Microsoft

Ed Wilson

Windows PowerShell® 3.0
Krok po kroku

Przekład: Marek Włodarz

APN Promise, Warszawa 2013

Windows PowerShell® 3.0 Krok po kroku
© 2013 APN PROMISE SA

Authorized Polish translation of the English edition Windows PowerShell® 3.0 Step by Step, ISBN 978-0-735-66339-8
Copyright © 2013 by Ed Wilson

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

APN PROMISE SA, biuro: ul. Kryniczna 2, 03-934 Warszawa
tel. +48 22 35 51 600, fax +48 22 35 51 699
e-mail: mspress@promise.pl

Wszystkie prawa zastrzeżone. Żadna część niniejszej książki nie może być powielana ani rozpowszechniana w jakiejkolwiek formie i w jakikolwiek sposób (elektroniczny, mechaniczny), włącznie z fotokopiowaniem, nagrywaniem na taśmy lub przy użyciu innych systemów bez pisemnej zgody wydawcy.

Książka ta przedstawia poglądy i opinie autorów. Przykłady firm, produktów, osób i wydarzeń opisane w niniejszej książce są fikcyjne i nie odnoszą się do żadnych konkretnych firm, produktów, osób i wydarzeń, chyba że zostanie jednoznacznie stwierdzone, że jest inaczej. Ewentualne podobieństwo do jakiejkolwiek rzeczywistej firmy, organizacji, produktu, nazwy domeny, adresu poczty elektronicznej, logo, osoby, miejsca lub zdarzenia jest przypadkowe i niezamierzone.

Nazwa Microsoft oraz znaki towarowe wymienione na stronie *http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx* są zastrzeżonymi znakami towarowymi grupy Microsoft. Wszystkie inne znaki towarowe są własnością ich odnośnych właścicieli.

APN PROMISE SA dołożyła wszelkich starań, aby zapewnić najwyższą jakość tej publikacji. Jednakże nikomu nie udziela się rękojmi ani gwarancji.
APN PROMISE SA nie jest w żadnym wypadku odpowiedzialna za jakiekolwiek szkody będące następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeśli APN PROMISE została powiadomiona o możliwości wystąpienia szkód.

ISBN: 978-83-7541-119-5

Przekład: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Spis treści

Przedmowa xi

Wprowadzenie xiii

1 Przegląd cech Windows PowerShell 3.0 1

Istota Windows PowerShell 1
Korzystanie z poleceń cmdlet 3
Instalowanie Windows PowerShell 3
Wdrażanie Windows PowerShell w starszych systemach operacyjnych 4
Korzystanie z narzędzi wiersza poleceń 5
Problemy dotyczące zabezpieczeń 7
Kontrolowanie wykonywania poleceń cmdlet 7
Potwierdzanie akcji 8
Wstrzymywanie potwierdzania poleceń 9
Posługiwanie się Windows PowerShell 10
Wywoływanie Windows PowerShell 10
Konfigurowanie konsoli Windows PowerShell 11
Przekazywanie opcji do poleceń cmdlet 12
Korzystanie z opcji pomocy 13
Poznawanie poleceń: ćwiczenia krok po kroku 20
Podsumowanie rozdziału 1 22

2 Korzystanie z poleceń cmdlet 23

Podstawy poleceń cmdlet 23
Korzystanie z cmdlet *Get-ChildItem* 24
Uzyskiwanie listingu katalogu 24
Formatowanie listingu katalogu przy użyciu polecenia *Format-List* *26*
Korzystanie z polecenia cmdlet *Format-Wide* *26*
Formatowanie listingu katalogu przy użyciu *Format-Table* *28*
Formatowanie danych wyjściowych przy użyciu *Out-GridView* 30
Korzystanie z polecenia *Get-Command* 35
Wyszukiwanie poleceń cmdlet za pomocą symboli wieloznacznych 35
Korzystanie z polecenia *Get-Member* 40
Korzystanie z polecenia *Get-Member* do badania właściwości i metod 40
Korzystanie z polecenia *New-Object* 45
Tworzenie i korzystanie z obiektu *wshShell* *46*
Korzystanie z polecenia *Show-Command* 47
Jak nazewnictwo poleceń cmdlet pomaga w nauce 49

Grupowanie czasowników Windows PowerShell 50
Dystrybucja czasowników Windows PowerShell 51
Tworzenie profilu Windows PowerShell 52
Wyszukiwanie wszystkich aliasów dla określonego obiektu 54
Stosowanie poleceń cmdlet: ćwiczenia krok po kroku 55
Podsumowanie rozdziału 2 59

3 Dostawcy PowerShell 61
Istota dostawców PowerShell 61
Dostawca aliasów 62
Dostawca certyfikatów 64
Dostawca środowiska 72
Dostawca systemu plików 76
Dostawca funkcji 80
Korzystanie z dostawcy rejestru do zarządzania zawartością rejestru systemu Windows 82
Dwa dyski rejestru 83
Dostawca zmiennych 93
Poznawanie dostawców PowerShell: ćwiczenia krok po kroku 98
Podsumowanie rozdziału 3 101

4 Korzystanie z funkcji zdalnych i zadań PowerShell 103
Funkcje zdalne Windows PowerShell 103
Klasyczna praca zdalna 103
WinRM 108
Korzystanie z zadań Windows PowerShell 116
Korzystanie z mechanizmów zdalnych Windows PowerShell: ćwiczenia krok po kroku 124
Podsumowanie rozdziału 4 128

5 Używanie skryptów PowerShell 129
Po co pisać skrypty Windows PowerShell? 129
Podstawy skryptowania 131
Jak uruchomić skrypt Windows PowerShell? 131
Włączanie obsługi skryptów w Windows PowerShell 132
Przechodzenie z wiersza poleceń do skryptu 134
Uruchamianie skryptów Windows PowerShell 137
Zmienne i stałe 140
Stałe 145
Korzystanie z wyrażenia *While* 146
Budowanie wyrażenia *While* w PowerShell 146
Praktyczny przykład wykorzystania wyrażenia *While* *148*
Używanie specjalnych funkcji Windows PowerShell 149
Korzystanie z wyrażenia *Do...While* 149

Stosowanie operatora zakresu . . . 150
Działania na tablicach . . . 150
Rzutowanie znaków na wartości ASCII i odwrotnie . . . 151
Korzystanie z wyrażenia *Do...Until* . . . 152
Porównanie konstrukcji *Do...Until* w PowerShell i w VBScript . . . 152
Stosowanie wyrażenia *Do* w Windows PowerShell . . . 153
Wyrażenie *For* . . . 154
Wyrażenie *For* w PowerShell . . . 155
Korzystanie z wyrażenia *Foreach* . . . *157*
Przedterminowe opuszczanie wyrażenia *Foreach* . . . *158*
Wyrażenie *If* . . . 160
Ocenianie wielu warunków . . . 162
Wyrażenie *Switch* . . . 163
Korzystanie z wyrażenia *Switch* . . . *164*
Kontrolowanie dopasowywania . . . 166
Tworzenie wielu folderów: ćwiczenia krok po kroku . . . 167
Podsumowanie rozdziału 5 . . . 170

6 Praca z funkcjami . . . 171
Czym są funkcje . . . 171
Wykorzystanie funkcji w celu ułatwienia ponownego użycia kodu . . . 179
Korzystanie z techniki *dot-sourcing* . . . *179*
Korzystanie z funkcji dołączonych . . . 183
Stosowanie dwóch parametrów wejściowych . . . 184
Stosowanie ograniczeń typów w funkcjach . . . 188
Używanie więcej niż dwóch parametrów wejściowych . . . 191
Dołączanie pomocy do funkcji . . . 193
Korzystanie z obiektu *here-string* . . . *193*
Wykorzystanie funkcji do kapsułkowania logiki biznesowej . . . 196
Wykorzystanie funkcji w celu ułatwienia modyfikowania kodu . . . 198
Istota filtrów . . . 203
Tworzenie funkcji: ćwiczenia krok po kroku . . . 208
Podsumowanie rozdziału 6 . . . 211

7 Tworzenie zaawansowanych funkcji i modułów . . . 213
Atrybut *[cmdletbinding]* . . . 213
Łatwe komunikaty szczegółowe . . . 214
Automatyczne sprawdzanie parametrów . . . 215
Dodawanie obsługi parametru *-whatif* . . . *218*
Obsługa przełącznika *-confirm* . . . *219*
Specyfikowanie domyślnego zbioru parametrów . . . 220
Atrybut *parameter* . . . 221
Właściwość *mandatory* . . . *222*

Właściwość *position* *223*
Właściwość *ParameterSetName* *224*
Właściwość *ValueFromPipeline* *225*
Właściwość *HelpMessage* *226*
Moduły 227
Lokalizowanie i ładowanie modułów 227
Wyświetlanie dostępnych modułów 227
Ładowanie modułów 230
Instalowanie modułów 232
Tworzenie folderu Modules użytkownika 232
Tworzenie dysku modułów 237
Sprawdzanie zależności modułów 239
Używanie modułu z udziału sieciowego 242
Tworzenie modułu 244
Tworzenie zaawansowanej funkcji: ćwiczenia krok po kroku 250
Podsumowanie rozdziału 7 254

8 Korzystanie z Windows PowerShell ISE 255
Uruchamianie Windows PowerShell ISE 255
Poruszanie się po narzędziu Windows PowerShell ISE 256
Korzystanie z panelu skryptu 259
Dopełnianie tabulatorem i IntelliSense 260
Korzystanie ze wstawek kodu w Windows PowerShell ISE 262
Używanie wstawek Windows PowerShell ISE do tworzenia kodu 262
Tworzenie nowych wstawek Windows PowerShell ISE 264
Usuwanie wstawek zdefiniowanych przez użytkownika 265
Korzystanie z przystawki Commands: ćwiczenia krok po kroku 266
Podsumowanie rozdziału 8 269

9 Stosowanie profili Windows PowerShell 271
Sześć profili PowerShell 271
Istota różnych profili Windows PowerShell 272
Badanie zmiennej *$profile* *272*
Ustalanie, czy konkretny profil istnieje 274
Tworzenie nowego profilu 275
Uwarunkowania projektowe profili 275
Korzystanie z jednego lub więcej profili 277
Korzystanie z profilu *All Users, All Hosts* *279*
Korzystanie z własnego pliku 280
Grupowanie zbliżonej funkcjonalności w modułach 282
Gdzie umieścić moduł profilu 282
Tworzenie profilu: ćwiczenia krok po kroku 283
Podsumowanie rozdziału 9 286

10 Korzystanie z WMI 287
Istota modelu WMI 288
Posługiwanie się obiektami i przestrzeniami nazw 288
Wyliczanie dostawców WMI 293
Korzystanie z klas WMI 294
Odpytywanie WMI 298
Uzyskiwanie informacji o usługach: ćwiczenia krok po kroku 303
Podsumowanie rozdziału 10 310

11 Odpytywanie WMI 311
Alternatywne metody łączenia się z WMI 311
Selektywne odczytywanie danych ze wszystkich instancji 320
Wybieranie wielu właściwości 321
Wybieranie konkretnych instancji 325
Stosowanie operatorów 326
Gdzie jest *where*? 330
Skracanie składni 330
Praca z oprogramowaniem: ćwiczenia krok po kroku 333
Podsumowanie rozdziału 11 340

12 Zdalne kwerendy WMI 341
Używanie WMI wobec systemów zdalnych 341
Dostarczanie alternatywnych poświadczeń dla połączenia zdalnego 343
Wykorzystanie mechanizmu zdalnego Windows PowerShell do wywoływania WMI 346
Wykorzystanie CIM do odpytywania klas WMI 347
Praca ze zdalnymi rezultatami 349
Redukowanie rozmiaru danych za pomocą argumentów Windows PowerShell 352
Uruchamianie zadań WMI 353
Korzystanie z mechanizmów zdalnych Windows PowerShell WMI: ćwiczenia krok po kroku 356
Podsumowanie rozdziału 12 358

13 Wywoływanie metod w klasach WMI 359
Wykorzystywanie poleceń cmdlet WMI do wykonywania metod instancji 359
Bezpośrednie stosowanie metody *terminate* *361*
Korzystanie z polecenia cmdlet *Invoke-WmiMethod* *363*
Korzystanie z akceleratora typu *[wmi]* *364*
Wykorzystanie WMI do pracy z metodami statycznymi 365
Wykonywanie metod instancji: ćwiczenia krok po kroku 369
Podsumowanie rozdziału 13 371

14 Korzystanie z poleceń CIM . . . 373
Eksplorowanie klas WMI przy użyciu poleceń CIM . . . 373
Korzystanie z argumentu *-classname* . . . *374*
Wyszukiwanie metod klas WMI . . . 374
Filtrowanie klas przy użyciu kwalifikatora . . . 375
Odczytywanie instancji klas WMI . . . 377
Redukowanie liczby zwracanych właściwości i instancji . . . 378
Czyszczenie wyjścia polecenia . . . 379
Praca ze skojarzeniami . . . 380
Odczytywanie instancji klas WMI: ćwiczenia krok po kroku . . . 385
Podsumowanie rozdziału 14 . . . 388

15 Praca z Active Directory . . . 389
Tworzenie obiektów w Active Directory . . . 389
Tworzenie OU . . . 390
Dostawcy ADSI . . . 391
Nazwy LDAP . . . 394
Tworzenie użytkowników . . . 400
Czym jest kontrola konta użytkownika? . . . 403
Praca z użytkownikami . . . 404
Tworzenie wielu jednostek organizacyjnych: ćwiczenia krok po kroku . . . 419
Podsumowanie rozdziału 15 . . . 424

16 Korzystanie z modułu AD DS . . . 425
Istota modułu Active Directory . . . 425
Instalowanie modułu Active Directory . . . 425
Wprowadzenie do modułu Active Directory . . . 427
Korzystanie z modułu Active Directory . . . 427
Wyszukiwanie wzorców operacji . . . 429
Poznawanie struktury Active Directory . . . 434
Przemianowywanie lokacji Active Directory . . . 437
Zarządzanie jednostkami organizacyjnymi, użytkownikami i grupami . . . 438
Tworzenie użytkownika . . . 441
Wyszukiwanie i odblokowywanie kont użytkowników Active Directory . . . 443
Selektywne wybieranie kont użytkowników . . . 445
Wyszukiwanie nieużywanych kont użytkowników . . . 446
Aktualizowanie obiektów Active Directory: ćwiczenia krok po kroku . . . 449
Podsumowanie rozdziału 16 . . . 452

17 Wdrażanie AD DS w systemie Windows Server 2012 . . . 453
Wdrażanie nowego lasu przy użyciu modułu Active Directory . . . 453
Dodawanie nowego kontrolera domeny do istniejącej domeny . . . 460
Dodawanie kontrolera domeny tylko do odczytu . . . 462
Instalowanie kontrolera domeny: ćwiczenia krok po kroku . . . 464

Podsumowanie rozdziału 17 467

18 Debugowanie skryptów 469

Debugowanie w Windows PowerShell 469
Trzy rodzaje błędów 470
Korzystanie z polecenia cmdlet *Set-PSDebug* 476
Śledzenie skryptu 476
Krokowe wykonywanie skryptu 480
Włączanie trybu ścisłego 487
Korzystanie z polecenia *Set-PSDebug -Strict* *487*
Korzystanie z polecenia *Set-StrictMode* *488*
Debugowanie skryptu 490
Ustawianie punktów wstrzymania 491
Ustawianie punktu wstrzymania według numeru wiersza 491
Ustawianie punktu wstrzymania dla zmiennej 493
Ustawianie punktu wstrzymania dla polecenia 497
Wyliczanie punktów wstrzymania 499
Włączanie i wyłączanie punktów wstrzymania 501
Usuwanie punktów wstrzymania 501
Debugowanie funkcji i skryptów: ćwiczenia krok po kroku 502
Podsumowanie rozdziału 18 506

19 Obsługa błędów 507

Obsługa brakujących parametrów 508
Tworzenie domyślnej wartości dla parametru 508
Ustawianie parametru jako wymaganego 509
Ograniczanie wyboru 510
Korzystanie z *PromptForChoice* do ograniczania wyboru 511
Wykorzystanie *Test-Connection* do sprawdzania dostępności komputerów . . 512
Wykorzystanie operatora *-contains* do badania zawartości tablicy 514
Wykorzystanie operatora *-contains* do testowania właściwości 516
Obsługiwanie brakujących uprawnień 519
Próba i niepowodzenie 519
Sprawdzenie uprawnień i wyjście 519
Obsługa niedostępnych dostawców WMI 520
Niewłaściwe typy danych 529
Błędy przekroczenia zakresu 533
Stosowanie funkcji sprawdzania zakresu 534
Narzucanie ograniczeń na parametry 535
Używanie konstrukcji *Try...Catch...Finally* 536
Przechwytywanie wielu błędów 539
Korzystanie z metody *PromptForChoice* do ograniczania wyboru: ćwiczenia krok po kroku 542
Podsumowanie rozdziału 19 544

20 Zarządzanie serwerem Exchange . . . 545
Poznawanie poleceń cmdlet Exchange 2010 . . . 545
Praca ze zdalnymi serwerami Exchange . . . 547
Konfigurowanie ustawień odbiorców . . . 549
Tworzenie konta użytkownika i skrzynki pocztowej . . . 550
Raportowanie ustawień użytkowników . . . 554
Zarządzanie ustawieniami magazynowania . . . 556
Badanie bazy danych skrzynek pocztowych . . . 556
Zarządzanie bazami danych skrzynek pocztowych . . . 557
Zarządzanie rejestrowaniem zdarzeń w Exchange . . . 559
Zarządzanie inspekcją . . . 562
Analizowanie pliku inspekcji w formacie XML . . . 567
Tworzenie kont użytkowników: ćwiczenia krok po kroku . . . 571
Podsumowanie rozdziału 20 . . . 576

A Bazowe polecenia cmdlet Windows PowerShell . . . 577

B Liczebność modułów Windows PowerShell . . . 587

C Nazewnictwo poleceń Windows PowerShell . . . 591

D Windows PowerShell FAQ . . . 595

E Użyteczne klasy WMI . . . 603

F Podstawowe wskazówki dotyczące rozwiązywania problemów . . . 629

G Ogólne wskazówki skryptowania . . . 633

H Czasowniki PowerShell i ich polskie znaczenie . . . 641

Indeks . . . 649

O autorze . . . 689

Przedmowa

Od zawsze zdawałem sobie sprawę, że automatyzacja codziennych czynności jest krytyczną umiejętnością dla profesjonalisty IT. W niezwykłym stopniu zwiększa zarówno produktywność, jak i jakość operacji; umiejętność ta poprawia zarówno jakość firm, jak i karier osób, które ją opanują. Ulepszania automatyzacji działań IT było moim czołowym priorytetem, gdy dołączyłem do firmy Microsoft w 1999 roku jako architekt produktów i technologii zarządzania. Doprowadziło to do wynalezienia Windows PowerShell, a następnie do mozolnego przekształcania go w centralny element technik zarządzania. W tym czasie branża dokonała kilku znaczących przesunięć. Zmiany te sprawiły, że zostanie ekspertem od automatyzacji stało się jeszcze ważniejsze.

W trakcie projektowania pierwszej wersji PowerShell zespół wypracował bardzo silny związek z Exchange. Wyobrażaliśmy sobie, że Exchange pomoże w akceptacji PowerShell przez środowiska administratorów i profesjonalistów IT. Można sobie wyobrazić nasze zaskoczenie (i zachwyt), gdy odkryliśmy, że najbardziej aktywną społecznością korzystającą z PowerShell byli klienci VMWare. Skontaktowałem się więc z zespołem VMWare, aby dowiedzieć się, dlaczego nasz nowy shell został tak dobrze przyjęty przez ich klientów. Wyjaśniono mi, że ich klientami byli przede wszystkim profesjonaliści IT, którzy głównie zajmowali się utrzymywaniem posiadanych przez ich firmy serwerów. Wraz z wdrożeniem wirtualizacji nagle mieli do obsługi 5-10 razy więcej serwerów, zatem znaleźli się w sytuacji „automatyzuj lub utoniesz". Stali w obliczu rozpalającego się pożaru, a PowerShell był wiadrem wody, które pojawiło się w ostatniej chwili.

Przechodzenie do chmury jest kolejnym przesunięciem, które zwiększa ważność automatyzacji. Cały wysiłek projektantów koncentruje się na bezpiecznym wykonywaniu zmian poprzez modyfikację ustawień ogólnych oraz automatyzację. Przy korzystaniu z aplikacji w chmurze nie możemy polegać tylko na sprawnym pracowniku z filiżanką kawy i myszą – musimy zautomatyzować operacje za pomocą skryptów i procedur. Gdy czytamy raporty z awarii największych chmur, często widzimy, że prawdziwą przyczyną była ręczna konfiguracja. Gdy używamy automatyzacji i wystąpi błąd, przeglądamy nasze skrypty i modyfikujmy je, aby nie zdarzył się ponownie. Przy automatyzacji prawdziwe jest powiedzenie Nietzsche'go: co nas nie zabije, to nas wzmocni. Nie jest zaskoczeniem, że Azure obsługuje PowerShell od jakiegoś czasu, ale byłem zachwycony widząc, że Amazon właśnie opublikował 587 poleceń cmdlet do zarządzania AWS.

