu, więc naturalnym wyborem jest przeniesienie takiego polecenia do pracy w tle. Aby to zrobić, użyj polecenia `Get-WmiObject` jak zwykle, ale dodatkowo powinniśmy

umieścić w wierszu wywołania parametr `-AsJob`. Niestety w tym przypadku nie będziesz miał możliwości określenia własnej, niestandardowej nazwy zadania, zatem jesteś skazany na nazwę domyślną, którą utworzy powłoka PowerShell.

ZRÓB TO SAM Jeżeli chcesz spróbować samodzielnie wykonać opisywane polecenia w systemie testowym, powinieneś utworzyć plik tekstowy o nazwie *allservers.txt*. W naszym przykładzie umieściliśmy go w katalogu głównym dysku C: (ponieważ tego katalogu używamy jako katalogu roboczego powłoki PowerShell w naszych przykładach). W pliku umieściliśmy kilka nazw komputerów, po jednej w każdym wierszu. Jeżeli nie dysponujesz kilkoma komputerami, możesz umieścić w pliku nazwę swojego komputera oraz adres localhost.

```
PS C:\> get-wmiobject win32_operatingsystem -computersname
(get-content allservers.txt) -asjob
WARNING: column "Command" does not fit into the display and was removed.
Id          Name          State          HasMoreData    Location
--          ---          -
5           Job5          Running        False           server-r2.lo...
```

Tym razem powłoka tworzy jedno zadanie nadrzędne (o nazwie Job5, która jest wyświetlana w tabeli bezpośrednio w wynikach działania polecenia) oraz po jednym zadaniu podrzędnym dla każdego wskazanego komputera zdalnego. Jak widać, w kolumnie Location (lokalizacja) w tabeli wyników znajduje się tyle nazw komputerów, ile się zmieściło, co wskazuje, że zadanie będzie wykonywane na wielu komputerach.

Bardzo ważne jest to, abyś zrozumiał, że polecenie `Get-WmiObject` działa tylko na Twoim komputerze i używa normalnej komunikacji WMI do kontaktowania się z określonymi komputerami zdalnymi. Wciąż jednak robi to po jednym jednocześnie, postępuje zgodnie ze standardowymi regułami pomijania niedostępnych komputerów i tak dalej. W rzeczywistości działa to identycznie jak przy synchronicznym wywołaniu polecenia `Get-WmiObject`, z tym wyjątkiem, że w naszym przykładzie polecenie to funkcjonuje w tle.

ZRÓB TO SAM Istnieją również inne polecenia niż `Get-WmiObject`, które można bezpośrednio uruchomić do pracy w tle. Spróbuj wykonać polecenie `Help *` -parameter `asjob` i samodzielnie poszukaj takich poleceń.

Zwróć uwagę, że nowsze polecenie `Get-CimInstance`, o którym mówiliśmy już w rozdziale 14., nie ma parametru `-AsJob`. Jeżeli chcesz użyć tego polecenia w zadaniu działającym w tle, powinieneś zastosować polecenie `Start-Job` lub `Invoke-Command` (które poznasz już za chwilę) i w parametrze `-scriptblock` umieścić odpowiednie polecenie `Get-CimInstance` (lub dowolne inne polecenie CIM).

15.5. Komunikacja zdalna jako zadanie działające w tle

Przeanalizujemy jeszcze jedną technikę, której można użyć do utworzenia nowego zadania, wykorzystującą mechanizm komunikacji zdalnej powłoki PowerShell (omawialiśmy

go w rozdziale 13.). Podobnie jak w przypadku polecenia `Get-WmiObject`, tworzenie takiego zadania rozpoczynamy od dodania parametru `-AsJob`, z tym że tym razem dodajemy go do cmdletu `Invoke-Command`.

Mamy tutaj bardzo ważną różnicę: bez względu na to, jakie polecenie zostanie umieszczone w bloku parametru `-scriptblock` (lub `-command`, który — jak pamiętasz — jest aliasem parametru `-scriptblock`), zostanie ono przesłane równolegle do wszystkich wskazanych komputerów. Maksymalnie można nawiązać połączenie z 32 komputerami jednocześnie (chyba że odpowiednio zmodyfikujesz parametr `-throttleLimit`, aby zezwolić na większą lub mniejszą liczbę równoległych sesji), więc jeżeli podasz więcej niż 32 nazwy komputerów, działanie polecenia rozpocznie się na pierwszych 32 i będzie sukcesywnie uruchamiane na kolejnych maszynach, w miarę jak zadanie będzie kończyć działanie na poszczególnych maszynach z pierwszego zestawu. Po zakończeniu działania na wszystkich komputerach zadanie najwyższego poziomu wyświetli informacje o statusie ukończonego zadania.

W przeciwieństwie do dwóch pozostałych sposobów uruchamiania zadań działających w tle taka technika wymaga zainstalowania powłoki PowerShell v2 lub v3 na każdym komputerze docelowym oraz uruchomienia mechanizmu komunikacji zdalnej powłoki PowerShell na każdym z tych komputerów. Ponieważ samo polecenie fizycznie wykonuje się na każdym komputerze zdalnym, całkowite obciążenie obliczeniowe zostaje rozłożone na wiele maszyn, co może znacząco poprawić wydajność w przypadku złożonych lub długo działających poleceń. Wyniki działania są przesyłane na Twój komputer i są przechowywane razem z zadaniem, dopóki nie będziesz gotowy do ich wyświetlenia.

W poniższym przykładzie używamy również parametru `-JobName`, który pozwala określić nazwę zadania inną niż niewiele mówiąca nazwa domyślna:

```
PS C:\> invoke-command -command { get-process }
-computername (get-content .\allservers.txt )
-asjob -jobname MyRemoteJob
```

WARNING: column "Command" does not fit into the display and was removed.

Id	Name	State	HasMoreData	Location
8	MyRemoteJob	Running	False	server-r2,lo...

15.6. Pobieranie wyników działania zadania uruchomionego w tle

Pierwszą rzeczą, którą prawdopodobnie będziesz chciał zrobić po uruchomieniu zadania drugoplanowego, jest sprawdzenie, czy zakończyło ono już swoje działanie. Polecenie `Get-Job` pobiera wszystkie zadania aktualnie zdefiniowane w systemie i pokazuje status każdego z nich:

```
PS C:\> get-job
```

Id	Name	State	HasMoreData	Location
1	Job1	Completed	True	localhost
3	Job3	Completed	True	localhost
5	Job5	Completed	True	server-r2,lo...
8	MyRemoteJob	Completed	True	server-r2,lo...

Możesz także pobrać określone zadanie, używając jego identyfikatora lub nazwy. Sugerujemy, abyś to zrobił i za pomocą potoku przekazał wyniki do polecenia `Format-List *`, ponieważ w ten sposób możesz zobaczyć kilka cennych informacji:

```
PS C:\> get-job -id 1 | format-list *
State           : Completed
HasMoreData     : True
StatusMessage   :
Location        : localhost
Command         : dir
JobStateInfo     : Completed
Finished        : System.Threading.ManualResetEvent
InstanceId      : e1ddde9e-81e7-4b18-93c4-4c1d2a5c372c
Id              : 1
Name            : Job1
ChildJobs       : {Job2}
Output          : {}
Error           : {}
Progress        : {}
Verbose         : {}
Debug           : {}
Warning         : {}
```

ZRÓB TO SAM Jeżeli na swoim komputerze wykonujesz wszystkie nasze przykładowe polecenia, pamiętaj, że Twoje identyfikatory i nazwy zadań mogą różnić się od naszych. Skup się na wynikach zadania `Get-Job`, aby pobrać swoje identyfikatory i nazwy zadań oraz odpowiednio zastąpić je w przykładach. Powinieneś również pamiętać, że firma Microsoft w kolejnych wersjach powłoki PowerShell sukcesywnie rozbudowuje obiekty zadania, więc wyniki działania tych poleceń podczas przeglądania wszystkich właściwości mogą być nieco inne.

Właściwość `ChildJobs` jest jedną z najważniejszych informacji, więc zajmiemy się nią już za chwilę.

Aby pobrać wyniki działania zadania, powinieneś użyć polecenia `Receive-Job`. Zanim jednak to zrobisz, musisz wiedzieć kilka rzeczy:

- Musisz wskazać zadanie, którego wyniki działania chcesz otrzymać. Możesz to zrobić, używając identyfikatora ID lub nazwy zadania albo pobierając zadanie za pomocą funkcji `Get-Job` i przesyłając je za pomocą potoku do polecenia `Receive-Job`.
- Jeżeli pobierzesz wyniki działania zadania nadrzędnego, będą one zawierały wyniki działania wszystkich zadań podrzędnych. Zamiast tego możesz oczywiście wybrać opcję pobierania wyników z jednego zadania podrzędnego lub z ich większej liczby.
- Zazwyczaj pobranie wyników działania zadania usuwa je z pamięci podręcznej zadań, tak że nie można ich pozyskać po raz drugi. Aby jednak zachować kopię wyników w pamięci, powinieneś użyć parametru `-keep`. Zamiast tego możesz również zapisać wyniki w pliku `CliXML`, którego możesz później użyć do dalszego przetwarzania danych.

- Wyniki zadania mogą mieć postać obiektów zdeserializowanych (mówiliśmy o nich w rozdziale 13.), będących migawkami reprezentującymi stan obiektów w chwili wygenerowania. Takie zdeserializowane obiekty nie posiadają żadnych metod, które można wykonać. W razie potrzeby możesz potokować wyniki działania zadania bezpośrednio do poleceń takich jak Sort-Object, Format-List, Export-CSV, ConvertTo-HTML czy Out-File.

Oto prosty przykład pobierania wyników działania zadania:

```
PS C:\> receive-job -id 1
Directory: C:\Users\Administrator\Documents
Mode                LastWriteTime         Length Name
----                -
d----             11/21/2015 11:53 AM                Integration Services Script Component
d----             11/21/2015 11:53 AM                Integration Services Script Task
d----              4/23/2010  7:54 AM                SQL Server Management Studio
d----              4/23/2010  7:55 AM                Visual Studio 2005
d----             11/21/2009 11:50 AM                Visual Studio 2008
```

Powyższy listing pokazuje interesujący zestaw wyników działania. Oto krótkie przypomnienie polecenia, które uruchomiło nasze zadanie:

```
PS C:\> start-job -scriptblock { dir }
```

Chociaż katalog roboczy naszej powłoki był ustawiony na główny katalog dysku C:, katalog widoczny w wynikach działania polecenia to *C:\Users\Administrator\Documents*. Jak widać, po uruchomieniu nawet lokalne polecenia zyskują nieco inny kontekst, co może skutkować zmianą lokalizacji katalogu roboczego i innych ustawień. Nigdy nie powinieneś z góry przyjmować żadnych założeń dotyczących ścieżek plików dla zadań działających w tle — zamiast tego powinieneś zawsze używać ścieżek bezwzględnych, tak aby upewnić się, że zawsze możesz prawidłowo odwoływać się do plików niezbędnych do wykonania zadania. Jeżeli chcesz, aby zadanie działające w tle wyświetliło zawartość katalogu C:\, powinieneś zatem uruchomić następujące polecenie:

```
PS C:\> start-job -scriptblock { dir c:\ }
```

Podczas pobierania wyników zadania 1., nie użyliśmy parametru `-keep`. Z tego powodu, jeżeli teraz ponownie spróbujemy uzyskać te same wyniki, nie otrzymamy nic, ponieważ wyniki nie są już przechowywane w pamięci podręcznej razem z zadaniem:

```
PS C:\> receive-job -id 1
```

Aby po zakończeniu pobierania zachować wyniki działania w pamięci podręcznej zadania, powinieneś użyć następującego polecenia:

```
PS C:\> receive-job -id 3 -keep
Index      Time      EntryType      Source      InstanceID      Message
-----
6542      Oct 04 11:55 SuccessA... Microsoft-Windows... 4634      An...
6541      Oct 04 11:55 SuccessA... Microsoft-Windows... 4624      An...
6540      Oct 04 11:55 SuccessA... Microsoft-Windows... 4672      Sp...
6539      Oct 04 11:54 SuccessA... Microsoft-Windows... 4634      An...
```

Kiedy wyniki działania danego zadania nie będą Ci już potrzebne, zapewne będziesz chciał zwolnić pamięć używaną do ich buforowania, musimy więc również powiedzieć kilka słów na ten temat. Najpierw jednak przyjrzyjmy się prostemu przykładowi potokowania wyników pracy bezpośrednio do innego polecenia:

```
PS C:\> receive-job -name myremotejob | sort-object PSComputerName |
Format-Table -groupby PSComputerName
PSComputerName: localhost
Handles      NPM(K)      PM(K)      WS(K)      VM(M)      CPU(s)      Id      ProcessName      PSComputerName
-----
195           10        2780        5692         30         0.70      484      lsmd             loca...
237           38       40704       36920        547         3.17     1244      Micro...         loca...
146           17        3260        7192         60         0.20     3492      msdtc            loca...
1318          100       42004       28896        154        15.31     476      lsass            loca...
```

Są to wyniki działania zadania, które utworzyliśmy przy użyciu polecenia `Invoke-Command`. Jak zwykle cmdlet ten dodał właściwość `PSComputerName`, dzięki czemu możemy śledzić, które obiekty pochodzą z danego komputera. Ponieważ pobieraliśmy wyniki zadania nadrzędnego, wyniki działania zawierają informacje otrzymane ze wszystkich wskazanych przez nas komputerów, a polecenie sortuje je według nazwy komputera, po czym tworzy indywidualną grupę tabel dla każdej z maszyn.

Polecenie `Get-Job` może również informować Cię o tym, które zadania mają gotowe wyniki działania:

```
PS C:\> get-job
WARNING: column "Command" does not fit into the display and was removed.
Id  Name                State      HasMoreData  Location
--  -
1   Job1                Completed  False        localhost
3   Job3                Completed  True         localhost
5   Job5                Completed  True         server-r2,lo...
8   MyRemoteJob         Completed  False        server-r2,lo...
```

Kolumna `HasMoreData` będzie miała wartość `False`, jeżeli wyniki działania tego zadania nie są buforowane. W przypadku zadań `Job1` i `MyRemoteJob` wyniki już zostały pobrane, a podczas pobierania nie używaliśmy parametru `-keep`.

15.7. Praca z zadaniami podrzędnymi

Wspomnieliśmy wcześniej, że wszystkie zadania składają się z jednego zadania nadrzędnego i przynajmniej jednego zadania podrzędnego. Spójrzmy ponownie na przykładowe zadanie:

```
PS C:\> get-job -id 1 | format-list *
State           : Completed
HasMoreData      : True
StatusMessage    :
Location         : localhost
Command          : dir
JobStateInfo     : Completed
```

```

Finished      : System.Threading.ManualResetEvent
InstanceId    : e1ddde9e-81e7-4b18-93c4-4c1d2a5c372c
Id            : 1
Name          : Job1
ChildJobs     : {Job2}
Output        : {}
Error         : {}
Progress      : {}
Verbose       : {}
Debug         : {}
Warning       : {}

```

ZRÓB TO SAM Nie próbuj teraz samodzielnie wykonywać opisywanych niżej poleceń, ponieważ jeżeli robiłeś to konsekwentnie do tej pory, pobrałeś już wyniki zadania Job1. Jeżeli chcesz jednak spróbować, uruchom nowe zadanie, wykonując polecenie `Start-Job -script {Get-Service}` i użyj identyfikatora tego nowego zadania zamiast identyfikatora ID 1, którego użyliśmy w naszym przykładzie.

Wyniki działania powyższego polecenia pokazują, że zadanie nadrzędne Job1 ma swoje zadanie podrzędne o nazwie Job2. Znasz już jego nazwę, więc możesz bezpośrednio pobrać wyniki jego działania:

```

PS C:\> get-job -name job2 | format-list *
State           : Completed
StatusMessage   :
HasMoreData     : True
Location        : localhost
Runspace        : System.Management.Automation.RemoteRunspace
Command         : dir
JobStateInfo    : Completed
Finished        : System.Threading.ManualResetEvent
InstanceId      : a21a91e7-549b-4be6-979d-2a896683313c
Id             : 2
Name            : Job2
ChildJobs       : {}
Output          : {Integration Services Script Component, Integration Services
                  Script Task, SQL Server Management Studio, Visual Studio 2005...}
Error           : {}
Progress        : {}
Verbose         : {}
Debug           : {}
Warning         : {}

```

Czasami określone zadanie nadrzędne ma zbyt wiele zadań podrzędnych, aby wyświetlać je w tej formie, a zatem możesz wyświetlić je w nieco inny sposób, tak jak to zostało pokazane poniżej:

```

PS C:\> get-job -id 1 | select-object -expand childjobs
WARNING: column "Command" does not fit into the display and was removed.
Id      Name      State      HasMoreData      Location
--      -
2       Job2       Completed  True              localhost

```

Takie polecenie tworzy tabelę zadań podrzędnych dla zadania o identyfikatorze ID równym 1; taka tabela może swobodnie pomieścić informacje o wszystkich zadaniach podrzędnych, niezależnie od ich ilości.

W razie potrzeby możesz bezpośrednio pobierać wyniki działania wybranych zadań podrzędnych, przekazując odpowiednią nazwę lub identyfikator zadania do polecenia `Receive-Job`.

15.8. Polecenia do zarządzania zadaniami

Istnieją jeszcze trzy dodatkowe polecenia pozwalające na zarządzanie zadaniami. Dla każdego z tych poleceń możemy wskazać określone zadanie, podając jego identyfikator, nazwę lub pobierając zadanie i przekazując za pomocą potoku do jednego z następujących poleceń:

- `Remove-Job` — usuwa zadanie wraz ze wszystkimi jego wynikami działania zapisanymi w pamięci podręcznej.
- `Stop-Job` — jeżeli próba wykonania zadania zakończyła się jego zawieszeniem czy zablokowaniem działania, to wykonanie tego polecenia zakończy działanie tego zadania. Nadal możesz jednak pobrać wszelkie wyniki jego działania, które zostały wygenerowane do tego momentu.
- `Wait-Job` — to polecenie jest bardzo przydatne w sytuacji, kiedy skrypt ma uruchomić określone zadanie i powinien kontynuować pracę dopiero po zakończeniu działania zadania. Polecenie to powoduje, że powłoka zatrzymuje się, czeka na zakończenie działania danego zadania i dopiero wtedy wznowia działanie.

Przykładowo, aby usunąć zadania, których wyniki działania już pobraliśmy, możemy użyć następującego polecenia:

```
PS C:\> get-job | where { -not $_.HasMoreData } | remove-job
PS C:\> get-job
WARNING: column "Command" does not fit into the display and was removed.
```

Id	Name	State	HasMoreData	Location
3	Job3	Completed	True	localhost
5	Job5	Completed	True	server-r2.lo...

Próba wykonania zadania może również zakończyć się niepowodzeniem, co zwykle oznacza, że coś poszło nie tak, jak powinno. Rozważmy następujący przykład:

```
PS C:\> invoke-command -command { nothing } -computer notonline -asjob -jobname ThisWillFail
WARNING: column "Command" does not fit into the display and was removed.
```

Id	Name	State	HasMoreData	Location
11	ThisWillFail	Failed	False	notonline

W tym przykładzie utworzyliśmy zadanie z nieistniejącym poleceniem i próbowaliśmy je uruchomić na nieistniejącym komputerze. Oczywiście próba wykonania takiego zadania nie powiodła się, co można zobaczyć w jego statusie. W takiej sytuacji nie musi-

my używać polecenia `Stop-Job`; takie zadanie nie działa. W razie potrzeby możemy jednak uzyskać listę jego zadań podrzędnych:

```
PS C:\> get-job -id 11 | format-list *
State                : Failed
HasMoreData          : False
StatusMessage        :
Location             : notonline
Command              : nothing
JobStateInfo          : Failed
Finished             : System.Threading.ManualResetEvent
InstanceId            : d5f47bf7-53db-458d-8a08-07969305820e
Id                   : 11
Name                  : ThisWillFail
ChildJobs             : {Job12}
Output               : {}
Error                : {}
Progress             : {}
Verbose              : {}
Debug                : {}
Warning              : {}
```

Mając odpowiednią nazwę, możemy pobrać zadanie podrzędne:

```
PS C:\> get-job -name job12
WARNING: column "Command" does not fit into the display and was removed.
Id  Name                State      HasMoreData  Location
--  ---                -
12  Job12                Failed     False        notonline
```

Jak widać, to zadanie podrzędne nie ma żadnych wyników działania, zatem nie mamy żadnych wyników do pobrania. Warto jednak pamiętać, że komunikaty o błędach wykonania zadania również są przechowywane w danych wyjściowych i można je uzyskać za pomocą polecenia `Receive-Job`:

```
PS C:\> receive-job -name job12
Receive-Job : [notonline] Connecting to remote server failed with the following error message
↳: WinRM cannot process the request. The following error occurred while using Kerberos
↳authentication: The network path was not found.
```

Pełny komunikat o błędzie jest znacznie dłuższy; skróciliśmy go tutaj, aby zaoszczędzić trochę miejsca. Zauważ, że komunikat zawiera nazwę komputera, z którego pochodzi błąd — `[notonline]`. Co się jednak stanie, jeżeli zadanie nie będzie mogło nawiązać połączenia z jednym z komputerów? Spróbujmy:

```
PS C:\> invoke-command -command { nothing }
-computer notonline,server-r2 -asjob -jobname ThisWillFail
WARNING: column "Command" does not fit into the display and was removed.
Id  Name                State      HasMoreData  Location
--  ---                -
13  ThisWillFail         Running    True         notonline,se...
```


Po uruchomieniu zadania czekamy chwilę i wykonujemy następujące polecenie:

```
PS C:\> get-job
WARNING: column "Command" does not fit into the display and was removed.
Id   Name                State      HasMoreData  Location
--   -
13   ThisWillFail         Failed     False        notonline,se...
```

Próba wykonania zadania nadal kończy się niepowodzeniem, ale spójrzmy na poszczególne zadania podrzędne:

```
PS C:\> get-job -id 13 | select -expand childjobs
WARNING: column "Command" does not fit into the display and was removed.
Id   Name                State      HasMoreData  Location
--   -
14   Job14               Failed     False        notonline
15   Job15               Failed     False        server-r2
```

No dobrze, oba zadania zakończyły się niepowodzeniem. Wydaje nam się, że wiemy, dlaczego zadanie Job14 nie działa, ale co jest nie tak z zadaniem Job15?

```
PS C:\> receive-job -name job15
Receive-Job : The term 'nothing' is not recognized as the name of a cmdlet, function, script
↳file, or operable program. Check the spelling of the name, or if a path was included, verify
↳that the path is correct and try again.
```

No tak, wszystko się zgadza — przecież nakazaliśmy, aby zadanie dokonało próby uruchomienia fałszywego polecenia. Jak widać, poszczególne zadania podrzędne mogą nie działać poprawnie z różnych powodów, a powłoka PowerShell śledzi każde z nich osobno.

15.9. Zaplanowane zadania

W powłoce PowerShell v3 wprowadzona została obsługa zaplanowanych zadań — przyjazny dla użytkownika „powershellowy” sposób tworzenia zadań w harmonogramie zadań systemu Windows. Takie zadania działają nieco inaczej niż zadania drugoplanowe, które omawialiśmy do tej pory; jak pisaliśmy wcześniej, zadania są mechanizmem rozszerzającym funkcjonalność powłoki PowerShell, co oznacza, że może istnieć wiele rodzajów zadań, z których każde działa nieco inaczej. Zadania zaplanowane są jednymi z tych innych rodzajów zadań. Są one również nieco inne niż standardowe zadania programu Task Scheduler, ponieważ wyniki działania tych zadań są przechowywane na dysku, dzięki czemu powłoka PowerShell może je później pobierać. W języku angielskim istnieją nawet dwa nieco różniące się od siebie określenia: *scheduled jobs*, odnoszące się do zaplanowanych zadań powłoki PowerShell, oraz *scheduled tasks*, bardziej odnoszące się do starych dobrych zadań programu Task Scheduler, z których zapewne już korzystałeś.

Tworzenie zaplanowanego zadania rozpoczynamy od utworzenia odpowiedniego wyzwalacza (New-JobTrigger), określającego, kiedy zadanie zostanie uruchomione. Następnie możesz ustawić opcje dla zadania (New-ScheduledTaskOption) i wreszcie rejestrujesz

zadanie (Register-ScheduledJob) w programie Task Scheduler. Spowoduje to utworzenie definicji zadania w formacie XML programu Task Scheduler, zapisanie go na dysku i utworzenie hierarchii folderów, w których będą przechowywane wyniki zadania po każdym uruchomieniu.

Spójrzmy na prosty przykład:

```
PS C:\> Register-ScheduledJob -Name DailyProcList -ScriptBlock { Get-Process } -Trigger
↳ (New-JobTrigger -Daily -At 2am) -ScheduledJobOption (New-ScheduledJobOption -WakeToRun
↳ -RunElevated)
```

WARNING: column "Enabled" does not fit into the display and was removed.

Id	Name	JobTriggers	Command
1	DailyProcList	{1}	Get-Process

Wykonanie tego polecenia spowoduje utworzenie nowego zadania, które uruchamia polecenie Get-Process codziennie o godzinie 2:00, w razie potrzeby budzi komputer ze stanu uśpienia i działa z podwyższonymi uprawnieniami. Po uruchomieniu zadania możesz powrócić do powłoki PowerShell i uruchomić polecenie Get-Job, aby wyświetlić listę standardowych zadań powiązanych z każdym zakończonym zaplanowanym zadaniem:

```
PS C:\> get-job
```

WARNING: column "Command" does not fit into the display and was removed.

Id	Name	State	HasMoreData	Location
6	DailyProcList	Completed	True	localhost
9	DailyProcList	Completed	True	localhost

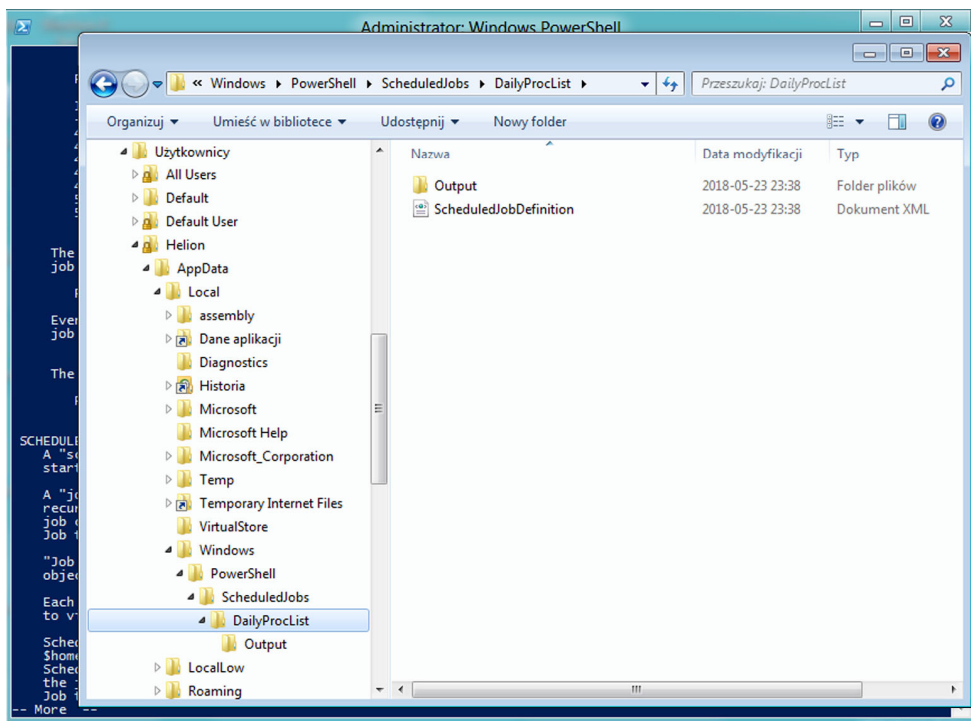
W przeciwieństwie do normalnych zadań powłoki pobieranie wyników zaplanowanych zadań nie powoduje ich usunięcia, ponieważ są one buforowane na dysku twardym komputera, a nie w jego pamięci operacyjnej, zatem w razie potrzeby możesz pobierać takie wyniki, ile razy zechcesz. Po usunięciu zadania wyniki również zostaną usunięte z dysku. Jak pokazano na rysunku 15.1, wyniki działania znajdują się w określonym folderze na dysku, a polecenie Receive-Job może je odczytywać.

Ilość zapisanych i przechowywanych zestawów wyników możesz kontrolować, używając parametru -MaxResultCount polecenia Register-ScheduledJob.

15.10. Najczęściej spotykane problemy

Korzystanie z zadań zazwyczaj nie sprawia większych kłopotów, ale na szkoleniach czasami zdarzało się, że nasi studenci miewali problemy z jednym zagadnieniem. Nigdy nie postępuj w taki sposób:

```
PS C:\> invoke-command -command { Start-Job -scriptblock { dir } } -computername Server-R2
```



Rysunek 15.1. Wyniki działania zaplanowanego zadania zapisane na dysku

Próba wykonania takiego polecenia spowoduje uruchomienie tymczasowego połączenia z komputerem o nazwie SERVER-R2 i uruchomienie zadania lokalnego. Niestety, natychmiast po utworzeniu zadania połączenie zostanie zakończone, więc nie będziesz miał możliwości ponownego połączenia się z tą sesją i pobrania wyników tego zadania. Ogólnie rzecz ujmując, nigdy nie powinieneś mieszać i próbować dopasowywać do siebie opisywanych wcześniej trzech sposobów uruchamiania zadań.

A oto kolejny nie najlepszy pomysł:

```
PS C:\> start-job -scriptblock { invoke-command -command { dir } -computername SERVER-R2 }
```

Takie polecenie jest niepotrzebnie nadmiarowe; zachowaj z niego tylko sekcję `Invoke-Command` i dodaj parametr `-AsJob`, aby uruchomić zadanie w tle.

Mniej zagmatwane, ale równie interesujące są pytania na temat zadań, które to pytania często zadają nasi studenci. Prawdopodobnie najważniejszym z nich jest „Czy możemy zobaczyć zadania uruchomione przez kogoś innego?”. Odpowiedź brzmi: „Nie, z wyjątkiem zaplanowanych zadań”. Normalne zadania funkcjonują w całości wewnątrz procesu powłoki PowerShell i chociaż można zobaczyć, że inny użytkownik uruchomił tę powłokę, nie można zobaczyć działania zadania uruchomionego wewnątrz tego procesu. To tak, jak w przypadku każdej innej aplikacji — możesz zobaczyć, że inny użytkownik używa na przykład edytora Microsoft Word, ale nie widzisz, jakie edytuje dokumenty, ponieważ istnieją one w całości wyłącznie w procesie Worda.

Poszczególne zadania „żyją” dopóty, dopóki otwarta jest dana sesja powłoki PowerShell. Po jej zamknięciu wszystkie zdefiniowane w niej zadania znikną. Definicja zadań nie jest przechowywana nigdzie poza powłoką PowerShell, więc ich istnienie jest uzależnione od działania procesu ich macierzystej powłoki.

Wyjątkiem od przedstawionej w poprzednim akapicie reguły są zaplanowane zadania programu Task Manager — każdy użytkownik z odpowiednimi uprawnieniami może je przeglądać, modyfikować, usuwać i pobierać wyniki ich działania, ponieważ są zapisywane na dysku. Pamiętaj, że definicje takich zadań wraz z wynikami są przechowywane w Twoim profilu użytkownika, więc modyfikowanie czy usuwanie takich plików zwykle wymaga uprawnień administratora systemu.

15.11. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie komputer działający pod kontrolą systemu Windows 8, Windows Server 2012 lub nowszego oraz z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Ćwiczenia przedstawione poniżej pomogą Ci zrozumieć, jak pracować z różnymi typami zadań w powłoce PowerShell. Pracując nad tymi ćwiczeniami, nie musisz tworzyć rozwiązań, które zmieści się w jednym wierszu — czasami znacznie łatwiej jest rozwiązać złożony problem, rozbijając go na szereg prostszych elementów składowych.

1. Utwórz jednorazowe zadanie działające w tle, którego celem będzie znalezienie wszystkich skryptów powłoki PowerShell znajdujących się na dysku C:. Każde zadanie, którego wykonanie może zająć dużo czasu, jest doskonałym kandydatem do uruchomienia w tle.
2. Załóżmy, że bardzo przydałaby Ci się możliwość zidentyfikowania wszystkich skryptów powłoki PowerShell przechowywanych na niektórych Twoich serwerach. Jak uruchomić polecenie z ćwiczenia 1. na grupie komputerów zdalnych?
3. Utwórz zadanie działające w tle, które będzie pobierało z dziennika systemowego Twojego komputera 25 najnowszych zdarzeń powiązanych z błędami i eksportowało je do pliku CliXML. Zadanie powinno być uruchamiane codziennie, od poniedziałku do piątku, o godzinie 6:00, tak aby wyniki były gotowe do przeglądania, kiedy przyjdiesz do pracy.
4. Jakiego polecenia użyjesz, aby pobrać wyniki działania zadania, i jak możesz spowodować, aby po zakończeniu pobierania wyniki nadal były dostępne?

15.12. Odpowiedzi

1. `Start-Job {dir c:\ -recurse -filter '*.psl'}`
2. `Invoke-Command -scriptblock {dir c:\ -recurse -filter *.psl} -computername`
 `↳(get-content computers.txt) -asjob`
3. `$Trigger=New-JobTrigger -At "6:00AM" -DaysOfWeek "Monday", "Tuesday",`
 `↳"Wednesday","Thursday","Friday" -Weekly`

```
$command={ Get-EventLog -LogName System -Newest 25 -EntryType Error  
            | Export-Clixml c:\work\25SysErr.xml}  
Register-ScheduledJob -Name "Get 25 System Errors" -ScriptBlock  
$Command -Trigger $Trigger
```

sprawdzenie, co zostało utworzone

```
Get-ScheduledJob | Select *
```

4. Receive-Job -id 1 -keep

Oczywiście zamiast naszego ID równego 1 powinniśmy wstawić odpowiedni identyfikator zadania.

16

Praca z wieloma obiektami jednocześnie

Przeznaczeniem powłoki PowerShell jest automatyzacja zadań związanych z zarządzaniem systemami, co często oznacza, że będziesz chciał wykonywać określone zadania na wielu obiektach docelowych. Przykładowo, możesz zrestartować kilka komputerów, zmienić konfigurację kilku usług, zmodyfikować kilka skrzynek pocztowych i tak dalej. W tym rozdziale nauczysz się trzech różnych technik wykonywania zadań operujących na wielu obiektach docelowych. Są to polecenia wsadowe, metody WMI (i innych obiektów) oraz wyliczanie obiektów. Pamiętaj, że większość przykładów w tym rozdziale nie będzie działać w systemie Linux ani macOS — wszystkie przykłady przeznaczone są (przynajmniej obecnie) tylko dla systemu Windows. Warto jednak zauważyć, że omawiane tutaj pojęcia, koncepty i techniki pozostają takie same, niezależnie od tego, jakiego systemu operacyjnego używasz.

16.1. Automatyzacja zarządzania wieloma obiektami docelowymi

Doskonale zdajemy sobie sprawę z tego, że nie jest to książka o programowaniu w języku VBScript, ale chcemy użyć przykładowego skryptu napisanego w tym języku, aby krótko zilustrować sposób, w jaki zarządzanie wieloma obiektami docelowymi (które Don lubi określać mianem **zarządzania masowego**) było realizowane w przeszłości. Przyjrzyj się uważnie temu przykładowi (nie musisz wpisywać tego kodu ani go uruchamiać — chcemy tutaj omówić samo podejście do zagadnienia, a nie wyniki działania skryptu):

```
For Each varService in colServices
    varService.ChangeStartMode("Automatic")
Next
```

Takie podejście jest powszechnie stosowane nie tylko w języku VBScript, ale praktycznie w całym świecie programowania. Poniżej zamieszczamy krótki opis sposobu działania tego fragmentu kodu:

1. Zakładamy, że zmienna `colServices` zawiera kolekcję wielu usług. Nie ma znaczenia, jak się tam znalazły, ponieważ można to zrobić na wiele różnych sposobów. Ważne jest to, że pobrałeś już odpowiednie usługi i umieściłeś je w tej zmiennej.
2. Pętla `ForEach` przechodzi przez wszystkie usługi i w każdej iteracji umieszcza w zmiennej `varService` kolejną usługę z kolekcji. Wewnątrz ciała pętli zmienna `varService` zawiera tylko jedną usługę jednocześnie. Przykładowo, jeżeli w kolekcji `colServices` znajduje się 50 usług, kod wewnątrz pętli będzie wykonywany 50 razy, a w każdej iteracji zmienna `varService` będzie zawierała inną z 50 usług.
3. Wewnątrz pętli w każdej iteracji wykonywany jest odpowiedni kod realizujący zadanie; w naszym przykładzie jest to metoda `ChangeStartMode()`.

Jeżeli się nad tym dobrze zastanowisz, uświadomisz sobie, że przedstawiony fragment kodu nie wykonuje żadnej operacji na wszystkich usługach jednocześnie. Zamiast tego odpowiednia metoda jest wywoływana dla jednej usługi w tym samym czasie, niemal dokładnie tak, jakbyś to robił ręcznie, rekonfigurując poszczególne usługi przy użyciu graficznego interfejsu użytkownika. Jedyna różnica polega na tym, że poszczególne usługi są kolejno modyfikowane przez komputer, po jednej jednocześnie.

Komputery znakomicie radzą sobie z wykonywaniem w kółko tych samych, nudnych i powtarzalnych zadań, więc nie ma w tym niczego złego. Problem polega na tym, że takie podejście wymaga od nas przekazywania komputerowi dość skomplikowanego zestawu instrukcji. Nauka języka niezbędnego do wydania takiego zestawu instrukcji może zająć trochę czasu, dlatego wielu administratorów próbuje unikać używania języka VBScript i innych języków skryptowych.

W przypadku powłoki PowerShell również możemy pisać skrypty w opisany wyżej sposób i w dalszej części tego rozdziału pokażemy, jak to zrobić, ponieważ czasami trzeba uciec się do tej metody. Jednak podejście polegające na tym, że komputer po kolei przetwarza obiekty z kolekcji, zdecydowanie nie jest najskuteczniejszym sposobem użycia powłoki PowerShell. W rzeczywistości powłoka ta oferuje dwie inne techniki, które nie dość, że są łatwiejsze do opanowania i „wpisania w komputer”, to jeszcze do tego są często bardziej wydajne.

16.2. Preferowany sposób: polecenia „wsadowe”

Jak już wspominaliśmy w kilku poprzednich rozdziałach, wiele poleceń powłoki PowerShell może wykonywać swoje operacje na grupach lub inaczej mówiąc — na *kolekcjach* obiektów. Przykładowo, w rozdziale 6. dowiedziałeś się, że obiekty mogą być przesyłane za pomocą potoku z wyjścia jednego polecenia bezpośrednio na wejście drugiego polecenia, tak jak to zostało pokazane poniżej (uwaga: nie uruchamiaj tego polecenia — może ono spowodować awarię systemu!):

```
Get-Service | Stop-Service
```


Jest to przykład wsadowego wykonywania operacji przy użyciu odpowiedniego polecenia powłoki. W tym przypadku polecenie `Stop-Service` zostało specjalnie zaprojektowane do pobierania jednego lub więcej obiektów reprezentujących usługi bezpośrednio z potoku, a następnie ich zatrzymywania. Polecenia `Set-Service`, `Stop-Process`, `Move-ADObject` czy `Move-Mailbox` są przykładami poleceń, które pobierają jeden lub więcej obiektów wejściowych, a następnie wykonują odpowiednią akcję na każdym z nich. Nie ma tutaj żadnej potrzeby wyliczania obiektów za pomocą pętli, tak jak to zrobiliśmy w przykładzie ze skryptem w języku VBScript w poprzednim podrozdziale. Powłoka PowerShell „wie”, jak pracować z kolekcjami obiektów, i może obsługiwać je za pomocą znacznie mniej złożonej składni.

Te tak zwane **polecenia wsadowe** (to nasza nazwa dla nich, a nie oficjalne określenie) to nasz preferowany sposób wykonywania operacji na wielu obiektach docelowych. Załóżmy, że musimy zmienić tryb uruchamiania trzech usług. Zamiast używać podejścia takiego jak w skryptach VBScript, możemy to zrobić w następujący sposób:

```
Get-Service -name BITS,Spooler,W32Time | Set-Service -startuptype Automatic
```

W pewnym sensie polecenie `Get-Service` jest także rodzajem polecenia wsadowego, ponieważ jest w stanie pobierać usługi z wielu komputerów. Przyjmijmy, że musisz zmienić te same trzy usługi na trzech różnych komputerach:

```
Get-Service -name BITS,Spooler,W32Time -computer Server1,Server2,Server3 |  
Set-Service -startuptype Automatic
```

Jedną z potencjalnych wad takiego podejścia jest to, że polecenia wykonujące interesujące nas operacje często nie generują żadnych wyników, które wskazują, że wykonały swoją pracę. Inaczej mówiąc, nie będziesz miał żadnego wizualnego powiadomienia, że dane polecenie zrobiło to, co do niego należało, a to może być dosyć frustrujące. Na szczęście polecenia takie często posiadają parametr `-passThru` powodujący, że wszystkie obiekty, które pojawiły się na wejściu polecenia, są przekazywane na jego wyjście. Po użyciu takiego parametru na wyjściu polecenia `Set-Service` pojawiają się te same usługi, które zostały przez to polecenie zmodyfikowane, dzięki czemu będziemy je ponownie mogli pobrać przy użyciu polecenia `Get-Service` i sprawdzić, czy zmiana weszła w życie.

Oto przykład użycia parametru `-passThru` z innym poleceniem:

```
Get-Service -name BITS -computer Server1,Server2,Server3 |  
Start-Service -passthru |  
Out-File NewServiceStatus.txt
```

Powyższe polecenie pobiera określoną usługę z trzech komputerów. Zebrane usługi są następnie przesyłane do polecenia `Start-Service`, które nie tylko je uruchamia, ale także przekazuje zaktualizowane obiekty na swoje wyjście, gdzie są przejmowane przez kolejne polecenie, zapisujące w pliku tekstowym zaktualizowane informacje o stanie tych usług.

Jeszcze raz przypominamy — to jest nasz sugerowany sposób pracy z powłoką PowerShell. Jeżeli istnieje gotowe polecenie, które potrafi wykonać potrzebną Ci operację,

powinieneś go użyć. W idealnym scenariuszu wszystkie polecenia powinny być dostosowane do pracy z wieloma obiektami, ale niestety nie zawsze tak jest (choć projektanci cmdletów starają się tworzyć polecenia jak najlepiej dostosowane do potrzeb użytkowników i wciąż dbają o to, aby je sukcesywnie ulepszać).

16.3. Wykorzystanie mechanizmów CIM/WMI — wywoływanie metod

Niestety nie zawsze mamy pod ręką gotowe polecenia, które mogą zrobić to, czego potrzebujemy. Jest to prawda, zwłaszcza jeżeli chodzi o komponenty, którymi możemy zarządzać za pomocą Windows Management Instrumentation (WMI, o którym mówiliśmy w rozdziale 14.).

UWAGA W tym podrozdziale przedstawimy krótką historię, która ma pomóc Ci w zrozumieniu, w jaki sposób użytkownicy korzystają z powłoki PowerShell. Niektóre rzeczy mogą wydawać się nieco nadmiarowe i omawiane już wcześniej, ale mimo to spróbuj przez to przebrnąć razem z nami raz jeszcze — samo nabieranie doświadczenia jest bardzo cenne.

Na przykład przyjrzyjmy się klasie Win32_NetworkAdapterConfiguration WMI, która reprezentuje określoną konfigurację karty sieciowej (karty sieciowe mogą mieć wiele zestawów konfiguracji, ale na razie założmy, że mają tylko jedną konfigurację, co jest powszechnie stosowane na komputerach klienckich). Przyjmijmy, że naszym celem jest włączenie DHCP na wszystkich kartach sieciowych Intel naszego komputera — nie chcemy włączać żadnych kart RAS ani innych wirtualnych adapterów sieciowych.

Możemy zacząć od próby wysłania zapytania o konfigurację kart sieciowych, co w naszym przypadku pozwoliło uzyskać następujące wyniki:

```
DHCPEnabled      : False
IPAddress        : {192.168.10.10, fe80::ec31:bd61:d42b:66f}
DefaultIPGateway :
DNSDomain        :
ServiceName      : E1G60
Description      : Intel(R) PRO/1000 MT Network Connection
Index            : 7
DHCPEnabled      : True
IPAddress        :
DefaultIPGateway :
DNSDomain        :
ServiceName      : E1G60
Description      : Intel(R) PRO/1000 MT Network Connection
Index            : 12
```

Aby osiągnąć takie wyniki, musimy wysłać zapytanie do odpowiedniej klasy WMI i przefiltrować wyniki tak, aby uwzględniały tylko konfigurację kart, w których opisie występuje słowo *Intel*. Możemy to zrobić przy użyciu następującego polecenia (zauważ, że w składni filtra WMI znak % odgrywa rolę symbolu wieloznacznego):

```
PS C:\> gwmi win32_networkadapterconfiguration -filter "description like '%intel%'"
```

ZRÓB TO SAM Gorąco zachęcamy Cię do samodzielnego wykonywania poleceń, które omawiamy w tym podrozdziale. Być może będziesz musiał nieco zmodyfikować niektóre polecenia, żeby je dostosować do swojego systemu. Na przykład jeżeli na Twoim komputerze nie ma żadnych kart sieciowych firmy Intel, będziesz musiał odpowiednio zmienić kryteria filtra.

Mając obiekty reprezentujące konfigurację poszczególnych kart w potoku, możemy włączyć na nich DHCP (widać, że na jednej z naszych kart DHCP nie jest włączone). Możemy zacząć od szukania polecenia o nazwie zbliżonej do Enable-DHCP. Niestety nie znajdziemy takiego polecenia, ponieważ po prostu powłoka PowerShell go nie ma. Nie istnieją żadne polecenia *cmdlet*, które mogą bezpośrednio obsługiwać wiele obiektów WMI jednocześnie.

Następnym krokiem jest sprawdzenie, czy sam obiekt konfiguracji ma metodę, która jest w stanie włączyć DHCP. Aby się tego dowiedzieć, przesyłamy obiekty konfiguracyjne do polecenia Get-Member (lub jego aliasu Gm):

```
PS C:\> gwmi win32_networkadapterconfiguration -filter "description like '%intel%'" | gm
```

Zaraz na początku listy powinniśmy zobaczyć metodę, której szukamy — EnableDHCP():

```
TypeName: System.Management.ManagementObject#root\cimv2\Win32_NetworkAdapterConfiguration
Name                                     MemberType   Definition
----                                     -
DisableIPSec                           Method       System.Management.ManagementB...
EnableDHCP                             Method       System.Management.ManagementB...
EnableIPSec                             Method       System.Management.ManagementB...
EnableStatic                           Method       System.Management.ManagementB...
```

Następnym krokiem, który często próbuje wykonać wielu początkujących użytkowników powłoki PowerShell, jest przekazywanie za pomocą potoku obiektów konfiguracyjnych bezpośrednio do tej metody:

```
PS C:\> gwmi win32_networkadapterconfiguration -filter "description like '%intel%'"
↪ | EnableDHCP()
```

Niestety to nie zadziała. Nie można przekazywać obiektów do metody za pomocą potoku; w ten sposób możesz przekazywać obiekty tylko do poleceń. EnableDHCP nie jest cmdletem powłoki PowerShell, a jest raczej swego rodzaju akcją bezpośrednio związaną z samym obiektem reprezentującym konfigurację karty sieciowej. Stare podejście w stylu skryptów języka VBScript byłoby podobne do przykładu, który pokazaliśmy na początku tego rozdziału, ale dzięki powłoce PowerShell możesz zrobić coś znacznie prostszego.

Chociaż nie mamy „wsadowego” polecenia o nazwie Enable-DHCP, do wywołania dowolnej metody WMI możemy użyć ogólnego polecenia Invoke-WmiMethod. To polecenie zostało specjalnie zaprojektowane do pobierania wielu obiektów WMI, takich jak nasze obiekty Win32_NetworkAdapterConfiguration, i wywoływania jednej z metod tych obiektów. Oto polecenie, którego użyjemy:

```
PS C:\> gwmi win32_networkadapterconfiguration
-filter "description like '%intel%'" |
Invoke-WmiMethod -name EnableDHCP
```

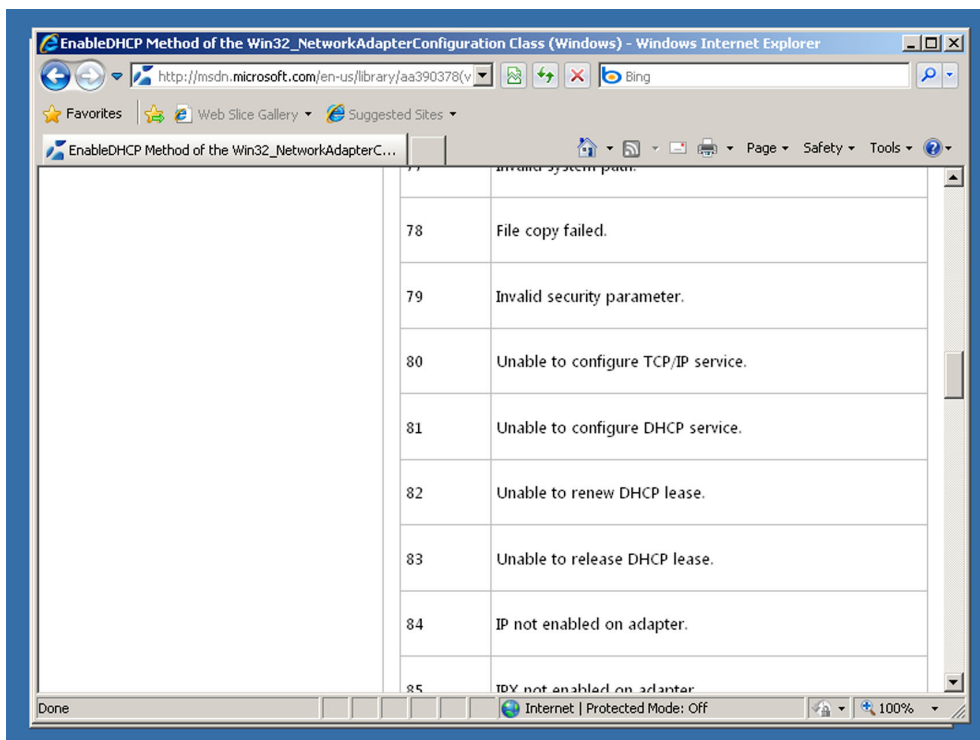
Musisz pamiętać o kilku rzeczach:

- Po nazwie metody nie umieszczamy nawiasów.
- W nazwach metod nie jest rozróżniana wielkość liter.
- Polecenie `Invoke-WmiMethod` może pobierać tylko jeden rodzaj obiektów WMI w tej samej chwili. W tym przypadku wysyłamy tylko obiekty `Win32_NetworkAdapterConfiguration`, co oznacza, że nasze polecenie będzie działać zgodnie z oczekiwaniami. Oczywiście możemy przysyłać więcej niż jeden obiekt (w końcu o to przecież tutaj chodzi), ale wszystkie obiekty muszą być tego samego typu.
- Można używać opcji `-WhatIf` i `-Confirm` polecenia `Invoke-WmiMethod`, z tym że nie możemy ich stosować podczas bezpośredniego wywoływania metody obiektu.

Wyniki działania polecenia `Invoke-WmiMethod` mogą być trochę mylące. WMI zawsze tworzy obiekt wynikowy, który posiada wiele właściwości systemowych (ich nazwy zaczynają się od dwóch znaków podkreślenia). W naszym przypadku wyniki działania tego polecenia są następujące:

```
__GENUS           : 2
__CLASS           : __PARAMETERS
__SUPERCLASS     : 
__DYNASTY        : __PARAMETERS
__RELPATH        : 
__PROPERTY_COUNT  : 1
__DERIVATION     : {}
__SERVER        : 
__NAMESPACE     : 
__PATH          : 
ReturnValue       : 0
__GENUS           : 2
__CLASS           : __PARAMETERS
__SUPERCLASS     : 
__DYNASTY        : __PARAMETERS
__RELPATH        : 
__PROPERTY_COUNT  : 1
__DERIVATION     : {}
__SERVER        : 
__NAMESPACE     : 
__PATH          : 
ReturnValue       : 84
```

Użyteczną informacją w tych wynikach działania jest jedyna właściwość, której nazwa nie rozpoczyna się od dwóch znaków podkreślenia: `ReturnValue`. Liczba będąca wartością tej właściwości reprezentuje wynik całej operacji. Szybkie poszukiwania klasy `Win32_NetworkAdapterConfiguration` w wyszukiwarce Google pozwalają na odkrycie i wyświetlenie strony z jej dokumentacją, gdzie możemy przejść do metody `EnableDHCP` i sprawdzić, jakie wartości zwraca i jakie jest ich znaczenie. Na rysunku 16.1 pokazujemy fragment wyników naszych poszukiwań.



Rysunek 16.1. Poszukiwanie opisów wartości zwracanych przez metodę WMI

Wartość 0 oznacza pomyślne zakończenie działania, natomiast wartość 84 wskazuje, że protokół IP nie jest włączony w tej konfiguracji karty i nie można włączyć DHCP. Ale jak dopasować poszczególne wyniki do naszych dwóch konfiguracji kart sieciowych? Trudno powiedzieć, ponieważ wyniki działania nie wskazują, który obiekt konfiguracyjny je wygenerował. Jest to irytujące, ale niestety tak działa WMI.

Polecenie `Invoke-WmiMethod` funkcjonuje w większości sytuacji, gdzie istnieje obiekt WMI posiadający metodę, którą chcesz wykonać. Działa to poprawnie nawet wtedy, kiedy zapytanie pobiera obiekty WMI również z komputerów zdalnych. Nasza podstawowa reguła brzmi: „Jeżeli możesz uzyskać dostęp do obiektu za pomocą polecenia `Get-WmiObject`, to również polecenie `Invoke-WmiObject` będzie mogło wywołać metody takiego obiektu”.

Jeżeli przypomnisz sobie, czego nauczyłeś się w rozdziale 14., szybko zorientujesz się, że polecenia `Get-WmiObject` i `Invoke-WmiMethod` są „starszymi” cmdletami, przeznaczonymi do pracy z WMI; ich następcami są polecenia `Get-CimInstance` oraz `Invoke-CimMethod`, które działają mniej więcej w ten sam sposób:

```
PS C:\> Get-CimInstance -classname win32_networkadapterconfiguration -filter "description
like '%intel%'" | Invoke-CimMethod -methodname EnableDHCP
```

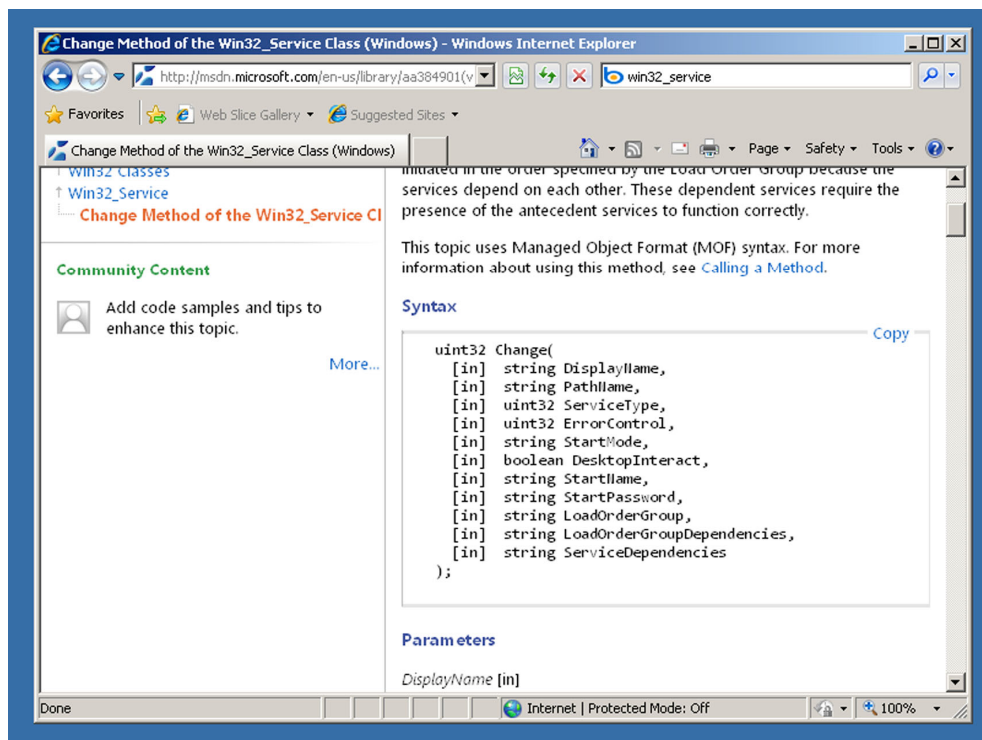
W rozdziale 14. pokazywaliśmy, kiedy należy korzystać z WMI, a kiedy z CIM. Nasze sugestie mają zastosowanie i tutaj: WMI działa z najszerszą gamą komputerów (obecnie), chociaż wymaga trudniejszych do skonfigurowania na zaporach sieciowych połączeń RPC; CIM wykorzystuje znacznie nowszy i łatwiejszy do skonfigurowania protokół WS-MAN, ale za to nie jest domyślnie instalowany w starszych wersjach systemu Windows.

Ale poczekaj — jest jeszcze jedna rzecz. W tym podrozdziale omawiamy WMI, a przecież w rozdziale 14. wspomnieliśmy, że firma Microsoft zrobiła naprawdę wiele, aby ukryć warstwę WMI przed użytkownikiem poprzez opakowywanie kluczowych funkcjonalności WMI w odpowiednie polecenia powłoki PowerShell (no dobrze, technicznie rzecz biorąc, są to funkcjonalności CIM, ale jesteśmy wystarczająco blisko). Przykładowo, spróbuj z poziomu powłoki PowerShell wykonać polecenie `Help Set-NetIPAddress`. W nowszych wersjach systemu Windows znajdziesz w ten sposób szereg doskonałych poleceń, które „ukrywają” przed użytkownikiem znaczną część zawiłości mechanizmów WMI/CIM. W takiej sytuacji zamiast wykonywać te wszystkie opisane wcześniej operacje WMI/CIM, do zmiany adresu IP moglibyśmy użyć tego jednego, prostego polecenia. To jedna z naprawdę cennych lekcji: nawet jeżeli przeczytałeś w sieci Internet o tym, jak coś zrobić, nie zakładaj z góry, że powłoka PowerShell v3 lub v4 (lub inne) nie oferuje jakiegoś lepszego rozwiązania. Niestety bardzo wiele informacji opublikowanych w Internecie opiera się na dokumentacji powłoki PowerShell w wersjach v1 i v2, a przecież możliwości powłoki w wersji 3 (i nowszych) są wielokrotnie większe.

16.4. Plan B — wyliczanie obiektów

Niestety natknęliśmy się na kilka sytuacji, w których polecenie `Invoke-WmiMethod` nie było w stanie wykonać wybranej metody i zwracało dziwne komunikaty o błędach (polecenie `Invoke-CimMethod` jest bardziej niezawodne). Zdarzały się też sytuacje, gdzie użytkownik miał cmdlet mogący wytwarzać obiekty, ale nie znał odpowiedniego polecenia wsadowego, do którego można by było hurtem przekazać te obiekty w celu wykonania jakiegoś działania. W obu przypadkach będziesz mógł nadal osiągnąć zamierzony cel, ale będziesz musiał powrócić do starego podejścia w stylu języka VBScript, instruując komputer, aby po kolei pobierał obiekty z kolekcji i wykonywał odpowiednie operacje na jednym obiekcie jednocześnie. Powłoka PowerShell pozwala na zrobienie tego na dwa sposoby: pierwszy z nich wymaga użycia odpowiedniego cmdletu, a drugi wykorzystuje rozwiązanie skryptowe. W tym rozdziale skoncentrujemy się na tej pierwszej technice, a o drugiej opowiemy w rozdziale 21., w którym będziemy omawiać zagadnienia związane z wbudowanym językiem skryptowym powłoki PowerShell.

W naszym przykładzie ilustrującym, jak to zrobić, użyjemy klasy WMI o nazwie `Win32_Service`, a w szczególności użyjemy jej metody `Change()`. Jest to złożona metoda, która może zmienić kilka elementów usługi jednocześnie. Rysunek 16.2 pokazuje fragment dokumentacji online (którą znaleźliśmy, wpisując w wyszukiwarce sieciowej nazwę klasy `Win32_Service`, a następnie klikając metodę `Change`). Kiedy zapoznasz się z tą



Rysunek 16.2. Strona dokumentacji metody `Change()` klasy `Win32_Service`

dokumentacją, szybko odkryjesz, że nie musisz podawać wszystkich parametrów metody — dla dowolnych parametrów, które chcesz pominąć, możesz podać wartość `Null` (która w powłoce PowerShell jest reprezentowana za pomocą wbudowanej zmiennej o nazwie `$null`).

W naszym przykładzie chcemy zmienić hasło logowania usługi, które jest ósmym parametrem wywołania metody. Aby to zrobić, dla pierwszych siedmiu parametrów musimy podać wartość `$null`. Oznacza to, że wywołanie naszej metody może wyglądać tak:

```
Change($ null, $ null, $ null, $ null, $ null, $ null, $ null, "P@ssw0rd")
```

Nawiasem mówiąc, ani polecenie `Get-Service`, ani polecenie `Set-Service` nie mogą wyświetlać ani ustawiać hasła logowania usługi. Ale można to zrobić za pomocą WMI, więc używamy WMI.

Ponieważ nie możemy użyć polecenia wsadowego `Set-Service`, które zwykle byłoby naszym preferowanym rozwiązaniem, wypróbujemy nasze drugie podejście, które polega na użyciu polecenia `Invoke-WmiMethod`. Polecenie to ma parametr `-ArgumentList`, w którym możemy podać argumenty dla wywoływanej metody. Poniżej przedstawiamy zapis naszej próby wraz z wynikami działania, które otrzymaliśmy:

```
PS C:\> gwmi win32_service -filter "name = 'BITS'" | invoke-wmimethod -name change -arg
↳ $null,$null,$null,$null,$null,$null,$null,$null,"P@ssw0rd"
Invoke-WmiMethod : Input string was not in a correct format.
At line:1 char:62
+ gwmi win32_service -filter "name = 'BITS'" | invoke-wmimethod <<<< -name change -arg
↳ $null,$null,$null,$null,$null,$null,$null,$null,"P@ssw0rd"
+ CategoryInfo          : NotSpecified: (:) [Invoke-WmiMethod], FormatException
+ FullyQualifiedErrorId : System.FormatException,Microsoft.PowerShell
.Commands.InvokeWmiMethod
```

UWAGA W naszym przykładzie używamy polecenia `Get-WmiObject`, ale polecenie `Get-CimInstance` ma praktycznie taką samą składnię.

