helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

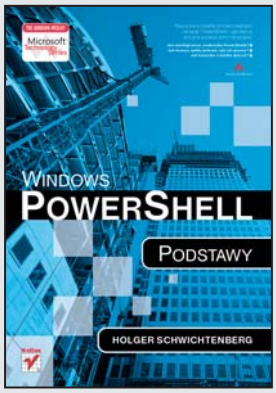
Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

Windows PowerShell. Podstawy



Autor: Holger Schwichtenberg
Tłumaczenie: Paweł Koronkiewicz
ISBN: 978-83-246-2088-3
Format: 168x237, stron: 488

Naucz się korzystać z nowoczesnych narzędzi PowerShella i usprawniaj żmudne zadania administracyjne!

- Jak skonfigurować środowisko PowerShella?
- Jak tworzyć aplety poleceń i jak ich używać?
- Jak korzystać z potoku danych?

Windows PowerShell to innowacyjne narzędzie do zarządzania systemem i przetwarzania skryptów dla serwerów oraz klientów pod kontrolą Windows. Charakterystyczną cechą tego interpretera poleceń jest obiektowa logika, która sprawia, że wynikiem każdego polecenia jest obiekt określonego typu. Dodatkowo wszystkie funkcje dostępne przez API systemu są możliwe do wywołania z poziomu PowerShella. Dzięki swojej uniwersalności oraz nowoczesnym narzędziom interpreter ten zapewnia spójny oraz jasny mechanizm zarządzania, oparty na wierszu poleceń i niewymagający od użytkownika dużej wiedzy programistycznej.

Książka „Windows PowerShell. Podstawy" zawiera dokładny opis innowacyjnej architektury PowerShella, zasad pisania skryptów oraz możliwości korzystania z podstawowych interfejsów systemowych: COM, WMI, ADSI i ADO.NET, a także obszerny zbiór opisów rozwiązań konkretnych zadań administracyjnych. Korzystając z tego przewodnika, poznasz funkcje i narzędzia PowerShella. Nauczysz się również korzystać z obiektowego potoku danych i zarządzać procesami, dziennikami zdarzeń oraz wpisami rejestru. Dzięki zawartej tu wiedzy z pewnością usprawnisz i zautomatyzujesz wszelkie czasochłonne zadania administracyjne, by móc swobodnie stosować PowerShell jako podstawową konsolę do codziennej pracy.

- Aplety poleceń
- Przetwarzanie potokowe
- Potoki danych
- Uniwersalny model nawigacji
- Definiowanie dysków
- Język skryptów PowerShella
- Tablice asocjacyjne
- Praca ze skryptami
- Różnicowanie reakcji na błędy
- Biblioteki klas
- Narzędzia PowerShella
- Zarządzanie użytkownikami i grupami w Active Directory
- Bazy danych
- Zaawansowane operacje dostępu do danych

PowerShell skróci czas Twojej pracy!

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2008

SPIS TREŚCI

Wstęp **11**

Podziękowania **15**

O autorze **17**

CZĘŚĆ I: WINDOWS POWERSHELL — WPROWADZENIE **19**

Rozdział 1. Pierwsze kroki **21**

Czym jest Windows PowerShell? 21
Pobieranie i instalowanie rozszerzeń PSCX 35
Testowanie rozszerzeń PSCX 37
Pobieranie i instalowanie pakietu PowerShellPlus 37
PowerShell Editor 38
Podsumowanie 42

Rozdział 2. Aplety poleceń **43**

Aplety poleceń — podstawy 43
Aliasy 47
Wyrażenia 51
Polecenia zewnętrzne 51
System pomocy WPS 53
Podsumowanie 56

Rozdział 3. Przetwarzanie potokowe **61**

Potok danych 61
Procesor potoków 65
Złożone potoki danych 66
Dane wyjściowe 67
Interakcje z użytkownikiem 73
Podsumowanie 75

Rozdział 4. Potoki danych 77

Analizowanie zawartości potoku 77
Filtrowanie obiektów 88
Przycinanie obiektów 90
Sortowanie obiektów 92
Grupowanie obiektów 93
Obliczenia 93
Wartości pośrednie 95
Porównywanie obiektów 96
Rozgałęzienia 96
Podsumowanie 97

Rozdział 5. Uniwersalny model nawigacji 99

Rejestr systemu Windows 99
Dostawcy danych i dyski 100
Aplety nawigacyjne 101
Ścieżki 103
Definiowanie dysków 104
Podsumowanie 105

Rozdział 6. Język skryptów PowerShella 107

System pomocy 108
Rozdzielanie poleceń 108
Komentarze 108
Zmienne 109
Typy danych PowerShella 110
Liczby 114
Liczby losowe 115
Ciągi znakowe 116
Data i godzina 119
Tablice 121
Tablice asocjacyjne (tabele skrótów) 123
Operatory 125
Struktury sterujące 126
Podsumowanie 128

Rozdział 7. Praca ze skryptami 131

Pierwszy skrypt 131
Uruchamianie skryptu PowerShella 133
Włączanie skryptu 133
Zabezpieczenia 134
Podpisywanie skryptów 136

Wstrzymywanie wykonywania skryptu 138
Błędy i obsługa błędów 138
Podsumowanie 144

Rozdział 8. Biblioteki klas 145

Klasy .NET 145
Klasy COM 150
Klasy WMI 152
Podsumowanie 166

Rozdział 9. Narzędzia PowerShella 169

Konsola Windows PowerShell 169
PowerTab 174
PowerShellPlus 174
PowerShell Analyzer 178
PrimalScript 181
Sapien PowerShell Help 184
Podsumowanie 186

Rozdział 10. Porady i rozwiązywanie problemów 187

Debugowanie i śledzenie 187
Dodatkowe aplety poleceń 191
Historia poleceń 203
Informacje o systemie i komputerze 203
Profile PowerShella 205
Graficzny interfejs użytkownika 211
Podsumowanie 215

Część II: Windows PowerShell w praktyce 217

Rozdział 11. Systemy plików 219

Aplety poleceń do administracji systemem plików 219
Dyski 220
Zawartość katalogu 224
Właściwości pliku 226
Właściwości plików wykonywalnych 228
Łącza w systemie plików 229
Kompresja 233
Udziały sieciowe 234
Podsumowanie 246

Rozdział 12. Dokumenty 247

Pliki tekstowe 247
Pliki binarne 250
Pliki CSV 251
Pliki XML 252
Pliki HTML 261
Podsumowanie 264

Rozdział 13. Rejestr i oprogramowanie 265

Rejestr systemu Windows 265
Administracja oprogramowaniem 271
Podsumowanie 278

Rozdział 14. Procesy i usługi 279

Procesy 279
Usługi systemu Windows 284
Podsumowanie 291

Rozdział 15. Komputery i urządzenia 293

Konfiguracja komputera 293
Urządzenia 296
Dzienniki zdarzeń 300
Liczniki wydajności 302
Podsumowanie 304