Nauka automatyzacji przy użyciu PowerShell jest więc sprawą krytyczną i nie ma zbyt wielu ludzi lepiej przygotowanych do tego zadania, niż Ed Wilson. Ed Wilson jest mężem *The Scripting Wife* i człowiekiem odpowiedzialnym za popularny blog *The Scripting Guy*. Nie będzie przesadą stwierdzenie, że Ed i jego żona Teresa to dwie najbardziej aktywne osoby w środowisku PowerShell. Ed znany jest z praktycznego podejścia „jak to zrobić" do PowerShell. Dzięki pracy z tak wielu klientami i ludźmi uczącymi się PowerShell Ed wie, jakie pytania będą chcieli zadać, jeszcze zanim sami sobie je uświadomią. I szczęśliwie znalazł czas, aby przekuć to w nową książkę: *Windows PowerShell 3.0 Krok po kroku*.

– Jeffrey Snover, Distinguished Engineer and Lead Architect, Microsoft Windows

Wprowadzenie

Windows PowerShell 3.0 jest podstawowym narzędziem zarządzania i automatyzacji, które wnosi prostotę wiersza polecenia do systemów operacyjnych następnej generacji. Dołączony do Windows 8 i Windows Server 2012, a także dostępny dla systemów Windows 7 i Windows Server 2008 R2, Windows PowerShell 3.0 oferuje niespotykaną dotąd siłę i elastyczność dla każdego – od zaawansowanych użytkowników, po administratorów sieci przedsiębiorstw i projektantów.

Kto powinien przeczytać tę książkę

Książka ta ma na celu pomóc profesjonalistom IT szybko i efektywnie zacząć korzystać z technik Windows PowerShell 3.0. Książka *Windows PowerShell 3.0 Krok po kroku* jest szczególnie adresowana do następujących osób:

- **Konsultanci sieci Windows** Każdy pragnący standaryzacji i automatyzacji procesów instalacyjnych oraz konfiguracji komponentów sieciowych .NET.
- **Administratorów sieci Windows** Każdy pragnący automatyzować codzienne zadania zarządzania w sieciach Windows.
- **Microsoft Certified Solutions Experts (MCSEs) oraz Microsoft Certified Trainers (MCTs)** Windows PowerShell jest kluczowym komponentem wielu cykli szkoleniowych firmy Microsoft i egzaminów certyfikacyjnych.
- **Ogólny personel techniczny** Każdy potrzebujący gromadzić informacje i konfigurować ustawienia na komputerach Windows.
- **Użytkownicy zaawansowani** Każdy chcący uzyskać maksymalną moc i konfiguralność swojego komputera Windows, czy to w domu, czy w niezarządzanym środowisku pracy.

Założenia

Autor oczekuje, że Czytelnik zna system operacyjny Windows, toteż nie zamieszcza wyjaśnień bazowych terminów sieciowych czy systemowych. Książka zakłada też, że Czytelnik ma przynajmniej podstawowe przygotowanie w zakresie programowania obiektowego i rozumie pojęcia i stosowalność takich technik, jak metody, właściwości czy rzutowanie. Przydatna będzie też przynajmniej podstawowa znajomość platformy .NET. Bardziej szczegółowe zagadnienia dotyczące programowania obiektowego, projektowania i skryptowania są wyjaśniane, gdy się pojawią.

Kto nie powinien czytać tej książki

Książka ta nie jest przewodnikiem referencyjnym po Windows PowerShell 3.0, ale ma na celu wprowdzić Czytelnika w świat PowerShell i zachęcić go do dalszych eksperymentów i poznawania środowiska na własną rękę. Oznacza to jednak, że szczególnie pogłębione, ezoteryczne tematy nie zostały w niej omówione. Choć pojawiają się niektóre zaawansowane zagadnienia, w ogólności zaczynamy od bardzo podstawowych pojęć i zmierzamy w kierunku bardziej złożonych zagadnień.

Z drugiej strony, jeśli ktoś nigdy nie używał komputera i nie orientuje się w elementarnych pojęciach dotyczących systemów operacyjnych, sieci czy programowania, ta książka zdecydowanie nie jest przeznaczona dla niego.

Konwencje użyte w książce

Książka prezentuje informacje stosując się do konwencji zaprojektowanych tak, aby wskazówki był jak najbardziej czytelne i łatwe do naśladowania.

- Każdy rozdział kończy się dwoma ćwiczeniami praktycznymi.
- Każde ćwiczenie składa się z serii zadań przedstawionych jako numerowane kroki (1, 2 i tak dalej), opisujące każde działanie, które trzeba wykonać, aby ukończyć ćwiczenie.
- Wyróżnione elementy z tytułami takimi jak „Uwaga” zawierają dodatkowe informacje lub alternatywne metody osiągnięcia celu.
- Tekst, który Czytelnik ma wpisać (poza blokami kodu) jest wyróżniony **wytłuszczeniem**.
- Znak plus (+) pomiędzy nazwami dwóch klawiszy oznacza, że klawisze te należy nacisnąć równocześnie. Na przykład „Naciśnij Alt+Tab” oznacza, że należy przytrzymać klawisz Alt, gdy naciskamy klawisz Tab.
- Elementy graficznego interfejsu użytkownika są wyróżnione czcionką bezszeryfową. Pionowa kreska pomiędzy dwoma lub więcej elementami menu (np. File | Close) oznacza, że należy wybrać pierwsze menu, a następnie kolejne i tak dalej.
- Słowa kluczowe, nazwy poleceń cmdlet, właściwości lub klas są wyróżnione czcionką bezszeryfową w kursywie, np. *Get-Help*.
- Szczególnie istotne pojęcia, a także wartości zmiennych są wyróżnione *kursywą*.

Wymagania systemowe

Do wykonania ćwiczeń praktycznych zawartych w książce potrzebne będą następujące elementy sprzętowe i programowe:

- Komputer z jednym z następujących systemów operacyjnych: Windows 7, Windows Server 2008 z dodatkiem Service Pack 2, Windows Server 2008 R2, Windows 8 lub Windows Server 2012.
- Zainstalowany pakiet Windows PowerShell 3.0 (niewymagane w przypadku systemu Windows 8 lub Windows Server 2012).
- Połączenie z Internetem do pobierania oprogramowania i przykładów.
- Część ćwiczeń wymaga dostępu do komputera zdalnego. Może być to maszyna wirtualna z dostępem sieciowym. Ćwiczenia z rozdziałów 15–17 wymagają dostępu do domeny Active Directory (kontrolera domeny). Ćwiczenia z rozdziału 20 wymagają dostępu do serwera Exchange.

Przykłady kodu

Wszystkie rozdziały książki zawierają przykłady kodu, które można wprowadzać interaktywnie, aby wypróbować nowe umiejętności i techniki. Część tych przykładów można pobrać z następującej strony:

http://aka.ms/PowerShellSBS_book

Podziękowania

Chciałbym podziękować za pomoc następującym osobom: mojemu agentowi Claudette Moore, gdyż bez niej książka ta nigdy by nie powstała. Moim redaktorom Devonowi Musgrave i Michaelowi Bolingerowi za przekształcenie tekstu w coś przypominającego prawdziwy angielski, oraz recenzentowi Thomasowi Lee, którego uwagi i spostrzeżenia pozwoliły stworzyć znacznie lepszą książkę. Na koniec wreszcie chcę podziękować mojej żonie Teresie (inaczej *Scripting Wife*), która przeczytała każdą stronę i dodała liczne sugestie, które będą bardzo przydatne początkującym skrypciarzom.

ROZDZIAŁ 1

Przegląd cech Windows PowerShell 3.0

Po zapoznaniu się z tym rozdziałem Czytelnik będzie umiał:

- Określić podstawowe zastosowania i możliwości Windows PowerShell
- Zainstalować Windows PowerShell
- Korzystać z podstawowych narzędzi wiersza polecenia wewnątrz Windows PowerShell
- Korzystać z systemu pomocy
- Uruchamiać podstawowe polecenia cmdlet
- Uzyskiwać pomoc na temat poleceń cmdlet

Opublikowanie wersji Windows PowerShell 3.0 stanowi znaczący postęp dla administratorów sieci systemów Windows. Dzięki połączeniu siły dojrzałego języka skryptowego z dostępem do narzędzi wiersza poleceń, interfejsów Windows Management Instrumentation (WMI) czy wręcz VBScript, Windows PowerShell zapewnia skuteczność i łatwość stosowania, której w platformie Windows brakowało od zawsze. Jako część standardu Microsoft Common Engineering Criteria, Windows PowerShell szybko staje się podstawowym rozwiązaniem zarządzania dla platformy Windows. Profesjonaliści IT używający systemu Windows Server 2012 muszą sięgnąć po Windows PowerShell – i to raczej prędzej, niż później.

Istota Windows PowerShell

Zapewne największą przeszkodą dla wielu administratorów sieci w przejściu do korzystania z Windows PowerShell jest niezrozumienie, co to naprawdę jest. Pod wieloma względami jest to zamiennik dla sędziwej powłoki CMD (*command* – polecenie). I rzeczywiście, w systemie Windows Server 2012 w trybie Core możliwe jest zastąpienie powłoki CMD przez Windows PowerShell, tak że po uruchomieniu serwera używa on PowerShell jako domyślnego interfejsu użytkownika. Jak można się przekonać, po uruchomieniu Windows PowerShell można używać w nim dobrze znanych

poleceń, na przykład *cd* do zmiany katalogu roboczego lub *dir* do wyświetlenia jego zawartości – tak samo jak w oknie CMD.

```
Windows PowerShell
Copyright (C) 2012 Microsoft Corporation. All rights reserved.

PS C:\Users\administrator> cd c:\
PS C:\> dir

    Directory: C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         3/22/2012   4:03 AM            PerfLogs
d-r--         3/22/2012   4:24 AM            Program Files
d-r--         3/23/2012   6:02 PM            Users
d----         3/23/2012   4:59 PM            Windows
-a---         3/22/2012   4:33 AM         24 autoexec.bat
-a---         3/22/2012   4:33 AM         10 config.sys

PS C:\>
```

Można również połączyć tradycyjne polecenia interpretera CMD z niektórymi nowszymi narzędziami, takimi jak *fsutil*, co prezentuje kolejny przykład:

```
PS C:\> md c:\test

    Directory: C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         4/22/2012   5:01 PM            test

PS C:\> fsutil file createnew C:\test\mynewfile.txt 1000
File C:\test\mynewfile.txt is created
PS C:\> cd c:\test
PS C:\test> dir

    Directory: C:\test

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         4/22/2012   5:01 PM       1000 mynewfile.txt

PS C:\test>
```

Przykłady te prezentują interaktywne wykorzystanie Windows PowerShell. Interaktywność jest jedną z głównych funkcji WIndows PowerShell. Aby użyć PowerShell interaktywnie, wystarczy otworzyć jego okno zachęty i zacząć wpisywać polecenia – po jednym na raz, albo łącząc je ze sobą, podobnie jak w pliku wsadowym. Do tego tematu powrócę później, gdyż najpierw muszę przedstawić bardziej podstawowe informacje.

Korzystanie z poleceń cmdlet

Oprócz aplikacji konsolowych systemu Windows i wbudowanych poleceń w środowisku PowerShell dostępne są *polecenia cmdlet* (wymawiane jako „komandlet"), budujące jego prawdziwą potęgę. Polecenia cmdlet mogą być tworzone przez każdego. Zespół programistów Windows PowerShell w firmie Microsoft utworzył podstawowy zbiór poleceń, ale oprócz tego powstały już setki poleceń utworzonych przez inne grupy projektowe. Większość z tych poleceń jest dołączonych do systemów Windows 8 i Windows Server 2012, podczas gdy inne pakiety stanowią część bardziej wyspecjalizowanych programów, takich jak Microsoft Exchange czy Microsoft SQL Server. Polecenia cmdlet są plikami wykonywalnymi, które wykorzystują funkcjonalności wbudowane w środowisko PowerShell i dzięki temu są tak łatwe do napisania. Nie są to skrypty zawierające nieskompilowany kod – są zbudowane przy wykorzystaniu usług specjalnej przestrzeni nazw .NET Framework. Windows PowerShell 3.0 w wersji dla Windows 8 wyposażony jest w około 1000 poleceń cmdlet i w miarę dodawania nowych funkcji lub ról dołączane są kolejne. Polecenia te mają na celu zapewnienie administratorom sieci czy konsultantom dostępu do siły Windows PowerShell bez konieczności tworzenia złożonych skryptów. Jedną z głównych przewag PowerShell jest to, że wszystkie polecenia wykorzystują standardową konwencję nazewniczą określaną terminem *czasownik-rzeczownik* (*verb-noun*), na przykład ***Get-Help*** („otrzymaj pomoc"), ***Get-EventLog*** lub ***Get-Process***. Polecenia używające czasownika ***Get*** (weź, pobierz) wyświetlają informacje o elemencie umieszczonym po prawej stronie łącznika. Polecenia używające czasownika ***Set*** (ustal, określ) powodują ustawienie odpowiednich cech elementu po prawej stronie łącznika. Na przykład polecenie cmdlet ***Set-Service*** pozwala (między innymi) zmienić tryb uruchomieniowy usługi (ang. *service*). Wszystkie polecenia używają jednego ze standardowych czasowników. W celu wyświetlenia listy tych czasowników można użyć polecenia cmdlet ***Get-Verb***. W Windows PowerShell 3.0 występuje około stu zaakceptowanych czasowników.

Instalowanie Windows PowerShell

Windows PowerShell 3.0 jest dołączony do systemu operacyjnego Windows Server 2012 i oraz Windows 8. W witrynie Microsoft Download Center dostępny jest pakiet Windows Management Framework 3.0, zawierający zaktualizowane wersje Windows Remote Management (WinRM), WMI oraz Windows PowerShell 3.0 dla systemów Windows 7, Windows Server 2008 R2 oraz Windows Server 2008. W systemach tych wymagane jest zainstalowanie dodatku Service Pack 1 (co najmniej) oraz platformy Microsoft .NET Framework 4.0. Nie jest dostępny pakiet dla systemu Windows Vista, Windows Server 2003 i wcześniejszych wersji. Nie istnieje także pakiet dla systemów Windows 8 ani Windows Server 2012, gdyż oprogramowanie jest zawarte w tych systemach.

W tym samym systemie operacyjnym może być zainstalowana zarówno powłoka Windows PowerShell 3.0, jak i wersja 2.0, ale wymaga to dostępności równocześnie .NET Framework 3.5 oraz 4.0.

Aby uniknąć problemów i frustracji podczas instalacji, sensowne jest wykorzystanie skryptu sprawdzającego system operacyjny, poziom zainstalowanych dodatków Service Pack oraz platformy .NET. Poniższy przykład prezentuje skrypt *Get-PowerShellRequirements.ps1* sprawdzający te warunki wstępne:

Get-PowerShellRequirements.ps1

```
Param([string[]]$computer = @($env:computername, "LocalHost"))
 foreach ($c in $computer)
  {
    $o = Get-WmiObject win32_operatingsystem -cn $c
    switch ($o.version)
    {
        {$o.version -gt 6.2} {"$c is Windows 8 or greater"; break}
        {$o.version -gt 6.1}
          {
           If($o.ServicePackMajorVersion -gt 0){$sp = $true}
           If(Get-WmiObject Win32_Product -cn $c |
              where { $_.name -match '.NET Framework 4'}) {$net = $true }
           If($sp -AND $net) { "$c meets the requirements for PowerShell 3" ; break}
           ElseIF (!$sp) {"$c needs a service pack"; break}
           ELSEIF (!$net) {"$c needs a .NET Framework upgrade"} ; break}
        {$o.version -lt 6.1} {"$c does not meet standards for PowerShell 3.0"; break}
        Default {"Unable to tell if $c meets the standards for PowerShell 3.0"}
    }

  }
```

Wdrażanie Windows PowerShell w starszych systemach operacyjnych

Po pobraniu pakietu instalacyjnego z witryny *http://www.microsoft.com/downloads* można wdrożyć go na komputerach przedsiębiorstwa wykonując jedno z poniższych działań:

- Utworzyć pakiet instalacyjny Windows Management Framework i ogłosić go dla odpowiedniej jednostki organizacyjnej (OU) lub zbioru jednostek.
- Utworzyć obiekt zasad grupy (GPO) instalacji oprogramowania w usługach AD DS i połączyć go z odpowiednią jednostką organizacyjną.
- Dołączyć pakiet jako zaaprobowany do serwera SUS (Software Update Services), jeśli jest dostępny.
- Umieścić pakiet w centralnym udziale plikowym lub witrynie, aby użytkownicy mogli go zainstalować we własnym zakresie.

Jeśli wdrożenie odbywa się na pojedynczym komputerze, a nie w całym przedsiębiorstwie, najprostszą metodą wdrażania będzie pobranie pakietu i przejście przez kroki kreatora instalacji.



Uwaga Aby móc zacząć używać narzędzi wiersza poleceń Windows PowerShell, należy uruchomić powłokę, wybierając Start | Run | PowerShell. Pojawi się okno polecenia ze znakiem zachęty PS>, w którym można zacząć wpisywać polecenia.

Korzystanie z narzędzi wiersza poleceń

Jak wspomniałem wcześniej, z wnętrza Windows PowerShell można bezpośrednio wywoływać narzędzia trybu wiersza poleceń. Przewagą wykorzystania PowerShell jako środowiska dla tych narzędzi, w przeciwieństwie do korzystania z tradycyjnego interpretera CMD, są funkcje przekierowywania wyjścia i formatowania dostępne w PowerShell. Dodatkowo, jeśli mamy już pliki wsadowe (.cmd lub .bat) wykorzystujące istniejące narzędzia wiersza poleceń, można je łatwo zmodyfikować, aby mogły być uruchamiane w środowisku PowerShell. Poniższa procedura ilustruje dołączenie wyników polecenia *ipconfig* do pliku tekstowego.

➜ Wywoływanie polecenia *ipconfig*

1. Uruchom powłokę Windows PowerShell, wybierając Start | Run | Windows PowerShell. Okno poleceń PowerShell otworzy się domyślnie w głównym folderze Documents zalogowanego użytkownika.
2. Wpisz polecenie **ipconfig /all**:

```
PS C:\> ipconfig /all
```

3. Przekieruj wyniki plecenia do pliku tekstowego, jak poniżej:

```
PS C:\> ipconfig /all >ipconfig.txt
```

4. Otwórz utworzony plik w Notatniku, aby przejrzeć zawartość:

```
PS C:\> notepad ipconfig.txt
```

Wpisywanie pojedynczych poleceń jest oczywiście użyteczne, ale zazwyczaj potrzebujemy więcej niż jednego, aby zrealizować założony cel – na przykład zgromadzić informacje konfiguracyjne potrzebne do rozwiązania problemów z instalacją lub niedostateczną wydajnością. I tu objawia się przewaga Windows PowerShell. W przeszłości w takim przypadku trzeba było albo napisać plik wsadowy, albo wpisywać kolejne polecenia ręcznie. Oto przykład prostego pliku wsadowego rejestrującego informacje przydatne do rozwiązywania problemów z siecią:

TroubleShoot.bat

```
ipconfig /all >c:\tshoot.txt
route print >>c:\tshoot.txt
hostname >>c:\tshoot.txt
net statistics workstation >>c:\tshoot.txt
```

Oczywiście przy ręcznym wpisywaniu poleceń musielibyśmy czekać na wykonanie każdego polecenia przed wywołaniem kolejnego. W takiej sytuacji łatwo można „zagubić się" we właściwej sekwencji lub wpisać kolejne polecenie przed zakończeniem przetwarzania poprzedniego, co doprowadzi do bezużytecznych wyników. Windows PowerShell eliminuje ten problem. Można teraz wpisać wiele poleceń w jednym wierszu, po czym odejść od komputera lub zająć się innymi zadaniami, podczas gdy system generuje wyniki. Nie ma potrzeby tworzenia pliku wsadowego w tym celu.



Wskazówka Można wpisywać wiele poleceń w jednym wierszu Windows PowerShell, aby zostały wykonane po kolei. Poszczególne polecenia należy rozdzielać średnikami.

Poniższy przykład prezentuje wywołanie wielu poleceń w jednym wierszu. Użyte polecenia są zawarte w pliku RunningMultipleCommands.txt.

➜ Uruchamianie wielu poleceń

1. Otwórz okno poleceń Windows PowerShell, jeśli nie jest jeszcze uruchomione.
2. Wpisz polecenie **ipconfig /all**, przesyłając wynik do pliku tekstowego Tshoot.txt przy użyciu znaku przekierowania (>):

   ```
   ipconfig /all >tshoot.txt
   ```

3. W tym samym wierszu wpisz średnik i dopisz polecenie **route print**. Wynik tego polecenia dodaj do pliku Tshoot.txt, używając znaku przekierowania z dołączeniem (>>). Wiersz polecenia będzie wyglądał następująco:

   ```
   ipconfig /all >tshoot.txt; route print >>tshoot.txt
   ```

4. Nadal w tym samym wierszu wstaw kolejny średnik i dodaj polecenie **hostname**, ponownie kierując wynik do pliku Tshoot.txt:

   ```
   ipconfig /all >tshoot.txt; route print >>tshoot.txt; hostname >>tshoot.txt
   ```

5. Dołącz kolejne polecenie **net statistics workstation**, oddzielając je od poprzednich znakiem średnika i dołączając wyjście polecenia do pliku Tshoot.txt. Ostateczna postać wiersza polecenia będzie wyglądać następująco:

   ```
   ipconfig /all >tshoot.txt; route print >>tshoot.txt; hostname >>tshoot.txt; net
   statistics workstation >>tshoot.txt
   ```

Problemy dotyczące zabezpieczeń

Biorąc pod uwagę ogromną elastyczność i wszechstronność Windows PowerShell, powstają nieuniknione obawy dotyczące zabezpieczeń systemowych. Jednak problemy te zostały uwzględnione już na etapie projektowania powłoki PowerShell.