W tym momencie musimy podjąć decyzję. Możliwe, że jednak używamy tego polecenia w nieprawidłowy sposób, więc musimy się zastanowić, czy nie warto by było poświęcić chwili czasu na zapoznanie się z jego dokumentacją. Teoretycznie istnieje również możliwość, że polecenie `Invoke-WmiMethod` nie współpracuje zbyt dobrze z metodą `Change()`, a w takim przypadku możemy zmarnować dużo czasu, próbując naprawić coś, nad czym nie mamy kontroli.

Naszym typowym wyborem w takich sytuacjach jest wypróbowanie innego podejścia: zamierzamy poprosić komputer (no dobrze, może nie sam komputer, a po prostu powłokę PowerShell), aby w pętli przeszedł przez odpowiednie obiekty usług i dla każdego z nich po kolei wykonał metodę `Change()`. Aby to zrobić, użyjemy cmdletu `ForEach-Object`:

```
PS C:\> gwmi win32_service -filter "name = 'BITS'" |
foreach-object {$_.change($null,$null,$null,$null,$null,$null,$null,$null,"P@ssw0rd")}
__GENUS                : 2
__CLASS                 : __PARAMETERS
__SUPERCLASS            :
__DYNASTY                : __PARAMETERS
__RELPATH               :
__PROPERTY_COUNT        : 1
__DERIVATION             : {}
__SERVER                :
__NAMESPACE             :
__PATH                  :
ReturnValue              : 0
```

Według dokumentacji wartość właściwości `ReturnValue` równa 0 wskazuje na pomyślne wykonanie polecenia, a zatem osiągnęliśmy nasz cel. Spójrzmy jednak na to polecenie bardziej szczegółowo, zmieniając nieco formatowanie, żeby całość była bardziej przejrzysta:

```
Get-WmiObject Win32_Service -filter "name = 'BITS'" |
ForEach-Object -process {
    $_.change($null,$null,$null,$null,$null,$null,$null,$null,"P@ssw0rd")
}
```

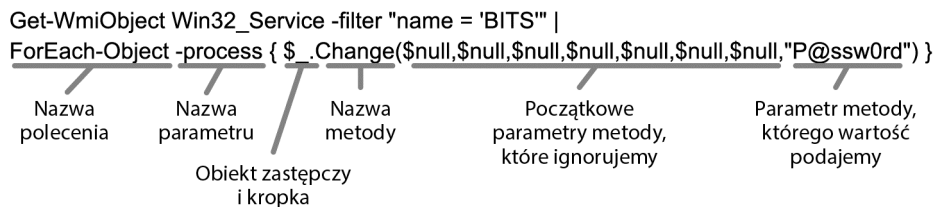
W tym poleceniu naprawdę wiele się dzieje. Pierwszy wiersz jest dosyć oczywisty: używamy polecenia `Get-WmiObject` do pobierania wszystkich instancji klasy `Win32_Service` pasujących do podanych kryteriów filtrowania, które poszukują usługi o nazwie `BITS` (jak zwykle w takich przykładach wybieramy usługę `BITS`, ponieważ jest ona mniej ważna

niż wiele innych, które mogliśmy wybrać, a jej przypadkowe zatrzymanie nie spowoduje awarii naszego komputera). Otrzymane obiekty `Win32_Service` przekazujemy za pomocą potoku do cmdletu `ForEach-Object`.

Spróbujemy teraz rozłożyć to polecenie na jego elementy składowe:

- Wiersz polecenia zaczynamy od nazwy cmdletu: `ForEach-Object`.
- Następnie używamy parametru `-process` do zdefiniowania bloku skryptu. W pierwszym przykładzie nie wpisywaliśmy nazwy tego parametru, ponieważ jest to parametr pozycyjny. Musimy jednak pamiętać, że cały blok skryptu, czyli wszystko, co jest zawarte w nawiasach klamrowych, jest wartością parametru `-process`, dlatego w kolejnym przykładzie, dla zwiększenia czytelności polecenia, zdecydowaliśmy się na użycie pełnej nazwy tego parametru.
- Polecenie `ForEach-Object` wykonuje cały blok skryptu osobno dla każdego obiektu przekazanego na wejście tego polecenia. W każdej iteracji pętli kolejny obiekt otrzymany za pośrednictwem potoku jest umieszczany w specjalnym obiekcie zastępczym `$_`.
- Po nazwie obiektu zastępczego (`$_`) następuje kropka i nazwa właściwości lub metody bieżącego obiektu, do której chcemy uzyskać dostęp.
- W naszym przykładzie używamy metody o nazwie `Change()`. Zwróć uwagę, że parametry metody są przekazywane jako lista wartości oddzielonych od siebie przecinkami i umieszczona w nawiasach. Dla wszystkich parametrów, których nie chcemy zmienić, wstawiamy zmienną `$null`, a jako ósmy parametr podajemy nowe hasło. Metoda może pobierać więcej parametrów, ale ponieważ nie chcemy zmieniać dziewiątego, dziesiątego ani jedenastego parametru, możemy całkowicie je pominąć (aczkolwiek równie dobrze możemy wstawić wartość `$null` dla tych trzech ostatnich parametrów).

Jak widać, składnia całego polecenia jest dosyć złożona — dla lepszego zilustrowania prezentujemy ją jeszcze raz, na rysunku 16.3.



Rysunek 16.3. Składnia polecenia `ForEach-Object`

Tego samego wzorca postępowania możesz użyć dla dowolnej metody WMI. Dlaczego więc miałbyś zamiast tego używać polecenia `Invoke-WmiMethod`? Cóż, to polecenie zwykle działa całkiem nieźle, jest łatwiejsze w użyciu, a cały wiersz polecenia z jego zastosowaniem jest bardziej przejrzysty. Jeżeli jednak wolisz zapamiętać tylko jeden sposób wykonywania takich operacji, to metoda z użyciem polecenia `ForEach-Object` sprawdzi się bardzo dobrze.

Musimy Cię ostrzec, że przykłady, które możesz znaleźć w sieci Internet, często są o wiele mniej czytelne. Różni guru powłoki PowerShell często używają aliasów, parametrów pozycyjnych i skróconych nazw parametrów, co znacznie zmniejsza czytelność całego polecenia (ale oszczędza też pisanie na klawiaturze). Dla przykładu pokazujemy to samo polecenie, ale zapisane w mocno skróconej formie:

```
PS C:\> gwmi win32_service -fi "name = 'BITS'" |
% {$_ .change($null,$null,$null,$null,$null,$null,$null,"P@ssw0rd") }
```

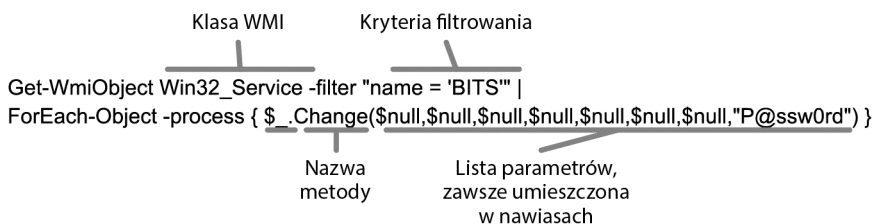
Przyjrzyjmy się zmianom w tym zapisie:

- Zamiast polecenia `Get-WmiObject` używamy aliasu `Gwmi`.
- Nazwę parametru `-filtr` skracamy do `-fi`.
- Zamiast polecenia `ForEach-Object` używamy znaku `%`. Tak, znak `%` jest aliasem tego polecenia. Naszym zdaniem użycie tego aliasu powoduje, że polecenie staje się mało czytelne, ale wielu użytkowników z niego korzysta.
- Usuwamy nazwę parametru `-process`, ponieważ jest to parametr pozycyjny.

Nie lubimy używać aliasów i skróconych nazw parametrów w skryptach, które umieszczamy w naszych blogach i udostępniamy innym użytkownikom, ponieważ sprawia to, że stają się trudniejsze do przeanalizowania. Jeżeli zamierzasz zapisać coś w pliku skryptu, warto poświęcić trochę czasu na wpisanie pełnych nazw poleceń i parametrów (lub po prostu korzystać z mechanizmu dopełniania poleceń, który posiada przecież powłoka PowerShell).

Jeśli kiedykolwiek będziesz chciał użyć tego przykładu w swoim środowisku, możesz zmienić kilka rzeczy (jak pokazano na rysunku 16.4):

- Zmień nazwę klasy WMI i podaj odpowiednie kryteria filtru tak, aby pobrać żądane obiekty WMI.
- Zmień nazwę metody `Change` na nazwę dowolnej innej metody, którą chcesz wykonać.
- Zmodyfikuj listę parametrów metody (zwaną również **argumentami**) tak, aby odpowiadała wymaganiom wywołania danej metody. Jest to zawsze lista wartości oddzielonych od siebie przecinkami i umieszczona w nawiasach. W przypadku wywoływania metod, które nie posiadają żadnych parametrów wywołania, nawiasy mogą być całkowicie puste (na przykład dla metody `EnableDHCP()`, którą omawialiśmy we wcześniejszej części tego rozdziału).



Rysunek 16.4. Zmiany w poleceniu, których musisz dokonać, aby wywołać inną metodę

Czy takie podejście było najlepszym sposobem na osiągnięcie założonego przez nas celu? Przeglądając zawartość pliku pomocy polecenia `Set-Service`, widzimy, że nie oferuje ona żadnego sposobu na zmianę haseł, co jednak możemy wykonać zarówno przy użyciu polecenia `Get-WmiObject`, jak i `Get-CimInstance`. Można stąd wyciągnąć wniosek, że nawet w przypadku stosowania powłoki PowerShell v3 takie zadanie musimy wykonać przy użyciu WMI.

16.5. Najczęściej spotykane problemy

Techniki opisywane w tym rozdziale należą do najtrudniejszych technik powłoki PowerShell i często sprawiają wiele problemów podczas naszych szkoleń. Przyjrzymy się teraz niektórym problemom, z którymi borykają się nasi studenci, i spróbujemy udzielić dodatkowych wyjaśnień, które pomogą Ci uniknąć tych samych problemów w Twojej pracy.

16.5.1. Który sposób postępowania jest właściwy?

W naszym przykładach używamy określenia *polecenie wsadowe*, odwołując się do dowolnego polecenia *cmdlet* wykonującego operację na grupie czy, inaczej mówiąc, na kolekcji wielu obiektów jednocześnie. W przypadku takich poleceń zamiast instruowania komputera, aby „przeglądał listę obiektów i wykonywał jedną akcję na każdym z tych obiektów po kolei”, możesz od razu wysłać całą grupę obiektów do cmdletu, który zajmie się resztą.

Firma Microsoft robi duże postępy w dostarczaniu tego typu poleceń dla swoich produktów, ale niestety daleko jeszcze do tego, aby takie możliwości miały wszystkie polecenia (i prawdopodobnie z taką sytuacją będziemy musieli sobie jeszcze radzić przez wiele lat ze względu na liczbę złożonych produktów tej firmy). Jeżeli jednak istnieje odpowiednie polecenie, które może wykonać interesujące nas zadanie, to z pewnością będziemy chcieli go użyć. Nie zmienia to jednak w niczym faktu, że wielu deweloperów preferuje różne alternatywne sposoby postępowania w zależności od tego, czego sami kiedyś się nauczyli i co wydaje im się najwygodniejszym rozwiązaniem danego zadania. Aby to zilustrować, poniżej przedstawiamy kilka różnych wariantów poleceń, które wykonują dokładnie takie samo zadanie: zatrzymują działanie wszystkich usług, których nazwy rozpoczynają się na literę B:

```
Get-Service -name *B* | Stop-Service ← ❶ Polecenie wsadowe
Get-Service -name *B* | ForEach-Object { $_.Stop() } ← ❷ Polecenie ForEach-Object
Get-WmiObject Win32_Service -filter "name LIKE '%B%'" | ← ❸ WMI
    Invoke-WmiMethod -name StopService
Get-WmiObject Win32_Service -filter "name LIKE '%B%'" | ← ❹ WMI i polecenie ForEach-Object
    ForEach-Object { $_.StopService() }
Stop-Service -name *B* ← ❺ Polecenie Stop-Service
```

Spójrzmy, jak działa każde z tych poleceń:

- Pierwszy sposób polega na użyciu polecenia wsadowego ❶. W tym przypadku korzystamy z polecenia `Get-Service`, za pomocą którego pobieramy wszystkie usługi z nazwą rozpoczynającą się na literę B i następnie przekazujemy je do polecenia `Stop-Service`, aby je zatrzymać.

- Drugie podejście jest podobne. Zamiast używać polecenia wsadowego, przekazujemy otrzymane obiekty do polecenia `ForEach-Object`, które wywołuje metodę `.Stop()` kolejnych obiektów ❷.
- Trzecia technika polega na wykorzystaniu WMI zamiast natywnych cmdletów powłoki PowerShell ❸. Za pomocą polecenia `Get-WmiObject` pobieramy żądane usługi (których nazwy rozpoczynają się na literę *B*) i przekazujemy je do polecenia `Invoke-WmiMethod`, które wywołuje metodę `StopService` obiektów WMI.
- Czwarty sposób zamiast polecenia `Invoke-WmiMethod` wykorzystuje pętlę `ForEach-Object`, która podobnie jak w poprzednim przypadku wywołuje metodę `StopService` obiektów WMI ❹. Jest to swego rodzaju połączenie metod ❷ i ❸, a nie zupełnie nowy sposób postępowania.
- Piąta technika bezpośrednio wykorzystuje polecenie `Stop-Service` ❺, ponieważ jego parametr `-Name` pozwala na stosowanie symboli wieloznacznych (w przypadku powłoki PowerShell v3 i nowszych).

W zasadzie istnieje jeszcze nawet szóste podejście — użycie języka skryptowego powłoki PowerShell do wykonania takiego zadania. Jak widać, pracując z powłoką PowerShell, możesz wykonywać poszczególne zadania na wiele różnych sposobów i żaden z nich nie jest zły. Niektóre z nich są jednak łatwiejsze do nauczenia się, zapamiętania i powtórzenia, dlatego skupiliśmy się na opisywaniu technik w preferowanej przez nas kolejności.

Nasze przykłady ilustrują również istotne różnice między sposobem użycia natywnych poleceń powłoki i poleceń WMI:

- Kryteria filtrowania w natywnych poleceniach powłoki używają znaku `*` jako symbolu wieloznacznego, podczas gdy przy filtrowaniu WMI używamy znaku procentu (`%`). Pamiętaj, aby nie mylić tego znaku z aliasem polecenia `ForEach-Object`. W przypadku kryteriów filtrowania WMI znak `%` występuje w wartościach parametru `-filter` polecenia `Get-WmiObject` i w żadnym razie nie jest to alias.
- Obiekty często mają metody podobne do klas WMI, ale ich składnia wywołania może być różna. W naszym przykładzie obiekty `ServiceController` utworzone przez polecenie `Get-Service` posiadają metodę `Stop()`; kiedy jednak uzyskujemy dostęp do tych samych usług za pośrednictwem klasy WMI `Win32_Service`, odpowiadającą jej metoda nosi nazwę `StopService()`.
- W kryteriach filtrowania powłoki PowerShell wykorzystujemy natywne operatory porównania, takie jak `-eq`; WMI stosuje operatory używane w językach programowania, takie jak `=` czy `LIKE`.

Które podejście powinieneś najczęściej stosować? Na to pytanie nie ma jednoznacznej odpowiedzi, bo nie można wskazać tylko jednej, właściwej drogi. W praktyce możesz wykonywać określone zadania na wiele różnych sposobów, a nawet używać ich kombinacji w zależności od tego, jakie zadanie chcesz wykonać i jakimi narzędziami dysponuje powłoka.

16.5.2. Metody WMI a polecenia powłoki

Kiedy należy używać metod WMI, a kiedy poleceń *cmdlet*? Tutaj odpowiedź jest prosta:

- Jeżeli pobierasz obiekty za pomocą polecenia `Get-WmiObject`, podejmij odpowiednie działania, używając metod WMI. Dostępne metody możesz uruchamiać za pomocą polecenia `Invoke-WmiMethod` lub pętli `ForEach-Object`.
- Jeżeli pobierasz obiekty za pomocą innych poleceń niż `Get-WmiObject`, do wykonania zadania użyj odpowiedniego, natywnego polecenia powłoki. Jeżeli nie znajdziesz takiego polecenia, ale pobrane obiekty będą posiadać odpowiednie metody, do ich wywołania możesz użyć polecenia `ForEach-Object`.

Zauważ, że wspólnym mianownikiem tych rozwiązań jest polecenie `ForEach-Object` — jego składnia jest być może najtrudniejsza do opanowania, ale zawsze możesz go użyć do wykonania praktycznie każdego zadania.

Pamiętaj, że za pomocą potoku nie możesz przekazywać niczego bezpośrednio do metody. W taki sposób można przekazywać obiekty wyłącznie z jednego polecenia do drugiego. Jeżeli polecenie, które mogłoby wykonać określone zadanie, nie istnieje, ale pobrane obiekty posiadają odpowiednie metody, możesz przekazać takie obiekty za pomocą potoku do polecenia `ForEach-Object` i zlecić wywołanie tej metody.

Załóżmy, że pobierasz obiekty przy użyciu fikcyjnego polecenia `Get-Something` (pobierz-coś). Chciałbyś to „coś” usunąć, ale powłoka nie ma odpowiedniego polecenia `Delete-Something` (skasuj-coś) czy `Remove-Something` (usuń-coś). Zauważyłeś jednak, że obiekty `Something` mają metodę `Delete`. W takiej sytuacji możesz ją wywołać w następujący sposób:

```
Get-Something | ForEach-Object { $_.Delete() }
```

16.5.3. Dokumentacja metod

Pamiętaj, że aby sprawdzić dostępne metody obiektów, możesz przekazać je za pomocą potoku do polecenia `Get-Member`. W naszym przykładzie ponownie użyjmy fikcyjnego polecenia `Get-Something`:

```
Get-Something | Get-Member
```

Wbudowany system pomocy PowerShell nie zawiera żadnej dokumentacji metod WMI; aby znaleźć opisy i przykłady użycia takich metod, musisz skorzystać z wyszukiwarki sieciowej (zazwyczaj jako kryterium wyszukiwania powinieneś wpisać nazwę klasy WMI). W systemie pomocy powłoki PowerShell nie znajdziesz również opisów metod innych obiektów, które nie mają nic wspólnego z WMI. Na przykład jeżeli wyświetlisz listę elementów składowych obiektu usługi, przekonasz się, że posiadają one metody `Stop` i `Start`:

```
TypeName: System.ServiceProcess.ServiceController
Name      MemberType Definition
-----
Name      AliasProperty      Name = ServiceName
RequiredServices AliasProperty      RequiredServices = ServicesDepe...
```

Disposed	Event	System.EventHandler Disposed(Sy...
Close	Method	System.Void Close()
Continue	Method	System.Void Continue()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef ...
Dispose	Method	System.Void Dispose()
Equals	Method	bool Equals(System.Object obj)
ExecuteCommand	Method	System.Void ExecuteCommand(int ...
GetHashCode	Method	int GetHashCode()
GetLifetimeService	Method	System.Object GetLifetimeService()
GetType	Method	type GetType()
InitializeLifetimeService	Method	System.Object InitializeLifetim...
Pause	Method	System.Void Pause()
Refresh	Method	System.Void Refresh()
Start	Method	System.Void Start(), System.Voi...
Stop	Method	System.Void Stop()
ToString	Method	string ToString()
WaitForStatus	Method	System.Void WaitForStatus(Syste...

Aby znaleźć dokumentację tych metod, powinieneś skupić się na typie obiektu (właściwość `TypeName`), który w tym przypadku nosi nazwę `System.ServiceProcess.ServiceController`. Wpisz pełną nazwę typu obiektu w wyszukiwarce sieciowej, a zwykle natkniesz się na oficjalną dokumentację takich obiektów dla programistów i deweloperów, gdzie zazwyczaj znajdziesz dokumentację interesującej Cię konkretnej metody.

16.5.4. Najczęstsze problemy z poleceniem *ForEach-Object*

Polecenie `ForEach-Object` ma dosyć złożoną składnię, wymagającą dużej uwagi, a dołożenie do tego odpowiedniej składni wywoływania różnych metod może spowodować, że wiersz polecenia będzie wyglądał naprawdę paskudnie. Aby ułatwić Ci życie, przygotowaliśmy kilka porad, jak sobie z tym radzić:

- Zamiast aliasów `%` lub `ForEach` zawsze staraj się używać pełnej nazwy polecenia. Takie postępowanie powoduje, że całe polecenie staje się bardziej przejrzyste i łatwiejsze do przeanalizowania. Jeżeli używasz cudzych fragmentów kodu, przed uruchomieniem polecenia staraj się pozamieniać wszystkie aliasy na pełne nazwy poleceń.
- Blok skryptu ujęty w nawiasy klamrowe wykonuje się po jednym razie dla każdego obiektu przekazanego za pomocą potoku do polecenia `ForEach-Object`.
- W bloku skryptu wyrażenie `$_` reprezentuje jeden z obiektów przekazanych za pomocą potoku.
- Wyrażenia `$_` używaj do pracy z całymi obiektami; wyrażenia `$_.` (z kropką) używaj do pracy z poszczególnymi metodami lub właściwościami obiektów.
- Po nazwach metod zawsze występuje para nawiasów okrągłych, nawet jeżeli dana metoda nie wymaga podawania żadnych parametrów wywołania. Jeżeli metoda wymaga podania określonych parametrów, poszczególne wartości są oddzielane od siebie przecinkami, a cała lista jest zawarta w nawiasach.

16.6. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Spróbuj samodzielnie odpowiedzieć na poniższe pytania i wykonać powiązane z nimi zadania. Jest to bardzo ważny zestaw ćwiczeń, ponieważ opiera się na wiedzy, którą zdobyłeś podczas pracy z poprzednimi rozdziałami tej książki. Pamiętaj, że kluczem do sukcesu jest konsekwentne pogłębianie swojej wiedzy i wykorzystywanie swoich umiejętności w praktyce.

1. Jaka metoda obiektu `ServiceController` (będącego wynikiem działania polecenia `Get-Service`) może wstrzymać działanie usługi bez całkowitego jej zatrzymywania?
2. Jaka metoda obiektu `Process` (będącego wynikiem działania polecenia `Get-Process`) może zakończyć działanie danego procesu?
3. Jaka metoda obiektu WMI klasy `Win32_Process` może zakończyć działanie danego procesu?
4. Napisz cztery polecenia, które mogą zostać użyte do zakończenia działania wszystkich procesów o nazwie `Notepad`, przy założeniu, że jednocześnie może działać wiele procesów o tej nazwie.
5. Załóżmy, że w pliku tekstowym masz listę nazw komputerów, ale chcesz wyświetlać wszystkie nazwy przy użyciu wielkich liter. Jakie polecenie powłoki PowerShell możesz zastosować?

16.7. Odpowiedzi

1. Aby wyświetlić listę dostępnych metod, możesz wykonać następujące polecenie:
`get-service | Get-Member -MemberType Method`
 W wynikach działania powinieneś znaleźć metodę `Pause()`.
2. Aby wyświetlić listę dostępnych metod, możesz wykonać następujące polecenie:
`get-process | Get-Member -MemberType Method`
 W wynikach działania powinieneś znaleźć metodę `Kill()`. Możesz to zweryfikować, sprawdzając dokumentację MSDN dla obiektów typu `Process`. Oczywiście zazwyczaj nie będziesz musiał wywoływać tej metody, ponieważ istnieje polecenie `Stop-Process`, które może wykonać to za Ciebie.
3. Poszukaj dokumentacji MSDN dla klasy `Win32_Process`. Do wyświetlenia listy metod możesz również użyć poleceń CIM, ponieważ one również współpracują z WMI, na przykład:
`Get-CimClass win32_process | select -ExpandProperty methods`
 Wynikiem Twoich poszukiwań (niezależnie od użytego sposobu) powinna być metoda `Terminate()`.
4. `get-process Notepad | stop-process stop-process -name Notepad`
`get-process notepad | foreach {$_.Kill()}`
`Get-WmiObject win32_process -filter {name='notepad.exe'} |`
`Invoke-WmiMethod -Name Terminate`
5. `Get-content computers.txt | foreach {$_.ToUpper()}`

17

Bezpieczeństwo wykonywania skryptów

Po przeczytaniu kilkunastu poprzednich rozdziałów tej książki prawdopodobnie zdajesz sobie już sprawę z tego, jak ogromne możliwości ma powłoka PowerShell, i zapewne zastanawiasz się, czy ta cała jej moc może stanowić jakiś problem z punktu widzenia bezpieczeństwa systemu. *Zdecydowanie tak*. Naszym celem w tym rozdziale jest pomóc Ci zrozumieć, jaki wpływ na bezpieczeństwo Twojego środowiska może mieć powłoka PowerShell i jak ją skonfigurować, aby zapewnić równowagę między wymaganiami bezpieczeństwa a możliwościami użytkowania. Warto zauważyć, że prawie wszystko w tym rozdziale dotyczy powłoki PowerShell dla systemu Windows — systemy macOS i Linux nie obsługują większości omawianych mechanizmów, ponieważ sposób działania powłoki na tych platformach jest zupełnie inny.

17.1. Zapewnienie bezpieczeństwa powłoki PowerShell

Pod koniec 2006 roku, kiedy pierwsza wersja powłoki PowerShell pojawiła się na rynku, firma Microsoft nie miała najlepszych doświadczeń z bezpieczeństwem rozwiązań skryptowych. W końcu VBScript i środowisko Windows Script Host (WSH) były w owym czasie jednymi z najpopularniejszych wektorów ataku dla wirusów i złośliwego oprogramowania, wykorzystywanych między innymi przez takie niesławne wirusy jak I Love You, Melissa i wiele innych. Jesteśmy więc pewni, że kiedy zespół deweloperów PowerShella ogłosił, że pracuje nad utworzeniem nowej powłoki wiersza poleceń, która będzie posiadała niespotykane do tej pory możliwości i ogromną funkcjonalność, a w dodatku będzie wyposażona w możliwości tworzenia i wykonywania skryptów, wszędzie odezwały się syreny alarmowe, w siedzibach wielu firm komputerowych ogłoszono ewakuację i wszyscy zgrzytali zębami z przerażenia.

Ale okazało się, że wszystko jest OK. Powłoka PowerShell została utworzona już po ogłoszeniu przez Billa Gatesa słynnego programu Trustworthy Computing Initiative, który miał na celu zwiększenie bezpieczeństwa aplikacji przez ulepszenie procesów tworzenia ich kodu i stawiał na pierwszym miejscu bezpieczeństwo rozwoju i użytkowania oprogramowania. Program przynosił wymierne efekty, wymuszając między innymi, aby każdy zespół deweloperów miał wykwalifikowanego eksperta do spraw bezpieczeństwa oprogramowania, który uczestniczy we wszystkich spotkaniach projektowych, przeglądach kodu i tak dalej. W wewnętrznym języku firmy Microsoft taki *ekspert do spraw bezpieczeństwa produktu* jest nazywany — to nie jest nasz wymysł — *The Security Buddy* (kumpel od bezpieczeństwa).

Ekspert do spraw bezpieczeństwa powłoki PowerShell (ang. *PowerShell's Security Buddy*) był jednocześnie jednym z autorów książki *Writing Secure Code* (tworzenie bezpiecznego kodu), będącej swego rodzaju wewnętrzną „biblią” firmy Microsoft opisującą zasady tworzenia bezpiecznego oprogramowania, które jest mniej podatne na ataki. Możesz być pewny, że powłoka PowerShell jest produktem bezpiecznym, a przynajmniej, że firma Microsoft zrobiła wszystko, aby takie domyślne bezpieczeństwo jej zapewnić. Oczywiście w każdej chwili możesz zmienić ustawienia domyślne, ale zanim to zrobisz, powinieneś rozważyć, jakie będą tego konsekwencje w zakresie bezpieczeństwa, a nie tylko funkcjonalności, i właśnie w tym spróbujemy Ci tutaj pomóc.

17.2. Model bezpieczeństwa powłoki Windows PowerShell

Musimy mieć jasność względem tego, co powłoka PowerShell może i czego nie może, jeżeli chodzi o bezpieczeństwo, a najlepszym sposobem na to jest zdefiniowanie niektórych celów jej bezpieczeństwa.

Przede wszystkim powłoka PowerShell nie nakłada żadnych dodatkowych warstw uprawnień na komponenty, z którymi wchodzi w interakcję. PowerShell pozwoli Ci wykonywać tylko to, do czego już masz odpowiednie uprawnienia. Jeżeli z poziomu konsoli graficznej nie możesz tworzyć nowych użytkowników w usłudze Active Directory, nie będziesz mógł tego zrobić również z poziomu powłoki PowerShell. Inaczej mówiąc, powłoka PowerShell to po prostu kolejny sposób korzystania z posiadanych uprawnień.

Powłoka PowerShell nie jest również sposobem na ominięcie jakichkolwiek istniejących uprawnień. Załóżmy, że zamierzasz wdrożyć nowy skrypt dla użytkowników i chcesz, aby ten skrypt wykonywał operacje, do których wykonania Twoi użytkownicy nie mają odpowiednich uprawnień. Taki skrypt nie zadziała. Jeżeli chcesz, aby Twoi użytkownicy mogli wykonać określone zadanie, musisz dać im do tego uprawnienia; powłoka PowerShell może wykonywać tylko takie operacje, do których użytkownik wykonujący polecenie lub skrypt ma już odpowiednie uprawnienia.

Model bezpieczeństwa powłoki PowerShell nie został zaprojektowany po to, aby uniemożliwiać czy utrudniać użytkownikom wykonywanie zadań, do których wykonania mają uprawnienia. Chodzi o to, że dosyć trudno jest skłonić użytkownika do wpisywania długich, złożonych poleceń, stąd powłoka PowerShell nie stosuje tutaj żadnych dodatkowych zabezpieczeń poza uprawnieniami użytkownika. Ale z doświadczenia

wiemy, że o wiele łatwiej jest nakłonić użytkownika do uruchomienia skryptu, który mógłby zawierać potencjalnie szkodliwe polecenia. Właśnie dlatego większość zabezpieczeń powłoki PowerShell została zaprojektowana, aby uniemożliwić użytkownikom niezamierzone, przypadkowe uruchamianie skryptów. Kluczowym elementem jest tutaj słowo *niezamierzone*, ponieważ zabezpieczenia powłoki PowerShell nie mają przecież na celu zupełnego zablokowania skryptów — chodzi o to, aby chronić użytkownika przed możliwością mniej lub bardziej przypadkowego uruchamiania skryptów pochodzących z niezaufanych źródeł.

Dla zainteresowanych

Choć wykracza to daleko poza zakres tej książki, to jednak chcielibyśmy, abyś poznał również *inne* sposoby zezwalania użytkownikom na uruchamianie skryptów działających z innymi poświadczeniami logowania niż ich własne. Zazwyczaj można to osiągnąć za pomocą techniki nazywanej **pakowaniem skryptów**, która jest dostępna w niektórych komercyjnych środowiskach programowania skryptów, takich jak SAPIEN PowerShell Studio (zobacz <https://www.sapien.com/>).

Po utworzeniu skryptu można użyć specjalnego programu pakującego, który „kompiluje” skrypt do postaci pliku wykonywalnego (.EXE). Nie jest to kompilacja w dosłownym, programistycznym tego słowa znaczeniu — tak „skompilowany” plik wykonywalny nie jest całkowicie autonomiczny i do jego uruchomienia wymagana jest obecność zainstalowanej powłoki PowerShell. W takim scenariuszu zazwyczaj możesz skonfigurować program pakujący do osadzenia w pliku wykonywalnym zaszyfrowanych poświadczeń alternatywnych.

Dzięki takiemu rozwiązaniu, kiedy ktoś uruchomi tak spreparowany plik wykonywalny, spakowany skrypt zostanie uruchomiony z użyciem osadzonych poświadczeń logowania, a nie w kontekście użytkownika, który go uruchomił.

Takie „zapakowane” poświadczenia logowania nie są jednak w 100% bezpieczne — w końcu pakiet zawiera osadzone: nazwę użytkownika i hasło dostępu, chociaż zwykle są one w taki czy inny sposób zaszyfrowane bądź przynajmniej zakodowane. Możemy zatem bezpiecznie założyć, że większość użytkowników nie zdoła odczytać nazwy użytkownika i hasła, ale jest całkowicie możliwe, że wykwalifikowany specjalista z dziedziny kryptografii czy informatyki śledczej będzie jednak w stanie takie poświadczenia odszyfrować.

Mechanizmy zabezpieczające powłoki PowerShell nie stanowią również ochrony przed złośliwym oprogramowaniem (ang. *malware*). Gdy masz już złośliwe oprogramowanie w swoim systemie, to może ono zrobić wszystko, do czego masz uprawnienia. Malware może używać powłoki PowerShell do wykonywania złośliwych poleceń, ale równie dobrze może wykorzystywać dowolną z tuzina innych technik pozwalających na uszkodzenie danych bądź nawet samego komputera. Jeżeli złośliwe oprogramowanie pojawiło się w Twoim systemie, jesteś w poważnych tarapatach, a powłoka PowerShell w żadnej mierze nie jest kolejną linią obrony i nadal będziesz potrzebował solidnego oprogramowania antywirusowego, aby zapobiegać przedostawaniu się szkodliwego oprogramowania do Twojego systemu. Jest to niezwykle ważna koncepcja, której wielu użytkowników nie rozumie: nawet jeżeli jakiś wirus czy inne złośliwe oprogramowanie użyje powłoki PowerShell do wyrządzenia szkody w systemie, to nadal nie oznacza to, że winę za to ponosi sama powłoka. Ochrona systemu przed wirusami i innymi złośliwymi

programami jest zadaniem Twojego oprogramowania antywirusowego. Powłoka PowerShell nie posiada żadnych mechanizmów, których zadaniem byłaby ochrona już raz skompromitowanego systemu.

17.3. Polityka wykonywania skryptów i podpisywanie kodu

Pierwszym mechanizmem bezpieczeństwa powłoki PowerShell jest polityka wykonywania skryptów (ang. *execution policy*). Jest to ustawienie dotyczące całego komputera, które określa, jakie skrypty mogą być wykonywane przez powłokę PowerShell. Jak już wspominaliśmy wcześniej w tym rozdziale, głównym celem tego ustawienia jest ochrona użytkownika przed możliwością przypadkowego uruchamiania skryptów pochodzących z niezaufanych źródeł.

17.3.1. Ustawienia polityki wykonywania skryptów

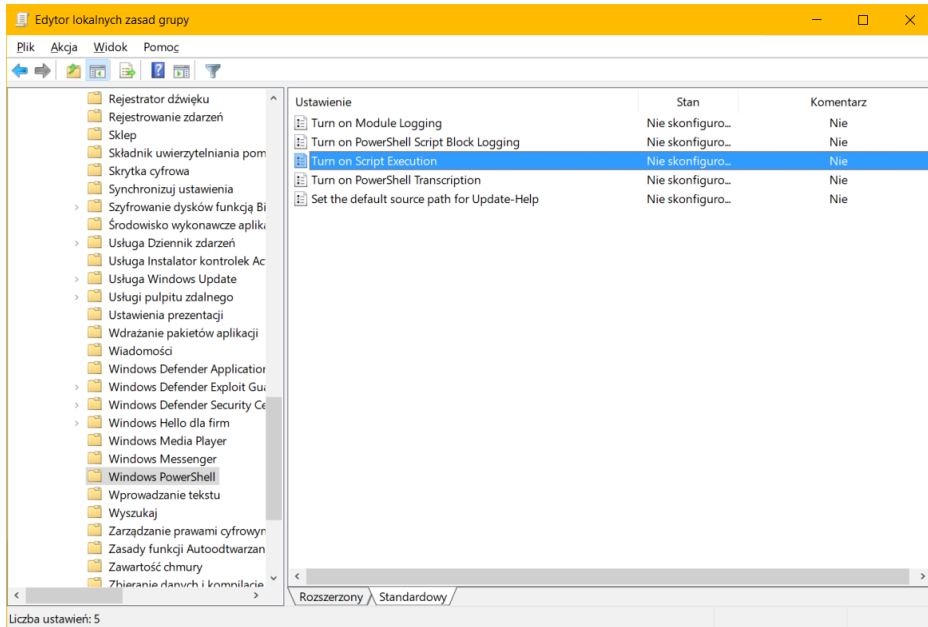
Domyślnie polityka wykonywania skryptów jest ustawiona na wartość *Restricted*, która w ogóle uniemożliwia wykonywanie skryptów. Zgadza się: domyślnie możesz używać powłoki PowerShell do interaktywnego uruchamiania poleceń, ale nie możesz tego robić do uruchamiania skryptów. Jeżeli spróbujesz to zrobić, na ekranie pojawi się następujący komunikat:

```
File C:\test.ps1 cannot be loaded because the execution of scripts is disabled
on this system. Please see "get-help about_signing" for more details. At line:1 char:7
+ ./test <<<<
    + CategoryInfo          : NotSpecified: (:) [], PSSecurityException
    + FullyQualifiedErrorId : RuntimeException
```

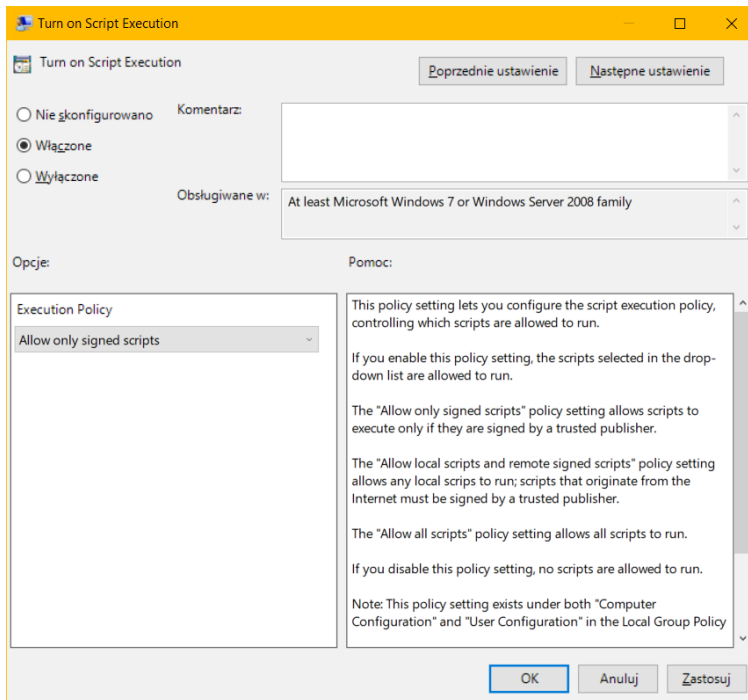
Bieżącą politykę wykonywania skryptów można wyświetlić, uruchamiając polecenie `Get-ExecutionPolicy`. W zależności od potrzeb możesz zmienić te ustawienia na jeden z trzech sposobów:

- *Za pomocą polecenia* `Set-ExecutionPolicy` — wykonanie tego polecenia zmienia ustawienie polityki wykonywania skryptów w kluczu `HKEY_LOCAL_MACHINE` rejestru systemu Windows i zazwyczaj musi być uruchamiane przez administratora, ponieważ zwykli użytkownicy nie mają uprawnień do zapisu w tej części rejestru.
- *Korzystając z ustawień zasad grupy (GPO)* — obsługa ustawień GPO związanych z powłoką Windows PowerShell została wprowadzona począwszy od systemu Windows Server 2008 R2. Jeżeli z jakiegoś powodu utknąłeś w starszej domenie (nasze kondolencje), możesz poszukać odpowiedniego szablonu PowerShell ADM na stronie download.microsoft.com.

Jak pokazano na rysunku 17.1, ustawienia powłoki PowerShell znajdują się w gałęzi *Konfiguracja komputera/Szablony administracyjne/Składniki systemu Windows/Windows PowerShell*. Na rysunku 17.2 pokazano włączoną opcję wykonywania skryptów. Po skonfigurowaniu tego ustawienia w zasadach grupy zastępuje ono wszelkie ustawienia lokalne. W praktyce po skonfigurowaniu w GPO ustawień wykonywania skryptów możesz uruchomić polecenie `Set-ExecutionPolicy`, ale na ekranie pojawi się komunikat z ostrzeżeniem, że próba zmiany ustawień nie przyniosła efektu z powodu nadrzędnych ustawień zasad grupy.



Rysunek 17.1. Wyszukiwanie ustawień powłoki PowerShell w zasadach grupy



Rysunek 17.2. Zmiana polityki wykonywania skryptów powłoki PowerShell w zasadach grupy

- *Za pomocą ręcznego uruchomienia programu PowerShell.exe z przełącznikiem -ExecutionPolicy* — po uruchomieniu powłoki w ten sposób reguły wykonywania skryptów zdefiniowane w wierszu wywołania polecenia zastępują wszelkie ustawienia lokalne, jak również wszelkie ustawienia zdefiniowane w zasadach grupy.

Politykę wykonywania skryptów możesz ustawić na jednym z pięciu poziomów (zauważ, że w zasadach grupy masz dostęp tylko do trzech środkowych ustawień z poniższej listy):

- *Restricted* — jest to ustawienie domyślne, które powoduje, że uruchamianie skryptów jest blokowane. Jedynym wyjątkiem od tej reguły jest kilka skryptów dostarczonych przez firmę Microsoft, które modyfikują ustawienia konfiguracji powłoki PowerShell. Skrypty te są podpisane cyfrowo przez firmę Microsoft, dzięki czemu wprowadzenie do nich jakichkolwiek modyfikacji powoduje, że nie będą mogły zostać uruchomione.
- *AllSigned* — po wybraniu tego ustawienia powłoka PowerShell wykona każdy skrypt, który został podpisany cyfrowo przy użyciu certyfikatu wydanego przez zaufany urząd certyfikacji (CA).
- *RemoteSigned* — wybranie tego ustawienia powoduje, że powłoka PowerShell wykona każdy skrypt lokalny i skrypty zdalne, które zostały podpisane cyfrowo przy użyciu certyfikatu wydanego przez zaufany urząd certyfikacji. **Skrypty zdalne** to takie skrypty, które są dostępne za pośrednictwem ścieżki UNC (ang. *Universal Naming Convention*) na komputerach zdalnych. Skrypty oznaczone jako pochodzące z sieci Internet są również traktowane jako skrypty zdalne; przeglądarki sieciowe, takie jak Internet Explorer, Firefox, czy programy pocztowe (na przykład Outlook) w specjalny sposób oznaczają wszystkie pliki pobierane z sieci Internet. Niektóre wersje systemu Windows mogą rozróżniać ścieżki internetowe i ścieżki UNC; w takich przypadkach ścieżki UNC w sieci lokalnej nie są uważane za zdalne.
- *Unrestricted* — pozwala na uruchamianie wszystkich skryptów.
- *Bypass* — to specjalne ustawienie, które jest przeznaczone dla deweloperów, którzy osadzają powłokę PowerShell w swoich aplikacjach. To ustawienie pozwala na pominięcie ustawień aktualnej polityki wykonywania skryptów i powinno być używane tylko wtedy, gdy aplikacja hostująca zapewnia własną warstwę zabezpieczeń skryptów. Używając tego ustawienia, dajesz powłoce PowerShell jasny przekaz: „Nie przejmuj się. Sprawy bezpieczeństwa biorę na siebie!”.

Zaraz, zaraz, że co?

Czy zwróciłeś uwagę na to, że nawet jeżeli ustawisz politykę wykonywania skryptów w zasadach grupy, możesz ją nadpisać przy użyciu odpowiedniego parametru wywołania programu *PowerShell.exe*? Jakże więc zalety ma tworzenie ustawień kontrolowanych przez GPO, które użytkownicy mogą tak łatwo ominąć? Takie rozwiązanie dodatkowo podkreśla to, że polityka wykonywania skryptów ma tylko na celu ochronę *niefrasobliwych* użytkowników przed *niezamierzonym* uruchamianiem skryptów pochodzących z niezaufanych, *anonimowych* źródeł.

Polityka wykonywania skryptów nie ma na celu powstrzymywania świadomego użytkownika od wykonywania jakichkolwiek zamierzonych operacji. To nie jest tego rodzaju mechanizm.

W praktyce inteligentny autor złośliwego oprogramowania mógłby równie łatwo uzyskać dostęp do funkcjonalności platformy .NET Framework bezpośrednio, bez konieczności wykorzystywania powłoki PowerShell jako pośrednika — inaczej mówiąc, jeżeli nieautoryzowany użytkownik uzyska dostęp do Twojego komputera na poziomie uprawnień administratora i będzie w stanie uruchomić dowolny kod, to masz poważne kłopoty.

Jeżeli chcesz korzystać ze skryptów, firma Microsoft zaleca ustawienie polityki wykonywania na wartość `RemoteSigned`, ale tylko na komputerach, na których takie skrypty mają być uruchamiane. Zgodnie z zaleceniami Microsoftu na wszystkich innych komputerach powinieneś pozostawić poziom `Restricted`. Microsoft twierdzi, że poziom `RemoteSigned` zapewnia dobrą równowagę między bezpieczeństwem skryptów a wygodą użytkownika; poziom `AllSigned` jest jeszcze bardziej rygorystyczny i wymaga, aby wszystkie Twoje skrypty były podpisane cyfrowo. W obszernej społeczności użytkowników powłoki PowerShell opinie na ten temat są bardziej podzielone i wielu użytkowników ma nieco odmienne zdanie na temat dobrych ustawień polityki wykonywania skryptów. Na razie jednak pozostaniemy przy rekomendacjach firmy Microsoft i pozwolimy Ci odkrywać kulisy tego zagadnienia samodzielnie, jeżeli oczywiście zechcesz.

A skoro już o tym mowa, to nadszedł dobry moment, aby nieco bardziej szczegółowo omówić proces podpisywania cyfrowego.

UWAGA Wielu uznanych ekspertów, w tym dobrze znany i popularny *Scripting Guy* z firmy Microsoft, sugeruje ustawienie polityki wykonywania skryptów na poziomie `Unrestricted`. Swoje stanowisko uzasadniają tym, że mechanizm ten nie zapewnia żadnej dodatkowej warstwy bezpieczeństwa i w związku z tym nie powinienieś robić sobie nadziei, że będzie ona w stanie Cię przed czymkolwiek uchronić.

17.3.2. Cyfrowe podpisywanie kodu

Cyfrowe podpisywanie kodu lub w skrócie po prostu **podpisywanie kodu** (ang. *code signing*) to proces zastosowania podpisu kryptograficznego do pliku tekstowego. Podpisy pojawiają się na końcu pliku i wyglądają mniej więcej tak:

```
<!-- SIG # Begin signature block -->
<!-- MIIXAYJKoZIhvcNAQcQoIIXTCCF0kCAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB -->
<!-- gjcCAQSwWZBZMDQGCisGAQQBgcCAR4wJgIDAQAABBAfzDtqWUUsITrek0sYpfeNR -->
<!-- AgEAAgEAAgEAAgEAMCEwCQYFKw4DAh0FAAQUJ7qroHx47P11dlt4Bg6Y5Jo -->
<!-- UVigghIxMIEYDCCA0ygAwIBAgIKLqsR3FD/XJ3LwDAJBgUrDgMCHQUAMHAAxKzAp -->
<!-- YjcCn4FqI4n2XGOPsFq7OddgjFWEGjP1O5igggyiX4uzLLehpcur2iC2vzAZhSAU -->
<!-- DSq8UvRB4F4w45IoaYfBcOLzp6vOgEJydg4wggR6MIIDYqADAqECAGphBieBAAAA -->
<!-- Zn9nZui2t+ +Fuc3uqv0SpAtZiikvz0DZVgQbdrVtZG1KVNvd8d6/n4PHgN9/TAI3 -->
<!-- an/xcmG4PNGSdjy8Dcbb5otiSjgByprAttPPf2EKUQrFPzREgZabAatwMKJbeRS4 -->
<!-- kd6Qy+RwkcN1UWlEaChbs0LJhix0jm38/pLCCOo1nL79E1sxJumCe6GtqjdWOIBn -->
<!-- KKe66D/GX7eCrjCVg2Vzpg4gG7fHADFEh3Oclvo1LWc= -->
<!-- SIG # End signature block -->
```

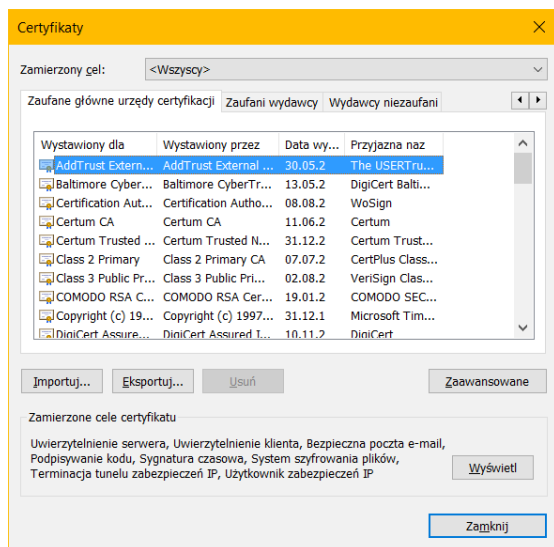
Podpis cyfrowy zawiera dwie ważne informacje: po pierwsze, są to dane identyfikujące tożsamość firmy lub organizacji, która podpisała skrypt; po drugie, jest to zaszyfrowana kopia skryptu, którą PowerShell może odszyfrować. Zrozumienie, jak to działa, wymaga nieco dodatkowych informacji, które pomogą Ci również podjąć ważne decyzje dotyczące bezpieczeństwa w Twoim środowisku.

Aby utworzyć podpis cyfrowy, musisz posiadać odpowiedni certyfikat do podpisywania kodu (ang. *code signing certificate*). Takie certyfikaty, nazywane również **certyfikatami cyfrowymi klasy 3** (ang. *Class 3 certificates*), są dostępne za pośrednictwem komercyjnych urzędów certyfikacji, takich jak DigiCert, GoDaddy, Thawte czy Verisign. Odpowiedni certyfikat możesz również uzyskać z wewnętrznej infrastruktury kłucza publicznego Twojej firmy (PKI — ang. *Public Key Infrastructure*), jeżeli oczywiście Twoja firma taką infrastrukturę posiada. Certyfikaty klasy 3. są zwykle wydawane tylko organizacjom i firmom, a nie osobom fizycznym, chociaż Twoja firma może je wewnętrznie wydawać również określonym użytkownikom. Przed wydaniem certyfikatu urząd certyfikacji jest odpowiedzialny za weryfikację tożsamości odbiorcy — certyfikat to rodzaj cyfrowej karty identyfikacyjnej, wymieniającej nazwisko posiadacza i inne dane. Na przykład przed wydaniem certyfikatu dla korporacji XY urząd certyfikacji musi zweryfikować, że żądanie wystawienia certyfikatu zostało wystawione przez upoważnionego przedstawiciela korporacji XYZ. Proces weryfikacji tożsamości jest najważniejszym krokiem w całym systemie zabezpieczeń. Określony urząd certyfikacji powinien traktować jako zaufany tylko wtedy, gdy wiesz, że przeprowadza szczegółową weryfikację tożsamości firm, którym wystawia certyfikaty. Jeżeli nie znasz procedur weryfikacyjnych danego urzędu certyfikacji, *nie powinieneś ufać* takiemu urzędowi.

W systemie Windows listę zaufanych urzędów certyfikacji możesz skonfigurować w oknie opcji programu Internet Explorer (możesz to również zrobić za pomocą zasad grupy). Po otwarciu okna właściwości przejdź na kartę *Zawartość*, a następnie naciśnij przycisk *Wydawcy*. Na ekranie pojawi się okno *Certyfikaty*. Przejdź na kartę *Zaufane główne urzędy certyfikacji*, gdzie zobaczysz listę urzędów certyfikacji, którym Twój komputer ufa (zobacz rysunek 17.3).

Ufając danemu urzędowi certyfikacji, obdarzasz zaufaniem także wszystkie wydane przez niego certyfikaty. Jeżeli ktoś używa certyfikatu do podpisania złośliwego skryptu, możesz użyć tego podpisu do wyśledzenia autora — dlatego podpisane skrypty są uważane za bardziej „zaufane” niż niepodpisane. Ale jeżeli zaufasz urzędowi, który nie dba o właściwą weryfikację tożsamości, autor złośliwego skryptu może otrzymać certyfikat, podając fałszywe dane, i w takiej sytuacji nie będziesz w stanie go zidentyfikować na podstawie podpisanego skryptu. Z tego właśnie względu wybór zaufanych urzędów certyfikacji to taka duża odpowiedzialność.

Po uzyskaniu certyfikatu klasy 3. (a w szczególności potrzebny Ci będzie pakiet spakowany jako tzw. certyfikat Authenticode — urzędy certyfikacji zwykle oferują różne rodzaje certyfikatów przeznaczonych dla różnych systemów operacyjnych i języków programowania) musisz zainstalować go na swoim komputerze. Po zainstalowaniu możesz rozpocząć cyfrowe podpisywanie skryptów, korzystając z polecenia `Set-AuthenticodeSignature`



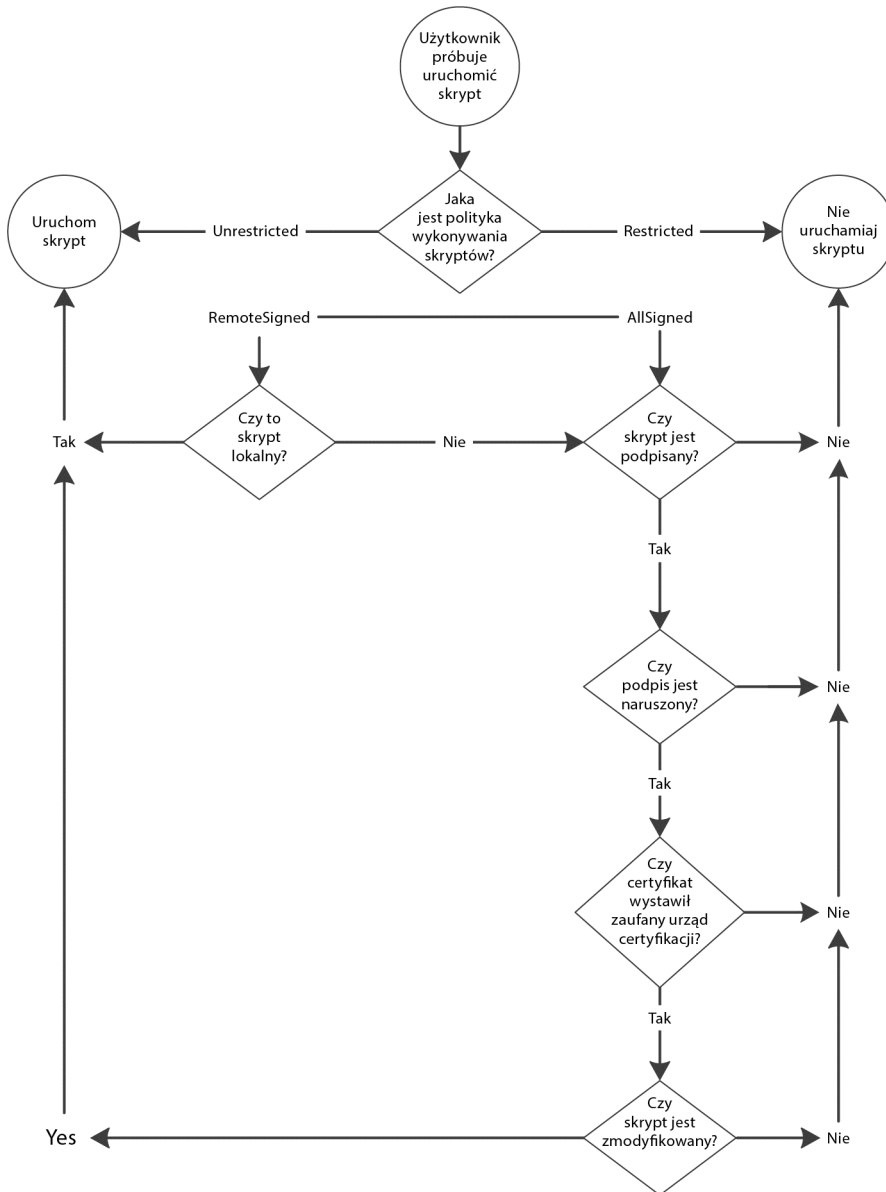
Rysunek 17.3. Konfigurowanie listy zaufanych głównych urzędów certyfikacji

powłoki PowerShell. Aby dowiedzieć się czegoś więcej na ten temat, powinieneś zapoznać się z zawartością pliku pomocy `about_signing`. Komercyjne środowiska programowania wspomagające tworzenie skryptów (PowerShell Studio, PowerShell Plus, PowerGUI i inne) również pozwalają na cyfrowe podpisywanie skryptów, a wiele z nich może to robić automatycznie podczas zapisywania skryptu, dzięki czemu proces podpisywania staje się bardziej wygodny.

Podpisy cyfrowe nie tylko dostarczają informacji o tożsamości autora skryptu, ale również zapewniają, że skrypt nie został zmodyfikowany od momentu jego podpisania przez autora. Działa to w następujący sposób:

1. Autor skryptu posiada certyfikat cyfrowy, który składa się z dwóch kluczy kryptograficznych: klucza publicznego i klucza prywatnego.
2. Podczas podpisywania skryptu podpis jest szyfrowany przy użyciu klucza prywatnego, do którego dostęp ma tylko autor skryptu. Zasyfrowany podpis może zostać odszyfrowany wyłącznie przy użyciu klucza publicznego. Podpis zawiera kopię skryptu.
3. Kiedy powłoka PowerShell podejmuje próbę uruchomienia skryptu, do odszyfrowania podpisu używa klucza publicznego autora, który to klucz jest dołączony do podpisu. Jeżeli odszyfrowanie się nie powiedzie, oznacza to, że podpis został zmodyfikowany, i w takiej sytuacji skrypt nie zostanie uruchomiony. Jeżeli kopia skryptu zawarta w podpisie nie jest zgodna z nieszyfrowaną wersją kodu, podpis jest uznawany za uszkodzony i skrypt również nie zostanie uruchomiony.

Rysunek 17.4 ilustruje cały proces weryfikacji, przez który przechodzi powłoka PowerShell podczas próby uruchomienia skryptu. Możesz się tutaj przekonać, dlaczego ustawienie polityki wykonywania skryptów na wartość `AllSigned` jest bezpieczniejszym



Rysunek 17.4. Proces weryfikacji podejmowany przez powłokę PowerShell przed uruchomieniem skryptu

rozwiązaniem — ponieważ wykonywane są wtedy tylko skrypty, które zostały podpisane cyfrowo, tak więc zawsze możesz zidentyfikować autora takiego skryptu. W takim scenariuszu musisz jednak podpisywać również wszystkie swoje skrypty, które chcesz uruchamiać, a co gorsza, musisz je ponownie podpisywać po wprowadzeniu każdej modyfikacji, a to już może być nieco uciążliwe.

17.4. Inne mechanizmy bezpieczeństwa

Powłoka PowerShell ma dwa inne kluczowe mechanizmy bezpieczeństwa, które działają przez cały czas i nie powinny być modyfikowane.

Po pierwsze, system Windows nie traktuje plików z rozszerzeniem *.PS1* (które jest używane przez powłokę do identyfikowania skryptów powłoki PowerShell) jako plików wykonywalnych. Dwukrotne kliknięcie pliku *.PS1* zamiast uruchamiać skrypt, zazwyczaj powoduje otwarcie go do edycji w Notatniku lub innym, domyślnym edytorze. Taka konfiguracja pomaga użytkownikom uniknąć przypadkowego, nieświadomego uruchomienia skryptu, nawet jeżeli polityka wykonywania skryptów na to zezwala.

Po drugie, nie możesz uruchomić skryptu z poziomu konsoli powłoki, wpisując w wierszu polecenia jego nazwę. Powłoka nigdy nie wyszukuje skryptów w bieżącym katalogu, co oznacza, że jeżeli masz skrypt o nazwie *test.ps1*, to przejście do jego folderu i wpisanie „polecenia” *test* czy nawet *test.ps1* nie spowoduje uruchomienia takiego skryptu.

Oto przykład:

```
PS C:\> test
```

```
The term 'test' is not recognized as the name of a cmdlet, function, scrip t file,
↳or operable program. Check the spelling of the name, or if a path was included, verify
↳that the path is correct and try again.
At line:1 char:5
+ test <<<<
+ CategoryInfo          : ObjectNotFound: (test:String) [], CommandNo tFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
Suggestion [3,General]: The command test was not found, but does exist in t he current
↳location. Windows PowerShell doesn't load commands from the curr ent location by default.
↳If you trust this command, instead type ".\test". See "get-help about_Command_Precedence"
↳for more details.
PS C:\>
```

Jak widać, powłoka PowerShell wykrywa nasz skrypt, ale ostrzega, że aby go uruchomić, należy podać jego względną lub bezwzględną ścieżkę. Ponieważ skrypt znajduje się w głównym katalogu dysku C:, można go uruchomić, wpisując polecenie *C:\test* (ścieżka bezwzględna) lub *.\test*, które jest ścieżką względną wskazującą na bieżący folder.

Celem takiego mechanizmu jest zabezpieczenie przed atakami typu *command hijacking*. Taki atak polega na umieszczeniu złośliwego skryptu w wybranym folderze i nadaniu mu tej samej nazwy co wbudowane polecenie, na przykład *Dir*. Pracując z powłoką PowerShell, nigdy nie umieszczasz ścieżki przed nazwą polecenia — jeżeli w wierszu polecenia wpiszesz *Dir*, wiesz, że uruchamiasz polecenie; jeżeli wpiszesz *.\Dir*, wiesz, że uruchamiasz skrypt o nazwie *Dir.ps1*.

17.5. Inne luki w zabezpieczeniach?

Jak wspominaliśmy wcześniej w tym rozdziale, mechanizmy bezpieczeństwa powłoki PowerShell koncentrują się przede wszystkim na zapobieganiu nieświadomemu uruchamianiu niezaufanych skryptów przez użytkownika. Żadne zabezpieczenia nie mogą

jednak powstrzymać użytkownika od ręcznego wpisywania komend w wierszu poleceń powłoki, kopiowania fragmentów lub całych skryptów i wklejania ich do powłoki (choć niektóre polecenia mogą nie działać tak samo, gdy są uruchamiane w ten sposób). Mimo wszystko znacznie trudniej jest przekonać użytkownika, aby tak postąpił, i wyjaśnić mu, jak to zrobić, dlatego Microsoft nie traktuje takiego scenariusza jako potencjalnego wektora ataku. Pamiętaj jednak, że powłoka PowerShell nie daje użytkownikom żadnych dodatkowych uprawnień — będą mogli wykonywać tylko te czynności, do których wykonania otrzymali uprawnienia od administratora systemu.

Teoretycznie złośliwy „ktoś” mógłby próbować zadzwonić do użytkownika lub za pośrednictwem poczty elektronicznej wysłać odpowiednio spreparowaną wiadomość opisującą, jak uruchomić powłokę PowerShell, a użytkownik w dobrej wierze mógłby wpisać odpowiednią sekwencję poleceń, które uszkodziłyby system lub przechowywane w nim dane. Równie dobrze ten sam „ktoś” mógłby przeprowadzić taki atak za pomocą czegoś innego niż powłoka PowerShell. Skala trudności przeprowadzenia takiego ataku przy użyciu powłoki PowerShell byłaby mniej więcej taka sama, jakby nasz złośliwy „ktoś” próbował namówić użytkownika do uruchomienia Eksploratora Windows, zaznaczenia folderu *Program Files* i naciśnięcia klawisza *Delete*. W pewnym sensie taki dosyć niedorzeczny scenariusz ataku byłby może nawet łatwiejszy do przeprowadzenia niż namówienie użytkownika do wykonania równoważnej sekwencji poleceń z użyciem powłoki PowerShell.