Rozdział 16. Sieć 305

Ping 305
Konfiguracja sieci 306
Odwzorowania nazw 309
Pobieranie plików z serwera HTTP 310
Poczta elektroniczna 311
Microsoft Exchange Server 2007 312
Internetowe usługi informacyjne 315
Podsumowanie 321

Rozdział 17. Usługi katalogowe 323

Dostęp do usług katalogowych — wprowadzenie 323
Zarządzanie użytkownikami i grupami przy użyciu WMI 324
Obszar nazw System.DirectoryServices i adapter ADSI 325

Wady adaptera ADSI 330
Identyfikacja obiektów usług katalogowych (ścieżki) 333
Typowe operacje katalogowe 335
Podsumowanie 342

Rozdział 18. Zarządzanie użytkownikami i grupami w Active Directory 343

Klasa Active Directory User 343
Tworzenie konta użytkownika 346
Uwierzytelnianie 349
Usuwanie konta użytkownika 349
Zmiana nazwy konta użytkownika 350
Przenoszenie konta użytkownika 350
Zarządzanie grupami 351
Jednostki organizacyjne 354
Podsumowanie 354

Rozdział 19. Przeszukiwanie Active Directory 355

Składnia zapytań LDAP 355
Zapytania LDAP w PowerShellu 356
Wyszukiwanie w katalogu — uwagi praktyczne 360
Przykłady zapytań LDAP 363
Używanie apletu polecenia Get-ADObject 364
Podsumowanie 364

Rozdział 20. Dodatkowe biblioteki rozszerzeń do administracji Active Directory 367

Dostawca nawigacji PowerShell Community Extensions 367
Rozszerzenia www.IT-Visions.de 368
Rozszerzenia firmy Quest 370
Pobieranie informacji o strukturze Active Directory 371
Zasady grup 372
Podsumowanie 377

Rozdział 21. Bazy danych 379

Biblioteka ADO.NET 379
Przykładowa baza danych 384
Bazy danych w PowerShellu 384
Podsumowanie 394

Rozdział 22. Zaawansowane operacje dostępu do danych 395

Obiekt DataSet 395
Dostęp do danych z użyciem apletów www.IT-Visions.de 401
Podsumowanie 406

Rozdział 23. Zabezpieczenia 407

Zabezpieczenia Windows 407
Klasy 412
Odczyt list ACL 414
Odczyt wpisów ACE 416
Podsumowanie 417

Rozdział 24. Zmiana ustawień zabezpieczeń 419

Translacja identyfikatorów kont 419
Odczytywanie informacji o właścicielu 424
Dodawanie wpisu ACE do listy ACL 424
Usuwanie z listy ACL wpisu ACE 427
Kopiowanie ACL 430
Definiowanie ACL w języku SDDL 430
Podsumowanie 432

CZĘŚĆ III: DODATKI 433

Dodatek A Aplety poleceń WPS 435

Dodatek B PowerShell 2.0 453

Dodatek C Bibliografia 457

Skorowidz 461

ROZDZIAŁ 6

JĘZYK SKRYPTÓW POWERSHELLA

W tym rozdziale:

System pomocy 108
Rozdzielanie poleceń 108
Komentarze 108
Zmienne 109
Typy danych PowerShella 110
Liczby 114
Liczby losowe 115
Ciągi znakowe 116
Data i godzina 119
Tablice 121
Tablice asocjacyjne (tabele skrótów) 123
Operatory 125
Struktury sterujące 126

Windows PowerShell (WPS) to nie tylko infrastruktura apletów poleceń, ale cały język skryptowy umożliwiający budowanie sekwencji poleceń o charakterze tradycyjnych programów. Język skryptów PowerShella (PSL, ang. *PowerShell Scripting Language*) to język ze zmiennymi, pętlami, warunkami, funkcjami i mechanizmem obsługi błędów.

Firma Microsoft nie zastosowała w nowym narzędziu żadnego istniejącego już języka, ale — jak twierdzi — „czerpała inspirację" z języków powłok UNIX oraz języków PERL, PHP, Python i C#. W języku PowerShella znajdziemy więc nawiasy klamrowe i znaki $. Brak natomiast wymogu kończenia poleceń średnikiem.

System pomocy

Opisy elementów składniowych języka skryptów PowerShella można wyświetlić podobnie jak opisy apletów poleceń. Są to proste dokumenty tekstowe, instalowane razem z narzędziem. Nazwy dokumentów opisujących konstrukcje języka rozpoczynają się od słowa „About". Przykładowo, polecenie:

```
Get-Help About_for
```

wyświetla informacje o pętlach `for`.

Polecenie

```
Get-Help About
```

wyświetla listę wszystkich dokumentów „About".

Rozdzielanie poleceń

Ogólnie, każdy wiersz skryptu PowerShella to jedno polecenie. Polecenie może składać się z wielu nazw apletów poleceń połączonych znakiem potoku (`|`). W jednym wierszu można umieścić więcej niż jedno polecenie, rozdzielając je znakiem średnika (`;`). Znak średnika można umieszczać na końcu każdego wiersza, ale nie jest to wymagane.

Jeżeli jedno polecenie musi zostać zapisane w kilku wierszach, należy użyć odwróconego apostrofu (`` ` ``). Umieszczenie takiego znaku na końcu wiersza powoduje, że następny wiersz jest traktowany jako dalszy ciąg tego samego polecenia:

```
gps | `
format-list
```

Komentarze

Komentarze oznacza się symbolem `#`:

```
# Komentarz
```

Zmienne

Nazwy zmiennych rozpoczyna znak `$`. Nazwa może zawierać litery i cyfry oraz znak _. Część nazw jest zarezerwowana dla zmiennych predefiniowanych, takich jak `$_`.

Typ zmiennej

Określanie typu zmiennej nie jest wymagane:

```
$a = 5
```

ale może on zostać zadeklarowany. Można stosować typy danych WPS („szybkie typy", ang. *type accelerator*) oraz klasy .NET:

```
$a = [int] 5
$a = [System.DateTime] "1972-8-1"
```

Typy WPS odpowiadają klasom .NET: typy `[int]`, `[System.Int32]` i `[int32]` są identyczne. `[int]` to znacznik typu WPS oznaczający 32-bitową liczbę całkowitą. Wewnętrznie reprezentuje go klasa .NET `[System.Int32]`. Dodatkowo, dostępny jest skrót `[int32]`.

WSKAZÓWKA: Umieszczenie nazwy typu przed wyrażeniem przypisania (na przykład `[int] $a = 5`) trwale przypisuje typ zmiennej. Odpowiada to zasadom deklaracji typów w językach takich jak C++, Java czy C#.