Po uruchomieniu Windows PowerShell domyślnie wybierany jest folder Documents bieżącego użytkownika. Gwarantuje to, że znajdujemy się w katalogu, w którym dysponujemy uprawnieniami do wykonywania szeregu działań. Jest to znacznie bezpieczniejsze, niż zaczynanie od głównego katalogu dysku lub katalogu systemowego.

Aby zmienić katalog w konsoli PowerShell, nie możemy automatycznie przejść na wyższy poziom. Konieczne jest jawne wpisanie miejsca docelowego w operacji zmiany katalogu (choć można użyć polecenia ***cd ..***, aby przejść do katalogu nadrzędnego).

Wykonywanie skryptów jest domyślnie zablokowane i można nim łatwo zarządzać poprzez zasady grupy. Możliwe jest również włączenie tej funkcjonalności dla określonego użytkownika lub sesji.

Kontrolowanie wykonywania poleceń cmdlet

Każdemu chyba zdarzyło się otworzyć interpreter CMD, wpisać jakieś polecenie i nacisnąć Enter, aby zobaczyć, co ono robi. Jednak co by było, gdyby tym poleceniem było *format C:*? Czy na pewno chcielibyśmy sformatować dysk C w naszym komputerze? W tym podrozdziale pokażę niektóre argumenty, które pozwalają sterować sposobem wykonywania poleceń cmdlet. Choć nie wszystkie polecenia obsługują te argumenty, robi to większość poleceń dołączonych do Windows PowerShell. Argumenty, których można użyć do sterowania wykonywaniem, to *-whatif*, *-confirm* oraz *suspend*. To ostatnie nie jest w istocie argumentem przekazywanym do polecenia cmdlet, ale działaniem, które można podjąć po zgłoszeniu potwierdzenia, a zatem dodatkową metodą kontrolowania wykonywania poleceń.

Uwaga Aby użyć argumentu *-whatif*, należy wpisać całe polecenie cmdlet i dopisać po nim **-whatif** na końcu listy argumentów. Opcja ta jest dostępna tylko dla poleceń zmieniających stan systemu. Z tego względu nie można (ani nie ma potrzeby) stosowania argumentu -whatif dla poleceń informacyjnych, takich jak *Get-Process*.

Polecenia cmdlet zmieniające stan systemu (na przykład *Set-Service*) wspierają *tryb prototypowy*, który jest włączany poprzez użycie argumentu *-whatif*. To, czy dane polecenie obsługuje ten argument, jest zależne od programisty tworzącego cmdlet. Zespół projektowy Windows PowerShell zaleca implementację argumentu *-whatif* we wszystkich poleceniach, które mogą dokonywać zmian. Poniższy przykład prezentuje wykorzystanie argumentu *-whatif*. Polecenia użyte w tej procedurze zawarte są w pliku UsingWhatif.txt.

➜ Korzystanie z argumentu *-whatif*

1. Otwórz okno Windows PowerShell, o ile nie zostało uruchomione wcześniej.
2. Uruchom program Notatnik. W tym celu wpisz **notepad** i naciśnij Enter.
3. Zidentyfikuj proces związany z uruchomioną sesją Notatnika, używając polecenia cmdlet *Get-Process*. Wpisz dostateczną część nazwy procesu, aby ją zidentyfikować, uzupełniając pozostałą część gwiazdką (*) – w ten sposób nie trzeba wpisywać całej (niekiedy długiej) nazwy procesu:

```
Get-Process note*
```

4. Przejrzyj wynik polecenia, aby znaleźć identyfikator procesu. Wynik polecenia z mojego komputera jest przedstawiony poniżej. Trzeba pamiętać, że niemal na pewno numer procesu na innym komputerze będzie się różnił od pokazanego:

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     39       2      944        400    29     0.05   1056 notepad
```

5. Użyj argumentu *-whatif*, aby się przekonać, co by się stało po użyciu polecenia *Stop-Process* wobec identyfikatora znalezionego w punkcie 4. Parametr *-id* pozwala podać identyfikator procesu w poleceniu. Polecenie to w moim przypadku wygląda następująco:

```
Stop-Process -id 1056 -whatif
```

6. Przejrzyj wynik polecenia. Informuje ono, że jego „prawdziwe” wykonanie spowoduje zatrzymanie procesu notepad o identyfikatorze wskazanym w poleceniu:

```
What if: Performing operation "Stop-Process" on Target "notepad (1056)"
(What if: Wykonanie operacji "Stop-Process" wobec celu "notepad (1056)"
```

Potwierdzanie akcji

Jak wiemy już z poprzedniego podpunktu, możemy wykorzystać argument *-whatif* do wykonania polecenia w trybie prototypowym. W ten sposób możemy dowiedzieć się, co polecenie zrobi. Jeśli jednak chcemy być monitowani przed rzeczywistym wykonaniem polecenia, możemy użyć argumentu *-confirm*. Polecenia użyte w kolejnym przykładzie są umieszczone w pliku tekstowym ConfirmingExecutionOfCmdlets.txt.

➜ Potwierdzanie wykonania poleceń

1. Otwórz okno Windows PowerShell, uruchom program Notatnik i zidentyfikuj jego proces, tak jak w krokach 1 – 4 poprzedniego ćwiczenia.
2. Użyj argumentu *-whatif*, aby wymusić monitowanie dla polecenia *Stop-Process* dla identyfikatora znalezionego w punkcie poprzednim:

```
Stop-Process -id 1768 -confirm
```

Polecenie wyświetli następujący monit:

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (936)".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is
"Y"):
(Potwierdź
Czy na pewno chcesz wykonać to działanie?
Tak  Tak na wszystkie  Nie  Nie na wszystkie  Wstrzymaj  Pomoc (domyślnie Tak))
```

3. Wpisz **y** i naciśnij Enter. Notatnik zostanie zamknięty, a Windows PowerShell wyświetli znak zachęty, jak poniżej:

```
PS C:\users\edw
```

Wskazówka W celu wstrzymania wykonywania polecenia należy w monicie potwierdzenia wpisać **s** i nacisnąć Enter.



Wstrzymywanie potwierdzania poleceń

Możliwość wyświetlania monitu o potwierdzenie przed wykonaniem polecenia jest niezwykle użyteczna i niekiedy może być krytycznym elementem utrzymywania najwyższego możliwego poziomu dostępności systemu. Łatwo można sobie wyobrazić sytuację, gdy wpisujemy długie, skomplikowane polecenie, po czym przypominamy sobie, że najpierw trzeba sprawdzić coś innego. Na przykład możemy chcieć zatrzymać wiele procesów, ale musimy przejrzeć ich szczegóły, aby się upewnić, że wymusimy zatrzymania nieodpowiedniego procesu. W takiej sytuacji, zamiast kasować pracowicie wpisany wiersz (i musieć wprowadzić go ponownie później), możemy posłużyć się opcją wstrzymania wykonania polecenia. Polecenia użyte w poniższym przykładzie zawiera plik SuspendConfirmationOfCmdlets.txt.

➜ Wstrzymywanie wykonywania polecenia cmdlet

1. Otwórz okno Windows PowerShell, uruchom program Notatnik i zidentyfikuj jego proces, tak jak w krokach 1 – 4 ćwiczenia „Korzystanie z argumentu *-whatif*”. Wynik na moim komputerze jest pokazany poniżej. Trzeba pamiętać, że niemal na pewno numer procesu na innym komputerze będzie się różnił od pokazanego:

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     39       2      944        400    29     0.05   3576 notepad
```

2. Użyj argumentu *-confirm*, aby wymusić monitowanie przed wykonaniem polecenia *Stop-Process*, jak poniżej:

```
Stop-Process -id 3576 -confirm
```

   Polecenie wyświetli monit podobny do pokazanego w poprzednim przykładzie.

3. Aby wstrzymać wykonanie polecenia, wpisz **s** i naciśnij Enter. Znak zachęty PowerShell zmieni się na „potrójną strzałkę", jak poniżej:

```
PS C:\users\edw>>>
```

4. Użyj polecenia *Get-Process*, aby uzyskać listę uruchomionych procesów o nazwach zaczynających się na literę *n*, jak poniżej:

```
Get-Process n*
```

   Na moim komputerze występowały dwa takie procesy: uruchomiony przeze mnie proces *notepad* oraz jeszcze jeden, związany z jedną z usług systemowych:

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     39       2      944        400    29     0.05   3576 notepad
     75       2     1776       2708    23     0.09    632 nvsvc32
```

5. Powróć do monitu potwierdzenia, wpisując **exit** i naciskając Enter. Monit o potwierdzenie zostanie wyświetlony ponownie.

6. Wpisz **y** i naciśnij Enter, aby zatrzymać proces Notepad. Nie ma już dalszych potwierdzeń, zatem znak zachęty zmieni się na standardowy, jak poniżej:

```
PS C:\users\edw
```

Posługiwanie się Windows PowerShell

W tej części zajmiemy się uzyskiwaniem dostępu do powłoki Windows PowerShell i konfigurowaniem konsoli.

Wywoływanie Windows PowerShell

Po zainstalowaniu Windows PowerShell w systemie Windows jest on natychmiast dostępny do użycia. Jednak ciągłe sięganie do menu Start czy też używanie sekwencji klawiszy Windows+R w celu wywołania monitu *run* jest czasochłonne i niewygodne. W przypadku systemu Windows 8 problem jest nieco mniejszy, gdyż można po prostu wpisać **PowerShell** na ekranie Start. Dla większej wygody wolałem przypiąć Windows PowerShell oraz PowerShell ISE do ekranu Start. W systemie Windows Server 2012 w wersji Core zdecydowałem się zastąpić domyślną powłokę CMD konsolą Windows PowerShell. Jest to idealne rozwiązanie dla mnie i mojego sposobu pracy, zatem napisałem skrypt, który to realizuje. Skrypt ten może być wywoływany podczas logowania,

aby automatycznie wykonać niezbędne zmiany. W systemie Windows 8 skrypt dodaje skróty do konsoli Windows PowerShell i PowerShell ISE do ekranu Start oraz paska zadań pulpitu. W Windows 7 skróty tworzone są w pasku zadań i menu Start. Skrypt ten działa tylko w systemach w języku angielskim. Aby móc użyć go w systemach lokalizowanych w innych językach, trzeba zmienić wartości zmiennych *$pinToStart* oraz *$pinToTaskBar* na odpowiedniki w tych językach.

> **Uwaga** Korzystanie ze skryptów Windows PowerShell omówione jest w rozdziale 5, „Używanie skryptów PowerShell". Rozdział ten zawiera szczegółowe informacje, jak działają skrypty i jak należy je uruchamiać.



PinToStartAndTaskBar.ps1

```
$pinToStart = "Pin to Start"
$pinToTaskBar = "Pin to Taskbar"
$file = @((Join-Path -Path $PSHOME  -childpath "PowerShell.exe"),
          (Join-Path -Path $PSHOME  -childpath "powershell_ise.exe") )
Foreach($f in $file)
 {$path = Split-Path $f
  $shell=New-Object -com "Shell.Application"
  $folder=$shell.Namespace($path)
  $item = $folder.parsename((Split-Path $f -leaf))
  $verbs = $item.verbs()
  foreach($v in $verbs)
    {if($v.Name.Replace("&","") -match $pinToStart){$v.DoIt()}}
  foreach($v in $verbs)
    {if($v.Name.Replace("&","") -match $pinToTaskBar){$v.DoIt()}} }
```

Konfigurowanie konsoli Windows PowerShell

Konsola Windows PowerShell ma wiele konfigurowalnych aspektów. Możliwe elementy można umieścić w pliku konfiguracyjnym typu Psconsole (z rozszerzeniem .psc1). Aby wyeksportować bieżącą konfigurację do pliku, można użyć polecenia *Export-Console*, jak w poniższym przykładzie:

```
PS C:\users\edw\> Export-Console myconsole
```

Plik konsoli zostanie zapisany w bieżącym katalogu z domyślnym rozszerzeniem .psc1. Jest to plik w formacie XML. Domyślny (niezmodyfikowany) plik konfiguracji konsoli został przedstawiony poniżej:

```
<?xml version="1.0" encoding="utf-8"?>
<PSConsoleFile ConsoleSchemaVersion="1.0">
  <PSVersion>3.0</PSVersion>
  <PSSnapIns />
</PSConsoleFile>
```

➜ Kontrolowanie opcji uruchamiania PowerShell

1. Okno PowerShell można uruchomić bez paska tytułu, używając przełącznika *-nologo*, jak poniżej:

```
PowerShell -nologo
```

2. Aby uruchomić konkretną wersję Windows PowerShell (przy założeniu, że jest zainstalowana na komputerze), można użyć przełącznika *-version* (w celu uruchomienia Windows PowerShel 2.0 musi być zainstalowana platforma .NET Framework 3.5):

```
PowerShell -version 2
```

3. Uruchomienie Windows PowerShell przy użyciu wskazanego pliku konfiguracyjnego wymaga użycia przełącznika *-psconsolefile*, jak poniżej:

```
PowerShell -psconsolefile myconsole.psc1
```

4. Możliwe jest uruchomienie Windows PowerShell w celu wykonania określonego polecenia, po czym nastąpi samoczynne wyjście (zamknięcie konsoli). W tym celu należy użyć polecenia *-command*. Samo polecenie do wykonania musi być poprzedzone znakiem & i ujęte w nawiasy klamrowe, jak poniżej:

```
PowerShell -command "& {Get-Process}"
```

Przekazywanie opcji do poleceń cmdlet

Jedną z najbardziej użytecznych cech Windows PowerShell jest standaryzacja składni używanej przez polecenia cmdlet. Znacząco upraszcza to naukę obsługi nowej powłoki i języka. Tabela 1-1 zawiera listę najczęściej stosowanych parametrów. Trzeba pamiętać, że nie wszystkie parametry są implementowane we wszystkich poleceniach. Jeśli jednak dany parametr jest zaimplementowany, zostanie zinterpretowany w taki sam sposób przez każde polecenie, gdyż interpretację wykonuje sam silnik PowerShell.

Tabela 1-1 Typowe parametry

Parametr	Znaczenie
-whatif	Powoduje, że polecenie nie jest wykonywane, ale wyświetlana jest informacja, co nastąpi po jego wykonaniu (tryb prototypu).
-confirm	Nakazuje wyświetlanie monitu przed właściwym wykonaniem polecenia.
-verbose	Powoduje wyświetlanie bardziej szczegółowych informacji o wykonaniu polecenia.
-debug	Powoduje wyświetlanie informacji debugowania.
-ErrorAction	Nakazuje wykonanie określonego działania, gdy wystąpi błąd. Dopuszczalne akcje to *continue* (kontynuuj), *stop* (zatrzymaj), *silentlyContinue* (kontynuuj bez powiadamiania) oraz *inquire* (pytaj).

Tabela 1-1 Typowe parametry

Parametr	Znaczenie
-ErrorVariable	Poleca użycie określonej zmiennej do przechowania informacji o błędzie. Jest to uzupełnienie standardowej zmiennej $error.
-OutVariable	Nakazuje przekierowanie informacji wyjściowych do wskazanej zmiennej.
-OutBuffer	Nakazuje przechowanie określonej liczby obiektów przed wywołaniem następnego polecenia cmdlet w ciągu przekazywania (*pipeline*).

Uwaga Aby uzyskać informacje pomocy na temat dowolnego polecenia cmdlet, należy użyć polecenia ***Get-Help*** <nazwa cmdlet>. Na przykład ***Get-Help Get-Process*** zwraca informacje o składni i stosowaniu polecenia ***Get-Process***.



Korzystanie z opcji pomocy

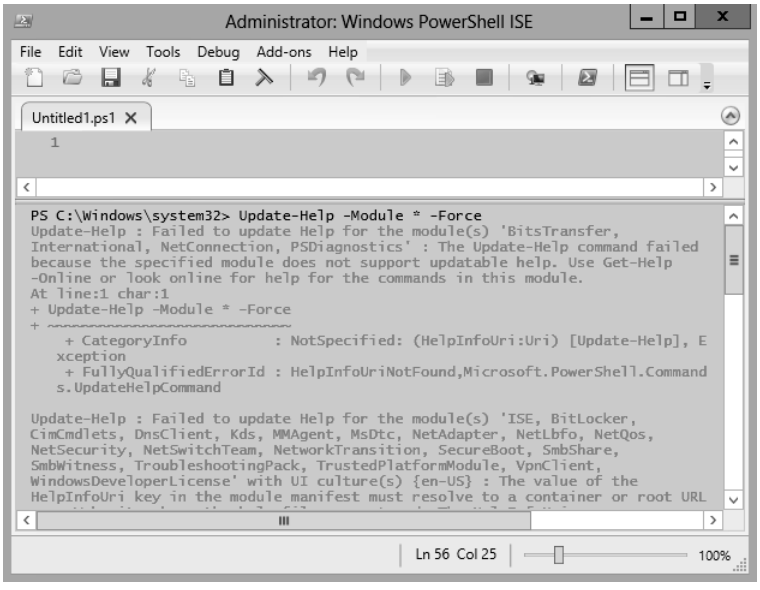
Jednym z pierwszych poleceń, które należy wywołać przy rozpoczynaniu pracy z Windows PowerShell, jest ***Update-Help***. Wynika to z faktu, że Windows PowerShell nie jest domyślnie wyposażony w pliki pomocy dla wszystkich poleceń cmdlet. Nie oznacza to, że taka pomoc nie istnieje – jedynie to, że pomoc wykraczająca poza podstawową składnię wymaga pobrania dodatkowych materiałów.

Domyślna instalacja Windows PowerShell 3.0 zawiera liczne moduły, które zmieniają się pomiędzy różnymi wdrożeniami zależnie od wersji systemu operacyjnego, zainstalowanych w nim funkcji i aktywnych ról. W istocie Windows PowerShell 3.0 zainstalowany w systemie Windows 7 zawiera mniej modułów i poleceń, niż dostępne w stacji roboczej Windows 8. Nie oznacza to jednak chaosu, gdyż najważniejsze polecenia cmdlet – *core* – nie zmieniają się pomiędzy poszczególnymi instalacjami. Różnice pomiędzy instalacjami wynikają z faktu, że dodatkowe funkcje i role systemu operacyjnego często powodują doinstalowanie nowych modułów Windows PowerShell i związanych z nimi poleceń cmdlet.

Modułowa budowa Windows PowerShell wymaga dodatkowej uwagi podczas aktualizowania plików pomocy. Proste wywołanie ***Update-Help*** nie uaktualni wszystkich modułów dostępnych w określonym systemie. W rzeczywistości niektóre moduły mogą w ogóle nie obsługiwać aktualizacji pomocy – w tym wypadku próba uaktualnienia wygeneruje błąd. Najprostszą metodą upewnienia się, że aktualizacja obejmie wszystkie możliwe elementy, jest użycie parametru ***-module*** oraz ***-force***. Polecenie powodujące uaktualnienie pomocy dla wszystkich zainstalowanych modułów (tych, które wspierają aktualizację pomocy) będzie wyglądało następująco:

```
Update-Help -Module * -Force
```

Rezultat uruchomienia polecenia *Update-Help* w typowym systemie klienckim Windows 8 przedstawia rysunek 1-1.



Rysunek 1-1 Błędy wyświetlane po próbie aktualizacji pomocy w modułach, które nie obsługują aktualizacji.