Zwracamy na to uwagę tylko dlatego, że przeciętny użytkownik komputera często podświadomie odczuwa jakiś atawistyczny lęk przed używaniem wiersza poleceń konsoli i jego pozornie nieograniczonych możliwości, choć w praktyce przecież ani Ty, ani żaden z Twoich użytkowników nie może za pomocą powłoki zrobić niczego, czego nie można by było zrobić na tuzin innych sposobów.

17.6. Zalecenia bezpieczeństwa

Jak już wspominaliśmy wcześniej, firma Microsoft zaleca ustawienie polityki wykonywania skryptów na poziom *RemoteSigned* na tych komputerach, na których trzeba uruchamiać skrypty. W zależności od potrzeb możesz jednak dla takich maszyn rozważyć także ustawienie tej polityki na poziomie *AllSigned* lub nawet *Unrestricted*.

Poziom *AllSigned* jest nieco mniej wygodny, ale możesz ułatwić korzystanie z niego dzięki następującym wskazówkom:

- Komercyjne urzędy certyfikacji pobierają nawet do 900 USD opłat rocznie za certyfikat do podpisywania kodu. Jeżeli Twoja organizacja nie ma wewnętrznej infrastruktury PKI, która może zapewnić uzyskanie bezpłatnego certyfikatu, możesz utworzyć swój własny. Zapoznaj się z zawartością pliku pomocy *about_signing*, gdzie znajdziesz informacje na temat używania programu *Makecert.exe*, czyli narzędzia pozwalającego na tworzenie własnych certyfikatów, które będą traktowane jako zaufane tylko przez Twój komputer lokalny. Jeżeli jest to jedyne miejsce, w którym musisz uruchomić skrypty, to użycie tego narzędzia jest szybkim i całkowicie darmowym sposobem na uzyskanie potrzebnego certyfikatu.

W zależności od tego, której wersji powłoki PowerShell używasz, możesz również skorzystać z polecenia o nazwie `New-SelfSignedCertificate`, spełniającego tę samą rolę co `makecert`.

- Jeżeli posiadasz już odpowiedni certyfikat do podpisywania kodu, swoje skrypty powinieneś pisać w jednym z edytorów, o których wspominaliśmy wcześniej. Wszystkie te edytory potrafią automatycznie podpisywać skrypt za każdym razem, gdy zapisujesz plik. Dzięki temu taki proces jest całkowicie niezauważalny dla użytkownika i znacząco ułatwia cyfrowe podpisywanie skryptów.

Wspominaliśmy również, że nie powinieneś zmieniać skojarzenia rozszerzenia `.PS1`. Zdarzało nam się widywać, jak niektórzy użytkownicy modyfikowali system Windows tak, aby traktował pliki z rozszerzeniem `.PS1` jako pliki wykonywalne, co oznaczało, że taki skrypt można było uruchomić poprzez dwukrotne kliknięcie jego ikony. Takie rozwiązanie, choć pozornie wygodne, przenosi nas jednak z powrotem do starych, nie najlepszych czasów skryptów w języku VBScript, a tego prawdopodobnie będziesz chciał uniknąć.

Chcemy podkreślić, że żaden ze skryptów dostarczanych jako materiały do tej książki nie jest podpisany cyfrowo. Można je zatem modyfikować bez naszej (lub Twojej) wiedzy. Zanim uruchomisz któryś z tych skryptów, powinieneś poświęcić trochę czasu na ich sprawdzenie, zrozumienie, co mają robić, i upewnienie się, że pasują do opisów w książce (tam, gdzie to konieczne). Specjalnie nie podpisywaliśmy tych skryptów, ponieważ *chcemy, abyś poświęcił trochę czasu na ich sprawdzenie* — powinieneś wyrobić w sobie nawyk dokładnego sprawdzania każdego skryptu, który pobrałeś z sieci Internet, bez względu na to, jak „zaufany” może się wydawać jego autor.

17.7. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Twoje zadanie do wykonania tym razem jest bardzo proste — tak proste, że nawet nie dajemy przykładowego rozwiązania. Chcemy, abyś skonfigurował swoją powłokę tak, aby umożliwić wykonywanie skryptów. Użyj polecenia `Set-ExecutionPolicy` — naszym zdaniem powinieneś ustawić politykę wykonywania skryptów na poziomie `RemoteSigned`. Jeżeli chcesz, możesz użyć poziomu `AllSigned`, ale będzie to dosyć niepraktyczne pod kątem wykonywania pozostałych ćwiczeń w dalszej części książki. Oczywiście możesz także wybrać poziom `Unrestricted`.

Przypominamy, że jeżeli używasz powłoki PowerShell w środowisku produkcyjnym, powinieneś się upewnić, że wybrane ustawienie polityki wykonywania skryptów jest zgodne z zasadami i procedurami bezpieczeństwa Twojej organizacji. Nie chcemy, abyś wpadł w tarapaty podczas wykonywania ćwiczeń z tej książki.

18

Zmienne — miejsca do przechowywania danych

Wspominaliśmy już, że powłoka PowerShell posiada wbudowany język skryptowy. Będziemy się nim zajmować w kilku kolejnych rozdziałach. Jeżeli jednak zaczynasz pisanie skryptów, to wcześniej czy później będziesz potrzebować *zmiennych*, którymi zajmiemy się już teraz. Poza skryptami ze zmiennych możesz korzystać w wielu innych miejscach, więc w tym rozdziale pokażemy Ci również kilka praktycznych sposobów ich użycia.

18.1. Wprowadzenie do zmiennych

W najprostszy sposób *zmienną* możesz sobie wyobrazić jako szufladkę czy może raczej pudełko w pamięci komputera, posiadające swoją unikatową nazwę. W takim „pudełku” możesz umieścić wszystko, co chcesz: nazwę wybranego komputera, kolekcję usług, dokument XML i tak dalej. Dostęp do zmiennej uzyskujesz za pośrednictwem jej nazwy i możesz w niej umieszczać dowolne elementy lub pobierać elementy w niej przechowywane. Elementy umieszczone w takim „pudełku” pozostają w nim dopóty, dopóki nie zostaną zmienione lub usunięte, dzięki czemu możesz je pobierać wielokrotnie.

Powłoka PowerShell nie narzuca zmiennym zbyt wielu wymogów formalnych. Na przykład nie musisz jawnie deklarować zmiennej przed jej pierwszym użyciem. Możesz dowolnie zmieniać typy wartości przechowywanych w zmiennej — w jednej chwili w wybranej zmiennej możesz przechowywać obiekt procesu, a już chwilę później możesz zapisać w niej ciąg alfanumeryczny reprezentujący nazwy kilku komputerów. Zmienna może nawet zawierać wiele różnych elementów, takich jak kolekcja usług *i* kolekcja procesów (choć przyznajemy, że wówczas użycie takiej zmiennej może być trudne).

18.2. Przechowywanie wartości w zmiennych

Wszystko w powłoce PowerShell — i mamy na myśli *wszystko* — jest traktowane jako obiekt. Nawet prosty ciąg znaków, taki jak nazwa komputera, jest uważany za obiekt. Na przykład przekazanie ciągu znaków za pomocą potoku do polecenia Get-Member (lub jego aliasu Gm) ujawnia, że taki ciąg znaków jest obiektem typu System.String i posiada wiele metod, z którymi można pracować (ze względu na długość pełnej listy zamieszczamy poniżej tylko jej fragment):

```
PS C:\> "SERVER-R2" | gm
      TypeName: System.String
Name      MemberType Definition
-----
Clone      Method      System.Object Clone()
CompareTo  Method      int CompareTo(System.Object valu...
Contains   Method      bool Contains(string value)
CopyTo     Method      System.Void CopyTo(int sourceInd...
EndsWith   Method      bool EndsWith(string value), boo...
Equals     Method      bool Equals(System.Object obj), ...
GetEnumerator Method      System.CharEnumerator GetEnumera...
GetHashCode Method      int GetHashCode()
GetType    Method      type GetType()
GetTypeCode Method      System.TypeCode GetTypeCode()
IndexOf    Method      int IndexOf(char value), int Ind...
IndexOfAny Method      int IndexOfAny(char[] anyOf), in...
```

ZRÓB TO SAM Spróbuj samodzielnie uruchomić to polecenie w powłoce PowerShell, żeby przekonać się, jak wyglądają: kompletna lista metod i właściwości obiektu System.String.

Chociaż ciąg znaków, tak jak wszystko inne w powłoce, z technicznego punktu widzenia jest obiektem, to wielu użytkowników odwołuje się do niego jako do prostej wartości. Dzieje się tak dlatego, że w większości przypadków użytkownik jest zainteresowany samym ciągiem znaków — w naszym przykładzie SERVER-R2 — a mniej zależy mu na pobieraniu informacji z jego właściwości. Różni się to na przykład od procesów, gdzie cały obiekt procesu jest dużą, abstrakcyjną strukturą danych, a Ty zazwyczaj zajmujesz się jego poszczególnymi właściwościami, takimi jak VM, PM, Name, CPU czy ID. Ciąg znaków String jest obiektem, ale jest znacznie mniej skomplikowany niż na przykład Process.

Powłoka PowerShell pozwala przechowywać takie proste wartości w zmiennej. Aby to zrobić, powinieneś podać nazwę zmiennej i użyć znaku równości — czyli inaczej mówiąc, *operatora przypisania* — po którym następuje to, co chcesz umieścić w zmiennej. Oto przykład:

```
PS C:\> $var = "SERVER-R2"
```

ZRÓB TO SAM Spróbuj samodzielnie wykonywać wszystkie opisywane tutaj przykłady, ponieważ wtedy będziesz w stanie zweryfikować wyniki, które pokazujemy. Pamiętaj, że zamiast SERVER-R2 powinieneś używać nazwy swojego serwera testowego.

Ważne jest, aby pamiętać, że znak dolara (\$) nie jest częścią nazwy zmiennej. W naszym przykładzie nazwa zmiennej to `var`. Znak dolara jest wskazówką dla powłoki, że to, co po nim następuje, będzie nazwą zmiennej i że chcemy uzyskać dostęp do zawartości tej zmiennej. W tym przypadku ustawiamy wartość zmiennej.

Poniżej zamieszczamy krótkie zestawienie najważniejszych informacji o zmiennych i ich nazwach:

- Nazwy zmiennych zwykle zawierają litery, cyfry i znaki podkreślenia, a najczęściej zaczynają się od litery lub znaku podkreślenia.
- Nazwy zmiennych mogą zawierać spacje, ale w takim przypadku nazwa musi być ujęta w nawiasy klamrowe. Na przykład wyrażenie `${Moja zmienna}` reprezentuje zmienną o nazwie `Moja zmienna`. Osobiście nie przepadam za nazwami zmiennych, które zawierają spacje, ponieważ wymagają one zwykle więcej pisania i są trudniejsze do odczytania.
- Zmienne nie są przechowywane między poszczególnymi sesjami powłoki. Po zamknięciu bieżącej sesji powłoki wszystkie zmienne, które w niej utworzyłeś, znikają.
- Nazwy zmiennych mogą być dość długie — na tyle długie, że nie musisz się martwić, że będą za długie. Postaraj się jednak, aby nazwy zmiennych były sensowne. Przykładowo, jeżeli używasz zmiennej do przechowywania nazwy komputera, możesz nadać takiej zmiennej nazwę na przykład `computername`. Jeżeli dana zmienna przechowuje informacje o procesach, wówczas dobrą nazwą dla niej może być `processes`.
- Z wyjątkiem osób, które mają doświadczenia z językiem VBScript, użytkownicy powłoki PowerShell zwykle nie używają prefiksów nazw zmiennych wskazujących typ wartości przechowywanych w zmiennej. Na przykład w języku VBScript nazwa `strComputerName` była typowym przykładem nazwy zmiennej wskazującej, że jej wartością jest ciąg znaków (wskazuje na to prefiks `str`). Powłoka PowerShell nie zwraca uwagi na to, czy to robisz, ale w społeczności użytkowników powłoki taka praktyka nie jest już powszechnie stosowana ani oczekiwana.

Aby pobrać zawartość zmiennej, użyj znaku dolara, a następnie nazwy zmiennej, tak jak pokazano w poniższym przykładzie. I znów, znak dolara informuje powłokę, że chcesz uzyskać dostęp do *zawartości* zmiennej, której nazwa została umieszczona po tym znaku.

```
PS C:\> $var  
SERVER-R2
```

Niemal zawsze zamiast podawać określoną wartość, możesz użyć zmiennej. Na przykład korzystając z usługi WMI, możesz określić nazwę komputera, na którym zostanie wykonane zapytanie. W „normalnym” wydaniu przykładowe polecenie może wyglądać tak:

```
PS C:\> get-wmiobject win32_computersystem -comp SERVER-R2  
Domain                : company.pri  
Manufacturer          : VMware, Inc.  
Model                 : VMware Virtual Platform  
Name                  : SERVER-R2  
PrimaryOwnerName      : Windows User  
TotalPhysicalMemory    : 3220758528
```

W wierszu wywołania polecenia zamiast dowolnej wartości możemy jednak podstawić odpowiednią zmienną:

```
PS C:\> get-wmiobject win32_computersystem -comp $var
Domain                : company.pri
Manufacturer          : VMware, Inc.
Model                 : VMware Virtual Platform
Name                  : SERVER-R2
PrimaryOwnerName      : Windows User
TotalPhysicalMemory   : 3220758528
```

Nawiasem mówiąc, oczywiście doskonale zdajemy sobie sprawę z tego, że `var` jest dosyć ogólną nazwą zmiennej. Normalnie użylibyśmy tutaj zapewne zmiennej o nazwie `computername`, ale w tym konkretnym przypadku planujemy ponowne użycie zmiennej `$var` jeszcze w kilku innych sytuacjach, więc zdecydowaliśmy się zachować jej ogólną nazwę. Nie pozwól jednak, aby ten przykład powstrzymał Cię od używania lepiej dobranych i bardziej przejrzystych nazw zmiennych.

Nieco wcześniej umieściliśmy w zmiennej `$var` ciąg znaków, ale możemy to zmienić w dowolnym momencie:

```
PS C:\> $var = 5
PS C:\> $var | gm
    TypeName: System.Int32
Name      MemberType      Definition
-----
CompareTo  Method                int CompareTo(System.Object value), int CompareT...
Equals     Method                bool Equals(System.Object obj), bool Equals(int ...
GetHashCode Method                int GetHashCode()
GetType    Method                type GetType()
GetTypeCode Method                System.TypeCode GetTypeCode()
ToString   Method                string ToString(), string ToString(string format...
```

W poprzednim przykładzie w zmiennej `$var` umieściliśmy liczbę całkowitą, a następnie za pomocą potoku przekazaliśmy jej wartość do polecenia `Gm`. Łatwo zauważyć, że powłoka rozpoznaje zawartość zmiennej `$var` jako obiekt typu `System.Int32` lub inaczej mówiąc, jako 32-bitową liczbę całkowitą.

18.3. Używanie zmiennych: zabawne sztuczki z apostrofami i cudzysłowami

Ponieważ mówimy o zmiennych, nadszedł dobry moment na omówienie ciekawej funkcjonalności powłoki PowerShell. Do tego momentu doradzaliśmy, aby ogólnie zamykać ciągi znaków w cudzysłowach. Powodem było to, że powłoka PowerShell traktuje wszystko ujęte w znaki apostrofu jako literały znakowe.

Rozważmy następujący przykład:

```
PS C:\> $var = 'Co zawiera zmienna $var?'
PS C:\> $var
Co zawiera zmienna $var?
```

Łatwo zauważyć, że zmienna `$var` w ciągu znaków umieszczonym w apostrofach jest traktowana jako literal. Jednak w przypadku umieszczenia ciągu znaków w cudzysłowie już tak nie jest. Przyjrzyj się następującemu przykładowi:

```
PS C:\> $computername = 'SERVER-R2'
PS C:\> $phrase = "Nazwa komputera to $computername"
PS C:\> $phrase
Nazwa komputera to SERVER-R2
```

Nasz przykład rozpoczynamy od umieszczenia ciągu znaków `SERVER-R2` w zmiennej o nazwie `$computername`. Następnie w zmiennej o nazwie `$phrase` umieszczamy ciąg znaków `"Nazwa komputera to $computername"`. Tym razem jednak zamiast apostrofów używamy cudzysłowu. Powłoka PowerShell automatycznie wyszukuje znaki dolara w ciągach znaków ujętych w cudzysłowy i *zastępuje wszystkie znalezione zmienne ich wartościami*. Ponieważ wyświetlamy treść wyrażenia `$phrase`, zmienna `$computername` zostaje zastąpiona przez jej zawartość, czyli ciąg znaków `SERVER-R2`.

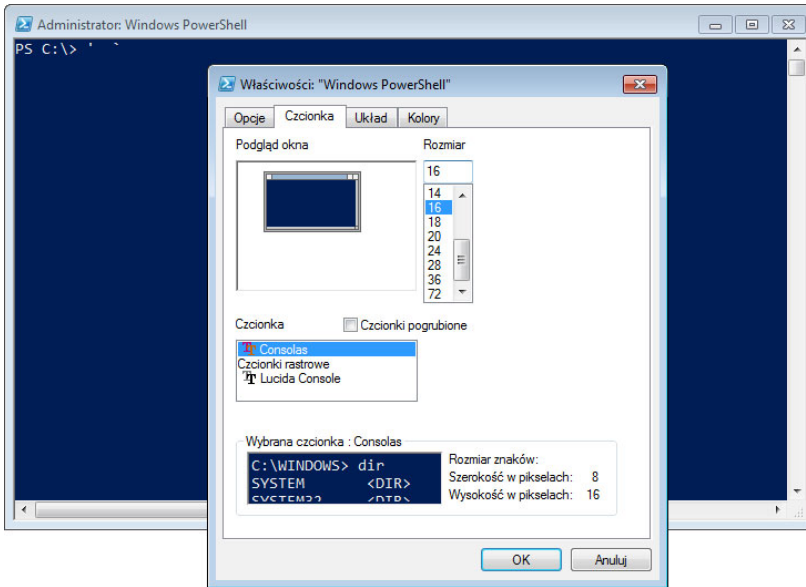
Operacja podstawiania wartości w miejsce zmiennych jest wykonywana, kiedy powłoka rozpoczyna parsowanie ciągu znaków. Po jej wykonaniu wartością zmiennej `$phrase` jest ciąg znaków `"Nazwa komputera to SERVER-R2"`, który nie zawiera już nazwy zmiennej `$computername`. Możemy się o tym przekonać, zmieniając zawartość zmiennej `$computername` i sprawdzając, czy zawartość zmiennej `$phrase` zostanie zaktualizowana:

```
PS C:\> $computername = 'SERVER1'
PS C:\> $phrase
Nazwa komputera to SERVER-R2
```

Jak łatwo zauważyć, zawartość zmiennej `$phrase` pozostała niezmieniona.

Kolejną sprawą związaną z umieszczaniem ciągów znaków w cudzysłowie jest zastosowanie odwróconego apostrofu (```), który w powłoce PowerShell odgrywa rolę znaku ucieczki. Na standardowych klawiaturach znajduje się on na jednym z klawiszy w lewym górnym rogu klawiatury, zwykle poniżej klawisza *Esc* (najczęściej na tym samym klawiszu co znak tyldy `~`). Problem polega na tym, że w przypadku niektórych krojów czcionek jest praktycznie nie do odróżnienia od zwykłego apostrofu. W praktyce najczęściej konfigurujemy powłokę tak, aby używała czcionki *Consolas*, ponieważ dzięki temu znacznie łatwiej odróżnić od siebie te dwa znaki niż w przypadku innych czcionek, takich jak *Lucida Console* czy *Raster*.

ZRÓB TO SAM Kliknij ikonę znajdującą się z lewej strony paska tytułowego okna powłoki PowerShell i z menu podręcznego wybierz polecenie *Właściwości*. Przejdź na kartę *Czcionka* i wybierz z listy czcionkę *Consolas*, tak jak przedstawiono na rysunku 18.1. Naciśnij przycisk *OK* i wpisz apostrof oraz odwrotny apostrof, aby zobaczyć różnicę między tymi znakami. Rysunek 18.1 pokazuje, jak to wygląda w naszym systemie. Nie widzisz różnicy? Nam też jest ją trudno zauważyć, nawet kiedy używamy dużego rozmiaru czcionki. Rozróżnienie tych dwóch znaków może faktycznie być nieco utrudnione, ale upewnij się, że potrafisz to zrobić niezależnie od wybranego kroju i rozmiaru czcionki.



Rysunek 18.1. Ustawianie czcionki pozwalającej na łatwiejsze rozróżnienie apostrofu i odwrotnego apostrofu

Spójrzmy, do czego możemy używać znaku ucieczki. Użycie tego znaku powoduje, że znak umieszczony bezpośrednio po nim jest traktowany jak zwykły znak (choć w niektórych przypadkach znak ucieczki nadaje następującemu po nim znakowi specjalne znaczenie). Poniżej zamieszczamy przykład pierwszego zastosowania znaku ucieczki:

```
PS C:\> $computername = 'SERVER-R2'
PS C:\> $phrase = "`$computername to $computername"
PS C:\> $phrase
$computername contains SERVER-R2
```

Kiedy przypisujemy ciąg znaków do zmiennej `$phrase`, dwukrotnie używamy nazwy `$computername`. Za pierwszym razem poprzedzamy znak dolara znakiem ucieczki (odwrotnym apostrofem). W ten sposób odbieramy specjalne znaczenie znaku `$` jako wskaźnika zmiennej i powodujemy, że powłoka traktuje go literalnie jako znak dolara. Zmienna `$computername` przechowuje nazwę komputera. W drugim przypadku nie używamy znaku ucieczki, więc zamiast zmiennej `$computername` zostaje podstawiona jej wartość.

Przyjrzyjmy się teraz drugiemu sposobowi zastosowania znaku ucieczki powłoki PowerShell:

```
PS C:\> $phrase = "`$computername`nzawiera `n$computername"
PS C:\> $phrase
$computername
zawiera
SERVER-R2
```

Jeżeli uważnie przyjrzyj się powyższemu przykładowi, zauważysz, że w pierwszym wyrażeniu dwukrotnie używamy ciągu znaków ``n` — pierwszy raz po pierwszym wystąpieniu

zmiennej `$computername`, a drugi raz po słowie zawiera. W tym przykładzie znak ucieczki nadaje literze *n* specjalne znaczenie. Zwykle znak *n* jest traktowany jako litera, ale poprzedzony znakiem ucieczki staje się znakiem reprezentującym powrót karetki i wysunięcie nowego wiersza.

Więcej szczegółowych informacji na ten temat znajdziesz w pliku pomocy `about_↵escape`, gdzie zamieszczona jest również lista innych znaków specjalnych. Na przykład za pomocą ciągu znaków ``t` możesz wstawić tabulator, a użycie ciągu znaków ``a` powoduje wygenerowanie przez komputer krótkiego dźwięku (*a* jak *alarm*).

18.4. Przechowywanie wielu obiektów w zmiennej

Do tej pory pracowaliśmy ze zmiennymi, które przechowują pojedynczy obiekt, a wszystkie te obiekty były prostymi wartościami. Co więcej, pracowaliśmy bezpośrednio z obiektami, a nie z ich właściwościami lub metodami. Teraz spróbujemy umieścić kilka obiektów w jednej zmiennej.

Jednym ze sposobów na zrobienie tego jest użycie listy wartości oddzielonych od siebie przecinkami, ponieważ powłoka PowerShell rozpoznaje takie listy jako kolekcje obiektów:

```
PS C:\> $computers = 'SERVER-R2','SERVER1','localhost'
PS C:\> $computers
SERVER-R2
SERVER1
localhost
```

Zwróć uwagę, że w tym przykładzie przecinki zostały umieszczone poza cudzysłowami. Gdybyśmy umieścili je wewnątrz cudzysłówów, otrzymalibyśmy pojedynczy obiekt zawierający przecinki i trzy nazwy komputerów. Dzięki naszej metodzie otrzymujemy trzy różne obiekty typu `String`. Jak widać, kiedy sprawdzamy zawartość zmiennej, powłoka PowerShell wyświetla każdy obiekt w osobnej linii.

18.4.1. Praca z pojedynczymi obiektami w zmiennej

W razie potrzeby możesz uzyskać dostęp do poszczególnych obiektów przechowywanych w zmiennej, po jednym na raz. Aby to zrobić, w nawiasach kwadratowych powinieneś podać numer indeksu żadanego elementu. Pierwszy obiekt ma zawsze numer indeksu równy 0, drugi obiekt ma numer indeksu 1 i tak dalej. Możesz również użyć indeksu o wartości -1, aby uzyskać dostęp do ostatniego obiektu, -2 dla przedostatniego obiektu i tak dalej. Oto przykład:

```
PS C:\> $computers[0]
SERVER-R2
PS C:\> $computers[1]
SERVER1
PS C:\> $computers[-1]
localhost
PS C:\> $computers[-2]
SERVER1
```

Sama zmienna posiada właściwość pozwalającą sprawdzić, ile obiektów jest w niej przechowywanych:

```
PS C:\> $computers.count
3
```

Możesz także uzyskać dostęp do właściwości i metod obiektów przechowywanych w zmiennej, tak jakby były właściwościami i metodami samej zmiennej. Na początku łatwiej się o tym przekonać, jeżeli zmienna zawiera tylko pojedynczy obiekt:

```
PS C:\> $computername.length
9
PS C:\> $computername.toupper()
SERVER-R2
PS C:\> $computername.tolower()
server-r2
PS C:\> $computername.replace('R2', '2008')
SERVER-2008
PS C:\> $computername
SERVER-R2
```

W tym przykładzie używamy zmiennej `$computername`, którą utworzyliśmy wcześniej w tym rozdziale. Jak zapewne pamiętasz, ta zmienna zawiera obiekt typu `System.String`, którego pełną listę właściwości i metod możemy wyświetlić po przekazaniu go za pomocą potoku do polecenia `Gm` (tak jak to robiliśmy w podrozdziale 18.2). Używamy tutaj właściwości `Length`, a także metod `ToUpper()`, `ToLower()` i `Replace()`. W każdym przypadku po nazwie metody musimy umieścić parę nawiasów okrągłych, nawet jeżeli metody takie jak `ToUpper()` czy `ToLower()` nie wymagają podawania żadnych parametrów wywołania. Ponadto żadna z tych metod nie zmienia oryginalnej zawartości zmiennej, co widać w ostatnim wierszu powyższego przykładu. Zamiast tego każde wywołanie metody tworzy nowy obiekt `String` bazujący na oryginalnej wartości zmiennej i odpowiednio zmodyfikowany przez metodę.

18.4.2. Praca z wieloma obiektami w zmiennej

Gdy zmienna zawiera wiele obiektów, wykonanie takich operacji może być trudniejsze. Nawet jeżeli wszystkie obiekty przechowywane w zmiennej są tego samego typu (tak jak to ma miejsce w przypadku naszej zmiennej `$computers`), powłoka PowerShell v2 *nie pozwoli na wywołanie metody lub uzyskanie dostępu do właściwości wielu obiektów w tym samym czasie*. Każda próba wykonania takiej operacji zakończy się błędem:

```
PS C:\> $computers.toupper()
Method invocation failed because [System.Object[]] doesn't contain a method named
'toupper'.
At line:1 char:19
+ $computers.toupper <<<< ( )
+ CategoryInfo          : InvalidOperation: (toupper:String) [], RuntimeException
+ FullyQualifiedErrorId : MethodNotFound
```

Zamiast tego musisz określić, z którym obiektem przechowywanym w zmiennej chcesz pracować, i dopiero wtedy możesz uzyskać dostęp do jego właściwości lub wykonać metodę tego konkretnego obiektu:

```
PS C:\> $computers[0].tolower()
server-r2
PS C:\> $computers[1].replace('SERVER', 'CLIENT')
CLIENT1
```

Podobnie jak poprzednio, wykonanie takich metod tworzy nowe ciągi znaków, nie zmieniając oryginalnych zawartości zmiennych. Możesz to sprawdzić, wyświetlając zawartość zmiennej `$computers`:

```
PS C:\> $computers
SERVER-R2
SERVER1
localhost
```

Co się stanie, jeżeli zechcesz zmienić zawartość zmiennej? W takiej sytuacji możesz przypisać nową wartość do jednego z istniejących obiektów:

```
PS C:\> $computers[1] = $computers[1].replace('SERVER', 'CLIENT')
PS C:\> $computers
SERVER-R2
CLIENT1
localhost
```

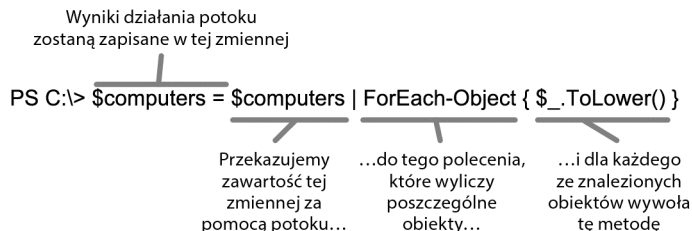
W tym przykładzie widać, że zamiast tworzyć nowy ciąg znaków, zmieniamy drugi obiekt przechowywany w zmiennej. Pokazujemy to na wypadek, gdybyś miał do czynienia z komputerami, na których zainstalowano tylko powłokę PowerShell w wersji v2; powłoka w wersji v3 zachowuje się nieco inaczej, o czym przekonasz się już niebawem.

18.4.3. Inne sposoby pracy z wieloma obiektami

Chcielibyśmy teraz pokazać Ci dwa inne sposoby pracy z właściwościami i metodami wielu obiektów przechowywanych w zmiennej. W poprzednich przykładach wykonywaliśmy metody tylko jednego obiektu wewnątrz zmiennej. Jeżeli chcesz uruchomić metodę `ToLower()` dla każdego obiektu przechowywanego w zmiennej i zapisać wyniki z powrotem w tej zmiennej, możesz użyć następującego polecenia:

```
PS C:\> $computers = $computers | ForEach-Object { $_.ToLower() }
PS C:\> $computers
server-r2
client1
localhost
```

Przedstawiony przykład jest nieco skomplikowany, więc na rysunku 18.2 rozłożyliśmy go na czynniki pierwsze. Potok rozpoczynamy od wyrażenia `$computers =`, co oznacza, że wyniki działania całego polecenia będą przechowywane w tej zmiennej i nadpiszą wszystko, co było w niej przed uruchomieniem polecenia.



Rysunek 18.2. Zastosowanie polecenia ForEach-Object do wykonania metody na wszystkich obiektach przechowywanych w zmiennej

Nasze polecenie rozpoczyna się od przekazania za pomocą potoku zawartości zmiennej `$computers` do polecenia `ForEach-Object`, które wylicza wszystkie obiekty w potoku (nasza zmienna przechowuje trzy obiekty typu `String`, reprezentujące nazwy komputerów) i wykonuje dla każdego z nich blok skryptu. W bloku skryptu do zmiennej zastępczej `$_` podstawiane są kolejno pojedyncze obiekty, na których wykonywana jest metoda `ToLower()`. Nowe obiekty typu `String` tworzone przez metodę `ToLower()` są umieszczane w potoku i finalnie zapisywane w zmiennej `$computers`.

Przy użyciu polecenia `Select-Object` możesz zrobić coś podobnego z właściwościami obiektów. W przykładzie zamieszczonym poniżej wybierana jest właściwość `Length` każdego obiektu przekazanego do polecenia za pomocą potoku:

```
PS C:\> $computers | select-object length
```

Length

9
7
9

Ponieważ wartość właściwości `Length` jest liczbą, powłoka PowerShell wyrównuje wyniki do prawej strony.

18.4.4. Rozwijanie właściwości i metod obiektów w powłoce PowerShell v3

Cała ta sprawa, że *PowerShell nie pozwala na wywołanie metody lub uzyskanie dostępu do właściwości wielu obiektów w tym samym czasie*, okazała się bardzo kłopotliwa dla użytkowników powłoki w wersjach v1 i v2. Było to do tego stopnia kłopotliwe, że w wersji v3 powłoki PowerShell firma Microsoft dokonała ważnej zmiany, implementując nowy mechanizm pozwalający na automatyczne rozwijanie metod i właściwości obiektów (ang. *automatic unrolling*), dzięki któremu *możesz* uzyskać dostęp do właściwości i metod poszczególnych obiektów za pomocą nazwy przechowującej je zmiennej:

```
$services = Get-Service
$services.Name
```

Jeżeli próbujesz wykonać takie polecenie, powłoka PowerShell „widzi”, że starasz się uzyskać dostęp do właściwości obiektów przechowywanych w zmiennej. Co więcej, powłoka „widzi” także, że kolekcja obiektów przechowywanych w zmiennej `$services`

nie ma właściwości Name, ale że mają ją poszczególne obiekty w tej kolekcji, stąd rozwija poszczególne obiekty i pobiera właściwość Name każdego z nich. Jest to równoważne wykonaniu następującego polecenia:

```
Get-Service | ForEach-Object {Write-Output $_.Name}
```

...lub takiego:

```
Get-Service | Select-Object -ExpandProperty Name
```

czyli dwóm opisywanym wcześniej metodom wykonania takiego zadania w powłoce PowerShell w wersji v1 lub v2. Automatyczne rozwijanie działa również w odniesieniu do metod:

```
$objects = Get-WmiObject -class Win32_Service -filter "name = 'BITS'"
$objects.ChangeStartMode("Disabled")
```

Pamiętaj jednak, że opisywany mechanizm jest dostępny w powłoce PowerShell w wersji v3 i nowszych, nie powinieneś zatem oczekiwać, że będzie to działać w ten sposób w starszych wersjach tej powłoki.

18.5. Więcej trików z cudzysłowami

Chcielibyśmy pokazać Ci jeszcze kolejną fajną technikę użycia cudzysłowu, która jest nieco koncepcyjnym rozszerzeniem mechanizmu podstawiania odpowiednich wartości w miejsce zmiennych. Załóżmy na przykład, że w zmiennej \$service umieściliśmy kilka usług, a teraz chcemy do nowej zmiennej wstawić tylko nazwę pierwszej usługi:

```
PS C:\> $services = get-service
PS C:\> $firstname = "$services[0].name"
PS C:\> $firstname
AeLookupSvc ALG AllUserInstallAgent AppIDSvc Appinfo AppMgmt AudioEndpoint
Builder Audiosrv AxInstSV BDESVc BFE BITS BrokerInfrastructure Browser bth
serv CertPropSvc COMSysApp CryptSvc CscService DcomLaunch defragSvc Device
AssociationService DeviceInstall Dhcp Dnscache dot3Svc DPS DsmSvc Eaphost
EFS ehRecvr ehSched EventLog EventSystem Fax fdPHost FDResPub fhSvc FontCa
che gpsvc hidserv hkmsvc HomeGroupListener HomeGroupProvider IKEEXT iphlps
vc KeyIso KtmRm LanmanServer LanmanWorkstation lltdSvc lmhosts LSM Mcx2Svc
MMCSS MpsSvc MSDTC MSiSCSI msiserver napagent NcaSvc NcdAutoSetup Netlogo
n Netman netprofm NetTcpPortSharing NlaSvc nsi p2pimsvc p2psvc Parallels C
oherence Service Parallels Tools Service PcaSvc PeerDistSvc PerfHost pla P
lugPlay PNRPAutoReg PNRPsvc PolicyAgent Power PrintNotify ProfSvc QWAVE Ra
sAuto RasMan RemoteAccess RemoteRegistry RpcEptMapper RpcLocator RpcSs Sam
Ss SCardSvr Schedule SCPolicySvc SDRSVc seclogon SENS SensrSvc SessionEnv
SharedAccess ShellHWDetection SNMPTRAP Spooler spsSvc SSDPSRV SstpSvc stis
vc StorSvc svsvc swprv SysMain SystemEventsBroker TabletInputService TapiS
rv TermService Themes THREADORDER TimeBroker TrkWks TrustedInstaller UIODE
tect UmRdpService upnphost VaultSvc vds vmicheartbeat vmickvpexchange vmic
rdv vmicshutdown vmictimesync vmicvss VSS W32Time wbengine WbioSvc Wcmsvc
wncsvc WcsPlugInService WdiServiceHost WdiSystemHost WdNisSvc WebClient
WeSvc werpcsupport WerSvc WiaRpc WinDefend WinHttpAutoProxySvc Winmgmt W
inRM WlanSvc wlidsvc wmiApSrv WMPNetworkSvc WPCSVc WPDBusEnum wscsvc WSear
ch WSService wuauserv wudfsvc WwanSvc[0].name
```

Ups, chyba jednak nie o to nam chodziło. Znak [otwierający nawias kwadratowy, umieszczony bezpośrednio po nazwie zmiennej \$services, nie jest traktowany jako legalna część nazwy zmiennej, w wyniku czego powłoka PowerShell próbuje zastąpić zmienną \$services jej zawartością, co powoduje umieszczenie nazw wszystkich usług w ciągu znaków. Wyrażenie [0].name jest traktowane jako literal i nie zostaje zamienione.

Rozwiązaniem jest użycie następującego wyrażenia:

```
PS C:\> $services = get-service
PS C:\> $firstname = "Pierwsza nazwa usługi to $($services[0].name)"
PS C:\> $firstname
Pierwsza nazwa usługi to AeLookupSvc
```

Wszystko wewnątrz wyrażenia \$() jest traktowane jako normalne polecenie powłoki PowerShell, a wynik jest umieszczany w ciągu znaków, zastępując wszystko, co istniało do tej pory. Wykonanie takiej operacji jest możliwe tylko z użyciem znaków cudzysłowu. Konstrukcja \$() jest nazywana **podwyrażeniem** (ang. *subexpression*).

Mamy jeszcze w zanadru kolejną fajną sztuczkę, z której możesz skorzystać, pracując z powłoką PowerShell v3. Czasami niezbędne okazuje się umieszczenie czegoś bardziej skomplikowanego w zmiennej, a następnie wyświetlenie zawartość tej zmiennej w cudzysłowie. W wersji v3 powłoka jest wystarczająco inteligentna, aby wyliczyć wszystkie obiekty w kolekcji, nawet jeśli odwołujesz się do pojedynczej właściwości lub metody, pod warunkiem jednak, że wszystkie obiekty w kolekcji są tego samego typu. Przykładowo, pobierzemy listę usług i wstawimy ją do zmiennej \$service, a następnie w ciągu znaków w cudzysłowie umieścimy odwołanie do nazw usług:

```
PS C:\> $services = get-service
PS C:\> $var = "Nazwy usług: $services.name"
PS C:\> $var
Nazwy usług: AeLookupSvc ALG AllUserInstallAgent AppIDSvc Appinfo App
Mgmt AudioEndpointBuilder Audiosrv AxInstSV BDESvc BFE BITS BrokerInfrastr
ucture Browser bthserv CertPropSvc COMSysApp CryptSvc CscService DcomLaunc
h defragSvc DeviceAssociationService DeviceInstall Dhcp Dnscache dot3Svc D
PS DsmSvc Eaphost EFS ehRecvr ehSched EventLog EventSystem Fax fdPHost FDR
esPub fhSvc FontCache FontCache3.0.0.0 gpSvc hidserv hkmsvc HomeGroupListe
ner HomeGroupProvider IKEEXT iphlpsvc KeyIso KtmRm LanmanServer LanmanWork
station lltdSvc lmhosts LSM Mcx2Svc MMCSS MpsSvc MSDTC MSiSCSI msiserver M
SSQL$SQLEXPRESS napagent NcaSvc NcdAutoSetup Netlogon Netman netprofm NetT
cpPortSharing NlaSvc nsi p2pimsvc p2psvc Parallels Coherence Service Paral
Iels Tools Service PcaSvc PeerDistSvc PerfHost pla PlugPlay PNRPAutoReg PN
RPsvc PolicyAgent Power PrintNotify ProfSvc QWAVE RasAuto RasMan RemoteAcc
ess RemoteRegistry RpcEptMapper RpcLocator RpcSs SamSs SCardSvr Schedule S
CPolicySvc SDRSvc seclogon SENS SensrSvc SessionEnv SharedAccess ShellHwDe
```

Wyniki działania poprzedniego polecenia zostały celowo obcięte, by zaoszczędzić nieco miejsca, ale mamy nadzieję, że zrozumiałeś, jak to działa. Oczywiście nie jest to do końca to, czego szukałeś, ale łącząc tę technikę z podwyrażeniami, które pokazaliśmy Ci nieco wcześniej w tej sekcji, powinieneś być w stanie z łatwością uzyskać zamierzony efekt.

18.6. Deklarowanie typu zmiennej

Do tej pory wstawialiśmy obiekty do zmiennych i pozwalaliśmy powłoce PowerShell samodzielnie określić, jakich typów obiektów używaliśmy. Powłoka PowerShell nie dba o to, jakiego typu obiekty umieszczasz w zmiennych, ale dla Ciebie może to być jednak ważne.

Załóżmy przykładowo, że masz zmienną, która powinna zawierać liczbę, ponieważ z użyciem tej wartości chcesz wykonać kilka obliczeń. Spójrzmy na przykład, który możesz wpisać bezpośrednio w wierszu poleceń:

```
PS C:\> $number = Read-Host "Wprowadź liczbę:"
Wprowadź liczbę: 100
PS C:\> $number = $number * 10
PS C:\> $number
100100100100100100100100100100
```

ZRÓB TO SAM Nie omawialiśmy jeszcze polecenia Read-Host — zrobimy to już w następnym rozdziale — ale jeżeli samodzielnie spróbujesz wykonać ten przykład, to jego działanie powinno być dla Ciebie oczywiste.

Co za licha? Jak to się stało, że w wyniku pomnożenia liczby 100 przez 10 otrzymujemy 100100100100100100100100100100? Co to za zwariowana nowa matematyka?

Jeżeli przyjrzałeś się uważnie wynikowi działania tego polecenia, z pewnością zauważyłeś, co się dzieje. Powłoka PowerShell nie traktuje wartości wpisanej przez użytkownika jak liczbę, tylko jako ciąg znaków, stąd zamiast mnożyć liczbę 100 razy 10, powłoka PowerShell *dziesięciokrotnie powiela ciąg znaków 100*. Rezultatem takiej operacji jest oczywiście ciąg znaków 100 powtórzony dziesięć razy z rzędu. Ups.

Na wszelki wypadek możemy jednak sprawdzić, czy powłoka rzeczywiście traktuje dane wejściowe jako ciąg znaków:

```
PS C:\> $number = Read-Host "Wprowadź liczbę: "
Enter a number: 100
PS C:\> $number | gm
TypeName: System.String
Name      MemberType      Definition
-----
Clone     Method          System.Object Clone()
CompareTo Method          int CompareTo(System.Object valu...
Contains  Method          bool Contains(string value)
```

No tak, przekazanie zmiennej \$number za pomocą potoku do polecenia Gm potwierdza, że powłoka „widzi” zawartość tej zmiennej jako obiekt typu System.String, a nie System.Int32. Istnieje kilka sposobów radzenia sobie z tym problemem, a my pokażemy Ci najłatwiejszy z nich.

Najpierw musimy poinformować powłokę, że zmienna \$number powinna zawierać liczbę całkowitą, co zmusi powłokę do przekonwertowania danych wpisywanych przez użytkownika na rzeczywistą liczbę. Pokazujemy to w poniższym przykładzie, gdzie w nawiasach kwadratowych bezpośrednio przed pierwszym użyciem zmiennej określamy żądany typ danych jako int:

```

PS C:\> [int]$number = Read-Host "Wprowadź liczbę: "
Enter a number: 100
PS C:\> $number | gm
    TypeName: System.Int32
Name      MemberType      Definition
-----
CompareTo  Method           int CompareTo(System.Object value), int CompareT...
Equals     Method           bool Equals(System.Object obj), bool Equals(int ...
GetHashCode Method           int GetHashCode()
GetType    Method           type GetType()
GetTypeCode Method           System.TypeCode GetTypeCode()
ToString   Method           string ToString(), string ToString(string format...
PS C:\> $number = $number * 10
PS C:\> $number
1000

```

1 Deklaracja zmiennej jako int

2 Potwierdzenie, że zmienna jest typu Int32

3 Zmienna została potraktowana jako liczba

W tym przykładzie używamy deklaracji typu `[int]` do wymuszenia, aby zmienna `$number` zawierała wyłącznie liczby całkowite ❶. Po wprowadzeniu danych wejściowych za pomocą potoku przekazujemy zmienną `$number` do polecenia `Gm`, aby potwierdzić, że jest to rzeczywiście liczba całkowita, a nie ciąg znaków ❷. Na koniec sprawdzamy, czy zmienna została potraktowana jako liczba i czy operacja mnożenia została wykonana prawidłowo ❸.

Inną korzyścią wynikającą z zastosowania tej techniki jest to, że jeżeli powłoka nie będzie w stanie przekształcić danych wejściowych na liczbę, wygenerowany zostanie błąd, ponieważ zmienna `$number` może przechowywać tylko liczby całkowite:

```

PS C:\> [int]$number = Read-Host "Wprowadź liczbę:"
Enter a number: Hello
Cannot convert value "Hello" to type "System.Int32". Error: "Input string
was not in a correct format."
At line:1 char:13
+ [int]$number <<<< = Read-Host "Enter a number"
    + CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
    + FullyQualifiedErrorId : RuntimeException

```

Jest to świetny przykład na to, jak zapobiegać potencjalnym problemom, ponieważ w takiej sytuacji masz pewność, że zmienna `$number` będzie zawierała dokładnie taki typ danych, jakiego oczekujesz.

Oczywiście w miejsce deklaracji `[int]` możesz użyć wielu innych typów obiektów; poniżej przedstawiamy listę najczęściej używanych typów:

- `[int]` — liczba całkowita;
- `[single]` i `[double]` — liczba zmiennoprzecinkowa o pojedynczej precyzji lub podwójnej precyzji (liczby z częścią dziesiętną);
- `[string]` — ciąg znaków;
- `[char]` — jeden znak (na przykład `[char]$c = 'X'`);
- `[xml]` — dokument XML; dowolny ciąg znaków przypisany do takiej zmiennej zostanie przeanalizowany, tak aby upewnić się, że zawiera poprawny kod XML (na przykład `[xml]$doc = Get-Content MyXML.xml`);

- [adsis] — zapytanie ADSI (ang. *Active Directory Service Interfaces*); powłoka wykona takie zapytanie i umieści wynikowy obiekt lub obiekty w zmiennej (na przykład [adsis]\$user = "WinNT:\\MYDOMAIN\Administrator,user").

Zadeklarowanie typu obiektu dla zmiennej jest świetnym sposobem na uniknięcie skomplikowanych błędów logicznych w bardziej złożonych skryptach. Jak pokazuje poniższy przykład, po określeniu typu obiektu powłoka PowerShell wymusza go, dopóki zmienna nie zostanie jawnie zadeklarowana jako zmienna innego typu:

```
PS C:\> [int]$x = 5 ← ❶ Deklarujemy zmienną $x typu int
PS C:\> $x = 'Hello' ← ❷ Błąd — przypisanie ciągu znaków do zmiennej $x
Cannot convert value "Hello" to type "System.Int32". Error: "Input string
was not in a correct format."
At line:1 char:3
+ $x <<< = 'Hello'
    + CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
    + FullyQualifiedErrorId : RuntimeException
PS C:\> [string]$x = 'Hello' ← ❸ Redeklaracja zmiennej $x jako string
PS C:\> $x | gm
TypeName: System.String ← ❹ Potwierdzenie nowego typu zmiennej
Name      MemberType Definition
-----
Clone     Method      System.Object Clone()
CompareTo Method      int CompareTo(System.Object valu...
```

W tym przykładzie rozpoczynamy od zadeklarowania zmiennej `$x` jako typu `[int]` ❶ i umieszczenia w niej liczby całkowitej. Kiedy próbujemy umieścić w niej ciąg znaków ❷, powłoka PowerShell zgłasza błąd, ponieważ nie może przekształcić tego konkretnego ciągu znaków w liczbę. Następnie dokonujemy redeklaracji zmiennej `$x` jako typu `[string]` i dzięki temu możemy w niej zapisać ciąg znaków ❸. Na koniec potwierdzamy nowy typ zmiennej, przekazując ją za pomocą potoku do polecenia `Gm` i sprawdzając nazwę jej typu ❹.

18.7. Polecenia do pracy ze zmiennymi

Jak widać, zaczęliśmy używać zmiennych bez konieczności ich formalnego zadeklarowania. Powłoka PowerShell nie wymaga zaawansowanej deklaracji zmiennych i nie można w niej wymusić konieczności ich deklarowania (użytkownicy mający doświadczenie w pracy z językiem VBScript, którzy szukają czegoś takiego jak dyrektywa `Option Explicit`, będą zapewne nieco rozczarowani; powłoka PowerShell ma co prawda polecenie o nazwie `Set-StrictMode`, ale to nie jest dokładnie to samo). Mimo wszystko powłoka PowerShell posiada następujące polecenia przeznaczone do pracy ze zmiennymi:

- `New-Variable`,
- `Set-Variable`,
- `Remove-Variable`,
- `Get-Variable`,
- `Clear-Variable`.

W praktyce nie musisz używać żadnego z nich, z wyjątkiem być może polecenia `Remove-Variable`, które jest przydatne do trwałego usuwania zmiennych (aby to zrobić, możesz również przejść na „dysk” `VARIABLE:` i użyć polecenia `del`). Wszystkie inne operacje, takie jak tworzenie nowych zmiennych czy przypisywanie i odczytywanie ich wartości, możesz wykonywać, stosując składnię, której używaliśmy do tej pory w tym rozdziale; korzystanie z wymienionych wyżej poleceń w większości przypadków nie daje żadnych wymiernych korzyści.

Jeśli zdecydujesz się na używanie tych poleceń, nazwę zmiennej przekazujesz do wybranego polecenia za pośrednictwem parametru `-name`. Jest to *tylko nazwa zmiennej* — nie dodajemy tutaj znaku dolara. W zasadzie wspomniane wyżej polecenia mogą być przydatne tylko w jednej sytuacji, podczas pracy ze zmiennymi typu *out-of-scope*. Korzystanie z takich zmiennych jest jednak kiepską praktyką i w tej książce nie będziemy już poruszać tego tematu (w razie potrzeby więcej szczegółowych informacji na temat takich zmiennych znajdziesz w pliku pomocy `about_scope`).

18.8. Dobre praktyki w pracy ze zmiennymi

O większości dobrych praktyk w pracy ze zmiennymi już wspominaliśmy, ale jest to właściwy moment, aby je szybko przypomnieć:

- Nazwy zmiennych powinny być opisowe, ale jednocześnie zwięzłe. Przykładowo, `$computername` jest świetną nazwą zmiennej, ponieważ jest ona jasna i zwięzła, ale już `$c` nie będzie najlepszym wyborem, ponieważ na podstawie jej nazwy nie możemy powiedzieć niczego na temat jej przeznaczenia. Z kolei zmienna `$computer_to_query_for_data` ma nazwę najzupełniej poprawną i opisową, ale jest jednak nieco przydługa. Jasne, jej przeznaczenie jest oczywiste, ale czy chciałbyś co chwilę mozolnie „wklepywać” taką długą nazwę?
- Nie używaj spacji w nazwach zmiennych. Wiemy, że jest to dozwolone i że wiesz, jak to zrobić, ale w praktyce nie sprawdza się to najlepiej.
- Jeżeli zmienna zawiera tylko jeden rodzaj obiektu, zadeklaruj jej typ podczas pierwszego użycia — może Ci to później pomóc uniknąć trudnych do znalezienia błędów logicznych; co więcej, jeżeli pracujesz w komercyjnym środowisku programowania skryptów (na przykład takim jak PowerShell Studio), to w sytuacji, gdy typ zmiennej jest z góry zadeklarowany, oprogramowanie edytora często może dostarczać użytecznych wskazówek i odpowiedzi do kodu.

18.9. Najczęściej spotykane problemy

Najczęstszym problemem, z jakim borykają się nasi studenci, są nazwy zmiennych. Mamy nadzieję, że udało nam się to dokładnie wyjaśnić w tym rozdziale, ale zawsze powinniśmy pamiętać, że znak dolara *nie jest częścią nazwy zmiennej*. Jest to wskazówka dla powłoki, że chcesz uzyskać dostęp do *zawartości* zmiennej; to, co następuje po znaku dolara, jest traktowane przez powłokę jako nazwa zmiennej.

Powłoka PowerShell posiada dwie reguły parsowania, które pozwalają jej rozpoznawać nazwy zmiennej:

- Jeżeli znak znajdujący zaraz po znaku dolara jest literą, cyfrą lub znakiem podkreślenia, nazwa zmiennej składa się ze wszystkich znaków następujących po znaku dolara, aż do następnej białej spacji (która może być spacją, tabulatorem lub znakiem powrotu karetki).
- Jeżeli znak występujący zaraz po znaku dolara jest otwierającym nawiasem klamrowym {, nazwa zmiennej składa się ze wszystkiego, co znajduje się po tym nawiasie klamrowym, aż do zamykającego nawiasu klamrowego }.

18.10. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Wróć do rozdziału 15. i odśwież sobie niektóre zagadnienia dotyczące pracy z zadaniami działającymi w tle. Następnie z poziomu wiersza poleceń wykonaj poniższe ćwiczenia:

1. Utwórz w tle zadanie, które wysyła zapytanie WMI do klasy Win32_BIOS dwóch komputerów zdalnych (jeżeli masz do dyspozycji tylko jeden komputer, dwukrotnie użyj adresu localhost).
2. Kiedy zadanie zakończy działanie, zapisz jego wyniki w zmiennej.
3. Wyświetl zawartość tej zmiennej.
4. Wyeksportuj zawartość zmiennej do pliku CliXML.

18.11. Co dalej?

Spróbuj poświęcić kilka chwil na przejrzanie niektórych poprzednich rozdziałów tej książki. Biorąc pod uwagę, że zmienne są przede wszystkim zaprojektowane do przechowywania czegoś, co możesz użyć więcej niż raz, to, czy możesz znaleźć zastosowanie dla zmiennych w tematach omawianych w poprzednich rozdziałach?

Na przykład w rozdziale 13. nauczyłeś się tworzyć połączenia z komputerami zdalnymi. W tym rozdziale tworzyłeś, stosowałeś i zamykałeś połączenia mniej lub bardziej w jednym kroku; czy nie byłoby zatem użyteczne utworzenie połączenia, przechowanie go w zmiennej i używanie dla kilku poleceń? Jest to tylko jedna z wielu sytuacji, w których zmienne mogą być bardzo przydatne (w rozdziale 20. pokażemy Ci, jak to zrobić w praktyce). Zastanów, czy możesz znaleźć więcej przykładów zastosowań zmiennych w poprzednich rozdziałach.

18.12. Odpowiedzi

1. PS C:\> invoke-command {get-wmiobject win32_bios} -computername
↳localhost,\$env:computername -asjob
2. PS C:\>\$results=Receive-Job 4 -keep
3. PS C:\>\$results
4. PS C:\>\$results | export-clipboard bios.xml

19

Wejście i wyjście

Do tej pory w książce bazowaliśmy głównie na wbudowanych mechanizmach powłoki PowerShell pozwalających na generowanie tabel i list. Kiedy jednak zaczniesz łączyć polecenia w bardziej złożone skrypty, prawdopodobnie będziesz chciał również uzyskać bardziej precyzyjną kontrolę nad tym, co jest wyświetlane na ekranie. Najprawdopodobniej będziesz również prosić użytkownika o wprowadzanie takich czy innych danych. W tym rozdziale dowiesz się, jak pobierać dane wejściowe i jak wyświetlać wyniki działania.

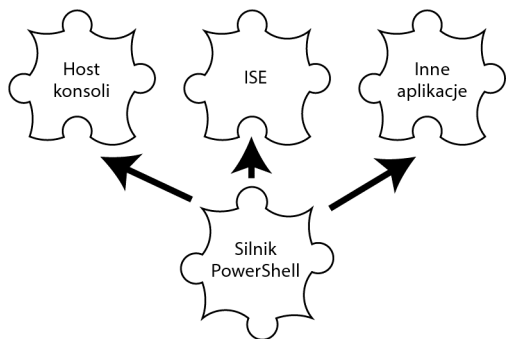
Chcemy jednak podkreślić, że zagadnienia omawiane w tym rozdziale będą użyteczne tylko w przypadku skryptów, które wchodzi w bezpośrednią interakcję z użytkownikiem. Dla skryptów działających bez nadzoru nie są to odpowiednie techniki, ponieważ takie skrypty nie wymagają udziału użytkownika.

19.1. Pobieranie i wyświetlanie informacji

Sposób, w jaki powłoka PowerShell pobiera informacje od użytkownika oraz wyświetla je na ekranie, zależy od sposobu jej uruchomienia. Dzieje się tak dlatego, że powłokę PowerShell możesz traktować jako rodzaj silnika ukrytego pod maską samochodu.

To, z czym się kontaktujesz podczas pracy z powłoką, nosi nazwę **aplikacji hosta**. Konsola wiersza polecenia widoczna podczas uruchamiania programu PowerShell.exe jest często nazywana **hostem konsoli** lub po prostu **konsolą tekstową**. Graficzna wersja programu PowerShell ISE jest zwykle nazywana **środowiskiem ISE**, **hostem graficznym** lub **konsolą graficzną**. Inne aplikacje spoza firmy Microsoft również mogą korzystać z silnika powłoki PowerShell. W takim przypadku pracujesz z aplikacją hosta, która przekazuje Twoje polecenia do silnika PowerShell, a następnie wyświetla wyniki ich działania.

Rysunek 19.1 ilustruje zależności między silnikiem powłoki PowerShell a różnymi aplikacjami hosta. Każda aplikacja hosta jest odpowiedzialna za fizyczną prezentację danych wyjściowych generowanych przez silnik powłoki, a także za pobieranie danych wejściowych, o które prosi silnik. Oznacza to, że powłoka PowerShell może wyświetlać dane wyjściowe i pobierać dane wejściowe na różne sposoby. W rzeczywistości host konsoli i środowisko ISE używają różnych metod pobierania danych wejściowych: host konsoli wyświetla monit w wierszu poleceń, a środowisko ISE wyświetla na ekranie okno dialogowe z polem wprowadzania tekstu i przyciskiem OK.



Rysunek 19.1. Silnik powłoki PowerShell może być wykorzystywany przez wiele różnych aplikacji

Chcemy zwrócić uwagę na te różnice, ponieważ mogą one czasami sprawiać kłopoty mniej doświadczonym użytkownikom. Dlaczego jedno polecenie zachowuje się w dany sposób w konsoli tekstowej, a w środowisku ISE działa zupełnie inaczej? Dzieje się tak dlatego, że sposób interakcji z powłoką zależy od aplikacji hosta, a nie od samej powłoki PowerShell. Polecenia, które mamy zamiar Ci pokazać, wykazują nieco inne zachowanie w zależności od tego, w jakim środowisku zostaną uruchomione.

19.2. Polecenie Read-Host

Polecenie Read-Host powłoki PowerShell służy do wyświetlania odpowiedniego komunikatu (monitu) i pobierania danych wejściowych od użytkownika. Ponieważ polecenia tego po raz pierwszy użyliśmy już w poprzednim rozdziale, jego składnia może Ci się wydawać znajoma:

```
PS C:\> read-host "Podaj nazwę komputera"
Podaj nazwę komputera: SERVER-R2
SERVER-R2
```

W tym przykładzie przedstawiono dwie ważne właściwości tego polecenia:

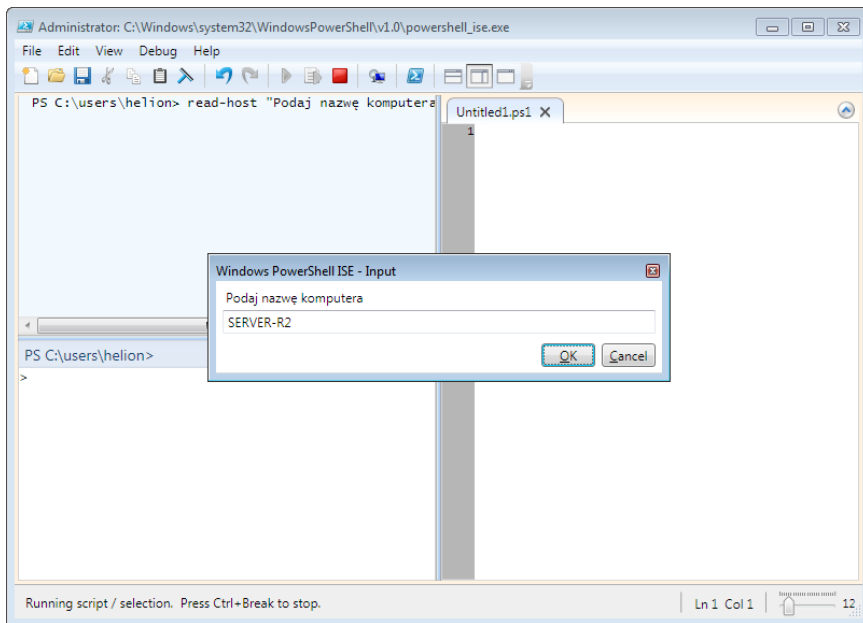
- Na końcu komunikatu automatycznie zostaje dodany dwukropek.
- Wynikiem działania tego polecenia jest ciąg znaków wpisany przez użytkownika (technicznie rzecz biorąc, taki ciąg znaków jest umieszczany w potoku).

Bardzo często przechwytujemy dane wejściowe i zapisujemy je w zmiennej, tak jak to zostało pokazane poniżej:

```
PS C:\> $computername = read-host "Podaj nazwę komputera"
Podaj nazwę komputera: SERVER-R2
```

ZRÓB TO SAM Nadszedł czas, abyś zaczął samodzielnie wykonywać polecenia z omawianych tutaj przykładów. W tym momencie w zmiennej `$computername` powinieneś mieć już zapisaną poprawną nazwę komputera. Nie używaj nazwy `SERVER-R2`, chyba że jest to prawdziwa nazwa komputera, na którym pracujesz.

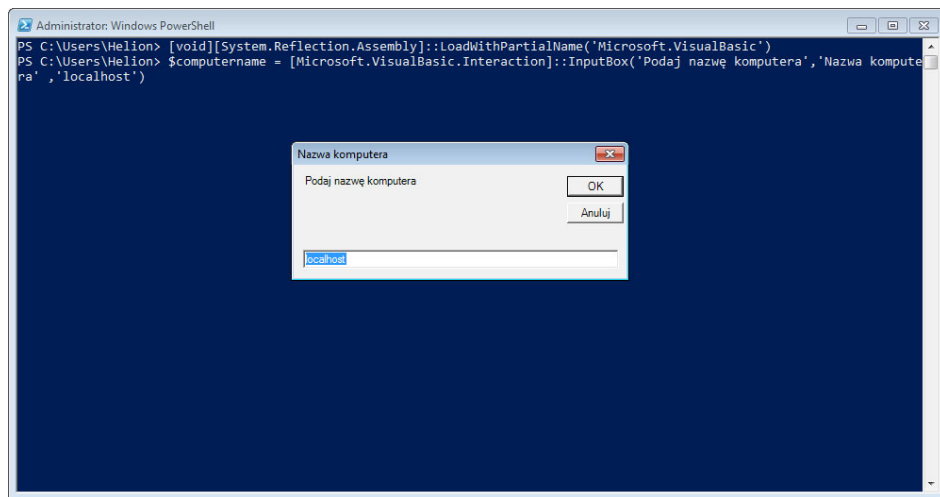
Jak już wspominaliśmy wcześniej, środowisko ISE powłoki PowerShell v2 zamiast w wierszu polecenia wyświetla monit w oknie dialogowym, jak pokazano na rysunku 19.2. Każda aplikacja hosta — w tym edytory skryptów, takie jak PowerGUI, PowerShell Plus czy PowerShell Studio — ma własny sposób implementacji polecenia `Read-Host`. Zauważ, że środowisko ISE powłoki PowerShell v3 (i jej nowszych wersji), które używa uproszczonego układu dwóch paneli (w porównaniu z ISE v2), wyświetla wiersz polecenia podobny do zwykłego okna konsoli.