W chwili przypisania wartości następuje niejawne zadeklarowanie zmiennej. Deklaracja ta ma określony zakres (skrypt, procedura lub blok kodu). Deklarację można usunąć apletem `Remove-Variable`.

Jeżeli typy zmiennych nie są deklarowane, należy liczyć się z możliwością wystąpienia błędów wynikających z błędnego określenia typu. Polecenie `Set-PSDebug –Strict` zapewni generowanie błędów przy próbach użycia zmiennych, których wartość nie została wcześniej przypisana.

W poniższym przykładzie, WPS zgłasza błąd przy próbie wykonania ostatniego polecenia, ponieważ niejawna deklaracja zmiennej `$y` ma zakres ograniczony do bloku wyróżnionego nawiasami klamrowymi:

```
Set-PSDebug –Strict
$x = 5
{
$y = 5
$x
}
$y
```

Typy danych PowerShella

W tabeli 6.1 przedstawione jest zestawienie wszystkich „szybkich typów" (ang. *type accelerators*) PowerShella. Wiele z nich opisanych będzie w dalszych rozdziałach (na przykład `[WMI]` i `[ADSI]`).

Sprawdzanie typu

Informacje o typie są dostępne niezależnie od tego, czy typ zmiennej został jawnie określony. Jeżeli typ nie został określony to zmienna ma typ ostatniej przypisanej wartości.

Metoda `GetType()` zwraca typ zmiennej w postaci obiektu .NET `System.Type`. Ponieważ każda zmienna WPS to obiekt klasy .NET, metoda ta jest zawsze dostępna (jest to metoda klasy `System.Object`, po której dziedziczą wszystkie klasy .NET). W większości przypadków potrzebna jest sama nazwa klasy — można ją pobrać z właściwości `Full` ➥`name` (pełna nazwa, z obszarem nazw) lub `Name` (bez obszaru nazw):

```
$b = [System.DateTime] "1972-8-1"
"Typ zmiennej $b to: " + $b.GetType().Fullname
```

Zmienne predefiniowane

Zestaw zmiennych PowerShella jest bardzo rozbudowany. Ich właściwa nazwa to **zmienne zintegrowane** (ang. *integrated variables*) lub **zmienne wewnętrzne** (ang. *internal variables*). W tabeli 6.2 przedstawione są tylko wybrane z nich.

Przykład

Oto przykład użycia zmiennej `$OFS`:

Tabela 6.1. „Szybkie typy" PowerShella

[int]	typeof(int)
[int[]]	typeof(int[])
[long]	typeof(long)
[long[]]	typeof(long[])
[string]	typeof(string)
[string[]]	typeof(string[])
[char]	typeof(char)
[char[]]	typeof(char[])
[bool]	typeof(bool)
[bool[]]	typeof(bool[])
[byte]	typeof(byte)
[double]	typeof(double)
[decimal]	typeof(decimal)
[float]	typeof(float)
[single]	typeof(float)
[regex]	typeof(System.Text.RegularExpressions.Regex)
[array]	typeof(System.Array)
[xml]	typeof(System.Xml.XmlDocument)
[scriptblock]	typeof(System.Management.Automation.ScriptBlock)
[switch]	typeof(System.Management.Automation.SwitchParameter)
[hashtable]	typeof(System.Collections.Hashtable)
[type]	typeof(System.Type)
[ref]	typeof(System.Management.Autmation.PSReference)
[psobject]	typeof(System.Management.Automation.PSObject)
[wmi]	typeof(System.Management.ManagementObject)
[wmisearcher]	typeof(System.Management.ManagementObjectSearcher)
[wmiclass]	typeof(System.Management.ManagementClass)

Polecenie:

```
$OFS="/" ; [string] ("a","b","c")
```

wyświetla:

```
a/b/c
```

Tabela 6.2. Wybrane zmienne wewnętrzne WPS

Zmienna	Funkcja
`$true`	wartość „prawda”
`$false`	wartość „fałsz”
`$OFS`	separator stosowany przy wyświetlaniu zbioru obiektów
`$Home`	katalog macierzysty użytkownika
`$PSHome`	katalog instalacyjny PowerShella
`$Args`	parametr (do użytku w funkcjach)
`$Input`	zawartość potoku (do użytku w funkcjach)
`$_`	bieżący obiekt potoku (do użytku w pętlach)
`$StackTrace`	bieżąca sekwencja wywołań
`$Host`	informacje o „hoście” WPS
`$LastExitCode`	wartość zwrócona przez ostatni wykonany program konsoli lub zewnętrzną aplikację Windows
`$Error`	pełna lista błędów od uruchomienia WPS; ograniczenie liczby błędów można zmienić przy użyciu zmiennej `$MaximumErrorCount`

Wskazówka: Wszystkie zadeklarowane zmienne, wewnętrzne i użytkownika, można wyświetlić poleceniem `Get-ChildItem Variable` (alias `Dir Variable:`).

Wywołanie `Dir Variable:p*` zwraca listę wszystkich zmiennych, których nazwa zaczyna się literą „P” (wielką lub małą). Tak samo działa wywołanie `Get-↪Variable p*`.

Stałe

Niektórych zmiennych wewnętrznych nie można zmieniać. Można też zablokować możliwość modyfikowania dowolnej zmiennej:

```
Set-Variable nazwa_zmiennej –Option readonly
```

OSTRZEŻENIE: W tym wywołaniu nazwa zmiennej nie zawiera znaku `$`!

Podstawianie zmiennych

Zasada podstawiania zmiennych jest stosowana nie tylko w typowych wyrażeniach, ale również wewnątrz ciągów znakowych. Jeżeli:

```
[int] $licznik = 1
[sting] $Komputer = "E01"
```

to zamiast:

```
$licznik.ToString() + ". Dostęp do komputera " + $Komputer
```

można napisać:

```
"$licznik. Dostęp do komputera $Komputer"
```

W obu przypadkach wartość ciągu jest ta sama:

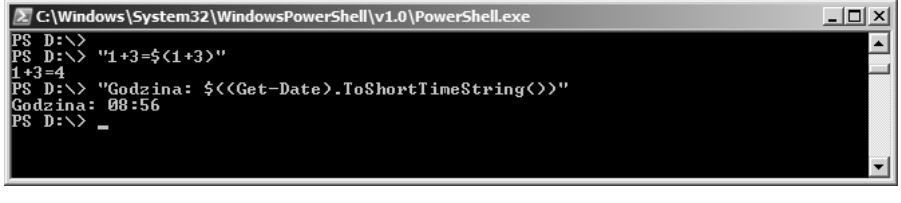
```
"1. Dostęp do komputera E01"
```

Z podstawiania zmiennych można korzystać w wywołaniach apletów poleceń. Poniższe dwa polecenia są równoważne — oba pobierają dane ze ścieżki *WinNT://E01*:

```
GetDirectoryEntry ("WinNT://" + $Komputer)
GetDirectoryEntry "WinNT://$Komputer"
```