Sposobem na uniknięcie wyświetlania ekranu pełnego błędów jest wywołanie polecenia *Update-Help* z przełącznikiem blokującym wyświetlanie wszystkich komunikatów o błędach:

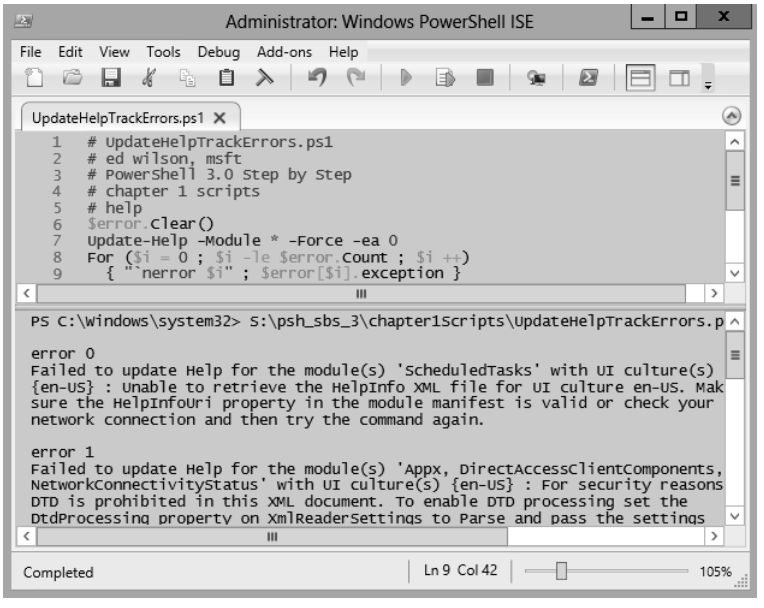
```
Update-Help -Module * -Force -ea 0
```

Podejście takie powoduje jednak, że nigdy nie będziemy mieć pewności, czy rzeczywiście uzyskaliśmy zaktualizowaną treść pomocy dla wszystkich elementów, dla których jest to możliwe. Lepszym podejściem byłoby ukrycie błędów pojawiających się w trakcie procesu aktualizacji, ale wyświetlenie tych, które występują po jego zakończeniu. Dodatkową korzyścią takiego podejścia jest wyświetlenie błędów procesu czyszczącego (*cleaner*). Technikę tę ilustruje skrypt UpdateHelpTrackErrors.ps1. Pierwszą rzeczą, jaką robi ten skrypt, jest wyczyszczenie stosu błędów poprzez wywołanie metody *clear*. Następnie skrypt wywołuje polecenie cmdlet *Update_Help* z parametrami *module* i *force*. Dodatkowo użyty jest przełącznik *ErrorAction* (*ea* widoczne w kodzie skryptu to alias) z wartością 0. Powoduje to, że błędy nie są wyświetlane podczas przebiegu polecenia. Skrypt kończy pętla *For*, przechodząca przez zgromadzone komunikaty błędów i wyświetlająca zarejestrowane wyjątki. Cały skrypt został przedstawiony poniżej.

UpdateHelpTrackErrors.ps1

```
$error.Clear()
Update-Help -Module * -Force -ea 0
For ($i = 0 ; $i -le $error.Count ; $i ++)
  { "'nerror $i" ; $error[$i].exception }
```

W trakcie wykonywania skryptu wyświetlany jest pasek postępu. Po jego ukończeniu wyświetlane są wszystkie napotkane błędy. Skrypt i powiązane z nim komunikaty błędów są widoczne na rysunku 1-2.



Rysunek 1-2 Wyczyszczone wyjście błędów dla procesu aktualizacji pomocy

Mogłoby się wydawać, że ustalenie, które moduły otrzymały zaktualizowane pliki pomocy, umożliwia również wykorzystanie parametru *-verbose* dla polecenia *Update-Help*. Jednak dane wyjściowe powodują tak szybkie przewijanie ekranu, że trudno jest cokolwiek na nim zobaczyć. Aby rozwiązać ten problem, można przekierować wyjście polecenia do pliku tekstowego. W przykładzie poniżej wszystkie moduły próbują zaktualizować swoje pliki pomocy. Pełne komunikaty zwrotne są kierowane do pliku *updatehelp.txt* zlokalizowanego w folderze *fso* na dysku C.

```
Update-Help -module * -force -verbose 4 >>c:\fso\updatedhelp.txt
```

Windows PowerShell zapewnia wysoki poziom „odkrywalności" (*discoverability*); innymi słowy, aby nauczyć się korzystać z PowerShell, należy po prostu zacząć go używać. Ważną rolę w tym praktycznym poznawaniu pełnią serwery pomocy online.

System pomocy dostępny w Windows PowerShell jest dostępny kilkoma metodami. Jedną z nich jest wspomniane wcześniej polecenie cmdlet *Get-Help*. Aby uzyskać informacje o stosowaniu tego polecenia, można je wywołać podając je jako argument:

```
Get-Help Get Help
```

Spowoduje to wyświetlenie informacji o składni polecenia, podobnych do pokazanych poniżej[1]:

```
NAZWA
    Get-Help

PODSUMOWANIE
    Wyświetla informacje o poleceniach i koncepcjach Windows PowerShell.

SKŁADNIA
    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>]
    [-Full [<SwitchParameter>]] [-Functionality <String[]>] [-Path <String>] [-Role
    <String[]>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>]
    [-Functionality <String[]>] [-Path <String>] [-Role <String[]>] -Detailed
    [<SwitchParameter>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>]
    [-Functionality <String[]>] [-Path <String>] [-Role <String[]>] -Examples
    [<SwitchParameter>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>]
    [-Functionality <String[]>] [-Path <String>] [-Role <String[]>] -Online
    [<SwitchParameter>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>]
    [-Functionality <String[]>] [-Path <String>] [-Role <String[]>] -Parameter
    <String> [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>]
    [-Functionality <String[]>] [-Path <String>] [-Role <String[]>] -ShowWindow
    [<SwitchParameter>] [<CommonParameters>]

OPIS
    Polecenie cmdlet Get-Help informacje o poleceniach i koncepcjach Windows
    PowerShell, w tym o poleceniach cmdlet, funkcjach, poleceniach CIM, przepływach,
    dostawcach, aliasach i skryptach.

    Aby uzyskać pomoc na temat polecenia Windows PowerShell, wpisz "Get-Help"
    uzupełnione o nazwę polecenia, tak jak: Get-Help Get-Process. Aby uzyskać listę
    wszystkich tematów pomocy zainstalowanych w systemie, wpisz: Get-Help *. Można
```

1 Pliki pomocy Windows PowerShell są zazwyczaj dostępne tylko w języku angielskim i taki rezultat uzyska Czytelnik na swoim komputerze. W tym miejscu zdecydowałem się umieścić tłumaczenie informacji zwracanych przez polecenie, ale należy pamiętać, że – przynajmniej w czasie pracy nad tą książką – polska wersja informacji pomocy nie była dostępna (wszystkie przypisy pochodzą od tłumacza).

```
    wyświetlić cały temat pomocy lub użyć parametrów cmdlet Get-Help do uzyskania
    wybranych fragmentów, takich jak opis składni, parametrów lub przykłady
    zastosowania.

    Koncepcyjne tematy pomocy w Windows PowerShell zaczynają się od prefiksu
    "about_", na przykład "about_Comparison_Operators". Aby wyświetlić wszystkie
    tematy "about_", wpisz: Get-Help about_*. Aby wyświetlić konkretny temat, wpisz:
    Get-Help about_<topic-name>, na przykład Get-Help about_Comparison_Operators.

    Oprócz stosowania "Get-Help" można również wpisać "help" lub "man",
    co spowowoduje wyświetlanie informacji po jednym ekranie na raz, albo "<cmdlet-
    name> -?", co daje efekt identyczny jak Get-Help, ale działa tylko dla poleceń.

    Get-Help pobiera zawartość pomocy z plików pomocy (help) zainstalowanych
    na komputerze. Bez plików pomocy Get-Help wyświetla tylko podstawowe informacje
    o poleceniach. Niektóre moduły Windows PowerShell są wyposażone w pliki pomocy.
    Jednak począwszy od wersji Windows PowerShell 3.0 moduły dołączone do systemu
    Windows nie zawierają plików pomocy. Aby pobrać lub uaktualnić pliki pomocy dla
    modułu Windows PowerShell 3.0, należy posłużyć się cmdlet Update-Help.

    Zawartość tematów pomocy jest również dostępna w trybie online w TechNet
    Library. Aby uzyskać wersję online tematu pomocy, należy użyć parametru Online,
    na przykład: Get-Help Get-Process -Online, co spowoduje wyświetlenie tematu
    w przeglądarce internetowej. Wszystkie tematy pomocy znajdują się w witrynie
    dostępnej pod adresem http://go.microsoft.com/fwlink/?LinkID=107116.

    Jeśli wpiszesz "Get-Help" uzupełnione o dokładną nazwę tematu pomocy lub słowo
    unikatowe dla tego tematu, zostanie wyświetlona treść tego tematu. Jeśli
    wpiszesz słowo lub wzorzec występujący w wielu tematach, Get-Help wyświetli
    listę pasujących tytułów. Jeśli użyjesz słowa, które nie występuje w żadnym
    tytule tematu pomocy, Get-Help wyświetli listę tematów, które zawierają to słowo
    w swojej treści.

    Get-Help może odczytywać tematy pomocy dla wszystkich obsługiwanych języków.
    Get-Help najpierw szuka plików pomocy dla ustawień językowych zdefiniowanych
    w systemie Windows, następnie dla lokalizacji nadrzędnej, tak jak "pt"
    (portugalski) dla "pt-BR" (portugalski brazylijski), a następnie dla języka
    zastępczego. Począwszy od wersji Windows PowerShell 3.0, jeśli Get-Help nie
    znajdzie pomocy w żadnym z tych języków, wyszuka temat pomocy w języku angielskim
    ("en-US"), zanim zwróci komunikat o błędzie lub wyświetli automatycznie
    wygenerowaną podpowiedź.

    Informacje o symbolach używanych przez Get-Help w diagramach składniowych poleceń
    udostępnia temat about_Command_Syntax. Informacje o atrybutach parametrów, takich
    jak Required lub Position, zawiera temat about_Parameters.

ŁĄCZA POWIĄZANE
    Online Version: http://go.microsoft.com/fwlink/?LinkID=113316
    Get-Command
    Get-Member
    Get-PSDrive
    about_Command_Syntax
    about_Comment_Based_Help
    about_Parameters
```

```
UWAGI
    Aby zobaczyć przykłady, wpisz: "get-help Get-Help -examples".
    Aby uzyskać więcej informacji, wpisz: "get-help Get-Help -detailed".
    Aby uzyskać informacje techniczne, wpisz: "get-help Get-Help -full".
    Aby uzyskać pomoc online, wpisz: "get-help Get-Help -online"
```

Cenną zaletą systemu pomocy Windows PowerShell jest to, że nie tylko wyświetla informacje o składni poleceń, ale udostępnia trzy poziomy wyświetlania: normalny, szczegółowy i pełny. Dodatkowo można uzyskać informacje o koncepcjach działania Windows PowerShell. Funkcja ta jest odpowiednikiem podręcznika instruktażowego online. Aby uzyskać listę wszystkich artykułów koncepcyjnych, należy użyć polecenia *Get-Help about**, zwracającego listę wszystkich tematów z prefiksem *about*.

Wyobraźmy sobie, że nie pamiętamy dokładnej nazwy polecenia cmdlet, którego chcielibyśmy użyć, ale pamiętamy, że to było cmdlet ***get***. W celu odszukania tej nazwy możemy posłużyć się znakiem zastępczym, takim jak gwiazdka (*), jak poniżej:

```
Get-Help get*
```

Technika znaków zastępczych może być dalej rozszerzana. Jeśli pamiętamy, że chodziło o polecenie ***get*** i że pierwszą literą rzeczownika w nazwie było ***p***, możemy użyć następującej składni, aby uzyskać listę pasujących poleceń:

```
Get-Help get-p*
```

Po ustaleniu, o które polecenie chodziło, może pojawić się problem, że nie mamy pewności co do prawidłowej składni. W takim przypadku możemy sięgnąć po argument ***-examples***, który zademonstruje składnię polecenia na przykładach. Poniższe polecenie spowoduje wyświetlenie przykładów zastosowania (z komentarzami) dla polecenia cmdlet *Get-PSDrive*:

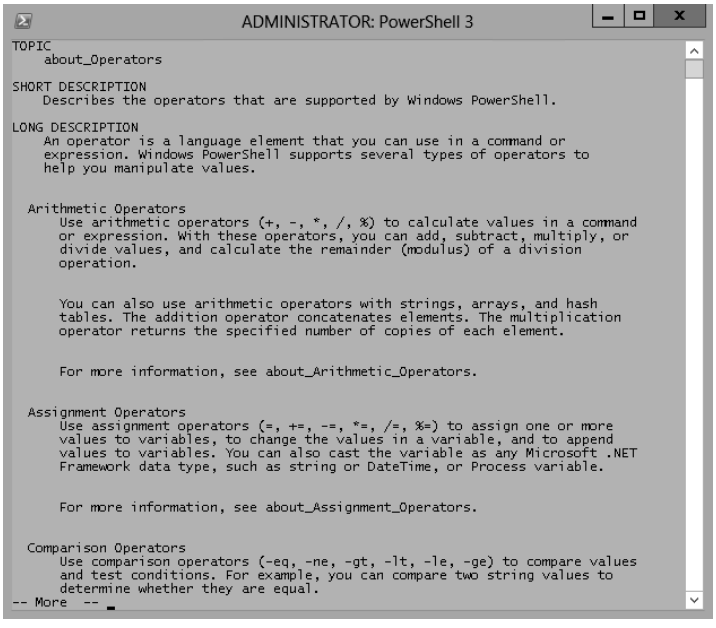
```
Get-Help Get-PSDrive -examples
```

Czytelnik łatwo zauważy, że uzyskane dane znacznie przekraczają pojemność jednego ekranu. Zawartość pomocy można wyświetlać po jednym ekranie na raz, używając funkcji ***Help***. Funkcja ta przesyła wprowadzone przez użytkownika dane do polecenia *Get-Help*, a odbierane informacje zwrotne przekierowuje do narzędzia ***more.com***. Powoduje to wyświetlenie jednego ekranu informacji oraz znaku zachęty *More* (Dalej). Naciśnięcie klawisza Enter wyświetla wówczas kolejne wiersze (po jednym), zaś klawiszem spacji można przewinąć cały kolejny ekran.



Uwaga Trzeba pamiętać, że przy korzystaniu z Windows PowerShell ISE funkcja stronicowania nie będzie działać, zatem nie dostrzeżemy żadnej różnicy pomiędzy *Get-Help* i *Help*. W tym środowisku obydwa polecenia zachowują się tak samo. Jednak należy się spodziewać, że przy pracy w Windows PowerShell ISE użytkownik będzie korzystał raczej z polecenia *Show-Command*, a nie *Get-Help*, aby uzyskać informacje o składni i zastosowaniu.

Rysunek 1-3 prezentuje przykład sformatowanego wyjścia dla tematu pomocy przekraczającego objętość jednego ekranu, wywołanego poprzez funkcję *Help*.



Rysunek 1-3 Wykorzystanie funkcji *Help* do wyświetlania informacji po jednej stronie na raz

Ciągłe wpisywanie *Get-Help* może okazać się męczące – ostatecznie, to aż osiem znaków! Rozwiązaniem jest utworzenie aliasu dla polecenia cmdlet. *Alias* to skrócona nazwa (kombinacja znaków), która wywoła odpowiedni program lub polecenie. Przedstawiona poniżej procedura „Tworzenie aliasu dla cmdlet Get-Help" przypisze to polecenie do kombinacji klawiszy G i H.

Uwaga Przed utworzeniem nowego aliasu warto sprawdzić, czy polecenie cmdlet nie ma już go zdefiniowanego, używając polecenia *Get-Alias*. Następnie można użyć *New-Alias*, aby przypisać poleceniu unikatową kombinację klawiszy.

➜ Tworzenie aliasu dla cmdlet Get-Help

1. Otwórz konsolę Windows PowerShell, wybierając Start | Run | Windows PowerShell.
2. Wywołaj alfabetycznie posortowaną listę aktualnie zdefiniowanych aliasów i sprawdź, czy istnieje alias dla polecenia *Get-Help* oraz czy kombinacja *gh* jest aktualnie wolna. Listę tę można uzyskać poniższym poleceniem:

```
Get-Alias sort
```

3. Po upewnieniu się, że nie istnieje jeszcze alias dla polecenia *Get-Help* i że żadne inne nie ma przypisanego skrótu *gh*, sprawdź składnię dla polecenia *New-Alias*. Użyj argumentu *-full*, aby wyświetlić pełny tekst pomocy:

```
Get-Help New-Alias -full
```

4. Użyj polecenia *New-Alias*, aby przypisać skrót *gh* do polecenia cmdlet *Get-Help*:

```
New-Alias gh Get-Help
```

Poznawanie poleceń: ćwiczenia krok po kroku

W kolejnych ćwiczeniach poznamy posługiwanie się narzędziami wiersza poleceń w Windows PowerShell. Przekonamy się, że korzystanie z tych narzędzi jest równie proste jak w interpreterze CMD; jednak dzięki stosowaniu tych narzędzi w środowisku Windows PowerShell uzyskujemy dostęp do nowych poziomów funkcjonalności.

➜ Korzystanie z narzędzi wiersza poleceń

1. Otwórz konsolę Windows PowerShell, jeśli nie jest jeszcze uruchomiona.
2. Zmień bieżący katalog na główny katalog dysku C:

```
cd c:\
```

3. Wyświetl listę plików w katalogu głównym, używając polecenia *dir*.
4. Utwórz nowy katalog w głównym katalogu dysku C, używając polecenia *md*:

```
md mytest
```

5. Uzyskaj listę wszystkich plików i katalogów zaczynających się od litery *m*.

```
dir m*
```

6. Zmień katalog bieżący na katalog roboczy PowerShell. Do tego celu użyj polecenia cmdlet *Set-Location* i zmiennej $pshome:

```
Set-Location $pshome
```

7. Uzyskaj listę liczników pamięci dotyczących dostępnej wielkości, używając polecenia *typeperf*:[2]

```
typeperf "\memory\available bytes"
```

8. Po wyświetleniu kilku kolejnych odczytów licznika przerwij działanie polecenia, naciskając Ctrl+C.

2 W lokalizowanej (nie-angielskiej) wersji systemu Windows należy użyć zlokalizowanej wersji nazw liczników. W przypadku wersji polskiej odpowiedni licznik nosi nazwę „\pamięć\dostępne bajty” (wielkość liter nie ma znaczenia).

9. Wyświetl bieżącą konfigurację rozruchu za pomocą polecenia *bootcfg* (zwróć uwagę, że to polecenie wymaga uprawnień administracyjnych):

   ```
   bootcfg
   ```

10. Zmień bieżący katalog na C:\Mytest utworzony wcześniej:

    ```
    cd c:\mytest
    ```

11. Utwórz plik o nazwie mytestfile.txt w katalogu C:\Mytest. Użyj narzędzia *fsutil*, aby utworzyć plik o wielkości 1000 bajtów. W tym celu posłuż się następującym poleceniem:

    ```
    fsutil file createnew mytestfile.txt 1000
    ```

12. Wyświetl listę wszystkich plików w katalogu Mytest przy użyciu polecenia cmdlet *Get-ChiltItem*.
13. Wyświetl aktualną datę za pomocą polecenia *Get-Date*.
14. Wyczyść ekran przy użyciu polecenia *cls*.
15. Wyświetl listę wszystkich poleceń cmdlet wbudowanych w Windows PowerShell. W tym celu posłuż się poleceniem *Get-Command*.
16. Użyj polecenia *Get-Command* do pobrania polecenia *Get-Alias*. W tym celu posłuż się argumentem *-name*, podając początek nazwy polecenia jako wartość:

    ```
    Get-Command -name Get-Al*
    ```

    Na tym kończy się to ćwiczenie. Zamknij konsolę Windows PowerShell, wpisując **exit** i naciskając klawisz Enter.

W kolejnym ćwiczeniu użyjemy różnych opcji pomocy do uzyskania informacji o różnych poleceniach cmdlet.

➜ Uzyskiwanie pomocy

1. Uruchom konsolę Windows PowerShell, wybierając Start | Run | Windows PowerShell. Konsola uruchomi się domyślnie w katalogu głównym bieżącego użytkownika.
2. Użyj cmdlet *Get-Help* do uzyskania informacji o poleceniu Get-Help:

   ```
   Get-Help Get-Help
   ```

3. Wyświetl szczegółowe informacje pomocy o poleceniu Get-Help, używając argumentu *-detailed*:

   ```
   Get-Help Get-Help -detailed
   ```

4. Odczytaj pogłębione informacje techniczne o poleceniu Get-Help, używając argumentu *-full*:

```
Get-Help Get-Help -full
```

5. Aby uzyskać tylko listę przykładów użycia polecenia, użyj argumentu *-examples*:

```
Get-Help Get-Help -examples
```

6. Wyświetl listę informacyjnych tematów pomocy, podając prefiks *about* z symbolem wieloznacznym * (gwiazdka) jako parametr:

```
Get-Help Get-Help about*
```

7. Wyświetl listę tematów pomocy dotyczących poleceń *get*. W tym celu użyj jako parametru słowa *get* z symbolem wieloznacznym:

```
Get-Help get*
```

8. Wyświetl listę tematów pomocy dla poleceń *set*.

```
Get-Help set*
```

Na tym kończy się to ćwiczenie. Zamknij konsolę Windows PowerShell, wpisując **exit** i naciskając klawisz Enter.

Podsumowanie rozdziału 1

Aby	Zrób to
Wywołać zewnętrzne polecenie lub narzędzie trybu wiersza poleceń	Wpisz nazwę polecenia (bez rozszerzenia) po znaku zachęty Windows PowerShell.
Wywołać sekwencyjnie wiele zewnętrznych poleceń lub narzędzi	Wpisz kolejne polecenia w jednym wierszu Windows PowerShell, rozdzielając je średnikami.
Uzyskać listę uruchomionych procesów	Użyj polecenia cmdlet *Get-Process*
Zasymulować efekt polecenia cmdlet przed rzeczywistym wykonaniem żądanej akcji	Użyj argumentu *-whatif*
Spowodować uruchomienie Windows PowerShell, wykonanie określonego polecenia cmdlet i zamknięcie PowerShell	Użyj polecenia **PowerShell**, podając jako parametr uruchomieniowy nazwę cmdlet ujętą w nawiasy klamrowe i poprzedzoną symbolem &.
Spowodować wyświetlanie monitu o potwierdzenie zatrzymania procesu	Użyj polecenia cmdlet *Stop-Process* z argumentem *-confirm*.