Rysunek 19.2. Środowisko ISE v2 dla polecenia `Read-Host` wyświetla okno dialogowe

Nie ma wiele więcej do powiedzenia na temat polecenia `Read-Host` — jest ono bardzo użyteczne, ale nie ma w nim niczego szczególnie ekscytującego. Co ciekawe, kiedy wprowadziliśmy polecenie `Read-Host` do materiałów omawianych w większości naszych klas, niemal zawsze któryś ze studentów pytał nas, czy istnieje sposób, aby zawsze używać graficznego okna dialogowego do wprowadzania danych. Wiele administratorów przygo-

towujących skrypty dla swoich użytkowników nie chciałoby, aby wprowadzali oni niezbędne informacje w wierszu poleceń konsoli tekstowej (w końcu nie jest to przecież „windowsowy” sposób załatwiania takich spraw). Odpowiedź na takie pytanie brzmi oczywiście „tak”, ale nie jest to całkiem proste. Ostateczny rezultat pokazano na rysunku 19.3.



Rysunek 19.3. Tworzenie graficznego okna wprowadzania danych

Aby utworzyć graficzne okno wprowadzania danych, musimy skorzystać ze środowiska .NET Framework. Zaczniemy od wykonania następującego polecenia:

```
PS C:\> [void][System.Reflection.Assembly]::LoadWithPartialName('Microsoft.VisualBasic')
```

Trzeba to zrobić tylko raz w danej sesji powłoki, ale z pewnością nie zaszkodzi, jeżeli przypadkowo zrobisz to po raz drugi.

Przedstawione polecenie ładuje część klas środowiska .NET Framework, Microsoft.VisualBasic, którego powłoka PowerShell nie ładuje automatycznie. W tych klasach znajduje się większość środowiska Visual Basic, w tym takie elementy jak okna dialogowe czy pola wprowadzania danych (pola tekstowe).

Przyjrzyjmy się zatem, co robi nasze polecenie:

- Słowo kluczowe `[void]` przekształca wynik działania polecenia w dane typu `void`. W poprzednim rozdziale dowiedziałeś się, jak wykonywać tego rodzaju konwersję na liczby całkowite; typ `void` to specjalny typ danych, który tutaj oznacza mniej więcej tyle co „zignoruj wyniki”. Nie chcemy widzieć wyniku działania tego polecenia, więc przekształcamy wynik na typ `void`. Innym sposobem na osiągnięcie takiego efektu byłoby przekazanie wyniku za pomocą potoku do polecenia `Out-Null`.
- Następnie odwołujemy się do klasy `System.Reflection.Assembly`, która reprezentuje naszą aplikację (w tym przypadku jest nią powłoka PowerShell). Nazwę klasy wpisujemy w nawiasach kwadratowych, tak jakbyśmy rozpoczynali deklarację

zmiennej, z tym że zamiast nazwy zmiennej wstawiamy dwa dwukropki, za pomocą których uzyskujemy dostęp do **statycznej metody** tej klasy (ang. *static method*). Metody statyczne są dostępne bez konieczności tworzenia instancji danej klasy.

- Wywoływana metoda statyczna nosi nazwę `LoadWithPartialName()`, a jej argumentem wywołania jest nazwa komponentu, który chcemy załadować.

Jeżeli to wszystko wydaje Ci się bardzo zagmatwane, to nie przejmuj się; po prostu zastosuj to polecenie i wszystko będzie dobrze. Po załadowaniu odpowiednich komponentów środowiska .NET możemy ich użyć w następujący sposób:

```
PS C:\> $computername = [Microsoft.VisualBasic.Interaction]::InputBox('Podaj nazwę  
→komputera','Nazwa komputera','localhost')
```

W tym przykładzie używamy innej metody statycznej, tym razem z klasy `Microsoft.VisualBasic.Interaction`, którą załadowaliśmy do pamięci za pomocą poprzedniego polecenia. I znów, jeżeli nie bardzo rozumiesz, czym są i jak działają „metody statyczne”, nie martw się — po prostu używaj tego polecenia tak, jak to pokazaliśmy w przykładzie.

Elementami, które możesz zmieniać, są trzy zmienne będące parametrami wywołania metody `InputBox()`:

- Pierwszy parametr to tekst zachęty.
- Drugi parametr to tytuł okna dialogowego.
- Trzeci parametr możesz pozostawić pusty lub całkowicie go pominąć; jest to domyślna wartość, która wstępnie pojawia się w polu wprowadzania danych.

Jak widać, używanie polecenia `Read-Host` jest znacznie mniej skomplikowane niż wykonywanie procedury opisanej powyżej, ale jeżeli naprawdę chcesz wprowadzać dane w oknie dialogowym, to możesz je utworzyć w taki właśnie sposób.

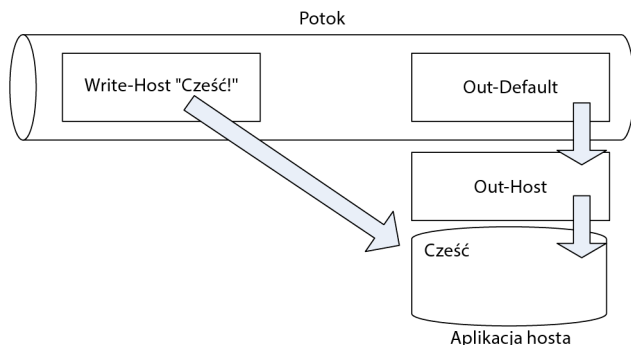
19.3. Polecenie *Write-Host*

Teraz, gdy możesz pobierać dane wejściowe, będziesz potrzebował jakiegoś sposobu wyświetlania danych wyjściowych. Jednym z takich sposobów jest polecenie `Write-Host`. Nie zawsze jest to najlepsza metoda, ale zawsze dostępna, dlatego tak ważne jest, abyś zrozumiał działanie tego polecenia.

Jak pokazuje rysunek 19.4, polecenie `Write-Host` działa w potoku, tak jak każde inne polecenie powłoki, z tym że samo nie przekazuje już dalej do potoku żadnych danych — zamiast tego wyświetla je bezpośrednio na ekranie aplikacji hosta, dzięki czemu za pomocą parametrów wywołania `-foregroundColor` i `-backgroundColor` możemy zmieniać kolor czcionki i tła wyświetlanego tekstu.

```
PS C:\> write-host "KOLOROWO!" -fore yellow -back magenta  
KOLOROWO!
```

ZRÓB TO SAM Wypróbuj przedstawione wyżej polecenie i zobacz jego wyniki działania w kolorze!



Rysunek 19.4. Polecenie Write-Host pomija potok i wyświetla dane bezpośrednio na ekranie aplikacji hosta

UWAGA Nie każda aplikacja wykorzystująca powłokę PowerShell obsługuje alternatywne kolory tekstu i nie każda aplikacja obsługuje pełny zestaw kolorów. Kiedy próbujesz zmieniać kolory tekstu, taka aplikacja zwykle ignoruje wszystkie kolory, które nie są lub nie mogą być wyświetlane. To jeden z powodów, dla których staramy się unikać nadawania kolorom nadmiernego znaczenia w aplikacjach powłoki.

Polecenia Write-Host powinieneś używać tylko wtedy, gdy chcesz wyświetlić konkretną wiadomość, być może z użyciem koloru, który ułatwi zwrócenie na nią uwagę. Ale nie jest to właściwy sposób na wyświetlanie normalnych wyników działania skryptu lub polecenia. Powinieneś również pamiętać, że informacje wyświetlane na ekranie przez polecenia -Host *nie mogą być przechwytywane*. Jeżeli uruchomisz skrypt zdalnie lub bez nadzoru, polecenia -Host nie będą działały tak, jak mogłoby Ci się wydawać. Jak napisaliśmy na początku tego rozdziału, polecenia -Host są przeznaczone wyłącznie do interakcji bezpośrednio z użytkownikami, co powinno być bardzo ograniczonym podzbiorem zadań wykonywanych w powłoce PowerShell.

Na przykład nigdy nie powinieneś używać polecenia Write-Host do ręcznego formatowania tabeli; możesz znaleźć lepsze sposoby tworzenia wyników, używając innych technik, które umożliwiają powłoce PowerShell wykorzystanie mechanizmów formatowania. Nie będziemy ich jednak tutaj omawiać, ponieważ są bardziej przydatne dla programistów tworzących złożone skrypty i narzędzia. Jeżeli jednak chcesz, możesz zajrzeć do książki *Learn PowerShell Toolmaking in a Month of Lunches* (wyd. Manning, 2012), gdzie znajdziesz szczegółowy opis takich technik. Polecenie Write-Host nie jest również najlepszym sposobem generowania komunikatów o błędach, ostrzeżeń, komunikatów debugowania i tak dalej — podobnie jak poprzednio, można znaleźć znacznie bardziej efektywne sposoby ich realizacji (wiele z nich poznasz już w tym rozdziale). Jeżeli korzystasz z powłoki PowerShell w poprawny sposób, prawdopodobnie nie będziesz musiał zbyt często używać polecenia Write-Host.

UWAGA Bardzo często spotykamy użytkowników, którzy używają polecenia `Write-Host` do wyświetlania bieżących komunikatów, takich jak „Nawiązywanie połączenia z SERVER2” czy „Sprawdzanie zawartości folderu”. Nie rób tego. Znacznie bardziej odpowiednim sposobem wyświetlania takich komunikatów jest użycie polecenia `Write-Verbose`.

Dla zainteresowanych

Więcej szczegółowych informacji na temat polecenia `Write-Verbose` i innych poleceń z rodziny `Write` znajdziesz w rozdziale 22. Jeżeli jednak jesteś nieco niecierpliwy i już teraz próbujesz działanie polecenia `Write-Verbose`, możesz się rozczarować, gdy odkryjesz, że nie wyświetla ono żadnych wyników — a przynajmniej nie robi tego domyślnie.

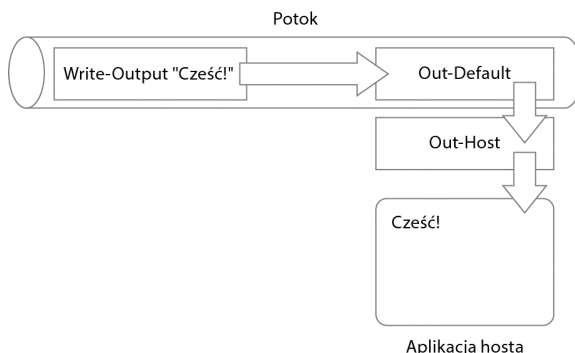
Jeśli zamierzasz używać poleceń z rodziny `Write`, musisz najpierw je włączyć. Na przykład aby włączyć polecenie `Write-Verbose`, powinieneś najpierw ustawić zmienną `$VerbosePreference` na wartość `"Continue"`; aby wyłączyć jego działanie, powinieneś ustawić zmienną `$VerbosePreference="SilentlyContinue"`. Podobne zmienne znajdziesz również dla poleceń `Write-Debug` (zmienna `$DebugPreference`) i `Write-Warning` (zmienna `$WarningPreference`).

W rozdziale 22. zaprezentujemy jeszcze lepszy sposób zastosowania polecenia `Write-Verbose`.

Używanie polecenia `Write-Host` może się wydawać *znacznie* łatwiejsze, więc oczywiście jeżeli chcesz, możesz z niego korzystać. Powinieneś jednak pamiętać, że stosowanie innych poleceń, takich jak `Write-Verbose`, wymusza nieco bardziej „powershellowy” sposób postępowania i zapewnia bardziej spójną pracę z powłoką.

19.4. Polecenie *Write-Output*

W przeciwieństwie do polecenia `Write-Host` polecenie `Write-Output` może przekazywać obiekty do potoku. Ze względu jednak na fakt, że polecenie to nie wysyła danych bezpośrednio na ekran, nie pozwala na określenie kolorów tekstu ani innych atrybutów. W rzeczywistości polecenie `Write-Output` (lub jego alias `Write`) w ogóle nie jest zaprojektowane do wyświetlania danych. Zamiast tego przekazuje dane do potoku i dopiero potok ostatecznie zajmuje się ich wyświetlaniem. Rysunek 19.5 pokazuje, jak to działa.



Rysunek 19.5. Polecenie `Write-Output` umieszcza obiekty w potoku, co w niektórych przypadkach powoduje ich wyświetlenie na ekranie

W rozdziale 10. zamieściliśmy krótki opis sposobu, w jaki obiekty przechodzą z potoku na ekran. Przyjrzyjmy się, jak to działa:

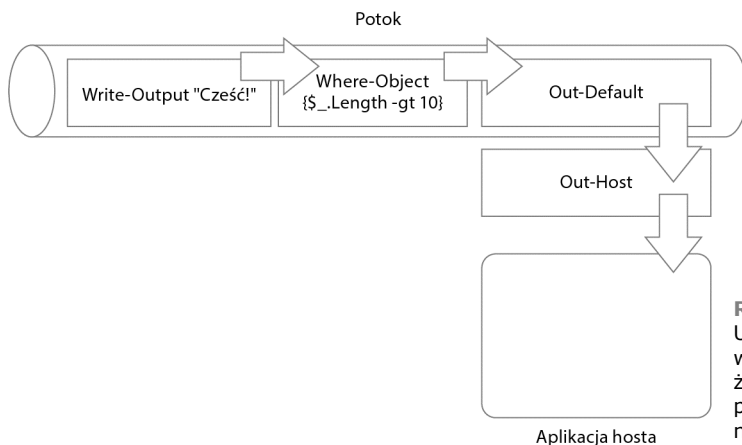
1. Polecenie `Write-Output` umieszcza ciąg znaków *Cześć!* (czyli obiekt typu `String`) w potoku.
2. Ponieważ w potoku nie ma niczego innego, ciąg znaków *Cześć!* jest przekazywany na koniec potoku, gdzie zawsze znajduje się polecenie `Out-Default`.
3. Polecenie `Out-Default` przekazuje otrzymany obiekt do polecenia `Out-Host`.
4. Polecenie `Out-Host` wysyła do systemu formatowania powłoki PowerShell żądanie sformatowania obiektu. Ponieważ w tym przykładzie jest to prosty obiekt typu `String`, system formatowania zwraca odpowiednio sformatowany ciąg znaków.
5. Polecenie `Out-Host` umieszcza otrzymany wynik na ekranie.

Wyniki są podobne do tego, co można uzyskać za pomocą polecenia `Write-Host`, ale obiekty docierają na ekran zupełnie inną trasą. Jest to bardzo ważne, ponieważ w potoku mogą znajdować się również inne elementy. Na przykład rozważ następujące polecenie (oczywiście możesz, a w zasadzie nawet powinieneś je samodzielnie wypróbować):

```
PS C:\> write-output "Cześć!" | where-object { $_.length -gt 10 }
```

Na ekranie nie zobaczymy żadnych wyników działania polecenia, a rysunek 19.6 ilustruje tego przyczynę. Ciąg znaków *Cześć!* zostaje przekazany do potoku, ale zanim dojdzie do cmdletu `Out-Default`, musi przejść przez polecenie `Where-Object`, które odfiltruje wszystko, co ma właściwość `Length` o wartości mniejszej lub równej 10, a w tym przypadku obejmuje to również nasz obiekt. Ciąg znaków *Cześć!* zostaje zatem usunięty z potoku, a ponieważ w takiej sytuacji w potoku nie ma już żadnych danych, polecenie `Out-Default` nie przekazuje już nic do cmdletu `Out-Host`, więc nic także nie jest wyświetlane na ekranie. Porównaj to polecenie z następującym:

```
PS C:\> write-host "Cześć!" | where-object { $_.length -gt 10 }  
Cześć!
```



Rysunek 19.6.
Umieszczenie obiektów
w potoku oznacza,
że można je filtrować
przed wyświetleniem
na ekranie

W tym przykładzie zastąpiliśmy tylko polecenie Write-Output poleceniem Write-Host. Tym razem jednak ciąg znaków *Cześć!* trafia bezpośrednio na ekran, a nie do potoku. Polecenie Where-Object nie otrzymuje żadnych danych wejściowych, więc nie generuje żadnych wyników działania i dlatego nic nie jest wyświetlane przez polecenia Out-Default i Out-Host. Zamiast tego nasz ciąg znaków zostaje bezpośrednio wysłany na ekran, dzięki czemu możemy go zobaczyć.

Polecenie Write-Output może Ci się wydawać nowe, ale w rzeczywistości używasz go już od dawna. Dzieje się tak dlatego, że jest to domyślny cmdlet powłoki. Kiedy pracując z powłoką, próbujesz wykonać coś, co nie jest poleceniem, powłoka przekazuje wszystko, co wpisałeś, do polecenia Write-Output.

19.5. Inne sposoby wyświetlania danych

Powłoka PowerShell ma jeszcze kilka innych poleceń pozwalających na wyświetlanie danych. Żadne z tych poleceń nie przekazuje jednak danych do potoku, tak jak to robi polecenie Write-Output; zamiast tego polecenia te działają trochę bardziej jak polecenie Write-Host. Warto jednak zauważyć, że każde z nich wyświetla dane w sposób, który można wyłączyć.

Powłoka posiada wbudowane zmienne konfiguracyjne dla każdej z tych alternatywnych metod wyświetlania. Gdy odpowiednia zmienna konfiguracyjna jest ustawiona na wartość Continue, polecenia, które będziemy za chwilę omawiać, faktycznie wyświetlają dane na ekranie. Gdy jednak zmienna konfiguracyjna jest ustawiona na wartość SilentlyContinue, skojarzone z nią polecenie nie wyświetla żadnych danych. W tabeli 19.1 zamieszczamy listę tych poleceń.

Tabela 19.1. Alternatywne polecenia pozwalające na wyświetlanie danych

Cmdlet	Przeznaczenie	Zmienna konfiguracyjna
Write-Warning	Wyświetla tekst ostrzeżenia; domyślnie w kolorze żółtym, poprzedzony etykietą <i>WARNING:</i> .	\$WarningPreference (domyślna wartość: Continue)
Write-Verbose	Wyświetla dodatkowy tekst informacyjny; domyślnie w kolorze żółtym, poprzedzony etykietą <i>VERBOSE:</i> .	\$VerbosePreference (domyślna wartość: SilentlyContinue)
Write-Debug	Wyświetla tekst debugowania; domyślnie w kolorze żółtym, poprzedzony etykietą <i>DEBUG:</i> .	\$DebugPreference (domyślna wartość: SilentlyContinue)
Write-Error	Wyświetla komunikat o błędzie.	\$ErrorActionPreference (domyślna wartość: Continue)

UWAGA W powłoce PowerShell v5 wprowadzone zostało nowe polecenie Write-Information, które zapisuje dane w unikatowym, uporządkowanym strumieniu informacji, umożliwiając w ten sposób wyświetlanie zarówno danych strukturalnych, jak i komunikatów informacyjnych. W powłoce PowerShell w wersji 5 i nowszych polecenie Write-Host do wyświetlania danych tak naprawdę „po cichu” wykorzystuje polecenie Write-Information. Więcej szczegółowych informacji na temat tego polecenia znajdziesz na stronie <https://technet.microsoft.com/en-us/library/dn998020.aspx>.

Polecenie `Write-Error` działa nieco inaczej, ponieważ przekazuje komunikaty o błędach bezpośrednio do strumienia błędów powłoki PowerShell.

Powłoka PowerShell posiada również polecenie `Write-Progress`, przy użyciu którego można wyświetlać paski postępu, ale działa ono zupełnie inaczej. W tej książce nie będziemy się nim zajmować; więcej szczegółowych informacji na temat tego polecenia znajdziesz w jego pliku pomocy.

Aby użyć dowolnego z wymienionych wyżej poleceń, powinieneś najpierw upewnić się, że skojarzona z nim zmienna konfiguracyjna jest ustawiona na wartość `Continue`. Jeżeli dana zmienna jest ustawiona na wartość `SilentlyContinue` (która jest domyślną wartością dla kilku z nich), na ekranie nie zobaczysz żadnych wyników działania. Po sprawdzeniu i ewentualnym ustawieniu odpowiedniej zmiennej możesz używać wybranych poleceń do wyświetlania komunikatów.

UWAGA Niektóre aplikacje hosta wykorzystujące powłokę PowerShell mogą wyświetlać dane wyjściowe z tych poleceń w innym miejscu. W PowerShell Studio firmy SAPIEN na przykład komunikaty debugowania są zapisywane w innym panelu niż główne wyjście skryptu, co pozwala na ich łatwiejszą analizę. W tej książce nie będziemy zajmować się zagadnieniami związanymi z debugowaniem, ale w razie potrzeby więcej szczegółowych informacji na ten temat znajdziesz w plikach pomocy powłoki PowerShell.

19.6. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Polecenia `Write-Host` i `Write-Output` mogą być nieco trudne w obsłudze. Zobacz, ile z tych zadań możesz wykonać samodzielnie. Jeżeli gdzieś utkniesz, zawsze możesz zajrzeć do przykładowych odpowiedzi dostępnych na końcu tego rozdziału.

1. Użyj polecenia `Write-Output`, aby wyświetlić wynik mnożenia liczby 100 przez 10.
2. Użyj polecenia `Write-Host`, aby wyświetlić wynik mnożenia liczby 100 przez 10.
3. Poproś użytkownika o podanie imienia, a następnie wyświetl to imię przy użyciu żółtej czcionki.
4. Poproś użytkownika o podanie imienia, a następnie wyświetl to imię, ale tylko jeżeli składa się z co najmniej sześciu znaków. Zrób to wszystko za pomocą pojedynczego polecenia powłoki PowerShell i nie używaj zmiennych.

To wszystko w tym zestawie ćwiczeń. Ponieważ używane tutaj polecenia są relatywnie proste, chcielibyśmy, abyś spędzał więcej czasu na samodzielnym eksperymentowaniu. Pamiętaj, aby to zrobić — w następnym podrozdziale przedstawimy Ci kilka pomysłów.

ZRÓB TO SAM Po ukończeniu ćwiczeń w tym rozdziale spróbuj wykonać trzeci zestaw ćwiczeń sprawdzających, który znajdziesz w dodatku do tej książki.

19.7. Co dalej?

Spróbuj poświęcić trochę czasu na zapoznanie się ze wszystkimi poleceniami omawianymi w tym rozdziale. Upewnij się, że potrafisz wyświetlać dane wyjściowe, pobierać dane wejściowe czy używać graficznych okien dialogowych do pobierania danych. Z poleceń omawianych w tym rozdziale będziesz korzystał bardzo często, powinieneś więc uważnie przeczytać ich pliki pomocy, a nawet zanotować sobie krótkie przypomnienia ich składni, aby móc później efektywnie ich używać.

19.8. Odpowiedzi

1. `write-output (100*10)`
lub po prostu w wierszu poleceń wpisz następującą formułę: `100*10`
2. Dowolne z tych poleceń będzie działać:
`$a=100*10`
`Write-Host $a`
`Write-Host "Wynikiem mnożenia 100*10 jest liczba $a"`
`Write-Host (100*10)`
3. `$name=Read-Host "Podaj imię" Write-host $name -ForegroundColor Yellow`
4. `Read-Host "Podaj imię" | where {$_length -gt 5}`

Sesje — ułatwienie komunikacji zdalnej

W rozdziale 13. omawialiśmy szereg zagadnień związanych z mechanizmami komunikacji zdalnej powłoki PowerShell. W tamtym rozdziale używaliśmy dwóch podstawowych poleceń — `Invoke-Command` i `Enter-PSSession` — do uzyskania zdalnego dostępu typu jeden-do-wielu i jeden-do-jednego. Oba polecenia działają w podobny sposób: tworzą nowe połączenie zdalne, wykonują określone zadanie i następnie zamykają to połączenie.

Oczywiście nie ma w tym nic złego, ale w praktyce to ciągle wklepywanie nazw komputerów, odwołań, alternatywnych numerów portów i tak dalej może być po prostu męczące. W tym rozdziale przyjrzymy się łatwiejszemu sposobowi rozwiązania problemu dostępu zdalnego. Dowiesz się również o trzecim sposobie nawiązywania połączeń zdalnych, który również może być bardzo przydatny.

20.1. Ułatwienie komunikacji zdalnej z użyciem powłoki PowerShell

Za każdym razem, gdy musisz połączyć się z komputerem zdalnym za pomocą polecenia `Invoke-Command` lub `Enter-PSSession`, musisz podać nazwę takiego komputera (lub ich nazwy, jeżeli wywołujesz polecenie na wielu komputerach). W zależności od środowiska, w jakim pracujesz, niezbędne może się okazać również podanie alternatywnych danych uwierzytelniających, co oznacza, że powłoka poprosi Cię o podanie hasła. W zależności od tego, w jaki sposób Twoja firma czy organizacja skonfigurowała mechanizmy zdalnego dostępu, konieczne może być także określenie alternatywnych numerów portów lub wybranie odpowiedniego mechanizmu uwierzytelniania.

Oczywiście nie ma w tym nic trudnego, aczkolwiek częste powtarzanie takiego procesu może być na dłuższą metę nieco uciążliwe. Na szczęście znamy lepszy sposób — zdalne sesje wielokrotnego użytku.

20.2. Tworzenie sesji wielokrotnego użytku i praca z nimi

Sesja to trwałe połączenie między Twoją lokalną powłoką PowerShell a jej odpowiednikiem działającym na komputerze zdalnym. Kiedy taka sesja jest aktywna, zarówno Twój komputer lokalny, jak i komputer zdalny poświęcają niewielką ilość zasobów systemowych na ciągłe utrzymywanie połączenia, które pozostaje otwarte nawet w sytuacji, kiedy ruch sieciowy na tym połączeniu jest minimalny lub nie ma go wcale. Powłoka PowerShell utrzymuje listę wszystkich sesji, które otworzyłeś, dzięki czemu możesz w dowolnym momencie ich użyć do wywołania polecenia lub rozpoczęcia pracy z powłoką zdalną.

Aby utworzyć nową sesję, powinieneś użyć polecenia `New-PSSession`. Podaj nazwę komputera zdalnego (lub ich nazwy, jeżeli tworzysz sesję dla wielu zdalnych maszyn) i jeżeli to konieczne, podaj alternatywną nazwę konta użytkownika, numer portu, mechanizm uwierzytelniania i tak dalej. Rezultatem wykonania takiego polecenia będzie obiekt sesji, który będzie przechowywany w pamięci powłoki PowerShell:

```
PS C:\> new-pssession -computername server-r2,server17,dc5
```

Aby pobrać utworzone sesje, powinieneś użyć polecenia `Get-PSSession`:

```
PS C:\> get-pssession
```

Chociaż to działa, naszym preferowanym rozwiązaniem jest utworzenie sesji i natychmiastowe przypisanie jej do odpowiedniej zmiennej. Na przykład Don posiada trzy serwery IIS, które bardzo często rekonfiguruje za pomocą polecenia `Invoke-Command`. Aby ułatwić sobie ten proces, przechowuje utworzone sesje w arbitralnie wybranej zmiennej, tak jak to zostało pokazane poniżej:

```
PS C:\> $iis_servers = new-pssession -comp web1,web2,web3 -credential WebAdmin
```

Nigdy nie zapominaj, że każda sesja pochłania pewien odsetek zasobów Twojego komputera. Jeżeli zamkniesz powłokę, utworzone w niej sesje również zostaną automatycznie zamknięte, ale jeżeli przez dłuższy czas z nich nie korzystasz, dobrze jest ręcznie je zamknąć, zwłaszcza jeżeli zamierzasz nadal używać powłoki do innych zadań.

Aby zamknąć sesję, użyj polecenia `Remove-PSSession`. Przykładowo, aby zamknąć utworzone w poprzednim poleceniu sesje serwerów IIS, powinieneś użyć następującego polecenia:

```
PS C:\> $iis_servers | remove-pssession
```

Jeżeli zamiast tego chcesz zamknąć wszystkie otwarte w danej chwili sesje, możesz zastosować takie oto polecenie:

```
PS C:\> get-pssession | remove-pssession
```

Jak widać, jest to dość łatwe.

Ale kiedy już uruchomisz kilka sesji ze zdalnymi maszynami, to co możesz z nimi zrobić i jak z nich korzystać? Na potrzeby kilku następnych podrozdziałów przyjmujemy założenie, że utworzyłeś zmienną o nazwie `$sessions`, która zawiera co najmniej dwie zdalne sesje. W naszym przypadku użyjemy komputerów `localhost` i `SERVER-R2` (tutaj powinniśśes podać odpowiednie nazwy swoich komputerów). Używanie adresu `localhost` nie jest oszukiwaniem: powłoka PowerShell uruchamia prawdziwą sesję zdalną z kolejną instancją samej siebie. Pamiętaj, że będzie to działać tylko wtedy, gdy na wszystkich komputerach docelowych będziesz miał włączoną opcję komunikacji zdalnej. Jeżeli nie pamiętasz, jak to zrobić, sięgnij po odpowiednie informacje do rozdziału 13.

ZRÓB TO SAM Jak zwykle powinniśśes samodzielnie próbować uruchamiać polecenia opisywane w naszych przykładach. Upewnij się, że używasz prawidłowych nazw komputerów. Jeżeli masz do dyspozycji tylko jeden komputer, użyj jego nazwy oraz adresu `localhost`.

Dla zainteresowanych

Istnieje fajne rozwiązanie, które pozwala na tworzenie wielu sesji za pomocą jednego polecenia i jednocześnie przypisanie każdej sesji do unikatowej zmiennej (zamiast pakowania wszystkich sesji do jednej zmiennej, tak jak to robiliśmy poprzednio):

```
$s server1,$s server2 = new-pssession -computer server-r2,dc01
```

Dzięki takiej składni polecenia sesję dla komputera `SERVER-R2` umieszczamy w zmiennej `$s server1`, a sesję dla komputera `DC01` — w zmiennej `$s server2`, co może znakomicie ułatwić później niezależne korzystanie z tych sesji.

Ale jak zawsze powinniśśes zachować pewną ostrożność — spotykaliśmy się z przypadkami, w których poszczególne sesje nie były tworzone dokładnie w określonej kolejności; w takiej sytuacji zmienna `$s server1` mogłaby zawierać sesję z komputerem `DC01` zamiast `SERVER-R2`. Aby się upewnić, jak zostały przypisane kolejne sesje, możesz wyświetlić zawartości odpowiednich zmiennych.

Nasze sesje używane w kolejnych przykładach zostały utworzone w następujący sposób:

```
PS C:\> $sessions = New-PSSession -comp SERVER-R2,localhost
```

Pamiętaj, że mamy już włączoną komunikację zdalną na tych komputerach i że znajdują się one w tej samej domenie. Jeżeli chcesz sobie nieco odświeżyć wiadomości na temat komunikacji zdalnej z użyciem powłoki PowerShell, powinniśśes zajrzeć do rozdziału 13.

20.3. Używanie sesji z poleceniem *Enter-PSSession*

Jak zapewne pamiętasz z rozdziału 13., do interaktywnej pracy z powłoką zdalną na jednym komputerze używaliśmy polecenia `Enter-PSSession`. Zamiast jednak podawać nazwę komputera zdalnego, można określić pojedynczy obiekt sesji. Ponieważ nasza

zmienna `$sessions` zawiera dwa obiekty sesji, musimy wybrać jeden z nich za pomocą indeksu (o indeksach wspominaliśmy w rozdziale 18.):

```
PS C:\> enter-ssession -session $sessions [0]
[server-r2]: PS C:\Users\Administrator\Documents>
```

Po wykonaniu tego polecenia zmiana wyglądu znaku zachęty wskazuje, że obecnie kontrolujemy komputer zdalny. Polecenie `Exit-PSSession` powoduje powrót do lokalnego monitu, ale sama sesja zdalna pozostaje otwarta do dalszych działań:

```
[server-r2]: PS C:\Users\Administrator\Documents> exit-ssession
PS C:\>
```

Pewne trudności może powodować konieczność zapamiętania numerów indeksów przypisanych do poszczególnych komputerów zdalnych. W takim przypadku możesz skorzystać z właściwości obiektu sesji. Na przykład kiedy prześlemy zmienną `$sessions` za pomocą potoku do polecenia `Gm`, otrzymamy następujące rezultaty:

```
PS C:\> $sessions | gm
      TypeName: System.Management.Automation.Runspaces.PSSession

Name      MemberType      Definition
-----
Equals    Method           bool Equals(System.Object obj)
GetHashCode Method         int GetHashCode()
GetType   Method           type GetType()
ToString  Method           string ToString()
ApplicationPrivateData Property        System.Management.Automation.PSPr...
Availability Property        System.Management.Automation.Runs...
ComputerName Property        System.String ComputerName {get;}
ConfigurationName Property        System.String ConfigurationName {...
Id         Property        System.Int32 Id {get;}
InstanceId Property        System.Guid InstanceId {get;}
Name       Property        System.String Name {get;set;}
Runspace   Property        System.Management.Automation.Runs...
State      ScriptProperty  System.Object State {get=$this.Ru...
```

W wynikach działania możemy zobaczyć, że obiekt sesji posiada właściwość `ComputerName`, co oznacza, że możemy dla wybranej sesji zastosować odpowiedni filtr:

```
PS C:\> enter-ssession -session ($sessions | where { $_.computername -eq 'server-r2' })
[server-r2]: PS C:\Users\Administrator\Documents>
```

Nie da się jednak ukryć, że składnia takiego polecenia jest nieco dziwaczna. Jeżeli chcesz użyć tylko jednej, wybranej sesji spośród sesji zapisanych w zmiennej i nie możesz sobie przypomnieć jej numeru indeksu, łatwiej będzie w ogóle zrezygnować z używania zmiennej.

Pamiętaj, że nawet jeżeli obiekty sesji zostały zapisane w wybranej zmiennej, to nadal są one przechowywane na głównej liście otwartych sesji powłoki PowerShell. Możesz uzyskać do nich dostęp za pomocą polecenia `Get-PSSession`:

```
PS C:\> enter-ssession -session (get-ssession -computer server-r2)
```


Polecenie `Get-PSSession` pobiera sesję powiązaną z komputerem zdalnym o nazwie `SERVER-R2` i przekazuje ją do parametru `-session` polecenia `Enter-PSSession`.

Kiedy po raz pierwszy odkryliśmy tę technikę, byliśmy pod jej wielkim wrażeniem, co skłoniło nas do bardziej dogłębnego zapoznania się z jej szczegółami. Przede wszystkim uważnie zapoznaliśmy się z treścią pliku pomocy polecenia `Enter-PSSession`, a zwłaszcza jego parametru `-session`. Oto co tam znaleźliśmy:

-Session <PSSession>

Specifies a Windows PowerShell session (PSSession) to use for the interactive session.

↳ This parameter takes a session object. You can also use the Name, InstanceID, or ID parameters to specify a PSSession.

Enter a variable that contains a session object or a command that creates or gets a session object, such as a `New-PSSession` or `Get-PSSession` command. You can also pipe

↳ a session object to `Enter-PSSession`. You can submit only one PSSession with this

↳ parameter. If you enter a variable that contains more than one PSSession, the command fails.

When you use `Exit-PSSession` or the `EXIT` keyword, the interactive session ends, but the

↳ PSSession that you created remains open and available for use.

Required? false

Position? 1

Default value

Accept pipeline input? true (ByValue, ByPropertyName)

Accept wildcard characters? True

Jeżeli przypomnisz sobie, o czym mówiliśmy w rozdziale 9., to z pewnością zauważysz, że bardzo obiecująco wygląda informacja o tym, że `-session` może pobierać obiekty `PSSession` bezpośrednio z potoku (właściwość `Accept pipeline input?` ma wartość `true`). Wiemy, że polecenie `Get-PSSession` tworzy obiekty typu `PSSession`, więc polecenie o składni przedstawionej poniżej również powinno działać poprawnie:

```
PS C:\> Get-PSSession -ComputerName SERVER-R2 | Enter-PSSession
[server-r2]: PS C:\Users\Administrator\Documents>
```

Jak widać, takie polecenie działa. Naszym skromnym zdaniem jest to znacznie bardziej elegancki sposób na pozyskanie pojedynczej sesji, nawet jeżeli wszystkie sesje zostały zapisane w zmiennej.

WSKAZÓWKA Przechowywanie sesji w zmiennej może być bardzo wygodne. Powinieneś jednak pamiętać, że powłoka PowerShell i tak przechowuje listę wszystkich otwartych sesji; zapisywanie ich w zmiennej jest przydatne tylko wtedy, gdy chcesz odnieść się do kilku sesji naraz, tak jak to będziemy robić w kolejnym podrozdziale.

20.4. Używanie sesji z poleceniem *Invoke-Command*

Sesje pokazują swoją przydatność zwłaszcza podczas korzystania z polecenia `Invoke-Command`, które stosowaliśmy do wykonywania wybranego polecenia (lub nawet całych skryptów) na wielu komputerach zdalnych równolegle. Dzięki zapisaniu naszych sesji w zmiennej `$sessions` możemy z łatwością uruchomić wybraną komendę lub skrypt na wielu maszynach za pomocą następującego polecenia:

```
PS C:\> invoke-command -command { get-wmiobject -class win32_process } -session $sessions
```

Jak zauważyłeś, w tym przykładzie do komputerów zdalnych wysyłamy polecenie `Get-WmiObject`. W zasadzie mogliśmy tutaj użyć własnego parametru `-computername` polecenia `Get-WmiObject`, ale nie zrobiliśmy tego z czterech powodów:

- Komunikacja zdalna powłoki PowerShell działa na jednym, predefiniowanym porcie; zapytania WMI już nie. Z tego względu komunikacja zdalna jest łatwiejszym rozwiązaniem w środowisku z zaporami sieciowymi, ponieważ łatwiej jest utworzyć dla niej odpowiednie reguły. Wbudowana zaporę sieciową systemu Windows posiada co prawda odpowiedni wyjątek dla usługi WMI, który obejmuje inspekcję stanu niezbędną do poprawnego działania mechanizmu wybierającego losowy port dla usługi WMI (ang. *endpoint mapping*; **mapowanie punktu końcowego**), ale właściwa konfiguracja niektórych zapór sieciowych innych firm może być trudna. Z drugiej strony, jeżeli używamy mechanizmu komunikacji zdalnej powłoki PowerShell, w zaporze sieciowej musimy utworzyć wyjątek tylko dla jednego, z góry znanego portu.
- Pobieranie informacji o wszystkich procesach ze wszystkich komputerów zdalnych może wymagać dużego nakładu pracy i zasobów systemowych. Dzięki zastosowaniu polecenia `Invoke-Command` zmuszamy każdy z komputerów zdalnych do samodzielnego wykonania własnej części zadania i odesłania do naszego komputera gotowych wyników działania.
- Mechanizm komunikacji zdalnej działa równolegle, domyślnie kontaktując się z maksymalnie 32 komputerami jednocześnie. Usługa WMI działa sekwencyjnie, łącząc się kolejno tylko z jednym komputerem w tym samym czasie.
- Polecenie `Get-WmiObject` nie potrafi korzystać z naszych predefiniowanych sesji, ale możemy ich używać, pracując z poleceniem `Invoke-Command`.

UWAGA Nowe polecenia z rodziny CIM (takie jak `Get-CimInstance`), dostępne w powłoce PowerShell v3, nie posiadają parametru `-computerName` (jaki ma polecenie `Get-WmiObject`). Te nowe polecenia zostały zaprojektowane do pracy z komputerami zdalnymi z wykorzystaniem polecenia `Invoke-Command`.

Parametr `-session` polecenia `Invoke-Command` może być również „nakarmiony” sekwencją poleceń umieszczonych w nawiasach, podobnie jak to robiliśmy w poprzednich rozdziałach z nazwami komputerów. Na przykład komenda przedstawiona poniżej wysłała polecenie `Get-WmiObject` do każdej sesji połączonej z komputerem, którego nazwa znajduje się na liście:

```
PS C:\> invoke-command -command { get-wmiobject -class win32_process }  
-session (get-pssession -comp server1,server2,server3)
```

W zasadzie mógłbyś oczekiwać, że polecenie `Invoke-Command` będzie mogło odbierać obiekty sesji z potoku, tak jak to może zrobić polecenie `Enter-PSSession`. Ale szybki rzut oka na zawartość pliku pomocy polecenia `Invoke-Command` pokazuje, że w tym przypadku nie możemy skorzystać z tej sztuczki. Szkoda, ale na szczęście przedstawiony wyżej przykład użycia wyrażenia w nawiasie zapewnia tę samą funkcjonalność bez zbędnego komplikowania składni polecenia.

20.5. Niejawna komunikacja zdalna — importowanie sesji

Niejawna komunikacja zdalna jest dla nas jedną z najciekawszych i najbardziej użytecznych funkcji, jakie kiedykolwiek miała konsola wiersza poleceń dowolnego systemu operacyjnego. Niestety z jakiegoś zupełnie dla nas niezrozumiałego powodu mechanizm ten praktycznie wcale nie jest opisany w dokumentacji powłoki PowerShell. Oczywiście nie da się ukryć, że poszczególne niezbędne polecenia są dobrze udokumentowane, ale praktycznie nie zostało nigdzie wspomniane, w jaki sposób można je ze sobą połączyć w celu skorzystania z tych niesamowitych możliwości. Na szczęście odpowiednie informacje znajdziesz właśnie w tym podrozdziale.

Przyjrzyjmy się zatem następującemu scenariuszowi — wiesz już, że firma Microsoft udostępnia coraz więcej modułów i przystawek obejmujących zarówno system Windows, jak i różne inne produkty, ale czasami z tego czy innego powodu nie możesz ich zainstalować na komputerze lokalnym. Doskonałym przykładem może być tutaj moduł ActiveDirectory, który po raz pierwszy został dostarczony wraz z systemem Windows Server 2008 R2 — jest on dostępny tylko dla systemu Windows Server 2008 R2 oraz na komputerach z systemem Windows 7, na których zainstalowany jest pakiet narzędzi Remote Server Administration Tools (RSAT). Co powinieneś jednak zrobić, jeżeli używasz komputera z systemem Windows XP lub Windows Vista? Pech? Nie. W takiej sytuacji możesz użyć mechanizmu niejawnej komunikacji zdalnej.

Spójrzmy na cały proces w jednym przykładzie:

```

PS C:\> $session = new-ssession -comp server-r2  ← ❶ Nawiązuje połączenie
PS C:\> invoke-command -command
{ import-module activedirectory } ← ❷ Ładuje moduł zdalny
-session $session
PS C:\> import-ssession -session $session
-module activedirectory ← ❸ Importuje polecenia zdalne
-prefix rem

```

ModuleType	Name	ExportedCommands
Script	tmp_2b9451dc-b973-495d...	{Set-ADOrganizationalUnit, Get-ADD...

❹ Sprawdza tymczasowy moduł lokalny

A oto co dzieje się w tym przykładzie:

1. Rozpoczynamy od ustanowienia sesji z komputerem zdalnym, na którym zainstalowany jest moduł ActiveDirectory ❶. Na komputerze tym musi być zainstalowana powłoka PowerShell v2 lub nowsza (co jest oczywiście spełnione w przypadku systemów Windows Server 2008 R2 i nowszych) i musi być włączony mechanizm komunikacji zdalnej.
2. Nakazujemy komputerowi zdalnemu, aby zaimportował swój lokalny moduł ActiveDirectory ❷. To tylko przykład; w praktyce moglibyśmy oczywiście załadować dowolny moduł, a nawet dodać przystawkę PSSnapin, jeżeli to właśnie byłoby nam potrzebne. Ponieważ sesja jest nadal otwarta, moduł pozostaje załadowany na komputerze zdalnym.
3. Następnie nakazujemy naszemu komputerowi, aby zaimportował polecenia z tej zdalnej sesji ❸. Interesują nas tylko polecenia z modułu ActiveDirectory i chcemy,

aby po załadowaniu do każdego rzeczownika w nazwie polecenia dodany został prefiks `rem`, co pozwoli nam na znacznie łatwiejsze rozpoznawanie poleceń zdalnych. Oznacza to również, że nowe polecenia nie będą kolidować z poleceniami o tej samej nazwie, istniejącymi już wcześniej w naszej powłoce.

4. Powłoka PowerShell tworzy na naszym komputerze lokalnym tymczasowy moduł, który będzie zawierał zaimportowane polecenia zdalne ❹. Same polecenia nie są kopiowane; zamiast tego powłoka PowerShell tworzy dla nich skróty, które wskazują na komputer zdalny.

Od tej chwili możemy uruchamiać polecenia modułu ActiveDirectory, a nawet wyświetlać zawartość ich plików pomocy. Zamiast uruchamiać polecenia `New-ADUser`, uruchamiamy polecenie `New-remADUser`, ponieważ do rzeczowników w nazwach poleceń zdalnych dodaliśmy prefiks `rem`. Polecenia pozostają dostępne, dopóki nie zamkniemy powłoki lub nie zamkniemy danej sesji z komputerem zdalnym. Aby odzyskać dostęp do poleceń zdalnych po uruchomieniu nowej powłoki, musimy powtórzyć cały proces.

Kiedy uruchamiamy zaimportowane polecenia, tak naprawdę nie działają one na naszym komputerze lokalnym. Zamiast tego są niejawnie powiązane z komputerem zdalnym, który je dla nas wykonuje i posłusznie wysyła wyniki działania do naszego komputera.

Możemy wyobrazić sobie świat, w którym nie będziemy musieli już więcej instalować narzędzi administracyjnych na naszych komputerach — to niepotrzebny kłopot, którego moglibyśmy uniknąć. Dziś potrzebujesz narzędzi, które można uruchomić w systemie operacyjnym komputera, aby komunikować się z dowolnym serwerem zdalnym, którym musisz zarządzać — a zgromadzenie i odpowiednie skonfigurowanie wszystkich niezbędnych narzędzi może zająć trochę czasu. W przyszłości być może nie będziesz musiał tego robić. Zamiast tego użyjesz niejawnej komunikacji zdalnej do pracy z serwerami, które same będą oferowały funkcje zarządzania jako kolejną usługę dostępną za pośrednictwem powłoki Windows PowerShell.

A teraz zła wiadomość: wyniki przekazywane do Twojego komputera lokalnego przez niejawne usługi dostępu zdalnego są deserializowane, co oznacza, że przed przesłaniem właściwości obiektów są konwertowane do postaci plików XML. Obiekty otrzymywane w ten sposób nie posiadają żadnych metod. W większości przypadków nie stanowi to problemu, ale niektóre moduły i przystawki generują obiekty, które mają być używane w sposób bardziej programowy, więc obiekty te nie nadają się do przekazywania z użyciem niejawnej komunikacji zdalnej. Mamy jednak nadzieję, że w praktyce nie będziesz się często spotykał z obiektami posiadającymi takie ograniczenia, ponieważ poleganie na metodach obiektów narusza niektóre praktyki i założenia projektowe powłoki PowerShell. Jeżeli jednak napotkasz takie obiekty, nie będziesz mógł ich używać za pośrednictwem niejawnej komunikacji zdalnej.

20.6. Korzystanie z rozłączonych sesji

W powłoce PowerShell v3 do mechanizmów komunikacji zdalnej zostały wprowadzone dwa istotne ulepszenia.

Po pierwsze, sesje są znacznie mniej wrażliwe, co oznacza, że mogą nawet przetrwać krótkie przerwy w funkcjonowaniu połączeń sieciowych i inne przejściowe zakłócenia. Jest to bardzo korzystne dla użytkownika, nawet jeżeli nie korzystasz z sesji w sposób jawny. Przykładowo, jeżeli użyłeś polecenia `Enter-PSSession` i jego parametru `-ComputerName`, „pod maską” tego polecenia technicznie nadal używasz sesji, co przekłada się na bardziej niezawodne połączenie.

Inną nową cechą wprowadzoną w wersji 3 powłoki jest możliwość użycia rozłączonych sesji. Załóżmy, że pracujesz na maszynie o nazwie `COMPUTER1`, zalogowałeś się jako użytkownik `Admin1`, który jest członkiem grupy administratorów domeny (`Domain Admins`) i tworzysz nowe połączenie z maszyną o nazwie `COMPUTER2`:

```
PS C:\> New-PSSession -ComputerName COMPUTER2
```

Id	Name	ComputerName	State
4	Session4	COMPUTER2	Opened

Teraz możesz już odłączyć tę sesję. Robisz to z poziomu komputera `COMPUTER1`, na którym pracujesz. Wykonanie takiej operacji rozłącza połączenie między dwoma komputerami, ale pozostawia jednak kopię powłoki PowerShell działającą na maszynie `COMPUTER2`. Aby to zrobić, powinieneś podczas rozłączania podać numer identyfikacyjny sesji, który był wyświetlany w czasie jej tworzenia:

```
PS C:\> Disconnect-PSSession -Id 4
```

Id	Name	ComputerName	State
4	Session4	COMPUTER2	Disconnected

Jest to coś, o czym oczywiście musisz pamiętać — rozłączając sesję, pozostawiasz kopię powłoki PowerShell działającą na maszynie `COMPUTER2`. W takiej sytuacji bardzo ważne staje się zdefiniowanie dodatkowych właściwości sesji, takich jak okres bezczynności. We wcześniejszych wersjach powłoki PowerShell sesja, którą rozłączałeś, była bezpowrotnie zamykana, więc nie było potrzeby żadnego dodatkowego czyszczenia jej pozostałości. Począwszy od powłoki w wersji 3 możliwe stało się zaśmiecanie środowiska poprzez pozostawianie uruchomionych sesji, co oznacza, że ponosisz większą odpowiedzialność.

Ale takie nowe możliwości mają też swoje niewątpliwe zalety: możemy zalogować się na kolejny komputer, na przykład `COMPUTER3`, jako ten sam administrator domeny o nazwie `Admin1` i pobierać listę sesji uruchomionych na komputerze `COMPUTER2`:

```
PS C:\> Get-PSSession -computerName COMPUTER2
```

Id	Name	ComputerName	State
4	Session4	COMPUTER2	Disconnected

Fajne, prawda? Jeżeli zalogujesz się jako inny użytkownik, nawet jako inny administrator, nie będziesz mógł zobaczyć tych sesji, zamiast tego wyświetlane będą tylko sesje utworzone na `COMPUTER2`. Logując się jednak jako `Admin1`, widzisz listę swoich otwartych wcześniej sesji i możesz ponownie się do nich podłączyć:

```
PS C:\> Get-PSSession -computerName COMPUTER2 | Connect-PSSession
```

Id	Name	ComputerName	State
4	Session4	COMPUTER2	Open

Teraz musimy poświęcić chwilę czasu na rozmowę o zarządzaniu sesjami. W napędzie WSMAN: powłoki PowerShell znajdziesz ustawienia, które pomogą Ci kontrolować rozłączone sesje. Większość z tych ustawień możesz konfigurować globalnie za pomocą zasad grupy (GPO — ang. *Group Policy Object*). Poniżej zamieszczamy krótkie zestawienie najważniejszych ustawień:

- Ścieżka `WSMAN:\localhost\Shell`:
 - `IdleTimeout` — określa czas, w którym sesja może być bezczynna, zanim zostanie automatycznie wyłączona. Wartość domyślna to około 2000 godzin (wyrażonych w sekundach) lub inaczej mówiąc, około 84 dni.
 - `MaxConcurrentUsers` — określa maksymalną liczbę użytkowników, którzy mogą mieć jednocześnie otwarte sesje.
 - `MaxShellRunTime` — określa maksymalny czas otwarcia aktywnej sesji. Domyślnie dla wszystkich praktycznych zastosowań jest to nieskończoność. Powinieneś jednak pamiętać, że w przypadku bezczynności powłoki czas życia sesji może być determinowany ustawieniami parametru `IdleTimeout`.
 - `MaxShellsPerUser` — określa maksymalną liczbę sesji, które pojedynczy użytkownik może otworzyć jednocześnie. Aby obliczyć maksymalną możliwą liczbę sesji dla wszystkich użytkowników na danym komputerze, należy pomnożyć tę wartość przez `MaxConcurrentUsers`.
- Ścieżka `WSMAN:\localhost\Service`:
 - `MaxConnections` — ustawia górny limit połączeń przychodzących do mechanizmu dostępu zdalnego. Nawet jeżeli zezwolisz na większą liczbę sesji na użytkownika lub zwiększenie maksymalnej liczby użytkowników z otwartymi sesjami, parametr `MaxConnections` determinuje bezwzględną, maksymalną liczbę jednoczesnych połączeń przychodzących.

Jako administrator systemu ponosisz oczywiście znacznie większą odpowiedzialność niż zwykły, szary użytkownik. Do Twoich zadań należy śledzenie swoich sesji, szczególnie jeżeli będziesz je odłączać i ponownie się z nimi łączyć. Rozsądne ustawienia limitów czasowych sesji mogą zapewnić, że poszczególne sesje powłoki nie będą niepotrzebnie pozostawać w stanie bezczynności przez dłuższy czas.

20.7. Ćwiczenia

UWAGA Do wykonania ćwiczeń w tym zestawie potrzebny Ci będzie komputer pracujący pod kontrolą systemu Windows Server 2008 R2 lub nowszego z zainstalowaną powłoką PowerShell v3 lub nowszą. Jeśli masz dostęp tylko do komputera z systemem Windows 7 lub nowszym, nie będziesz w stanie wykonać ćwiczeń od 6. do 9. z tego zestawu.

Aby wykonać przedstawiony niżej zestaw zadań, powinieneś mieć dwa komputery — jeden, z którego będziesz nawiązywał połączenia, a drugi, z którym będziesz się zdalnie łączył. Jeżeli masz do dyspozycji tylko jeden komputer, użyj jego nazwy, aby się z nim „zdalnie” połączyć. W ten sposób przynajmniej część zadań powinno się dać wykonać w podobny sposób.

WSKAZÓWKA W rozdziale 1. wspominaliśmy o środowisku wirtualnym CloudShare (<https://www.cloudshare.com>), w którym za niewielką opłatą możesz uruchamiać w chmurze kilka maszyn wirtualnych. W sieci Internet znajdziesz również wiele innych, podobnych usług opartych na wirtualnych środowiskach działających w chmurze. Korzystając z CloudShare, nie musieliśmy konfigurować systemu operacyjnego Windows, ponieważ usługa miała gotowe do użycia szablony maszyn wirtualnych. Niestety cała usługa jest płatna i nie jest dostępna we wszystkich krajach, ale jeżeli możesz z niej korzystać, to jest to świetny sposób na szybkie utworzenie i uruchomienie środowiska testowego (jeżeli nie możesz takiego środowiska uruchomić lokalnie).

1. Zamknij wszystkie otwarte sesje w Twojej powłoce.
2. Utwórz nową sesję z komputerem zdalnym. Zapisz sesję w zmiennej o nazwie `$session`.
3. Użyj zmiennej `$session` do nawiązania sesji powłoki typu jeden-do-jednego z komputerem zdalnym. Wyświetl listę procesów działających na komputerze zdalnym, a następnie zakończ sesję powłoki.
4. Użyj zmiennej `$session` oraz polecenia `Invoke-Command` do uzyskania listy usług działających na komputerze zdalnym.
5. Użyj poleceń `Get-PSSession` oraz `Invoke-Command`, aby uzyskać listę 20 ostatnich rekordów z dziennika zdarzeń Security komputera zdalnego.
6. Użyj polecenia `Invoke-Command` oraz zmiennej `$session`, aby załadować moduł `ServerManager` na komputerze zdalnym.
7. Zaimportuj polecenia modułu `ServerManager` z komputera zdalnego do komputera lokalnego. Dodaj prefiks `rem` do nazw zaimportowanych poleceń.
8. Uruchom zaimportowane polecenie `Get-WindowsFeature`.
9. Zamknij sesję, która znajduje się w zmiennej `$session`.

UWAGA Dzięki nowej funkcji powłoki PowerShell v3 zadania 6. i 7. można również wykonać w jednym kroku za pomocą polecenia `Import-Module`. Zapoznaj się z zawartością pliku pomocy tego polecenia i sprawdź, czy możesz go użyć do zaimportowania modułu z komputera zdalnego.

20.8. Co dalej?

Przeprowadź szybki przegląd swojego środowiska. Jakie produkty obsługują powłokę PowerShell? Exchange Server? SharePoint Server? VMware vSphere? System Center Virtual Machine Manager? Te i inne produkty posiadają swoje moduły dla powłoki PowerShell lub przystawki, a wiele z nich jest dostępnych za pośrednictwem mechanizmu komunikacji zdalnej powłoki PowerShell.

20.9. Odpowiedzi

1. `get-pssession | Remove-PSSession`
2. `$session=new-pssession -computername localhost`
3. `enter-pssession $session`
`Get-Process`
`Exit`
4. `invoke-command -ScriptBlock { get-service } -Session $session`
5. `Invoke-Command -ScriptBlock {get-eventlog -LogName System -Newest 20}`
`-Session (Get-PSSession)`
6. `Invoke-Command -ScriptBlock {Import-Module ServerManager} -Session $session`
7. `Import-PSSession -Session $session -Prefix rem -Module ServerManager`
8. `Get-RemWindowsFeature`
9. `Remove-PSSession -Session $session`

21

Nazywasz to pisanem skryptów?

Do tej pory wszystko, o czym pisaliśmy w tej książce, można było zrobić z poziomu wiersza poleceń konsoli powłoki PowerShell. Nie musiałeś tworzyć żadnego skryptu. To dla nas bardzo ważne, ponieważ często spotykamy administratorów, którzy początkowo unikają tworzenia skryptów, ponieważ słusznie postrzegając to jako rodzaj programowania i podświadomie odczuwając, że nauka nowego języka skryptowego może czasami zająć więcej czasu, niż jest to warte. Mam nadzieję, że samodzielnie przekonałeś się już, ile możesz osiągnąć, pracując z wierszem poleceń powłoki PowerShell bez konieczności zostania programistą.

Ale w tym momencie możesz również zacząć już odczuwać, że powtarzanie w kółko tych samych poleceń staje się dość nudne. Masz najzupełniej rację, więc w tym rozdziale zajmiemy się skryptami powłoki PowerShell — ale obiecujemy, że nadal nie będziemy programować. Zamiast tego skupimy się na tworzeniu skryptów jako na sposobie na oszczędzanie palcom niepotrzebnego „wklepywania” często powtarzających się sekwencji poleceń.

21.1. Nie tworzymy programów, ale raczej pliki wsadowe

Większość administratorów systemu Windows w pewnym momencie utworzyła przynajmniej jeden plik wsadowy wiersza polecenia (który zwykle ma rozszerzenie nazwy `.BAT` lub `.CMD`). Pliki wsadowe to nic innego jak proste pliki tekstowe, które można edytować za pomocą Notatnika, zawierające listy poleceń do wykonania w określonej kolejności. Technicznie takie zestawienie poleceń nazywamy **skryptem**, ponieważ tak jak w filmowym scenariuszu, informuje aktora (czyli w tym przypadku Twój komputer), co

dokładnie należy zrobić i w jakiej kolejności wykonać poszczególne zadania. Jednak pliki wsadowe rzadko wyglądają jak programy, po części dlatego, że powłoka `Cmd.exe` ma dosyć ograniczony język poleceń, który nie pozwala na tworzenie skomplikowanych skryptów.

Skrypty powłoki PowerShell — lub **pliki wsadowe**, jeżeli wolisz — działają podobnie. Zapisujesz w nich listę poleceń, które chcesz uruchomić, a powłoka wykona je w podanej kolejności. Prosty skrypt możesz utworzyć, kopiując polecenia z okna hosta i wklejając je do Notatnika. Oczywiście Notatnik jest, nazwijmy to, dosyć prostym edytorem tekstu. Zakładamy, że będziesz znacznie bardziej zadowolony ze środowiska PowerShell ISE lub z innego edytora wspomagającego tworzenie skryptów, takiego jak dostępny w pakietach PowerGUI, PrimalScript czy PowerShell Plus.

W praktyce używanie środowiska ISE sprawia, że „wykonywanie skryptów” w zasadzie nie różni się niczym od interaktywnego korzystania z powłoki. Korzystając z panelu edytora skryptów środowiska ISE, powinienes wpisać polecenie lub sekwencję poleceń, które mają zostać uruchomione, a następnie kliknąć przycisk *Run script* (uruchom skrypt) znajdujący się na pasku narzędzi. Naciśnij przycisk *Save* (zapisz), a utworzysz skrypt bez konieczności kopiowania i przeklejanania jakichkolwiek poleceń.

21.2. Tworzenie poleceń wielokrotnego użytku

Ideą skryptów powłoki PowerShell jest przede wszystkim ułatwienie uruchamiania wielokrotnie powtarzanych poleceń, bez konieczności ponownego, ręcznego wpisywania ich za każdym razem. Aby się o tym przekonać, musimy wybrać polecenie, które będziemy chcieli często uruchamiać i którego będziemy używać jako przykładu w całym tym rozdziale. Chcielibyśmy, aby było to niezmiernie złożone polecenie, zaczniemy więc od czegoś z WMI i dodamy trochę filtrowania, sortowania i innych rzeczy.

W tym momencie przejdziemy do korzystania ze środowiska PowerShell ISE zamiast z normalnego okna konsoli, ponieważ ISE ułatwi nam migrację naszego polecenia do postaci skryptu. Szczerze mówiąc, środowisko ISE znakomicie ułatwia tworzenie złożonych poleceń, ponieważ zamiast pracować liniowo w jednym wierszu polecenia konsoli, dostajesz do dyspozycji całkiem rozbudowany, pełnoekranowy edytor tekstu.

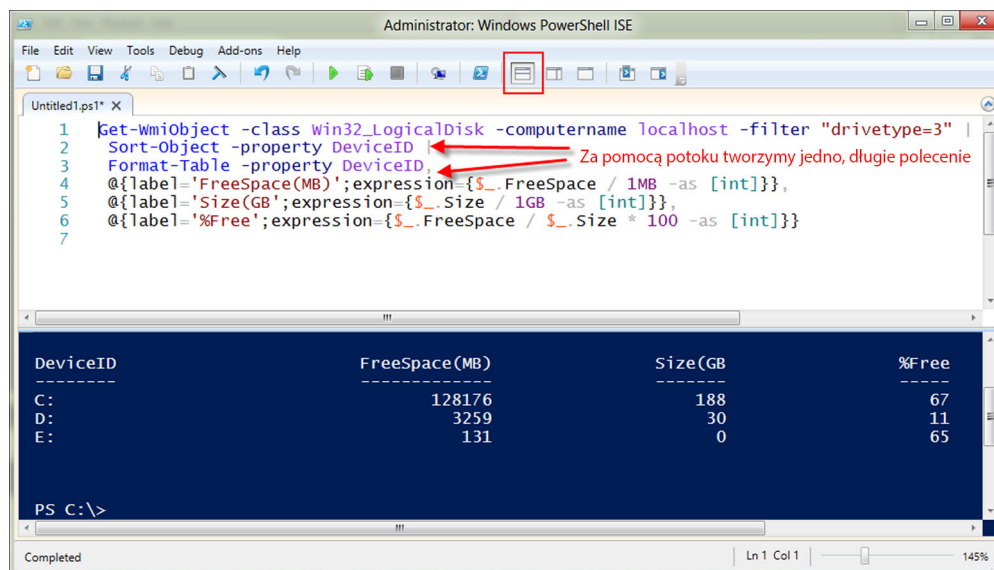
A oto nasze polecenie:

```
Get-WmiObject -class Win32_LogicalDisk -computername localhost -filter "drivetype=3" |
➤ Sort-Object -property DeviceID |
➤ Format-Table -property DeviceID,
➤ @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
➤ @{label='Size(GB)';expression={$_.Size / 1GB -as [int]}},
➤ @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

WSKAZÓWKA Pamiętaj, że zamiast `label` możesz używać `name`, a każda z tych właściwości może być skrócona do pojedynczego znaku, `n` lub `l`. Ale małe litery `L` mogą wyglądać jak cyfry `1`, więc bądź ostrożny!

Rysunek 21.1 pokazuje wpisywanie naszego polecenia w edytorze środowiska ISE. Zwróć uwagę, że wybieramy układ dwóch paneli (możemy to zrobić za pomocą pierwszego

od lewej przycisku na pasku narzędzi w grupie opcji układu). Zauważ również, że formatujemy nasze polecenie tak, aby każdy fizyczny wiersz kończył się znakiem potoku lub przecinkiem. W ten sposób zmuszamy powłokę do rozpoznawania tych wielu wierszy jako pojedynczego polecenia. Możesz zrobić to samo w oknie konsoli, ale takie modelowanie polecenia jest szczególnie skuteczne w edytorze ISE, ponieważ sprawia, że polecenie jest dużo bardziej czytelne. Zauważ również, że używamy pełnych nazw poleceń i nazw parametrów oraz że wpisujemy pełne nazwy parametrów, zamiast używać parametrów pozycyjnych. Wszystko to sprawia, że nasz skrypt będzie łatwiejszy do odczytania i analizowania zarówno teraz, jak i w przyszłości, kiedy sami już zapomnimy, „co autor miał na myśli”.



Rysunek 21.1. Wpisywanie i uruchamianie poleceń w środowisku ISE

Gotowe polecenie uruchamiamy, klikając zieloną ikonę przycisku *Run Script* (uruchom skrypt), znajdującego się na pasku narzędzi środowiska ISE (zamiast tego możesz po prostu nacisnąć klawisz *F5*). Polecenie zostanie uruchomione, a jego wyniki działania pokażą, że działa ono idealnie. Oto świetna sztuczka, której możesz używać w środowisku ISE — w razie potrzeby możesz podświetlić wybraną część polecenia i nacisnąć klawisz *F8*, aby uruchomić tylko tę podświetloną część. Ponieważ nasz „skrypt” sformatowaliśmy w taki sposób, że w każdym fizycznym wierszu znajdowało się jedno polecenie składowe, znakomicie ułatwi nam to testowanie działania naszego polecenia wiersz po wierszu. Pierwszy wiersz możemy zaznaczyć i uruchomić niezależnie od reszty „skryptu”. Jeżeli będziemy zadowoleni z wyników, możemy zaznaczyć pierwszy i drugi wiersz i uruchomić je razem. Jeżeli zadziałają zgodnie z oczekiwaniami, możemy uruchomić całe polecenie.

W tym momencie możemy zapisać całe polecenie i od tej chwili formalnie zacząć nazywać je skryptem. Nasz skrypt zapiszemy pod nazwą *Get-DiskInventory.ps1*. Lubimy nadawać skryptom nazwy w stylu nazw poleceń, składające się z odpowiedniego czasownika i rzeczownika, odpowiadających funkcji realizowanej przez skrypt. Szybko przekonasz się, że nasz skrypt zaczyna wyglądać i działać jak normalne polecenie powłoki, warto więc nadać mu nazwę w tym samym stylu.

ZRÓB TO SAM Zakładamy, że już przeczytałeś rozdział 14. i włączyłeś obsługę skryptów, ustawiając odpowiednią politykę ich wykonywania. Jeżeli jeszcze tego nie zrobiłeś, wróć do rozdziału 17. i wykonaj zamieszczone tam ćwiczenia, tak aby mieć możliwość uruchamiania skryptów w swojej powłoce PowerShell.

21.3. Parametryzowanie poleceń

Jeżeli zastanowisz się nad wielokrotnym wykonywaniem tych samych poleceń, możesz zdać sobie sprawę, że jakiś fragment polecenia od czasu do czasu będzie musiał być zmodyfikowany. Załóżmy na przykład, że chcesz udostępnić skrypt *Get-DiskInventory.ps1* niektórym ze swoich współpracowników, którzy mogą mieć nieco mniejsze doświadczenie w pracy z powłoką PowerShell. Nasze polecenie jest złożone, trudne do samodzielnego napisania i uruchomienia, więc Twój współpracownicy mogą docenić umieszczenie go w znacznie łatwiejszym do uruchomienia skrypcie. Niestety nasz skrypt działa tylko na komputerze lokalnym. Z pewnością można sobie wyobrazić, że niektórzy z Twoich kolegów mogą zamiast tego chcieć uzyskać informację o dyskach z jednego lub większej liczby komputerów zdalnych.

Jednym z rozwiązań jest poproszenie ich o otwarcie skryptu w edytorze tekstu i zmianę wartości parametru `-computername`. Ale jest całkiem możliwe, że takie rozwiązanie nie będzie dla nich komfortowe, nie będą się z tym dobrze czuli, no i jest spore prawdopodobieństwo, że zmienią coś zupełnie innego i po prostu zepsują nasz skrypt. Lepiej zatem byłoby zapewnić jakiś bardziej formalny sposób przekazania nazwy innego komputera (lub zestawu takich nazw). Na tym etapie musisz zidentyfikować elementy, które mogą wymagać zmiany po uruchomieniu skryptu, i zastąpić je odpowiednimi zmiennymi.

Na razie zmienną `$computername` ustawimy na wartość statyczną, żeby móc przetestować działanie polecenia. Oto nasz poprawiony skrypt (zobacz listing 21.1):

Listing 21.1. Get-DiskInventory.ps1 ze sparametryzowanym poleceniem

```
$computername = 'localhost' ← ❶ Ustawiamy nową zmienną
Get-WmiObject -class Win32_LogicalDisk ← ❷ Znak kontynuacji wiersza polecenia
    -computername $computername ` (odwrotny apostrof)
    -filter "drivetype=3" |
Sort-Object -property DeviceID | ❸ Używamy zmiennej
Format-Table -property DeviceID,
    @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}}.
    @{label='Size(GB)';expression={$_.Size / 1GB -as [int]}}.
    @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

Robimy tu trzy rzeczy, z których dwie są funkcjonalne, a jedna czysto kosmetyczna:

- Dodajemy zmienną `$computername` i ustawiamy jej wartość na `localhost` ❶. Zauważyliśmy, że większość poleceń powłoki PowerShell, które pobierają nazwę komputera, używa parametru o nazwie `-computername`, i chcemy powielić tę konwencję, dlatego wybraliśmy tę, a nie inną nazwę zmiennej.
- Zastępujemy wartość parametru `-computername` naszą zmienną ❸. W tym momencie skrypt powinien działać dokładnie tak samo jak wcześniej (uruchom skrypt i upewnij się, że tak jest!), ponieważ zmiennej `$computername` nadajemy wartość `localhost`, wskazującą na komputer lokalny.
- Po nazwie parametru `-computername` i jego wartości umieszczamy znak kontynuacji wiersza polecenia (odwrotny apostrof) ❷, który unieważnia specjalne znaczenie znaku powrotu karetki na końcu wiersza i informuje powłokę, że kolejny wiersz jest dalszą częścią tego samego polecenia. Nie musisz tego robić, kiedy wiersz kończy się znakiem potoku lub przecinkiem, ale aby dopasować listing do wymogów tej książki, musieliśmy przełamać wiersz przed znakiem potoku, a to zadziała tylko wtedy, gdy ostatnim znakiem w wierszu będzie odwrócony apostrof (ang. *backtick*)!

Po wprowadzeniu wszystkich zmian po raz kolejny uruchamiamy skrypt i sprawdzamy, czy nadal działa. Postępujemy tak zawsze po wprowadzeniu jakiegokolwiek zmiany, aby upewnić się, że nie zrobiliśmy przypadkowej literówki lub nie popełniliśmy jakiegoś innego głupiego błędu.

21.4. Tworzenie sparametryzowanego skryptu

Teraz, gdy zidentyfikowaliśmy już elementy skryptu, które od czasu do czasu mogą się zmieniać, musimy zapewnić innej osobie możliwość podawania nowych wartości dla tych elementów. Musimy zatem zamienić naszą zmienną statyczną `$computername` na pełnoprawny parametr wejściowy naszego skryptu.

Na szczęście powłoka PowerShell bardzo ułatwia takie zadanie (zobacz listing 21.2).

Listing 21.2. Get-DiskInventory.ps1 z parametrem wejściowym

```
param (
    $computername = 'localhost' ← ❶ Blok parametrów
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=3" |
Sort-Object -property DeviceID |
Format-Table -property DeviceID,
    @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
    @{label='Size(GB)';expression={$_.Size / 1GB -as [int]}},
    @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

Wszystko, co musimy zrobić, to dodanie bloku `Param()`, w którym umieszczamy deklarację naszej zmiennej ❶, co spowoduje, że zmienna `$computername` zostanie zdefiniowana jako

parametr wejściowy skryptu o wartości domyślnej localhost, która zostanie użyta w sytuacji, kiedy skrypt będzie uruchamiany bez podania nazwy komputera. W praktyce nie musisz podawać wartości domyślnej, ale dobrze to zrobić, gdy jest taka możliwość.

Wszystkie parametry zadeklarowane w taki sposób są zarówno nazwane, jak i pozycyjne, co oznacza, że możemy teraz uruchomić skrypt z wiersza poleceń w dowolny z poniższych sposobów:

```
PS C:\> .\Get-DiskInventory.ps1 server-r2
PS C:\> .\Get-DiskInventory.ps1 -computername server-r2
PS C:\> .\Get-DiskInventory.ps1 -comp server-r2
```

W pierwszym przypadku parametr ten jest używany jako pozycyjny — podajemy jego wartość, a nie nazwę parametru. W drugim i trzecim wystąpieniu określamy nazwę parametru i jego wartość, ale w trzeciej instancji skracamy tę nazwę zgodnie z normalnymi regułami skracania nazw parametrów poleceń powłoki PowerShell. Zauważ, że we wszystkich trzech przypadkach musimy określić ścieżkę do skryptu (.\, czyli bieżący folder), ponieważ powłoka automatycznie nie sprawdza bieżącego katalogu w poszukiwaniu skryptu.

W skrypcie możemy zdefiniować tyle parametrów, ile jest nam potrzebnych, oddzielając je od siebie za pomocą przecinków. Załóżmy na przykład, że chcemy sparаметryzować kryteria filtrowania. Obecnie nasz skrypt wyświetla tylko dyski logiczne typu 3, które reprezentują dyski stałe. Możemy to kryterium zmienić na parametr wejściowy, jak pokazano na listingu 21.3.

Listing 21.3. Get-DiskInventory.ps1 z dodatkowym parametrem wejściowym

```
param (
    $computername = 'localhost',
    $drivetype = 3
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=$drivetype" |
Sort-Object -property DeviceID |
Format-Table -property DeviceID,
    @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
    @{label='Size(GB)';expression={$_.Size / 1GB -as [int]}},
    @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

Definiujemy dodatkowy parametr

Używamy dodatkowego parametru

Zwróć uwagę, że wykorzystujemy zdolność powłoki PowerShell do zastępowania zmiennych ich wartościami w znakach cudzysłowu (poznaleś tę sztuczkę w rozdziale 18.).

Możemy uruchomić ten skrypt w dowolny z trzech podanych wcześniej sposobów, chociaż możemy też pominąć dowolny z parametrów, jeśli chcemy użyć jego wartości domyślnej. Oto kilka przykładów wywołania:

```
PS C:\> .\Get-DiskInventory.ps1 server-r2 3
PS C:\> .\Get-DiskInventory.ps1 -comp server-r2 -drive 3
PS C:\> .\Get-DiskInventory.ps1 server-r2
PS C:\> .\Get-DiskInventory.ps1 -drive 3
```

W pierwszym przypadku określamy oba parametry w kolejności, w jakiej są zadeklarowane w bloku `Param()`. W drugim wywołaniu podajemy skrócone nazwy obu parametrów. Za trzecim razem całkowicie pomijamy parametr `-drivetype`, używając jego domyślnej wartości 3. W ostatnim przypadku pomijamy parametr `-computername`, używając jego domyślnej wartości `localhost`.

21.5. Dokumentowanie skryptu

Tylko prawdziwie złośliwa osoba stworzy przydatny skrypt i nie powie nikomu, jak go używać. Na szczęście powłoka PowerShell ułatwia dodawanie do skryptu komentarzy objaśniających znaczenie poszczególnych elementów. W skryptach możesz umieszczać typowe komentarze spotykane w kodach źródłowych wielu języków programowania, ale jeżeli używasz pełnych nazw poleceń i parametrów, działanie poszczególnych elementów skryptu będzie często oczywiste. Korzystając ze specjalnej składni komentarzy, możesz jednak utworzyć tekst pomocy, która będzie naśladowała własne pliki pomocy powłoki PowerShell.

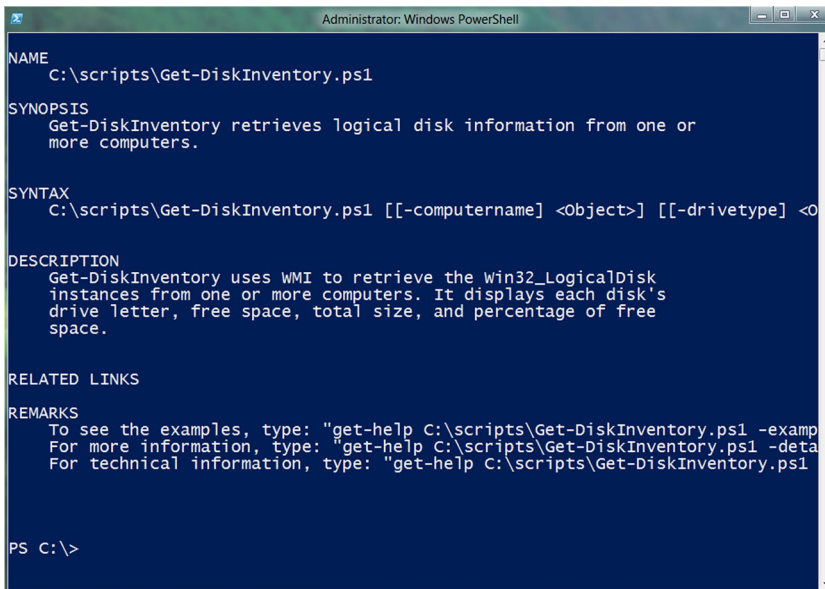
Listing 21.4 pokazuje wygląd naszego skryptu po wprowadzeniu modyfikacji.

Listing 21.4. Dodawanie pomocy do skryptu `Get-DiskInventory.ps1`

```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or more computers.
.DESCRPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk instances from one or more computers. It displays each disk's drive letter, free space, total size, and percentage of free space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.PARAMETER drivetype
The drive type to query. See Win32_LogicalDisk documentation for values. 3 is a fixed disk, and is the default.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
#>
param (
    $computername = 'localhost',
    $drivetype = 3
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=$drivetype" |
Sort-Object -property DeviceID |
Format-Table -property DeviceID,
    @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}}.
    @{label='Size(GB)';expression={$_.Size / 1GB -as [int]}}.
    @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

Zwykle powłoka PowerShell ignoruje wszystko, co znajduje się w wierszu po znaku `#`, z czego wynika, że znak `#` oznacza dalszą zawartość wiersza jako komentarz. Zamiast niego używamy tutaj jednak znaczników bloku komentarza `<#` `#>`, ponieważ nasz komentarz składa się z kilku wierszy i nie chcemy zaczynać każdego z nich od oddzielnego znaku `#`.

Teraz możemy przejść na chwilę do normalnej konsoli tekstowej powłoki PowerShell i spróbować wyświetlić zawartość pomocy dla naszego skryptu. Aby to zrobić, wykonujemy polecenie `help .\Get-DiskInventory` (i znów, jak pamiętasz, musimy podać ścieżkę, ponieważ jest to skrypt, a nie wbudowane polecenie powłoki). Rysunek 21.2 pokazuje wyniki działania tego polecenia, które dowodzą, że powłoka PowerShell potrafi przeanalizować takie komentarze i na ich podstawie utworzyć standardowy ekran pomocy. Możemy nawet uruchomić polecenie `help .\Get-DiskInventory -full`, aby wyświetlić pełną zawartość pomocy, w tym informacje o parametrach i przykład wywołania. Rysunek 21.3 pokazuje wyniki działania takiego polecenia.



```
Administrator: Windows PowerShell

NAME
    C:\scripts\Get-DiskInventory.ps1

SYNOPSIS
    Get-DiskInventory retrieves logical disk information from one or
    more computers.

SYNTAX
    C:\scripts\Get-DiskInventory.ps1 [[-computername] <Object>] [[-drivetype] <O

DESCRIPTION
    Get-DiskInventory uses WMI to retrieve the win32_LogicalDisk
    instances from one or more computers. It displays each disk's
    drive letter, free space, total size, and percentage of free
    space.

RELATED LINKS

REMARKS
    To see the examples, type: "get-help C:\scripts\Get-DiskInventory.ps1 -examp
    For more information, type: "get-help C:\scripts\Get-DiskInventory.ps1 -deta
    For technical information, type: "get-help C:\scripts\Get-DiskInventory.ps1

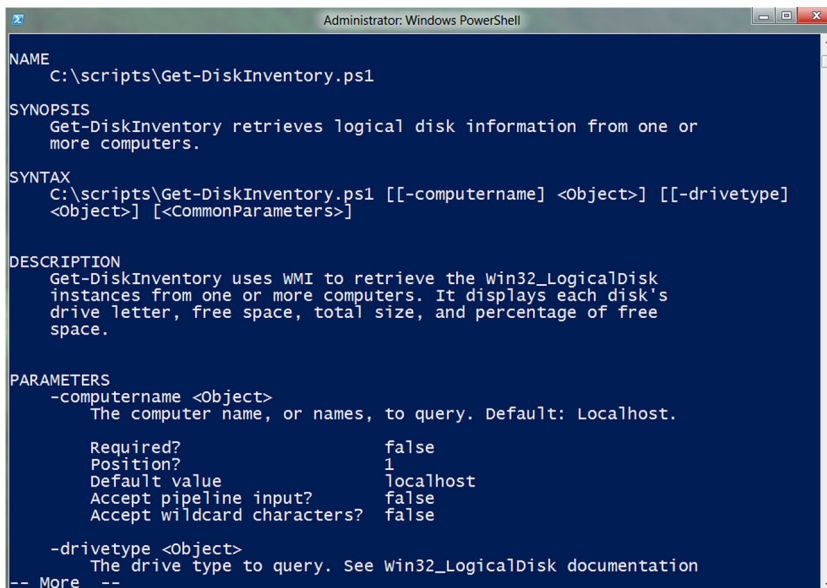
PS C:\>
```

Rysunek 21.2. Wyświetlanie pomocy przy użyciu normalnego polecenia `help`

Takie specjalne komentarze, nazywane systemem **pomocy opartej na komentarzach** (ang. *comment-based help*), muszą pojawić się na początku pliku skryptu. Oprócz znaczników takich jak `.DESCRIPTION`, `.SYNOPSIS` i innych, z których korzystaliśmy, istnieje jeszcze kilka dodatkowych słów kluczowych. Aby uzyskać ich pełną listę, uruchom polecenie `help about_comment_based_help`.

21.6. Jeden skrypt, jeden potok

Zazwyczaj mówimy użytkownikom, że wszystko w skrypcie będzie działało dokładnie tak, jakby poszczególne polecenia zostały ręcznie wpisane do powłoki lub jakby sekwencja poleceń skryptu została skopiowana do schowka i wklejona do powłoki. To jednak nie do końca jest prawda.



```

NAME
    C:\scripts\Get-DiskInventory.ps1

SYNOPSIS
    Get-DiskInventory retrieves logical disk information from one or
    more computers.

SYNTAX
    C:\scripts\Get-DiskInventory.ps1 [[-computername] <Object>] [[-drivetype]
    <Object>] [<CommonParameters>]

DESCRIPTION
    Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
    instances from one or more computers. It displays each disk's
    drive letter, free space, total size, and percentage of free
    space.

PARAMETERS
    -computername <Object>
        The computer name, or names, to query. Default: Localhost.

        Required?                false
        Position?                 1
        Default value             localhost
        Accept pipeline input?    false
        Accept wildcard characters? false

    -drivetype <Object>
        The drive type to query. See Win32_LogicalDisk documentation
    -- More --
  
```

Rysunek 21.3. Opcje polecenia help, takie jak -example, -detailed i -full, są w pełni obsługiwane również w przypadku pomocy opartej na komentarzach

Przyjrzyj się następującemu bardzo prostemu skryptowi:

```

Get-Process
Get-Service
  
```

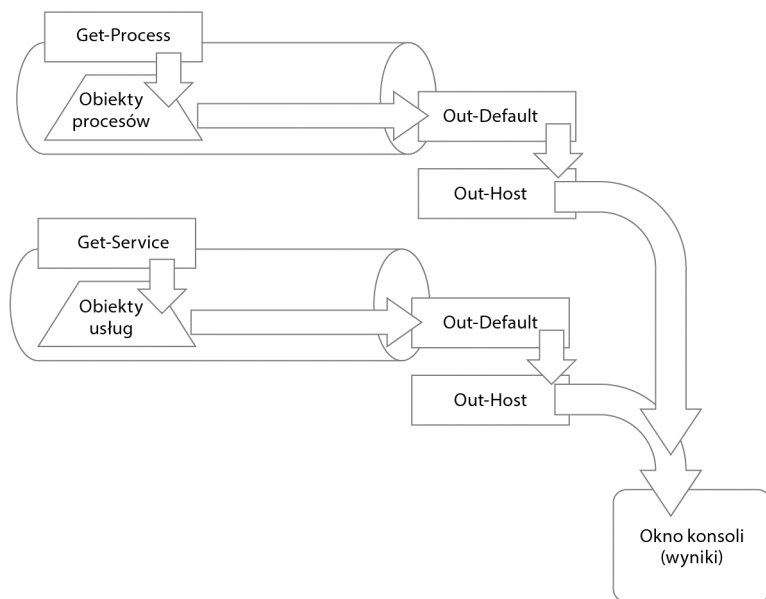
Nasz skrypt składa się tylko z dwóch poleceń. Co się stanie, gdy będziesz w wierszu powłoki wpisywał te polecenia ręcznie, naciskając po każdym z nich klawisz *Enter*?

ZRÓB TO SAM Spróbuj samodzielnie wykonywać opisane polecenia i sprawdź wyniki ich działania (jest to bardzo długa lista, która zupełnie nie będzie pasować do tej książki, a już na pewno nie zmieści się nawet na największym zrzucie ekranu).

Po uruchomieniu komend pojedynczo dla każdego z poleceń tworzony jest nowy potok. Na końcu każdego potoku powłoka PowerShell sprawdza, jak należy sformatować wyniki działania, i tworzy tabele, które niewątpliwie widziałeś na ekranie. Kluczowym elementem jest to, że *każde polecenie działa w oddzielnym potoku*. Rysunek 21.4 ilustruje taką sytuację: dwa całkowicie oddzielne polecenia, dwa osobne potoki, dwa procesy formatowania i dwa różne zestawy wyników.

Być może myślisz, że mamy jakieś „zwarcie”, skoro poświęcamy tyle czasu, aby wyjaśnić coś, co wydaje się najzupełniej oczywiste. Ale to naprawdę bardzo ważne. Oto co się dzieje, gdy uruchamiasz te dwa polecenia osobno:

1. Uruchamiasz polecenie `Get-Process`.
2. Polecenie umieszcza obiekty procesu w potoku.
3. Potok kończy się w cmdletem `Out-Default`, który pobiera te obiekty.



Rysunek 21.4. Dwa polecenia, dwa potoki i dwa zestawy wyników w jednym oknie konsoli

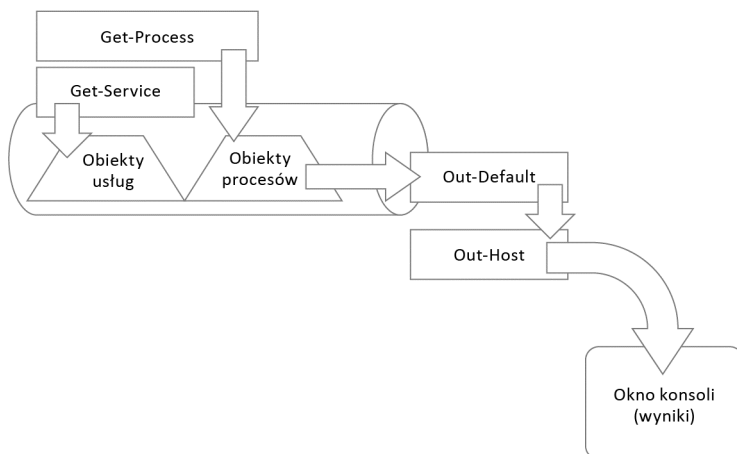
4. Out-Default przekazuje obiekty do cmdletu Out-Host, który wywołuje system formatowania w celu utworzenia odpowiednio sformatowanego tekstu reprezentującego wyniki działania (dowiedziałeś się o tym w rozdziale 10.).
5. Tekst jest wyświetlany na ekranie.
6. Uruchamiasz polecenie Get-Service.
7. Polecenie umieszcza obiekty usługi w potoku.
8. Potok kończy się w cmdletem Out-Default, który pobiera te obiekty.
9. Out-Default przekazuje obiekty do cmdletu Out-Host, który wywołuje system formatowania w celu utworzenia odpowiednio sformatowanego tekstu reprezentującego wyniki działania.
10. Wyniki są wyświetlane na ekranie.

Teraz patrzysz na ekran zawierający wyniki działania dwóch poleceń. Chcielibyśmy, abyś umieścił te dwa polecenia w pliku skryptu. Nazwij go *Test.ps1* lub podobnie. Zanim jednak uruchomisz ten skrypt, skopiuj oba polecenia do schowka systemowego. W edytorze środowiska ISE można to zrobić, podświetlając oba wiersze tekstu jednocześnie i naciskając kombinację klawiszy *Ctrl+C*.

Mając oba polecenia w schowku, przejdź do hosta konsoli powłoki PowerShell i wklej je w wierszu poleceń powłoki. Teoretycznie w obu przypadkach polecenia powinny wykonywać dokładnie to samo, ponieważ znaki powrotu karetki również zostaną przeklejone, czyli ponownie uruchamiasz dwa różne polecenia w dwóch oddzielnych potokach.

Teraz wróć do środowiska ISE i uruchom nasz prosty skrypt. Otrzymałeś inne wyniki, prawda? Dlaczego tak się stało?

W powłoce PowerShell każde polecenie wykonuje się w ramach jednego potoku, a to obejmuje również skrypty. W skrypcie każde polecenie, które generuje na wyjściu dane, będzie zapisywać je do jednego i tego samego potoku, tego, w którym działa sam skrypt. Spójrz na rysunek 21.5. Spróbujemy wyjaśnić, co się tam dzieje:



Rysunek 21.5. Wszystkie polecenia w skrypcie wykorzystują jeden potok danych

1. Skrypt uruchamia polecenie `Get-Process`.
2. Polecenie umieszcza obiekty procesu w potoku.
3. Skrypt uruchamia polecenie `Get-Service`.
4. Polecenie umieszcza obiekty usługi w potoku.
5. Potok kończy się cmdletem `Out-Default`, który zbiera oba rodzaje obiektów.
6. `Out-Default` przekazuje obiekty do cmdletu `Out-Host`, który wywołuje system formatowania w celu utworzenia odpowiednio sformatowanego tekstu reprezentującego wyniki działania.
7. Ponieważ obiekty `Process` były pierwsze, system formatowania powłoki wybiera format odpowiedni dla procesów, dlatego obiekty procesów wyglądają najzupełniej normalnie. Ale wtedy powłoka przechodzi do wyświetlania obiektów usługi i nie może utworzyć zupełnie nowej tabeli, więc dla obiektów usług formatowanie kończy się utworzeniem listy.
8. Wyniki działania pojawiają się na ekranie.

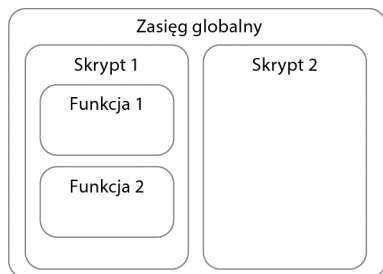
Różnica w wyglądzie wyników działania występuje, ponieważ nasz skrypt zapisuje dwa rodzaje obiektów do pojedynczego potoku. Jest to bardzo ważna różnica między uruchamianiem poleceń ze skryptu a uruchamianiem ich ręcznie: w skrypcie masz do dyspozycji tylko jeden potok. W normalnych warunkach Twoje skrypty powinny wyprowadzać tylko jeden rodzaj obiektów, aby powłoka PowerShell mogła tworzyć rozsądnie wyglądające wyniki działania.

21.7. Szybkie spojrzenie na zasięg

Ostatnim tematem, który musimy omówić, są **zasięgi** (ang. *scope*). Zasięgi są pewną formą kontenera określającego dostępność niektórych typów elementów powłoki PowerShell, głównie aliasów, zmiennych i funkcji.

Sama powłoka ma zasięg najwyższego poziomu, co nazywamy **zasięgiem globalnym** (ang. *global scope*). Po uruchomieniu skryptu tworzony jest dla niego nowy zasięg, nazywany **zasięgiem skryptu** (ang. *script scope*). Zasięg skryptu jest zależny od zasięgu globalnego i mówimy, że jest **elementem podrzędnym** (ang. *child*) zasięgu globalnego, który jest **elementem nadrzędnym** (ang. *parent*) zasięgu skryptu. Funkcje również mają swój **prywatny zasięg** (ang. *private scope*).

Rysunek 21.6 ilustruje relacje zasięgów, z globalnym zasięgiem zawierającym elementy podrzędne, które posiadają swoje elementy podrzędne i tak dalej.



Rysunek 21.6. Zasięg globalny, skrypty i funkcje (prywatne)

Zasięg istnieje tylko tak długo, jak długo jest potrzebny do wykonania tego, co się w nim znajduje. Globalny zasięg istnieje tylko podczas działania powłoki PowerShell, zasięg skryptu istnieje tylko podczas działania tego skryptu i tak dalej. Gdy cokolwiek to jest, przestaje działać, powiązany z tym elementem zasięg znika, zabierając ze sobą wszystko. Powłoka PowerShell ma specyficzne — i czasami mylące — reguły dla elementów o ustalonym zasięgu, takich jak aliasy, zmienne i funkcje, ale główna zasada jest taka: jeżeli próbujesz uzyskać dostęp do elementu o określonym zasięgu, powłoka PowerShell sprawdzi, czy element ten istnieje w bieżącym zasięgu. Jeżeli nie, powłoka PowerShell sprawdzi, czy taki element istnieje w zasięgu elementu nadrzędnego bieżącego zasięgu, i kontynuuje przechodzenie w górę drzewa hierarchii aż do osiągnięcia zasięgu globalnego.

ZRÓB TO SAM Aby uzyskać poprawne wyniki, powinieneś bardzo dokładnie i uważnie wykonać czynności opisane poniżej.

Zobaczmy to w działaniu. Wykonaj następujące kroki:

1. Zamknij wszystkie otwarte okna powłoki PowerShell lub środowiska PowerShell ISE, tak aby można było rozpocząć od zera.
2. Otwórz nowe okno powłoki PowerShell i nowe okno środowiska PowerShell ISE.
3. W ISE utwórz skrypt zawierający jeden wiersz: `Write $x`.

4. Zapisz skrypt jako `C:\Scope.ps1`.
5. W zwykłym oknie powłoki PowerShell uruchom skrypt za pomocą polecenia `C:\Scope`. Nie powinieneś zobaczyć żadnych wyników. Po uruchomieniu skryptu tworzony jest dla niego nowy zasięg. Zmienna `$x` nie istnieje w tym zasięgu, więc powłoka PowerShell przechodzi do zasięgu nadrzędnego — czyli zasięgu globalnego — aby sprawdzić, czy istnieje tam zmienna `$x`. Tam też nie ma takiej zmiennej, więc powłoka PowerShell decyduje, że zmienna `$x` jest pusta, i wyświetla to (czyli nic) jako wynik.
6. W normalnym oknie powłoki PowerShell uruchom polecenie `$x = 4`. Następnie uruchom ponownie skrypt `C:\Scope`. Tym razem jako wynik działania skryptu powinieneś zobaczyć 4. Zmienna `$x` nadal nie jest zdefiniowana w zasięgu skryptu, ale powłoka PowerShell może ją znaleźć w zasięgu globalnym, więc skrypt używa tej wartości.
7. W środowisku ISE wstaw wiersz `$x = 10` na początek skryptu (przed istniejącym poleceniem `Write`) i zapisz skrypt.
8. W normalnym oknie powłoki PowerShell uruchom ponownie skrypt `C:\Scope`. Tym razem jako wynik działania zobaczysz liczbę 10. Dzieje się tak dlatego, że zmienna `$x` jest zdefiniowana w zasięgu skryptu i powłoka nie musi jej szukać w zasięgu globalnym. Teraz wpisz `$x` w wierszu poleceń powłoki. Zobaczysz liczbę 4, co udowadnia, że wartość zmiennej `$x` w zasięgu skryptu nie ma wpływu na wartość `$x` w zasięgu globalnym.

Jedną z ważnych koncepcji jest to, że gdy w danym zasięgu definiujemy zmienną, alias lub funkcję, zasięg ten traci dostęp do zmiennych, aliasów lub funkcji o tej samej nazwie w zasięgu nadrzędnym. Powłoka PowerShell będzie zawsze starała się używać elementów zdefiniowanych lokalnie. Na przykład jeżeli umieścisz polecenie `New-Alias Dir Get-Service` w skrypcie, to wywołanie w takim skrypcie aliasu `Dir` uruchomi polecenie `Get-Service` zamiast zwykłego `Get-ChildItem` (w rzeczywistości powłoka prawdopodobnie nie pozwoli Ci tego zrobić, ponieważ chroni wbudowane aliasy przed ponownym zdefiniowaniem). Przez zdefiniowanie aliasu w zasięgu skryptu uniemożliwisz powłoce przejście do zasięgu nadrzędnego i znalezienie normalnego, domyślnego polecenia `Dir`. Oczywiście redefinicja polecenia `Dir` dla skryptu będzie aktywna tylko w czasie działania tego skryptu, a domyślne polecenie `Dir` zdefiniowane w zasięgu globalnym pozostanie nienaruszone.

Zagadnienia związane z zasięgami mogą łatwo wprowadzać Cię w błąd. Najprościej możesz uniknąć nieporozumień, nie polegając nigdy na elementach znajdujących się w innym zasięgu niż bieżący. Zanim więc spróbujesz uzyskać dostęp do zmiennej w skrypcie, upewnij się, że jej wartość została już przypisana w ramach tego samego zasięgu. Jednym ze sposobów jest użycie parametrów w bloku `Param()`, a ponadto istnieje wiele innych sposobów na przypisanie wartości i obiektów do zmiennej.

21.8. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Poniżej zamieszczamy polecenie, które powinieneś umieścić w skrypcie. Następnie powinieneś zidentyfikować wszelkie sparametryzowane elementy, takie jak nazwa komputera. Twój końcowy skrypt powinien zawierać definicję odpowiedniego parametru oraz pomoc opartą na komentarzach. Uruchom skrypt, aby go przetestować, i użyj polecenia `help`, aby upewnić się, że treść pomocy opartej na komentarzach działa poprawnie. Aby uzyskać więcej informacji, nie zapomnij przeczytać plików pomocy wymienionych w tym rozdziale.

A oto nasze polecenie:

```
Get-WmiObject Win32_LogicalDisk -comp $env:computername -filter "drivetype=3" |
➤ Where { ($_.FreeSpace / $_.Size) -lt .1 } |
➤ Select -Property DeviceID,FreeSpace,Size
```

Podpowiedź: powinieneś sparametryzować co najmniej dwa elementy. Nasze polecenie ma na celu wyświetlenie wszystkich dysków, które mają mniejszą niż podana ilość wolnego miejsca. Oczywiście nie zawsze będziesz chciał sprawdzać tylko komputer lokalny `localhost` (w naszym przykładzie używamy „powershellowego” odpowiednika zmiennej systemowej `%computername%`) i powinieneś mieć również możliwość zmiany wielkości progowej ilości miejsca na dysku (domyślnie 10%). Możesz również sparametryzować typ napędu (tutaj 3), ale na potrzeby tego ćwiczenia pozostaw ten element na stałe z wartością 3.

21.9. Odpowiedzi

```
<#
.Synopsis
Get drives based on percentage free space
.Description
This command will get all local drives that have less than the specified percentage of free space available.
.Parameter Computername
The name of the computer to check. The default is localhost.
.Parameter MinimumPercentFree
The minimum percent free disk space. This is the threshold. The default value is 10. Enter a number between 1 and 100.
.Example
PS C:\> Get-Disk -minimum 20

Find all disks on the local computer with less than 20% free space.
.Example
PS C:\> Get-Disk -comp SERVER02 -minimum 25

Find all local disks on SERVER02 with less than 25% free space.
#>
```

```
Param (  
    $Computername='localhost',  
    $MinimumPercentFree=10  
)
```

Convert minimum percent free

```
$minpercent = $MinimumPercentFree/100
```

```
Get-WmiObject -class Win32_LogicalDisk -computername $computername -filter "drivetype=3" |  
Where { $_.FreeSpace / $_.Size -lt $minpercent } |  
Select -Property DeviceID,FreeSpace,Size
```


22

Ulepszanie sparametryzowanego skryptu

W poprzednim rozdziale utworzyliśmy całkiem fajny skrypt, który następnie został sparametryzowany. Idea takiego skryptu polega na tym, że inny użytkownik może go uruchomić bez konieczności modyfikowania jego zawartości. Użytkownicy skryptów dostarczają dane wejściowe za pośrednictwem odpowiednio przygotowanego interfejsu — czyli parametrów wywołania skryptu — i to wszystko, co mogą w nim „zmienić”. W tym rozdziale zamierzamy pójść nieco dalej.

22.1. Skrypt bazowy

Aby upewnić się, że rozpoczynamy dyskusję w tym samym miejscu, przyjmujemy założenie, że punktem wyjścia do naszych rozważań będzie skrypt przedstawiony na listingu 22.1. Skrypt zawiera sekcję pomocy opartej na komentarzach, dwa parametry wejściowe i polecenie, które je wykorzystuje. W porównaniu z poprzednim rozdziałem wprowadziliśmy do skryptu jedną niewielką zmianę — wynikiem jego działania są teraz wybrane obiekty, a nie sformatowana tabela.

Listing 22.1. Skrypt bazowy: `Get-DiskInventory.ps1`

```
<#  
.SYNOPSIS  
Get-DiskInventory retrieves logical disk information from one or  
more computers.  
.DESCRIPTION  
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk  
instances from one or more computers. It displays each disk's  
drive letter, free space, total size, and percentage of free  
space.
```

.PARAMETER computername

The computer name, or names, to query. Default: Localhost.

.PARAMETER drivetype

The drive type to query. See Win32_LogicalDisk documentation for values. 3 is a fixed disk, and is the default.

.EXAMPLE

Get-DiskInventory -computername SERVER-R2 -drivetype 3

#>

```
param (
    $computername = 'localhost',
    $drivetype = 3
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=$drivetype" |
    Sort-Object -property DeviceID |
    Select-Object -property DeviceID,
        @(name='FreeSpace(MB)':expression={$_.FreeSpace / 1MB -as [int]}),
        @(name='Size(GB)':expression={$_.Size / 1GB -as [int]}),
        @(name='%Free':expression={$_.FreeSpace / $_.Size * 100 -as [int]})
```

Dlaczego zamiast polecenia `Format-Table` użyliśmy `Select-Object`? Po prostu wychodzimy z założenia, że tworzenie skryptu, który generuje wstępnie sformatowane dane wyjściowe, jest złym pomysłem. Przykładowo, użytkownik, który potrzebuje danych wyjściowych w formacie CSV, z pewnością nie będzie zadowolony ze skryptu, który generuje wyniki w postaci ładnie sformatowanych tabel. Dzięki wprowadzeniu takiej zmiany możemy uruchomić nasz skrypt i uzyskać wyniki w formacie odpowiednim do potrzeb. Aby uzyskać wyniki w postaci tabeli, możesz użyć następującego polecenia:

```
PS C:\> .\Get-DiskInventory | Format-Table
```

Jeżeli potrzebne Ci są wyniki w formacie CSV, możesz zastosować takie oto polecenie:

```
PS C:\> .\Get-DiskInventory | Export-CSV disks.csv
```

Cała zaleta takiego rozwiązania polega na tym, że dzięki wyprowadzaniu na wyjście obiektów (za pomocą polecenia `Select-Object`) zamiast sformatowanego tekstu, na dłuższą metę nasz skrypt staje się znacznie bardziej elastyczny.

22.2. Zmuszamy powłokę PowerShell do ciężkiej pracy

Teraz zamierzamy użyć trochę magii powłoki PowerShell, dodając do naszego skryptu jeden wiersz. Ta pozornie błaha zmiana z technicznego punktu widzenia zmienia nasz prosty skrypt w skrypt *zaawansowany*, który pozwala na korzystanie z całego szeregu przydatnych możliwości powłoki PowerShell. Na poniższym listingu zamieszczamy zaktualizowaną wersję skryptu (zobacz listing 22.2).

Listing 22.2. Zamiana `Get-DiskInventory.ps1` na skrypt zaawansowany

```
<#
```

```
.SYNOPSIS
```

```
Get-DiskInventory retrieves logical disk information from one or
```

more computers.

.DESCRIPTION

Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk instances from one or more computers. It displays each disk's drive letter, free space, total size, and percentage of free space.

.PARAMETER computername

The computer name, or names, to query. Default: Localhost.

.PARAMETER drivetype

The drive type to query. See Win32_LogicalDisk documentation for values. 3 is a fixed disk, and is the default.

.EXAMPLE

Get-DiskInventory -computername SERVER-R2 -drivetype 3

```
#>[BP1]
```

```
[CmdletBinding()]
```

```
param (
```

```
    $computername = 'localhost',
```

```
    $drivetype = 3
```

```
)
```

```
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
```

```
-filter "drivetype=$drivetype" |
```

```
Sort-Object -property DeviceID |
```

```
Select-Object -property DeviceID,
```

```
    @{name='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
```

```
    @{name='Size(GB)';expression={$_.Size / 1GB -as [int]}},
```

```
    @{name='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

Ważne jest, aby dyrektywa [CmdletBinding()] była pierwszym wierszem skryptu po sekcji pomocy opartej na komentarzach, ponieważ powłoka PowerShell będzie go szukać tylko w tym miejscu. Po wprowadzeniu tej zmiany skrypt dalej będzie działał normalnie, ale dodając ten wiersz kodu, włączamy kilka fajnych funkcji, które omówimy już za chwilę.

22.3. Definiowanie parametrów obligatoryjnych

Jesteśmy trochę niezadowoleni z naszego skryptu w jego obecnej formie, ponieważ zapewnia on domyślną wartość parametru -ComputerName — a nie jesteśmy pewni, czy jest ona naprawdę potrzebna. Zamiast polegać na ustawieniu domyślnym, lepiej będzie, jeżeli poprosimy użytkownika o podanie tej wartości. Na szczęście powłoka PowerShell ułatwia wykonanie takiej zmiany — wystarczy dodać tylko jeden wiersz kodu, tak jak pokazano na następnym listingu (zobacz listing 22.3).

Listing 22.3. Definiowanie obligatoryjnego parametru wywołania skryptu Get-DiskInventory.ps1

```
<#
```

```
.SYNOPSIS
```

Get-DiskInventory retrieves logical disk information from one or more computers.

```
.DESCRIPTION
```

Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk instances from one or more computers. It displays each disk's drive letter, free space, total size, and percentage of free

space.

.PARAMETER computername

The computer name, or names, to query. Default: Localhost.

.PARAMETER drivetype

The drive type to query. See Win32_LogicalDisk documentation for values. 3 is a fixed disk, and is the default.

.EXAMPLE

Get-DiskInventory -computername SERVER-R2 -drivetype 3

#>[BP2]

[CmdletBinding()]

param (

[Parameter(Mandatory=\$True)]

[string]\$computername,

[int]\$drivetype = 3

)

Get-WmiObject -class Win32_LogicalDisk -computername \$computername `

-filter "drivetype=\$drivetype" |

Sort-Object -property DeviceID |

Select-Object -property DeviceID,

@{name='FreeSpace(MB)';expression={\$_.FreeSpace / 1MB -as [int]}}.

@{name='Size(GB)';expression={\$_.Size / 1GB -as [int]}}.

*@{name='%Free';expression={\$_.FreeSpace / \$_.Size * 100 -as [int]}}*

Dla zainteresowanych

Kiedy inny użytkownik uruchomi Twój skrypt, ale nie wpisze obligatoryjnego parametru wywołania, powłoka PowerShell poprosi go o to. Istnieją dwa sposoby, aby znaki zachęty powłoki PowerShell były bardziej zrozumiałe dla użytkownika.

Przed wszystkim powinieneś użyć odpowiedniej nazwy parametru. Monitowanie użytkownika o podanie wartości parametru `comp` nie jest tak pomocne, jak prośba o podanie nazwy komputera dla parametru `computername`; z tego względu powinieneś stosować nazwy parametrów, które są opisowe i spójne z innymi parametrami poleceń powłoki PowerShell.

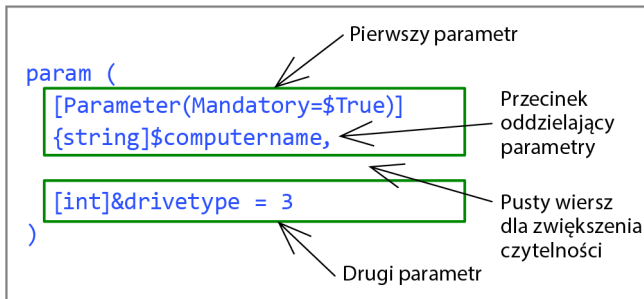
W razie potrzeby możesz również dodać pomocniczy komunikat:

```
[Parametr (Mandatory = $ True, HelpMessage = "Wprowadź nazwę komputera")
```

Niektóre hosty powłoki PowerShell wyświetlają taki komunikat jako część monitu, dzięki czemu całość staje się jeszcze bardziej czytelna dla użytkownika. Niestety nie każda aplikacja hosta będzie używać tego atrybutu, więc nie powinieneś przejmować się, jeżeli podczas testowania skryptu ten komunikat nie zawsze będzie widoczny. Tak czy inaczej, w naszych skryptach zawsze staramy się dołączać takie komunikaty, zwłaszcza jeżeli z danego skryptu ma z założenia korzystać wielu użytkowników — to nigdy nie zaszkodzi. Aby jednak zachować zwięźłość omawianego przykładu, w kodzie źródłowym naszego skryptu tym razem pominiemy komunikat `HelpMessage`.

Dodanie do naszego parametru tylko tego jednego **dekoratora** (ang. *decorator*), `[Parameter(Mandatory = $ True)]`, sprawi, że powłoka PowerShell automatycznie poprosi o podanie nazwy komputera, jeżeli użytkownik uruchamiający skrypt zapomni jej podać. Aby jeszcze bardziej ułatwić powłocie zadanie, zdefiniowaliśmy dla obu parametrów wywołania oczekiwany typ danych: `[string]` dla parametru `-computername` oraz `[int]` (liczba całkowita) dla parametru `-drivetype`.

Dodawanie tego rodzaju atrybutów do parametrów wywołania skryptu może być nieco mylące, więc przyjrzyjmy się bliżej składni bloku `Param()`, która została przedstawiona na rysunku 22.1.



Rysunek 22.1. Składnia bloku `Param()`

Poniżej zamieszczamy zestawienie kilku najważniejszych spraw, o których powinienś pamiętać:

- Wszystkie parametry wywołania skryptu znajdują się wewnątrz nawiasów bloku `Param()`.
- Pojedynczy parametr może składać się z wielu dekoratorów, które można umieścić w jednym wierszu kodu lub rozłożyć na wiele wierszy, jak to pokazano na rysunku 22.1. Naszym zdaniem użycie wielu wierszy powoduje, że definicja parametru jest bardziej czytelna, ale w sumie najważniejsze jest to, że wszystkie dekoratory mamy w jednym miejscu. W tym przykładzie atrybut `Mandatory` modyfikuje tylko parametr `-computerName` i nie ma żadnego wpływu na parametr `-drivetype`.
- Po każdej nazwie parametru (oprócz ostatniego) wstawiamy przecinek.
- Aby zwiększyć czytelność bloku parametrów, pomiędzy definicjami parametrów możemy wstawiać pusty wiersz. Ułatwia to wizualne odseparowanie od siebie poszczególnych definicji parametrów i powoduje, że blok `Param()` jest łatwiejszy do przeanalizowania.
- Wszystkie parametry definiujemy tak, jakby były zmiennymi — w tym przypadku `$computername` i `$drivetype` — ale użytkownik uruchamiający ten skrypt będzie je traktował jak normalne parametry wywołania polecenia powłoki PowerShell, takie jak `-computername` i `-drivetype`.

ZRÓB TO SAM Zapisz w pliku skrypt przedstawiony na listingu 22.3 i następnie spróbuj uruchomić go w powłoce. Nie podawaj parametru `-computername` i zobacz, jak powłoka PowerShell zareaguje, wyświetlając prośbę o podanie wartości tego parametru.

22.4. Dodawanie aliasów parametrów

Czy parametr `computername` jest pierwszą rzeczą, która przychodzi Ci do głowy, kiedy myślisz o nazwach komputerów? Prawdopodobnie nie. Użyliśmy `-computerName` jako naszej nazwy parametru, ponieważ jest ona spójna z nazwą podobnych parametrów innych poleceń powłoki PowerShell. Jeżeli przyjrzyś się poleceniom takim jak `Get-Service`, `Get-WmiObject` czy `Get-Process`, wszędzie tam znajdziesz parametr `-computerName` i dlatego tworząc nasz skrypt, poszliśmy tą samą drogą.

Jeżeli jednak uważasz, że znacznie bardziej odpowiednią nazwą dla tego parametru będzie `-hostname` czy coś podobnego, możesz to dodać jako alternatywną nazwę lub alias tego parametru. Aby to zrobić, wystarczy po prostu w definicji parametru umieścić kolejny dekorator, tak jak pokazano na listingu 22.4 poniżej.

Listing 22.4. Dodawanie aliasu parametru w skrypcie `Get-DiskInventory.ps1`

```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or
more computers.
.DESCRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
drive letter, free space, total size, and percentage of free
space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.PARAMETER drivetype
The drive type to query. See Win32_LogicalDisk documentation
for values. 3 is a fixed disk, and is the default.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
#>
[CmdletBinding()]
param (
    [Parameter(Mandatory=$True)]
    [Alias('hostname')]
    [string]$computername,
    [int]$drivetype = 3
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=$drivetype" |
    Sort-Object -property DeviceID |
    Select-Object -property DeviceID,
        @{name='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
        @{name='Size(GB)';expression={$_.Size / 1GB -as [int]}},
        @{name='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

Dzięki wprowadzeniu tej niewielkiej zmiany możemy teraz uruchomić następujące polecenie:

```
PS C:\>. \Get-DiskInventory -host SERVER2
```

UWAGA Jak pamiętasz, aby powłoka PowerShell była w stanie rozpoznać parametr, który masz na myśli, powinieneś wpisać przynajmniej taki fragment jego nazwy, który będzie w jednoznaczny sposób pozwalał na identyfikację tego parametru. W tym przypadku ciąg znaków `-host` jest wystarczający dla powłoki PowerShell do zidentyfikowania parametru `-hostname`. Oczywiście nic nie stoi na przeszkodzie, abyś używał pełnych nazw parametrów.

Podobnie jak w poprzednim przypadku, nowy dekorator jest częścią definicji parametru `-computername` i nie ma żadnego wpływu na parametr `-drivetype`. Definicja parametru `-computername` składa się teraz z trzech wierszy kodu, choć oczywiście moglibyśmy połączyć to wszystko w jeden wiersz.

```
[Parameter(Mandatory=$True)][Alias('hostname')][string]$computername.
```

Naszym skromnym zdaniem taka forma zapisu powoduje, że definicja parametrów skryptu jest znacznie mniej czytelna.

22.5. Sprawdzanie poprawności wartości parametru

Popracujemy teraz trochę z parametrem `-drivetype`. Zgodnie z dokumentacją MSDN dla klasy WMI `Win32_LogicalDisk` (wpisz nazwę tej klasy w wyszukiwarce sieciowej, a jednym z pierwszych wyników będzie zapewne jej dokumentacja), dyski typu 3 odpowiadają lokalnym dyskom twardym. Typ 2 to dysk wymienny, który również posiada takie właściwości jak rozmiar czy ilość wolnego miejsca. Dyski typów 1, 4, 5 i 6 są mniej interesujące (czy ktoś jeszcze używa popularnych niegdyś RAM-dysków, typ 6?); niektóre typy dysków nie mają właściwości reprezentujących ilość wolnego miejsca (na przykład typ 5, dyski optyczne) — z tych i innych powodów nie chcemy, aby takie typy dysków były używane podczas wywoływania naszego skryptu.

Listing 22.5 pokazuje zmiany, jakie musieliśmy wprowadzić w kodzie.

Listing 22.5. Dodawanie sprawdzania poprawności parametrów w skrypcie `Get-DiskInventory.ps1`

```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or
more computers.
.DESCRPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
drive letter, free space, total size, and percentage of free
space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.PARAMETER drivetype
The drive type to query. See Win32_LogicalDisk documentation
for values. 3 is a fixed disk, and is the default.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
```

```
#>
[CmdletBinding()]
param (
    [Parameter(Mandatory=$True)]
    [Alias('hostname')]
    [string]$computername,
    [ValidateSet(2,3)]
    [int]$drivetype = 3
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=$drivetype" |
Sort-Object -property DeviceID |
Select-Object -property DeviceID,
    @{name='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
    @{name='Size(GB)';expression={$_.Size / 1GB -as [int]}},
    @{name='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

Nowy dekorator w definicji parametru `-drivetype` informuje powłokę PowerShell, że parametr ten powinien przyjmować tylko dwie wartości, 2 lub 3, gdzie 3 to wartość domyślna.

Istnieje kilka innych technik sprawdzania poprawności wartości parametrów, a kiedy sytuacja tego wymaga, można nawet zastosować więcej niż jeden sposób sprawdzania tego samego parametru. Aby uzyskać więcej szczegółowych informacji na ten temat, powinieneś uruchomić polecenie `help about_functions_advanced_parameters`. Póki co dla naszych potrzeb pozostaniemy przy użyciu dekoratora `ValidateSet()`. Jeffery Hicks jest autorem świetnego zestawu artykułów na temat niektórych atrybutów rodziny `Validate`. Możesz je znaleźć na jego blogu <https://jdhitsolutions.com/> (poszukaj słowa kluczowego *validate*).

ZRÓB TO SAM Zapisz skrypt na dysku i spróbuj go ponownie uruchomić. Przy kolejnym wywołaniu skryptu spróbuj użyć parametru `-drivetype 5` i zobacz, co zrobi powłoka PowerShell.

22.6. Wyświetlanie szczegółowych wyników działania

W rozdziale 19. wspominaliśmy, że wolimy używać polecenia `Write-Verbose` niż `Write-Host` do wyświetlania szczegółowych informacji o postępach działania skryptu, które niektórzy użytkownicy lubią umieszczać w swoich skryptach. Nadszedł czas na przedstawienie konkretnego przykładu.

W naszym kodzie (listing 22.6) umieściliśmy kilka dodatkowych wierszy, które mają na celu informowanie użytkowników o działaniu skryptu.

Listing 22.6. Dodawanie dodatkowych komunikatów do skryptu `Get-DiskInventory.ps1`

```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or
more computers.
.DESCRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
```


drive letter, free space, total size, and percentage of free space.

.PARAMETER computername

The computer name, or names, to query. Default: Localhost.

.PARAMETER drivetype

The drive type to query. See Win32_LogicalDisk documentation for values. 3 is a fixed disk, and is the default.

.EXAMPLE

Get-DiskInventory -computername SERVER-R2 -drivetype 3

#>[BP3]

[CmdletBinding()]

param (

[Parameter(Mandatory=\$True)]

[Alias('hostname')]

[string]\$computername,

[ValidateSet(2,3)]

[int]\$drivetype = 3

)

Write-Verbose "Łączenie z \$computername"

Write-Verbose "Wyszukiwanie dysków typu \$drivetype"

Get-WmiObject -class Win32_LogicalDisk -computername \$computername `

-filter "drivetype=\$drivetype" |

Sort-Object -property DeviceID |

Select-Object -property DeviceID,

@{name='FreeSpace(MB)';expression={\$_.FreeSpace / 1MB -as [int]}},

@{name='Size(GB)';expression={\$_.Size / 1GB -as [int]}},

*@{name='%Free';expression={\$_.FreeSpace / \$_.Size * 100 -as [int]}}*

Write-Verbose "Polecenie zakończyło działanie."

Teraz spróbuj uruchomić ten skrypt na dwa sposoby. W pierwszej próbie na ekranie nie powinny się pojawić żadne dodatkowe informacje:

```
PS C:\> .\Get-DiskInventory -computername localhost
```

W drugim przypadku chcemy wyświetlić szczegółowe informacje o działaniu skryptu:

```
PS C:\> .\Get-DiskInventory -computername localhost -verbose
```

ZRÓB TO SAM To wszystko będzie znacznie fajniejsze, jeżeli na własne oczy przekonasz się, jak to działa. Uruchom skrypt tak, jak to pokazaliśmy powyżej, i zobacz różnice w jego działaniu.

Dlaczego to takie fajne? Kiedy chcesz, aby skrypt wyświetlał szczegółowe informacje o swoim działaniu, możesz to łatwo zrobić i nie musisz w ogóle kodować parametru `-Verbose`! Jest on dostępny od razu po dodaniu wiersza `[Cmdlet-Binding()]`. A naprawdę fajne jest to, że dodanie tego parametru aktywuje również wyświetlanie szczegółowych danych wyjściowych dla każdego polecenia znajdującego się w Twoim skrypcie, stąd w takiej sytuacji wszystkie polecenia, które są używane do generowania szczegółowych wyników, będą robić to „automagicznie”. Taka technika znacznie ułatwia włączanie i wyłączanie wyświetlania szczegółowych danych, dzięki czemu jest bardziej elastyczna niż zastosowanie polecenia `Write-Host`, a dodatkowo nie musisz się martwić sprawami

związanymi z odpowiednim ustawianiem zmiennej `$VerbosePreference` do wyświetlania danych na ekranie.

W szczegółowych wynikach działania skryptu możesz również zauważyć, jak wykorzystaliśmy sztuczkę z interpretacją zmiennych ujętych w cudzysłów: dzięki umieszczeniu zmiennej `$computername` w cudzysłowie dane wyjściowe zawierają zawartość tej zmiennej, co pozwala nam zobaczyć, z jakim komputerem łączy się powłoka PowerShell.

22.7. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Zadania zamieszczone w tym zestawie wymagają przypomnienia sobie części tego, czego nauczyłeś się w rozdziale 21., ponieważ będziesz musiał wziąć przedstawione niżej polecenie, sparametryzować je i zamienić w skrypt — tak jak to robiliśmy w ćwiczeniach w rozdziale 21. Tym razem jednak chcielibyśmy, aby parametr `-computerName` był obligatoryjny i posiadał alias o nazwie `hostname`. Skrypt powinien również wyświetlać pełne wyniki przed uruchomieniem tego polecenia i po wykonaniu tej czynności. Pamiętaj, musisz sparametryzować nazwę komputera — ale tym razem to jedyny element, który musisz sparametryzować.

Zanim rozpoczniesz modyfikowanie, powinieneś uruchomić polecenie przedstawione poniżej, tak aby upewnić się, że działa ono w Twoim systemie:

```
get-wmiobject win32_networkadapter -computername localhost |
  where { $_.PhysicalAdapter } |
  select MACAddress,AdapterType,DeviceID,Name,Speed
```

Podsumowując, oto Twoja pełna lista zadań:

- Przed rozpoczęciem modyfikacji upewnij się, że przedstawione polecenie działa poprawnie w Twoim systemie.
- Dokonaj parametryzacji nazwy komputera.
- Ustaw parametr reprezentujący nazwę komputera jako obligatoryjny.
- Do parametru reprezentującego nazwę komputera dodaj alias `hostname`.
- Utwórz system pomocy opartej na komentarzach, zawierający co najmniej jeden przykład korzystania ze skryptu.
- Dodaj możliwość wyświetlania szczegółowych danych wyjściowych przed uruchomieniem zmodyfikowanego zapytania WMI i po wykonaniu tej czynności.
- Zapisz skrypt w pliku o nazwie *Get-PhysicalAdapters.ps1*.

22.8. Odpowiedzi

```
#Get-PhysicalAdapters.ps1
<#
.Synopsis
Get physical network adapters
```

*.Description**Display all physical adapters from the Win32_NetworkAdapter class.**.Parameter Computername**The name of the computer to check.**.Example**PS C:\> c:\scripts\Get-PhysicalAdapters -computer SERVER01*

#>

[cmdletbinding()]

Param (

[Parameter(Mandatory=\$True,HelpMessage="Enter a computername to query")]

[alias('hostname')]

[string]\$Computername

)

Write-Verbose "Pobieranie informacji o kartach sieciowych z komputera \$computername"

Get-Wmiobject -class win32_networkadapter -computername \$computername |

where { \$_.PhysicalAdapter } |

select MACAddress,AdapterType,DeviceID,Name,Speed

Write-Verbose "Koniec działania skryptu."

Zaawansowana konfiguracja komunikacji zdalnej

W rozdziale 13. staraliśmy się wprowadzić Cię w świat komunikacji zdalnej z użyciem powłoki PowerShell. Celowo jednak nie poruszyliśmy kilku ważnych tematów, abyś mógł się skupić na podstawowych technologiach i technikach związanych z działaniem tego mechanizmu w powłocie PowerShell. W tym rozdziale chcielibyśmy jednak do nich powrócić i omówić niektóre bardziej zaawansowane funkcje i scenariusze. Z góry zastrzegamy, że nie wszystko w tym rozdziale będzie użyteczne dla wszystkich — ale wychodzimy z założenia, że każdy powinien wiedzieć coś niecoś o tych opcjach, na wypadek gdyby w przyszłości stanął przed koniecznością ich zastosowania.