Mechanizm podstawiania zmiennych obejmuje również obliczanie wartości wyrażeń. Po znaku dolara można umieścić w ciągu znakowym dowolne wyrażenie, którego wartość zostanie włączona do ciągu wynikowego (patrz rysunek 6.1). Łatwo się o tym przekonać, wpisując na przykład:

```
"1+3=$(1+3)"
"Godzina: $((Get-Date).ToShortTimeString())"
```



Rysunek 6.1. Przykład włączania wyrażeń do ciągu znakowego

Ostrzeżenie: Podstawianie zmiennych nie zostanie przeprowadzone, jeżeli ciąg zostanie zapisany przy użyciu znaków apostrofu zamiast cudzysłowu:

```
'$count. Dostęp do komputera $Komputer'
```

Liczby

Konstruktorami wartości liczbowych mogą być literały liczby, wyrażenia i zakresy (patrz rysunek 6.2). Liczby szesnastkowe poprzedza się znakami `0X` (na przykład `0Xff = 255`) i można je stosować wszędzie tam, gdzie liczby dziesiętne (na przykład `0xff+1 = 256`).



Rysunek 6.2. Liczby w PowerShellu

Jeżeli następuje przypisanie literału liczby do zmiennej, której typ nie został jawnie określony, WPS tworzy obiekt typu `System.Int32`. Jeżeli zakres wartości `Int32` nie jest wystarczający, stosowany jest typ `Int64` lub `Decimal`. Jeżeli literał liczby jest ułamkiem (separatorem dziesiętnym jest zawsze kropka), WPS deklaruje typ `Double` lub `Decimal`.

Aby przypisać zmiennej określony typ danych, należy poprzedzić deklarację nazwą typu, na przykład `[Byte]` lub `[Decimal]`. W przypadku typu `Decimal` istnieje dodatkowo możliwość zadeklarowania typu przez dodanie do liczby litery „d", na przykład `5.1d`.

```
# Niejawna deklaracja typu Integer
$i = 5
$i.GetType().Name

# Niejawna deklaracja typu Long
$i = 5368888888888888
$i.GetType().Name
```

```
# Niejawna deklaracja liczby typu Decimal
$i = 536888888888888888888888888888
$i.GetType().Name

# Jawna deklaracja typu Long (t.j. 64-bitowej liczby całkowitej)
[Int64] $l = 5
$l.GetType().Name

# Jawna deklaracja typu Byte
[Byte] $b = 5
$b.GetType().Name

# Niejawna deklaracja typu Double
$d = 5.1
$d.GetType().Name

# Jawna deklaracja typu Double
$d = 5.1d
$d.GetType().Name

# Jawna deklaracja typu Decimal
[Decimal] $d = 5.1
$d.GetType().Name
```

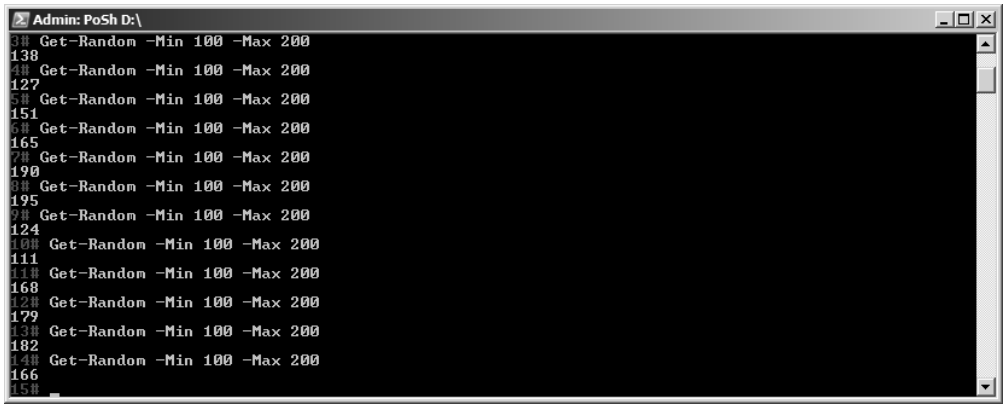
Przy jawnym określaniu typów możemy stosować zarówno typy WPS, takie jak `[int]` i `[long]`, jak i odpowiadające im nazwy klas .NET (`[Sys` ➥`tem.Int32]` i `[System.Int64]`.

OSTRZEŻENIE: Skróty `KB`, `MB` i `GB` pozwalają używać jednostek kilobajta, megabajta i gigabajta (na przykład `5MB` to liczba 5242880 (5*1024*1024).

Skróty te zastąpiły obowiązujące przed wersją WPS 1.0 RC2: `M`, `K` i `G`.

Liczby losowe

Do generowania liczb losowych służy aplet polecenia `Get-Random`, wchodzący w skład rozszerzeń PowerShell Community (PSCX) [CODEPLEX01]. Standardowo, aplet generuje liczby z zakresu od 0 do 1. Można to zmienić przy użyciu parametrów `–Min` i `–Max` (patrz rysunek 6.3).



Rysunek 6.3. Przykład zastosowania apletu polecenia Get-Random do generowania liczb losowych z zakresu od 100 do 200

Ciągi znakowe

Ciągi WPS to egzemplarze klasy .NET `System.String`. Początek i koniec ciągu sygnalizują znaki cudzysłowu lub apostrofu, ewentualnie kombinacja znaków `@`, cudzysłowu i końca wiersza. Ostatnia z tych opcji pozwala zapisywać ciągi wielowierszowe. Określa się ją potocznie nazwą *here-string* („ciąg tutaj", patrz listing 6.1).

Listing 6.1. Przykład here-string

```
# Ciąg tutaj
@'
Długi tekst
można podzielić
na wiele wierszy
używając cudzysłowu lub apostrofu
'@
```

W obu przypadkach, ciągi mogą zawierać zmienne lub wyrażenia, które zostaną automatycznie podstawione. Ilustruje to listing 6.2.

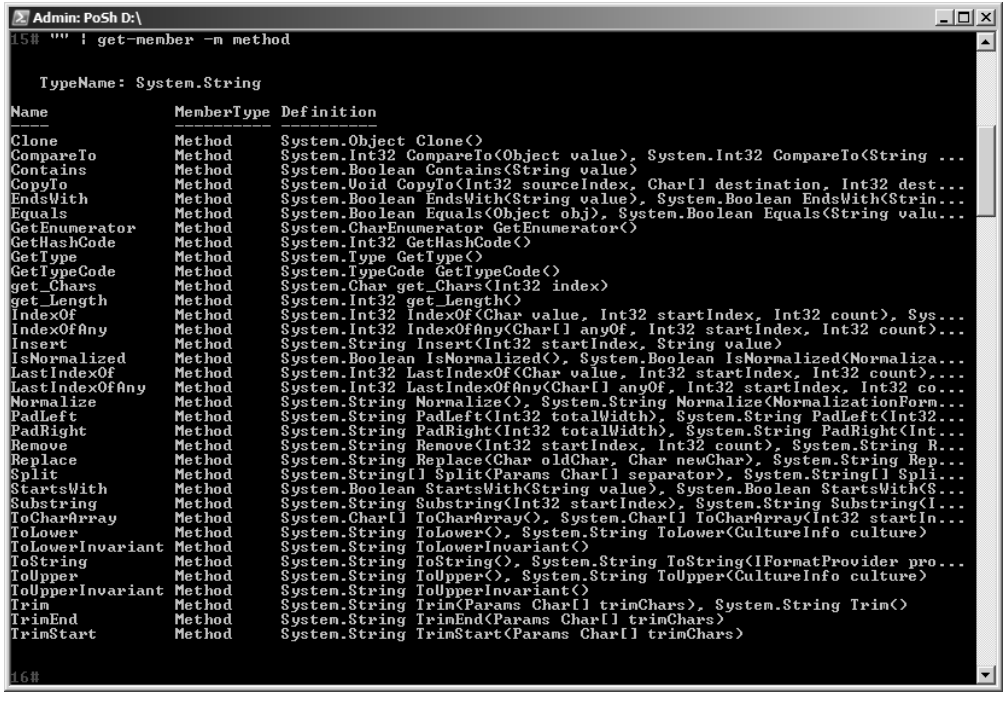
Listing 6.2. Podstawienia zmiennych w ciągu

```
$a = 10
$b = "Bieżąca wartość to $a!"
Write-Warn $b
```

UWAGA: Przy przekazywaniu parametrów do apletów poleceń ciąg można otaczać *tylko* znakami cudzysłowu ("). Jest to konieczne dla zachowania zasad oddzielania kolejnych parametrów.

Praca z ciągami

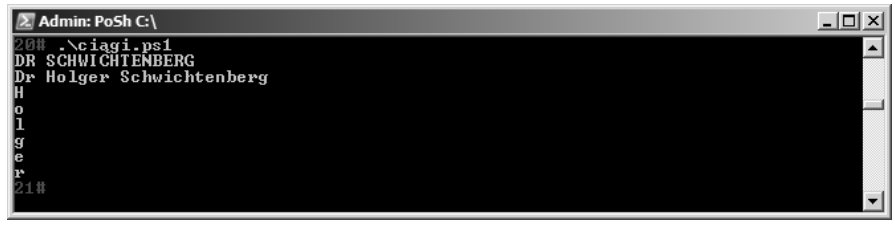
WPS pozwala korzystać z wszystkich możliwości klasy `System.String` (wyposażonej w metody takie jak `Insert()`, `Remove()`, `Replace()` i `Split()`). Pełną listę metod widać na rysunku 6.4. Listing 6.3 przedstawia najbardziej typowe przykłady, a rysunek 6.5 — wynik uruchomienia kodu w listingu.



Rysunek 6.4. Metody klasy System.String

Listing 6.3. Operacje na ciągach

```
# Zmiana wielkości liter
$a = "Dr Schwichtenberg"
$a.ToUpper()
```



Rysunek 6.5. Wynik uruchomienia skryptu z listingu 6.3

```
$b
# Wstawianie ciągu w określonej pozycji
$a = $a.Insert(3, "Holger ")
$a
# Wyodrębnianie części ciągu
$c = $a[3..8]
$c
```

Listing 6.3 przedstawia następujące operacje na ciągach:

- zmiana wszystkich liter na wielkie,
- wstawianie tekstu wewnątrz ciągu,
- wyodrębnianie części ciągu jako listy pojedynczych znaków.

Dzielenie i łączenie ciągów znakowych

Czasem pojawia się potrzeba podzielenia ciągu (na przykład `"Holger;`
`➥Schwichtenberg;Essen;Niemcy;www.IT-Visions.de"`).

.NET Framework zapewnia metodę `Split()` klasy `System.String` (patrz listing 6.4).

Listing 6.4. Przykład użycia metody Split()

```
System.String.
[String] $CSVString =
➥"Holger;Schwichtenberg;Essen;Niemcy;www.IT-Visions.de"
$CSVArray = $CSVString.Split(";")
$Nazwisko = $CSVArray[1]
$Nazwisko
```

Rozszerzenia PSCX wzbogacają WPS o aplet polecenia `Split-String`, którego działanie jest podobne (patrz listing 6.5).

Listing 6.5. Przykład użycia apletu polecenia Split-String

```
[String] $CSVString =
➥"Holger;Schwichtenberg;Essen;Niemcy;www.IT-Visions.de"
$CSVArray = Split-String $CSVString -Separator ";"
$Nazwisko = $CSVArray[1]
$Nazwisko
```

Analogicznie, do łączenia ciągów służy metoda `Join()` i aplet polecenia `Join-String` (patrz listing 6.6 i listing 6.7). Warto od razu zapamiętać, że metoda `Join()` to statyczna metoda klasy `System.String`.

Listing 6.6. Użycie metody statycznej Join()

```
$Array = "Holger", "Schwichtenberg", "Essen", "Niemcy",
➥"www.IT-Visions.de"
$CSVString = [System.String]::Join(";", $Array)
$CSVString
```

Listing 6.7. Użycie apletu polecenia Join-String

```
$Array = "Holger", "Schwichtenberg", "Essen", "Niemcy",
➥"www.IT-Visions.de"
$CSVString = Join-String $Array -Separator ";"
$CSVString
```

Data i godzina

Aplet polecenia `Get-Date` tworzy obiekt klasy .NET `System.DateTime`, który zawiera bieżącą datę i godzinę:

```
Get-Date
```

Następujące polecenie pozwala zredukować ilość danych wyjściowych do samej daty:

```
Get-Date –displayhint date
```

W podobny sposób można ograniczyć ilość danych do samej godziny:

```
Get-Date –displayhint time
```

Aplet `Get-Date` może posłużyć jako konstruktor wartości zapisywanej w zmiennej:

```
$a = Get-Date "1972-8-1 12:11:10"
```

Można obliczyć różnicę między bieżącą godziną (i datą) a godziną (i datą) zapisaną w zmiennej. Służy do tego metoda `Subtract()`:

```
(Get-Date).Subtract((Get-Date "1972-8-1 12:11:10"))
```

Ten sam efekt osiągniemy przy użyciu operatora – (minus):

```
(Get-Date) – (Get-Date "1972-8-1 12:11:10")
```

Powyższe przykłady generują następujące dane wyjściowe:

```
Days : 12662
Hours : 11
Minutes : 56
Seconds : 57
Milliseconds : 927
Ticks : 10940398179276185
TotalDays : 12662,4978926808
TotalHours : 303899,949424338
TotalMinutes : 18233996,9654603
TotalSeconds : 1094039817,92762
TotalMilliseconds : 1094039817927,62
```

Wewnętrznie WPS przetwarza okresy czasu jako obiekty klasy `System.TimeSpan`. Można je tworzyć poleceniem `New-TimeSpan`:

```
$okres = New-TimeSpan –Days 10 –Hours 4 –Minutes 3
➥–seconds 50
$teraz = Get-Date
$przyszłość = $teraz + $okres
```

UWAGA: Polecenie `New-TimeSpan` pozwala określić długość okresu w dniach, godzinach, minutach i sekundach. Nie można stosować miesięcy i lat.

Komputery zdalne

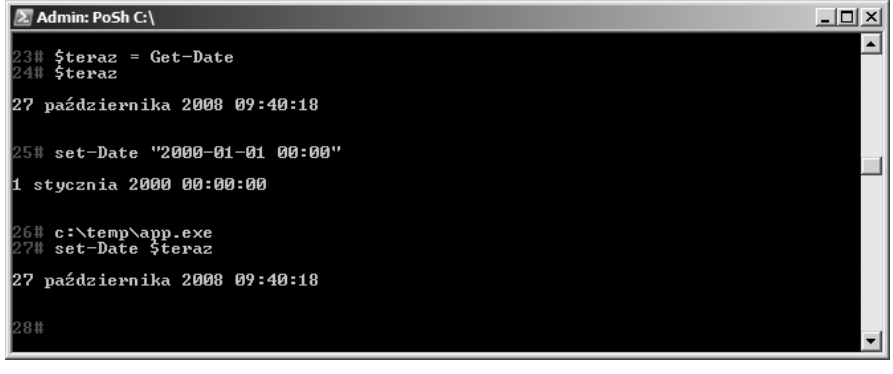
Aplet polecenia `Get-Date` nie pozwala pobrać czasu ze zdalnego komputera. Umożliwia to jednak klasa Windows Management Instrumentation (WMI) `Win32_Currenttime`:

```
Get-Wmiobject Win32_CurrentTime –computername E02
```

Wynikiem powyższej operacji nie jest obiekt .NET `System.DateTime`, ale obiekt .NET typu `System.Management.ManagementObject`. Jest on osłoną obiektu WMI typu `root\cimv2\Win32_LocalTime`.

Zmiana daty i godziny

Do ustawiania daty i godziny komputera służy polecenie `Set-Date` (patrz rysunek 6.6).



Rysunek 6.6. Przykład użycia apletu polecenia Set-Date do uruchomienia aplikacji z inną datą

Tablice

Deklaracja tablicy następuje w momencie przypisania zbioru wartości rozdzielanych przecinkami:

```
$a = 01,08,72,13,04,76
```

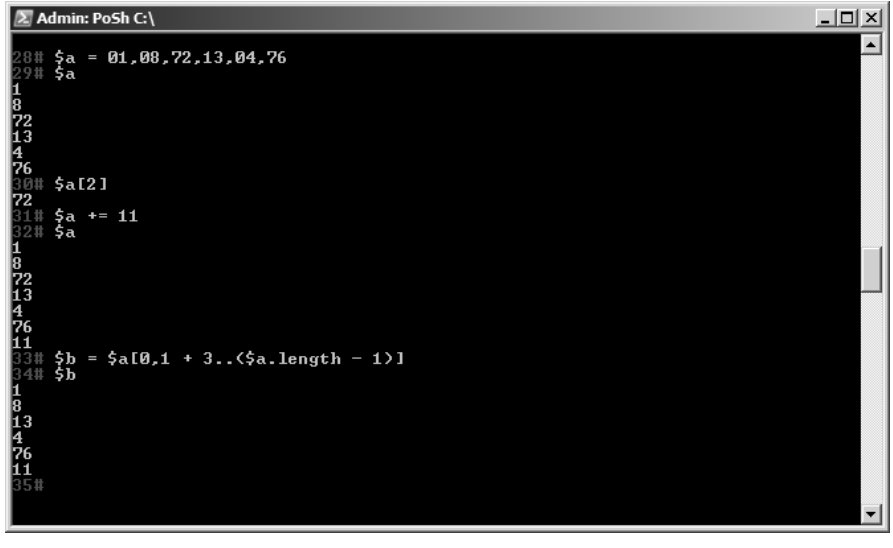
Tablicę można zadeklarować jawnie, używając identyfikatora typu WPS `[Array]`:

```
[Array] $b
$b 1,2,3
```

Aby zdefiniować tablicę jednoelementową, można zastosować deklarację jawną lub rozpocząć listę elementów przecinkiem:

```
$a = ,"Tylko jeden element"
[Array] $a = "Tylko jeden element"
```

Aby wykonać pewne operacje na wszystkich elementach tablicy, można użyć apletu polecenia `Foreach-Object`. Nie zawsze jednak jest to konieczne. Jeżeli tablica jest zwracana przez ostatni aplet polecenia w potoku, zostają wyświetlone wszystkie jej elementy (patrz rysunek 6.7).



Rysunek 6.7. Wyświetlanie zawartości tablicy

Właściwość `Count` pozwala uzyskać informację o ilości elementów w tablicy:

```
[array] $b
$b =1,2,3
$b.Count
```

Dostęp do elementów uzyskujemy po podaniu indeksu (od 0 wzwyż) lub zakresu indeksów, ujętych w nawiasy kwadratowe. Zakres indeksów sygnalizują dwie kropki, na przykład `$a[3..6]`. Operator += zastosowany do tablicy dodaje nowy element na końcu (patrz rysunek 6.7). Usunięcie elementu nie jest możliwe (można jedynie skopiować elementy do innej tablicy).

Tablice można łączyć operatorem + (plus):

```
$KontroleryDomeny = "E01", "E02", "E03"
$SerweryZwykłe = "E04", "E05", "E06"
$WszystkieSerwery = $KontroleryDomeny + $SerweryZwykłe
$WszystkieSerwery.Count # Wynik: 6
```

Dostępne są także tablice wielowymiarowe. Tablica taka powstanie, jeżeli połączymy elementy listy w grupy, stosując znaki nawiasów. Poniższy przykład przedstawia definicję tablicy dwuwymiarowej. Elementy pierwszego wymiaru zawierają trzyelementowe tablice. Czwarty element zostaje dołączony przy użyciu operatora +=:

```
$KontroleryDomeny = ("E01", "192.168.1.10", "Budynek 1"),
↪("E02", "192.168.1.20", "Budynek 2"),
↪("E03", "192.168.1.30", "Budynek 3")
"LiczbaKomputerów: " + $KontroleryDomeny.Count
"Adres IP komputera 2: " + $KontroleryDomeny[1][1]
↪# 192.168.1.20
"Komputer 2 jest w budynku nr: " + $KontroleryDomeny[1][2]
↪# Budynek 3
$KontroleryDomeny += ("E04", "192.168.1.40", "Budynek 4")
"Komputer 4 jest w budynku nr: " + $KontroleryDomeny[3][2]
↪# Budynek 4
```

Tablice asocjacyjne (tabele skrótów)

Poza tablicami, WPS pozwala korzystać z list nazwanych elementów (**listy asocjacyjnych**, ang. *associative list*), zaimplementowanych jako **tabele skrótów** (ang. *hash table*). Element w tabeli skrótów identyfikuje nie pozycja, ale pewna wartość, nazywana **kluczem** (ang. *key*). Podobną strukturę można znaleźć w wielu innych językach. Często określa się ją nazwą **tablica asocjacyjna** (ang. *associative array*). Definicją takich obiektów jest klasa `System.Collections.Hashtable`.

Aby utworzyć tabelę skrótów, używamy znaku `@` oraz listy elementów ujętych w nawiasy klamrowe (`{}`). Kolejne elementy rozdziela średnik (`;`).

Każdy element składa się z nazwy i wartości, rozdzielonych znakiem równości (=). *Nie można* ująć nazwy elementu w cudzysłów. Identyfikatorem służącym do jawnego deklarowania tabel skrótów jest `[Hashtable]`.

```
# Niejawna tabela skrótów
$Computers = @{ E01 = "192.168.1.10"; E02 = "192.168.1.20";
➥E03 = "192.168.1.30"; }
# Jawna tabela skrótów
[Hashtable] $Komputery = @{ E01 = "192.168.1.10"; E02 =
➥"192.168.1.20"; E03 = "192.168.1.30"; }
```

Dostęp do elementów tabeli skrótów można uzyskać, stosując zapis z nawiasami kwadratowymi, podobnie jak do elementów tablic, lub też przy użyciu operatora kropki (`.`). Pozwala to na bardzo przejrzyste odwołania:

```
# Pobieranie adresu IP komputera E02
$Komputery["E02"]
$Komputery.E02
```

Równie prosta jest operacja przypisania:

```
# Zmiana elementu
$Komputery.E02 = "192.168.1.21"
```

Dużą wygodą jest automatyczne tworzenie nowego elementu w momencie przypisania wartości. Pozwala to generować tabelę krok po kroku (zaczynając od listy pustej). Lista pusta to `@{ }`. Oto przykład:

```
# Dodawanie nowego elementu
$Komputery.E04 = "192.168.1.40"
# Rozpoczynanie od listy pustej
$WięcejKomputerów = @{ }
$WięcejKomputerów.E05 = "192.168.1.50"
$WięcejKomputerów.E06 = "192.168.1.60"
$WięcejKomputerów.Count # Wynik: 2
```

Tabele skrótów można łączyć tak samo jak tablice. Wymaga to jednak, aby w obu tabelach nie powtarzały się nazwy elementów. Wystąpieniu duplikatu w tabeli docelowej powoduje błąd czasu wykonania:

```
# Łączenie tabel skrótów
$WszystkieKomputery = $Komputery + $WięcejKomputerów
$WszystkieKomputery.Count # Wynik: 6
```

Tabel skrótów można używać nie tylko do przechowywania rzeczywistych list, ale także jako proste definicje własnych struktur danych (na przykład do zapisania informacji o osobie):

```
# Użycie tabeli skrótów jako struktury danych użytkownika
$Autor = @{ ImięNazwisko="Dr Holger Schwichtenberg";
➥Wiek=35; Kraj="Niemcy" }
$Autor.ImięNazwisko
$Autor.Wiek
$Autor.Kraj
```

Operatory

PowerShell pozwala korzystać z podstawowych operatorów arytmetycznych +, –, *, / i % (modulo, czyli reszta z dzielenia). Znak plus (+) służy do dodawania liczb i łączenia ciągów oraz tablic i tabel skrótów. Znak gwiazdki (*) służy przede wszystkim do mnożenia liczb, ale i on ma dodatkowe zastosowanie: zwielokrotnianie ciągów znakowych i tablic. Pozwala to w prosty sposób generować powtórzenia ich znaków i elementów. Nie można zwielokrotnić tabeli skrótów — w jej naturze leży niepowtarzalność kluczy elementów.

```
# Zwielokrotnienie ciągu znakowego
$Ciąg = "abcdefghijklmnopqrstuvwxyz"
$DługiCiąg = $Ciąg * 20
"Liczba znaków: " + $DługiCiąg.Length # = 520
# Zwielokrotnienie tablicy
$a = 1,2,3,4,5
$b = $a * 10
"Liczba elementów: " + $b.Count # = 50
```

Znak równości (=) służy jako operator przypisania. Szczególnie interesujące są przypisania krzyżowe, umożliwiające elegancką wymianę wartości dwóch zmiennych. W większości języków wymaga to użycia zmiennej pośredniej. W WPS wystarczy napisać `$x, $y = $y, $x` (patrz rysunek 6.8).

Kolejny ciekawy operator to ampersand (&). Może on służyć do wykonania polecenia, którego nazwa zapisana jest w ciągu znakowym. Powstaje wtedy dynamiczny, samomodyfikujący się kod.

Oto przykład:

```
$Co = "Process"
& ("Get-"+$Co)
```



Rysunek 6.8. Wymiana wartości zmiennych przy użyciu przypisania krzyżowego

Powyższa sekwencja poleceń prowadzi do uruchomienia apletu polecenia `Get-Process`. Wartość zmiennej `$Co` można pobierać z różnych źródeł (na przykład od użytkownika).

Operator `&` można zastąpić apletem polecenia `Invoke-Expression`:

```
$WartośćUżytkownika = "Process"
invoke-expression("Get-"+$WartośćUżytkownika)
```

OSTRZEŻENIE: Warto pamiętać, że użycie możliwości dynamicznego wykonywania kodu do interpretowania wpisów użytkownika jako polecenia jest zagrożeniem dla bezpieczeństwa systemu. Choć polecenie `Get` w poprzednim przykładzie może wydawać się nieszkodliwe, wcale tak nie jest:

```
$WartośćUżytkownika = "Process | Stop-Process"
invoke-expression("Get-"+$WartośćUżytkownika)
```

Struktury sterujące

Język skryptów PowerShella (PSL) pozwala stosować następujące struktury sterujące:

```
if (warunek) {...} else {...}
switch ($zmienna) {wartość {...} wartość {...}
➥default {..} } }
while(warunek) { ... }
do { ... } while (warunek)
do { ... } until (warunek)
foreach ($zmienna in $kolekcja) {...}
function nazwa {...}
break
continue
return
exit
```

```
trap błąd class { ... } else { ... }
throw "informacja o błędzie"
throw obiekt_błędu
```

UWAGA: Więcej informacji o tych poleceniach można znaleźć w pomocy PowerShella. W tej książce rezygnujemy ze szczegółowego opisu tych podstawowych konstrukcji języka, aby skupić się na ciekawszych zagadnieniach. Instrukcje sterujące PowerShella nie różnią się znacząco od stosowanych w innych popularnych językach. `Throw` i `Trap` omawiamy w rozdziale 7., „Praca ze skryptami".

Pętle

Listing 6.8 przedstawia proste przykłady konstrukcji `for`, `while` i `foreach`.

Listing 6.8. Pętle

```
# Pętla od 1 do 5
"For:"
for ($i = 1; $i -lt 6; $i++) { $i }

"While:"
$i = 0
while($i -lt 5)
{ $i++
$i
}

"Foreach:"
$i = 1,2,3,4,5
foreach ($z in $i) { $z }
```

Warunki

Listing 6.9 przedstawia proste przykłady instrukcji `if` i `switch`.

Listing 6.9. Warunki

```
if ($i -lt 10)
{ "Mniejsze niż 10" }
else
{ "Większe od 10" }

switch ($i)
```

```
{
    1 {"jeden"}
    5 {"pięć"}
    10 {"dziesięć"}
    default { "inna liczba" }
}
```

Podprocedury

Listing 6.10 przedstawia proste przykłady procedur z parametrami i zwracaniem wartości.

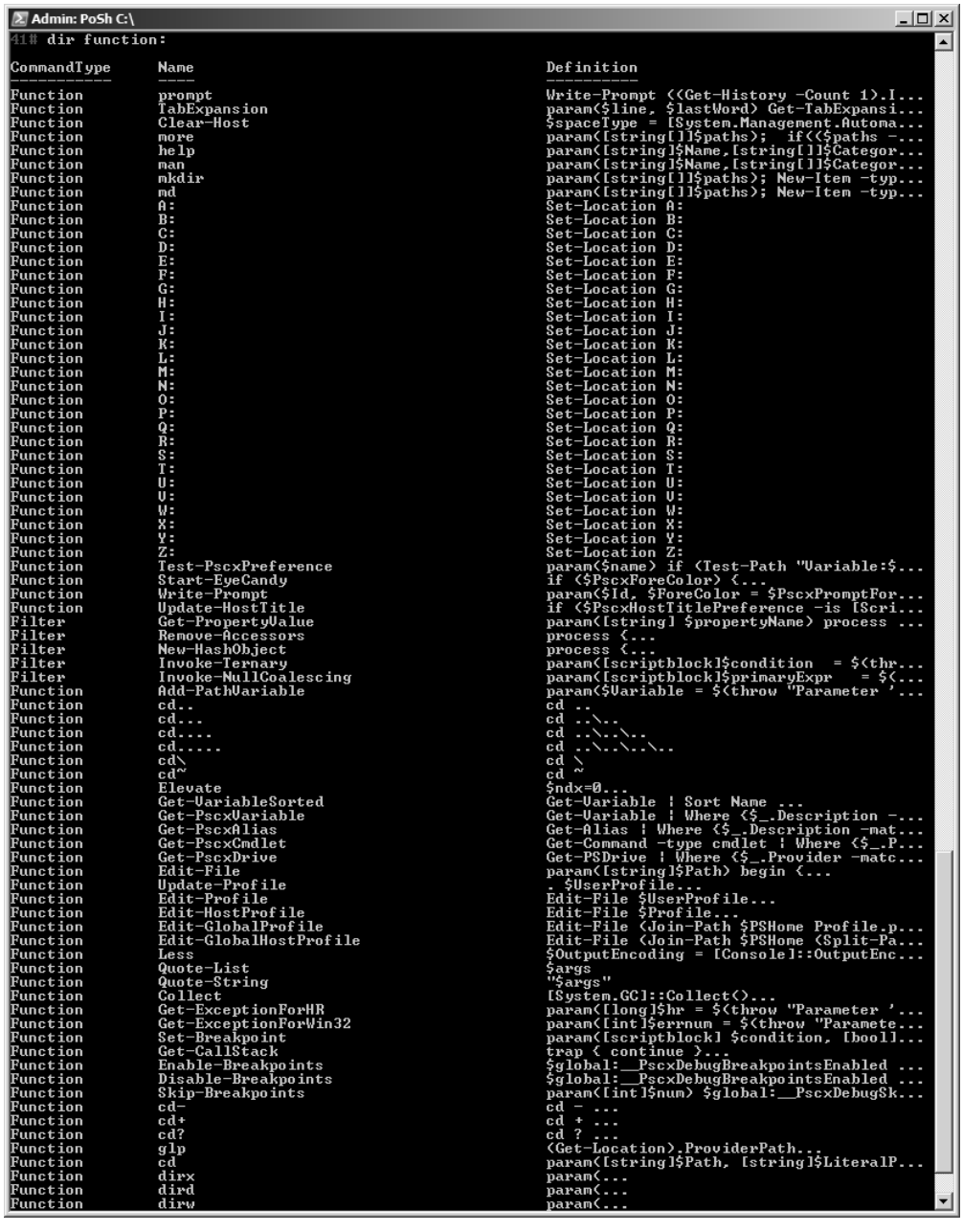
Listing 6.10. Podprocedury

```
function ParametryNieNazwane()
{
"Funkcja pobrała parametry: $args[0] and $args[1]"
return $args[0] + $args[1]
}
ParametryNieNazwane 1 2
function ParametryNazwane([int] $a, [int] $b)
{
"Funkcja pobrała parametry: $a and $b"
return $b + $a
}
ParametryNazwane 1 4
```

WSKAZÓWKA: WPS ma wiele funkcji wewnętrznych (patrz rysunek 6.9). Instalacja PSCX jeszcze powiększa ich zbiór. Polecenie `dir function:` wyświetla ich listę. Łatwo się przekonać, że niektóre polecenia, takie jak `C:` i `Dir`, zapewniające zgodność z tradycyjną konsolą Windows, są zaimplementowane właśnie jako funkcje.

Podsumowanie

Język skryptów PowerShella PSL nie wykorzystuje składni żadnego wcześniejszego języka. Jest podobny do PERL-a, PHP, Pythona i C#. Typ zmiennych może być określany jawnie lub niejawnie. Wszystkie dostępne



Rysunek 6.9. Lista funkcji wewnętrznych (łącznie z funkcjami PSCX)

typy to klasy biblioteki .NET Framework, w tym również typy proste, takie jak `string` czy `int`. Pozwala to korzystać z wszystkich możliwości platformy .NET w zakresie operowania danymi.

Zmienne mogą zawierać pojedyncze wartości, tablice lub tabele skrótów. Elementy tablic są numerowane, a elementy tabel skrótów są identyfikowane nazwami.

Poza zmiennymi, WPS zawiera konstrukcje składniowe do programowania strukturalnego (takie jak pętle, instrukcje wykonania warunkowego i podprocedury).