ROZDZIAŁ 2

Korzystanie z poleceń cmdlet

Po zapoznaniu się z tym rozdziałem Czytelnik będzie umiał:

- Używać poleceń cmdlet Windows PowerShell w podstawowym zakresie
- Wykorzystać *Get-Command* do uzyskania listy poleceń cmdlet
- Konfigurować opcje wyszukiwania poleceń cmdlet
- Konfigurować wyświetlanie wyników
- Korzystać z polecenia *Get-Member*
- Używać polecenia *Show-Command*
- Korzystać z polecenia *New-Object*

Umieszczenie ogromnej liczby poleceń cmdlet w Windows PowerShell sprawia, że powłoka ta jest natychmiast użyteczna dla administratorów sieci i innych osób, które muszą wykonywać najrozmaitsze zadania konserwacyjne i administracyjne w systemach biurkowych i serwerowych. W tym rozdziale poznamy wiele użytecznych poleceń cmdlet, dobranych z myślą o zaprezentowaniu siły i elastyczności Windows PowerShell. Jednak prawdziwą korzyścią, jaką Czytelnik może wynieść z tego rozdziału, jest metodologia, której będziemy używać do odkrywania zastosowań różnych poleceń.

Podstawy poleceń cmdlet

W rozdziale 1, „Przegląd cech Windows PowerShell 3.0”, pokazaliśmy wykorzystanie różnych dostępnych narzędzi pomocy, aby zademonstrować korzystanie z poleceń cmdlet. Przedstawiliśmy kilka poleceń, które są przydatne przy ustalaniu, jakie polecenia są dostępne i jak należy ich używać. W tym podrozdziale pokażemy kilka innych sposobów posługiwania się poleceniami cmdlet w Windows PowerShell.



Uwaga Wpisywanie długich nazw cmdlet jest często nużące, a ponadto podatne na błędy literowe. Aby uprościć ten proces, wystarczy wpisać fragment nazwy polecenia dostateczny do jego jednoznacznego wskazania, po czym nacisnąć klawisz Tab. Co otrzymamy? Samoczynne dopełnienie nazwy (funkcja ta nazywana jest *tab completion* – uzupełnianie tabulatorem). Działa to również dla nazw parametrów i innych wpisywanych elementów, takich jak obiekty .NET, katalogi, klucze rejestru i tak dalej. Warto poeksperymentować z tą techniką. Być może już nigdy więcej nie będzie potrzeby wpisywania ***Get-Command***!

Ponieważ polecenia cmdlet zwracają obiekty, a nie wartości łańcuchów, można otrzymać dodatkowe informacje o uzyskiwanych obiektach. Te informacje nie byłyby dostępne, gdybyśmy posługiwali się tylko danymi tekstowymi. Aby pobrać informacje z jednego polecenia cmdlet i przekazać je do innego, można użyć znaku potoku (|, ang. *pipe*). Może się to wydawać skomplikowane, ale w rzeczywistości jest to bardzo proste i po ukończeniu tego rozdziału będzie całkowicie naturalne. Na najbardziej podstawowym poziomie rozważmy odczytywanie zawartości katalogu; po uzyskaniu listingu plików zapewne chcielibyśmy określić sposób jego wyświetlenia (sformatowania) – na przykład jako tabelę lub listę. Jak można zauważyć, odczytanie informacji o zawartości katalogu i sformatowanie listy to dwie oddzielne operacje. Druga z nich znajdzie się po prawej stronie symbolu potoku.

Korzystanie z cmdlet *Get-ChildItem*

W rozdziale 1 posłużyliśmy się poleceniem ***dir*** do wyświetlenia listy wszystkich folderów i plików w danym katalogu. Polecenie to działa, gdyż istnieje alias wbudowany w Windows PowerShell, który przypisuje cmdlet ***Get-ChildItem*** do sekwencji liter ***dir***.

Uzyskiwanie listingu katalogu

Listę zawartości katalogu dyskowego uzyskamy wpisując w Windows PowerShell polecenie ***Get-ChildItem*** uzupełnione o nazwę katalogu (lub puste, aby uzyskać listing bieżącego katalogu). Przypomnijmy, że można użyć uzupełniania tabulatorem. Wystarczy wpisać **get-ch** i nacisnąć Tab, aby dopełnić nazwę. Oto przykład polecenia:

```
Get-Childitem C:\
```



Uwaga Windows PowerShell nie rozróżnia wielkości liter, zatem ***get-childitem***, ***Get-ChildItem*** i ***GET-CHILDITEM*** zostaną zintepretowane tak samo – dla PowerShell jest to to samo polecenie.

W Windows PowerShell nie istnieje w rzeczywistości cmdlet o nazwie *dir*; powłoka nie odwołuje się też do polecenia *dir* z czasów DOS. Zamiast tego istnieje wbudowany alias *dir* dla polecenia cmdlet *Get-ChildItem*. Dlatego też wynik polecenia *dir* w Windows PowerShell różni się od wyglądu wyniku polecenia *dir* w interpreterze CMD. Przy użyciu polecenia cmdlet *Get-Alias* można sprawdzić, jak wygląda powiązanie pomiędzy *dir* a *Get-ChildItem*:

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

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Alias           dir -> Get-ChildItem
```

Tak więc zarówno użycie *Get-ChildItem*, jak i *dir* zwraca dokładnie tak samo wyglądającą listę, co widać na poniższym przykładzie:

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

    Directory: C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         3/22/2012   4:03 AM            PerfLogs
d-r--         3/22/2012   4:24 AM            Program Files
d----         4/22/2012   7:14 PM            test
d-r--         3/23/2012   6:02 PM            Users
d----         3/23/2012   4:59 PM            Windows
-a---         3/22/2012   4:33 AM         24 autoexec.bat
-a---         3/22/2012   4:33 AM         10 config.sys

PS C:\> Get-ChildItem c:\

    Directory: C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         3/22/2012   4:03 AM            PerfLogs
d-r--         3/22/2012   4:24 AM            Program Files
d----         4/22/2012   7:14 PM            test
d-r--         3/23/2012   6:02 PM            Users
d----         3/23/2012   4:59 PM            Windows
-a---         3/22/2012   4:33 AM         24 autoexec.bat
-a---         3/22/2012   4:33 AM         10 config.sys

PS C:\>
```

Analogicznie, jeśli użyjemy polecenia *Get-Help*, aby uzyskać informacje o *dir*, otrzymamy ten sam opis, co dla polecenia *Get-ChildItem*. Widać to w poniższym przykładzie (skróconym tylko do nazwy i podsumowania).

```
PS C:\> Get-Help dir | select name, synopsis | Format-Table -AutoSize
```

```
Name          Synopsis
----          --------
Get-ChildItem Gets the files and folders in a file system drive.

PS C:\>
```

W Windows PowerShell alias i pełna nazwa cmdlet działają w taki sam sposób. Aliasy nie umożliwiają zmodyfikowania zachowania polecenia cmdlet – do tego celu trzeba użyć własnej, zdefiniowanej przez siebie funkcji lub funkcji proxy.

Formatowanie listingu katalogu przy użyciu polecenia *Format-List*

Jak wspomniano wcześniej, wynik jednego polecenia cmdlet można przekierować do innego za pomocą znaku potoku. Poniższy przykład tworzy listing katalogu i kieruje go do polecenia cmdlet *Format-List*:

```
Get-ChildItem C:\ | Format-List
```

➜ Formatowanie wyjścia przy użyciu polecenia cmdlet *Format-List*

1. Otwórz konsolę Windows PowerShell.
2. Użyj polecenia cmdlet *Get-ChildItem* do uzyskania listingu katalogu głównego dysku C:

   ```
   Get-ChildItem C:\
   ```

3. Użyj cmdlet *Format-List*, aby zmienić uporządkowanie listingu katalogu:

   ```
   Get-ChildItem C:\ | Format-List
   ```

4. Użyj argumentu *-property* polecenia *Format-List*, aby wyświetlić tylko nazwy plików i katalogów:

   ```
   Get-ChildItem C:\ | Format-List -Property Name
   ```

5. Zmodyfikuj polecenie, aby uwzględnić wielkości plików z katalogu głównego:

   ```
   Get-ChildItem C:\ | Format-List -Property Name, Length
   ```

Korzystanie z polecenia cmdlet *Format-Wide*

W taki sam sposób, w jaki wykorzystaliśmy *Format-List* do przekształcenia wyjścia polecenia na listę, można użyć polecenia *Format-Wide* do uzyskania bardziej zwartego wyniku. Różnica polega na tym, że *Format-Wide* umożliwia wybranie tylko jednej właściwości dla wyświetlanych elementów. Możemy jednak określić, w jak wielu kolumnach będą wyświetlane informacje. Domyślnie *Format-Wide* używa dwóch kolumn.

➜ Formatowanie listingu katalogu przy użyciu *Format-Wide*

1. W oknie Windows PowerShell wywołaj cmdlet *Get-ChildItem* z nazwą katalogu, uzupełniając polecenie znakiem potoku i poleceniem *Format-Wide*, jak poniżej:

```
Get-ChildItem C:\ | Format-Wide
```

2. Zmień wyświetlanie na trzykolumnowe i jawnie wskaż wyświetlanie właściwości *name* (nazwa):

```
Get-ChildItem C:\ | Format-Wide -Column 3 -Property Name
```

3. Pozwól Windows PowerShell na wyświetlenie tak wielu kolumn, jak to możliwe dla bieżącej szerokości okna i długości nazw. Służy do tego argument *-Autosize*:

```
Get-ChildItem C:\ | Format-Wide -Property Name -Autosize
```

4. Wymuś obcinanie nazw w kolumnach, wybierając liczbę kolumn większą niż wyświetlona w poprzednim przykładzie (zbyt długie nazwy zostaną obcięte, co sygnalizowane jest wielokropkiem na końcu):

```
Get-ChildItem C:\ | Format-Wide -Column 8 -Property Name
```

➜ Formatowanie wyjścia polecenia przy użyciu *Format-Wide*

1. Uruchom konsolę Windows PowerShell, jeśli nie jest jeszcze otwarta.
2. Użyj polecenia *Get-ChildItem* do wyświetlenia listingu katalogu C:\Windows:

```
Get-ChildItem C:\Windows
```

3. Użyj argumentu *-recurse*, aby nakazać poleceniu *Get-ChildItem* przejście przez całą zagnieżdżoną strukturę katalogu, uwzględniając tylko pliki .txt. Ukryj błędy, używając parametru *-ea* (będącego aliasem dla *ErrorAction*) i przypisując mu wartość 0 (co oznacza, że wszystkie błędy zostaną zignorowane):

```
Get-ChildItem C:\Windows -recurse -include *.txt -ea 0
```

   Wykonanie tego polecenia może trochę potrwać. Fragment wyniku jest przedstawiony poniżej:

```
PS C:\> Get-ChildItem C:\Windows -recurse -include *.txt -ea 0
    Directory: C:\Windows\ehome\en-US

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         3/22/2012   4:17 AM      15826 epgtos.txt
-a---         3/22/2012   4:17 AM      14642 playready_eula.txt
-a---         3/22/2012   4:17 AM      36672 playReady_eula_oem.txt
PS C:\>
```

4. Użyj polecenia cmdlet *Format-Wide*, aby zmienić wygląd danych wyjściowych, używając parametru *-column* i wartości 3:

```
Get-ChildItem C:\Windows -recurse -include *.txt -ea 0 | Format-Wide -column 3
```

Wykonanie tego polecenia da wyniki podobne do poniższych (pokazanych w niewielkim fragmencie):

```
PS C:\> Get-ChildItem C:\Windows -recurse -include *.txt -ea 0 | Format-Wide
-Column 3

    Directory: C:\Windows\ehome\en-US

epgtos.txt                  playready_eula.txt          playReady_eula_oem.txt

    Directory: C:\Windows\SoftwareDistribution\SelfUpdate

wuident.txt
```

5. Użyj polecenia *Format-Wide*, aby zmienić wygląd danych wyjściowych z polecenia *Get-ChildItem*. Tym razem użyjemy argumentu *-property*, aby wyspecyfikować właściwość *name* i pogrupujemy wyniki według wielkości. Pokazany przykład zajmuje dwa wiersze ze względu na szerokość strony, ale w oknie Windows PowerShell jest to jedno polecenie i powinno być wpisane w jednym ciągu. Podczas wpisywania poleceń w oknie PowerShell, po osiągnięciu prawego brzegu okna wiersz jest automatycznie zawijany do następnej linii; nie należy w tym celu naciskać klawisza Enter.

```
Get-ChildItem C:\Windows -recurse -include *.txt -ea 0 | Format-Wide -property
name -groupby lenght -column 3
```

Fragment wyniku pokazany jest poniżej. Zauważmy, że pomimo wskazania 3 kolumn, jeśli nie występują pliki o tej samej długości, użyta została tylko jedna kolumna.

```
PS C:\> Get-ChildItem C:\Windows -recurse -include *.txt -ea 0 |
Format-Wide -Column3 -GroupBy length

    Directory: C:\Windows\ehome\en-US

epgtos.txt

    Directory: C:\Windows\ehome\en-US

playready_eula.txt

    Directory: C:\Windows\ehome\en-US
```

Formatowanie listingu katalogu przy użyciu *Format-Table*

Polecenie cmdlet *Format-Table* pozwala sformatować wyjście innego polecenia jako tabeli. Poniższy przykład przekierowuje wyjście polecenia *Get-ChildItem* do polecenia *Format-Table*:

```
Get-ChildItem C:\ | Format-Table
```

➜ Formatowanie wyjścia przy użyciu polecenia *Format-Table*

1. Otwórz konsolę Windows PowerShell, jeśli nie jest jeszcze uruchomiona.
2. Użyj polecenia cmdlet *Get-ChildItem*, aby uzyskać listing katalogu C:\Windows:

```
Get-ChildItem C:\Windows -ea 0
```

3. Użyj argumentu *-recurse*, aby uzyskać listing wszystkich zagnieżdżonych katalogów i uwzględnij tylko pliki .txt w wyświetlanych wynikach:

```
Get-ChildItem C:\Windows -recurse -include *.txt -ea 0
```

4. Przekieruj wyjście polecenia do cmdlet *Format-Table*, aby móc dostosować wygląd listingu:

```
Get-ChildItem C:\Windows -recurse -include *.txt -ea 0 | Format-Table
```

   Wynik polecenia nie różni się jeszcze od „zwykłego" wyjścia cmdlet *Get-ChildItem*:

```
PS C:\> Get-ChildItem C:\Windows -recurse -include *.txt –ea 0 | Format-Table

    Directory: C:\Windows\ehome\en-US

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         3/22/2012   4:17 AM      15826 epgtos.txt
-a---         3/22/2012   4:17 AM      14642 playready_eula.txt
-a---         3/22/2012   4:17 AM      36672 playReady_eula_oem.txt

    Directory: C:\Windows\SoftwareDistribution\SelfUpdate

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-----         3/13/2012   5:51 PM        275 wuident.txt
```

5. Użyj argumentu *-property* polecenia cmdlet *Format-Table*, wybierając właściwości *name*, *length* oraz *lastwritetime* (czas ostatniego zapisu). Demonstruje to poniższy przykład:

```
Get-ChildItem C:\Windows -recurse -include *.txt -ea 0 | Format-Table -property
name, lenght, lastwritetime
```

   W wyniku otrzymamy tabelę zawierającą nazwy, długość i czas zapisu pliku, jak w poniższym przykładzie:

```
PS C:\> Get-ChildItem C:\Windows -recurse -include *.txt –ea 0 | Format-Table
-property name, length, lastwritetime

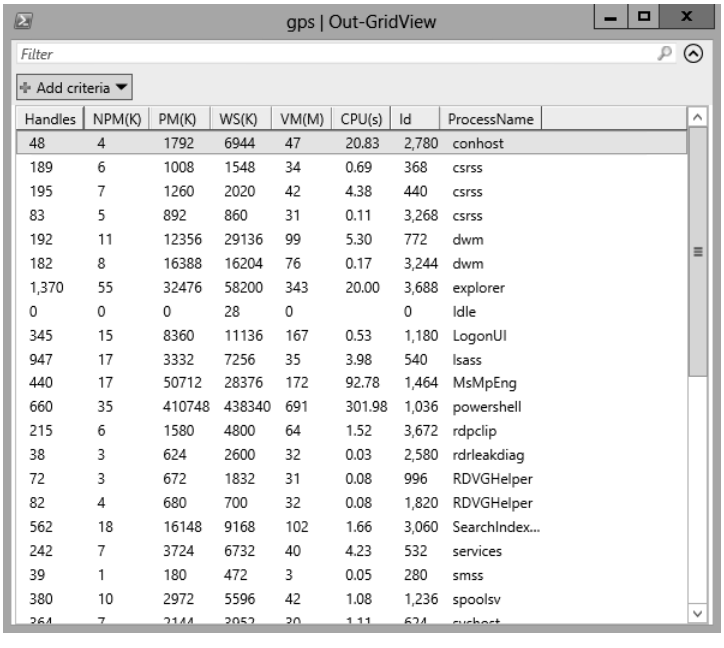
Name                                                Length LastWriteTime
----                                                ------ -------------
epgtos.txt                                           15826 3/22/2012 4:17:20 AM
playready_eula.txt                                   14642 3/22/2012 4:17:20 AM
playReady_eula_oem.txt                               36672 3/22/2012 4:17:20 AM
wuident.txt                                            275 3/13/2012 5:51:28 PM
dberr.txt                                           177095 3/23/2012 4:55:52 PM
```

Formatowanie danych wyjściowych przy użyciu *Out-GridView*

Polecenie cmdlet *Out-GridView* różni się zasadniczo od innych poleceń omawianych dotąd w tej książce. Jest to polecenie interaktywne – nie wykonuje formatowania do wyświetlenia w konsoli Windows PowerShell lub do przesłania wyników na drukarkę. Zamiast tego *Out-GridView* udostępnia kontrolkę pozwalającą na eksplorowanie danych przekazanych przez potok. Kontrolka ta pojawia się jako dodatkowe, pływające okno. Na przykład poniższy przykład przekazuje dane odczytane przez polecenie *Get-Process* do polecenia *Out-GridView* (*gps* to domyślny alias dla polecenia *Get-Process*):

```
gps | Out-GridView
```

Po zakończeniu przetwarzania cmdlet *Get-Process* pojawia się okno z informacjami o procesach uporządkowanych w siatkę wierszy i kolumn, przypominającą arkusz kalkulacyjny. Rysunek 2-1 ukazuje okno siatki zawierające dane procesów. Jedną z użytecznych cech polecenia *Out-GridView* jest fakt, że pasek tytułu okna kontrolki zawiera polecenie, które ją wywołało.

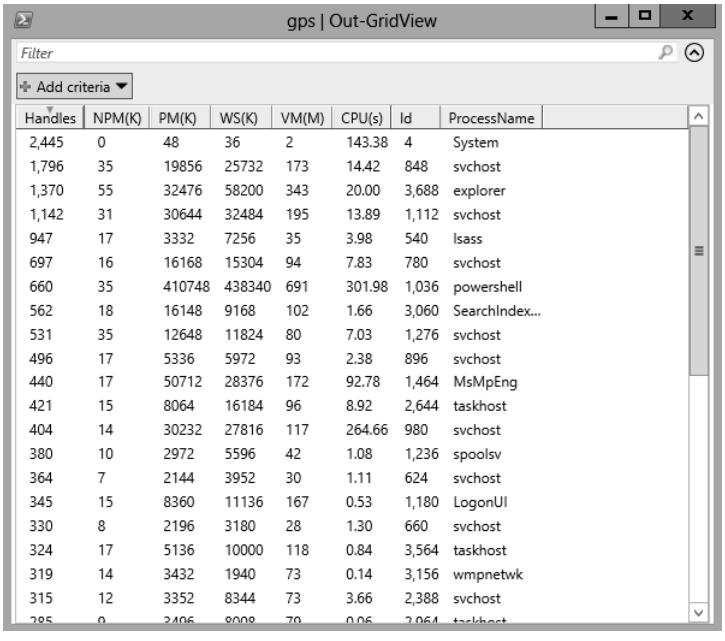


Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
48	4	1792	6944	47	20.83	2,780	conhost
189	6	1008	1548	34	0.69	368	csrss
195	7	1260	2020	42	4.38	440	csrss
83	5	892	860	31	0.11	3,268	csrss
192	11	12356	29136	99	5.30	772	dwm
182	8	16388	16204	76	0.17	3,244	dwm
1,370	55	32476	58200	343	20.00	3,688	explorer
0	0	0	28	0		0	Idle
345	15	8360	11136	167	0.53	1,180	LogonUI
947	17	3332	7256	35	3.98	540	lsass
440	17	50712	28376	172	92.78	1,464	MsMpEng
660	35	410748	438340	691	301.98	1,036	powershell
215	6	1580	4800	64	1.52	3,672	rdpclip
38	3	624	2600	32	0.03	2,580	rdrleakdiag
72	3	672	1832	31	0.08	996	RDVGHelper
82	4	680	700	32	0.08	1,820	RDVGHelper
562	18	16148	9168	102	1.66	3,060	SearchIndex...
242	7	3724	6732	40	4.23	532	services
39	1	180	472	3	0.05	280	smss
380	10	2972	5596	42	1.08	1,236	spoolsv

Rysunek 2-1 Cmdlet *Out-GridView* akceptuje potok z wyjściem innego polecenia i wyświetla kontrolkę umożliwiającą dalszą eksplorację danych.