Ponadto przypominamy, że w tej książce koncentrujemy się na powłocie PowerShell v3 i jej nowszych wersjach, które zasadniczo korzystają z tego samego mechanizmu zdalnego dostępu. Jeżeli nie jesteś pewien, z jakiej wersji powłoki korzystasz, odpowiednie informacje znajdziesz w rozdziale 1. — pamiętaj, że wiele z tego, co tutaj prezentujemy, nie będzie działać w starszych wersjach powłoki.

23.1. Korzystanie z innych punktów końcowych

Jak dowiedziałeś się z rozdziału 13., jeden komputer może posiadać wiele **punktów końcowych** (ang. *endpoints*), które powłoka PowerShell określa jako **konfiguracje sesji** (ang. *session configurations*). Na przykład włączenie komunikacji zdalnej na komputerze działającym pod kontrolą 64-bitowego systemu operacyjnego powoduje włączenie punktu końcowego dla 32-bitowej powłoki PowerShell oraz 64-bitowej powłoki PowerShell, przy czym 64-bitowy punkt końcowy jest traktowany jako domyślny.

Listę dostępnych konfiguracji sesji możesz sprawdzić na dowolnym komputerze, do którego masz dostęp na poziomie administratora:

```
PS C:\> Get-PSSessionConfiguration
Name           : microsoft.powershell
PSVersion      : 3.0
StartupScript  :
RunAsUser      :
Permission     : NT AUTHORITY\NETWORK AccessDenied, BUILTIN\Administrators
                AccessAllowed
Name           : microsoft.powershell.workflow
PSVersion      : 3.0
StartupScript  :
RunAsUser      :
Permission     : NT AUTHORITY\NETWORK AccessDenied, BUILTIN\Administrators
                AccessAllowed
Name           : microsoft.powershell32
PSVersion      : 3.0
StartupScript  :
RunAsUser      :
Permission     : NT AUTHORITY\NETWORK AccessDenied, BUILTIN\Administrators
                AccessAllowed
```

Każdy punkt końcowy ma swoją nazwę; ten o nazwie Microsoft.PowerShell to taki, który jest domyślnie używany przez polecenia takie jak New-PSSession, Enter-PSSession czy Invoke-Command. W systemach 64-bitowych ten punkt końcowy to 64-bitowa powłoka PowerShell; w systemach 32-bitowych Microsoft.PowerShell to oczywiście powłoka PowerShell w wersji 32-bitowej.

Z pewnością zauważyłeś, że nasz 64-bitowy system ma alternatywny punkt końcowy Microsoft.PowerShell32 z 32-bitową powłoką zapewniającą kompatybilność wsteczną. Aby połączyć się z alternatywnym punktem końcowym, należy podać jego nazwę w parametrze -ConfigurationName polecenia, za pomocą którego łączymy się z takim systemem:

```
PS C:\> Enter-PSSession -ComputerName DONJONES1D96 -ConfigurationName
'Microsoft.PowerShell32'
[DONJONES1D96]: PS C:\Users\donjones\Documents>
```

Kiedy powinieneś używać alternatywnego punktu końcowego? Przykładowo, jeżeli musisz uruchomić polecenie wykorzystujące 32-bitową przystawkę PSSnapin, to może to być jeden z powodów konieczności jawnego połączenia się z 32-bitowym punktem końcowym na maszynie 64-bitowej. W razie potrzeby możesz także skonfigurować niestandardowe punkty końcowe i łączyć się z nimi w celu wykonania określonego zadania.

23.2. Tworzenie niestandardowych punktów końcowych

Tworzenie niestandardowego punktu końcowego jest procesem dwuetapowym:

1. Używamy polecenia New-PSSessionConfigurationFile do utworzenia nowego pliku konfiguracyjnego sesji, który powinien mieć rozszerzenie .PSSC. W tym pliku zdefiniowanych jest wiele charakterystyk punktu końcowego, takich jak polecenia i możliwości.

2. Używamy polecenia `Register-PSSessionConfiguration`, aby załadować plik `.PSSC` i utworzyć nowy punkt końcowy w usłudze WinRM. Podczas rejestrowania można ustawić wiele parametrów operacyjnych, takich jak zdefiniowanie, kto może łączyć się z tym punktem końcowym. W razie potrzeby można zmienić te ustawienia później, za pomocą polecenia `Set-PSSession-Configuration`.

W tym podrozdziale omówimy przykład, który wykorzystuje niestandardowe punkty końcowe do delegowanego zarządzania, co jest prawdopodobnie jednym z ich najfajniejszych zastosowań. Utworzymy punkt końcowy, z którym będą mogli łączyć się członkowie grupy HelpDesk naszej domeny. W tym punkcie końcowym udostępnimy tylko polecenia pozwalające na zarządzanie kartami sieciowymi. Nie planujemy udzielania naszemu działowi pomocy technicznej uprawnień niezbędnych do uruchamiania tych poleceń, chcemy tylko, aby te polecenia były dla niego widoczne. Skonfigurujemy również nasz punkt końcowy, tak aby można było uruchamiać polecenia na podstawie dostarczonych przez nas alternatywnych poświadczeń logowania, dzięki czemu będziemy mogli korzystać z tych poleceń bez konieczności nadawania uprawnień naszemu zespołowi pomocy technicznej.

23.2.1. Tworzenie konfiguracji sesji

Poniżej przedstawiamy polecenie, które musimy uruchomić (na potrzeby tej książki polecenie zostało ładnie sformatowane, ale w rzeczywistości zostało napisane jako jeden długi wiersz):

```
PS C:\> New-PSSessionConfigurationFile
➤ -Path C:\HelpDeskEndpoint.pssc
➤ -ModulesToImport NetAdapter
➤ -SessionType RestrictedRemoteServer
➤ -CompanyName "Our Company"
➤ -Author "Don Jones"
➤ -Description "Net adapter commands for use by help desk"
➤ -PowerShellVersion '3.0'
```

Mamy tutaj kilka kluczowych parametrów, które zostały wyróżnione pogrubioną czcionką. Poniżej objaśniamy znaczenie wyróżnionych parametrów. Pozostawiamy Ci samodzielne odszukanie w pliku pomocy znaczenia innych opcji:

- Parametr `-Path` jest obligatoryjny, a nazwa pliku, którą podajesz, powinna kończyć się na `.PSSC`.
- Parametr `-ModulesToImport` określa listę modułów (w tym przypadku tylko jeden, o nazwie `NetAdapter`), które mają być dostępne za pośrednictwem tego punktu końcowego.
- Parametr `-SessionType RestrictedRemoteServer` usuwa wszystkie podstawowe polecenia powłoki PowerShell z wyjątkiem krótkiej listy najbardziej niezbędnych, takich jak `Select-Object`, `Measure-Object`, `Get-Command`, `Get-Help`, `Exit-PSSession` i kilku innych.
- Parametr `-PowerShellVersion` domyślnie przyjmuje wartość 3.0, ale uwzględniliśmy go dla zachowania kompletności składni polecenia.

Istnieje jeszcze kilka innych dostępnych parametrów, których nazwy zaczynają się od `-Visible`, takich jak `-VisibleCmdlets`. Zazwyczaj, gdy importujesz moduł za pomocą parametru `-ModulesToImport`, każde polecenie w module jest widoczne dla użytkowników korzystających z tego punktu końcowego; w razie potrzeby jednak można to zmienić za pomocą odpowiedniego parametru `-Visible`. Wymieniając tylko cmdlety, aliasy, funkcje i dostawców, których użytkownicy powinni widzieć, ukrywasz całą resztę. To dobry sposób na ograniczenie tego, co użytkownik może zrobić z Twoim punktem końcowym. Korzystając z parametrów rodziny `-Visible`, powinieneś jednak zachować ostrożność, ponieważ mogą one być nieco mylące. Na przykład jeżeli importujesz moduł złożony zarówno z poleceń `cmdlet`, jak i funkcji, użycie opcji `-VisibleCmdlets` ogranicza tylko widoczność cmdletów, ale nie ma żadnego wpływu na funkcje, co oznacza, że domyślnie wszystkie funkcje pozostają widoczne.

Zauważ, że nie ma możliwości ograniczenia parametrów, z których mogą korzystać poszczególne polecenia: powłoka PowerShell obsługuje co prawda ograniczenia na poziomie parametrów, ale aby skorzystać z takich możliwości, musisz użyć Visual Studio do wykonania kilku naprawdę zaawansowanych trików programistycznych, co wykracza daleko poza ramy tej książki. Oprócz tego możesz użyć również innych, mocno zaawansowanych sztuczek, takich jak tworzenie specjalnych funkcji, które ukrywają określone parametry, ale takie zagadnienia są również poza zakresem naszej książki, przeznaczonej dla początkujących użytkowników powłoki PowerShell.

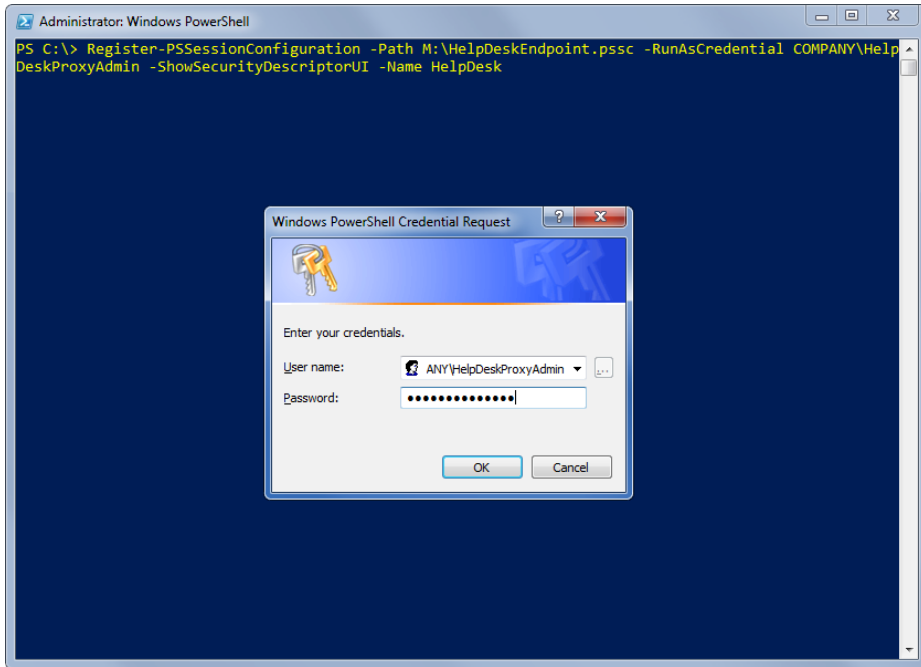
23.2.2. Rejestrowanie sesji

Po utworzeniu pliku konfiguracji sesji musimy ją zarejestrować za pomocą polecenia, które przedstawiamy poniżej. Podobnie jak poprzednio, sformatujemy je na potrzeby książki, ale w rzeczywistości wpisujemy w jednym długim wierszu poleceń:

```
PS C:\> Register-PSSessionConfiguration
    ➤ -Path .\HelpDeskEndpoint.pssc
    ➤ -RunAsCredential COMPANY\HelpDeskProxyAdmin
    ➤ -ShowSecurityDescriptorUI
    ➤ -Name HelpDesk
```

Wykonanie tego polecenia spowoduje utworzenie nowego punktu końcowego o nazwie po prostu `HelpDesk` (w przeciwieństwie do `Microsoft.PowerShell` lub innych tego typu). Jak pokazano na rysunku 23.1, powłoka poprosi o podanie hasła do konta `COMPANY\HelpDeskProxyAdmin`; jest to konto, które będzie używane do uruchamiania wszystkich poleceń po połączeniu z punktem końcowym. Oczywiście musimy się upewnić, że to konto ma odpowiednie uprawnienia wymagane do uruchamiania poleceń karty sieciowej.

Po uruchomieniu tego polecenia trzeba odpowiedzieć na kilka pytań typu „czy jesteś pewien”, które zdecydowanie powinieneś bardzo uważnie przeczytać przed udzieleniem odpowiedzi. Na przykład polecenie to spowoduje zatrzymanie i ponowne uruchomienie usługi WinRM, co może skutkować przerwaniem pracy innych administratorów, dlatego powinieneś zachować ostrożność.



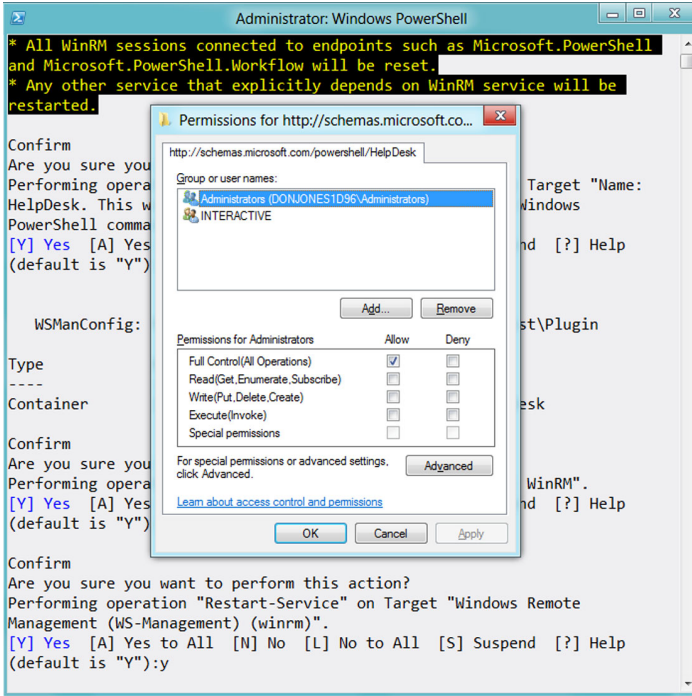
Rysunek 23.1. Prośba o podanie hasła dostępu do konta zdefiniowanego w parametrze `RunAsCredential`

Jak pokazano na rysunku 23.2, na ekranie pojawi się również okno dialogowe pozwalające na wybranie, którzy użytkownicy mogą łączyć się z punktem końcowym. Okno dialogowe pojawia się, ponieważ w wierszu wywołania, zamiast definiować uprawnienia punktu końcowego za pomocą złożonego języka SDDL (ang. *Security Descriptor Definition Language*), w którym, szczerze mówiąc, nie czujemy się zbyt pewnie, użyliśmy po prostu parametru `-ShowSecurityDescriptorUI`. Jest to jedna z tych sytuacji, w których graficzny interfejs użytkownika naprawdę się przydaje — dodajemy naszą grupę użytkowników `HelpDesk` i zapewniamy im uprawnienia `Execute` oraz `Read`. Uprawnienia do wykonania (`Execute`) to minimalne wymagane uprawnienia, biorąc pod uwagę to, co planujemy zrobić z punktem końcowym, a `Read` to jedyna rzecz, której nasi użytkownicy powinni potrzebować.

Po ustawieniu uprawnień zadanie zostało wykonane. Jak widać na podstawie poniższego fragmentu wyników działania, użytkownicy naszego nowego punktu końcowego mają dostęp tylko do ograniczonego zestawu poleceń:

```
PS C:\> Enter-PSSession -ComputerName DONJONES1D96 -ConfigurationName HelpDesk
[DONJONES1D96]: PS>Get-Command
```

Capability	Name	ModuleName
CIM	Disable-NetAdapter	NetA...
CIM	Disable-NetAdapterBinding	NetA...
CIM	Disable-NetAdapterChecksumOffload	NetA...
CIM	Disable-NetAdapterEncapsulatedPacketTaskOffload	NetA...



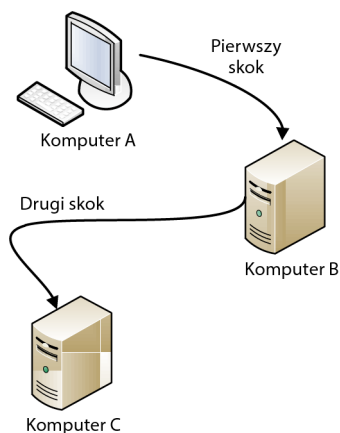
Rysunek 23.2. Ustawianie uprawnień punktu końcowego

CIM	Disable-NetAdapterIPsecOffload	NetA...
CIM	Disable-NetAdapterLso	NetA...
CIM	Disable-NetAdapterPowerManagement	NetA...
CIM	Disable-NetAdapterQos	NetA...
CIM	Disable-NetAdapterRdma	NetA...
CIM	Disable-NetAdapterRsc	NetA...
CIM	Disable-NetAdapterRss	NetA...
CIM	Disable-NetAdapterSriov	NetA...
CIM	Disable-NetAdapterVmq	NetA...
CIM	Enable-NetAdapter	NetA...
CIM	Enable-NetAdapterBinding	NetA...
CIM	Enable-NetAdapterChecksumOffload	NetA...
CIM	Enable-NetAdapterEncapsulatedPacketTaskOffload	NetA...
CIM	Enable-NetAdapterIPsecOffload	NetA...
CIM	Enable-NetAdapterLso	NetA...
CIM	Enable-NetAdapterPowerManagement	NetA...
CIM	Enable-NetAdapterQos	NetA...

To świetny sposób na stworzenie określonego zestawu możliwości dla wybranej grupy użytkowników. Nie muszą oni koniecznie łączyć się z powłoką PowerShell z sesji konsoli, tak jak to zrobiliśmy na potrzeby tego testu; zamiast tego mogą używać narzędzi wyposażonych w graficzny interfejs użytkownika, które „pod maską” wykorzystują mechanizmy komunikacji zdalnej powłoki PowerShell. Przy założeniu, że używane narzędzie potrzebuje tylko tych wybranych poleceń, opisana technika jest świetnym sposobem na przekazanie innym użytkownikom wyłącznie takich delegowanych możliwości.

23.3. Włączanie mechanizmu komunikacji zdalnej za pośrednictwem hostów pośrednich

Włączanie mechanizmu komunikacji zdalnej za pośrednictwem wielu hostów pośrednich jest tematem, o którym krótko wspomnieliśmy w rozdziale 13., ale z pewnością zasługuje on na nieco więcej. Rysunek 23.3 ilustruje tzw. **problem drugiego skoku** (ang. *second hop problem*) lub inaczej mówiąc, **problem wielu skoków** (ang. *multihop problem*): zaczynasz pracę na komputerze A i tworzysz połączenie PSSession z komputerem B. To pierwszy skok i prawdopodobnie wszystko będzie dobrze. Ale wtedy próbujesz zmusić komputer B, aby utworzył drugi skok lub inaczej mówiąc, drugie połączenie, do komputera C i wtedy cała operacja bierze w łeb.



Rysunek 23.3. Komunikacja zdalna z wykorzystaniem hosta pośredniego

Opisany problem jest związany ze sposobem, w jaki powłoka PowerShell deleguje Twoje poświadczenia logowania z komputera A na komputer B. **Delegowanie** to proces umożliwiający komputerowi B wykonywanie zadań tak, jakby był Tobą, zapewniając w ten sposób, że możesz zrobić wszystko, do czego normalnie masz uprawnienia, ale nic więcej. Domyślnie delegowanie może przechodzić tylko jeden taki skok, ponieważ komputer B nie ma uprawnień do delegowania Twoich poświadczeń na trzeci komputer, komputer C.

W systemie Windows Vista i wszystkich nowszych wersjach systemu Windows można włączyć wieloskokową delegację uprawnień (ang. *multihop delegation*) w wielu aplikacjach. Potrzebne są do tego dwa kroki:

1. Na Twoim komputerze (w naszym przykładzie jest to komputer A) uruchom polecenie `Enable-WSManCredSSP-Role Client -DelegateComputer x`, gdzie zamiast `x` wpisz nazwę komputera, do którego chcesz przekazywać swoje poświadczenia. Możesz tutaj podać nazwę komputera, ale równie dobrze użyć symboli wieloznacznych. Nie zalecamy jednak stosowania znaku gwiazdki (*), może to spowodować bardzo poważne problemy związane z bezpieczeństwem; zamiast tego możesz na przykład autoryzować całą domenę: `*.company.com`.

2. Na pierwszym komputerze, z którym się łączysz (w tym przykładzie jest to komputer B), uruchom polecenie `Enable-WSManCredSSP -Role Server`.

Zmiany wprowadzone przez polecenie są stosowane do ustawień lokalnej polityki bezpieczeństwa; można również wprowadzić je ręcznie za pośrednictwem zasad grupy (GPO), co może mieć więcej sensu w dużym środowisku domenowym. Zarządzanie delegowaniem poświadczeń za pomocą zasad grupy wykracza daleko poza zakres tego rozdziału, ale więcej szczegółowych informacji na ten temat możesz znaleźć w pliku pomocy dla polecenia `Enable-WSManCredSSP`. Warto zauważyć, że Don jest autorem przewodnika *Secrets of PowerShell Remoting*, dostępnego na stronie <https://powershell.org/>, który zawiera szczegółowy opis tego zagadnienia.

23.4. Zaawansowane uwierzytelnianie na komputerach zdalnych

Niejednokrotnie przekonywaliśmy się, że wielu użytkowników ma tendencję do myślenia o uwierzytelnianiu jako procesie jednokierunkowym: jeżeli chcesz uzyskać dostęp do komputera zdalnego, musisz dostarczyć mu swoje dane uwierzytelniające. Ale mechanizm komunikacji zdalnej powłoki PowerShell wykorzystuje wzajemne uwierzytelnianie, co oznacza, że maszyna zdalna również musi udowodnić swoją tożsamość. Jeżeli uruchomisz polecenie `Enter-PSSession -computerName DC01`, to zanim ono zostanie nawiązane, komputer o nazwie DC01 musi udowodnić, że naprawdę jest tym, za kogo się podaje.

Dlaczego? Zazwyczaj Twój komputer zamienia nazwę komputera (na przykład DC01) na odpowiadający mu adres IP przy użyciu usługi DNS (ang. *Domain Name System*). Niestety DNS nie jest odporny na ataki, więc nie jest nie do pomyślenia, aby atakujący mógł się dostać do serwera DNS i zmodyfikować wpis dla komputera DC01 tak, aby wskazywał na inny adres IP, który kontroluje napastnik. W takiej sytuacji możesz zupełnie nieświadomie połączyć się z „falszywym” DC01, a następnie zacząć przekazywać swoje poświadczenia logowania — a to oznacza kłopoty! Wzajemne uwierzytelnianie uniemożliwia takie działanie — jeżeli komputer, z którym się łączysz, nie może udowodnić, że jest tym, z którym chcesz się połączyć, próba nawiązania takiego połączenia zakończy się niepowodzeniem. To bardzo dobry mechanizm i nie powinieneś wyłączać takiej ochrony bez dokładnego zaplanowania działania i rozważenia konsekwencji, jakie taki krok ze sobą niesie.

23.4.1. Ustawienia domyślne wzajemnego uwierzytelniania

Firma Microsoft przyjęła założenie, że w większości przypadków powłoka PowerShell będzie używana w środowiskach domenowych Active Directory. W takiej sytuacji, jeżeli do tworzenia połączeń będziesz używał nazw komputerów posiadających konta w usłudze Active Directory, wzajemne uwierzytelnianie będzie obsługiwane przez domenę.

Dzieje się tak nawet wtedy, kiedy uzyskujesz dostęp do komputerów w innych zaufanych domenach. Cała sztuczka polega na tym, że musisz podać powłoce PowerShell nazwę komputera, która spełnia następujące wymagania:

- Usługa DNS musi być w stanie zamienić taką nazwę komputera na adres IP.
- Podana nazwa musi być zgodna z nazwą komputera zarejestrowaną w katalogu usługi Active Directory.

Podanie nazwy komputera z tej samej domeny lub pełnej, kwalifikowanej nazwy komputera z domeny ufającej (pełna nazwa kwalifikowana obejmuje nazwę komputera oraz nazwę domeny, takiej jak `DC01.COMPANY.LOC`) zwykle spełnia oba warunki. Ale jeżeli musisz podać adres IP komputera zdalnego lub zamiana nazwy na adres IP przez usługę DNS wymaga podania innej nazwy (na przykład aliasu `CNAME`), domyślne uwierzytelnianie wzajemne nie będzie działać. W takiej sytuacji do wyboru pozostają Ci dwie możliwości: `SSL` lub `TrustedHosts`.

23.4.2. Wzajemne uwierzytelnianie przez SSL

Aby dokonać wzajemnego uwierzytelnienia przy użyciu protokołu `SSL`, musisz uzyskać odpowiedni certyfikat cyfrowy `SSL` dla urządzenia docelowego. Certyfikat musi zostać wydany dla tej samej nazwy komputera, którą wpisujesz, aby uzyskać dostęp do komputera. Oznacza to, że jeżeli używasz polecenia `Enter-PSsession -computerName DC01.COMPANY.LOC -UseSSL -credential COMPANY\Administrator`, to certyfikat zainstalowany na komputerze `DC01` musi być wydany dla `dc01.company.loc` lub cały proces zakończy się niepowodzeniem. Zauważ, że w tym scenariuszu parametr `-credential` jest obowiązkowy.

Po uzyskaniu certyfikatu należy go zainstalować w osobistym magazynie certyfikatów danego komputera — co najłatwiej możesz uzyskać za pomocą przystawki Certyfikaty konsoli MMC (ang. *Microsoft Management Console*). Proste dwukrotne kliknięcie pliku certyfikatu zwykle umieszcza go w osobistym magazynie certyfikatów konta użytkownika, a to nie zadziała.

Po zainstalowaniu certyfikatu powinieneś utworzyć odpowiedniego listenera `HTTPS` na komputerze, informując go o użyciu nowo zainstalowanego certyfikatu. Szczegółowa instrukcja postępowania składa się z wielu kroków, a ponieważ nie jest to coś, co prawdopodobnie zrobi wielu użytkowników, nie będziemy tego tutaj opisywać. Zapoznaj się z napisanym przez Dona przewodnikiem *Secrets of PowerShell Remoting* (jest całkowicie darmowy), gdzie znajdziesz instrukcje, jak to zrobić krok po kroku, łącznie ze zrzutami ekranu.

23.4.3. Wzajemne uwierzytelnianie za pośrednictwem TrustedHosts

Używanie `TrustedHosts` jest nieco prostszą techniką niż używanie certyfikatu `SSL` i wymaga znacznie mniej przygotowań. Z drugiej strony jest to trochę bardziej niebezpieczne, ponieważ zasadniczo wyłącza wzajemne uwierzytelnianie dla wybranych hostów. Zanim takiego rozwiązania spróbujesz, musisz być w stanie jednoznacznie stwierdzić: „Nie jest możliwe, aby ktoś mógł podszyć się pod jeden z tych hostów lub zmodyfikować odpowiednie rekordy DNS”. Takie jednak dosyć odważne stwierdzenie może być prawdziwe na przykład dla wewnętrznych komputerów działających w sieci intranet Twojej firmy czy organizacji.

Następnie potrzebujesz jakiegoś sposobu na zidentyfikowanie komputerów, którym będziesz ufał bez konieczności wzajemnego uwierzytelniania. Na przykład w środowisku domenowym może to być coś w rodzaju *.COMPANY.COM dla wszystkich hostów znajdujących się w domenie *Company.com*.

W takim przypadku prawdopodobnie będziesz chciał skonfigurować ustawienie dla całej domeny, więc aby Ci to ułatwić, podajemy instrukcje dotyczące ustawienia zasad grupy. Możesz użyć tych samych instrukcji dla ustawienia zasad zabezpieczeń lokalnych jednego komputera.

W edytorze obiektów zasad grupy lub zasad zabezpieczeń lokalnych wykonaj następujące kroki:

1. Rozwiń opcję *Konfiguracja komputera*.
2. Rozwiń opcję *Szablony administracyjne*.
3. Rozwiń opcję *Składniki systemu Windows*.
4. Rozwiń opcję *Zdalne zarządzanie systemem Windows (WinRM)*.
5. Rozwiń opcję *Klient usługi WinRM*.
6. Przejdź do prawego panelu i dwukrotnie kliknij opcję *Hosty zaufane*.
7. Zaznacz opcję *Włączone* i w polu *TrustedHostList* wpisz listę zaufanych hostów. Nazwy poszczególnych hostów zaufanych można oddzielić przecinkami, na przykład *.company.com, *. Sales.company.com.

UWAGA Starsze wersje systemu Windows mogą nie mieć szablonu koniecznego do wyświetlenia tych ustawień w zasadach komputera lokalnego, a starsze kontrolery domeny mogą nie mieć takich ustawień w swoich obiektach zasad grupy. W takich sytuacjach możesz zmienić listę zaufanych hostów z poziomu powłoki PowerShell. Więcej szczegółowych informacji na ten temat znajdziesz po uruchomieniu polecenia `help about remote troubleshooting`.

Od tej chwili będziesz mógł łączyć się z wymienionymi komputerami bez konieczności wzajemnego uwierzytelniania. Aby połączyć się z takimi komputerami, we wszystkich poleceniach komunikacji zdalnej używanych do łączenia się z tymi komputerami będziesz musiał także podawać parametr `-Credential`, w przeciwnym razie próba nawiązania takiego połączenia zakończy się niepowodzeniem.

23.5. Ćwiczenia

UWAGA Do wykonania ćwiczeń w tym zestawie potrzebny Ci będzie komputer pracujący pod kontrolą systemu Windows 8, Windows Server 2008 R2 lub nowszego z zainstalowaną powłoką PowerShell v3 lub nowszą.

Na swoim komputerze lokalnym utwórz punkt końcowy o nazwie *TestPoint*. Skonfiguruj go tak, aby moduł *SmbShare* był ładowany automatycznie, ale z całego modułu widoczne powinno być tylko polecenie `Get-SmbShare`. Upewnij się także, że dostępne są również polecenia kluczowe, takie jak `Exit-PSsession`, ale nie można używać innych

cmdletów powłoki PowerShell. Nie zajmuj się określaniem specjalnych uprawnień punktów końcowych ani wyznaczaniem poświadczeń alternatywnych (ang. *Run As*).

Przetestuj swój nowo utworzony punkt końcowy, łącząc się z nim za pomocą polecenia `Enter-PSSession` (jako nazwę komputera podaj `localhost`, a jako nazwę konfiguracji podaj `TestPoint`). Po nawiązaniu połączenia uruchom polecenie `Get-Command`, aby upewnić się, że masz dostęp tylko do określonej grupy poleceń.

Pamiętaj, że to zadanie możesz wykonać tylko w systemie Windows 8, Windows Server 2012 i nowszych wersjach systemu Windows; moduł `SmbShare` nie był dostarczany z wcześniejszymi wersjami systemu Windows.

23.6. Odpowiedzi

#tworzenie konfiguracji sesji w bieżącej lokalizacji

#to dosyć długie polecenie

```
New-PSSessionConfigurationFile -Path .\SMBShareEndpoint.pssc  
-ModulesToImport SMBShare -SessionType RestrictedRemoteServer  
-CompanyName "My Company" -Author "Jane Admin"  
-Description "restricted SMBShare endpoint" -PowerShellVersion '4.0'
```

#rejestrowanie konfiguracji

```
Register-PSSessionConfiguration -Path .\SMBShareEndpoint.pssc -Name TestPoint
```

#łączenie z punktem końcowym

```
Enter-PSSession -ComputerName localhost -ConfigurationName TestPoint  
get-command  
exit-pssession
```


24

Zastosowanie wyrażeń regularnych do parsowania plików tekstowych

Wyrażenia regularne (*regex* — ang. *regular expressions*) to jeden z tych niezręcznych, mocno kłopotliwych tematów. Często nasi studenci proszą nas, aby je wytłumaczyć, a potem w połowie rozmowy dochodzą do wniosku, że jednak wcale ich nie potrzebują. Często zdarza się, że wyrażenia regularne są używane do parsowania tekstu, co jest bardzo ważne w systemach operacyjnych takich jak UNIX czy Linux. W powłocie PowerShell zazwyczaj wykonujemy znacznie mniej operacji wymagających parsowania tekstu, stąd mniejsze zapotrzebowanie na stosowanie wyrażeń regularnych. Nie zmienia to jednak w niczym faktu, że od czasu do czasu zdarzają się sytuacje, w których, pracując z powłoką PowerShell, musisz przeanalizować zawartość jakiegoś pliku tekstowego, na przykład pliku dziennika serwera IIS. W tym rozdziale będziemy opisywać wyrażenia regularne właśnie pod kątem takich zastosowań: jako narzędzi do parsowania plików tekstowych.

Nie zrozum nas źle: za pomocą wyrażeń regularnych można zrobić naprawdę o wiele więcej. Kilka innych ich zastosowań omówimy na końcu tego rozdziału. Ale aby zachęcić Cię do samodzielnej pracy, z góry zastrzegamy, że w tej książce pod żadnym względem nie wyczerpujemy zagadnień związanych z wyrażeniami regularnymi. Wyrażenia regularne mogą być *niezwykle* skomplikowane i mogą być wręcz traktowane jako technologia sama dla siebie. Z naszej strony postaramy Ci się je przybliżyć tak, abyś mógł

niemal od razu rozpocząć z nimi pracę, a jednocześnie pokażemy Ci kierunki dalszych, samodzielnych poszukiwań, jeżeli będziesz chciał się dowiedzieć czegoś więcej o wyrażeniach regularnych.

Naszym zadaniem w tym rozdziale będzie omówienie uproszczonej składni wyrażeń regularnych i pokazanie, w jaki sposób powłoka PowerShell może ich używać. Jeżeli później będziesz chciał samodzielnie korzystać z bardziej skomplikowanych wyrażeń, będziesz już wiedział, jak się do tego zabrać i jak ich używać w powłoce PowerShell.

24.1. Przeznaczenie wyrażeń regularnych

Wyrażenia regularne są pisane w specjalnym języku, a ich zadaniem jest zdefiniowanie wzorca tekstowego. Przykładowo, adres IPv4 składa się z jednej do trzech cyfr, kropki, jednej do trzech kolejnych cyfr, kropki i tak dalej. Wyrażenie regularne może być definiowane przez taki wzorzec, chociaż w naszej uproszczonej definicji spowoduje to również zaakceptowanie niepoprawnych adresów, takich jak *211.193.299.299*. Na tym polega różnica między rozpoznawaniem wzorca tekstowego a sprawdzaniem poprawności danych.

Jednym z najczęstszych zastosowań wyrażeń regularnych — i o tym właśnie będziemy mówić w tym rozdziale — jest wykrywanie określonych wzorców tekstowych w plikach tekstowych, takich jak pliki dzienników zdarzeń. Na przykład możesz napisać wyrażenie regularne, które będzie wyszukiwało konkretny tekst reprezentujący błąd HTTP 500 w pliku dziennika serwera WWW lub wyszukiwało adresy e-mail w pliku dziennika serwera SMTP. Wyrażenia regularnych oprócz stosowania do wykrywania wzorców tekstowych można używać do przechwytywania fragmentów tekstu pasujących do wzorca, co umożliwia na przykład wyodrębnienie adresów e-mail z pliku dziennika i zapisanie ich w osobnym pliku.

24.2. Podstawy składni wyrażeń regularnych

Najprostszym wyrażeniem regularnym jest ciąg tekstu, który chcesz odnaleźć. Na przykład ciąg tekstu *Don* jest, technicznie rzecz biorąc, wyrażeniem regularnym, a powłoka PowerShell dopasuje do niego takie ciągi tekstu jak *DON*, *don*, *Don* czy *DoN*. Dzieje się tak, ponieważ domyślnie powłoka PowerShell nie uwzględnia wielkości liter.

Niektóre znaki w wyrażeniach regularnych mają specjalne znaczenie i pozwalają na tworzenie wzorców wyszukiwania umożliwiających wykrywanie zmieniających się ciągów znaków. Oto kilka przykładów takich znaków:

- `\w` — zastępuje znaki występujące w słowach, takie jak litery, cyfry i znaki podkreślenia, ale bez znaków interpunkcyjnych i bez białych znaków. Wyrażenie regularne `\won` wyszuka takie ciągi znaków jak *Don*, *Ron* i *ton*, ponieważ znak `\w` zastępuje pojedynczą literę, cyfrę lub znak podkreślenia.
- `\W` — zastępuje dowolny znak niebędący znakiem występującym w słowach, czyli jest przeciwieństwem znaku `\w` (jak widać, w wyrażeniach regularnych w powłoce PowerShell wielkość liter ma znaczenie), z czego wynika, że pozwala na wyszukiwanie białych znaków i znaków interpunkcyjnych.

- `\d` — zastępuje dowolną cyfrę od 0 do 9 włącznie.
- `\D` — zastępuje dowolny znak, który nie jest cyfrą.
- `\s` — zastępuje dowolny biały znak włącznie z tabulatorami, spacjami lub znakiem powrotu karetki.
- `\S` — zastępuje dowolny znak, który nie jest białym znakiem.
- `.` (kropka) — zastępuje dowolny pojedynczy znak.
- `[abcde]` — zastępuje dowolny znak z podanego zestawu. Wyrażenie regularne `d[aeiou]n` wyszuka takie ciągi znaków jak *don* i *dan*, ale już nie *doun* ani *deen*.
- `[a-z]` — zastępuje dowolny znak z podanego zakresu. Możesz podać wiele zakresów znaków, oddzielając je od siebie przecinkami, na przykład `[a-f, m-z]`.
- `[^abcde]` — zastępuje dowolny znak spośród tych, których nie ma w podanym zbiorze, co oznacza, że wyrażenie `d[^aeiou]` pasuje do ciągu znaków *dns*, ale nie pasuje do ciągu znaków *don*.
- `?` — umieszczony za innym literałem lub znakiem reprezentuje brak wzorca lub jedno jego wystąpienie. Przykładowo, wyrażenie regularne `do?n` pasuje do ciągów znaków *don* i *dn*, ale nie pasuje do ciągu znaków *doon*.
- `*` — reprezentuje dowolną liczbę wystąpień poprzedzającego go znaku lub wzorca. Wyrażenie `do*n` będzie pasowało zarówno do ciągu znaków *doon*, jak i *don*, ale pasuje również do ciągu znaków *dn*, ponieważ znak `*` reprezentuje także brak wystąpień poprzedzającego wzorca.
- `+` — reprezentuje jedno lub więcej wystąpień poprzedzającego znaku lub wzorca. Przekonasz się, że ten kwantyfikator jest często używany w połączeniu ze wzorcem w nawiasach, tworzącym rodzaj podwyrażenia. Na przykład wyrażenie `(dn)+o` pasuje do ciągu znaków *dndndndno*, ponieważ taki zapis reprezentuje powtarzające się instancje podwyrażenia *dn*.
- `\` (lewy ukośnik) — odgrywa w wyrażeniach regularnych rolę znaku ucieczki. Użyty przed znakiem specjalnym usuwa jego znaczenie i powoduje, że taki znak jest traktowany jak literał. Na przykład wyrażenie `\.` powoduje, że znak kropki jest traktowany jak zwykła kropka, a nie jak znak specjalny zastępujący dowolny inny znak. Aby w wyrażeniu użyć znaku lewego ukośnika, również należy poprzedzić go znakiem ucieczki: `\\.`
- `{n}` — reprezentuje dokładnie *n* wystąpień poprzedzającego znaku lub wzorca. Na przykład wyrażenie `\d{1}` zastępuje dowolną jedną cyfrę, wyrażenie `{2,}` reprezentuje dwa lub więcej wystąpień wzorca, a wyrażenie `{1,3}` zastępuje co najmniej jedno, ale nie więcej niż trzy wystąpienia wzorca.
- `^` — reprezentuje dopasowanie na początku ciągu znaków. Na przykład wyrażenie `d.n` będzie pasowało zarówno do ciągu znaków *don*, jak i *pteranodon*. Ale wyrażenie `^d.n` będzie pasowało do ciągu znaków *don*, ale nie będzie pasowało do ciągu *pteranodon*, ponieważ znak `^` wskazuje, że dopasowanie musi występować na początku ciągu znaków. Jest to inne zastosowanie tego znaku niż w nawiasach kwadratowych, gdzie `^` wskazywał wyłączenie klasy znaków z dopasowania.
- `$` — reprezentuje dopasowanie na końcu ciągu znaków. Na przykład wyrażenie `.icks` będzie pasowało do ciągów znaków *hicks* i *sticks* (w tym drugim przykładzie

nasze wyrażenie zostało dopasowane do ciągu znaków *ticks*), a także będzie pasowało do ciągu znaków *Dickson*. Z kolei wyrażenie `.icks$` nie będzie pasowało do ciągu znaków *Dickson*, ponieważ znak `$` wskazuje, że dopasowanie powinno następować na końcu ciągu znaków.

I to by było na tyle, jeżeli chodzi o podstawową składnię wyrażeń regularnych. Jak już wspominaliśmy wcześniej, takich znaków jest o wiele więcej, ale to, co pokazaliśmy, powinno Ci już w zupełności wystarczyć do wykonywania całkiem złożonych zadań. Spójrzmy na przykładowe wyrażenia regularne:

- `\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}` — reprezentuje wzorec adresu IPv4, chociaż taki wzorec akceptuje zarówno niepoprawne adresy, takie jak *432.567.875.000*, jak i całkowicie prawidłowe adresy IP, takie jak *192.169.15.12*.
- `\\\\w+(\\\\w+)+` — reprezentuje wzorec ścieżki UNC (ang. *Universal Naming Convention*). Użycie znaków lewego ukośnika zarówno w roli znaków ucieczki, jak i znaków ukośnika powoduje, że takie wyrażenie regularne jest dosyć trudne do przeanalizowania. Jest to jeden z powodów, dla których niezmiernie ważną sprawą jest odpowiednie przetestowanie i modyfikowanie wyrażeń regularnych przed użyciem ich w zadaniu produkcyjnym.
- `\w{1}\. \w+@company\.com` — reprezentuje wzorec adresu e-mail składającego się z pierwszej litery imienia, kropki i nazwiska pracownika oraz ciągu znaków *@company.com*. Przedstawiony wzorec pozwala na przykład na znalezienie następującego adresu e-mail: *d.jones@company.com*. Korzystając z takich wzorców, musisz zachować pewną ostrożność, ponieważ nasz wzorec zostanie odnaleziony również w następującym adresie e-mail: *donald.jones@company.com.org*. Nasze przykładowe wyrażenie regularne, reprezentujące wzorec adresu, nie ma problemów z dodatkowym tekstem występującym przed dopasowaną częścią i (lub) po niej. W takich sytuacjach z pomocą mogą nam przyjść omawiane nieco wcześniej znaki specjalne `^` i `$`.

UWAGA Więcej szczegółowych informacji na temat podstawowej składni wyrażeń regularnych możesz znaleźć w systemie pomocy po uruchomieniu z poziomu powłoki PowerShell polecenia `help about_regular_expressions`. Pod koniec tego rozdziału powiemy kilka słów na temat dodatkowych źródeł informacji o wyrażeniach regularnych, które to źródła będziesz mógł eksplorować samodzielnie.

24.3. Używanie wyrażeń regularnych z operatorem *-Match*

W powłoce PowerShell dostępny jest operator porównania `-Match` i jego kuzyn wrażliwy na wielkość liter, `-CMatch`, działające z wyrażeniami regularnymi. Oto kilka przykładów zastosowania tego pierwszego operatora:

```
PS C:\> "don" -match "d[aeiou]n"
True
PS C:\> "dooon" -match "d[aeiou]n"
False
```

```
PS C:\> "dooon" -match "d[aeiou]+n"
True
PS C:\> "djinn" -match "d[aeiou]+n"
False
PS C:\> "dean" -match "d[aeiou]n"
False
```

Choć operator `-Match` posiada wiele zastosowań, przede wszystkim będziemy używać go do testowania wyrażeń regularnych i sprawdzania, czy działają poprawnie. Jak widać, jego lewy operand jest dowolnym testowanym ciągiem znaków, a prawy operand jest wyrażeniem regularnym. Jeżeli dopasowanie zostanie znalezione, wynikiem działania takiego polecenia jest wartość `True`; jeżeli nie, wynikiem działania jest `False`.

ZRÓB TO SAM To dobry moment na to, abyś zrobił sobie małą przerwę w czytaniu i spróbował samodzielnie użyć operatora `-Match`. Spróbuj wykonać omawiane wcześniej przykłady, tak aby nabrać wprawy w używaniu operatora `-Match` w powłoce PowerShell.

24.4. Używanie wyrażeń regularnych z poleceniem *Select-String*

Teraz docieramy do prawdziwego celu tego rozdziału. W omawianych dalej przykładach będziemy używać wybranych plików dzienników serwera IIS, ponieważ są one dokładnie takim typem plików tekstowych, do których przetwarzania przeznaczone są wyrażenia regularne. Oczywiście byłoby miło, gdybyśmy mogli odczytać takie pliki dzienników w powłoce PowerShell w sposób bardziej obiektowy, ale... no cóż, nie możemy, więc pozostają nam wyrażenia regularne.

Zacniemy od skanowania plików dziennika pod kątem występowania w nich komunikatów o błędach `40x`, które często zawierają informacje o nieodnalezionych plikach i innych podobnych błędach. Na podstawie tych informacji chcemy dla naszych deweloperów witryny internetowej wygenerować raport o brakujących plikach. W plikach dziennika każde żądanie HTTP jest zapisywane w osobnym wierszu, a każdy wiersz jest podzielony na pola rozdzielane spacjami. Niektóre pliki mają w nazwach ciąg znaków `401` (na przykład `error401.html`) i takich plików nie chcemy uwzględniać w naszych wynikach. Do wykonania zadania użyjemy następującego wyrażenia regularnego: `\s40[0-9]\s`, ponieważ definiuje ono biały znak po obu stronach kodu błędu `40x`. Użyte wyrażenie powinno być w stanie znaleźć wszystkie błędy o kodach od `400` do `409` włącznie. Oto nasze polecenie:

```
PS C:\logfiles> get-childitem -filter *.log -recurse |
➤ select-string -pattern "\s40[0-9]\s" |
➤ format-table Filename,LineNumber,Line -wrap
```

Zauważ, że aby wykonać to polecenie, zmieniliśmy bieżący katalog roboczy na `C:\LogFiles`. Zaczynamy od nakazania powłoce PowerShell, aby pobrała wszystkie pliki pasujące do wzorca `*.log` znajdujące się w bieżącym katalogu i wszystkich jego podkatalogach

(parametr `-recurse`). Gwarantuje to, że wszystkie pliki dziennika zostaną uwzględnione w danych wyjściowych. Następnie używamy polecenia `Select-String` i jako wzorzec wyszukiwania stosujemy nasze wyrażenie regularne. Wynikiem działania polecenia jest obiekt `MatchInfo`, zatem używamy polecenia `Format-Table` do wyświetlenia wyników zawierających nazwę pliku oraz numer i zawartość wiersza tekstu, w którym znaleziony został ciąg znaków pasujący do wzorca. Tak sformatowane wyniki można łatwo przekierować do pliku i przekazać naszym deweloperom.

Następnie chcielibyśmy przeskanować pliki dzienników pod kątem plików, które były przeglądane za pośrednictwem przeglądarek internetowych opartych na silniku Gecko. Nasi deweloperzy wspominali nam, że niektórzy użytkownicy mają problemy z dostępem do witryny za pomocą tych przeglądarek, i chcieli zobaczyć, które pliki są najczęściej pobierane i przeglądane. Deweloperzy twierdzą, że zawężili problem do przeglądarek działających pod kontrolą systemu Windows NT 6.2, co oznacza, że szukamy ciągów *User-Agent*, które wyglądają tak:

```
(Windows+NT+6.2;+WOW64;+rv:11.0)+Gecko
```

Nasi deweloperzy podkreślali, że problem nie jest specyficzny dla systemów 64-bitowych, więc nie chcą, aby nasze wyniki analizy dzienników zdarzeń były ograniczane tylko do ciągów *User-Agent* zawierających frazę *WOW64*. Po krótkim zastanowieniu utworzyliśmy zatem następujące wyrażenie regularne: `6\\.2;[\\w\\W]+\\+Gecko`. Przyjrzyjmy mu się nieco bliżej:

- `6\\.2;` — to wyrażenie reprezentuje ciąg znaków `6.2;`. Zauważ, że znak kropki został poprzedzony znakiem ucieczki, tak aby był traktowany jako literal, a nie jednoznakowy symbol wieloznaczny (którym zwykle jest kropka).
- `[\\w\\W]+` — takie wyrażenie reprezentuje jeden lub więcej znaków występujących lub niewystępujących w słowach. Inaczej mówiąc, taki zapis reprezentuje dowolny ciąg znaków.
- `\\+Gecko` — to wyrażenie składa się z literału `+` oraz ciągu znaków *Gecko*.

A tak wygląda pełne polecenie, którego zadaniem jest znalezienie w dziennikach zdarzeń wierszy pasujących do podanego wzorca, oraz pierwszych kilka wierszy będących wynikami działania tego polecenia:

```
PS C:\\logfiles> get-childitem -filter *.log -recurse |  
➔ select-string -pattern "6\\.2;[\\w\\W]+\\+Gecko"
```

```
W3SVC1\\u_ex120420.log:14:2012-04-20 21:45:04 10.211.55.30 GET  
  /MyApp1/Testpage.asp - 80 - 10.211.55.29  
  Mozilla/5.0+(Windows+NT+6.2;+WOW64;+rv:11.0)+Gecko/20100101+Firefox/11.0  
  200 0 0 1125  
W3SVC1\\u_ex120420.log:15:2012-04-20 21:45:04 10.211.55.30 GET /TestPage.asp -  
  80 - 10.211.55.29  
  Mozilla/5.0+(Windows+NT+6.2;+WOW64;+rv:11.0)+Gecko/20100101+Firefox/11.0  
  200 0 0 1 109
```

Tym razem zamiast przysyłać wyniki działania do polecenia formatującego, pozostawiliśmy je w domyślnym formacie.

W naszym finalnym przykładzie przejdziemy od plików dziennika serwera IIS do dziennika zdarzeń Windows Security. Wpisy dziennika zdarzeń posiadają właściwość `Message`, która zawiera szczegółowe informacje o zdarzeniu. Informacje te są niestety sformatowane w taki sposób, aby można było je łatwo przeczytać, co nie idzie w parze z łatwością zautomatyzowanej analizy przez komputer. Chcielibyśmy wyszukać wszystkie zdarzenia z identyfikatorem ID 4624, które wskazują logowanie na konto (numer zdarzenia może się różnić w innych wersjach systemu Windows; nasz przykład pochodzi z systemu Windows Server 2008 R2). Chcemy zobaczyć tylko zdarzenia związane z logowaniem się kont o nazwach zaczynających się od ciągu znaków `WIN`, dotyczących kont komputerów w naszej domenie i takich, których nazwy kończą się ciągami znaków od `TM20$` do `TM40$`. Odpowiednie wyrażenie regularne może wyglądać na przykład tak: `WIN[\\W\\w]+TM[234][0-9]\\$`. Zwróć uwagę, że przed ostatnim znakiem `$` musimy użyć znaku ucieczki, żeby znak dolara został potraktowany jako literal, a nie znak specjalny wskazujący dopasowanie na końcu ciągu znaków. W wyrażeniu musimy uwzględnić wzorzec reprezentujący dowolne znaki `[\\W\\w]`, ponieważ w nazwach kont może znajdować się myślnik, który nie pasuje do klasy reprezentowanej tylko przez znak specjalny `\\w`. Oto nasze polecenie:

```
PS C:\> get-eventlog -LogName security | where { $_.eventid -eq 4624 } |  
➔ select -ExpandProperty message | select-string -pattern  
➔ "WIN[\\W\\w]+TM[234][0-9]\\$"
```

Analizę plików dziennika zaczynamy od użycia polecenia `Where-Object`, które przepuszcza tylko zdarzenia o identyfikatorze ID 4624. Następnie pobieramy zawartość właściwości `Message` w postaci zwykłego ciągu znaków i przekazujemy do polecenia `Select-String`. Zwróć uwagę, że takie rozwiązanie spowoduje wyświetlenie tylko zawartości właściwości `Message` zdarzeń pasujących do podanego wzorca; gdyby naszym celem było wyświetlenie całych pasujących rekordów zdarzeń, postępowalibyśmy nieco inaczej:

```
PS C:\> get-eventlog -LogName security | where { $_.eventid -eq 4624 -and  
➔ $_.message -match "WIN[\\W\\w]+TM[234][0-9]\\$" }
```

W tym przypadku zamiast wyświetlać zawartość właściwości `Message`, po prostu szukamy rekordów, w których właściwość `Message` zawiera tekst pasujący do naszego wyrażenia regularnego, a następnie wysyłamy na wyjście całe obiekty zdarzeń. Sposób postępowania zależy jak zawsze od tego, jaki rezultat chcemy osiągnąć.

24.5. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Nie ulega wątpliwości, że wyrażenia regularne to bardzo trudne zagadnienia, które mogą być naprawdę kłopotliwe, zatem nie próbuj na siłę od razu tworzyć rozbudowanych,

skomplikowanych wzorców — zacznij od czegoś prostszego. Poniżej zamieszczamy kilka propozycji ćwiczeń, które powinny Ci to ułatwić. Użyj wyrażeń regularnych i operatorów, aby wykonać następujące zadania:

1. Pobierz i wyświetl listę wszystkich plików znajdujących się w katalogu *Windows*, w których nazwach występują dwucyfrowe numery.
2. Znajdź wszystkie procesy działające na Twoim komputerze, których autorem jest firma Microsoft, i wyświetl identyfikatory tych procesów, ich nazwy oraz nazwę firmy. Wskazówka: aby odszukać nazwy odpowiednich właściwości, powinieneś przekazać za pomocą potoku wyniki działania polecenia *Get-Process* do polecenia *Get-Member*.
3. Z dziennika usługi Windows Update, zwykle znajdującego się w katalogu *C:\Windows*, wyświetl tylko wiersze zawierające informację o tym, że agent zaczął instalować pliki. Być może będziesz musiał wcześniej otworzyć plik dziennika w Notatniku, aby dowiedzieć się, jakie ciągi znaków powinieneś wybrać.
4. Za pomocą polecenia *Get-DNSClientCache* wyświetl wszystkie rekordy, w których wartością właściwości *Data* jest adres IPv4.

24.6. Co dalej?

Wyrażenia regularne są używane w wielu innych miejscach powłoki PowerShell, a w licznych zastosowaniach wykorzystuje się funkcje i mechanizmy powłoki, których nie omawiamy w tej książce. Oto kilka przykładów:

- Instrukcja *Switch* używana w skryptach powłoki posiada parametr, który pozwala porównać daną wartość z jednym lub kilkoma wyrażeniami regularnymi.
- Zaawansowane skrypty i funkcje powłoki PowerShell mogą wykorzystywać mechanizmy używające wyrażeń regularnych do sprawdzania poprawności danych wprowadzanych przez użytkownika, co pozwala na zminimalizowanie ryzyka pobierania nieprawidłowych wartości parametrów wejściowych.
- Operator *-Match* (omówiony pokrótce w tym rozdziale) pozwala na sprawdzanie dopasowania ciągów znaków do wyrażenia regularnego oraz automatycznie pobiera pasujące ciągi znaków i umieszcza je w kolekcji *\$matches* (nie wspominaliśmy o tym wcześniej).

Powłoka PowerShell wykorzystuje standardową składnię wyrażeń regularnych, a jeżeli chcesz dowiedzieć się czegoś więcej na ten temat, zalecamy lekturę znakomitej książki Jeffreya E. F. Friedla, zatytułowanej *Mastering Regular Expressions* (wyd. O'Reilly, 2006). Oczywiście istnieją tysiące innych książek poświęconych zagadnieniom zastosowania wyrażeń regularnych. Wiele z nich jest przeznaczonych dla systemu Windows i środowiska .NET (a więc i powłoki PowerShell), inne koncentrują się na zastosowaniu wyrażeń regularnych w konkretnych sytuacjach i tak dalej. Zajrzyj do swojej ulubionej księgarni internetowej i sprawdź, czy znajdziesz tam odpowiednie książki omawiające zagadnienia dostosowane do Twoich potrzeb.

Bardzo często korzystamy również z bezpłatnego internetowego repozytorium wyrażeń regularnych, <http://RegExLib.com>, w którym można znaleźć wiele przykładów wyrażeń regularnych przeznaczonych do realizacji różnych zadań (wyszukiwania numerów telefonów, adresów e-mail, adresów IP i wielu innych). Często zaglądamy również na stronę <http://RegExTester.com>, która pozwala interaktywnie testować wyrażenia regularne i sprawdzać, czy działają dokładnie w taki sposób, jakiego potrzebujesz.

24.7. Odpowiedzi

1. `dir c:\windows | where {$_name -match "\d{2}"}`
2. `get-process | where {$_company -match "^Microsoft"} |
Select Name,ID,Company`
3. `get-content C:\Windows\WindowsUpdate.log |
Select-string "Start[\w+\W+]+Agent: Installing Updates"`
4. Możesz rozpocząć od zastosowania wzorca, w którym na początku znajduje się od jednej do trzech cyfr, po których następuje znak kropki (literal), na przykład:
`get-dnsclientcache | where { $_.data -match "^\\d{1,3}\\." }`
Zamiast tego możesz oczywiście użyć wyrażenia wyszukującego pełny adres IPv4:
`get-dnsclientcache | where { $_.data -match
"^\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}" }`

Różne wskazówki, techniki, sztuczki i chwytty

Nasze lunchowe spotkania z powłoką PowerShell powoli dobiegają końca, więc chcielibyśmy podzielić się z Tobą jeszcze kilkoma dodatkowymi poradami i technikami, które pozwolą Ci poszerzyć zdobytą do tej pory wiedzę.

25.1. Profile, odpowiedzi i kolory — dostosowywanie powłoki

Każda sesja powłoki PowerShell rozpoczyna się tak samo — otrzymujesz do dyspozycji ciągle te same aliasy, dyski PSD, kolory i tak dalej. Dlaczego jednak nie miałbyś nieco bardziej dostosować powłoki PowerShell do własnych potrzeb?

25.1.1. Profile powłoki PowerShell

Wspominaliśmy wcześniej, że istnieje różnica między aplikacją hosta powłoki PowerShell a samym silnikiem powłoki PowerShell. Aplikacja hosta, taka jak konsola tekstowa lub środowisko PowerShell ISE, jest mechanizmem pozwalającym na wysyłanie poleceń do silnika PowerShell. Silnik wykonuje polecenia, a aplikacja hosta odpowiada za wyświetlanie wyników ich działania. Inną rzeczą, za którą odpowiedzialna jest aplikacja hosta, jest ładowanie i uruchamianie **skryptów profilu** (ang. *profile scripts*) za każdym razem, gdy powłoka rozpoczyna działanie.

Skrypty profilu można wykorzystać do dostosowania środowiska powłoki PowerShell poprzez ładowanie wybranych wcześniej przystawek lub modułów, zmianę domyślnego katalogu roboczego na inny, zdefiniowanie funkcji, których chcesz używać, i tak dalej. Poniżej przedstawiamy przykładowy skrypt profilu, którego Don używa na swoim komputerze:

```
Import-Module ActiveDirectory
Add-PSSnapin SqlServerCmdletSnapin100
cd c:\
```

Skrypt profilu ładuje dwa rozszerzenia powłoki, których Don używa najczęściej, i zmienia domyślny katalog roboczy na katalog główny dysku C:, gdzie Don po prostu lubi pracować. Oczywiście w swoim skrypcie profilu możesz umieścić dowolne polecenia.

UWAGA Możesz pomyśleć, że nie ma potrzeby jawnego ładowania modułu `ActiveDirectory`, ponieważ powłoka `PowerShell` załaduje go automatycznie, gdy tylko użytkownik spróbuje użyć jakiegoś polecenia z tego modułu. Ale dzięki wcześniejszemu załadowaniu go poprzez skrypt profilu moduł ten mapuje dysk `PSDrive` o nazwie `AD:`, a Don lubi go mieć pod ręką od razu po uruchomieniu powłoki.

Na dzień dobry użytkownik nie posiada żadnego profilu domyślnego, a ewentualna wartość utworzonego skryptu profilu zależy wyłącznie od indywidualnych potrzeb użytkownika. Więcej szczegółów znajdziesz w pliku pomocy po uruchomieniu polecenia `help about_profiles`, ale musisz przede wszystkim zastanowić się, czy będziesz pracować w wielu różnych aplikacjach hosta. Na przykład w naszym przypadku mamy tendencję do przełączania się pomiędzy standardową konsolą tekstową a środowiskiem `PowerShell ISE` i lubimy mieć ten sam profil dla obu aplikacji, więc musimy uważać, aby utworzyć właściwy plik skryptu profilu we właściwej lokalizacji. Musimy także uważać na to, co się dzieje w tym profilu, ponieważ używamy go zarówno dla konsoli tekstowej, jak i dla środowiska `ISE`, a niektóre polecenia modyfikujące ustawienia specyficzne dla konsoli, takie jak kolory, mogą powodować problemy lub błędy podczas uruchamiania w `ISE`.

Oto lista plików oraz kolejność, w jakiej host konsoli próbuje je załadować:

1. `$pshome\profile.ps1` — ten skrypt będzie wykonywany dla wszystkich użytkowników komputera bez względu na hosta, którego używają (pamiętaj, że `$pshome` jest predefiniowaną zmienną powłoki `PowerShell` i zawiera ścieżkę do folderu instalacyjnego `PowerShell`).
2. `$pshome\Microsoft.PowerShell_profile.ps1` — ten skrypt będzie wykonywany dla wszystkich użytkowników komputera korzystających z hosta konsoli tekstowej. Jeżeli używają środowiska `PowerShell ISE`, zamiast tego zostanie uruchomiony skrypt `$pshome\Microsoft.PowerShellISE_profile.ps1`.
3. `$home\Documents\WindowsPowerShell\profile.ps1` — skrypt będzie wykonywany tylko dla bieżącego użytkownika (ponieważ znajduje się w jego katalogu domowym) bez względu na hosta, z którego korzysta ten użytkownik.
4. `$home\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1` — ten skrypt będzie wykonywany dla bieżącego użytkownika używającego hosta konsoli. Jeżeli korzysta ze środowiska `PowerShell ISE`, zamiast tego zostanie uruchomiony skrypt `$home\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1`.

Jeżeli jeden lub więcej z tych skryptów nie istnieje, nie ma żadnego problemu. Aplikacja hosta po prostu pominie nieistniejący skrypt i przejdzie do następnego.

W systemach 64-bitowych istnieją 32- i 64-bitowe odmiany tych skryptów, ponieważ są oddzielne 32- i 64-bitowe wersje samej powłoki PowerShell. Nie zawsze będziesz potrzebować tych samych poleceń w powłoce 64-bitowej co w powłoce 32-bitowej. Rozumiemy przez to, że niektóre moduły i inne rozszerzenia są dostępne tylko dla jednej lub drugiej architektury, więc z pewnością nie będziemy chcieli, aby 32-bitowy profil powłoki próbował załadować 64-bitowy moduł do 32-bitowej wersji powłoki, ponieważ z pewnością nie będzie to działać.

