Technologia i rozwiązania

Windows PowerShell 4.0 dla programistów .NET

Poznaj potencjał zaawansowanej konsoli!

Sherif Talaat

Helion

[PACKT] PUBLISHING enterprise professional expertise distilled

Tytuł oryginału: Windows PowerShell 4.0 for .NET Developers

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-0327-0

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/winpo4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/winpo4.zip

Printed in Poland.

Spis treści

2

ROZDZIAŁ

PowerShell w służbie programisty

Narzędzie Windows PowerShell to rewolucyjne rozwiązanie w dziedzinie skryptów powłoki i technik automatyzacji. I jest tak nie tylko dlatego, że jest to język obiektowy działający na platformie .NET, ale również dlatego, że narzędzie to unifikuje kilka różnych narzędzi skryptowych i automatyzacyjnych w jednym spójnym, dynamicznym mechanizmie. Przy użyciu jednego silnika i języka można obsługiwać różne technologie, takie jak **Windows Management Instrumentation** (**WMI**), **Common Information Model** (**CIM**) oraz **Component Object Model** (**COM**). Ponadto za jego pomocą można zbudować interfejs automatyzacyjny dla tworzonych aplikacji.

W rozdziale tym zajrzymy głębiej do Windows PowerShell, aby dobrze poznać różne technologie, którymi można zarządzać za pomocą tego narzędzia, nauczyć się tego robić oraz poznać bardziej zaawansowane techniki pisania skryptów, bazując na posiadanych już umiejętnościach programistycznych.

W tym rozdziale omawiam następujące tematy:

- Podstawy WMI, CIM, COM oraz XML
- Rozszerzanie funkcjonalności narzędzia Windows PowerShell przy użyciu technologii .NET, COM, XML oraz WMI
- Moduły Windows PowerShella
- Tworzenie modułów Windows PowerShella
- Debugowanie skryptów i obsługa błędów

CIM i WMI

CIM to otwarty standard zdefiniowany przez **Distributed Management Task Force** (**DMTF**) w ramach inicjatywy **Web-Based Enterprise Management** (**WBEM**). Standard CIM określa rozszerzalny model danych opisujący, przetwarzający i pozyskujący charakterystyczne informacje o zarządzanych zasobach, takich jak składniki sprzętowe i oprogramowanie. CIM jest obiektowym modelem programowania niezależnym od produkcji, co znaczy, że można zarządzać różnymi zasobami od różnych dostawców przy użyciu tylko standardu CIM. Natomiast WMI jest implementacją standardu CIM firmy Microsoft, wprowadzoną w systemie Windows 2000, aby umożliwić zarządzanie wszystkimi składnikami programowymi i sprzętowymi systemu.

CIM i WMI w Windows PowerShellu

W Windows PowerShellu 2.0 zaimplementowano kilka poleceń umożliwiających wykorzystanie WMI jako warstwy pośredniej między użytkownikiem końcowym (administratorem systemu i programistą) a CIM. Później, w Windows PowerShellu 3.0, wprowadzono bezpośrednią obsługę standardu CIM w postaci dodatkowych poleceń w systemach Windows Server 2012 i Windows 8, dzięki czemu użytkownicy narzędzia PowerShell mogą bezpośrednio ujawniać schemat i model danych CIM.

Aby wyświetlić listę poleceń dla WMI i CIM, należy użyć polecenia `Get-Command` z parametrem `-Name` przefiltrowanym przez wieloznacznik i parametrem `-Type` przefiltrowanym przez argument `cmdlet` powodujący pobranie tylko poleceń, z pominięciem funkcji i aliasów. Opisane polecenie jest pokazane poniżej:

```
# Wyświetla listę wszystkich poleceń dotyczących WMI
PS C:\> Get-Command *WMI* -Type Cmdlet
CommandType     Name               ModuleName
-----------     ----               ----------
Cmdlet          Get-WmiObject      Microsoft.PowerShell.Management
Cmdlet          Invoke-WmiMethod   Microsoft.PowerShell.Management
Cmdlet          Register-WmiEvent  Microsoft.PowerShell.Management
Cmdlet          Remove-WmiObject   Microsoft.PowerShell.Management
Cmdlet          Set-WmiInstance    Microsoft.PowerShell.Management
# Wyświetla listę wszystkich poleceń dotyczących CIM
PS C:\> Get-Command *CIM* -Type Cmdlet
CommandType     Name               ModuleName
-----------     ----               ----------
Cmdlet          Get-CimAssociatedInstance   CimCmdlets
Cmdlet          Get-CimClass                CimCmdlets
Cmdlet          Get-CimInstance             CimCmdlets
Cmdlet          Get-CimSession              CimCmdlets
Cmdlet          Invoke-CimMethod            CimCmdlets
Cmdlet          New-CimInstance             CimCmdlets
Cmdlet          New-CimSession              CimCmdlets
Cmdlet          New-CimSessionOption        CimCmdlets
```

```
Cmdlet          Register-CimIndicationEvent CimCmdlets
Cmdlet          Remove-CimInstance          CimCmdlets
Cmdlet          Remove-CimSession           CimCmdlets
Cmdlet          Set-CimInstance             CimCmdlets
```

A teraz przyjrzyj się obu tym listom. Zauważyłeś pewną prawidłowość? Niektóre polecenia CIM są podobne do poleceń WMI, co w świetle wcześniejszej informacji, że WMI jest implementacją CIM, nie powinno być żadnym zaskoczeniem.

Mimo że obie listy poleceń wyglądają prawie tak samo, można zauważyć, że polecenia z listy CIM mają więcej parametrów i że jest ich więcej, dzięki czemu można przy ich użyciu uzyskać więcej informacji.

CIM i WMI reprezentują dostarczone informacje w postaci przestrzeni nazw i klas. Na przykład istnieje klasa dla BIOS o nazwie `Win32_BIOS` i inna klasa dla systemu operacyjnego o nazwie `Win32_OperatingSystem`. Istnieją też klasy, których nazwy zaczynają się od znaku _, w rodzaju `_NAZWAKLASY` na wewnętrzny użytek systemu operacyjnego oraz `CIM_NAZWAKLASY` dla pewnych podstawowych klas, ale najczęściej używane są klasy o nazwach typu `Win32_NAZWAKLASY`.

Jeśli nie znasz nazwy klasy lub chcesz obejrzeć listę klas dostępnych w swoim systemie, możesz użyć polecenia `Get-WmiObject -List` lub `Get-CimClass`.

```
# Wyświetla listę dostępnych klas przy użyciu WMI
PS C:\> Get-WmiObject –Class * -List

# Wyświetla listę dostępnych klas przy użyciu CIM
PS C:\> Get-CimClass –ClassName *
Win32_CurrentTime
Win32_LocalTime
Win32_OperatingSystem
Win32_Process
Win32_ComputerSystem
Win32_BIOS
Win32_SoftwareElement
(...)

# Porównuje liczby klas pobranych przez oba polecenia
PS C:\> (Get-WmiObject -List).count -eq (Get-CimClass).count
True
```

Po znalezieniu potrzebnej klasy należy utworzyć jej egzemplarz, aby pobrać dostarczane przez nią informacje. W tym celu można użyć polecenia `Get-WmiObject` lub `Get-CimInstance`, podając nazwę klasy jako parametr.

```
# Tworzy egzemplarz klasy przy użyciu WMI
PS C:\> Get-WmiObject -Class Win32_BIOS

# Tworzy egzemplarz klasy przy użyciu CIM
PS C:\> Get-CimInstance -ClassName Win32_BIOS
```

```
SMBIOSBIOSVersion : 8BET59WW (1.39 )
Manufacturer      : LENOVO
Name              : Default System BIOS
SerialNumber      : R9T081V
Version           : LENOVO - 1390
```

Ponadto zamiast nazwy klasy można użyć parametru `-Query`, aby wykonać predefiniowane zapytanie WMI napisane w języku ***WMI Query Language*** (**WQL**).

```
# Tworzy zapytanie WQL odczytujące dane z klasy Win32_NetworkAdapter
PS C:\> $Query = "Select * From Win32_NetworkAdapter Where Name like '%Intel%'"

# Wykonuje zapytanie WQL przy użyciu WMI
PS C:\> Get-WmiObject -Query $Query | Select DeviceID, Name

# Wykonuje zapytanie WQL przy użyciu CIM
PS C:\> Get-CimInstance -Query $Query | Select DeviceID, Name

DeviceID             Name
--------             ----
0                    Intel(R) 82579LM Gigabit Network Connection
2                    Intel(R) Centrino(R) Ultimate-N 6300 AGN
```

W podobny sposób można usunąć egzemplarz klasy. Polecenia `Remove-WmiObject` i `Remove-CimInstance` służą do usuwania obiektów, a polecenia `Set-WmiInstance` i `Set-CimInstance` do ich modyfikowania. Poniżej znajduje się przykład pobrania informacji o wybranym folderze w instancji WMI i usunięcia go za pomocą poleceń `Get-WmiObject` i `Remove-WmiObject`.

```
# Pobiera folder o nazwie myOldBackup
$folder = Get-WmiObject -Class Win32_Directory -Filter "Name='D:\\myOldBackup'"
# Usuwa ten folder
$folder | Remove-WmiObject
```

Inne ciekawe polecenia to `Register-CimIndicationEvent` i `Register-WmiEvent`. Za ich pomocą można wykonać pewne działania w bloku skryptowym PowerShella w reakcji na wystąpienie pewnych zdarzeń WMI lub CMI. Przykładowo można wysłać powiadomienie o tym, że stopień wykorzystania procesora przekroczył 85% albo że jakaś usługa przestała działać.

Powody przyjęcia standardu CIM

Wiedząc, czym są WMI i CIM, i po przeanalizowaniu paru podobnych poleceń z obu technologii pewnie się zastanawiasz, po co w ogóle wprowadzono nowe polecenia CIM i dlaczego ktoś miałby ich używać, skoro dostępne w poprzednich wersjach Windows PowerShella polecenia WMI są równie dobre.

Aby rozwiać Twoje wątpliwości i uniknąć dyskusji filozoficznych, poniżej przedstawiam listę zalet CIM w punktach:

- Jest to otwarty standard, co oznacza, że nie jest on zarezerwowany tylko dla systemu Windows, dzięki czemu za pomocą CIM można zarządzać także innymi dostawcami i producentami.
- Do zdalnego zarządzania CIM używa protokołu **WS-Management** (**WS-MAN**), dzięki czemu można pracować z każdym zdalnym serwerem i urządzeniem zawierającym implementację tego protokołu. Natomiast za pomocą WMI można zarządzać tylko systemem Windows za pośrednictwem protokołu DCOM.
- CIM można używać z urządzeniami zgodnymi z **Open Management Infrastructure** (**OMI**).

Więcej informacji na temat OMI znajduje się w artykule dostępnym pod adresem *http://blogs.technet.com/b/windowsserver/archive/2012/06/28/open-management-infrastructure.aspx*.

- Za pomocą CIM można zarządzać każdym komputerem i urządzeniem z systemem operacyjnym zgodnym z **CIM Object Manager** (**CIMOM**), niezależnie od dostawcy. W związku z tym przy użyciu CIM można zarządzać zarówno systemami operacyjnymi Windows, jak i innymi.

Praca z XML

Przetwarzanie i zapisywanie danych w formacie XML jest jedną z najczęściej wykonywanych czynności przez programistów. PowerShell zawiera wbudowane narzędzie do obsługi formatu XML, przy użyciu którego praca z plikami w tym formacie jest łatwa i nie wymaga pisania dużej ilości kodu. To wystarczy, aby używać konsoli PowerShell na co dzień do pracy z danymi w formacie XML.

Wczytywanie plików XML

Pliki XML można wczytywać w konsoli PowerShell na dwa sposoby: za pomocą polecenia `Get-Content` lub `Select-Xml` z zapytaniami `XPath`.

Polecenie Get-Content

Aby załadować plik i odczytać jego zawartość, można użyć polecenia `Get-Content`. Służy ono do ładowania treści ze zwykłych plików tekstowych i XML, które również są tekstowe, ale zawierają dane o określonej strukturze.

```
# Wczytuje zawartość pliku za pomocą polecenia
PS C:\> Get-Content C:\Employees.xml
```

Polecenie to wczyta zawartość pliku XML jako zwykły tekst. Aby konsola potraktowała tę treść jako dane w formacie XML, należy dokonać rzutowania wyniku polecenia `Get-Content` albo zapisać go w zmiennej o ściśle określonym typie XML, jak zostało pokazane poniżej:

```
# Rzutowanie wyników
$employee = [xml](Get-Content D:\Employees.xml)

# Zapisanie wyniku w zmiennej typu XML
[xml] $employees = Get-Content D:\Employees.xml
```

Zmiennym o ściśle określonym typie, np. `[xml] $employees`, można przypisywać tylko obiekty typu `System.Xml.XmlDocument`. Inaczej nastąpi zgłoszenie błędu.

Poniżej znajduje się przykładowa treść pliku *Employees.xml*:

```
<staff>
<branch location="cairo">
    <employee>
        <Name>Sherif Talaat</Name>
        <Role>IT</Role>
    </employee>
</branch>
</staff>
```

Załadowany plik XML zawiera informacje dotyczące pracowników różnych działów firmy. Jego treść jest już zapisana w zmiennej `$employees`, której można używać w normalny sposób, tak jak używa się wszystkich obiektów obsługujących XML. Pokazuje to poniższy przykład:

```
# Odczytuje węzły potomne dokumentów XML
PS C:\> $employees.staff.ChildNodes
location                          employee
--------                          --------
cairo                             {Sherif Talaat, Raymond Elias}
redmond                           {Bill Gates, Steve Jobs}

# Pobiera informacje z atrybutów węzłów
PS C:\> $employees.staff.branch.Get_Attributes()

#text
-----
cairo
Redmond

# Pobiera wartości atrybutów wg nazw atrybutów
PS C:\> $employees.staff.branch. location
cairo
Redmond

# Zmienia wartość atrybutu
PS C:\> $employees.staff.branch[0]. location
```

```
= 'Seattle'

# Zmienia i modyfikuje jeden węzeł
PS C:\> $employees.staff.branch.employee

Name                    Role
----                    ----
Sherif Talaat           IT
Raymond Elias           Inżynier

PS C:\> $emp = $employees.staff.branch.employee[0]

PS C:\> $emp.Role = "PowerShell Guru"

PS C:\> $employees.SelectNodes("//employee[Name='Sherif Talaat']")

Name                            Role
----                            ----
Sherif Talaat                   PowerShell Guru
# Dodaje nowy węzeł
PS C:\> $newemployee = $employees.CreateElement("employee")
PS C:\> $newemployee.set_InnerXML("<Name>Ahmad Mofeed</
Name><Role>Security Consultant</Role>")
PS C:\> $employees.staff.branch[0].AppendChild($newemployee)

PS C:\> $employees.staff.branch[0].employee

Name                             Role
----                             ----
Sherif Talaat                    PowerShell Guru
Raymond Elias                    Inżynier
Ahmad Mofeed                     Security Consultant
```

Polecenie Select-Xml

Innym sposobem załadowania zawartości pliku XML do PowerShella jest użycie polecenia `Select-Xml`, za pomocą którego można wpisać bezpośrednią ścieżkę do pliku z dodatkiem zapytań `XPath` służących do pobrania określonych danych i węzła, jak zostało pokazane poniżej:

```
#Get data from XML file using XPath query
PS C:\> Select-Xml -Path D:\Employees.xml -XPath "staff/branch/
employee"

Node                    Path                Pattern
----                    ----                -------
employee                D:\Employees.xml    staff/branch/employee
employee                D:\Employees.xml    staff/branch/employee
employee                D:\Employees.xml    staff/branch/employee
```

Polecenie to pobiera węzły XML za pomocą zapytania XPath. Wynikiem jest obiekt węzłów bez wartości. Aby rozwinąć te węzły i wypisać ich wartości, należy użyć polecenia Select-Object z parametrem -ExpandProperty.

```
PS C:\> Select-Xml -Path D:\Employees.xml -XPath "staff/branch/
employee" | Select-Object -ExpandProperty Node

Name                            Role
----                            ----
Sherif Talaat                   IT
Raymond Elias                   Inżynier
Bill Gates                      Programista
```

Importowanie i eksportowanie plików XML

W PowerShellu znajduje się też kilka poleceń przeznaczonych specjalnie do pracy z danymi w formacie XML. Są to polecenia Export-CliXml, służące do eksportowania obiektów do plików XML, oraz Import-CliXml, służące do importowania i ładowania wcześniej wyeksportowanych za pomocą PowerShella plików, jak w poniższym przykładzie:

```
# Eksportuje obiekt do pliku XML
PS C:\> Get-Service | Export-Clixml D:\Services.xml

# Importuje obiekt z pliku XML
PS C:\> Import-Clixml D:\Services.xml -First 5

Status Name               DisplayName
------ ---- -----------
Running AdobeARMservice   Adobe Acrobat Update Service
Stopped AeLookupSvc       Application Experience
Stopped ALG Application   Layer Gateway Service
Running AppIDSvc          Application Identity
Running Appinfo           Application Information
```

Ponadto istnieje jeszcze polecenie ConvertTo-Xml, które działa podobnie jak Export-CliXml pod tym względem, że tak jak ono tworzy reprezentację jednego lub większej liczby obiektów w formacie XML. Jedyna różnica między nimi polega na tym, że Export-CliXml zapisuje kod XML w pliku, a ConvertTo-Xml zwraca obiekt XML, który można przekazać na wejście innego polecenia.

Obiekty typu COM

W PowerShellu można pracować także z obiektami typu COM. W tym podrozdziale objaśniam sposób działania technologii COM w PowerShellu na bazie dwóch przykładów dotyczących przeglądarki Internet Explorer i programu Microsoft Excel.

Tworzenie egzemplarza obiektu COM

Aby utworzyć obiekt COM, należy użyć polecenia `New-Object` z parametrem `-ComObject` i argumentem `ProgID` reprezentującym przyjazną nazwę klasy COM użytą podczas rejestracji klasy. Zatem ostatecznie całe polecenie powinno wyglądać tak:

```
# Tworzy nowy obiekt COM
PS C:\> $com = New-Object –ComObject <ProgID>
```

Automatyzacja przeglądarki Internet Explorer przy użyciu technologii COM i narzędzia PowerShell

Jak napisałem powyżej, do utworzenia egzemplarza aplikacji potrzebny jest argument `ProgID`. W przypadku przeglądarki Internet Explorer wartość tego argumentu to `InternetExplorer.Application`. Mając te informacje, możemy utworzyć obiekt COM Internet Explorera i rozpocząć z nim pracę.

Pierwszą czynnością jest utworzenie obiektu za pomocą polecenia `New-Object` i zapisanie go dla wygody w zmiennej o nazwie `$ie`.

```
# Tworzy nowy obiekt klasy COM IE
PS C:\> $ie = New-Object -ComObject InternetExplorer.Application
```

Następnie można zdefiniować własności utworzonego egzemplarza. W przypadku Internet Explorera należy zdefiniować wysokość i szerokość okna, jego widoczność, adres URL itd.

```
$ie.navigate("about:blank")
$ie.height = 800
$ie.width = 1200
$ie.visible = $true
```

> Więcej informacji na temat obiektowego modelu Internet Explorera znajduje się na stronie *http://msdn.microsoft.com/en-us/library/ms970456.aspx*.

Powyższy kod spowoduje uruchomienie okna przeglądarki IE z pustą stroną. Czy to wszystko, co możemy zrobić z tą przeglądarką? Oczywiście, że nie. Jest wiele ciekawszych możliwości. Teraz napiszemy kod przeszukujący witrynę *outlook.com*, znajdujący pola tekstowe adresu i hasła, wstawiający w nich dane i klikający przycisk *Zaloguj*.

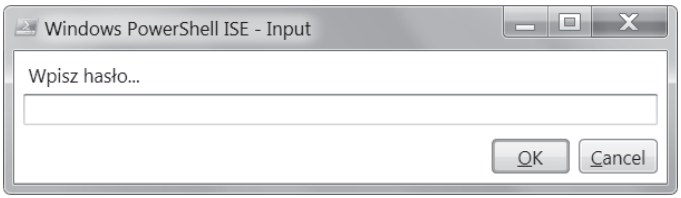
Najpierw wyświetlimy prośbę o wpisanie adresu e-mail i hasła za pomocą polecenia `Read-Host`.

```
$EmailAddress = Read-Host -Prompt "Wpisz nazwę konta Microsoft..."
```

Dla hasła należy dodać parametr `-AsSecureString`, aby zamiast wpisywanych liter w polu pojawiały się gwiazdki i aby hasło zostało zapisane w zaszyfrowanej zmiennej.

```
$Password = Read-Host -AsSecureString -Prompt "Wpisz hasło..."
```

Okno z polem do wpisania hasła powinno wyglądać tak:



Następnie utworzymy obiekt COM i zdefiniujemy jego własności. Tym razem zamiast pustej strony otworzymy stronę *outlook.com*.

```
$ie = New-Object -ComObject InternetExplorer.Application
$ie.height = 800
$ie.width = 1200
$ie.navigate("http://outlook.com")
$ie.visible = $true
```

Aby skrypt poprawnie zadziałał, przed wykonaniem następnego polecenia należy się upewnić, czy strona jest już w pełni załadowana.

```
while($ie.Busy){Start-Sleep -Milliseconds 500}
```

Teraz przeglądamy znajdujące się na stronie elementy, pola tekstowe i przyciski i wstawiamy do nich otrzymane od użytkownika wartości. Do przeglądania elementów strony internetowej można używać narzędzi programistycznych przeglądarki Internet Explorer uruchamianych za pomocą klawisza *F12*.

```
$doc = $ie.document
$tbUsername = $doc.getElementByID("i0116")
$tbUsername.value = $EmailAddress
$tbPassword = $doc.getElementByID("i0118")
$tbPassword.value = $Password
$btnSubmit = $doc.getElementByID("idSIButton9")
```

Na koniec wywołujemy zdarzenie `Click` na przycisku *Zaloguj*.

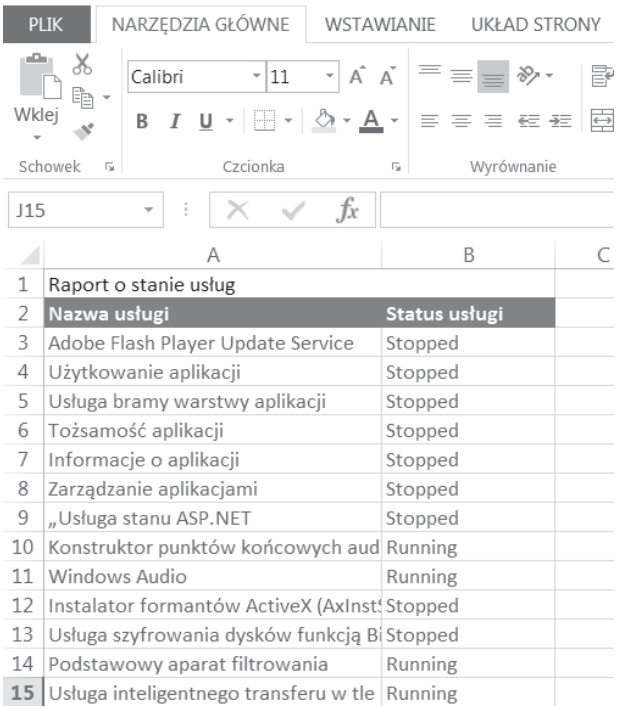
```
$btnSubmit.Click();
```

Teraz powinieneś widzieć swoją skrzynkę odbiorczą. Ciekawe, prawda?

Automatyzacja programu Microsoft Excel przy użyciu technologii COM i narzędzia PowerShell

Innym popularnym zastosowaniem technologii COM jest automatyzacja programów z pakietu Microsoft Office. W tym podrozdziale pokazuję, jak pracować z klasą COM programu Microsoft Excel, ale wszystkie wskazówki w równym stopniu dotyczą także programów Word, Access, Outlook itd.

W ramach przykładu utworzymy raport na podstawie arkusza kalkulacyjnego Excel (pokazany na poniższym zrzucie ekranu) przedstawiającego aktualny stan wszystkich usług systemu Windows. Wyznaczony cel w poniższym przykładowym kodzie osiągniemy przy użyciu interfejsu COM programu Excel `Excel.Application`.



	A	B	C
1	Raport o stanie usług		
2	**Nazwa usługi**	**Status usługi**	
3	Adobe Flash Player Update Service	Stopped	
4	Użytkowanie aplikacji	Stopped	
5	Usługa bramy warstwy aplikacji	Stopped	
6	Tożsamość aplikacji	Stopped	
7	Informacje o aplikacji	Stopped	
8	Zarządzanie aplikacjami	Stopped	
9	„Usługa stanu ASP.NET	Stopped	
10	Konstruktor punktów końcowych aud	Running	
11	Windows Audio	Running	
12	Instalator formantów ActiveX (AxInst	Stopped	
13	Usługa szyfrowania dysków funkcją Bi	Stopped	
14	Podstawowy aparat filtrowania	Running	
15	Usługa inteligentnego transferu w tle	Running	

Pierwszą czynnością jest utworzenie egzemplarza `Excel.Application`.

```
$Excel = New-Object -ComObject Excel.Application
```

W tym momencie w Windowsie powinien zostać już uruchomiony proces Excela, ale samo okno programu stanie się widoczne dopiero po ustawieniu jego widoczności.

```
$Excel.visible = $True
```

Następnie musimy utworzyć skoroszyt i dodać do niego jeden arkusz kalkulacyjny.

```
$ExcelWB = $Excel.Workbooks.Add()
$ExcelWS = $ExcelWB.Worksheets.Item(1)
```

Po przygotowaniu podstawowych składników możemy przystąpić do wypełniania arkusza danymi. Najpierw w pierwszym wierszu zapiszemy tytuł raportu.

```
$ExcelWS.Cells.Item(1,1) = "Raport o stanie usług"
$ExcelWS.Range("A1","B1").Cells.Merge()
```

Następnie w drugim wierszu utworzymy nagłówek tabeli składający się z dwóch kolumn: *Nazwa usługi* i *Stan usługi*.

```
$ExcelWS.Cells.Item(2,1) = "Nazwa usługi"
$ExcelWS.Cells.Item(2,2) = "Stan usługi"
```

Później za pomocą polecenia `Get-Service` pobierzemy listę wszystkich usług działających w systemie Windows i za pomocą pętli `ForEach` utworzymy dla każdej z nich po jednym wierszu w arkuszu.

```
$row = 3
ForEach($Service in Get-Service)
{
    $ExcelWS.Cells.Item($row,1) = $Service.DisplayName
    $ExcelWS.Cells.Item($row,2) = $Service.Status.ToString()
    if($Service.Status -eq "Running")
        {
            $ExcelWS.Cells.Item($row,1).Font.ColorIndex = 10
            $ExcelWS.Cells.Item($row,2).Font.ColorIndex = 10
        }
    Elseif($Service.Status -eq "Stopped")
        {
            $ExcelWS.Cells.Item($row,1).Font.ColorIndex = 3
            $ExcelWS.Cells.Item($row,2).Font.ColorIndex = 3
        }
  $row++
}
```

Na koniec zapisujemy raport i zamykamy instancję programu Excel.

```
$ExcelWS.SaveAs("D:\ServicesStatusReport.xlsx")
$Excel.Quit()
```

> Więcej informacji na temat interfejsu COM programu Excel znajduje się na stronie *http://msdn.microsoft.com/en-us/library/microsoft.office.interop.excel.application.aspx*.

Obiekty .NET

W rozdziale 1. opisałem, co łączy narzędzie Windows PowerShell i platformę .NET, a także pokazałem różne sposoby adaptacji tej platformy w PowerShellu. W tym podrozdziale poszerzymy wiadomości na temat obiektów .NET w PowerShellu.

Tworzenie obiektów .NET

Do tworzenia obiektów .NET najczęściej używa się polecenia `New-Object`, które ma podobne działanie jak operator `new` w takich językach jak C#. Tak, napisałem „najczęściej", bo można też rzutować obiekt PowerShella na .NET w sposób pokazany w rozdziale 1. Polecenia `New-Object` używa się do tworzenia zarówno obiektów .NET, jak i COM, ale w tym drugim przypadku należy podać inne parametry.

```
PS C:\> $date = New-Object -TypeName System.DateTime -ArgumentList 2013,10,24
PS C:\> $date
24 października 2013 00:00:00
```

Typ obiektu można zdefiniować bezpośrednio, bez użycia parametru `-TypeName`, ponieważ jest to parametr pozycyjny, a więc taki, którego nazwę można opuścić.

```
PS C:\> $string = New-Object System.String -ArgumentList "PowerShell jest super!"
PS C:\> $string
PowerShell jest super!
```

W tym przykładzie utworzyliśmy za pomocą polecenia `New-Object` dwa obiekty .NET typów `DateTime` i `String` oraz przekazaliśmy wartości do konstruktora przy użyciu parametru `-ArgumentList`.

Kod ten jest równoważny z poniższym:

```
PS C:\> [datetime] $date = "2013/10/24"

PS C:\> [string] $string = "PowerShell jest super!"
```

Rozszerzanie obiektów .NET

Egzemplarze obiektów .NET można rozszerzać o własne własności i składowe, które dodaje się za pomocą polecenia `Add-Member`.

Poniżej znajduje się przykład dodania za pomocą polecenia `Add-Member` składowej `NoteProperty` do istniejącego obiektu. W tym przypadku ładujemy plik XML w obiekcie `xml`, a następnie dodajemy do niego składową typu `NoteProperty` o nazwie `Description`, w której wpisujemy opis zawartości pliku.

```
#Load XML file
PS C:\> [xml] $xml = Get-Content D:\Employees.xml

#Add new NoteProperty Member using Add-Member
PS C:\>Add-Member -InputObject $xml -MemberType NoteProperty -Name
Description -Value "Baza danych pracowniczych"

#Show the new added member
```

```
PS C:\> $xml | Get-Member -MemberType NoteProperty | fl
TypeName   : System.Xml.XmlDocument
Name       : Description
MemberType : NoteProperty
Definition : System.String Description=Employees information database
```

W drugim przykładzie pokażę, jak dodać własną metodę typu `ScriptMethod`, która będzie wykonywała blok skryptowy na obiekcie `array`. Metodę tę nazwiemy `Censored()` i będzie ona sprawdzała tekst i zastępowała nieprzyzwoite słowa gwiazdkami:

```
# Tworzy tablicę adresów URL
PS C:\> $websites = @("facebook.com","twitter.com","google.com","xxx.com")

# Dodaje do obiektu tablicowego nową składową typu ScriptMethod
PS C:\> Add-Member -InputObject $websites -MemberType ScriptMethod -Name
Censored -Value {$this -replace "xxx","***"}

# Wykonuje nowo dodaną metodę
PS C:\> $websites.Censored()
facebook.com
twitter.com
google.com
***.com
```

Więcej informacji na temat typów składowych znajduje się w artykule na stronie *http://msdn.microsoft.com/en-us/library/windows/desktop/system.management.automation.psmembertypes(v=vs.85).aspx*.

Rozszerzanie typów platformy .NET

W Windows PowerShellu można definiować typy (klasy) platformy .NET, aby móc później tworzyć ich obiekty za pomocą polecenia `New-Object`. Typy te można definiować w plikach z kodem źródłowym, plikach złożeń, a nawet przy użyciu śródliniowego kodu w językach C#, VB oraz JScript.

Definiowanie typu obiektów przy użyciu śródliniowej klasy C#

Poniżej znajduje się przykład utworzenia nowego typu obiektów ze śródliniowej klasy w języku C#. Najpierw została utworzona prosta klasa reprezentująca kalkulator zawierająca cztery metody odpowiadające czterem działaniom arytmetycznym. Następnie za pomocą polecenia `Add-Type` dodamy tę klasę do bieżącej sesji PowerShella. Na koniec utworzymy nowy obiekt tej klasy za pomocą polecenia `New-Object`.

```
PS C:\> $myCalc = @"
public class PSCalc
{
    public int Add(int x, int y) {return x + y;}
```

```
    public int Subtract(int x, int y) {return x – y;}
    public int Multiply(int x, int y) {return x * y;}
    public int Divid(int x, int y) {return x / y;}
}
"@

PS C:\> Add-Type –TypeDefinition $myCalc

PS C:\> $op = New-Object PSCalc
```

Teraz przy użyciu obiektu `$op` można wykonać dowolną ze zdefiniowanych w klasie metod.

```
PS C:\> $op.Multiply(4,8)
32
```

Ciekawe jest to, że można nawet wywoływać statyczne metody klas bezpośrednio w PowerShellu. Na przykład klasa `System.Math` zawiera wiele metod statycznych, z których jedna nazywa się `Pow()` i służy do obliczania potęg. Aby wywołać ją w PowerShellu, należy napisać następujące polecenie:

```
PS C:\> [System.Math]::Pow(2,4)
```

Definiowanie typów obiektowych przy użyciu nazwy złożenia lub pliku

Innym sposobem zdefiniowania nowego typu obiektowego jest użycie nazwy złożenia (przestrzeni nazw) lub pliku złożenia (DLL) i wykonanie polecenia `New-Object`.

Poniżej znajduje się przykład dodania nowego typu przy użyciu nazwy złożenia:

```
PS C:\> $form = New-Object System.Windows.Forms

New-Object : Cannot find type [System.Windows.Forms]: verify that the assembly
↪containing this type is loaded.
At line:1 char:8
+ $form = New-Object System.Windows.Forms
+        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidType: (:) [New-Object],
    ↪PSArgumentException
    + FullyQualifiedErrorId  :
TypeNotFound,Microsoft.PowerShell.Commands.NewObjectCommand
```

Próbowaliśmy utworzyć za pomocą polecenia `New-Object` ściśle typowaną ogólną kolekcję, ale program zgłosił błąd, ponieważ konsoli PowerShell nie udało się znaleźć złożenia zawierającego ten typ. Rozwiązaniem jest załadowanie odpowiedniego złożenia.

```
PS C:\> Add-Type -AssemblyName System.Windows.Forms
```

Ponadto zamiast używać parametru `-AssemblyName` możemy załadować klasy prosto z pliku DLL przy użyciu parametru `-Path`.

```
PS C:\> Add-Type -Path D:\myApp\program.dll
```

Moduły Windows PowerShella

Moduły w Windows PowerShellu służą do porządkowania i pakowania skryptów i plików z kodem w nadające się do wielokrotnego użytku paczki. Dostępnych jest dużo wbudowanych modułów zawierających polecenia dotyczące prawie wszystkich ról i funkcji Windows Servera. Na przykład istnieje moduł dla menedżera serwerów, Hyper-V, Active Directory oraz IIS.

Aby wyświetlić listę wszystkich modułów zainstalowanych w systemie operacyjnym, należy wykonać polecenie `Get-Module` z parametrami `-ListAvailable`.

```
PS C:\> Get-Module -ListAvailable | Select Name,Version,ModuleType
Name                    Version          ModuleType
----                    -------          ----------
AppLocker               2.0.0.0          Manifest
AssignedAccess          1.0.0.0          Script
BitLocker               1.0.0.0          Manifest
Dism                    2.0              Script
DnsClient               1.0.0.0          Manifest
Hyper-V                 1.1              Binary
(...)
```

Z tego wynika, że istnieje wiele różnych rodzajów modułów PowerShella. Omówiłem je w następnym podrozdziale, w którym pokazuję, jak się tworzy nowe moduły.

Aby móc użyć jakiegokolwiek modułu — nieważne, czy wbudowanego czy zewnętrznego — należy go zaimportować do sesji PowerShella za pomocą polecenia `Import-Module<Nazwa-↪Modułu>`.

```
#Import Hyper-V and AppLocker modules
PS C:\> Import-Module –Name Hyper-V,AppLocker
```

Jeśli importowany moduł znajduje się w domyślnym katalogu modułów konsoli, to jako argument wystarczy wpisać tylko jego nazwę. Ale jeśli importowany jest moduł z innego katalogu, należy podać pełną ścieżkę. Ponadto, jeżeli używasz wersji PowerShell ISE 3.0 lub nowszej i wpiszesz nazwę polecenia znajdującego się w module, to odpowiedni moduł zostanie załadowany automatycznie.

> Aby wyświetlić domyślne ścieżki modułów, należy użyć zmiennej środowiskowej `$env:PSModulePath`.
>
> Można też dodać ścieżkę do własnych modułów: `$env:PSModulePath += "; C:\MyModules"`.
>
> Więcej informacji na temat `PSModulePath` znajduje się na stronie *http://msdn.microsoft.com/en-us/library/dd878326(v=vs.85).aspx*.

Tworzenie modułów Windows PowerShella

W tym podrozdziale przedstawiam opisy i przykłady tworzenia różnych rodzajów modułów.

Moduły skryptowe

Moduł skryptowy to plik PowerShella z rozszerzeniem *psm1* i kod zawierającym funkcje, zmienne oraz aliasy.

Tworzenie modułu rozpoczniemy od napisania dwóch prostych funkcji (jedna będzie sumować, a druga odejmować dwie liczby) i dla każdej z nich utworzymy alias. Następnie zapiszemy skrypt w pliku o rozszerzeniu *psm1*. Nazwa pliku będzie stanowić nazwę modułu przy jego importowaniu.

```
Function Add-Numbers($x,$y)
{
    $x + $y
}

Function Subtract-Numbers($x,$y)
{
    $x - $y
}

New-Alias -Name an -Value Add-Numbers
New-Alias -Name sn -Value Subtract-Numbers

# Eksportuje składowe modułu
Export-ModuleMember -Function * -Alias *
```

Pewnie zauważyłeś na końcu tego kodu nowe polecenie `Export-ModuleMember`. Służy ono do znajdowania typów PowerShella, to znaczy funkcji, aliasów i zmiennych, i ich eksportowania jako składowych modułu podczas importowania tego modułu za pomocą polecenia `Import-Module`.

```
PS C:\> Import-Module D:\myModules\ScriptModule.psm1 -Force
```

Podczas importowania tego modułu zostanie wyświetlone następujące ostrzeżenie:

> ***WARNING: The names of some imported commands from the module 'ScriptModule' include unapproved verbs that might make them less discoverable. To find the commands with unapproved verbs, run the Import-Module command again with the Verbose parameter. For a list of approved verbs, type Get-Verb.***
>
> (*OSTRZEŻENIE: Nazwy niektórych zaimportowanych poleceń z modułu ScriptModule zawierają niezatwierdzone czasowniki, przez które może być trudno je wykryć. Aby znaleźć polecenia zawierające niezatwierdzone czasowniki, ponownie wykonaj polecenie Import-Module z parametrem Verbose. Listę zatwierdzonych czasowników można wyświetlić, wykonując polecenie Get-Verb*).

Ostrzeżenie to zostanie wyświetlone z powodu użycia niestandardowego (niezatwierdzonego) czasownika w nazwie funkcji. Nie ma to wpływu na działanie funkcji, ale dla zapewniania spójności zaleca się używanie standardowych czasowników.

Listę zatwierdzonych czasowników i ich kategorii można wyświetlić za pomocą polecenia `Get-Verb`. Ponadto można wyłączyć pojawianie się tego ostrzeżenia, dodając przełącznik `-Disable ↪NameChecking`.

Załadowany moduł można znaleźć za pomocą polecenia `Get-Module`.

```
PS C:\> Get-Module ScriptModule | fl

Name              : scriptmodule
Path              : D:\scriptmodule.psm1
ModuleType        : Script
Version           : 0.0
NestedModules     : {}
ExportedFunctions : {Add-Numbers, Subtract-Numbers}
ExportedCmdlets   :
ExportedVariables :
ExportedAliases   : {an, sn}
```

Moduły binarne

Moduł binarny to plik DLL zawierający skompilowany kod, np. klas poleceń i dostawców. Bardzo dobrym przykładem tego rodzaju modułów są moduły wbudowane PowerShella.

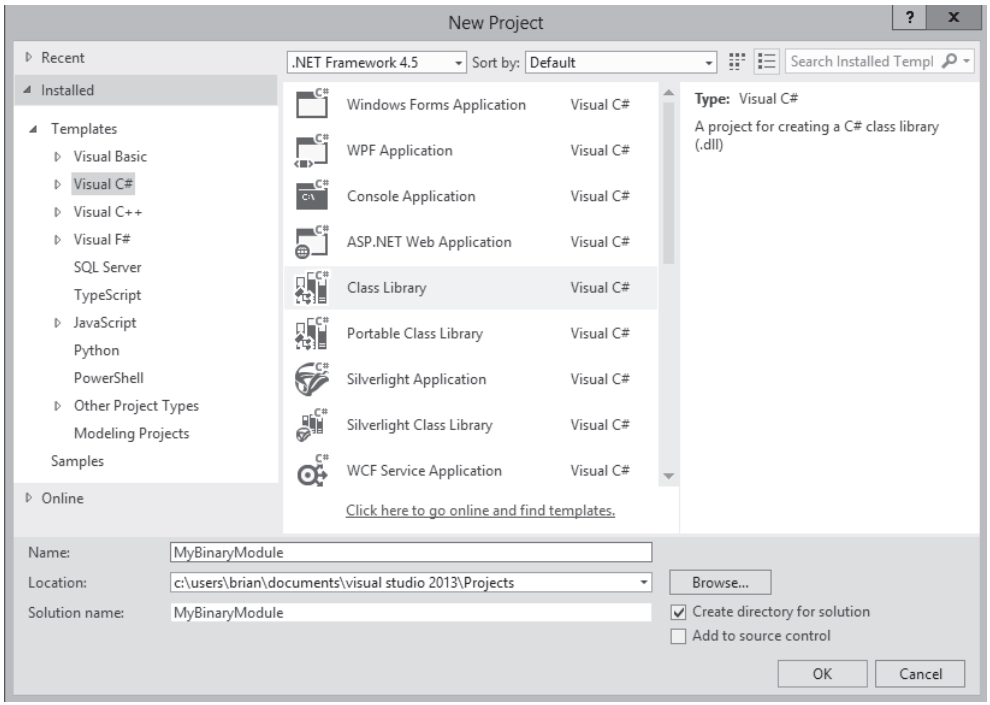
```
PS C:\> Get-Module –Name Microsoft.PowerShell.* | Select Name,NestedModules
```

Tworzenie modułu binarnego

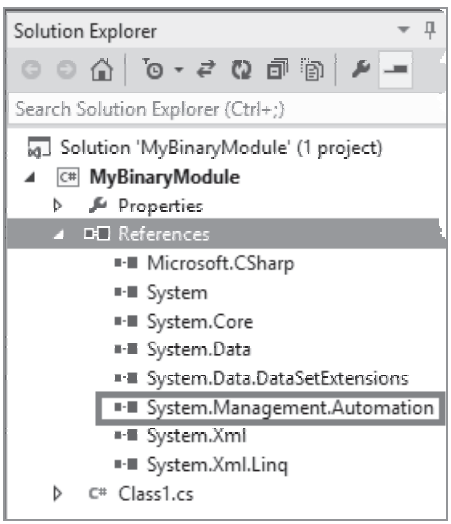
W tym podrozdziale pokażę Ci krok po kroku, jak utworzyć moduł binarny. W odróżnieniu od innych modułów moduły binarne tworzy się w środowisku Microsoft Visual Studio. Utworzymy moduł o nazwie `MyBinaryModule` zawierający dwa polecenia: `Get-EvenOrOdd`, przyjmujące tablicę liczb całkowitych i sprawdzające wartości parzyste i nieparzyste, oraz `Validate-EmailAddress`, przyjmujące łańcuch i sprawdzające, czy ma poprawny format adresu e-mail.

Pracę należy zacząć od utworzenia projektu biblioteki klas w Visual Studio. Nazwa tej biblioteki będzie później nazwą modułu, więc w polu *Name* (nazwa) wpiszemy `MyBinaryModule` (jak na zrzucie ekranu na następnej stronie), chociaż jeśli wolisz coś innego, to nie mam nic przeciwko temu.

Następnie dodajemy odwołanie do głównej przestrzeni nazw Windows PowerShella, czyli `System.Management.Automation`.



Plik DLL `System.Management.Automation` znajduje się w katalogu *C:|Windows|Assembly|GAC_MSIL|System.Management.Automation|1.0.0.0__31bf3856ad364e35*.



Teraz wszystko powinno być gotowe do rozpoczęcia pisania poleceń w języku C#. Aby zaznaczyć, że tworzymy klasę poleceń PowerShella, musimy dodać jej atrybut `[Cmdlet()]`. Atrybut ten zawiera nazwę klasy złożoną z dwóch słów: czasownika i rzeczownika. Ponadto klasa polecenia *cmdlet* powinna być wyprowadzona z klasy bazowej `Cmdlet`, która zawiera trzy metody wirtualne wywoływane przez system wykonawczy: `BeginProcessing()`, `ProcessRecord()` i `EndProcessing()`. W klasie musi być zdefiniowana przynajmniej jedna z tych metod.

```
[Cmdlet(VerbsCommon.Get,"EvenOrOdd")]
    public class EvenorOdd: Cmdlet
    {
protected override void ProcessRecord()
    {
        base.ProcessRecord();
    }
}
```

Ponadto w klasie można zdefiniować parametr za pomocą atrybutu `[Parameter()]`.

```
[Parameter(Position = 0,
           ValueFromPipeline = true,
           ValueFromPipelineByPropertyName = true,
           HelpMessage = @"Zakres liczb do sprawdzenia.")]
        public int[] Numbers
        {
            get { return num; }
            set { num = value; }
        }
        private int[] num;
```

Nad atrybutem `[Parameter()]` można zdefiniować atrybut `[Validate*()]`, służący do sprawdzenia poprawności argumentów tego parametru. Przykładowo można zdefiniować zbiór trzech dopuszczalnych wartości dla parametru `PersonName`:

```
[ValidateSet("Gates", "Jobs", "Ballmer")]
[Parameter(Position = 0, Mandatory = true)]
public string PersonName
{
get { return personName; }
set { personName = value; }
}
private string personName;
```

Można też dodać metody `WriteVerbose()` i `WriteDebug()`, aby podczas wykonywania polecenia umożliwić wyświetlanie danych diagnostycznych za pomocą przełączników `-Debug` i `-Verbose`. Ponadto za pomocą metody `WriteObject()` zwracamy wynik działania polecenia.

> Więcej informacji o głównej przestrzeni nazw PowerShella znajduje się na stronie *http://msdn.microsoft.com/en-us/library/System.Management.Automation(v=vs.85).aspx*.

Ukończony kod źródłowy powinien wyglądać tak, jak na poniższym zrzucie ekranu:

```
// Sprawdza adres e-mail przy użyciu wyrażenia regularnego.
[Cmdlet("Validate", "EmailAddress")]
0 references
public class ValidateEmailAddress : Cmdlet
{
    [Parameter(Position = 0,
       ValueFromPipeline = true,
       ValueFromPipelineByPropertyName = true,
       HelpMessage = @"Adres e-mail do sprawdzenia")]
    3 references
    public string EmailAddress
    {
        get { return Email; }
        set { Email = value; }
    }
    private string Email;

    3 references
    protected override void ProcessRecord()
    {
        base.ProcessRecord();

        // Wyświetla rozszerzone informacje przy użyciu parametru -Verbose.
        WriteVerbose("Validating Email Address: " + EmailAddress);

        // Wyświetla rozszerzone informacje przy użyciu parametru -Debug.
        WriteDebug("Validating Email Address: " + EmailAddress);

        Regex reg = new Regex(@"(\w+@[a-zA-Z_-]+?\.[a-zA-Z_-]{2,6})");

        // Zapisuje informacje obiektu do wyświetlenia jako wynik wykonania polecenia.
        WriteObject(reg.IsMatch(EmailAddress));
```

Teraz trzeba skompilować projekt do postaci binarnego modułu. W Visual Studio należy w tym celu kliknąć polecenie *Build-Build Solution* (kompilacja-kompiluj rozwiązanie). Po zakończeniu kompilacji w podfolderze projektu *bin/debug* pojawi się plik o nazwie *MyBinaryModule.dll*.

Gratulacje, właśnie utworzyłeś pierwszy binarny moduł. Przejdź do konsoli PowerShell i zaimportuj go za pomocą polecenia `Import-Module`.

```
PS C:\> Import-Module "D:\MyBinaryModule\MyBinaryModule.dll"

PS C:\> Get-Command -Module MyBinaryModule | Select CommandType, Name

CommandType  Name
-----------  ----
Cmdlet       Get-EvenOrOdd
Cmdlet       Validate-EmailAddress
# Użycie polecenia Validate-EmailAddress
PS C:\ > Validate-EmailAddress -EmailAddress sherif@xyz -Verbose
VERBOSE: Validating Email Address: sherif@xyz
False
```

```
PS C:\ > Validate-EmailAddress -EmailAddress sherif@xyz.com
True

# Użycie polecenia Get-EvenOrOdd
PS C:\Users\v-shta> Get-EvenOrOdd -Numbers @(2,5,13,17,24,33)
2 to liczba parzysta
5 to liczba nieparzysta
13 to liczba nieparzysta
(...)
```

Moduły z manifestem

Moduł z manifestem to moduł zawierający plik z danymi PowerShella, `-manifest-` (*.psd1*), opisującymi składniki i zawartość oraz sposób przetwarzania modułu. Plik modułu z manifestem może zawierać jeden lub więcej zagnieżdżonych modułów skryptowych lub binarnych.

Manifest jest plikiem tekstowym zawierającym informacje o module, np. kto go utworzył, w jakiej firmie powstał, ogólny opis działania, jakie pliki trzeba dołączyć, jakie złożenia trzeba załadować, najstarsza obsługiwana wersja PowerShella oraz najstarsza obsługiwana wersja platformy .NET. W większości przypadków plik manifestu jest niepotrzebny, chyba że chce się wyeksportować złożenie zainstalowane w globalnym buforze złożeń, użyć funkcji pomocy z możliwością aktualizacji lub ustawić pewne ograniczenia.

Aby utworzyć manifest dla modułu, należy za pomocą polecenia `New-ModuleManifest` utworzyć pusty szablon manifestu, który następnie można otworzyć i zmodyfikować w dowolnym edytorze tekstu. Ponadto dane manifestu można zdefiniować podczas tworzenia szablonu. W tym celu należy użyć parametrów polecenia `New-ModuleManifest`.

```
New-ModuleManifest -Author "Sherif Talaat" -CompanyName "Packt Publishing" -
↪ModuleVersion "1.0" -ProcessorArchitecture Amd64 -PowerShellVersion "3.0" -
↪PowerShellHostName "ConsoleHost, Windows PowerShell ISE Host" -Description
↪"Mój pierwszy manifest modułu" -FileList "myScriptModule.psm1" -ModuletoProcess
↪"Bitlocker" -Path "D:\Modules\myScriptModule\myScriptModule.psd1"
```

Na następnej stronie zrzut ekranu przedstawia przykładową zawartość pliku manifestu.

Moduły dynamiczne

Moduł dynamiczny to moduł, który nie jest przechowywany na dysku twardym, lecz w pamięci, i zostaje usunięty po zakończeniu sesji działania konsoli. Tego rodzaju moduły można tworzyć z funkcji i bloków skryptowych w sesji. Są one przydatne programistom posługującym się technikami obiektowymi oraz administratorom, którzy chcą wykonywać wybrane moduły na zdalnych komputerach przy użyciu narzędzi PowerShella do pracy zdalnej.

```
# Module manifest for module 'myModule'
# Generated by: Sherif Talaat
# Generated on: 10/22/2013
#

@{

# Script module or binary module file associated with this manifest.
# RootModule = ''

# Version number of this module.
ModuleVersion = '1.0'

# ID used to uniquely identify this module
GUID = 'b9fbbfb6-04aa-4c58-aa52-f0ab9c7db83e'

# Author of this module
Author = 'Sherif Talaat'

# Company or vendor of this module
CompanyName = 'Innovation-Hut'

# Copyright statement for this module
Copyright = '(c) 2013 Sherif Talaat. All rights reserved.'

# Description of the functionality provided by this module
Description = 'my first module manifest'

# Minimum version of the Windows PowerShell engine required by this module
 PowerShellVersion = '3.0'

# Name of the Windows PowerShell host required by this module
 PowerShellHostName = 'ConsoleHost, Windows PowerShell ISE Host'

# Minimum version of the Windows PowerShell host required by this module
 PowerShellHostVersion = '3.0'
```

Do tworzenia modułów dynamicznych używa się polecenia `New-Module` z parametrami `-Function` i `-ScriptBlock`. Za pomocą tych parametrów określa się funkcje i bloki skryptowe, które mają się znaleźć w danym module.

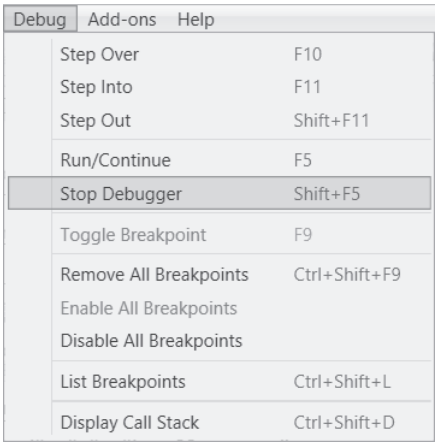
```
# Tworzy dynamiczny moduł z jedną funkcją
PS C:\> New-Module -ScriptBlock {Function Send-Greetings($name){"Dzień dobry,
$name"}}

# Uruchamia funkcję
PS C:\> Send-Greetings –name Sherif
Dzień dobry, Sherif
```

Diagnostyka skryptów i obsługa błędów

W poprzednim rozdziale napisałem, że w PowerShellu można diagnozować zarówno lokalne, jak i zdalne skrypty. Funkcja diagnostyczna w PowerShellu działa podobnie jak w innych językach programowania. Można ustawiać punkty wstrzymania, wykonywać funkcje krok po kroku, przeskakiwać instrukcje, a nawet wywoływać stos. W konsoli PowerShella funkcje diagnostyczne obsługuje się za pomocą poleceń, a w PowerShell ISE — za pomocą graficznego interfejsu użytkownika i poleceń.

Narzędzia do diagnostyki w PowerShell ISE znajdują się w menu *Debug* (diagnostyka), którego zawartość widać na poniższym zrzucie:



Ponadto konsola Windows PowerShell zawiera zestaw poleceń, za pomocą których można przeprowadzić diagnostykę skryptu bez używania graficznego interfejsu użytkownika. Polecenia te są bardzo przydatne, gdy używa się systemu Windows Server Core, w którym brak PowerShell ISE. Wszystkie te polecenia służą do zarządzania punktami wstrzymania w skryptach.

```
PS C:\ > Get-Command -Name *Breakpoint | Select Name
Name
----
Disable-PSBreakpoint
Enable-PSBreakpoint
Get-PSBreakpoint
Remove-PSBreakpoint
Set-PSBreakpoint
```

Oprócz poleceń `PSBreakpoint` można używać jeszcze paru dodatkowych poleceń, które są dostępne wyłącznie w trybie diagnostycznym.

Punkty wstrzymania

Punkt wstrzymania jest wyznaczonym w kodzie źródłowym miejscem, w którym program powinien wstrzymać działanie i włączyć diagnostykę. W Windows PowerShellu dostępne są trzy rodzaje punktów wstrzymania, które można włączać i wyłączać za pomocą polecenia `Set-PSBreakpoint`:

- **Punkt wstrzymania na wybranym wierszu —** wykonywanie skryptu zatrzymuje się na wyznaczonym wierszu kodu. Punkty wstrzymania tego typu definiuje się przez podanie numeru wiersza i przy użyciu przełącznika `-Line`.

```
PS C:\> Set-PSBreakpoint –script c:\myscript.ps1 –Line 7
```

- **Punkt wstrzymania na zmiennej —** wykonywanie skryptu zatrzymuje się po zmianie wartości określonej zmiennej. Punkty wstrzymania tego typu definiuje się przez podanie nazwy zmiennej bez znaku `$` i przy użyciu przełącznika `-Variable`.

```
PS C:\> Set-PSBreakpoint –script c:\myscript.ps1 –Variable
Services
```

- **Punkt wstrzymania na poleceniu —** wykonywanie skryptu zatrzymuje się przed rozpoczęciem wykonywania określonego polecenia. Poleceniem może być polecenie *cmdlet* lub nazwa utworzonej przez programistę funkcji. Punkty wstrzymania tego typu definiuje się przez podanie nazwy polecenia i przy użyciu przełącznika `-Command`.

```
PS C:\> Set-PSBreakpoint –script c:\myscript.ps1 –Command GetProcess
```

Zdefiniowaliśmy w naszym skrypcie trzy punkty wstrzymania, każdy innego typu. Wszystkie punkty wstrzymania ustawione w skrypcie można wyświetlić za pomocą polecenia `Get-PSBreakpoint`.

```
PS C:\> Get-PSBreakpoint –Script myscript.ps1
ID Script               Line Command       Variable
  -- ------               ---- -------       --------
11 myscript.ps1 7
12 myscript.ps1 Services
13 myscript.ps1 Get-Process
```

Do usuwania punktów wstrzymania służy polecenie `Remove-PSBreakpoint`, ale można też je tylko czasowo wyłączać przy użyciu polecenia `Disable-PSBreakpoint`. Czasowo wyłączony punkt wstrzymania można z powrotem włączyć za pomocą polecenia `Enable-PSBreakpoint`.

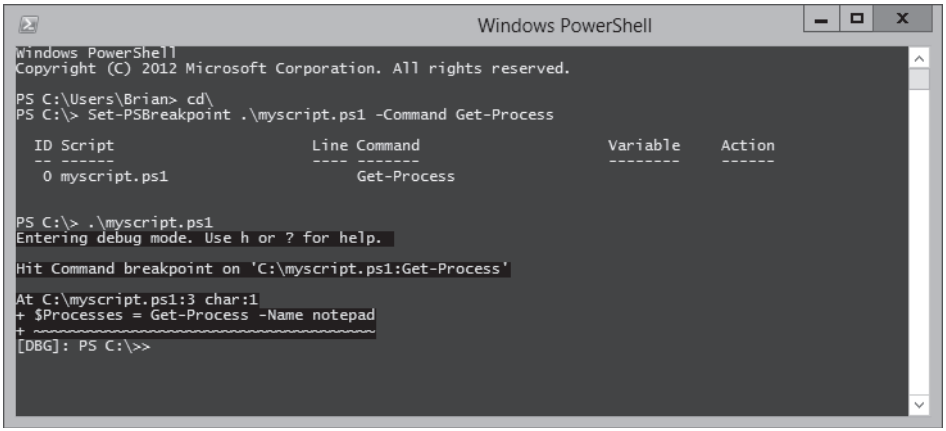
```
# Wyłącza punkty wstrzymania na zmiennej
Get-PSBreakpoint -Variable Services | Disable-PSBreakpoint

# Włącza punkty wstrzymania na zmiennej
Get-PSBreakpoint -Variable Services | Enable-PSBreakpoint

# Usuwa punkty wstrzymania na zmiennej
Get-PSBreakpoint -Variable Services | Remove-PSBreakpoint
```

Diagnozowanie skryptów

Po zdefiniowaniu punktów wstrzymania skrypt można uruchomić w normalny sposób. Gdy program dojdzie do pierwszego punktu, wyświetli stosowną informację. Od tej pory na początku wiersza poleceń, przed napisem `PS C:\>>`, będzie się znajdował znacznik `[DBG]:` informujący, że aktywny jest tryb diagnostyczny. Tryb ten pozostanie włączony, aż wyłączy się debuger za pomocą klawiszy *Shift+F5*.



W trybie diagnostycznym można prowadzić normalną diagnostykę programu za pomocą poleceń wymienionych w poniższej tabeli:

Czynność	Polecenie	Skrót
Wkroczenie	`StepInto`	*S*
Wyjście	`StepOut`	*O*
Pominięcie	`StepOver`	*V*
Kontynuacja	`Continue`	*C*
Wyświetlenie listy	`List`	*L*
Zatrzymanie	`Quit`	*Q*
Pobranie stosu wywołań	`Get-PSCallStack`	*K*

Kiedyś w Windows PowerShellu można było diagnozować tylko skrypty uruchomione lokalnie. Próby ustawienia punktów wstrzymania w zdalnej sesji kończyły się błędem. Ale w Windows PowerShellu 4.0 zmieniono to i można już ustawiać punkty wstrzymania w zdalnych sesjach, a także diagnozować zdalne skrypty w taki sam sposób jak lokalne. Aby można było korzystać z możliwości zdalnego diagnozowania skryptów, zarówno na lokalnym, jak i na zdalnym komputerze musi być zainstalowane narzędzie Windows PowerShell 4.0.

Techniki obsługi błędów

W Windows PowerShellu do obsługi błędów terminalnych (wyjątków), podobnie jak w języku C#, używa się instrukcji `Try{}`, `Catch{}` oraz `Finally{}`. Błędy terminalne są obsługiwane przez instrukcję `Catch{}` tylko wtedy, gdy zmieni się wartość zmiennej `$ErrorActionPreference` na `stop`.

```
$ErrorActionPreference = "stop"

Try
{
    Get-ChildItem C:\movies
}
Catch [System.Exception]
{
    "Nie znaleziono obiektu."
}
Finally
{
    New-item -ItemType Directory -Path C:\Movies
    "Obiekt został utworzony."
}
```

Więcej informacji na temat instrukcji `Try`, `Catch` i `Finally` można znaleźć w pomocy *About* w następujących tematach:

- *About_Trap*
- *About_Throw*
- *About_Try_Catch_Finally*

Zmienne $Error i $LastExitCode

Gdy podczas działania PowerShella wystąpi błąd, zostaje on zapisany w globalnej zmiennej `$Error`. Jest to egzemplarz klasy `ArrayList`, który zawiera obiekty błędów PowerShella i w którym ostatni błąd jest zapisany pod indeksem o numerze zero. Ze zmiennej tej można wyciągnąć różne szczegółowe informacje na temat zaistniałych błędów, jak zostało pokazane poniżej:

```
PS D:\> $Error[0].Exception
Cannot find path 'C:\movies' because it does not exist.

PS D:\> $Error[0].FullyQualifiedErrorId
PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

PS D:\> $Error[0].ScriptStackTrace
at <ScriptBlock>, <No file>: line 5
```

Więcej informacji na temat rejestracji błędów można znaleźć na stronie *http://msdn.microsoft.com/en-us/library/system.management.automation.errorrecord_members(v=vs.85).aspx*.

Na podstawie kodu zakończenia określa się status wykonawczy macierzystych aplikacji, takich jak *ping.exe* czy *robocopy.exe*, to znaczy czy ich wykonywanie zakończyło się pomyślnie, czy nie. Najczęściej wartość 0 oznacza powodzenie, a 1 niepowodzenie, ale niektóre aplikacje mogą

zwracać bardziej zróżnicowane kody oznaczające różne rodzaje błędów. W PowerShellu kod zakończenia działania aplikacji macierzystych i procesów zewnętrznych jest zapisywany w zmiennej `$LastExitCode`.

Tworzenie graficznego interfejsu użytkownika w PowerShellu

Powiedzieliśmy już bardzo dużo na temat różnych zastosowań narzędzia Windows PowerShell i jego powiązań z platformą .NET. Na zakończenie tego rozdziału powiemy sobie jeszcze o technikach tworzenia graficznego interfejsu użytkownika w PowerShellu przy użyciu udogodnień platformy .NET.

Poniższy kod demonstruje sposób użycia normalnej przestrzeni nazw platformy .NET do utworzenia prostego formularza WPF z jednym przyciskiem:

```
$form = new-object Windows.Forms.Form
$form.Text = "Main Form"
$button = new-object Windows.Forms.Button
$button.text="Zamknij mnie!"
$button.Dock="fill"
$button.add_click({$form.close()})
$form.controls.add($button)
$form.Add_Shown({$form.Activate()})
$form.ShowDialog()
```

Efekt wykonania powyższego kodu w konsoli został przedstawiony na poniższym zrzucie ekranu. Ładnie, prawda?

Wprawdzie można w ten sposób utworzyć dowolny formularz z jakimi się chce kontrolkami, ale i tak trzeba w tym celu napisać kilkaset wierszy kodu. Dlatego lepiej jest użyć jakiegoś dodatkowego narzędzia z wizualnym projektantem, np. SAPIEN PowerShell Studio.

Podsumowanie

Dzięki lekturze tego rozdziału nauczyłeś się pracować z technologiami WMI, COM oraz XML w konsoli PowerShell. Dowiedziałeś się też, czym jest CIM i jak się nim posługiwać poprzez PowerShella. Ponadto pokazałem, jak pracować z obiektami .NET i jak rozszerzać możliwości konsoli PowerShell za pomocą narzędzi platformy .NET.

Dodatkowo nauczyłeś się tworzyć różne rodzaje modułów i wykorzystywać je do automatyzacji własnych programów. Na końcu przejrzeliśmy techniki diagnozowania skryptów i obsługi błędów w PowerShellu.

W następnym rozdziale nauczysz się wykorzystywać konsolę PowerShell do codziennej administracji, na przykład do przygotowywania wymagań dla programów, definiowania uprawnień użytkowników i grup, zarządzania serwerem IIS i jego konfigurowania oraz zarządzania bazami danych SQL Servera.

Skorowidz

J

K

L, Ł

M

N

O

P

R

S, Ś

T

U

V

W

X

Z, Ż

Windows PowerShell 4.0 dla programistów .NET

PowerShell to — jak sama nazwa wskazuje — konsola z ogromem możliwości! Pozwala kontrolować system Windows oraz wiele innych aplikacji przeznaczonych dla serwerów, a także nimi zarządzać. Dzięki swoim atutom jest szczególnie doceniana przez zaawansowanych użytkowników systemu Windows oraz administratorów tej platformy. Jeżeli chcesz w pełni wykorzystać potencjał PowerShell, jesteś programistą platformy .NET i chciałbyś ułatwić sobie pracę, to trafiłeś na superksiążkę!

Sięgnij po nią i poznaj podstawy Windows PowerShell! Gdy zaczniesz już swobodnie korzystać z nowych możliwości, będziesz mógł poznać najlepsze techniki pracy z plikami XML i modułami oraz zaznajomić się z obiektami COM i .NET. Następnie dowiesz się, jak administrować systemem Windows z wykorzystaniem możliwości PowerShell. Dzięki tej wiedzy większość zadań wykonasz zdecydowanie szybciej — i to bez myszki! Na sam koniec nauczysz się korzystać z zasobów sieciowych oraz narzędzi dla platformy TFS. Książka ta jest obowiązkową pozycją dla wszystkich użytkowników systemu Windows, chcących poznać zaawansowane możliwości zarządzania systemem… i nie tylko!

Administrowanie systemem Windows jeszcze nigdy nie było tak przyjemne!

Dzięki tej książce:

- poznasz techniki pracy z Windows PowerShell
- wykorzystasz obiekty COM i .NET
- zdiagnozujesz błędy w Twoich skryptach
- skorzystasz z usług sieciowych

[PACKT] PUBLISHING enterprise professional expertise distilled

Helion

28911 numer katalogowy
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-283-0327-0

Informatyka w najlepszym wydaniu

cena: 32,90 zł