Kliknięcie nagłówka kolumny powoduje uporządkowanie (sortowanie) danych według zawartości tej kolumny. Kolejne kliknięcia tego samego nagłówka odwraca porządek

(z rosnącego na malejący i odwrotnie). W przykładzie pokazanym na rysunku 2-2 procesy są uporządkowane malejąco według liczby uchwytów (*handles*) używanych przez każdy proces.



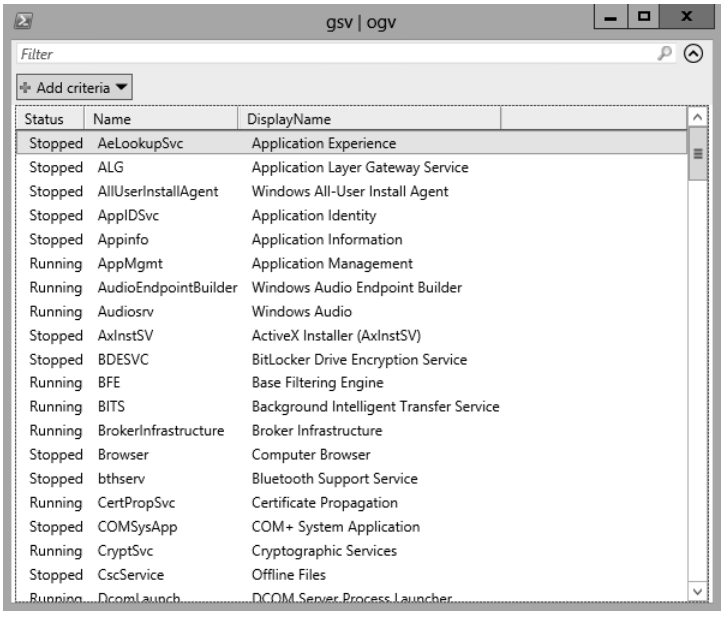
Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
2,445	0	48	36	2	143.38	4	System
1,796	35	19856	25732	173	14.42	848	svchost
1,370	55	32476	58200	343	20.00	3,688	explorer
1,142	31	30644	32484	195	13.89	1,112	svchost
947	17	3332	7256	35	3.98	540	lsass
697	16	16168	15304	94	7.83	780	svchost
660	35	410748	438340	691	301.98	1,036	powershell
562	18	16148	9168	102	1.66	3,060	SearchIndex...
531	35	12648	11824	80	7.03	1,276	svchost
496	17	5336	5972	93	2.38	896	svchost
440	17	50712	28376	172	92.78	1,464	MsMpEng
421	15	8064	16184	96	8.92	2,644	taskhost
404	14	30232	27816	117	264.66	980	svchost
380	10	2972	5596	42	1.08	1,236	spoolsv
364	7	2144	3952	30	1.11	624	svchost
345	15	8360	11136	167	0.53	1,180	LogonUI
330	8	2196	3180	28	1.30	660	svchost
324	17	5136	10000	118	0.84	3,564	taskhost
319	14	3432	1940	73	0.14	3,156	wmpnetwk
315	12	3352	8344	73	3.66	2,388	svchost

Rysunek 2-2 Kliknięcie nagłówka kolumny powoduje posortowanie całej kontrolki według wybranej kolumny w kolejności rosnącej lub malejącej.

Cmdlet *Out-GridView* może przyjmować jako wejście dane z wielu innych poleceń. Rysunek 2-3 (na następnej stronie) demonstruje skutek działania poniższego wyrażenia, zawierającego cmdlet *Get-Service* (alias *gsv*), którego wyjście jest skierowane do wejścia cmdlet *Out-GridView* (zapisanego przy użyciu aliasu *ogv*):

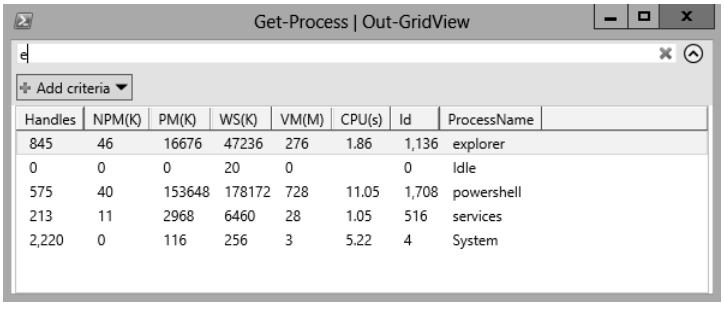
```
gsv | ogv
```

Cmdlet *Out-GridView* automatycznie wykrywa typ danych przekazywanych właściwości, po czym wykorzystuje ten typ do utworzenia przefiltrowanego i posortowanego widoku. Na przykład typ danych dla właściwości *Status* to łańcuch (*string*). Kliknięcie przycisku Add Criteria (Dodaj kryteria), zaznaczenie właściwości *Status* i kliknięcie Add dodaje opcję filtru, która umożliwia różnorodne metody interakcji z tą właściwością. Dostępne opcje obejmują następujące operatory: *contains* (zawiera), *does not contain* (nie zawiera), *starts with* (zaczyna się od), *ends with* (kończy się na) *equals* (jest równe), *does not equal* (nie jest równe), *is empty* (jest pusty) oraz *is not empty* (nie jest pusty). Dla właściwości o innym typie danych dostępny będzie inny zestaw operatorów.



Rysunek 2-3 Informacje o usługach zdefiniowanych w systemie wyświetlane w kontrolce siatki *Out-GridView*

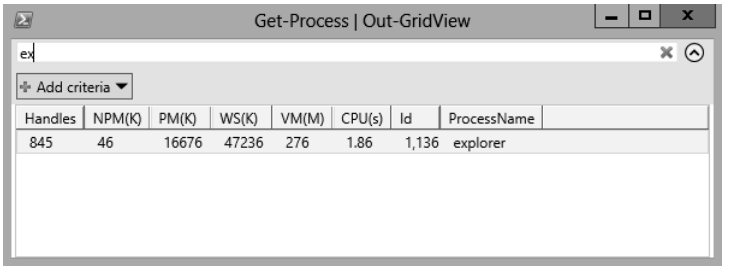
Aby w pokazanym przykładzie wyfiltrować tylko uruchomione usługi, możemy zmienić operator na *equals* i wpisać wartość **running**. Trzeba mieć na uwadze, że wybranie operatora równości wymaga dokładnego wpisywania łańcucha filtrowania, pomijając wielkość liter. Innymi słowy, warunek *equals run* nie zwróci żadnego wyniku, działać będzie tylko warunek *equals running* (lub np. *equals Running*). Z drugiej strony, jeśli wybierzemy np. operator *starts widh*, wystarczy wpisanie jednej litery, aby uzyskać filtrowanie. W przypadkach bardziej urozmaiconych (na przykład przy filtrowaniu nazw usług, a nie ich statusów) można zauważyć dynamiczną zmianę widoku w miarę wpisywania kolejnych znaków w polu filtra (rysunek 2-4).



Rysunek 2-4 Kontrolka *Out-GridView* aktualizuje widok w miarę wprowadzania danych w polu filtra.

Uwaga Należy mieć na uwadze różnice w zachowaniu różnych filtrów. Zależnie od wybranego operatora automatyczna aktualizacja widoku może być niezwykle użyteczna. Działa to szczególnie skutecznie przy próbach filtrowania danych numerycznych, jeśli nie mamy pewności, jakie są typowe wartości i w jakich przedziałach mogą się znajdować.

Po wpisaniu pierwszych dwóch liter nazwy *explorer* w polu filtra zobaczmy już tylko pojedynczą pasującą nazwę procesu, co prezentuje rysunek 2-5.



Rysunek 2-5 Kliknięcie czerwonego znaku X na prawo od pola filtra usuwa filtr i przywraca wyświetlanie wszystkich danych.

➜ Filtrowanie procesów na podstawie właściwości

1. Otwórz kontrolkę *Out-GridView* dla listy procesów:

   ```
   gps | ogv
   ```

2. Kliknij przycisk Add Criteria i zaznacz pole wyboru CPU(s), po czym kliknij Add.
3. Kliknij podkreślone słowo contains i wybierz opcję not empty z listy wyboru. Na liście pozostaną tylko procesy z niepustą wartością (czyli te, które miały niepusty czas procesora w chwili wywołania kontrolki).
4. Kliknij przycisk Add Criteria i zaznacz pole wyboru WS(K) (Working Set – zestaw roboczy), po czym kliknij Add.
5. Zmień operator filtrowania z is less then or equal to (jest mniejsze lub równe) na is greater then or equal (jest większe lub równe).
6. Wpisz liczbę **20000** w pole kryterium obok operatora. Zwróć uwagę, że w miarę wpisywania kolejnych cyfr lista wyświetlanych procesów jest stopniowo zawężana.

➜ Tworzenie posortowanej listy procesów

1. Wpisz poniższe polecenie w konsoli Windows PowerShell:

   ```
   Get-Process
   ```

2. Prześlij wyjście polecenia ***Get-Process*** do cmdlet ***Get-Member***:

```
Get-Process | Get-Member
```

3. Przeanalizuj listę dostępnych właściwości (*property*). Zwróć uwagę, że CPU (procesor) jest właściwością typu *scriptproperty* (łańcuchową).
4. Przekieruj rezultaty polecenia ***Get-Process*** do polecenia ***Sort-Object***, używając właściwości ***cpu***:

```
Get-Process | Sort-Object cpu
```

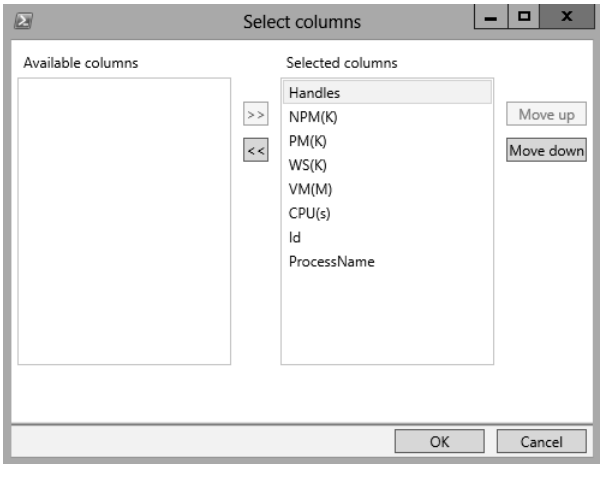
5. Wywołaj ponownie poprzednie polecenie, naciskając klawisz ze strzałką w górę, po czym dodaj parametr ***-Descending*** (malejąco):

```
Get-Process | Sort-Object cpu – Descending
```

6. Prześlij uzyskany rezultat do polecenia cmdlet ***Out-GridView***. Odpowiednie polecenie wygląda następująco:

```
Get-Process | Sort-Object cpu – Descending | Out-GridView
```

7. Prawym klawiszem myszy kliknij nagłówek kolumny i wybierz polecenie Select columns. Pojawi się okno dialogowe pokazane na rysunku 2-6



Rysunek 2-6 Okno dialogowe Select Columns pozwala wyłączyć wyświetlanie części kolumn dostępnych w kontrolce *gridview*.



Uwaga Ponieważ proces wybierania kolumn jest dość powolny, jeśli potrzebujemy tylko wybranych kolumn, lepszą metodą jest użycie polecenia ***Select-Object*** do odfiltrowania potrzebnych kolumn, zanim prześlemy je do polecenia ***Out-GridView***.

Korzystanie z polecenia *Get-Command*

Polecenie cmdlet *Get-Command* zwraca szczegółowe informacje o każdym dostępnym poleceniu. Do poleceń tych zaliczają się cmdlet, funkcje, przepływy, aliasy oraz pliki wykonywalne (narzędzia zewnętrzne). Przy użyciu *Get-Command* można uzyskać listę wszystkich poleceń cmdlet zainstalowanych w Windows PowerShell, ale polecenie to jest znacznie bardziej elastyczne i umożliwia wykonywanie znacznie bardziej złożonych działań, niż tylko zwracanie prostej listy. Możemy na przykład użyć symboli wieloznacznych w wyszukiwaniu poleceń. Zademonstrujemy to w kolejnym ćwiczeniu.

Wyszukiwanie poleceń cmdlet za pomocą symboli wieloznacznych

Wpisanie *Get-Command* uzupełnione o symbol * zwraca bardzo długą listę dostępnych poleceń. Zwróćmy uwagę, że nie ogranicza się ona do poleceń, funkcji i aliasów Windows PowerShell – znajdziemy na niej również wszystkie pliki aplikacji (.exe), które system znalazł w zmiennej środowiskowej PATH (inaczej mówiąc, te, które można wywołać bezpośrednio, bez podawania ścieżki dostępu do pliku wykonywalnego).

```
Get-Command *
```

➜ Wyszukiwanie poleceń przy użyciu cmdlet *Get-Command*

1. Otwórz konsolę Windows PowerShell, jeśli nie jest uruchomiona.
2. Wyszukaj alias dla polecenia *Get-Command*. W tym celu użyj polecenia *Get-Alias*:

   ```
   Get-Alias g*
   ```

   Polecenie to utworzy listę wszystkich zdefiniowanych aliasów, które rozpoczynają się od litery *g*. Fragment tej listy jest pokazany poniżej:

   ```
   CommandType     Name                                               ModuleName
   -----------     ----                                               ----------
   Alias           gal -> Get-Alias
   Alias           gbp -> Get-PSBreakpoint
   Alias           gc -> Get-Content
   Alias           gci -> Get-ChildItem
   Alias           gcm -> Get-Command
   Alias           gcs -> Get-PSCallStack
   Alias           gdr -> Get-PSDrive
   Alias           ghy -> Get-History
   Alias           gi -> Get-Item
   Alias           gjb -> Get-Job
   Alias           gl -> Get-Location
   Alias           gm -> Get-Member
   Alias           gmo -> Get-Module
   Alias           gp -> Get-ItemProperty
   ```

```
Alias           gps -> Get-Process
Alias           group -> Group-Object
Alias           gsn -> Get-PSSession
Alias           gsnp -> Get-PSSnapin
Alias           gsv -> Get-Service
Alias           gu -> Get-Unique
Alias           gv -> Get-Variable
Alias           gwck -> Get-WmiKey                                  HSGWMImoduleV6
Alias           gwcm -> Get-WmiClassMethods                         HSGWMImoduleV6
Alias           gwcp -> Get-WmiClassProperties                      HSGWMImoduleV6
Alias           gwcq -> Get-WMIClassesWithQualifiers                HSGWMImoduleV6
Alias           gwkv -> Get-WmiKeyvalue                             HSGWMImoduleV6
Alias           gwmi -> Get-WmiObject
Alias           gwq -> Get-WmiClassesAndQuery                       HSGWMImoduleV6
```

3. Jak można zauważyć, alias dla polecenia *Get-Command* to *gcm*. W dalszej części ćwiczenia będziemy korzystać z tego aliasu.

Na początek użyjemy polecenia *Get-Command* do zwrócenia informacji o nim samym:

```
gcm Get-Command
```

Wynik tego polecenia jest widoczny poniżej:

```
CommandType     Name                                ModuleName
-----------     ----                                ----------
Cmdlet          Get-Command                         Microsoft.PowerShell.Core
```

Jak pokażemy za chwilę, polecenie zwraca w rzeczywistości znacznie więcej informacji, jednak trzeba je specjalnie sformatować, aby móc je wyświetlić.

4. Przekieruj wyniki działania polecenia *Get-Command* do polecenia cmdlet *Format-List*, używając znaku wieloznacznego * do odczytania wszystkich właściwości polecenia *Get-Command*. Kod ten jest pokazany poniżej:

```
gcm Get-Command | Format-List *
```

Polecenie to zwróci wszystkie właściwości polecenia cmdlet *Get-Command*, jak poniżej:

```
HelpUri             : http://go.microsoft.com/fwlink/?LinkID=113309
DLL                 : C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\System.
                      Management.Automation\v4.0_3.0.0.0__31bf3856ad364e35\
                      System.Management.Automation.dll
Verb                : Get
Noun                : Command
HelpFile            : System.Management.Automation.dll-Help.xml
PSSnapIn            : Microsoft.PowerShell.Core
ImplementingType    : Microsoft.PowerShell.Commands.GetCommandCommand
Definition          : Get-Command [[-ArgumentList] <Object[]>] [-Verb <string[]>]
                      [-Noun <string[]>] [-Module <string[]>] [-TotalCount <int>]
                      [-Syntax] [-All] [-ListImported] [-ParameterName
```

```
                    <string[]>] [-ParameterType <PSTypeName[]>]
                    [<CommonParameters>]

                    Get-Command [[-Name] <string[]>] [[-ArgumentList]
<Object[]>]
                    [-Module <string[]>] [-CommandType <CommandTypes>]
                    [-TotalCount <int>] [-Syntax] [-All] [-ListImported]
                    [-ParameterName <string[]>] [-ParameterType <PSTypeName[]>]
                    [<CommonParameters>]

DefaultParameterSet : CmdletSet
OutputType          : {System.Management.Automation.AliasInfo,
                      System.Management.Automation.ApplicationInfo,
                      System.Management.Automation.FunctionInfo,
                      System.Management.Automation.CmdletInfo...}
Options             : ReadOnly
Name                : Get-Command
CommandType         : Cmdlet
Visibility          : Public
ModuleName          : Microsoft.PowerShell.Core
Module              :
RemotingCapability  : PowerShell
Parameters          : {[Name, System.Management.Automation.ParameterMetadata],
                      [Verb, System.Management.Automation.ParameterMetadata],
                      [Noun, System.Management.Automation.ParameterMetadata],
                      [Module, System.Management.Automation.ParameterMetadat...}
ParameterSets       : {[[-ArgumentList] <Object[]>] [-Verb <string[]>] [-Noun
                      <string[]>] [-Module <string[]>] [-TotalCount <int>]
                      [-Syntax] [-All] [-ListImported] [-ParameterName
                       <string[]>]
                      [-ParameterType <PSTypeName[]>] [<CommonParameters>],
                      [[-Name] <string[]>] [[-ArgumentList] <Object[]>] [-Module
                      <string[]>] [-CommandType <CommandTypes>] [-TotalCount
                      <int>]
                      [-Syntax] [-All] [-ListImported] [-ParameterName
                      <string[]>]
                      [-ParameterType <PSTypeName[]>] [<CommonParameters>]}
```

5. Uzyskany wynik jest zbyt obszerny i przez to mało czytelny. W kolejnej próbie użyj argumentu *-property*, specyfikując *definition*, aby uzyskać jedynie definicję polecenia. Zamiast przepisywania całego wyrażenia, naciśnij strzałkę w górę, aby przywrócić poprzednio wpisane polecenie *gcm Get-Command | Format-List **. Użyj klawisza Backspace, aby usunąć gwiazdkę, po czym dopisz *-property definition* (przypomnijmy o uzupełnianiu tabulatorem – wystarczy wpisać *-p* i nacisnąć tabulator, aby uzyskać wpis *-Property*):

```
gcm Get-Command | Format-List -Property definition
```

   Wynikiem polecenia jest definicja polecenia (w tym przypadku *Get-Command*), pokazana wcześniej.

6. Jak wspomnieliśmy wcześniej, polecenia cmdlet zwracają obiekty, a nie po prostu dane tekstowe. Oznacza to, że można uzyskać definicję polecenia bezpośrednio

używając właściwości *definition* i składni obiektowej (notacji kropkowej), jak poniżej:

```
(gcm Get-Command).definition
```

Wynik tego polecenia jest (z wyjątkiem formatowania) identyczny z uzyskanym wcześniej poprzez cmdlet *Format-List*.