Zauważ, że dokumentacja zawarta w pliku `about_profiles` różni się od tego, co tutaj wymieniliśmy, ale nasze doświadczenie pokazuje, że powyższa lista jest poprawna. Oto kilka dodatkowych punktów dotyczących tej listy:

- Zmienna `$pshome` jest wbudowaną zmienną powłoki PowerShell, która zawiera ścieżkę do folderu instalacyjnego tej powłoki; w większości systemów będzie to katalog `C:\Windows\System32\Windows-PowerShell\v1.0` (dla 64-bitowej wersji powłoki w 64-bitowym systemie operacyjnym).
- Zmienna `$home` to kolejna wbudowana zmienna powłoki, wskazująca folder profilu bieżącego użytkownika (na przykład `C:\Users\Administrator`).
- Aby odwołać się do folderu dokumentów, używamy katalogu o nazwie *Documents* (*Dokumenty*), ale pamiętaj, że w niektórych wersjach systemu Windows będzie to folder *My Documents* (*Moje Dokumenty*).
- Napisaaliśmy „bez względu na hosta, którego używają”, ale to — technicznie rzecz biorąc — nie jest do końca prawda. Dotyczy to tylko aplikacji hosta (konsoli tekstowej i środowiska ISE) napisanych przez firmę Microsoft, ale nie ma żadnego sposobu, aby zmusić autorów aplikacji hosta spoza firmy Microsoft do przestrzegania tych zasad.

Ponieważ w naszym przypadku chcemy załadować te same rozszerzenia powłoki niezależnie od tego, czy używamy hosta konsoli, czy środowiska ISE, zdecydowaliśmy się na dostosowanie skryptu `$home\Documents\WindowsPowerShell\profile.ps1`, ponieważ ten profil jest uruchamiany dla obu aplikacji hosta dostarczonych przez firmę Microsoft.

ZRÓB TO SAM Dlaczego nie miałbyś samodzielnie spróbować utworzyć na własne potrzeby jednego lub więcej skryptów profilu? Nawet jeżeli wszystko, co w nich umieścisz, będzie prostym komunikatem wyświetlanym na ekranie, takim jak `Write "To działa!"`, jest to znakomity sposób na zobaczenie różnych plików w akcji. Pamiętaj, że aby zobaczyć uruchamianie skryptów profilu, musisz zamknąć powłokę (lub środowisko ISE) i ponownie ją otworzyć.

Pamiętaj, że skrypty profilu są skryptami i podlegają aktualnym ustawieniom polityki uruchamiania skryptów na Twoim komputerze. Jeżeli Twoja polityka uruchamiania skryptów jest ustawiona na poziom `Restricted`, skrypt profilu nie będzie działał; jeżeli Twoja polityka jest ustawiona na poziom `AllSigned`, Twój skrypt profilu będzie musiał być podpisany i tak dalej. Więcej szczegółowych informacji na temat ustawień polityki wykonywania skryptów i sposobów ich podpisywania znajdziesz w rozdziale 17.

25.1.2. Dostosowywanie znaku zachęty powłoki

Domyślny znak zachęty powłoki PowerShell (*PS C:\>*), który widziałeś przez większą część tej książki, jest generowany przez wbudowaną funkcję o nazwie `Prompt`. Jeżeli chcesz dostosować wygląd tego znaku zachęty, możesz zastąpić tę funkcję. Definiowanie nowej funkcji `Prompt` jest czymś, co można zrobić w skrypcie profilu, dzięki czemu zmiana będzie obowiązywać za każdym razem, gdy uruchamiasz powłokę.

Oto kod domyślnej funkcji `Prompt`:

```
function prompt
{
    $(if (test-path variable:/PSDebugContext) { '[DBG]: ' }
    else { '' }) + 'PS ' + $(Get-Location) `
    + $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
}
```

Funkcja najpierw sprawdza, czy na dysku *VARIABLE:* powłoki jest zdefiniowana zmienna `$DebugContext`. Jeżeli tak, na początku znaku zachęty funkcja dodaje ciąg znaków *[DBG]:*. Jeżeli nie, domyślny znak zachęty przyjmuje postać ciągu znaków *PS* wraz z bieżącą lokalizacją, która jest zwracana przez polecenie `Get-Location`. Jeżeli powłoka znajduje się w połączeniu zagnieżdżonym, zgodnie z definicją wbudowanej zmiennej `$nestedpromptlevel`, do znaku zachęty zostaje dodany ciąg znaków *>>*.

A oto alternatywny kod funkcji `Prompt`. Możesz umieścić go bezpośrednio w dowolnym skrypcie profilu, tak aby nowy znak zachęty był wyświetlany we wszystkich sesjach powłoki:

```
function prompt {
    $time = (Get-Date).ToShortTimeString()
    "$time [ $env:COMPUTERNAME ]:> "
}
```

Ten alternatywny znak zachęty wyświetla bieżący czas, a następnie bieżącą nazwę komputera (umieszczoną w nawiasach kwadratowych).

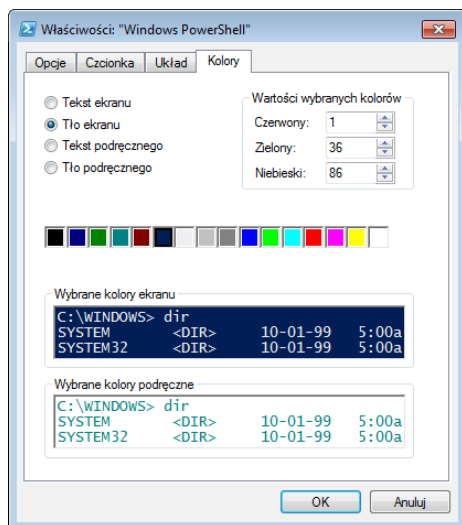
```
6:07 [CLIENT01]:>
```

Zauważ, że nowa funkcja używa znanej Ci już sztuczki z umieszczaniem zmiennych w cudzysłowie, dzięki czemu powłoka zastępuje nazwy zmiennych (takie jak `$time`) ich wartościami.

25.1.3. Modyfikowanie ustawień kolorów

W poprzednich rozdziałach wspominaliśmy o tym, jak można się łatwo zestresować, kiedy na ekranie powłoki przewija się długa seria komunikatów o błędach. Don zawsze miał problemy na lekcjach języka angielskiego, kiedy był dzieckiem, i do tej pory, kiedy widzi taki czerwony tekst, przed oczyma pojawiają mu się jego eseje, które otrzymywał od pani Hannes gęsto oznakowane czerwonym flamastrem. Brrr. Na szczęście powłoka PowerShell pozwala modyfikować większość domyślnych ustawień kolorów, których używa.

Domyślny kolor tekstu na pierwszym planie i kolory tła można modyfikować. Należy kliknąć ikonę w lewym górnym rogu okna programu PowerShell i wybrać z menu podręcznego polecenie *Właściwości*, a następnie, po pojawieniu się na ekranie okna dialogowego *Właściwości: "Windows PowerShell"*, przejść na kartę *Kolory*, która została pokazana na rysunku 25.1.



Rysunek 25.1. Konfigurowanie domyślnych kolorów ekranu powłoki

Modyfikowanie kolorów tekstu błędów, ostrzeżeń i innych wiadomości jest nieco trudniejsze i wymaga uruchomienia odpowiedniego polecenia. Ale możesz umieścić to polecenie w swoim profilu, żeby było uruchamiane za każdym razem, gdy uruchamiasz powłokę. Oto jak zmienić kolor tekstu komunikatu o błędzie na zielony, co według nas jest dużo bardziej kojące:

```
(Get-Host).PrivateData.ErrorForegroundColor = "green"
```

Możesz zmienić kolory dla następujących ustawień:

- `ErrorForegroundColor`,
- `ErrorBackgroundColor`,
- `WarningForegroundColor`,
- `WarningBackgroundColor`,
- `DebugForegroundColor`,
- `DebugBackgroundColor`,
- `VerboseForegroundColor`,
- `VerboseBackgroundColor`,
- `ProgressForegroundColor`,
- `ProgressBackgroundColor`.

Możesz wybierać spośród między innymi takich kolorów:

- Red,
- Yellow,
- Black,
- White,
- Green,
- Cyan,
- Magenta,
- Blue.

Istnieją również ciemne wersje większości z tych kolorów: DarkRed, DarkYellow, DarkGreen, DarkCyan, DarkBlue i tak dalej.

25.2. Operatory: **-as**, **-is**, **-replace**, **-join**, **-split**, **-in**, **-contains**

Takie dodatkowe operatory są przydatne w wielu różnych sytuacjach i pozwalają pracować z typami danych, kolekcjami i ciągami.

25.2.1. Operatory **-as** i **-is**

Operator **-as** tworzy nowy obiekt, dokonując próby przekonwertowania istniejącego obiektu na inny typ. Przykładowo, jeżeli masz liczbę zawierającą część dziesiętną (będącą na przykład wynikiem operacji dzielenia), możesz odrzucić część dziesiętną, dokonując konwersji, czy inaczej mówiąc, **rzutowania** (ang. *casting*) tej liczby na typ `integer`, reprezentujący liczby całkowite:

```
1000 / 3 -as [int]
```

W wierszu polecenia najpierw podajemy obiekt, który chcemy przekonwertować, następnie operator **-as** i na końcu w nawiasach kwadratowych typ, na który chcemy dokonać konwersji. Możemy używać następujących typów: `[string]`, `[xml]`, `[int]`, `[single]`, `[double]`, `[datetime]` i innych, choć najczęściej będziemy zapewne korzystać z wymienionych tutaj typów. Warto zauważyć, że z technicznego punktu widzenia konwersja na liczbę całkowitą przedstawiona w naszym przykładzie spowoduje zaokrąglenie liczby ułamkowej do odpowiedniej liczby całkowitej, a nie tylko obcięcie ułamkowej części liczby wymiernej.

Operator **-is** działa podobnie — został zaprojektowany tak, aby zwracać wartość, odpowiednio, `True` lub `False`, jeżeli obiekt jest danego typu lub nie jest. Oto kilka przykładów zastosowania tego operatora:

```
123.45 -is [int]
"SERVER-R2" -is [string]
$True -is [bool]
(Get-Date) -is [datetime]
```

ZRÓB TO SAM Spróbuj samodzielnie wykonać przedstawione wyżej polecenia i sprawdź wyniki ich działania.

25.2.2. Operator -replace

Operator -replace jest przeznaczony do lokalizowania wszystkich wystąpień jednego ciągu znaków w drugim i zastępowania tych wystąpień innym ciągiem znaków:

```
PS C:\> "192.168.34.12" -replace "34", "15"
192.168.15.12
```

W wierszu polecenia najpierw podajemy źródłowy łańcuch tekstu i operator -replace. Następnie podajemy poszukiwany ciąg znaków, a potem przecinek i ciąg znaków, na który poszukiwany ciąg znaków powinien zostać zamieniony. W przykładzie pokazanym powyżej 34 zastępujemy 15.

25.2.3. Operatory -join i -split

Operatory -join i -split są przeznaczone do konwertowania tablic na listy elementów oddzielonych od siebie separatorami i odwrotnie.

Załóżmy na przykład, że tworzymy tablicę składającą się z pięciu elementów:

```
PS C:\> $array = "jeden", "dwa", "trzy", "cztery", "pięć"
PS C:\> $array
jeden
dwa
trzy
cztery
pięć
```

Takie rozwiązanie działa, ponieważ powłoka PowerShell automatycznie traktuje listę elementów oddzielonych od siebie przecinkami jako tablicę. Teraz założmy, że chcesz połączyć poszczególne elementy tej tablicy w jeden ciąg znaków, w którym poszczególne elementy są od siebie oddzielone znakiem potoku. Możesz to zrobić za pomocą operatora -join:

```
PS C:\> $array -join "|"
jeden|dwa|trzy|cztery|pięć
```

Zapisanie wyniku działania tego polecenia w zmiennej pozwoli Ci ponownie go użyć lub nawet zapisać w pliku:

```
PS C:\> $string = $array -join "|"
PS C:\> $string
jeden|dwa|trzy|cztery|pięć
PS C:\> $string | out-file data.dat
```

Operator -split działa w odwrotny sposób — pobiera ciąg znaków zawierający listę elementów oddzielonych od siebie znakiem separatora i tworzy z niej tablicę. Załóżmy na przykład, że mamy plik rozdzielany tabulatorami, zawierający jeden wiersz z czterema elementami. Wyświetlanie zawartości pliku może wyglądać tak:

```
PS C:\> gc computers.tdf
Server1 Windows East    Managed
```

Pamiętaj, że `gc` to alias polecenia `Get-Content`.

Teraz możemy użyć operatora `-split`, aby podzielić ten wiersz na cztery pojedyncze elementy tablicy:

```
PS C:\> $array = (gc computers.tdf) -split "`t"
PS C:\> $array
Server1
Windows
East
Managed
```

Zwróć uwagę na użycie znaku ucieczki (lewy apostrof) i litery `t` (``t`) do zdefiniowania znaku tabulacji. Aby znak ucieczki został poprawnie rozpoznany, całe wyrażenie musi być umieszczone w cudzysłowie.

Tablica będąca wynikiem działania tego polecenia ma cztery elementy, do których możemy uzyskać dostęp indywidualnie, używając ich numerów indeksu:

```
PS C:\> $array[0]
Server1
```

25.2.4. Operatory `-contains` i `-in`

Operator `-contains` bywa źródłem wielu nieporozumień, szczególnie dla początkujących użytkowników powłoki PowerShell. Niedoświadczeni użytkownicy bardzo często próbują używać go w następujący sposób:

```
PS C:\> 'this' -contains '*his*'
False
```

W rzeczywistości zamiast operatora `-contains` w takiej sytuacji powinien zostać użyty operator `-like`:

```
PS C:\> 'this' -like '*his*'
True
```

Operator `-like` jest przeznaczony do porównywania łańcuchów znaków z użyciem symboli wieloznacznych, natomiast operator `-contains` służy do sprawdzenia, czy dany obiekt istnieje w określonej kolekcji obiektów. Aby to zilustrować, utworzymy przykładową kolekcję obiektów typu `String`, a następnie sprawdzimy, czy dany ciąg znaków znajduje się w tej kolekcji:

```
PS C:\> $collection = 'abc','def','ghi','jkl'
PS C:\> $collection -contains 'abc'
True
PS C:\> $collection -contains 'xyz'
False
```

Operator `-in` robi to samo, ale odwraca kolejność operandów tak, że kolekcja obiektów znajduje się po prawej stronie operatora, a testowany obiekt — po jego lewej stronie:

```
PS C:\> $collection = 'abc','def','ghi','jkl'
PS C:\> 'abc' -in $collection
```

```
True
PS C:\> 'xyz' -in $collection
False
```

25.3. Operowanie na ciągach znaków

Załóżmy, że mamy ciąg tekstu i musimy przekonwertować wszystkie jego znaki na wielkie litery. Albo musimy wyświetlić lub pobrać trzy ostatnie znaki tego ciągu. Jak możemy to zrobić?

W powłoce PowerShell ciągi są obiektami i posiadają wiele metod. Zapewne pamiętasz, że metoda jest sposobem poinformowania obiektu, aby wykonał jakąś operację, zwykle dotyczącą samego siebie, i że dostępne metody obiektu możesz odkryć, przekazując taki obiekt za pomocą potoku do polecenia `Gm`:

```
PS C:\> "Hello" | gm
TypeName: System.String
Name      MemberType      Definition
-----
Clone      Method           System.Object Clone()
CompareTo   Method           int CompareTo(System.Object value...)
Contains    Method           bool Contains(string value)
CopyTo      Method           System.Void CopyTo(int sourceInde...
EndsWith    Method           bool EndsWith(string value), bool...
Equals      Method           bool Equals(System.Object obj), b...
GetEnumerator Method           System.CharEnumerator GetEnumerat...
GetHashCode Method           int GetHashCode()
GetType     Method           type GetType()
GetTypeCode Method           System.TypeCode GetTypeCode()
IndexOf     Method           int IndexOf(char value), int Inde...
IndexOfAny  Method           int IndexOfAny(char[] anyOf), int...
Insert      Method           string Insert(int startIndex, str...
IsNormalized Method           bool IsNormalized(), bool IsNorma...
LastIndexOf Method           int LastIndexOf(char value), int ...
LastIndexOfAny Method          int LastIndexOfAny(char[] anyOf)...
Normalize   Method           string Normalize(), string Normal...
PadLeft     Method           string PadLeft(int totalWidth), s...
PadRight    Method           string PadRight(int totalWidth), ...
Remove      Method           string Remove(int startIndex, int...
Replace     Method           string Replace(char oldChar, char...
Split       Method           string[] Split(Params char[] sepa...
StartsWith  Method           bool StartsWith(string value), bo...
Substring   Method           string Substring(int startIndex)...
ToCharArray Method           char[] ToCharArray(), char[] ToCh...
ToLower     Method           string ToLower(), string ToLower(...
ToLowerInvariant Method          string ToLowerInvariant()
ToString    Method           string ToString(), string ToStrin...
ToUpper     Method           string ToUpper(), string ToUpper(...
ToUpperInvariant Method          string ToUpperInvariant()
Trim        Method           string Trim(Params char[] trimCha...
TrimEnd     Method           string TrimEnd(Params char[] trim...
TrimStart   Method           string TrimStart(Params char[] tr...
Chars       ParameterizedProperty char Chars(int index) {get;}
Length      Property          System.Int32 Length {get;}
```

Poniżej zamieszczamy krótkie zestawienie wybranych, najbardziej użytecznych metod obiektu `String`:

- Metoda `IndexOf()` informuje o położeniu danego znaku w ciągu znaków:

```
PS C:\> "SERVER-R2".IndexOf("-")
6
```
- Metody `Split()`, `Join()` i `Replace()` działają podobnie do operatorów `-split`, `-join` i `-replace`, które opisywaliśmy w poprzedniej sekcji. W naszym przypadku zazwyczaj bardziej wolimy używać operatorów powłoki PowerShell niż metod obiektu `String`.
- Metody `ToLower()` i `ToUpper()` przekształcają ciąg znaków, odpowiednio, na małe i wielkie litery:

```
PS C:\> $computername = "SERVER17"
PS C:\> $computername.ToLower()
server17
```
- Metoda `Trim()` usuwa białe znaki z obu końców łańcucha; metody `TrimStart()` i `TrimEnd()` usuwają białe znaki, odpowiednio, na początku lub na końcu ciągu znaków:

```
PS C:\> $username = "    Don "
PS C:\> $username.Trim()
Don
```

Te i inne metody obiektów `String` są świetnymi sposobami operowania na ciągach znaków. Zauważ, że wszystkie metody takich obiektów mogą być używane zarówno ze zmiennymi zawierającymi ciągi znaków (jak w przykładach zastosowania metod `ToLower()` i `Trim()`), jak i bezpośrednio ze statycznymi ciągami znaków (jak w przykładzie zastosowania metody `IndexOf()`).

25.4. Operowanie na datach

Podobnie jak obiekty typu `String`, obiekty `Date` (lub `DateTime`, jeżeli wolisz) mają wiele metod pozwalających na operowanie na datach i wykonywanie obliczeń daty i czasu:

```
PS C:\> get-date | gm
```

```
TypeName: System.DateTime
```

Name	MemberType	Definition
----	-----	-----
Add	Method	System.DateTime Add(System.TimeSpan ...
AddDays	Method	System.DateTime AddDays(double value)
AddHours	Method	System.DateTime AddHours(double value)
AddMilliseconds	Method	System.DateTime AddMilliseconds(doub...
AddMinutes	Method	System.DateTime AddMinutes(double va...
AddMonths	Method	System.DateTime AddMonths(int months)
AddSeconds	Method	System.DateTime AddSeconds(double va...
AddTicks	Method	System.DateTime AddTicks(long value)
AddYears	Method	System.DateTime AddYears(int value)
CompareTo	Method	int CompareTo(System.Object value), ...
Equals	Method	bool Equals(System.Object value), bo...
GetDateTimeFormats	Method	string[] GetDateTimeFormats(), strin...
GetHashCode	Method	int GetHashCode()

GetType	Method	type GetType()
GetTypeCode	Method	System.TypeCode GetTypeCode()
IsDaylightSavingTime	Method	bool IsDaylightSavingTime()
Subtract	Method	System.TimeSpan Subtract(System.Date...
ToBinary	Method	long ToBinary()
ToFileTime	Method	long ToFileTime()
ToFileTimeUtc	Method	long ToFileTimeUtc()
ToLocalTime	Method	System.DateTime ToLocalTime()
ToLongDateString	Method	string ToLongDateString()
ToLongTimeString	Method	string ToLongTimeString()
ToOADate	Method	double ToOADate()
ToShortDateString	Method	string ToShortDateString()
ToShortTimeString	Method	string ToShortTimeString()
Tostring	Method	string ToString(), string ToString(s...
ToUniversalTime	Method	System.DateTime ToUniversalTime()
DisplayHint	NoteProperty	Microsoft.PowerShell.Commands.Displa...
Date	Property	System.DateTime Date {get;}
Day	Property	System.Int32 Day {get;}
DayOfWeek	Property	System.DayOfWeek DayOfWeek {get;}
DayOfYear	Property	System.Int32 DayOfYear {get;}
Hour	Property	System.Int32 Hour {get;}
Kind	Property	System.DateTimeKind Kind {get;}
Millisecond	Property	System.Int32 Millisecond {get;}
Minute	Property	System.Int32 Minute {get;}
Month	Property	System.Int32 Month {get;}
Second	Property	System.Int32 Second {get;}
Ticks	Property	System.Int64 Ticks {get;}
TimeOfDay	Property	System.TimeSpan TimeOfDay {get;}
Year	Property	System.Int32 Year {get;}
DateTime	ScriptProperty	System.Object DateTime {get;if ((& {...

Pamiętaj, że te właściwości umożliwiają także dostęp do wybranych części obiektu `DateTime`, na przykład do dnia, roku lub miesiąca:

```
PS C:\> (get-date).month
10
```

Metody umożliwiają dwie rzeczy: wykonywanie obliczeń na danych oraz konwersję na inne formaty. Na przykład aby obliczyć datę, która była 90 dni temu, możemy użyć metody `AddDays()` i jako argument wywołania podać odpowiednią liczbę ujemną:

```
PS C:\> $today = get-date
PS C:\> $90daysago = $today.adddays(-90)
PS C:\> $90daysago
Sobota, 24 czerwca 2016 11:26:08
```

Metody, których nazwy zaczynają się ciągiem znaków `To`, mają na celu podawanie dat i czasu w alternatywnym formacie, takim jak skrócony format daty:

```
PS C:\> $90daysago.toshortdatestring()
2016-07-24
```

Wszystkie metody obiektów `Date` wykorzystują bieżące ustawienia regionalne komputera do określenia poprawnego sposobu formatowania daty i czasu.

25.5. Operowanie na danych WMI

Usługa WMI przechowuje informacje o danych i czasie w trudnym do bezpośredniego użycia formacie. Na przykład klasa Win32_OperatingSystem przechowuje informacje o czasie ostatniego uruchomienia komputera, które w natywnym formacie WMI wyglądają tak:

```
PS C:\> get-wmiobject win32_operatingsystem | select lastbootuptime
-----
20101021210207.793534-420
```

Deweloperzy powłoki PowerShell wiedzieli, że nie będzie łatwo korzystać z takich informacji, więc do każdego obiektu WMI dodali kilka metod umożliwiających konwersję znaczników daty i czasu. Jeżeli przekażesz za pomocą potoku dowolny obiekt WMI do polecenia Gm, metody konwersji dat znajdziesz zazwyczaj gdzieś pod koniec listy:

```
PS C:\> get-wmiobject win32_operatingsystem | gm
      TypeName: System.Management.ManagementObject#root\cimv2\Win32_OperatingSystem
Name      MemberType Definition
-----
Reboot     Method      System.Management...
SetDateTime Method      System.Management...
Shutdown   Method      System.Management...
Win32Shutdown Method      System.Management...
Win32ShutdownTracker Method      System.Management...
BootDevice Property     System.String Boo...
...
PSStatus   PropertySet  PSSStatus {Status,...
ConvertFromDateTime ScriptMethod System.Object Con...
ConvertToDateTime ScriptMethod System.Object Con...
```

W powyższym przykładzie celowo pominęliśmy większość wyników działania tego polecenia, tak aby można było łatwo znaleźć metody ConvertFromDateTime() i ConvertToDateTime(). Jeżeli chcesz skorzystać z tych metod i dokonać konwersji znaczników czasu WMI na normalny format daty i czasu, możesz to zrobić w następujący sposób:

```
PS C:\> $os = get-wmiobject win32_operatingsystem
PS C:\> $os.ConvertToDateTime($os.lastbootuptime)
czwartek, 20 października 2015 21:02:07
```

Jeżeli chcesz umieścić takie informacje o dacie i czasie w normalnej tabeli, możesz użyć polecenia Select-Object lub Format-Table do utworzenia niestandardowych kolumn obliczeniowych i właściwości:

```
PS C:\> get-wmiobject win32_operatingsystem | select BuildNumber,__SERVER,
  ↳ @{l='LastBootTime';e={$_.ConvertToDateTime($_.LastBootupTime)}}
BuildNumber      __SERVER      LastBootTime
-----
7600             SERVER-R2     2015-10-20 21:02:07
```

Operacje na danych są mniej kłopotliwe, jeżeli używasz poleceń CIM, ponieważ takie polecenia automatycznie tłumaczą większość znaczników daty i czasu na format znacznie bardziej przyjazny dla użytkownika.

25.6. Ustawianie domyślnych wartości parametrów

Większość poleceń powłoki PowerShell ma co najmniej kilka parametrów, które posiadają wartości domyślne. Na przykład jeżeli uruchomisz polecenie `Dir` bez żadnych parametrów, domyślnie wyświetlona zostanie zawartość bieżącego katalogu roboczego, bez konieczności jawnego określania wartości parametru `-Path`. W powłoce PowerShell v3 można również zdefiniować własne wartości domyślne dla dowolnego parametru dowolnego polecenia — lub nawet dla wielu poleceń. Twoje ustawienia domyślne mają zastosowanie tylko wtedy, gdy polecenie zostanie uruchomione bez określonych parametrów; oczywiście w razie potrzeby zawsze możesz przesłonić swoje ustawienia domyślne, podając nazwy parametrów i ich odpowiednie wartości podczas wywoływania polecenia.

Wartości domyślne są przechowywane w specjalnej wbudowanej zmiennej o nazwie `$PSDefaultParameterValues`. Po każdym uruchomieniu nowego okna powłoki wartość tej zmiennej jest zerowana i może być na nowo wypełniona odpowiednią tablicą asocjacyjną (którą można utworzyć w skrypcie profilu, tak aby zawsze mieć dostęp do własnych domyślnych ustawień parametrów).

Na przykład założmy, że chcesz utworzyć nowy obiekt poświadczeń zawierający nazwę użytkownika i hasło i że to nowe poświadczenie powinno być automatycznie zastosowane do wszystkich poleceń, które posiadają parametr `-Credential`:

```
PS C:\> $credential = Get-Credential -UserName Administrator
➤ -Message "Podaj hasło dostępu administratora systemu: "
PS C:\> $PSDefaultParameterValues.Add('*:Credential',$credential)
```

W innym scenariuszu możesz spowodować, aby polecenie `Invoke-Command` przy każdym wywołaniu prosiło o podanie poświadczeń administratora. W tym przypadku zamiast przypisywać wartość domyślną, musimy do tego parametru przypisać blok skryptu, który będzie wykonywał polecenie `Get-Credential`:

```
PS C:\> $PSDefaultParameterValues.Add('Invoke-Command:Credential',
➤ {Get-Credential -Message 'Podaj hasło dostępu administratora systemu '
➤ -UserName Administrator})
```

Łatwo zauważyć, że podstawowa składnia pierwszego argumentu metody `Add()` ma postać `<-cmdlet>:<parametr>`, gdzie `<cmdlet>` może przyjmować symbole wieloznaczne, takie jak `*`. Drugim argumentem metody `Add()` jest nowa wartość domyślna albo blok skryptu, który będzie wykonywał inne polecenie lub polecenia.

W razie potrzeby możesz zawsze sprawdzić zawartość zmiennej `$PSDefaultParameterValues`:

```
PS C:\> $PSDefaultParameterValues
Name                                     Value
----                                     -
*:Credential                           System.Management.Automation.PSCredenti
Invoke-Command:Credential              Get-Credential -Message 'Podaj hasło do
```

Więcej szczegółowych informacji na temat definiowania nowych wartości domyślnych parametrów poleceń powłoki PowerShell znajdziesz w pliku pomocy `about_parameters_↪default_values`.

Dla zainteresowanych

Zmienne powłoki PowerShell są kontrolowane przez coś, co nazywamy **zasięgiem**. Podstawowe zagadnienia związane z zasięgiem zmiennych zostały pokrótce omówione w rozdziale 21. Pamiętaj, że zasięg determinuje również dostępność nowych domyślnych wartości parametrów poleceń.

Jeżeli ustawisz zmienną `$PSDefaultParameterValues` z poziomu wiersza poleceń, nowe ustawienia będą miały wpływ na wszystkie skrypty i polecenia uruchamiane w tej sesji powłoki. Jeżeli jednak ustawisz zmienną `$PSDefaultParameterValues` w skrypcie, nowe ustawienia będą dotyczyły tylko operacji wykonywanych przez ten skrypt. Jest to bardzo przydatne rozwiązanie, ponieważ oznacza ono, że możesz uruchomić skrypt z wieloma ustawieniami domyślnymi i nie będą one miały zastosowania do innych skryptów lub ogólnie do sposobu funkcjonowania powłoki.

Koncepcja zakładająca, że „to, co się dzieje w skrypcie, pozostaje w skrypcie”, jest kluczowym elementem działania zasięgów. Jeżeli chcesz dowiedzieć się czegoś więcej na ten temat zasięgów, to wiele ciekawych informacji znajdziesz w pliku pomocy `about_scope`.

25.7. Zastosowanie bloków skryptu

Bloki skryptów są kluczowym komponentem powłoki PowerShell i do tej pory używalimy ich już całkiem sporo:

- Parametr `-FilterScript` polecenia `Where-Object` pobiera blok skryptu.
- Parametr `-Process` polecenia `ForEach-Object` pobiera blok skryptu.
- Tablica asocjacyjna używana do tworzenia niestandardowych właściwości za pomocą polecenia `Select-Object` albo niestandardowe kolumny tworzone za pomocą polecenia `Format-Table` pobierają bloki skryptu jako wartość klucza `Expression`.
- Domyślne wartości parametrów, jak opisano w poprzedniej sekcji, mogą być definiowane przy użyciu bloków skryptu.
- Niektóre polecenia wykorzystywane do komunikacji zdalnej i tworzenia zadań, takie jak `Invoke-Command` czy `Start-Job`, mogą pobierać bloki skryptów w parametrze `-ScriptBlock`.

Czym więc jest blok skryptu? Ogólnie rzecz biorąc, jest to wszystko, co zostało ujęte w nawiasy klamrowe `{}`, z wyjątkiem tabel haszujących, które używają nawiasów klamrowych, ale poprzedzone są znakiem `@`. W razie potrzeby blok skryptu możesz nawet wpisać bezpośrednio z poziomu wiersza i przypisać go do zmiennej, a następnie użyć operatora wywołania, `&`, do uruchomienia takiego bloku:

```
PS C:\> $block = { get-process | sort -Property vm -Descending | select -first 10 }
PS C:\> &$block
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----

680	42	14772	13576	1387	3.84	404	svchost
454	26	68368	75116	626	1.28	1912	powershell
396	37	179136	99252	623	8.45	2700	powershell
497	29	15104	6048	615	0.41	2500	SearchIndexer
260	20	4088	8328	356	0.08	3044	taskhost
550	47	16716	13180	344	1.25	1128	svchost
1091	55	19712	35036	311	1.81	3056	explorer
454	31	56660	15216	182	45.94	1596	MsMpEng
163	17	62808	27132	162	0.94	2692	dwm
584	29	7752	8832	159	1.27	892	svchost

Dzięki blokom skryptu możesz zrobić znacznie więcej. Jeżeli chcesz samodzielnie poznać ich możliwości, powinieneś uważnie zapoznać się z zawartością pliku pomocy `about_script_blocks`.

25.8. Więcej wskazówek, trików i technik

Jak powiedzieliśmy na wstępie tego rozdziału, jest to nieco przypadkowy przegląd niektórych małych zagadnień, które chcieliśmy Ci pokazać, a które nie pasowały do żadnego z poprzednich rozdziałów. Oczywiście w miarę jak będziesz nabierał doświadczenia w pracy z powłoką, będziesz poznawał coraz więcej takich przydatnych technik, trików i sztuczek.

Jeżeli Cię to zainteresowało, możesz również zajrzeć na nasze kanały na Twitterze, *@JeffHicks* i *@concentratedDon*, gdzie regularnie dzielimy się wskazówkami i technikami, które odkrywamy i uważamy za przydatne. Strony takie jak *PowerShell.com* oferują również listy mailingowe zawierające wiele ciekawych materiałów i wskazówek. Nie zapomnij o forach na stronie *PowerShell.org*. Bardzo często samodzielne uczenie się krok po kroku jest łatwym sposobem na zdobywanie nowych umiejętności i nabieranie doświadczenia, dlatego powinieneś potraktować te i inne źródła informacji, z którymi się spotykasz, jako doskonały sposób na stopniowe i ciągłe doskonalenie swojej wiedzy na temat powłoki PowerShell.

Korzystanie ze skryptów innych użytkowników

Chociaż mamy nadzieję, że będziesz w stanie konstruować od zera własne polecenia i skrypty powłoki PowerShell, zdajemy sobie sprawę z tego, że będziesz również polegać na wielu źródłach znalezionych w sieci Internet. Niezależnie od tego, czy będziesz kopiował przykłady z czyjegoś bloga, czy modyfikował na własne potrzeby skrypt znaleziony w internetowym repozytorium skryptów, takim jak PowerShell Code Repository (<http://PoshCode.org>), możliwość wykorzystania skryptu PowerShell napisanego i udostępnionego przez kogoś innego jest bardzo ważną umiejętnością. W tym rozdziale prezentujemy Ci proces, którego używamy, aby zrozumieć czyjś skrypt i zaadaptować go do własnych potrzeb.

PODZIĘKOWANIA Specjalne podziękowania należą się Christophowi Tohermesowi i Kai Taylor, którzy udostępnili nam scenariusze do wykorzystania w tym rozdziale. Z rozmysłem poprosiliśmy ich o napisanie skryptów nie-całkiem-doskonałych, niekoniecznie odzwierciedlających najlepsze praktyki programowania, które tak lubimy promować. W niektórych przypadkach nieco „pogorszyliśmy” ich skrypty, żeby ten rozdział lepiej odzwierciedlał rzeczywisty świat internetowych skryptów. Naprawdę doceniamy ich wkład w to ćwiczenie edukacyjne!

Zauważ, że wybraliśmy właśnie te skrypty, ponieważ wykorzystują zaawansowane mechanizmy powłoki PowerShell, których nie omawialiśmy w tej książce. Podobnie jak poprzednio, było to działanie celowe, ponieważ wychodzimy z założenia, że to jest bardzo realistyczny scenariusz — w codziennym życiu będziesz często spotykał się ze skryptami, których przeznaczenie i sposób działania będą dla Ciebie dosyć enigmatyczne, stąd

duża część tego ćwiczenia jest poświęcona temu, jak szybko możesz się dowiedzieć, co robi dany skrypt, nawet jeżeli nie do końca znasz i rozumiesz techniki, z których korzysta.

26.1. Skrypt

Listing 26.1 przedstawia kompletny skrypt o nazwie *New-WebProject.ps1*. Ten skrypt został zaprojektowany do pracy z cmdletami serwera IIS firmy Microsoft, dostępnymi w systemie Windows Server 2008 R2 i nowszych wersjach po zainstalowaniu usługi serwera WWW.

Listing 26.1. New-WebProject.ps1

```
param(
    [parameter(Mandatory = $true)]
    [string] $Path,
    [parameter(Mandatory = $true)]
    [string] $Name
)
$System = [Environment]::GetFolderPath ("System")
$script:hostsPath = ([System.IO.Path]::Combine($System, "drivers\etc\"))+"hosts"
function New-localWebsite([string] $sitePath, [string] $siteName)
{
    try
    {
        Import-Module WebAdministration
    }
    catch
    {
        Write-Host "IIS Powershell module is not installed. Please install it first, by adding
        ↳the feature"
    }
    Write-Host "AppPool is created with name: " $siteName
    New-WebAppPool -Name $siteName
    Set-ItemProperty IIS:\AppPools\$Name managedRuntimeVersion v4.0
    Write-Host
    if(-not (Test-Path $sitePath))
    {
        New-Item -ItemType Directory $sitePath
    }
    $header = "www."+$siteName+".local"
    $value = "127.0.0.1 " + $header
    New-Website -ApplicationPool $siteName -Name $siteName -Port 80 -PhysicalPath $sitePath
    ↳-HostHeader ($header)
    Start-Website -Name $siteName
    if(-not (HostsFileContainsEntry ($header)))
    {
        AddEntryToHosts -hostEntry $value
    }
}
function AddEntryToHosts ([string] $hostEntry)
{
    try
    {
```

```

        $writer = New-Object System.IO.StreamWriter($hostsPath, $true)
        $writer.Write([Environment]::NewLine)
        $writer.Write($hostEntry)
        $writer.Dispose()
    }
    catch [System.Exception]
    {
        Write-Error "An Error occured while writing the hosts file"
    }
}
function HostsFileContainsEntry ([string] $entry)
{
    try
    {
        $reader = New-Object System.IO.StreamReader($hostsPath + "hosts")
        while(-not($reader.EndOfStream))
        {
            $line = $reader.Readline()
            if($line.Contains($entry))
            {
                return $true
            }
        }
        return $false
    }
    catch [System.Exception]
    {
        Write-Error "An Error occured while reading the host file"
    }
}

```

Pierwszą częścią skryptu jest blok parametrów, którego sposób tworzenia omawialiśmy w rozdziale 21.:

```

param(
    [parameter(Mandatory = $true)]
    [string] $Path,
    [parameter(Mandatory = $true)]
    [string] $Name
)

```

Ten blok parametrów wygląda nieco inaczej, ale wydaje się, że definiuje parametry - Path i -Name, z których każdy jest obligatoryjny. Jak dotąd, idzie dobrze. Po uruchomieniu tego skryptu musisz podać wartości obu parametrów.

Następne kilka wierszy jest bardziej tajemnicze:

```

$System = [Environment]::GetFolderPath ("System")
$script:hostsPath = ([System.IO.Path]::Combine($System, "drivers\etc\"))+"hosts"

```

Nie wydaje się, żeby oba polecenia miały robić coś potencjalnie niebezpiecznego — polecenia takie jak `GetFolderPath` nie wzbudzają żadnych podejrzeń. Aby zobaczyć, co robią, uruchamiamy je bezpośrednio z poziomu wiersza poleceń powłoki:

```
PS C:\> $system = [Environment]::GetFolderPath('System')
PS C:\> $system
C:\Windows\system32
PS C:\> $script:hostsPath = ([System.IO.Path]::Combine($system,"drivers\etc \"))+"hosts"
PS C:\> $hostsPath
C:\Windows\system32\drivers\etc\hosts
PS C:\>
```

Polecenie `$script:hostsPath` tworzy nową zmienną, więc będziemy mieć dodatkową zmienną oprócz nowej zmiennej `$system`. Te dwa wiersze kodu ustawiają ścieżkę do folderu i ścieżkę do pliku. Zapamiętaj zawartość tych zmiennych, aby móc odnieść się do nich podczas dalszej analizy skryptu.

Pozostała część skryptu składa się z trzech funkcji: `New-LocalWebsite`, `AddEntryToHosts` i `HostsFileContainsEntry`. Funkcja jest jak skrypt w skrypcie: każda z nich wykonuje określone zadania, które możesz w odpowiednim momencie wywołać. Widać, że każda z funkcji definiuje jeden lub więcej parametrów wejściowych, chociaż nie robią tego w bloku `Param()`. Zamiast tego używają alternatywnej techniki deklaracji parametrów, która jest dozwolona tylko dla funkcji: lista parametrów jest wymieniana w nawiasach zaraz po nazwie funkcji (tak samo jak robiliśmy to w bloku `Param`). To taki rodzaj skrótu.

Jeżeli przeanalizujemy zawartość skryptu, okaże się, że żadna z tych funkcji nie jest wywoływana z poziomu samego skryptu, więc jeżeli miałbyś uruchomić ten skrypt w takiej postaci, jak jest teraz, nic by się nie wydarzyło. Ale wewnątrz kodu funkcji `New-LocalWebsite` można zobaczyć, gdzie wywoływana jest funkcja `HostsFileContainsEntry`:

```
if(-not (HostsFileContainsEntry ($header)))
{
    AddEntryToHosts -hostEntry $value
}
```

Możesz również zobaczyć, gdzie wywoływana jest funkcja `AddEntryToHosts`. To wszystko odbywa się wewnątrz instrukcji warunkowej `If`. Aby dowiedzieć się czegoś więcej na ten temat, możesz uruchomić polecenie `help *if*`:

```
PS C:\> help *if*
```

Name	Category	Module
----	-----	-----
diff	Alias	
New-ModuleManifest	Cmdlet	Microsoft.PowerShell.Core
Test-ModuleManifest	Cmdlet	Microsoft.PowerShell.Core
Get-AppxPackageManifest	Function	Appx
Get-PfxCertificate	Cmdlet	Microsoft.PowerShell.S...
Export-Certificate	Cmdlet	PKI
Export-PfxCertificate	Cmdlet	PKI
Get-Certificate	Cmdlet	PKI
Get-CertificateNotificationTask	Cmdlet	PKI
Import-Certificate	Cmdlet	PKI
Import-PfxCertificate	Cmdlet	PKI
New-CertificateNotificationTask	Cmdlet	PKI
New-SelfSignedCertificate	Cmdlet	PKI
Remove-CertificateNotification...	Cmdlet	PKI
Switch-Certificate	Cmdlet	PKI
Test-Certificate	Cmdlet	PKI
about_If	HelpFile	

Pliki pomocy są zwykle wyświetlane jako ostatnie i jak widać, istnieje również plik `about_tryCatchFinally.ps1`. Czytając go, możesz się dowiedzieć, jak działa taka konstrukcja warunkowa. W kontekście naszego przykładowego skryptu polecenie `If` sprawdza, czy funkcja `HostsFileContainsEntry` zwraca wartość `True`, czy `False`; jeżeli jest to `False`, wywoływana jest funkcja `AddEntryToHosts`. Taka struktura sugeruje, że funkcja `New-LocalWebsite` to „główna” funkcja w tym skrypcie lub funkcja, którą trzeba uruchomić, aby coś się stało. Funkcje `HostsFileContainsEntry` i `AddEntryToHosts` wydają się funkcjami usługowymi, które są wywoływane w razie potrzeby przez funkcję `New-LocalWebsite`. Skupmy się zatem na funkcji `New-LocalWebsite`:

```
function New-LocalWebsite([string] $sitePath, [string] $siteName)
{
    try
    {
        Import-Module WebAdministration
    }
    catch
    {
        Write-Host "IIS Powershell module is not installed. Please install it first, by adding
        ↳the feature"
    }
    Write-Host "AppPool is created with name: " $siteName
    New-WebAppPool -Name $siteName
    Set-ItemProperty IIS:\AppPools\$Name managedRuntimeVersion v4.0
    Write-Host
    if(-not (Test-Path $sitePath))
    {
        New-Item -ItemType Directory $sitePath
    }
    $header = "www."+$siteName+".local"
    $value = "127.0.0.1 " + $header
    New-Website -ApplicationPool $siteName -Name $siteName -Port 80 -PhysicalPath $sitePath
    ↳-HostHeader ($header)
    Start-Website -Name $siteName
    if(-not (HostsFileContainsEntry ($header)))
    {
        AddEntryToHosts -hostEntry $value
    }
}
```

Możesz mieć problemy ze zrozumieniem konstrukcji `Try`. Szybkie wyszukiwanie pomocy (`help *try*`) pozwala uzyskać plik pomocy `about_try_catch_finally`. Wyjaśnia on, że w części `Try` umieszczone są komponenty, których wykonanie może spowodować błąd. Jeżeli tak się stanie, wykonany zostanie kod umieszczony w części `catch`. No dobrze, w takim razie oznacza to, że funkcja spróbuje załadować moduł `WebAdministration`, a jeżeli to nie zadziała, wyświetli komunikat o błędzie. Szczerze mówiąc, uważamy, że po wystąpieniu błędu prawdopodobnie funkcja powinna po prostu całkowicie zakończyć działanie, ale nie robi tego, więc jeżeli moduł ten nie zostanie załadowany, możesz spodziewać się wystąpienia kolejnych błędów. Przed uruchomieniem tej funkcji powinieneś zatem upewnić się, że moduł `WebAdministration` jest dostępny!

Funkcja `Write-Host` pomaga w śledzeniu postępu działania skryptu. Kolejne polecenie to `New-WebAppPool`. Wyszukiwanie w systemie pomocy pokazuje, że jest ono częścią modułu `WebAdministration`, a jego przeznaczenie i opis sposobu działania można znaleźć w odpowiednim pliku pomocy. Kolejne polecenie, `Set-ItemProperty`, wydaje się ustawiać coś w właśnie utworzonym obiekcie `AppPool`.

Proste polecenie `Write-Host` jest tam chyba po to, aby wyświetlić na ekranie pusty wiersz. W porządku. Jeżeli spojrzysz na polecenie `Test-Path`, zobaczysz, że sprawdza ono, czy dana ścieżka (w tym przypadku prowadząca do folderu) istnieje. Jeżeli nie, skrypt wykorzystuje polecenie `New-Item` do utworzenia takiego folderu.

Zmienna `$head` wykorzystuje parametr `$siteName` do utworzenia adresu w stylu `www.sitename.local`, a zmienna `$value` zapamiętuje adres IP. Następnie uruchamiane jest polecenie `New-WebSite` z różnymi parametrami; więcej szczegółowych informacji na temat znaczenia poszczególnych parametrów znajdziesz w pliku pomocy tego polecenia.

Na koniec uruchamiane jest polecenie `Start-WebSite`. Plik pomocy „mówi”, że polecenie to uruchamia witrynę internetową. W tym miejscu kodu dochodzimy do wywołań funkcji `HostsFileContainsEntry` i `AddEntryToHosts`. Wygląda na to, że nowa strona internetowa, wymieniona w zmiennej `$value`, jest umieszczana w lokalnym pliku `HOSTS`, umożliwiającym lokalnie zamianę adresu IP na nazwę hosta.

26.2. Analiza wiersz po wierszu

Proces przedstawiony w poprzednim podrozdziale jest analizą skryptu wiersz po wierszu i jest to sugerowany i zalecany przez nas sposób postępowania z nowymi, nieznanymi skryptami. Analizując taki skrypt, w każdym wierszu powinieneś wykonać następujące czynności:

- Zidentyfikuj zmienne, spróbuj ustalić, co zawierają, i zapisz je na kartce papieru. Ponieważ zmienne są często przekazywane do parametrów poleceń, posiadanie zaktualizowanego zestawienia zmiennych wraz z opisem ich potencjalnego przeznaczenia i zawartości pomoże Ci przewidzieć rolę danego wiersza kodu.
- Gdy natkniesz się na nowe, nieznane Ci polecenia, uważnie zapoznaj się z ich plikami pomocy i spróbuj zrozumieć, co robią. Polecenia z rodziny `Get-` spróbuj uruchamiać, podając wartości, które skrypt przekazuje w zmiennych do parametrów, i sprawdź, jakie dane wyjściowe są generowane.
- Gdy natkniesz się na nieznane konstrukcje programowe, takie jak `if` czy `[environment]`, rozważ możliwość uruchamiania krótkich fragmentów kodu w maszynie wirtualnej, aby zobaczyć, co one robią (dzięki zastosowaniu maszyny wirtualnej pomagasz chronić środowisko produkcyjne). Wyszukaj odpowiednie słowa kluczowe w systemie pomocy (za pomocą symboli wieloznacznych) i zapoznaj się z ich plikami pomocy.

A przede wszystkim pamiętaj, że nie powinieneś pomijać żadnego wiersza kodu. Diabeł tkwi w szczegółach, więc nie postępuj według zasady „Cóż, skoro nie wiem, co to oznacza, to po prostu przejdę dalej”. Zawsze powinieneś poświęcić trochę czasu na

wyjaśnienie, co robi każdy wiersz skryptu, lub przynajmniej, co wydaje Ci się, że robi. Przeprowadzenie takiej dokładnej analizy pomoże Ci potem ustalić, gdzie i jak powinieneś zmodyfikować skrypt, aby dostosować go do własnych potrzeb.

26.3. Ćwiczenia

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej.

Na listingu 26.2 przedstawiono kompletny kod skryptu. Sprawdź, czy będziesz w stanie rozpoznać, co robi i jak z niego korzystać. Czy potrafisz przewidzieć jakieś błędy, które może spowodować jego uruchomienie? Co możesz zrobić, aby użyć tego skryptu w swoim środowisku?

Listing 26.2. Get-LastOn.ps1

```
function get-LastOn {
<#
.DESCRIPTION
Tell me the most recent event log entries for logon or logoff.
.BUGS
Blank 'computer' column
.EXAMPLE
get-LastOn -computername server1 | Sort-Object time -Descending |
Sort-Object id -unique | format-table -AutoSize -Wrap
ID          Domain      Computer  Time
--          -
LOCAL SERVICE  NT AUTHORITY      4/3/2012 11:16:39 AM
NETWORK SERVICE NT AUTHORITY      4/3/2012 11:16:39 AM
SYSTEM        NT AUTHORITY      4/3/2012 11:16:02 AM
Sorting -unique will ensure only one line per user ID, the most recent.
Needs more testing
.EXAMPLE
PS C:\Users\administrator> get-LastOn -computername server1 -newest 10000
-maxIDs 10000 | Sort-Object time -Descending |
Sort-Object id -unique | format-table -AutoSize -Wrap
ID          Domain      Computer  Time
--          -
Administrator  USS          4/11/2012 10:44:57 PM
ANONYMOUS LOGON NT AUTHORITY  4/3/2012 8:19:07 AM
LOCAL SERVICE  NT AUTHORITY  10/19/2011 10:17:22 AM
NETWORK SERVICE NT AUTHORITY  4/4/2012 8:24:09 AM
student        WIN7         4/11/2012 4:16:55 PM
SYSTEM        NT AUTHORITY  10/18/2011 7:53:56 PM
USSDC$        USS          4/11/2012 9:38:05 AM
WIN7$         USS          10/19/2011 3:25:30 AM
PS C:\Users\administrator>
.EXAMPLE
get-LastOn -newest 1000 -maxIDs 20
Only examines the last 1000 lines of the event log
.EXAMPLE
get-LastOn -computername server1 | Sort-Object time -Descending |
Sort-Object id -unique | format-table -AutoSize -Wrap
```

```
#>
param (
    [string]$ComputerName = 'localhost',
    [int]$Newest = 5000,
    [int]$maxIDs = 5,
    [int]$logonEventNum = 4624,
    [int]$logoffEventNum = 4647
)
$eventsAndIDs = Get-EventLog -LogName security -Newest $Newest |
Where-Object {$_.instanceid -eq $logonEventNum -or $_.instanceid -eq $logoffEventNum} |
Select-Object -Last $maxIDs -Property TimeGenerated,Message,ComputerName
foreach ($event in $eventsAndIDs) {
    $id = ($event |
        parseEventLogMessage |
        where-Object {$_.fieldName -eq "Account Name"} |
        Select-Object -Last 1).fieldValue
    $domain = ($event |
        parseEventLogMessage |
        where-Object {$_.fieldName -eq "Account Domain"} |
        Select-Object -Last 1).fieldValue
    $props = @{'Time'=$event.TimeGenerated;
        'Computer'=$ComputerName;
        'ID'=$id
        'Domain'=$domain}
    $output_obj = New-Object -TypeName PSObject -Property $props
    write-output $output_obj
}
}
function parseEventLogMessage()
{
    [CmdletBinding()]
    param (
        [parameter(ValueFromPipeline=$True,Mandatory=$True)]
        [string]$Message
    )
    $eachLineArray = $Message -split "`n"
    foreach ($oneLine in $eachLineArray) {
        write-verbose "line:_$oneLine_"
        $fieldName,$fieldValue = $oneLine -split ":", 2
        try {
            $fieldName = $fieldName.trim()
            $fieldValue = $fieldValue.trim()
        }
        catch {
            $fieldName = ""
        }
        if ($fieldName -ne "" -and $fieldValue -ne "" )
        {
            $props = @{'fieldName'="$fieldName";
                'fieldValue'=$fieldValue}
            $output_obj = New-Object -TypeName PSObject -Property $props
            Write-Output $output_obj
        }
    }
}
Get-LastOn
```

Pamiętaj, że ten skrypt powinien działać w takiej postaci, w jakiej jest. Nie daje się go jednak uruchomić w Twoim systemie. Czy potrafisz zdiagnozować przyczynę problemu? Pamiętaj, że większość z tych poleceń była omawiana w książce, a dla pozostałych odpowiednie informacje znajdziesz w systemie pomocy powłoki PowerShell. Wszystkie techniki zastosowane w tym skrypcie mają swoje odzwierciedlenie w przykładach zamieszczonych w plikach pomocy poszczególnych poleceń.

26.4. Odpowiedzi

W skrypcie zdefiniowane są dwie funkcje, które nic nie robią, dopóki nie zostaną wywołane. Na końcu skryptu znajduje się polecenie `Get-LastOn`, które ma taką samą nazwę jak jedna z funkcji, więc możemy założyć, że to ona jest wykonywana. Patrząc na tę funkcję, widać, że ma wiele parametrów domyślnych, co wyjaśnia, dlaczego w jej wywołaniu nie musi być innych parametrów. Pomoc oparta na komentarzach wyjaśnia również, co robi ta funkcja. Pierwsza jej część wykorzystuje polecenie `Get-EventLog`:

```
$eventsAndIDs = Get-EventLog -LogName security -Newest $Newest |
Where-Object {$_.instanceid -eq $logonEventNum -or $_.instanceid -eq
$logoffEventNum} | Select-Object -Last $maxIDs -Property
TimeGenerated,Message,ComputerName
```

Jeżeli byłby to nowy cmdlet, przyjrzelibyśmy się plikom pomocy i przykładom. Wydaje się, że użyte tutaj wyrażenie pobiera najnowsze zdarzenia z dziennika zabezpieczeń. Zmienna `$Newest` to parametr wywołania skryptu, który ma wartość domyślną 5000. Pobrane zdarzenia są następnie filtrowane przez polecenie `Where-Object`, które wybiera zdarzenia o konkretnych dwóch wartościach identyfikatorów, również zdefiniowanych w parametrach wywołania skryptu.

Następnie wygląda na to, że w pętli `foreach` przetwarzane są poszczególne pobrane zdarzenia. Oto potencjalna pułapka: jeżeli dziennik nie zawiera żadnych pasujących zdarzeń, kod w tej pętli najprawdopodobniej nie zostanie wykonany poprawnie, chyba że zaimplementowana będzie dobra obsługa błędów.

W pętli `foreach` ustawiane są inne zmienne. Pierwsza definicja zmiennej pobiera obiekt zdarzenia i przekazuje go za pomocą potoku do czegoś o nazwie `parseEvent` → `message`. To nie wygląda jak nazwa cmdletu, ale widzieliśmy w skrypcie funkcję o takiej nazwie. Przechodząc do niej, zauważamy, że pobiera ona komunikat jako parametr, dzieli go i zapisuje poszczególne elementy w tablicy. Być może będziemy musieli szczegółowo zapoznać się z działaniem operatora `-Split`.

Każda wiersz w tablicy jest przetwarzany przez inną pętlę `foreach`. Wygląda na to, że poszczególne wiersze są ponownie dzielone i że mamy tutaj blok `try/catch` do obsługi błędów. I znów być może będziemy musieli zapoznać się z jego plikiem pomocy, aby dowiedzieć się, jak to działa. Wreszcie na koniec mamy instrukcję `if`, która sprawdza, czy poszczególne podzielone elementy nie są puste i jeżeli nie są, tworzona jest zmienna o nazwie `$props` będąca tablicą haszującą lub inaczej mówiąc, tablicą asocjacyjną. Cała funkcja byłaby niewątpliwie znacznie łatwiejsza do przeanalizowania, gdyby autor zamieścił w kodzie choć kilka komentarzy. W każdym razie funkcja parsowania

kończy się po wywołaniu polecenia `New-Object` (to kolejne polecenie, o którym musimy trochę poczytać).

Wyniki działania tej funkcji są przekazywane do funkcji wywołującej. Zdaje się, że podobny proces jest powtarzany dla zmiennej `$domain`.

Na koniec mamy kolejną tabelę mieszającą i polecenie `New-Object`, ale teraz powinniśmy już wiedzieć, co robi ta funkcja. Wyświetlenie wyników kończy działanie funkcji i zarazem całego skryptu.

27

To jeszcze nie koniec

Dotarliśmy już prawie do końca książki, ale to nie oznacza bynajmniej, że o powłoce PowerShell dowiedziałeś się już wszystkiego. W rzeczywistości pozostało jeszcze całe mnóstwo zagadnień, o których nie mieliśmy tutaj okazji nawet wspomnieć, ale to, czego się do tej pory nauczyłeś, z pewnością będzie dla Ciebie dobrym punktem wyjścia do dalszego, samodzielnego pogłębiania swojej wiedzy na temat powłoki PowerShell. Ten krótki rozdział pomoże Ci podążać we właściwym kierunku.

27.1. Pomysły na dalszą eksplorację powłoki PowerShell

W tej książce skupiliśmy się na umiejętnościach i technikach, które powinieneś znać, aby efektywnie pracować z powłoką PowerShell. Innymi słowy, korzystając z nabytej tutaj wiedzy, powinieneś być już w stanie rozpocząć wykonywanie zadań przy użyciu tysięcy poleceń dostępnych dla powłoki PowerShell, niezależnie od tego, czy takie zadania są związane z systemem Windows, serwerami Exchange, SharePoint, czy czymś zupełnie innym.

Twoim następnym krokiem powinno być rozpoczęcie łączenia poleceń w skrypty, pozwalające na zautomatyzowanie złożonych wieloetapowych procesów i utworzenie gotowych do użycia narzędzi, które można udostępnić innym użytkownikom. Cały proces tworzenia nowych narzędzi to znakomity temat na kolejną książkę, taką jak na przykład *Learn PowerShell Toolmaking in a Month of Lunches* (wyd. Manning, 2012). Ale nawet korzystając tylko z tego, czego nauczyłeś się w tej książce, możesz tworzyć rozbudowane, sparametryzowane skrypty, zawierające tyle poleceń, ile potrzebujesz do wykonania zadania — a to już początek tworzenia prawdziwych narzędzi.

Jakie jeszcze zagadnienia są związane z tworzeniem narzędzi powłoki PowerShell? Poniżej przedstawiamy krótkie zestawienie:

- uproszczony język skryptowy powłoki PowerShell,
- zasięgi,
- funkcje i możliwość tworzenia wielu narzędzi w jednym pliku skryptu,
- obsługa błędów,
- tworzenie plików pomocy,
- debugowanie,
- tworzenie niestandardowego formatowania,
- niestandardowe rozszerzenia typów,
- moduły skryptów i manifestów,
- korzystanie z baz danych,
- przepływy,
- rozwiązywanie problemów z potokami,
- złożone hierarchie obiektów,
- globalizacja i lokalizacja,
- funkcje proxy,
- ograniczone przekazywanie i delegowanie zarządzania,
- korzystanie ze środowiska .NET.

Jest jeszcze dużo więcej. Jeżeli jesteś wystarczająco zainteresowany i masz odpowiednie umiejętności oraz doświadczenie, możesz nawet zostać członkiem trzeciej grupy użytkowników powłoki PowerShell: deweloperem oprogramowania. Istnieje przecież cały zestaw technik i technologii związanych z rozbudową samej powłoki PowerShell, wykorzystywaniem jej podczas tworzenia nowych programów i nie tylko. To naprawdę produkt o ogromnych możliwościach!

27.2. „Od czego mam zacząć, kiedy przeczytałem już tę książkę?”

Najlepszym rozwiązaniem będzie znalezienie sobie konkretnego zadania do wykonania. Wybierz zadanie ze swojego środowiska produkcyjnego, takie, które uważasz za powtarzalne, i spróbuj zautomatyzować je za pomocą powłoki. Niemal na pewno znajdziesz się w sytuacji, w której nie będziesz wiedział, jak coś zrobić, i to właśnie jest idealne miejsce na rozpoczęcie samodzielnej nauki.

Oto kilka zadań, które są często wykonywane przez administratorów:

- Napisz skrypt zmieniający hasło używane przez usługę do zalogowania się i wykonujący taką operację na wielu komputerach, na których działa taka usługa (możesz to zrobić za pomocą jednego polecenia).
- Napisz skrypt automatyzujący obsługę nowych użytkowników, w tym tworzenie kont użytkowników, ich skrzynek pocztowych i katalogów domowych. Ustawianie uprawnień NTFS przy użyciu powłoki PowerShell jest dosyć trudne, ale powinienś rozważyć użycie narzędzia takiego jak *Cacls.exe* lub *Xcacls.exe* z poziomu skryptu PowerShell zamiast natywnych (i złożonych) poleceń *Get-ACL* i *Set-ACL* powłoki PowerShell.

- Napisz skrypt, który w wybrany sposób będzie zarządzał skrzynkami pocztowymi Exchange — na przykład pobierając informacje o rozmiarach skrzynek pocztowych i tworząc raporty o użytkownikach, których skrzynki mają największe rozmiary.
- Zautomatyzuj obsługę nowych stron internetowych na serwerze IIS, korzystając z modułu WebAdministration dostępnego w systemie Windows Server 2008 R2 i nowszych (moduł ten działa również z IIS 7 w systemie Windows Server 2008).

Najważniejszą rzeczą do zapamiętania jest to, aby *nie przekombinować* wybranego zadania. Don spotkał się kiedyś z administratorem, który przez wiele tygodni walczył z napisaniem w powłoce PowerShell solidnego, odpornego na błędy skryptu do kopiowania plików, za pomocą którego mógłby dostarczać pliki do farmy serwerów WWW. Don zadał mu wtedy proste pytanie: „A dlaczego nie chcesz po prostu użyć do tego celu gotowego, sprawdzonego narzędzia, takiego jak Xcopy czy Robocopy?”. Administrator przez długą chwilę patrzył na Dona bez słowa, a potem się roześmiał. Był tak pochłonięty „robieniem tego w PowerShellu”, że po prostu zapomniał, że ta powłoka może z powodzeniem korzystać ze wszystkich narzędzi, które są już dostępne w każdym systemie.

27.3. Inne zasoby, o których warto pamiętać

Spędzamy dużo czasu, pracując z powłoką PowerShell, pisząc o niej książki i artykuły oraz ucząc posługiwania się nią naszych studentów. Zapytajcie nasze rodziny — czasami nawet nie przestajemy o tym rozmawiać przy obiedzie. Oznacza to, że w naszej podręcznej bazie adresów zgromadziliśmy wiele zasobów internetowych, z których korzystamy codziennie i które polecamy wszystkim naszym studentom. Mamy zatem nadzieję, że również dla Ciebie będą stanowiły dobry punkt wyjścia.

- <https://powershell.org/> — to powinien być Twój pierwszy przystanek. Znajdziesz tu niemal wszystko, począwszy od forów Q&A, po bezpłatne e-booki, bezpłatne webinaria, podcasty, aż do transmisji na żywo z różnych ciekawych wydarzeń edukacyjnych i innych. To centralne miejsce spotkań dużej części społeczności użytkowników powłoki PowerShell, które działa już od wielu lat.
- <https://www.youtube.com/powershellorg> i <http://youtube.com/powershellldon> — kanały YouTube serwisu *PowerShell.org* oraz kanał filmowy Dona, gdzie znajdziesz całą masę darmowych filmów na temat powłoki PowerShell, w tym sesje nagrane na konferencji PowerShell + DevOps Global Summit.
- <https://jdhitsolutions.com/> — to blog Jeffa na temat powłoki PowerShell i ogólnie tworzenia skryptów.
- <https://donjones.com/> — to osobisty blog Dona, w którym często pisze na temat powłoki PowerShell i różnych powiązanych z nią zagadnień.
- <http://devopscollective.org/> — to organizacja nadrzędna dla *PowerShell.org*, która koncentruje się na szerszym zastosowaniu podejścia DevOps do zarządzania w rozwinięciach IT.

Nasi studenci często pytają, czy są jakieś inne książki na temat powłoki PowerShell, które polecamy. Dwie z nich to *Learn PowerShell Toolmaking in a Month of Lunches* oraz *PowerShell in Depth* (obie wydane przez wydawnictwo Manning). Jesteśmy autorami lub współautorami tych książek, więc jeżeli spodobała Ci się książka, którą trzymasz w ręku, to prawdopodobnie te dwie wymienione pozycje również będą Ci odpowiadać. Polecamy także książkę *PowerShell Deep Dives* (wyd. Manning, 2013), która jest zbiorem mocno technicznych artykułów napisanych przez różnych autorów będących PowerShell MVP (dochody z książki są przeznaczone dla organizacji charytatywnej Save the Children, więc nie zaszkodzi, jak od razu kupisz ze trzy egzemplarze). Wreszcie, jeżeli jesteś fanem szkoleń wideo, na stronie <https://www.pluralsight.com/> znajdziesz zarówno sporo naszych ciekawych filmów, jak i tysiące innych filmów związanych z IT.

PowerShell — podręczna ściągawka

Ten rozdział jest dla nas świetną okazją, aby zebrać wiele małych, ale bardzo istotnych zagadnień w jednym miejscu. Jeżeli kiedykolwiek będziesz miał problemy z przypomnieniem sobie jakiegoś niuansu funkcjonowania powłoki, najpierw powinieneś zajrzeć do tego rozdziału.

28.1. Interpunkcja

Nie ma wątpliwości, że powłoka PowerShell jest pełna znaków interpunkcyjnych, a znaczna ich część ma inne znaczenie w plikach pomocy niż w samej powłocie. Oto co takie znaki oznaczają w powłocie:

- ``` (*odwrócony apostrof* albo *gravis* — ang. *backtick*) — w powłocie PowerShell odgrywa rolę znaku ucieczki (ang. *escape character*). Usuwa specjalne znaczenie dowolnego znaku, który następuje po nim. Na przykład spacja jest zwykle znakiem separatora, dlatego próba wykonania polecenia `cd c:\Program Files` kończy się wygenerowaniem błędu. Jednak poprzedzenie spacji w nazwie ścieżki znakiem ucieczki, `cd c:\Program` Files`, usuwa to specjalne znaczenie spacji i wymusza traktowanie jej jako literału, dzięki czemu całe polecenie działa poprawnie.
- `~` — kiedy *tylda* jest używana jako część ścieżki, reprezentuje katalog domowy bieżącego użytkownika, zdefiniowany w zmiennej środowiskowej `UserProfile`.
- `()` — *nawiasy okrągłe* są używane na kilka sposobów:
 - Podobnie jak w matematyce, nawiasy określają kolejność wykonywania operacji. Powłoka PowerShell wykonuje najpierw polecenia znajdujące się w nawiasach, od najbardziej zagnieżdżonych nawiasów do skrajnych. Jest to dobry spo-

sób na uruchomienie polecenia i przekazanie jego wyników działania jako parametru wejściowego innego polecenia, na przykład `Get-Service -computerName (Get-Content c:\computernames.txt)`.

- W nawiasach umieszczamy również parametry wywołania metod i musimy je dodawać nawet w sytuacji, kiedy dana metoda nie wymaga podawania żadnych parametrów, na przykład `Change-Start-Mode('Automatic')` lub `Delete()`.
- `[]` — *nawiasy kwadratowe* mają dwa główne zastosowania w powłoce:
 - Umieszczamy w nich numer indeksu w sytuacji, kiedy chcemy odwołać się do pojedynczego obiektu w tablicy lub kolekcji, na przykład `$services[2]` pobiera trzeci obiekt z tablicy `$services` (indeksy są zawsze numerowane od zera).
 - Zawierają nazwę typu podczas konwersji danych z jednego typu na inny (rzutowanie danych). Na przykład polecenie `$myresult / 3 -as [int]` dokonuje rzutowania wyniku na liczbę całkowitą (integer), a polecenie `[xml]$data = Get-Content data.xml` odczytuje zawartość pliku `Data.xml` i dokonuje próby parsowania zawartości jako dokumentu w formacie XML.
- `{ }` — *nawiasy klamrowe* mają w powłoce PowerShell trzy zastosowania:
 - Zawierają bloki kodu wykonywalnego lub poleceń, zwane *blokami skryptu* (ang. *script blocks*). Są często przekazywane do parametrów, które oczekują bloku skryptu lub bloku filtru, na przykład `Get-Service | Where-Object { $_.Status -eq "Running" }`.
 - Zawierają pary klucz-wartość podczas tworzenia nowej tablicy mieszającej. Klamra otwierająca jest zawsze poprzedzona znakiem `@`. W poniższym przykładzie używamy nawiasów klamrowych, aby objąć pary klucz-wartość tabeli mieszającej (dwie pary) i aby dołączyć blok skryptu wyrażenia, który jest wartością drugiego klucza, `e`: `$hashtable = @{'l'='Label';e={expression}}`.
 - Gdy nazwa zmiennej zawiera spacje lub inne znaki normalnie niedozwolone w nazwie zmiennej, nazwa takiej zmiennej musi zostać ujęta w nawiasy klamrowe, na przykład `${Moja zmienna}`.
- `' '` — w *apostrofach* umieszczamy ciągi znaków. W ciągach znaków ujętych w apostrofy powłoka PowerShell nie będzie szukała znaków ucieczki ani nazw zmiennych.
- `" "` — w *znakach cudzysłowu* również umieszczamy ciągi znaków. W ciągach znaków ujętych w cudzysłów powłoka PowerShell będzie szukała znaków ucieczki oraz znaków `$`. Znaki ucieczki są przetwarzane, natomiast znaki następujące po symbolu `$` (do następnego białego znaku) są przyjmowane jako nazwa zmiennej, która jest zastępowana przez jej zawartość. Na przykład jeżeli zmienna `$one` zawiera słowo *World*, to zmienna `$two = "Hello $one`n "` będzie zawierać ciąg znaków *Hello World* i znak powrotu karetki (znak specjalny ``n` reprezentuje powrót karetki).
- `$` — *znak dolara* informuje powłokę, że ciąg znaków, który po nim następuje (aż do następnego białego znaku), reprezentuje nazwę zmiennej. Może to być nieco mylące podczas pracy z cmdletami zarządzającymi zmiennymi. Przy założeniu, że zmienna `$one` przechowuje ciąg znaków *two*, wykonanie polecenia `New-Variable`

-name \$one -value 'Hello' spowoduje utworzenie nowej zmiennej o nazwie two, której wartością będzie ciąg znaków *Hello*, ponieważ znak dolara użyty w tym poleceniu informuje powłokę, że chcesz użyć zawartości zmiennej \$one. Dla porównania wykonanie polecenia New-Variable -name one -value 'Hello' spowoduje utworzenie nowej zmiennej o nazwie \$one.

- % — *znak procentu* jest aliasem dla polecenia ForEach-Object. Jest to również operator dzielenia modulo, który zwraca resztę z operacji dzielenia.
- ? — *znak zapytania* jest aliasem dla polecenia Where-Object.
- > — *znak większości* to rodzaj aliasu dla polecenia Out-File. Technicznie rzecz biorąc, nie jest to prawdziwy alias, ale zapewnia przekierowywanie wyjścia poleceń w starym stylu konsoli Cmd.exe, na przykład dir> files.txt.
- +, -, *, /, % — to *operatory matematyczne*, które działają jako standardowe operatory arytmetyczne. Zwróć uwagę, że operator + jest również używany do łączenia ciągów znaków.
- - — *myślnik* lub inaczej *łącznik* poprzedza zarówno nazwy parametrów, jak i wiele operatorów, na przykład -computerName czy -eq. Za pomocą myślnika oddzielamy również czasowniki i rzeczowniki składające się na nazwę cmdletu, jak na przykład w poleceniu Get-Content; znak myślnika służy również jako arytmetyczny operator odejmowania.
- @ — *znak at*, czyli popularna *małpa*, ma cztery zastosowania w powłoce PowerShell:
 - Poprzedza nawias klamrowy otwierający tabelę mieszającą (patrz hasło *nawiasy klamrowe* powyżej).
 - Używany przed nawiasami oznacza listę wartości rozdzielonych przecinkami, które tworzą tablicę, na przykład \$array = @(1,2,3,4). Zarówno znak @, jak i nawiasy są opcjonalne, ponieważ powłoka normalnie traktuje każdą listę elementów oddzielonych od siebie przecinkami jako tablicę.
 - Znakiem @ oznaczane są bloki tekstu, tzw. *here-strings*, w których wszystkie znaki są traktowane jako literały. Taki blok tekstu rozpoczyna się od ciągu znaków @" i kończy ciągiem znaków "@, przy czym ten zamykający ciąg znaków musi znajdować się na początku nowego wiersza. Więcej szczegółowych informacji i przykładów takich bloków tekstu znajdziesz w pliku pomocy about_↪quoting_rules. Bloki *here-strings* mogą być również definiowane z użyciem apostrofów.
 - Znak @ odgrywa również rolę tzw. operatora splattingu powłoki PowerShell. Jeżeli tworzysz tabelę mieszającą, w której nazwy kluczy odpowiadają nazwom parametrów, a wartości kluczy są wartościami tych parametrów, możesz za pomocą tego operatora przekazać taką tablicę mieszającą do polecenia powłoki. Don napisał artykuł na temat splattingu dla „TechNet Magazine” (zobacz <http://technet.microsoft.com/en-us/magazine/gg675931.aspx>).
- & — *znak i* (*ampersand*) odgrywa w powłoce PowerShell rolę operatora wywołania, który nakazuje jej traktowanie następującego po nim elementu jako polecenia i jego uruchomienie. Na przykład wykonanie polecenia \$a = "Dir" spowoduje

umieszczenie ciągu znaków *Dir* w zmiennej *\$a*, a następnie wykonanie polecenia *&\$a* spowoduje uruchomienie polecenia *Dir*.

- *:* — *średnik* służy do oddzielania od siebie dwóch niezależnych poleceń powłoki PowerShell, które są zawarte w jednym wierszu. Na przykład uruchomienie polecenia *Dir*; *Get-Process* spowoduje wykonanie polecenia *Dir*, a następnie wykonanie polecenia *Get-Process*. Wyniki działania obu poleceń są wysyłane do tego samego potoku, z tym że wyniki działania pierwszego z nich (w tym przykładzie *Dir*) nie są przekazywane do drugiego polecenia (*Get-Process*).
- *#* — znak *hash* jest używany jako znak komentarza. Wszelkie znaki następujące po znaku *#*, aż do następnego znaku powrotu karetki, są ignorowane przez powłokę. Nawiasy kątowe *< i >* są używane jako część znaczników definiujących blok komentarza: ciąg znaków *<#* rozpoczyna blok komentarza, a ciąg znaków *#>* go kończy. Wszystko, co zostanie umieszczone w obrębie bloku komentarza, zostanie zignorowane przez powłokę.
- *=* — *znak równości* jest operatorem przypisania, pozwalającym na przypisanie wartości do zmiennej, na przykład *\$one = 1*. Znak ten nie jest wykorzystywany do porównywania dwóch wielkości; zamiast tego powinieneś użyć operatora *-eq*. Zauważ, że znak równości może być użyty w połączeniu z operatorami matematycznymi, na przykład wykonanie operacji *\$var +=5* spowoduje dodanie liczby 5 do bieżącej wartości zmiennej *\$var*.
- *|* — *znak potoku* służy do przekazywania wyników działania z wyjścia jednego polecenia na wejście drugiego. Drugie polecenie (otrzymujące dane z wyjścia pierwszego) wykorzystuje mechanizm wiązania parametrów do określenia, który parametr lub parametry będą odbierać obiekty z potoku. Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 9.
- */* lub ** — *prawy ukośnik* (ang. *forward slash*) jest używany jako operator dzielenia w wyrażeniach matematycznych; oba ukośniki (lewy i prawy) mogą być używane jako separatory nazw katalogów w ścieżkach; ścieżka *C:\Windows* jest taka sama jak *C:/Windows*. Lewy ukośnik jest również stosowany jako znak ucieczki w kryteriach filtrowania WMI i wyrażeniach regularnych.
- *.* — *kropka* ma trzy główne zastosowania:
 - Służy do wskazania, że chcesz uzyskać dostęp do elementu takiego jak właściwość, metoda lub obiekt. Przykładowo, wyrażenie *\$_ .Status* pozwala uzyskać dostęp do właściwości *Status* obiektu, który zostanie podstawiony do symbolu zastępczego *\$_*.
 - Jest używana do uruchamiania skryptów w ramach bieżącego zasięgu, co oznacza, że wszystko, co zostało zdefiniowane w takim skrypcie, pozostanie dostępne po zakończeniu działania skryptu, na przykład *. c:\myscript.ps1*.
 - Dwie kropki (*..*) tworzą operator zakresu, który zostanie omówiony w dalszej części tego rozdziału. Za pomocą dwóch kropek możemy się również odwoływać do folderu nadrzędnego w systemie plików, na przykład *..\ścieżka*.
- *,* — *przecinek* znajdujący się poza cudzysłowem oddziela kolejne elementy listy lub tablicy, na przykład "Jeden", 2, "Trzy", 4. Można go używać do przekazywania

wielu wartości statycznych do parametru, który może pobierać listy wartości, na przykład `Get-Process -computername Server1, Server2, Server3`.

- `:` — *dwukropki* (a technicznie rzecz biorąc, właściwie dwa dwukropki) pozwala na uzyskiwanie dostępu do statycznych elementów klasy zgodnie z koncepcją programowania w środowisku .NET Framework, na przykład `[-datetime]::now` (choć można osiągnąć taki sam rezultat, wykonując polecenie `Get-Date`).
- `!` — *wykrzyknik* jest aliasem dla operatora logicznego `-not`.

Mamy nieodparte wrażenie, że jedynym elementem interpunkcyjnym znajdującym się na standardowej klawiaturze, którego aktywnie nie używa powłoka PowerShell, jest znak `^` (daszek), chociaż i on jest używany na przykład w wyrażeniach regularnych.

28.2. Pliki pomocy

Znaki interpunkcyjne w plikach pomocy mogą mieć nieco inne znaczenie:

- `[]` — *nawiasy kwadratowe* otaczające dowolny tekst wskazują, że jest on opcjonalny. Może to obejmować całe polecenie (`[-Name <string>]`) lub może wskazywać, że parametr jest pozycjonowany i że jego nazwa jest opcjonalna (`[-Name] <string>`). Nawiasy kwadratowe mogą również wskazywać, że parametr jest opcjonalny i jeżeli jest używany, może być pozycjonowany (`[[-Name] <string>]`). Pamiętaj, że jeżeli masz jakieś wątpliwości co do pozycjonowania parametru, zawsze w wierszu polecenia możesz użyć jego pełnej nazwy.
- `[]` — *puste nawiasy kwadratowe* wskazują, że parametr może akceptować wiele wartości (`<string []>` zamiast `<string>`).
- `< >` — *nawiasy kątowe* otaczają typy danych, wskazując, jakiego rodzaju wartości lub obiektu oczekuje dany parametr: `<string>`, `<int>`, `<process>` i tak dalej.

Zawsze powinienś uważnie zapoznać się z pełną zawartością pliku pomocy (wywołując polecenie `help`, za każdym razem dodając opcję `-full`), ponieważ znajdziesz tam szczegółowe opisy przeznaczenia i sposobu użycia poleceń oraz w większości przypadków przykłady ich zastosowania.

28.3. Operatory

Powłoka PowerShell nie używa tradycyjnych operatorów porównania, które występują w większości języków programowania. Zamiast tego wykorzystuje następujące operatory:

- `-eq` — *równe* (`-ceq` w przypadku porównywania ciągów znaków z uwzględnianiem wielkości liter).
- `-ne` — *różne* (`-cne` w przypadku porównywania ciągów znaków z uwzględnianiem wielkości liter).
- `-ge` — *większe lub równe* (`-cge` w przypadku porównywania ciągów znaków z uwzględnianiem wielkości liter).

- `-le` — *mniejsze lub równe* (`-cle` w przypadku porównywania ciągów znaków z uwzględnianiem wielkości liter).
- `-gt` — *większe niż* (`-cgt` w przypadku porównywania ciągów znaków z uwzględnianiem wielkości liter).
- `-lt` — *mniejsze niż* (`-clt` w przypadku porównywania ciągów znaków z uwzględnianiem wielkości liter).
- `-contains` — zwraca wartość `True`, jeżeli podana kolekcja zawiera określony obiekt (`$collection -contains $object`); `-notcontains` ma działanie przeciwne (nie zawiera).
- `-in` — zwraca wartość `True`, jeżeli określony obiekt znajduje się w określonej kolekcji (`$object -in $collection`); `-notin` ma działanie przeciwne (nie znajduje się).

Operatory logiczne są używane do łączenia ze sobą wielu operacji porównania:

- `-not` — zamienia wartości `True` i `False` (znak `!` jest aliasem tego operatora).
- `-and` — oba podwyrażenia muszą być prawdziwe, aby całe wyrażenie było prawdziwe.
- `-or` — każde podwyrażenie może być prawdziwe, aby całe wyrażenie było prawdziwe.

Ponadto istnieją operatory, które wykonują określone funkcje:

- `-join` — łączy elementy tablicy w jeden ciąg znaków, w którym poszczególne elementy są od siebie oddzielone znakiem separatora.
- `-split` — pobiera ciąg znaków zawierający listę elementów oddzielonych od siebie znakiem separatora i tworzy z niej tablicę.
- `-replace` — zastępuje wystąpienia jednego ciągu drugim.
- `-is` — zwraca wartość `True`, jeżeli element jest określonego typu (`$one -is [int]`).
- `-as` — dokonuje rzutowania danego elementu na określony typ danych (`$one -as [int]`).
- `..` — operator zakresu; `1..10` zwraca dziesięć obiektów, od 1 do 10.
- `-f` — operator formatu; zastępuje symbole zastępcze określonymi wartościami: `"{0}, {1}" -f "Hello", "World"`.

28.4. Niestandardowe właściwości i kolumny

W kilku rozdziałach pokazywaliśmy, jak definiować niestandardowe właściwości za pomocą polecenia `Select-Object` lub niestandardowe kolumny i elementy listy za pomocą, odpowiednio, poleceń `Format-Table` i `Format-List`. W przypadku tworzenia tabel mieszających składnia wyrażenia dla każdej niestandardowej właściwości lub kolumny jest następująca:

```
@{label='Column_or_Property_Name'; expression={Value_expression}}
```

Oba klucze, Label i Expression, mogą być skracane, odpowiednio, jako l i e (pamiętaj, aby wpisywać małą literę l, a nie cyfrę 1; zamiast małej litery l można również użyć małej litery n od słowa kluczowego Name).

```
@{n='Column_or_Property_Name'; e={Value_expression}}
```

Wewnątrz wyrażenia możesz użyć symbolu zastępczego \$_ do odwoływania się do bieżącego obiektu (takiego jak bieżący wiersz tabeli lub obiekt, do którego dodajesz nie-standardową właściwość):

```
@{n='ComputerName'; e={$_.Name}}
```

Zarówno polecenie Select-Object, jak i polecenia z rodziny Format- szukają klucza n (lub name, lub label, lub po prostu l) i klucza e; polecenia z rodziny Format- mogą również wykorzystywać parametry width i align (dotyczy to polecenia Format-Table) oraz format ↪ string. Więcej szczegółowych informacji i przykładów znajdziesz w pliku pomocy polecenia Format-Table.

28.5. Pobieranie parametrów z potoku

W rozdziale 9. dowiedziałeś się, że istnieją dwa typy wiązania parametrów: ByValue i ByPropertyName. Wiązanie ByValue jest realizowane jako pierwsze, a wiązanie ByProperty ↪ Name występuje tylko wtedy, gdy nie zadziała wiązanie ByValue.

W przypadku wiązania ByValue powłoka analizuje typ obiektu, który jest przekazywany za pomocą potoku. Jeżeli chcesz samodzielnie sprawdzić typ takiego obiektu, powinieneś przekazać go za pomocą potoku do polecenia Gm. Następnie powłoka sprawdza, czy którykolwiek z parametrów polecenia akceptuje taki typ danych wejściowych i czy może pobierać dane wejściowe ByValue bezpośrednio z potoku. Żadne polecenie nie może posiadać dwóch lub więcej parametrów pobierających dane tego samego typu bezpośrednio z potoku w taki sam sposób. Innymi słowy, nie powinieneś spotkać polecenia posiadającego dwa parametry, z których każdy pobiera ByValue dane typu <string> bezpośrednio z potoku.

Jeśli wiązanie ByValue nie działa, powłoka sprawdza wiązanie ByPropertyName. W takim scenariuszu powłoka analizuje właściwości potokowanego obiektu i próbuje dopasować do nich parametry o takich samych nazwach, które mogą pobierać wartości z potoku ByPropertyName. Jeżeli potokowany obiekt ma właściwości Name, Status i ID, powłoka będzie sprawdzać, czy polecenie ma parametry o nazwach Name, Status i ID. Parametry te muszą być również oznaczone jako akceptujące pobieranie danych z potoku ByPropertyName, co można sprawdzić w pliku pomocy (wywołując polecenie help, nie zapomnij o dodaniu opcji -full).

Spójrzmy, jak to robi powłoka PowerShell. Na potrzeby tego przykładu będziemy mówić o *pierwszym poleceniu* i *drugim poleceniu*, zakładając, że chcesz uruchomić komendę taką jak Get-Service | Stop-Service czy Get-Service | Stop-Process. Powłoka PowerShell postępuje w taki sposób:

1. Jaki jest typ obiektów (TypeName) tworzonych przez pierwsze polecenie? Aby to samodzielnie sprawdzić, możesz przekazać wyniki działania tego polecenia za pomocą potoku do polecenia `Gm`. Jeżeli nazwa typu obiektu składa się z wielu części, na przykład `System.Diagnostics.Process`, wystarczy, że zapamiętasz tylko ten ostatni człon nazwy: `Process`.
2. Czy jakiś parametr wejściowy drugiego polecenia akceptuje typ obiektów utworzonych przez pierwsze polecenie? Aby to sprawdzić, powinieneś uważnie zapoznać się z treścią pliku pomocy dla drugiego polecenia: `help <nazwa polecenia> -full`. Jeżeli tak, to czy taki parametr akceptuje pobieranie danych wejściowych bezpośrednio z potoku przy użyciu wiązania `ByValue`? Informację o tym znajdziesz w szczegółowych opisach poszczególnych parametrów w pliku pomocy.
3. Jeżeli odpowiedź na pytanie z punktu 2. brzmi *tak*, to cały obiekt utworzony przez pierwsze polecenie `cmdlet` zostanie przekazany do parametru określonego w punkcie 2. To koniec procesu — nie musisz przechodzić do punktu 4. Jeżeli jednak odpowiedź na pytanie z punktu 2. brzmi *nie*, przejdź do punktu 4.
4. Przyjrzyj się dokładnie obiektom będącym wynikiem działania pierwszego polecenia. Jakich właściwości mają te obiekty? Możesz to sprawdzić, ponownie przekazując wyniki działania pierwszego polecenia za pomocą potoku do polecenia `Get-Member`.
5. Przyjrzyj się parametrom drugiego polecenia (ponownie musisz zajrzeć do pliku pomocy). Czy są jakieś parametry, które (a) mają taką samą nazwę jak jedna z właściwości obiektu z punktu 4. i (b) akceptują dane wejściowe bezpośrednio z potoku za pomocą wiązania `ByPropertyName`?
6. Jeżeli dowolne parametry spełniają kryteria z punktu 5., wartości właściwości zostaną przekazane do parametrów o tej samej nazwie i drugie polecenie zostanie uruchomione. Jeżeli żadne dopasowania między nazwami właściwości i parametrami akceptującymi wiązanie `ByPropertyName` nie zostaną znalezione, drugie polecenie zostanie uruchomione bez pobierania parametrów bezpośrednio z potoku.

Pamiętaj, że zawsze możesz ręcznie wprowadzić parametry i ich wartości dla każdego polecenia. Jeżeli tak zrobisz, uniemożliwi to takiemu parametrowi pobieranie danych bezpośrednio z potoku, nawet jeżeli normalnie tak by się stało.

28.6. Kiedy używać symbolu zastępczego `$_`

To prawdopodobnie jeden z aspektów pracy z powłoką PowerShell wprowadzających największe zamieszanie: kiedy stosowanie symbolu zastępczego `$_` jest dozwolone?

Ten symbol zastępczy będzie działał tylko wtedy, gdy powłoka będzie go jawnie poszukiwać i będzie przygotowana do wypełnienia go obiektami. Ogólnie rzecz biorąc, dzieje się tak tylko w bloku skryptu przetwarzającego dane wejściowe z potoku, gdzie w miejsce symbolu `$_` podstawiane jest po jednym obiekcie wejściowym naraz. Z takim rozwiązaniem możesz się spotkać w kilku miejscach:

- w bloku skryptu filtra używanym przez polecenie Where-Object:
`Get-Service | 3 Where-Object {$_.Status -eq "Running"}`
- w blokach skryptu w pętli ForEach-Object, takich jak główny blok Process używany zwykle w tym poleceniu *cmdlet*:
`Get-WmiObject -class Win32_Service -filter "name = 'mssqlserver'" |
 ➤ ForEach-Object -process {$_.ChangeStartMode ('Automatic')}`
- w bloku skryptu Process funkcji filtrującej lub innej funkcji zaawansowanej — więcej szczegółowych informacji na ten temat znajdziesz w innej naszej książce, *Learn PowerShell Toolmaking in a Month of Lunches*;
- w wyrażeniach używanych do tworzenia niestandardowych właściwości lub kolumn tabeli mieszającej — więcej szczegółowych informacji na ten temat znajdziesz w podrozdziale 28.4. i rozdziałach 8., 9. i 10.

W każdym z wymienionych przypadków symbol \$_ występuje tylko w nawiasach klamrowych bloku skryptu. To całkiem dobra zasada wskazująca miejsce, w którym można użyć tego symbolu.

Ćwiczenia podsumowujące

Niniejszy dodatek zawiera trzy zestawy ćwiczeń podsumowujących, nad którymi możesz pracować po ukończeniu wybranych rozdziałów i ćwiczeń znajdujących się na ich końcu. Te zestawy ćwiczeń to świetny sposób na to, aby zrobić sobie przerwę w procesie uczenia się i utrwalić najważniejsze rzeczy, których się do tej pory nauczyłeś. Przykładowe odpowiedzi znajdują się na końcu tego dodatku.

Ponieważ opisy niektórych ćwiczeń są dosyć złożone, podzieliliśmy je na osobne sekcje. Na początku każdego z ćwiczeń zamieszczamy również wskazówki, które będą Ci przypominać o niektórych poleceniach, ich składni oraz plikach pomocy, które mogą być przydatne do ukończenia danego ćwiczenia.

Zestaw 1.: rozdziały 1. – 6.

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej. Przed przystąpieniem do wykonywania tego zestawu ćwiczeń powinieneś wykonać wszystkie ćwiczenia w rozdziałach 1. – 6. tej książki.

Podpowiedzi

- Sort-Object
- Select-Object
- Import-Module
- Export-CSV
- Help
- Get-ChildItem (Dir)

Zadanie 1.

Uruchom polecenie, które wyświetli najnowszych 100 wpisów z dziennika zdarzeń Application. Nie używaj polecenia `Get-WinEvent`.

Zadanie 2.

Napisz polecenie, które będzie wyświetlało listę pięciu procesów wykorzystujących najwięcej pamięci wirtualnej (VM).

Zadanie 3.

Utwórz plik CSV zawierający listę wszystkich usług dostępnych na Twoim komputerze. Lista powinna zawierać tylko nazwę i status usługi. Usługi uruchomione powinny znajdować się na liście *przed* usługami zatrzymanymi.

Zadanie 4.

Napisz polecenie, które zmienia typ uruchamiania usługi BITS na Manual (ręczny).

Zadanie 5.

Wyświetl listę wszystkich plików o nazwie *win**. * znajdujących się na Twoim komputerze. Poszukiwanie plików rozpocznij od folderu `C:\`. Uwaga: aby wykonać to zadanie, być może będziesz musiał trochę poeksperymentować i zastosować nowe parametry polecenia.

Zadanie 6.

Wyświetl listę wszystkich plików z katalogu `C:\Program Files`. Uwzględnij wszystkie podfoldery i pliki. Przekieruj listę katalogów do pliku tekstowego o nazwie `C:\Dir.txt` (do przekierowania strumienia danych możesz użyć znaku `>` lub polecenia `Out-File`).

Zadanie 7.

Wyświetl listę 20 najnowszych wpisów z dziennika zdarzeń Security i przekonwertuj wyniki działania na format XML. Nie zapisuj pliku na dysku; zamiast tego wyświetl kod XML w oknie konsoli.

Zauważ, że wyniki w formacie XML mogą być wyświetlone jako pojedynczy obiekt najwyższego poziomu, a nie jako surowy kod XML — to dobrze. W ten sposób powłoka PowerShell wyświetla dane w formacie XML. Jeżeli chcesz wyświetlić obiekt `XmlDocument` w postaci hierarchii obiektów, możesz przekazać go do polecenia `Format-Custom`.

Zadanie 8.

Wyświetl listę wszystkich dzienników zdarzeń dostępnych na Twoim komputerze, zawierającą nazwę dziennika, jego maksymalny rozmiar i operację wykonywaną w razie przepełnienia, a następnie przekonwertuj plik na format CSV, ale bez zapisywania w pliku na dysku. Do wykonania tego zadania może być konieczne znalezienie innego polecenia związanego z CSV.

Zadanie 9.

Wyświetl listę usług zawierającą tylko skrócone nazwy usług, pełne nazwy (ang. *display name*) oraz statusy i zapisz te informacje w postaci strony HTML zatytułowanej „Raport”. Przed tabelą usług zamieść nagłówek „Lista zainstalowanych usług”. Jeżeli potrafisz, wyświetl ten nagłówek za pomocą znacznika `<h1>`. Użyj przeglądarki internetowej do sprawdzenia, czy strona jest wyświetlana poprawnie.

Zadanie 10.

Utwórz nowy alias o nazwie `D`, który uruchamia program `Get-ChildItem`. Wyeksportuj ten alias do pliku. Teraz zamknij powłokę i otwórz nowe okno konsoli. Zimportuj nowo utworzony alias do powłoki. Upewnij się, że możesz uruchomić polecenie `D` i uzyskać na ekranie listę katalogów.

Zadanie 11.

Wyświetl listę zainstalowanych poprawek, które są oznaczone jako poprawki (ang. *hot-fix*) albo aktualizacje (ang. *update*), ale nie są aktualizacjami zabezpieczeń (ang. *security update*).

Zadanie 12.

Uruchom polecenie, które wyświetli nazwę bieżącego katalogu roboczego powłoki.

Zadanie 13.

Uruchom polecenie, które wyświetli listę poleceń ostatnio uruchomionych w powłoce. Znajdź polecenie, które uruchomiłeś dla zadania 11. Używając dwóch poleceń połączonych potokiem, ponownie uruchom komendę z zadania 11.

Podpowiedź: jeżeli `Pobierz-Cośtam` jest poleceniem pobierającym polecenia historyczne, 5 to numer identyfikacyjny polecenia z zadania 11., a `Wykonaj-Cośtam` to polecenie uruchamiające komendy historyczne, to takie zadanie możesz wykonać w następujący sposób:

```
Pobierz-Cośtam id 5 | Wykonaj-Cośtam
```

Oczywiście nie są to poprawne nazwy cmdletów — właściwe polecenia musisz znaleźć samodzielnie.

Wskazówka: oba polecenia, których potrzebujesz, mają w nazwach ten sam rzeczownik.

Zadanie 14.

Uruchom polecenie, które modyfikuje dziennik zdarzeń *Security* tak, aby w razie przepełnienia nadpisywał najstarsze zdarzenia.

Zadanie 15.

Użyj polecenia `New-Item` do utworzenia nowego katalogu o nazwie `C:\Review`. To nie jest to samo co uruchomienie polecenia `mkdir`. Polecenie `New-Item` będzie musiało wiedzieć, jaki rodzaj nowego elementu chcesz utworzyć. Zapoznaj się z zawartością pliku pomocy dla tego polecenia.

Zadanie 16.

Wyświetl zawartość następującego klucza rejestru:

HKCU:\Software\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders

Uwaga: klucz User Shell Folders nie przypomina katalogu. Jeżeli z poziomu powłoki przejdziesz do tego „katalogu” i spróbujesz wyświetlić jego zawartość, to nic nie zobaczysz. Klucz User Shell Folders to *element* (ang. *item*) rejestru, a wartości w nim zapisane to *właściwości elementu* (ang. *item properties*). W powłoce PowerShell dostępny jest cmdlet, który może wyświetlać właściwości elementów (pamiętaj, że w nazwach cmdletów powłoki rzeczowniki występują w liczbie pojedynczej, a nie mnogiej).

Zadanie 17.

Znajdź (ale nie uruchamiaj) polecenia, za pomocą których możesz:

- uruchomić ponownie komputer,
- wyłączyć komputer,
- usunąć komputer z grupy roboczej lub domeny,
- przywrócić stan komputera do wybranego punktu przywracania systemu (ang. *System Restore Point*).

Zadanie 18.

Jakiego polecenia możesz użyć do zmiany wartości w rejestrze? Podpowiedź: takie polecenie ma w nazwie ten sam rzeczownik co polecenie dla zadania 16.

Zestaw 2.: rozdziały 1. – 14.

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej. Przed przystąpieniem do wykonywania tego zestawu ćwiczeń powinieneś wykonać wszystkie ćwiczenia w rozdziałach od 1. do 14. tej książki.

Podpowiedzi

- Format-Table
- Invoke-Command
- Get-Content (**lub** Type)
- polecenia w nawiasach
- @{label='column_header';expression={\$_.property}}
- Get-WmiObject
- Where-Object
- -eq -ne -like -notlike

Zadanie 1.

Wyświetl w postaci tabeli listę uruchomionych procesów, zawierającą tylko nazwy procesów oraz ich identyfikatory ID. Tabela nie powinna mieć dużego, pustego obszaru między kolumnami.

Zadanie 2.

Uruchom poniższe polecenie:

```
Get-WmiObject -class win32_systemdriver
```

Następnie ponownie uruchom to samo polecenie, ale sformatuj dane wyjściowe do postaci listy, która wyświetla krótką nazwę sterownika, jego pełną nazwę (ang. *display name*), ścieżkę do pliku sterownika, tryb uruchamiania i bieżący stan. Nazwa właściwości reprezentującej ścieżkę powinna być wyświetlana jako Path, a nie jako rzeczywista nazwa tej właściwości.

Zadanie 3.

Uruchom następujące polecenie:

```
Get-PSProvider
```

na dwóch komputerach zdalnych (w razie potrzeby możesz dwukrotnie użyć adresu localhost). Do wykonania zadania użyj poleceń komunikacji zdalnej. Upewnij się, że w wynikach działania wyświetlane są nazwy użytych komputerów zdalnych.

Zadanie 4.

Użyj Notatnika do utworzenia pliku o nazwie *C:\Computers.txt*. W pliku umieść następujące elementy:

```
Localhost  
localhost
```

W każdym wierszu umieść po jednym elemencie (czyli w sumie plik powinien mieć dwa wiersze). Zapisz plik na dysku i zamknij Notatnik. Następnie napisz polecenie, które wyświetli listę uruchomionych usług działających na komputerach zdalnych, których nazwy znajdują się w pliku *C:\Computers.txt*.

Zadanie 5.

Utwórz zapytanie WMI wyświetlające wszystkie instancje klasy Win32_LogicalDisk, które mają typ dysku równy 3 (*drivetype=3*). Wyświetl literę dysku, jego rozmiar, wolną przestrzeń i procent wolnego miejsca. Kiedy to zadziała, zmodyfikuj polecenie tak, aby wyświetlało tylko te instancje, które mają co najmniej 50 procent wolnego miejsca na dysku (jeżeli nie masz tyle wolnego miejsca na dyskach swojego komputera, możesz odpowiednio dostosować procentowy rozmiar wolnego miejsca, tak aby zapytanie działało).

Wskazówka: aby obliczyć procentową ilość wolnego miejsca, powinieneś skorzystać z formuły *freespace/size*100*.

Pamiętaj, że parametr *-Filter* polecenia *Get-WmiObject* nie może zawierać wyrażeń matematycznych.

Zadanie 6.

Korzystając z poleceń CIM, wyświetl listę wszystkich klas WMI w przestrzeni *root\CIMv2*, których nazwy zaczynają się od ciągu znaków *win32*.

Zadanie 7.

Wyświetl listę wszystkich instancji klasy Win32_Service, dla których właściwość Start-Mode jest ustawiona na Auto i właściwość State ma wartość inną niż Running.

Zadanie 8.

Znajdź polecenie, za pomocą którego możesz wysyłać wiadomości e-mail. Jakie są obligatoryjne parametry tego polecenia?

Zadanie 9.

Uruchom polecenie, które wyświetli uprawnienia dla katalogu C:\. Wyniki działania będą bardziej czytelne, jeżeli je sformatujesz do postaci listy.

Zadanie 10.

Uruchom polecenie, które wyświetli uprawnienia dla każdego podkatalogu C:\Users. Sprawdź uprawnienia tylko dla bezpośrednich podkatalogów; nie musisz zagłębiać się do wszystkich plików i podkatalogów. Aby wykonać takie zadanie, musisz przesłać wyniki działania jednego polecenia do innego. Następnie zmodyfikuj polecenie tak, aby wyświetlało uprawnienia dla ukrytych podkatalogów.

Zadanie 11.

Odszukaj polecenie, które może uruchomić Notatnik. Użyj innych poświadczeń logowania niż te, których użyłeś do zalogowania się w powłoce.

Zadanie 12.

Uruchom polecenie, które spowoduje zatrzymanie działania powłoki przez 10 sekund.

Zadanie 13.

Odszukaj plik lub pliki pomocy objaśniające przeznaczenie i sposób użycia różnych operatorów wykorzystywanych przez powłokę PowerShell.

Zadanie 14.

Korzystając z polecenia Get-Winevent, wyświetl listę wszystkich dzienników zdarzeń, w których znajdują się jakieś wpisy, i posortuj wyniki w porządku malejącym według liczby rekordów.

Zadanie 15.

Uruchom następujące polecenie:

```
Get-CimInstance -Classname Win32_Processor
```

Przyjrzyj się domyślnym wynikom jego działania. Zmodyfikuj polecenie tak, aby wyniki działania były wyświetlane w tabeli, która powinna zawierać liczbę rdzeni procesora, producenta i nazwę. Dołącz także kolumnę o nazwie MaxSpeed, w której będzie wyświetlana maksymalna prędkość zegara procesora.

Zadanie 16.

Uruchom następujące polecenie:

```
Get-CimInstance -Classname Win32_Process
```

Przyjrzyj się domyślnym wynikom jego działania (jeżeli chcesz, możesz je również przekazać za pomocą potoku do polecenia `Get-Member`). Zmodyfikuj to polecenie tak, aby pobierało tylko procesy, których wartość właściwości `PeakWorkingSetSize` jest większa niż 100 000, i wyświetlało nazwę procesu, jego ścieżkę i wszystkie właściwości, których nazwy rozpoczynają się od ciągu znaków *Peak*.

Zadanie 17.

Jeżeli używasz powłoki PowerShell v5 lub nowszej, zastosuj polecenie `Find-Module` do odszukania pakietów ze znacznikiem `Network`. Wyświetl nazwy znalezionych modułów, ich wersje i opisy.

Zestaw 3.: rozdziały 1. – 19.

UWAGA Do wykonania opisanych niżej ćwiczeń potrzebny Ci będzie dowolny komputer z zainstalowaną powłoką PowerShell w wersji 3 lub nowszej. Przed przystąpieniem do wykonywania tego zestawu ćwiczeń powinieneś wykonać wszystkie ćwiczenia w rozdziałach od 1. do 19. tej książki. Rozpocznij od udzielenia odpowiedzi na następujące pytania:

1. Jakiego polecenia użyjesz do uruchomienia zadania, które działa wyłącznie na Twoim komputerze lokalnym?
2. Jakiego polecenia użyjesz do uruchomienia zadania, które będzie koordynowane przez Twój komputer, ale wykonywane przez komputery zdalne?
3. Czy ``${nazwa_komputera}` to poprawna nazwa zmiennej?
4. Jak wyświetlić listę wszystkich zmiennych aktualnie zdefiniowanych w powłoce?
5. Jakiego polecenia można użyć, aby poprosić użytkownika o wprowadzenie danych wejściowych?
6. Jakie polecenie można zastosować do uzyskania wyników działania, które normalnie są wyświetlane na ekranie, ale które można łatwo zamienić na inne formaty?

Jeżeli odpowiedziałeś już na wszystkie pytania, możesz rozpocząć wykonywanie zadań przedstawionych poniżej.

Zadanie 1.

Utwórz listę uruchomionych procesów. Lista powinna zawierać tylko nazwę procesu, ID, VM i kolumny PM. Wartości VM i PM powinny być wyświetlane w megabajtach. Zapisz listę w pliku w formacie HTML — o nazwie `C:\procs.html`. Nadaj stronie HTML tytuł *Lista bieżących procesów*. Wyświetl plik w przeglądarce i upewnij się, że tytuł pojawia się na pasku tytułu okna przeglądarki. Formuła obliczania zużycia pamięci w megabajtach, wyświetlanego w postaci liczby całkowitej, może dla właściwości VM

mieć taką postać: `$_VM / 1MB as [int]`. Po utworzeniu pliku sprawdź, czy możesz go ponownie zaimportować do powłoki PowerShell.

Zadanie 2.

Używając poleceń WMI lub CIM, utwórz plik rozdzielany tabulatorami, o nazwie `C:\Services.tdf`, który zawiera listę wszystkich usług zainstalowanych na Twoim komputerze. Pamiętaj, że znak tabulatora w poleceniach powłoki PowerShell jest reprezentowany przez ciąg znaków `"`t"` (odwrócony apostrof ``` i litera `t`, ujęte w cudzysłów). Lista usług powinna zawierać skróconą nazwę usługi, pełną nazwę usługi, status, tryb uruchamiania i nazwę konta używanego do jej uruchomienia.

Zadanie 3.

Napisz polecenie, które poprosi użytkownika o wpisanie nazwy komputera i zapisze podaną nazwę w zmiennej. Następnie napisz kolejne polecenie, które za pomocą cmdletów CIM zapyta podany w zmiennej komputer o nazwę systemu operacyjnego, numer wersji, datę ostatniego uruchomienia i czas działania (ang. *uptime*). Nie zapomnij o umieszczeniu nazwy komputera w wynikach działania. Czas działania możesz obliczyć, odejmując czas ostatniego uruchomienia od aktualnej daty i godziny.

Zadanie 4.

Weź polecenia z poprzedniego zadania i przekształć je w sparametryzowany skrypt. Mamy nadzieję, że jest dla Ciebie oczywiste, iż nazwa komputera jest dobrym kandydatem na parametr. Dodaj alias `CN` i zdefiniuj parametr jako obligatoryjny. Wyniki działania powinny pokazywać te same właściwości co w poprzednim zadaniu, ale warto chyba pomyśleć o lepszej nazwie dla właściwości reprezentującej nazwę systemu operacyjnego.

Zadanie 5.

Korzystając z poleceń WMI i klasy `Win32_Product`, znajdź wszystkie produkty zainstalowane na Twoim komputerze. Wykonanie takiego polecenia może chwilę potrwać, więc uruchom go jako zadanie w tle. Po zakończeniu działania polecenia pobierz wyniki i wyświetl nazwę produktu, firmę, wersję, datę instalacji i lokalizację instalacji w widoku `GridView`.

Odpowiedzi

Zestaw 1.

Zadanie 1.

```
Get-EventLog -LogName Security -Newest 100
```

Zadanie 2.

```
Get-Process | Sort -Property VM -Descending | Select -First 5
```

Zadanie 3.

```
Get-Service | Select -Property Name,Status |  
    Sort -Property Status -Descending |  
    Export-CSV services.csv
```

Zadanie 4.

```
Set-Service -Name BITS -StartupType Automatic
```

Zadanie 5.

```
Get-ChildItem -Path C:\ -Recurse -file -Filter 'Win*.*'
```

Zadanie 6.

```
Get-ChildItem -Path 'c:\program files' -recurse | Out-File c:\dir.txt
```

Zadanie 7.

```
Get-EventLog -LogName Security -Newest 20 | ConvertTo-XML
```

Zadanie 8.

```
Get-EventLog -list | Select Log,MaximumKilobytes,OverflowAction | convertto-csv
```

Zadanie 9.

```
Get-Service | Select -Property Name,DisplayName,Status |  
ConvertTo-HTML -PreContent "<H1>Lista zainstalowanych usług</H1>" -title  
"Raport" | Out-File c:\services.html
```

Zadanie 10.

```
New-Alias -Name D -Value Get-ChildItem -PassThru | Export-Alias c:\alias.xml
```

Po uruchomieniu nowego okna powłoki PowerShell:

```
Import-Alias c:\alias.xml  
D
```

Zadanie 11.

```
get-hotfix -description "Update","Hotfix"
```

Zadanie 12.

```
Get-Location lub jego alias pwd.
```

Zadanie 13.

```
Get-History
```

Po uruchomieniu tego cmdletu zlokalizuj polecenie, które uruchomiłeś dla zadania 11. Będziesz potrzebował jego numeru identyfikacyjnego, który wstawiś zamiast x w następnym poleceniu:

```
Get-History -id x | Invoke-History
```

Zadanie 14.

```
Limit-EventLog -LogName Security -OverwriteAction OverwriteAsNeeded
```

Zadanie 15.

```
New-Item -Name C:\Review -Type Directory
```

Zadanie 16.

```
Get-ItemProperty -Path  
'HKCU:\Software\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders'
```

Zadanie 17.

- Restart-Computer
- Stop-Computer
- Remove-Computer
- Restore-Computer

Zadanie 18.

```
Set-ItemProperty
```

Zestaw 2.**Zadanie 1.**

```
Get-Process | Format-Table -Property Name,ID -AutoSize
```

Zadanie 2.

```
Get-WmiObject -class win32_systemdriver | select -property Name,Displayname,  
@{Name="Path";Expression={$_.pathname}},StartMode,State
```

Zadanie 3.

```
Invoke-Command -ScriptBlock { Get-PSProvider } -computerName Computer1,Computer2
```

Zadanie 4.

```
Get-Service -computerName (Get-Content C:\Computers.txt)
```

Zadanie 5.

```
Get-WmiObject -class Win32_LogicalDisk -Filter "drivetype=3" |  
Select DeviceID,Size,Freespace,  
@{Name="PctFree";Expression = {($_.freespace/$_.size)*100}} |  
where {$_.PctFree -gt 50}
```

Zadanie 6.

```
get-cimclass -classname win32*
```

Zadanie 7.

```
Get-WmiObject -class Win32_Service -filter "StartMode='Auto' AND State<>'Running'"
```

Polecenie przedstawione poniżej również będzie działać, ale nie jest najlepszą praktyką, ponieważ nie filtruje danych tak wcześnie, jak to możliwe.

```
Get-WmiObject -class Win32_Service |  
Where-Object { $_.StartMode -eq 'Auto' -and $_.State -ne 'Running' }
```

Zadanie 8.

Send-MailMessage (listę parametrów obligatoryjnych znajdziesz w pliku pomocy tego polecenia).

Zadanie 9.

```
Get-ACL -Path C:\ | format-list
```

Zadanie 10.

```
Get-ChildItem -path C:\Users | Get-ACL  
Get-ChildItem -path C:\Users -Directory -Hidden | Get-ACL
```

Zadanie 11.

```
Start-Process
```

Zadanie 12.

```
Start-Sleep -seconds 10
```

Zadanie 13.

```
Help *operators*
```

Zadanie 14.

```
get-winevent -ListLog * | where {$_.recordcount -gt 0} | sort RecordCount -Descending
```

Zadanie 15.

```
Get-CimInstance -classname Win32_Processor |  
Select-Object -property Manufacturer,NumberOfCores,Name,@{  
name='MaxSpeed';expression={$_.MaxClockSpeed}}
```

Zadanie 16.

```
Get-WmiObject -class Win32_Process -filter "PeakWorkingSetSize >= 100000" |  
Select Name,ExecutablePath,Peak*
```

Zadanie 17.

```
find-module -tag network | Sort Name | Select Name,Version,Description
```

Zestaw 3.

1. Start-Job
2. Invoke-Command
3. Tak

4. Read-Host
5. Write-Output

Zadanie 1.

```
Get-Process | Select-Object -property Name,ID,VM,PM |
ConvertTo-HTML -Title "Lista bieżących procesów" | Out-File C:\Procs.html
```

Zadanie 2.

```
Get-CimInstance -classname win32_service |
Select Name,State,StartMode,Startname |
Export-CSV c:\services.tdf -Delimiter "`t"
```

Następnie wypróbuj takie polecenie:

```
import-csv C:\services.tdf -Delimiter "`t"
```

Zadanie 3.

```
$computer = Read-Host "Wprowadź nazwę komputera"
Get-CimInstance -ClassName Win32_Operatingsystem -CimSession $computer |
Select Caption,Version,LastBootUptime,
@{Name="Uptime";Expression=((Get-Date) - $_.lastBootUpTime)},PSComputername
```

Zadanie 4.

Przykładowy kod skryptu może być taki:

```
[cmdletbinding()]
Param(
[Parameter(Mandatory=$True,HelpMessage = "Wprowadź nazwę komputera")]
[Alias("CN")]
[string]$Computername
)
Write-Verbose "Pobieranie informacji o systemie operacyjnym z komputera $Computername."
Get-CimInstance -ClassName Win32_Operatingsystem -CimSession $computername |
Select @{Name="OS";Expression={$_.Caption}},Version,LastBootUptime,
@{Name="Uptime";Expression=((Get-Date) - $_.lastBootUpTime)},
@{Name="Computername";Expression={$_.PSComputername}}
Write-Verbose "Gotowe."
```

Zadanie 5.

Najpierw utwórz zadanie na przykład w taki sposób:

```
get-wmiobject win32_product -asjob
```

Następnie pobierz wyniki działania:

```
$prod = Receive-job 31 -Keep
```

Na koniec przetwarzaj wyniki działania stosownie do potrzeb:

```
$prod | Select Name,Vendor,InstallDate,InstallLocation |
Out-GridView -Title "Moje produkty"
```

Skorowidz

A

- Active Directory, 94
- akcje, 135
- aktualizowanie systemu pomocy, 45
- aliasy, 69
 - parametrów, 354
- analiza skryptu, 404
- anatomia polecenia, 67
- aplikacja, application, 68
 - konsolowa, 38
- apostrof, 294
- atrybuty, 134
- automatyzacja zarządzania obiektami, 259

B

- bezpieczeństwo, 277
- bloki skryptu, 396
- błędy, 78

C

- certyfi­kat, 284, 369
- ciągi znaków, 391
- CIM, 232
- CLI, Command Line Interface, 43
- Cmdlety
 - CIM, 232
 - WMI, 232
- cudzysłów, 294, 301

D

- daty, 392
 - WMI, 394
- definiowanie
 - parametrów obligatoryjnych, 351
 - widoku tabeli, 168
- deklarowanie typu zmiennej, 303
- delegowanie, 367
- deserializacja, 219
- dodawanie
 - aliasów parametrów, 354
 - modułów, 117
 - przystawek, 115
- dokumentacja
 - metod, 273
 - skryptu, 339
 - WMI, 237
- domyślne wartości parametrów, 395
- dostawca, 83
- dostęp
 - do pomocy online, 60
 - do statycznej metody, 313
 - do tematów pomocy, 59
 - do właściwości obiektu, 300
- dostosowywanie znaku zachęty, 386
- dyski, 85
 - PSDrive, 91

E

eksportowanie wyników
 do pliku CSV, 98, 99
 do pliku XML, 98, 100
 dodawanie, 113
elementy
 nadrzędne, 344
 podrzędne, 86, 344

F

filtrowanie, 185
 obiektów, 189
 z lewej, 186, 193
foldery, 85
format HTML, 105
formatowanie
 danych, 179
 domyślne, 168
 list, 173
 tabel, 171
 wyników, 167
funkcje powłoki, 68

G

graficzne okno dialogowe, 74
graficzny interfejs użytkownika, GUI, 22

H

host pośredni, 367

I

IIS, Internet Information Server, 94
importowanie sesji, 327
instalowanie Windows PowerShell, 28
instancja, 227
IntelliSense, 39
interpunkcja, 413
ISE, Integrated Scripting
 Environment, 35

K

kolekcja, 130
kolor powłoki, 386
kolumny, 418
komputer
 badany, 101
 odniesienia, 101
komunikacja zdalna, 245, *patrz także* sesje,
 połączenie zdalne
 konfiguracja, 361
 niejawna, 327
 opcje, 221
 z wykorzystaniem hosta pośredniego,
 367
komunikat o błędzie, 78
konfiguracja
 komunikacji zdalnej, 361
 sesji, 206, 361, 363
 środowiska testowego, 27
konflikty poleceń, 120
konsola
 graficzna, 309
 MMC, 113, 369
 tekstowa, 309

L

listy
 formatowanie, 173
 niestandardowe elementy, 175
 szerokie, 174
luki w zabezpieczeniach, 287

Ł

łączenie poleceń, 97

M

mechanizm
 CIM, 262
 IntelliSense, 39
 WMI, 225, 262
mechanizmy bezpieczeństwa, 287

metody, 130, 135
 dokumentacja, 273
 WMI, 273
 wywoływanie, 262
 minipowłoka, 115
 MMC, Microsoft Management Console, 369
 model
 bezpieczeństwa, 278
 połączeń, 210
 moduł, 115, 117, 121
 ActiveDirectory, 155
 moduły
 dodawanie, 117
 pobieranie, 125
 problemy, 126
 używanie, 121
 wyszukiwanie, 117
 modyfikowanie
 systemu, 106
 ustawień kolorów, 386

N

nazwy skrócone poleceń, 69
 niestandardowe
 kolumny, 175, 418
 właściwości, 418

O

obiekty, 129, 259
 akcje, 135
 atrybuty, 134
 automatyzacja zarządzania, 259
 deserializacja, 219
 elementy składowe, 134
 metody, 135
 odkrywanie, 133
 przechowywane w zmiennej, 297
 sortowanie, 136
 używanie, 131
 właściwości, 134
 wybieranie żądanych właściwości, 137
 wylizanie, 266
 odkrywanie obiektów, 133

okno
 konsoli, 33
 środowiska ISE, 37
 opcje komunikacji zdalnej, 221
 operator, 417
 -as, 388
 -contains, 390
 -in, 390
 -is, 388
 -join, 389
 -Match, 376
 -replace, 389
 -split, 389
 operatory porównania, 187
 operowanie
 na ciągach znaków, 391
 na datach, 392
 na datach WMI, 394

P

pakowanie skryptów, 279
 parametr -computerName, 216
 parametry
 dodawanie aliasów, 354
 domyślne wartości, 395
 obligatoryjne, 52, 351
 opcjonalne, 52
 pozycyjne, 53, 72
 sprawdzanie wartości, 355
 wartości, 55
 wspólne, 51
 parametryzowanie poleceń, 336
 parsowanie plików tekstowych, 373
 pliki, 85
 HTML, 105
 pomocy, 51, 417
 wsadowe, 333
 pobieranie
 informacji, 309
 modułów, 125
 parametrów z potoku, 419
 wyników, 246
 podpisywanie kodu, 280, 283
 podwyrażenie, 302

- polecenia, 43, 65
 - aliasy, 69, 71
 - anatomia, 67
 - do pracy ze zmiennymi, 305
 - do zarządzania zadaniami, 251
 - konflikty, 120
 - lokalne, 216
 - parametryzowanie, 336
 - potoki, 97, 144
 - przekazywania danych, 146, 149
 - składnia, 214
 - skrótowe nazwy parametrów, 71
 - uruchamianie, 65
 - w nawiasach, 158
 - wielokrotnego użytku, 334
 - wpisywanie nazw, 79
 - wpisywanie parametrów, 80
 - wsadowe, 260
 - wyodrębnianie wartości, 159
 - wyszukiwanie, 198
 - zdalne, 216
 - zewnętrzne, 75
 - polecenie, command, 68
 - Invoke-Command, 213, 216
 - cmdlet, 68, 88, 106
 - Enter-PSSession, 210, 323
 - Exit-PSSession, 210
 - ForEach-Object, 274
 - Get-ADComputer, 160
 - Get-CimInstance, 237
 - Get-Help, 47
 - Get-Member, 133
 - Get-Process, 129
 - Get-WmiObject, 233
 - help, 101, 341
 - Invoke-Command, 213, 325
 - ise, 35
 - Man, 47
 - Out-GridView, 178
 - Read-Host, 310
 - Save-Help, 47
 - Select-String, 377
 - Show-Command, 73
 - Where-Object, 190, 192
 - Write-Host, 313
 - Write-Output, 315
 - polityka wykonywania skryptów, 280
 - połączenie zdalne
 - jeden-do-jednego, 210
 - jeden-do-wielu, 210, 213
 - tworzone ad hoc, 220
 - pomoc, *Patrz* system pomocy
 - porównywanie, 187
 - danych, 185
 - plików, 101
 - potoki, 97, 144, 340, 419
 - potokowe wiązanie parametrów, 146
 - PowerShell, 23, 31
 - PowerShell ISE, 36
 - problem
 - drugiego skoku, 367
 - wielu skoków, 367
 - problemy, 39, 79, 108, 141, 178, 192, 222, 237, 254, 271, 306
 - profile powłoki, 383
 - protokół
 - SSH, 204
 - SSL, 369
 - przechowywanie
 - wartości, 292
 - wielu obiektów, 297
 - przekazywanie danych
 - ByPropertyName, 149
 - ByValue, 146
 - przełącznik, 55
 - przepływ pracy, workflow, 68
 - przestrzeń nazw, 226
 - przesyłanie
 - danych, 178
 - wyników działania, 104
 - przetwarzanie
 - lokalne, 217
 - zdalne, 217
 - przystawka, 113, 115
 - Certyfikaty, 369
 - przystawki
 - dodawanie, 115
 - wyszukiwanie, 115
 - punkty końcowe, 206, 361, 362
- ## R
- reguła formatowania, 170
 - rejestrwanie sesji, 364
 - repozytorium WMI, 227
 - rodzaje rozszerzeń, 115

rozszerzenia, 115, 117
 usuwanie, 120
 rzutowanie, 388

S

sesje, 321
 importowanie, 327
 konfiguracja, 361, 363
 rejestrwanie, 364
 rozłączone, 328
 wielokrotnego użytku, 322
 składnia
 bloku Param(), 353
 poleceń, 214
 wyrażeń regularnych, 374
 skrypt, 66, 333
 New-WebProject.ps1, 400
 skrypty
 analiza wiersz po wierszu, 404
 bazowe, 349
 bloki, 396
 dokumentacja, 339
 sparametryzowane, 337
 profilu powłoki, 123, 383
 zdalne, 282
 sortowanie obiektów, 136
 sprawdzanie wersji powłoki, 40
 SQL Server, 94
 symbol
 zastępczy \$_, 193, 420
 wieloznaczny, 90
 system
 plików, 85, 87
 pomocy, 43
 aktualizowanie, 45
 dostęp do ogólnych tematów, 59
 online, 60
 wyszukiwanie poleceń, 48

Ś

ścieżka, 90
 środowisko
 PowerShell ISE, 36
 testowe, 27

T

tabele
 formatowanie, 171
 widok, 168
 TrustedHosts, 369
 tworzenie
 konfiguracji sesji, 363
 niestandardowych
 elementów list, 175
 kolumn, 175
 punktów końcowych, 362
 sesji, 322
 zadań lokalnych, 243
 typ wyjścia
 GridViews, 178

U

uprawnienia punktu końcowego, 366
 uruchamianie
 poleceń, 65
 powłoki, 32, 123
 urzędy certyfikacji, 285
 usługa
 WinRM, 206
 WMI, 228
 usuwanie rozszerzeń, 120
 uwierzytelnianie wzajemne, 368
 przez SSL, 369
 za pośrednictwem TrustedHosts, 369
 zaawansowane, 368
 używanie
 modułu, 121
 obiektów, 131
 symbolu zastępczego, 420
 zmiennych, 294

W

wartości parametrów, 55
 wcześnie filtrowanie, 185
 wejście i wyjście, 309
 wersja powłoki, 40
 weryfikacja, 286
 wielozadaniowość, 241

- wiersz poleceń, 38
 - model iteracyjny, 190
- właściwości, 130, 134, 418
 - niestandardowe, 155
- WMI, Windows Management
 - Instrumentation, 225
 - dokumentacja, 237
 - eksplorowanie, 230
 - elementy, 226
 - repozytorium, 227
 - wywoływanie metod, 262
- wstępne ładowanie rozszerzeń, 123
- wyliczanie
 - obiektów, 266
 - parametrów, 38
- wyodrębnianie wartości, 159
- wyrażenia regularne, 373, 374
 - polecenie Select-String, 377
 - składnia, 374
 - z operatorem -Match, 376
- wyszukiwanie
 - modułów, 117
 - parametrów pozycyjnych, 54
 - poleceń, 48, 58, 198
 - przystawek, 115
- wyświetlanie
 - danych, 317
 - informacji, 309
 - wyników szczegółowych, 356
- wywoływanie metod, 262
- zakończenie działania procesów, 106
- zalecenia bezpieczeństwa, 288
- zapytania WMI, 244
- zarządzanie
 - masowe obiektami, 259
 - produktem, 114
 - zadaniami, 251
- zasięg, scope, 344, 396
 - globalny, 344
 - prywatny, 344
 - skryptu, 344
- zasoby, 411
- zatrzymywanie usług, 106
- zdalna sesja powłoki, 204
- zestawy parametrów, 51
- zintegrowane środowisko skryptowe, ISE, 35
- zmiennie, 291
 - deklarowanie typu, 303
 - dobre praktyki, 306
 - polecenia, 305
 - przechowywanie
 - wartości, 292
 - wielu obiektów, 297
- znak zachęty, 386

Z

- zadania
 - asynchroniczne, 242
 - działające w tle, 244–246
 - podrzędne, 244, 249
 - synchroniczne, 242
 - zaplanowane, 253

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



IT



BIZNES



PROJEKTY



PROCESY

NASZE SZKOLENIA SĄ PROWADZONE
ZGODNIE Z METODĄ

BLENDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie
z dostępem do nowoczesnych narzędzi - wideokursów,
e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

WWW.HELIONSZKOLENIA.PL