7. Użyj aliasu *gcm* z argumentem *-verb* (czasownik). Jako wartości dla argumentu wpisz *se**, jak poniżej:

```
gcm -verb se*
```

Polecenie to zwraca listę wszystkich funkcji i poleceń cmdlet, które zawierają człon czasownika rozpoczynający się od *se*. Wynik ten wygląda podobnie do poniższego (listing ten został skrócony – pełna lista zajęłaby kilkanaście stron książki):

```
CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        Set-BCAuthentication                               BranchCache
Function        Set-BCCache                                        BranchCache
Function        Set-BCDataCacheEntryMaxAge                         BranchCache
Function        Set-BCMinSMBLatency                                BranchCache
Function        Set-BCSecretKey                                    BranchCache
Function        Set-ClusteredScheduledTask                         ScheduledTasks
Function        Set-CmdLetBoldFace                                 DocModule
Function        Set-Disk                                           Storage
Function        Set-DnsClient                                      DnsClient
Function        Set-DnsClientGlobalSetting                         DnsClient
Function        Set-DnsClientNrptGlobal                            DnsClient
Function        Set-DnsClientNrptRule                              DnsClient
Function        Set-DnsClientServerAddress                         DnsClient
Function        Set-DtcAdvancedHostSetting                         MsDtc
Function        Set-DtcAdvancedSetting                             MsDtc
Function        Set-DtcClusterDefault                              MsDtc
Function        Set-DtcClusterTMMapping                            MsDtc
Function        Set-DtcDefault                                     MsDtc
Function        Set-DtcLog                                         MsDtc
Function        Set-DtcNetworkSetting                              MsDtc
Function        Set-DtcTransaction                                 MsDtc
Function        Set-DtcTransactionsTraceSession                    MsDtc
Function        Set-DtcTransactionsTraceSetting                    MsDtc
Function        Set-FileIntegrity                                  Storage
Function        Set-InitiatorPort                                  Storage
Function        Set-IscsiChapSecret                                iSCSI
...
```

Zwróćmy uwagę, że lista ta zależy od wersji systemu operacyjnego oraz zainstalowanych ról lub oprogramowania – jak wspomnieliśmy wcześniej, programy takie jak Microsoft SQL Server lub Microsoft Exchange instalują własne moduły PowerShell.

8. Użyj aliasu *gcm* z argumentem *-noun* (rzeczownik), podając jako wartość *o**, jak poniżej:

```
gcm -noun o*
```

To polecenie zwróci listę wszystkich funkcji i poleceń cmdlet, których człon rzeczownika rozpoczyna się od litery *o*. Wynik będzie podobny do poniższego:

```
CommandType     Name                                ModuleName
-----------     ----                                ----------
Function        Add-OdbcDsn                         Wdac
Function        Disable-OdbcPerfCounter             Wdac
Function        Enable-OdbcPerfCounter              Wdac
Function        Get-OdbcDriver                      Wdac
Function        Get-OdbcDsn                         Wdac
Function        Get-OdbcPerfCounter                 Wdac
Function        Get-OffloadDataTransferSetting      Storage
Function        Get-OptimalSize                     BasicFunctions
Function        Remove-OdbcDsn                      Wdac
Function        Set-OdbcDriver                      Wdac
Function        Set-OdbcDsn                         Wdac
Cmdlet          Compare-Object                      Microsoft.Powe...
Cmdlet          ForEach-Object                      Microsoft.Powe...
Cmdlet          Group-Object                        Microsoft.Powe...
Cmdlet          Measure-Object                      Microsoft.Powe...
Cmdlet          New-Object                          Microsoft.Powe...
Cmdlet          Register-ObjectEvent                Microsoft.Powe...
Cmdlet          Select-Object                       Microsoft.Powe...
Cmdlet          Sort-Object                         Microsoft.Powe...
Cmdlet          Tee-Object                          Microsoft.Powe...
Cmdlet          Where-Object                        Microsoft.Powe...
Cmdlet          Write-Output                        Microsoft.Powe...
```

9. Odczytaj tylko składnię polecenia *Get-Command*, używając argumentu *-syntax*. Odpowiednie wyrażenie jest przedstawione poniżej:

```
gcm -syntax Get-Command
```

W wyniku tego wyrażenia zostanie wyświetlona składnia polecenia *Get-Command*, jak poniżej:

```
Get-Command [[-ArgumentList] <Object[]>] [-Verb <String[]>] [-Noun <String[]>]
[-PSSnapin <String[]>] [-TotalCount <Int32>] [-Syntax] [-Verbose] [-Debug]
 [-ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-OutVariable
 <String>] [-OutBuffer <Int32>]
Get-Command [[-Name] <String[]>] [[-ArgumentList] <Object[]>] [-CommandType
 <CommandTypes>] [-TotalCount <Int32>] [-Syntax] [-Verbose] [-Debug]
 [-ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-OutVariable
 <String>] [-OutBuffer <Int32>]
```

10. Spróbuj użyć jedynie aliasów, aby uzyskać składnię polecenia *Get-Command*:

```
gcm -syntax gcm
```

Wynikiem tego wyrażenia nie jest opis składni. Zamiast tego uzyskujemy mało zadowalający rezultat pokazany poniżej:

```
Get-Command
```

Wynika to z faktu, że *gcm* podane jako parametr dla argumentu nie jest poleceniem cmdlet, ale aliasem i jego jedynym atrybutem jest nazwa właściwego polecenia.

Na tym kończy się to ćwiczenie. Zamknij konsolę Windows PowerShell, wpisując **exit** i naciskając klawisz Enter.

Szybki test

P. Jakiego polecenia należy użyć, aby odczytać definicję cmdlet *Get-Command*, posługując się obiektową notacją kropkową?

O. *(gcm Get-command).definition*

Korzystanie z polecenia *Get-Member*

Polecenie cmdlet *Get-Member* odczytuje informacje o elementach członkowskich obiektu. Choć może nie wydawać się to nadmiernie interesujące, trzeba pamiętać, że wszystko, co zwracają polecenia cmdlet, to obiekty, zatem można wykorzystać *Get-Member* do poznania metod i właściwości danego obiektu. Jeśli użyjemy na przykład polecenia *Get-Member* do wyników polecenia *Get-ChildItem* w systemie plików, uzyskamy listing wszystkich metod i właściwości, których można użyć względem obiektów klas *DirectoryInfo* lub *FileInfo*.

Korzystanie z polecenia *Get-Member* do badania właściwości i metod

Wynik dowolnego polecenia cmdlet można przekierować do polecenia *Get-Member*, aby zbadać właściwości i metody zwracanego obiektu. Dla przykładu użyjmy polecenia *Get-ChildItem* dla folderu dyskowego i przekażemy wyjście do *Get-Member*:

```
Get-ChildItem C:\ | Get-Member
```

Obiekty, właściwości i metody

Jedną z fundamentalnych cech Windows PowerShell jest fakt, że polecenia cmdlet zwracają obiekty. Czym jednak jest obiekt? Jest to byt, co daje nam możliwość bądź opisania czegoś, bądź zrobienia czegoś. Jeśli nie mamy potrzeby opisywania ani zrobienia czegoś, nie ma żadnego powodu, aby tworzyć obiekt. Zależnie od okoliczności, możemy być bardziej zainteresowani metodami albo właściwościami. Te pierwsze umożliwiają zrobienie czegoś. Te drugie zapewniają możliwość opisania.

Dla przykładu rozważmy procedurę wynajmowania samochodu. Wiele podróżowałem, gdy pracowałem jako konsultant w firmie Microsoft, i bardzo często musiałem wynająć samochód.

Aby ująć to językiem programisty, gdy przybywałem na lotnisko, musiałem udać się do stanowiska wynajmu, gdzie używałem polecenia cmdlet *New-Object*, aby utworzyć obiekt *wynajętySamochód*. Od tego momentu interesowały mnie tylko metody dostępne dla obiektu *wynajętySamochód*. Potrzebowałem metod *JedźProstoTąDrogą*, *StańNaCzerwonym*, a być może także *OdtwarzajMuzykę*. Jednak nie interesowały mnie właściwości tego obiektu.

W domu posiadam mały, zgrabny sportowy samochód. Dysponuje on tymi samymi metodami, co obiekt *wynajętySamochód*, ale utworzyłem obiekt *sportowySamochód* głównie ze względu na jego właściwości. Jest zielony i ma aluminiowe felgi, składany dach oraz 3,5-litrowy silnik. Choć ma te same metody, co *wynajętySamochód*, decydującym czynnikiem utworzenia tego obiektu były właściwości, a nie metody.

➜ Korzystanie z polecenia cmdlet *Get-Member*

1. Otwórz konsolę Windows PowerShell, jeśli nie była wcześniej uruchomiona.
2. Zlokalizuj alias odpowiadający poleceniu *Get-Member*, korzystając z polecenia *Get-Alias*:

```
Get-Alias g*
```

3. Przeglądając listę uzyskanych wyników możemy zauważyć, że alias dla polecenia *Get-Member* to *gm*. Przy okazji można też spostrzec (i zapamiętać), że poleceniu *Get-Alias* odpowiada skrót *gal*, zaś poleceniu *Get-ChildItem* – skrót *gci*.

   Przeglądanie uzyskanej listy można sobie uprościć, porządkując ją według właściwości *definition*. W tym celu możemy użyć polecenia cmdlet *Sort-Object*, jak poniżej:

```
gal g* | Sort-Object -property definition
```

W rezultacie otrzymamy listę par alias – cmdlet, posortowaną według nazw poleceń, a nie skrótów, co ułatwia wyszukiwanie, jak poniżej:

```
CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Alias           gal -> Get-Alias
Alias           gci -> Get-ChildItem
Alias           gcm -> Get-Command
Alias           gc -> Get-Content
Alias           ghy -> Get-History
Alias           gi -> Get-Item
Alias           gp -> Get-ItemProperty
Alias           gjb -> Get-Job
Alias           gl -> Get-Location
Alias           gm -> Get-Member
Alias           gmo -> Get-Module
Alias           gps -> Get-Process
Alias           gdr -> Get-PSDrive
Alias           gsn -> Get-PSSession
Alias           gsv -> Get-Service
Alias           gu -> Get-Unique
Alias           gv -> Get-Variable
Alias           gwq -> Get-WmiClassesAndQuery                      HSGWMImoduleV6
Alias           gwcq -> Get-WMIClassesWithQualifiers               HSGWMImoduleV6
Alias           gwcm -> Get-WmiClassMethods                        HSGWMImoduleV6
Alias           gwcp -> Get-WmiClassProperties                     HSGWMImoduleV6
Alias           gwck -> Get-WmiKey                                 HSGWMImoduleV6
Alias           gwkv -> Get-WmiKeyvalue                            HSGWMImoduleV6
Alias           group -> Group-Object
```

4. Użyj aliasu dla polecenia cmdlet *Get-ChildItem* i przekieruj wynik do aliasu polecenia *Get-Member*, jak poniżej:

```
gci | gm
```

5. Aby wyświetlić tylko właściwości dostępne dla obiektów zwracanych przez *Get-ChildItem*, użyj argumentu *-membertype* z wartością *property* (właściwość). Tym razem wpisując polecenie posłuż się uzupełnianiem, zamiast używania aliasów, jak poniżej:

```
Get-ChildItem -Force | Get-Member -membertype property
```

Wyjście tego polecenia jest pokazane poniżej:

```
   TypeName: System.IO.DirectoryInfo

Name              MemberType Definition
----              ---------- ----------
Attributes        Property   System.IO.FileAttributes Attributes {get;set;}
CreationTime      Property   datetime CreationTime {get;set;}
CreationTimeUtc   Property   datetime CreationTimeUtc {get;set;}
Exists            Property   bool Exists {get;}
Extension         Property   string Extension {get;}
FullName          Property   string FullName {get;}
```

```
LastAccessTime    Property   datetime LastAccessTime {get;set;}
LastAccessTimeUtc Property   datetime LastAccessTimeUtc {get;set;}
LastWriteTime     Property   datetime LastWriteTime {get;set;}
LastWriteTimeUtc  Property   datetime LastWriteTimeUtc {get;set;}
Name              Property   string Name {get;}
Parent            Property   System.IO.DirectoryInfo Parent {get;}
Root              Property   System.IO.DirectoryInfo Root {get;}

   TypeName: System.IO.FileInfo

Name              MemberType Definition
----              ---------- ----------
Attributes        Property   System.IO.FileAttributes Attributes {get;set;}
CreationTime      Property   datetime CreationTime {get;set;}
CreationTimeUtc   Property   datetime CreationTimeUtc {get;set;}
Directory         Property   System.IO.DirectoryInfo Directory {get;}
DirectoryName     Property   string DirectoryName {get;}
Exists            Property   bool Exists {get;}
Extension         Property   string Extension {get;}
FullName          Property   string FullName {get;}
IsReadOnly        Property   bool IsReadOnly {get;set;}
LastAccessTime    Property   datetime LastAccessTime {get;set;}
LastAccessTimeUtc Property   datetime LastAccessTimeUtc {get;set;}
LastWriteTime     Property   datetime LastWriteTime {get;set;}
LastWriteTimeUtc  Property   datetime LastWriteTimeUtc {get;set;}
Length            Property   long Length {get;}
Name              Property   string Name {get;}
```

6. Użyj argumentu *-membertype* dla polecenia cmdlet *Get-Member*, aby wyświetlić listę metod dostępnych dla obiektu zwracanego przez polecenie *Get-ChildItem*. W tym celu należy podać wartość *method* jako parametr argumentu, jak poniżej:

```
Get-ChildItem | Get-Member -membertype method
```

W wyniku tego polecenia uzyskamy listę wszystkich metod. Skrócony fragment tego listingu jest pokazany poniżej (zwróćmy uwagę, że informacje zawarte w kolumnie *Definition* są obcięte ze względu na szerokość okna konsoli):

```
   TypeName: System.IO.DirectoryInfo

Name                      MemberType Definition
----                      ---------- ----------
Create                    Method     void Create(), void Create(System.S...
CreateObjRef              Method     System.Runtime.Remoting.ObjRef Crea...
CreateSubdirectory        Method     System.IO.DirectoryInfo CreateSubdi...
Delete                    Method     void Delete(), void Delete(bool rec...
EnumerateDirectories      Method     System.Collections.Generic.IEnumera...
EnumerateFiles            Method     System.Collections.Generic.IEnumera...
EnumerateFileSystemInfos  Method     System.Collections.Generic.IEnumera...
Equals                    Method     bool Equals(System.Object obj)
GetAccessControl          Method     System.Security.AccessControl.Direc...
GetDirectories            Method     System.IO.DirectoryInfo[] GetDirect...
GetFiles                  Method     System.IO.FileInfo[] GetFiles(strin...
...
```

7. Użyj strzałki w górę, aby wyświetlić poprzednie polecenie i zmień parametr *method* na *m**; wynik polecenia będzie dokładnie taki sam, gdyż jedynym typem członkowskim zaczynającym się na literę *m* jest *method*. Przykład ten demonstruje jednak możliwość stosowania symboli wieloznacznych w poleceniu:

```
Get-ChildItem | Get-Member -membertype m*
```

8. Użyj argumentu *-inputobject* w poleceniu *Get-Member*, aby odczytać definicje członkowskie dla każdej właściwości lub metody na liście. Polecenie będzie wyglądać następująco:

```
Get-Member -inputobject Get-ChildItem
```

Wynik (skrócony) polecenia jest pokazany poniżej:

```
PS C:\> Get-Member -inputobject Get-ChildItem

   TypeName: System.String

Name             MemberType            Definition
----             ----------            ----------
Clone            Method                System.Object Clone()
CompareTo        Method                System.Int32 CompareTo(Object value),...
Contains         Method                System.Boolean Contains(String value)
CopyTo           Method                System.Void CopyTo(Int32 sourceIndex,...
EndsWith         Method                System.Boolean EndsWith(String value)...
Equals           Method                System.Boolean Equals(Object obj), Sy...
GetEnumerator    Method                System.CharEnumerator GetEnumerator()
GetHashCode      Method                System.Int32 GetHashCode()
GetType          Method                System.Type GetType()
GetTypeCode      Method                System.TypeCode GetTypeCode()
get_Chars        Method                System.Char get_Chars(Int32 index)
get_Length       Method                System.Int32 get_Length()
Insert           Method                System.String Insert(Int32 startIndex...
IsNormalized     Method                System.Boolean IsNormalized(), System...
LastIndexOf      Method                System.Int32 LastIndexOf(Char value, ...
LastIndexOfAny   Method                System.Int32 LastIndexOfAny(Char[] an...
Normalize        Method                System.String Normalize(), System.Str...
PadLeft          Method                System.String PadLeft(Int32 totalWidt...
PadRight         Method                System.String PadRight(Int32 totalWid...
Remove           Method                System.String Remove(Int32 startIndex...
Replace          Method                System.String Replace(Char oldChar, C...
Split            Method                System.String[] Split(Params Char[] s...
Substring        Method                System.String Substring(Int32 startIn...
ToCharArray      Method                System.Char[] ToCharArray(), System.C...
ToLower          Method                System.String ToLower(), System.Strin...
ToLowerInvariant Method                System.String ToLowerInvariant()
ToString         Method                System.String ToString(), System.Stri...
ToUpper          Method                System.String ToUpper(), System.Strin...
ToUpperInvariant Method                System.String ToUpperInvariant()
Chars            ParameterizedProperty System.Char Chars(Int32 index) {get;}
Length           Property              System.Int32 Length {get;}
```

Na tym kończy się ćwiczenie z poleceniem cmdlet *Get-Member*.

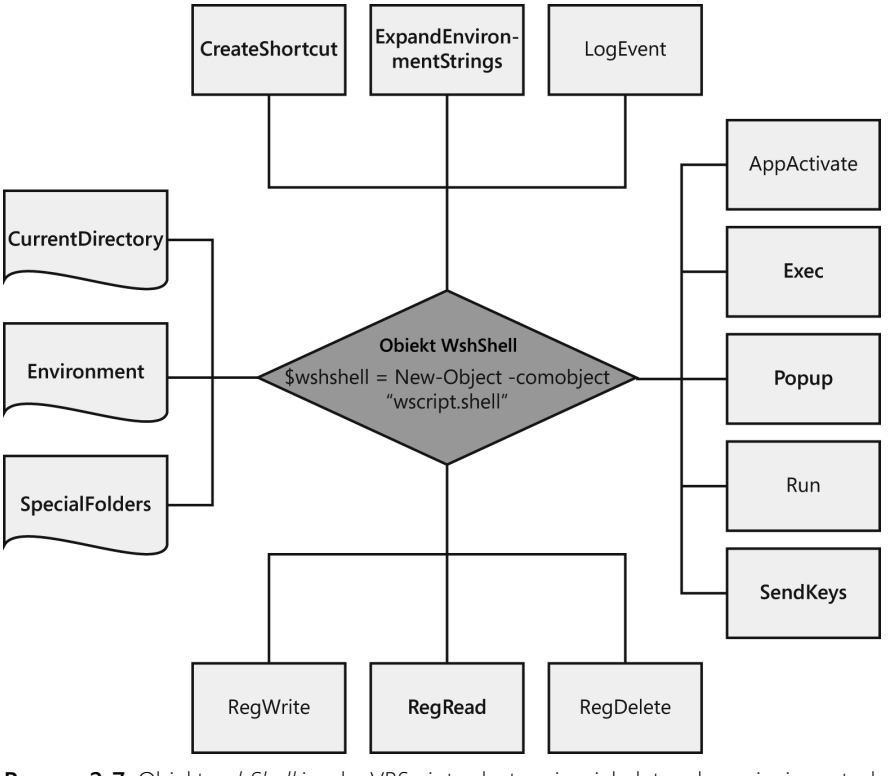
Szybki test

P. Jakiego polecenia należy użyć, aby odczytać listę aliasów rozpoczynających się na literę *g*, posortowaną według właściwości *definition*?

O. *gal g* | Sort-Object -property definition*

Korzystanie z polecenia *New-Object*

Stosowanie obiektów Windows PowerShell udostępnia wiele atrakcyjnych możliwości wykonywania działań, które nie są wbudowane w sam mechanizm PowerShell. Możemy sobie przypomnieć, że przy korzystaniu z języka VBScript odwoływaliśmy się często do obiektu o nazwie *wshShell* (powłoki systemowej). Dla tych Czytelników, który nie mieli styczności z tym zagadnieniem, przedstawiam rysunek 2-7, który prezentuje schemat modelu tego obiektu.



Rysunek 2-7 Obiekt *wshShell* języka VBScript udostępnia wiele łatwych w użyciu metod i właściwości, przydatnych w administrowaniu siecią.

Tworzenie i korzystanie z obiektu *wshShell*

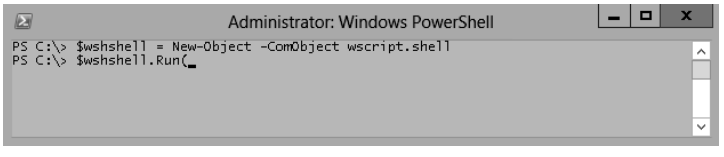
Aby utworzyć nową instancję obiektu *wshShell*, użyjemy polecenia cmdlet ***New-Object*** z argumentem ***-comobject***, dostarczając identyfikator programu *wscript.shell*. Utworzony obiekt przechowamy w zmiennej. Oto przykład:

```
$wshShell = New-Object -comobject „wscript.shell”
```

Po utworzeniu obiektu i umieszczeniu go w zmiennej można bezpośrednio korzystać z dowolnej metody udostępnianej przez ten obiekt. Demonstruje to poniższy przykład (tym razem są to dwa wiersze kodu):

```
$wshShell = New-Object -comobject „wscript.shell”
$wshShell.run(„calc.exe”)
```

W tym przykładzie najpierw tworzona jest nowa instancja obiektu *wshShell*. Następnie została wywołana metoda ***run*** tego obiektu, aby uruchomić kalkulator. Po utworzeniu obiektu i umieszczeniu go w zmiennej można użyć techniki uzupełniania tabulatorem, aby przeglądać nazwy metod udostępnianych przez obiekt. Technikę tę demonstruje rysunek 2-8.



Rysunek 2-8 Uzupełnianie tabulatorem wylicza kolejno metody udostępniane przez obiekt.

➜ Tworzenie obiektu *wshShell*

1. Otwórz konsolę Windows PowerShell, jeśli nie jest jeszcze uruchomiona.
2. Utwórz instancję obiektu *wshShell*, posługując się poleceniem cmdlet ***New-Object*** i umieść obiekt w zmiennej *$wshShell*. Wymagany kod jest przedstawiony poniżej:

   ```
   $wshShell = New-Object -comobject „wscript.shell”
   ```

3. Uruchom program Kalkulator (calc.exe), wywołując metodę ***run*** obiektu *wshShell*. Wykorzystaj uzupełnianie tabulatorem, aby uniknąć konieczności wpisywania całej nazwy metody. Rozpocznij wiersz od nazwy zmiennej, w której został umieszczony obiekt, wpisz kropkę, po czym nazwę metody. Następnie podaj nazwę programu do uruchomienia w nawiasach i cudzysłowach, jak poniżej:

   ```
   $wshShell.run(„calc.exe”)
   ```

4. Posłuż się metodą ***ExpandEnvironmentStrings***, aby wyświetlić ścieżkę do katalogu Windows. Jest ona przechowywana w zmiennej środowiskowej ***%windir%***. Funkcja uzupełniania tabulatorem będzie szczególnie przydatna w przypadku

nazwy tej metody. Zmienna środowiskowa musi być umieszczona w cudzysłowach, jak poniżej:

```
$wshShell.ExpandEnvironmentStrings("%windir%")
```

Polecenie to zwróci pełną ścieżkę do katalogu systemu Windows na danym komputerze:

```
C:\Windows
```

Korzystanie z polecenia *Show-Command*

Polecenie cmdlet *Show-Command* wyświetla graficzne narzędzie wyszukiwania poleceń, pozwalające wybrać odpowiednie polecenie z listy. Na pierwszy rzut oka *Show-Command* może się wydać graficznym odpowiednikiem polecenia *Get-Command*, ale w rzeczywistości robi znacznie więcej. Pierwszą wskazówką jest fakt, że blokuje ono konsolę Windows PowerShell – inaczej mówiąc, nie można powrócić do wiersza poleceń PowerShell, dopóki albo nie wybierze się polecenia z listy, albo nie anuluje tej operacji.

Wywołanie polecenia *Show-Command* bez parametrów powoduje wyświetlenie okna o wysokości 600 pikseli i szerokości 300 pikseli. Wielkość okna można kontrolować parametrami *-Height* (wysokość) oraz *-Width* (szerokość). Poniższy przykład tworzy okno wysokie na 500 i szerokie na 350 pikseli:

```
Show-Command -Height 500 -Width 350
```

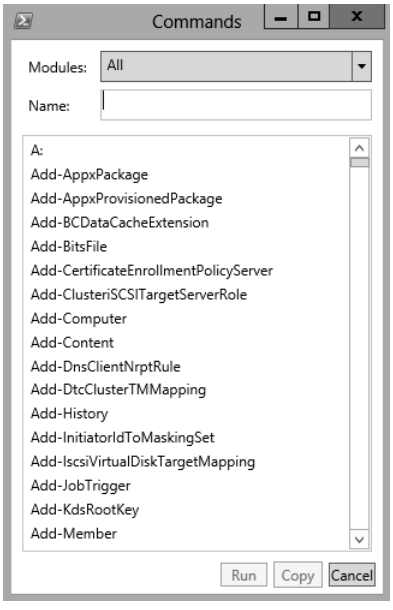
Okno utworzone tym poleceniem zostało pokazane na rysunku 2-9.

Możliwe jest również ograniczenie zawartości okna do pojedynczego polecenia cmdlet, podając je jako ostatni argument polecenia *Show-Command*, co demonstruje poniższy przykład:

```
Show-Command -Height 500 -Name Get-Process
```

W oknie kontrolki wyświetlanym w tej sytuacji dostępne są wszystkie przełączane parametry jako pola wyboru oraz pola tekstowe dla wartości innych parametrów. Technika ta jest pokazana na rysunku 2-10.

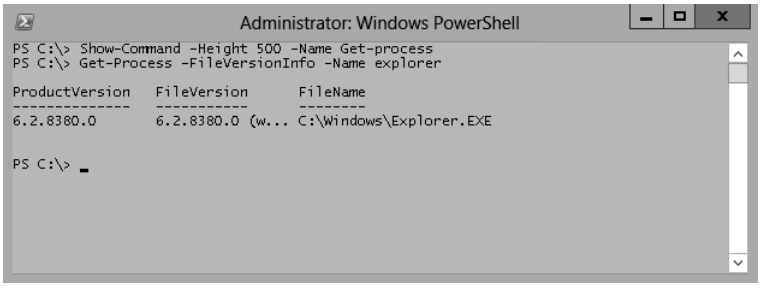
Po wybraniu wszystkich argumentów i podaniu wartości można albo skopiować całe polecenie do schowka, albo bezpośrednio je uruchomić. W tym drugim przypadku w oknie Windows PowerShell pojawi się zarówno utworzone polecenie, jak i jego rezultaty, co prezentuje rysunek 2-11.



Rysunek 2-9 Kontrolka wyszukiwania poleceń domyślnie zawiera wszystkie polecenia ze wszystkich modułów.



Rysunek 2-10 Pola wyboru umożliwiają dołączenie do polecenia cmdlet argumentów przełączanych, zaś pola tekstowe umożliwiają wpisanie wartości dla innych parametrów.



Rysunek 2-11 Utworzone polecenie oraz wynik jego wykonania pojawia się w konsoli Windows PowerShell po kliknięciu Run w oknie kontrolki ***Show-Command***.

Jak nazewnictwo poleceń cmdlet pomaga w nauce

Jedną z najlepszych rzeczy, które można zauważyć na temat Windows PowerShell, jest konwencja nazewnicza *czasownik-rzeczownik*. Podobnie jak w „żywym" języku, czasownik wskazuje działanie, które ma zostać wykonane, na przykład ***set*** przy dokonywaniu zmian lub ***get*** przy odczytywaniu wartości. Rzeczownik wskazuje element bądź elementy, których to działanie ma dotyczyć, na przykład proces lub usługę. W miarę nabierania wprawy w posługiwaniu się poleceniami cmdlet i poznawaniem konwencji czasownik-rzeczownik, użytkownik zaczyna się domyślać, jak może nazywać się pożądane polecenie (o ile oczywiście istnieje). Na przykład, jeśli potrzebujemy zdobyć informacje na temat procesu uruchomionego w komputerze, to wiedząc, że za pobieranie informacji odpowiada czasownik ***get***, można postawić hipotezę, że poszukiwane polecenie ma postać ***Get-Process***. Następnie po prostu można spróbować użyć tego polecenia i po raz kolejny potwierdzić postawioną tezę[1].



Uwaga Gdy próbujemy odgadnąć nazwy poleceń cmdlet, zawsze należy najpierw wypróbować formę liczby pojedynczej, a nie mnogiej (tzn. bez końcowego *s*). Nie jest to wymóg projektowy, ale silnie akcentowana preferencja. Na przykład polecenia zwracające informacje o usługach i procesach nazywają się odpowiednio ***Get-Service*** i ***Get-Process***, a nie ***Get-Services*** i ***Get-Processes***.

1 Przedstawiona przez autora prostota jest faktem, ale z jednym zastrzeżeniem – dotyczy osób, które mają przynajmniej podstawową znajomość języka angielskiego. W trakcie całej książki staram się podawać wyjaśnienia angielskich słów pełniących funkcję poleceń, argumentów czy parametrów, można jednak zauważyć, że opanowanie języka PowerShell bez rozumienia przynajmniej podstawowych składników będzie raczej trudne. Na początek jednak trzeba zapamiętać dwa podstawowe słowa: *verb* – czasownik oraz *noun* – rzeczownik.

W celu uzyskania listy wszystkich akceptowalnych czasowników (ang. *verb*), czyli tych, które są (i mogą być) używane w poleceniach cmdlet, należy posłużyć się poleceniem ***Get-Verb***.

Obecnie – w wersji Windows PowerShell 3.0 – istnieje 98 zaakceptowanych czasowników, co oznacza, że liczba czasowników wzrosła tylko o dwa w stosunku do wersji 2.0. Nowe czasowniki to ***use*** (użyj) oraz ***unprotect*** (usuń ochronę). Można to sprawdzić poleceniem pokazanym poniżej, gdzie cmdlet ***Measure-Object*** (zmierz obiekt) zwraca liczbę różnych czasowników:

```
PS C:\> (Get-Verb | Measure-Object).count
98
```

W istocie nie musimy w tym konkretnym przypadku sięgać po polecenie ***Measure-Object***, gdyż polecenie ***Get-Verb*** zwraca obiekt typu tablica (*array*), zaś taki obiekt zawsze zawiera właściwości ***count***. Tak więc można posłużyć się prostszą wersją polecenia:

```
PS C:\> (Get-Verb).count
98
```

Grupowanie czasowników Windows PowerShell

Opanowanie blisko 100 różnych czasowników mogłoby być trudne, jednak zespół projektowy Windows PowerShell pogrupował je w celu ułatwienia nauki. Analiza grupy powszechnych (*common*) czasowników ujawnia wzór. Poniższy przykład wyświetla czasowniki tej grupy:

```
PS C:\> Get-Verb | where group -match 'common' | Format-Wide verb -auto
Add       Clear     Close     Copy      Enter     Exit      Find      Format    Get
Hide      Join      Lock      Move      New       Open      Optimize  Pop       Push
Redo      Remove    Rename    Reset     Resize    Search    Select    Set       Show
Skip      Split     Step      Switch    Undo      Unlock    Watch
```

Gdy przyjrzymy się tym wyrazom, zauważymy prosty wzorzec „sparowanych" czasowników: ***Add/Remove*** (dodaj/usuń), ***Enter/Exit*** (wprowadź/wyjdź), ***Get/Set*** (pobierz/ustaw), ***Select/Skip*** (zaznacz/pomiń) i tak dalej[2]. Dzięki opanowaniu wzorca przeciwstawnych czasowników, można szybko wciągnąć się w konwencję nazewniczą Windows PowerShell. Wprawdzie nie każdy czasownik ma przeciwstawnego partnera, jednak par takich jest dostatecznie wiele, aby warto było je wyszukać i zapamiętać.

Dzięki grupowaniu czasowników można określić, w jaką stronę powinniśmy kierować swoje wysiłki. Zespół projektowy PowerShell podzielił czasowniki na siedem grup w oparciu o typowe zadania IT, takie jak przetwarzanie danych lub wykonywanie diagnostyki. Poniższe polecenie zwraca nazwy grup czasowników Windows PowerShell:

2 Listę wszystkich czasowników Windows PowerShell 3.0 z ich tłumaczeniem na język polski zawiera Dodatek H.

```
PS C:\Users\marekw> Get-Verb | select group -Unique

Group
-----
Common                    //powszechne
Data                      //dane
Lifecycle                 //cykl życiowy
Diagnostic                //diagnostyka
Communications            //łączność
Security                  //zabezpieczenia
Other                     //inne
```

Dystrybucja czasowników Windows PowerShell

Inną metodą uzyskania lepszej orientacji wśród poleceń cmdlet jest przeanalizowanie dystrybucji czasowników. Choć istnieje blisko sto zaakceptowanych czasowników (a także pewna liczba takich, które nie uzyskały oficjalnej akceptacji, ale są stosowane przez niezależnych programistów), zazwyczaj korzysta się zaledwie z ich ułamka w standardowej instalacji, zaś po niektóre nigdy nie będzie trzeba sięgnąć. Jeśli posłużymy się poleceniem ***Group-Object*** (dla którego aliasem jest ***group***) w połączeniu z ***Sort-Object*** (alias ***sort***), ta nierównomierność rozkładu stanie się oczywista.

```
Get-Command -CommandType cmdlet | group verb | sort count -Descending
```

Rysunek 2-12 prezentuje wynik powyższego polecenia:



Rysunek 2-12 Wykorzystanie polecenia ***Get-Command*** do zaprezentowania dystrybucji czasowników Windows PowerShell.



Uwaga Dokładna liczba poszczególnych poleceń cmdlet w każdej grupie i dystrybucja czasowników i rzeczowników zależy od wersji systemu operacyjnego, a także od funkcji i ról włączonych w konkretnej instalacji. Dodatkowo zainstalowanie określonych programów i aplikacji może dołączać dodatkowe polecenia cmdlet. Z tego względu wartości uzyskiwane na komputerze Czytelnika będą się najprawdopodobniej różniły od prezentowanych w tym rozdziale. Nie jest to sygnał żadnego problemu, ale zjawisko normalne i oczekiwane.

Wynik widoczny na rysunku 2-12 pokazuje jasno, że większość poleceń używa zaledwie niewielkiej liczby czasowników. Na przykład spośród 436 poleceń cmdlet dostępnych na moim komputerze 278 z nich używa jednego z pierwszych 10 czasowników. Można to sprawdzić w następujący sposób:

```
PS C:\> (Get-Command -CommandType cmdlet | measure).count
436

PS C:\> $count = 0 ; Get-Command -CommandType cmdlet | group verb | sort count
 -Descending | select -First 10 | % { $count += $_.count ; $count }
94
142
180
210
225
236
247
258
268
278
```

W praktyce oznacza to, że opanować trzeba tylko 10 najczęściej stosowanych czasowników, aby móc posługiwać się więcej niż połową poleceń cmdlet, które są zawarte w Windows PowerShell.

Tworzenie profilu Windows PowerShell

W miarę tworzenia różnych aliasów i funkcji możemy stwierdzić, że pewne sekwencje klawiszy „pasują” nam bardziej od innych. W takiej sytuacji chcielibyśmy, aby nasze definicje były zawsze dostępne, bez konieczności ich tworzenia przy każdym uruchomieniu Windows PowerShell.



Uwaga Zalecam przejrzenie listy wszystkich aliasów istniejących już w Windows PowerShell, zanim zaczniemy tworzyć nowe. Inaczej można łatwo utworzyć duplikujące się konfiguracje, różniące się tylko nieznacznymi szczegółami, co może znacznie utrudnić pracę w przyszłości.

Naturalnie, można utworzyć swój własny skrypt, który wykona odpowiednią konfigurację, o ile oczywiście będziemy pamiętać o jego uruchomieniu. Jednak to, do czego powinniśmy dążyć, to stosowanie bardziej standaryzowanych metod pracy. Rozwiązaniem jest utworzenie niestandardowego profilu, który będzie przechowywał nasze ustawienia. Tym, co sprawia, że utworzenie profilu Windows PowerShell jest naprawdę użyteczne, jest fakt, że profil taki jest automatycznie ładowany podczas uruchamiania powłoki.



Uwaga Profil Windows PowerShell w istocie jest skryptem, który jest wykonywany za każdym razem, gdy uruchamiamy powłokę. Obsługa skryptów w Windows PowerShell nie jest włączona domyślnie. W środowisku sieciowym zasady wykonywania skryptów PowerShell mogą być narzucone przez administratora sieci za pośrednictwem zasad grupy. W środowisku grupy roboczej lub na domowym komputerze zasady grupy nie są stosowane w tym celu. Więcej informacji na temat włączania wykonywania skryptów zawiera rozdział 5, „Używanie skryptów PowerShell".

Tworzenie profilu Windows PowerShell demonstruje poniższa procedura.

➜ Tworzenie osobistego profilu Windows PowerShell

1. Uruchom konsolę Windows PowerShell, jeśli jeszcze nie jest otwarta, po czym sprawdź bieżące zasady wykonywania skryptów:

   ```
   Get-ExecutionPolicy
   ```

2. Jeśli w wyniku uzyskasz odpowiedź ***Restricted*** (Ograniczone), zmień to ustawienie na ***remotesigned*** (zdalnie podpisane), ale tylko dla bieżącego użytkownika:

   ```
   Set-ExecutionPolicy -Scope currentuser -ExecutionPolicy remotesigned
   ```

3. Przejrzyj wyświetlone informacje o zasadach wykonywania skryptów i wpisz **Y**, aby zgodzić się na wykonanie zmian.
4. Sprawdź, czy istnieje już jakiś profil, używając poniższego polecenia (domyślnie profil Windows PowerShell nie istnieje):

   ```
   Test-Path $profile
   ```

5. Jeśli polecenie zwróci ***false*** (fałsz), oznacza to, że profil nie istnieje. W takiej sytuacji utwórz nowy profil, posługując się poniższym poleceniem:

   ```
   New-Item -path $profile -itemtype file -force
   ```

6. Otwórz plik profilu w narzędziu Windows PowerShell ISE, używając następującego polecenia:

   ```
   ise $profile
   ```

7. Utwórz w tym profilu alias *gh* dla polecenia cmdlet *Get-Help*. Wymaga to użycia następującego polecenia:

```
Set-Alias gh Get-Help
```

8. Utwórz funkcję edytowania twojego profilu Windows PowerShell. Odpowiedni fragment kodu jest pokazany poniżej:

```
Function Set-Profile
{
  Ise $profile
}
```

9. Wywołaj funkcjonalność *Transcript* Windows PowerShell w profilu. W tym celu dodaj polecenie *Start-Transcript* do kodu skryptu, jak poniżej (polecenie *Start-Transcript* tworzy rejestr wszystkich wywoływanych poleceń Windows PowerShell, a także wyjścia tych poleceń):

```
Start-Transcript
```

10. Zapisz modyfikacje pliku profilu, klikając ikonę Save (Zapisz) w pasku narzędzi lub wybierając polecenie Save z menu File.
11. Zamknij okno Windows PowerShell ISE, a następnie konsolę Windows PowerShell.
12. Uruchom konsolę Windows PowerShell. Powinieneś zauważyć komunikat o uruchomieniu funkcjonalności *Transcript*.
13. Przetestuj działanie aliasu *gh*.
14. Otwórz profil w Windows PowerShell ISE, używając nowo utworzonej funkcji *Set-Profile*.
15. Przejrzyj zawartość profilu i zamknij narzędzie Windows PowerShell ISE.

Na tym kończy się ćwiczenie z tworzenia profilu Windows PowerShell.

Wyszukiwanie wszystkich aliasów dla określonego obiektu

Jeśli znamy nazwę obiektu i chcemy uzyskać wszystkie aliasy zdefiniowane dla tego obiektu, można wykorzystać polecenie *Get-Alias* w celu uzyskania listy tych aliasów. Aby odfiltrować tylko aliasy dla naszego poszukiwanego obiektu, przekierowujemy wyjście do polecenia *Wher-Object* i podajemy wartość właściwości *definition*. Poniższy przykład demonstruje tę technikę dla polecenia cmdlet *Get-ChildItem*:

```
gal | where definition -match „Get-ChildItem”
```

Stosowanie poleceń cmdlet: ćwiczenia krok po kroku

W poniższym ćwiczeniu zbadamy działanie poleceń *Get-ChildItem* oraz *Get-Member* w Windows PowerShell. Zobaczymy, że można łatwo wykorzystać te polecenia do automatyzacji rutynowych zadań administracyjnych. Będziemy też kontynuować eksperymetowanie z funkcjonalnością potoków w Windows PowerShell.

➜ Korzystanie z poleceń *Get-ChildItem* i *Get-Member*

1. Otwórz konsolę Windows PowerShell, jeśli nie jest jeszcze uruchomiona.
2. Użyj polecenia cmdlet *Get-Alias*, aby uzyskać listę wszystkich aliasów zdefiniowanych w systemie. Przekieruj wyjście tego polecenia do polecenia *Where-Object* (alias *where*), specyfikując argument *-match* (pasuje do) dla właściwości *definition*, aby wybrać tylko aliasy dla polecenia *Get-ChildItem*. Realizuje to poniższy kod:

   ```
   gal | where definition -match „Get-ChildItem"
   ```

   W rezultacie uzyskamy trzy aliasy zdefiniowane dla polecenia *Get-ChildItem*:

   ```
   CommandType     Name                                               ModuleName
   -----------     ----                                               ----------
   Alias           dir -> Get-ChildItem
   Alias           gci -> Get-ChildItem
   Alias           ls -> Get-ChildItem
   ```

3. Używając aliasu *gci* uzyskaj listing plików i folderów zawartych w bieżącym katalogu, wpisując go po znaku zachęty.
4. Ogranicz wyświetlaną listę do dużych plików, kierując wyjście do polecenia *Where-Object*, podając argument *-gt* (*greater then* – większe niż) z wartością 1000 dla właściwości *length*. Pokazuje to poniższy kod:

   ```
   gal | where length -gt 1000
   ```

5. Wyczyść ekran, używając aliasu *cls*.
6. Użyj polecenia *Get-Alias*, aby ustalić, jakie polecenie cmdlet jest wywoływane przez alias *cls*. Użyj aliasu *gal*, by nie wypisywać całej nazwy polecenia, jak poniżej:

   ```
   gal cls
   ```

7. Użyj polecenia *Get-Alias*, aby znaleźć polecenie cmdlet, do którego odwołuje się alias *mred*:

   ```
   gal mred
   ```

8. Najprawdopodobniej alias *mred* nie jest jeszcze zdefiniowany w komputerze. W takim przypadku zobaczymy następujący komunikat błędu: