POWERSHELL DLA PROGRAMISTÓW (NIE TYLKO .NET)

PowerShell to interpreter poleceń oraz powłoka skryptowa firmy Microsoft. Została ona opracowana jako następca command.com i cmd.exe znana z systemów takich jak MS DOS czy ze starszych wersji Windowsa. PowerShell funkcjonuje od 2006 roku. Można go było doinstalować do Windowsa XP oraz do Windowsa Visty jako dodatkowe rozszerzenie. W Windowsie 7 i w nowszych wersjach jest to już domyślny składnik systemu (oprócz Windows Server 2008, gdzie PowerShell jest również nie preinsalowanym dodatkiem).

Początkowo PowerShell znany był jako Microsoft Shell (w skrócie MSH) oraz pod nazwą kodową Monad. Wraz z wyjściem produktu na rynek ustalono nazwę obowiązującą do dzisiaj. Oczywiście PowerShell nie jest niczym innowacyjnym. Za to dodaje do systemów z Redmont pewną ważną dla wielu funkcję, której brakowało w poprzednich wersjach okienek – pełna zarządzalność systemem z poziomu tekstowego. Akurat w przypadku PowerShella nie jest to tylko zwykły tryb tekstowy, ale o tym za chwilę. Pamiętać należy również, że już wcześniej Microsoft oferował możliwość zarządzania systemem z poziomu linii komend i z poziomu skryptów napisanych w języku MS VisualBasic. Mowa tutaj o narzędziu host skryptów, które dostępne było w Okienkach od wersji 98.

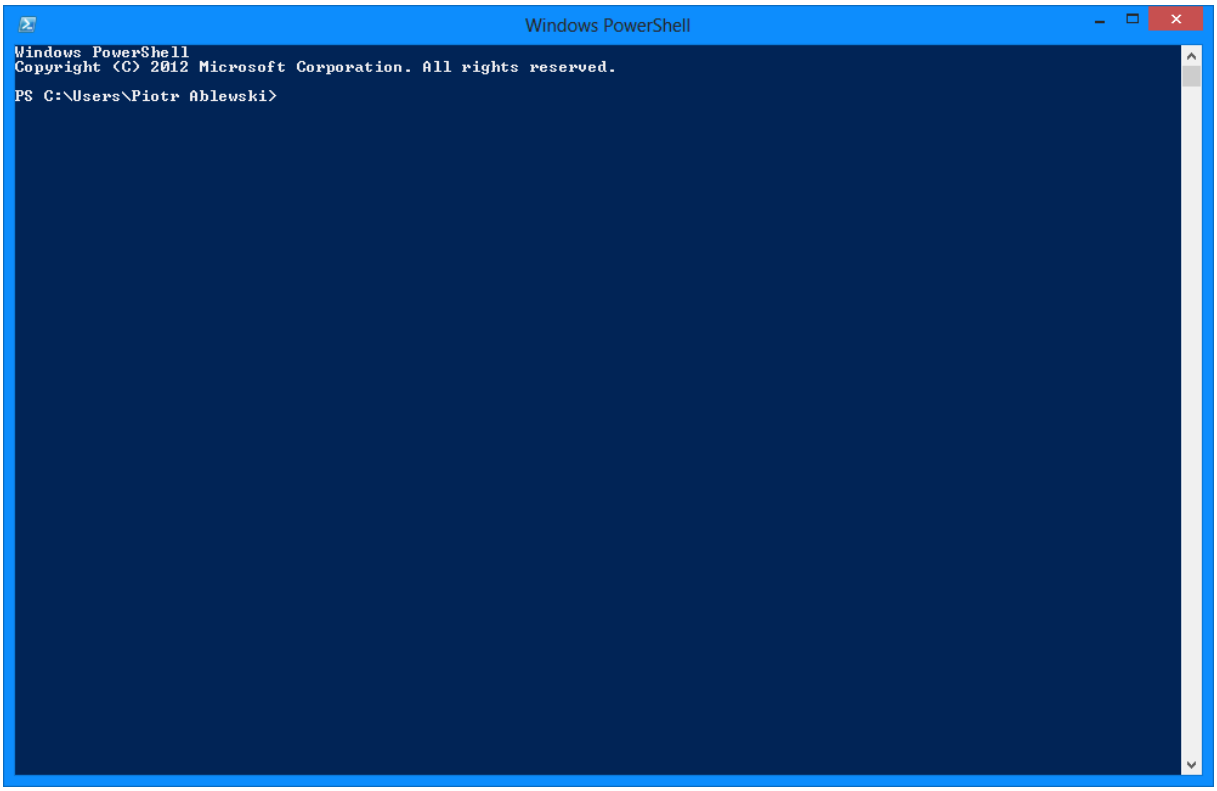
PowerShell stał się narzędziem, które w znaczny sposób ułatwiło administratorom sieciowym pracę oraz spowodowało, że mogą o wiele szybciej wykonać codzienne czynności związane z utrzymaniem systemu. PowerShell dostarcza bardzo prosty składniowo język skryptowy, którego większość zapytań i wyników jest obiektami .NET oraz działaniami na obiektach .NET. Co ciekawe – cały PowerShell oparty jest na platformie .NET.

Oczywiście PowerShell to narzędzie nie tylko dla tych, którym bliżej jest do IT, ale również dla programistów. Dzięki niemu można w znaczny sposób zautomatyzować swoją pracę oraz ułatwić sobie analizę działania stworzonego przez siebie kodu. Z poziomu PowerShella można kontrolować usługi, procesy jak również uruchamiać zewnętrzne programy. Jeśli pojawia się jakaś potrzeba – w większości przypadków PowerShell udostępnia już gotową komendę z odpowiednimi modyfikatorami. Jeśli tak nie jest możemy zawsze stworzyć swój własny skrypt wykonujący odpowiednie działania.

Co ciekawe programistom bardzo spodoba się możliwość pisania programów, które wykorzystują obiekty .NET, COM i WMI. Programy te są na bieżąco interpretowane. Dostępność obiektów znanych programistom daje potężne możliwości – nie trzeba być doświadczonym administratorem

systemowym, aby poradzić sobie z opanowaniem PowerShella. Dodatkowo skrypty są bardzo proste w konstrukcji. Jest to niesamowity krok do przodu w stosunku do skryptów pisanych w VisualBasic'u i wykonywanych poprzez WSH (Windows Script Host). Na temat konstrukcji językowych można przeczytać w dalszej części tutorialu.

INTERFEJS UŻYTKOWNIKA



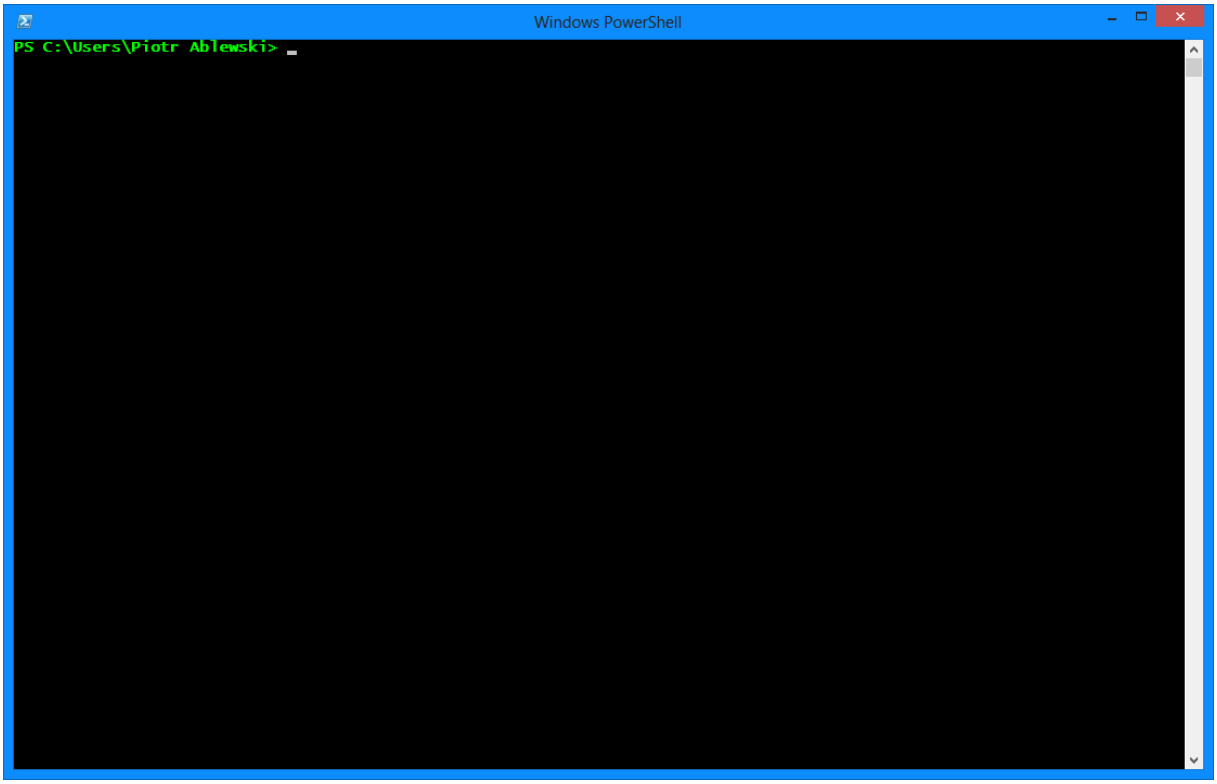
Rys. 1 – Interfejs Użytkownika

Jak widać interfejs użytkownika nie wiele różni się od interfejsu typowej linii komend systemu Windows. Różnica tkwi tak naprawdę w szczegółach. I to szczegóły tworzą z PowerShella potężne narzędzie pracy. Chodzi tutaj między innymi o budowę komend, która jest obecnie bardziej jasna (mimo, że początkowo wydaje się o wiele bardziej złożona) oraz kilka udogodnień, których od dawna brakowało linii komend.

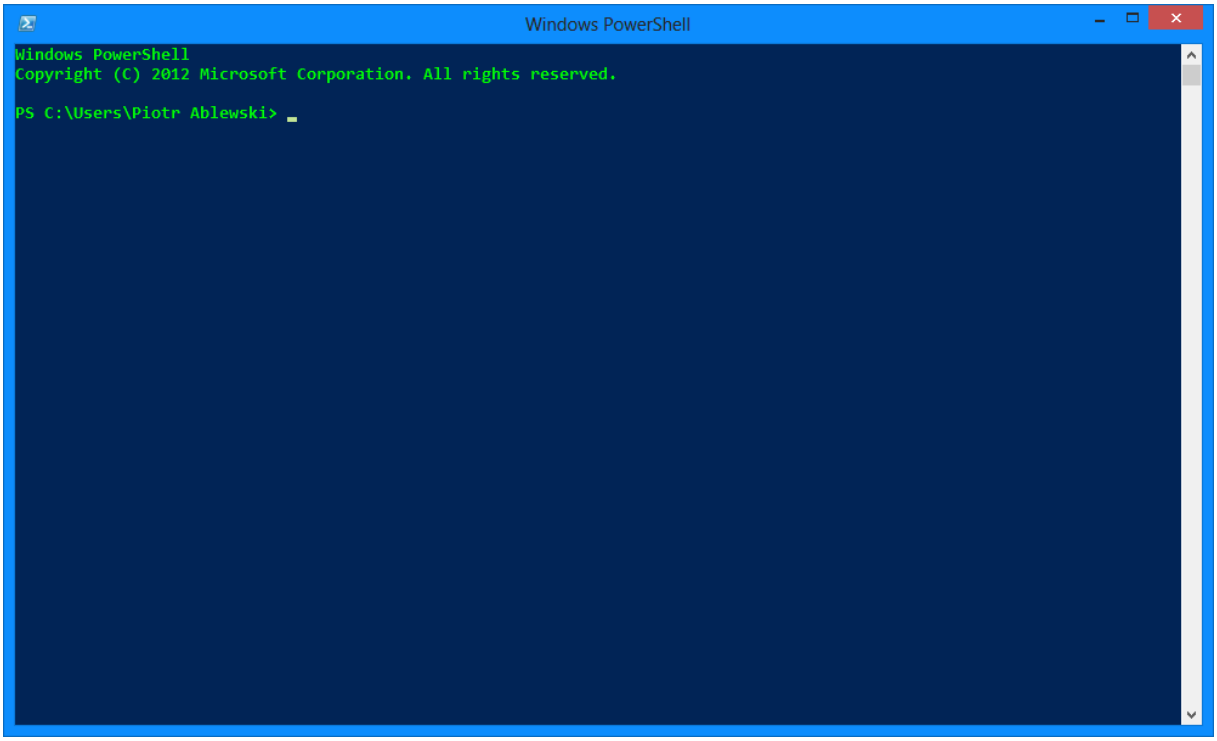
Jak widać na Rys. 1. Zarówno wygląd jak i „znak zachęty", czyli popularny command prompt nie zmieniły się od czasu MS DOS. Wygląd można modyfikować w zależności od swoich upodobań. Korzystając z wersji „out of box" możemy wybrać pomiędzy czcionkami rastrowymi a nieśmiertelnym i niesamowicie czytelnym Consolas'em czy czcionką Lucida Console. Dodatkowo mamy możliwość wybrania rozmiaru fontu, jego koloru i koloru tła. Kolor można wybrać z gotowej palety systemowej jak również podać kolor z palety RGB.

Oczywiście wygląd PowerShella można znacznie bardziej zmodyfikować. W Internecie dostępna jest potężna ilość programów, które umożliwią użytkownikowi dostosowanie wyglądu do własnych potrzeb – począwszy od zmiany tła i czcionki, po modyfikacje takie jak przezroczystość okna. Dzięki nim PowerShell będzie bardziej przypominał Terminal znany użytkownikom systemów Unixowych.

Dodatkowo można zmodyfikować znak zachęty poprzez zmianę zmiennej systemowej PROMPT. Jej wartość można zmienić na taką, jaka nam odpowiada jako znak zachęty naszego systemu operacyjnego, np.:



Rys. 2. – zmodyfikowane kolory PowerShell'a



Rys. 3. – spersonalizowany PowerShell na moim laptopie

KOMENDY W POWERSHELL, CZYLI CMDLET'Y

Komendy w PowerShell'u nazywają się cmdlet (poprawna wymowa command-let). W większości przypadków składają się one z dwóch części – czasownikowej, która informuje o tym jaka operacja będzie wykonywana oraz rzeczownikowej, która określa jaki typ operacji zostanie wykonany. Obie części oddzielone są od siebie znakiem - .

Oczywiście można ulec wrażeniu, że taki sposób wcale nie ułatwia pracy, a nawet ją utrudnia, gdyż trzeba zapamiętać znacznie dłuższe komendy. Otóż nic bardziej mylnego – dzięki takiej konstrukcji, już po kiludziesięciu minutach pracy z PowerShellem bez problemu można odnaleźć potrzebną nam komendę. Dodatkowo, dla osób, których t tłumaczenie nie przekonuje, można zastosować aliasy. Na przykład świetnie znane polecenie help, które znane jest wszystkim użytkownikom DOS'a czy polecenie man, z którego korzystają Linux'owcy to alias dla polecenia Get-Help. Co ciekawe – oba aliasy działają – zarówno ten, który znany jest z DOS', jak i ten Linux'owy. Jest to ukłon w stronę użytkowników, którzy nie muszą zmieniać stosowanej nomenklatury przy zmianie systemu operacyjnego.

Taka budowa poleceń ułatwia również korzytsanie z pomocy, gdyż wiedząc jaką operację chce się wykonać, można w prosty sposób odnaleźć zestawienie, które zwróci oczekiwany przez nas wynik. Dodatkowo, dzięki niej, zapamiętując kilka słów można złożyć zestaw ponad 200 cmdlet'ów, które posłużą do kontroli naszego komputera.

Dodatkowo warto wiedzieć, że wydawane przez nas polecenia nie wywołują zewnętrznych programów, tak jak miało to miejsce w przypadku Linii Komend, tylko są one kompilowane już przez PowerShell'a. Jaki ma to efekt dla użytkownika? Dzięki temu wyniki działania komend to nie tylko czysty tekst, tak jak miało to miejsce wcześniej. Wynik zapytania, które zostało zinterpretowane przez PS to obiekt, pełnoprawny obiekt znany z współcześnie stosowanych języków programowania, który posiada swoje metody i własności. Właśnie dzięki temu, wynik działania jednej komendy może stać się obiektem wejściowym (albo mówiąc bardziej poprawnie argumentem wejściowym) kolejnej. Oczywiście sposób reprezentacji wyników, to również tekst, tak jak miało to miejsce wcześniej, ale zwracane przez cmdlet'y wyniki to bardziej zaawansowane struktury.

Dzięki temu, że wynikiem działania polecenia nie jest tylko tekst, można tworzyć

PODSTAWOWE CMDLET'Y

Skupmy się najpierw na kilku najważniejszych cmdlet'ach (warto pamiętać o tej nomenklaturze), które w znacznym stopniu ułatwią nam oswajanie się z PowerShellem.

GET-HELP

Najważniejszą komendą dla każdego administratora czy użytkownika systemu jest zazwyczaj pomoc – to dzięki niej można w miarę szybko odnaleźć się w nowej sytuacji. W rpzypadku PowerShell pomoc wywoła komenda **Get-Help** (czyli zgodnie z wspomnianą wcześniej budową – GET – uzyskaj – Help – pomoc). Manual w PowerShell umożliwia uzyskanie czterech głównych zakresów informacji. W zależności od tego, jak zostanie wywołana komenda Get-Help, można uzyskać różne informacje.

Wykorzystanie:

```
Get-Help Komenda
```

Spowoduje, że uzyskamy informacje na temat danej komendy takie jak jej nazwę, składnię, aktualnie zadeklarowane aliasy i dodatkowe uwagi.

Aby zobaczyć przykłady użycia danej komendy należy wykorzystać opcję :

```
Get-Help Komenda –Examples
```

Dodatkowo będziemy mogli wtedy przejrzeć opis komendy.

Aby uzyskać szczegółowy opis polecenia, zilustrowany przykładami pomocna będzie opcja **–Detailed**:

```
Get-Help Komenda –Detailed
```

Aby przejrzeć pełną dokumentację techniczną zawierającą opis komendy, parametry z opisami i przykłady zastosowania, należy użyć opcji **–Full**:

```
Get-Help Komenda –Full
```

Jeżeli żadna z zaprezentowanych powyżej opcji nie rozwiała naszych wątpliwości i nie przekazała informacji, której poszukiwaliśmy, zawsze możemy poszukać informacji na temat danego cmdlet'a w Internecie, prosto z poziomu PowerShell, wykorzystując opcję **–Online**:

```
Get-Help Komenda –Online
```

GET-COMMAND

Kolejną ważną dla początkujących komendą będzie cmdlet **Get-Command**. Pozwoli ona na znalezienie cmdlet'a, którego składni dokładnie nie pamiętamy.

Jednak w pierwszej kolejności skupmy się na samej komendzie **Get-Command**. Wywołanie:

```
Get-Command
```

Powoduje, że wyświetlona zostaje lista wszystkich cmdlet'ów zainstalowanych na komputerze oraz ich aliasów, funkcji, filtrów, skryptów i dostępnych aplikacji. Polecenie to daje wyniki dotyczące wszystkich sesji. Jeżeli chcemy ograniczyć się do bieżącej sesji przydatny okaże się parametr **-ListImported**:

```
Get-Command –ListImported
```

Jeśli wykonamy polecenie:

```
Get-Command *
```

Otrzymamy listę wszystkich typów komend, wliczając w to programy, które nie są integralną częścią PowerShell'a, a zawarte są w katalogach zapisanych w zmiennej systemowej **$PATH**.

Można zadać sobie pytanie: jaka jest różnica pomiędzy poleceniami **Get-Help** a **Get-Command**. W pierwszy przypadku, wynik jest brany z manuala do danego polecenia, a w przypadku drugim – prosto z kodu komendy.

Jakie inne parametry mogą być jeszcze przydatne? Oto ich lista z wyjaśnieniem użycia. Oczywiście dokładny opis wszystkich opcji można uzyskać korzystając z pomocy PowerShell'a, a więc wywołując cmdlet'a Get-Help.

Parametry charakterystyczne dla cmdlet'a Get-Command:

GET-COMMAND –ALL

Zwraca listę wszystkich komend, włączając w to komendy tego samego typu, które mają te same nazwy (są w innych przestrzeniach nazw).

```
Get-Command –ArgumENtList argument
```

Zwaca listę wszystkich komend, które można wywołać z podanym parametrem

```
Get-Command –CommandType typ
```

Zwraca tylko te komendy, które należą do danego typu komend. Dostępne typy to:

- **Alias** – alias do polecenia w PowerShell'u
- **All** – komendy wszystkich typów
- **Applications** – pliki, które nie są częścią PowerShell'a, ale zawarte są w ścieżkach zadeklarowanych w zmiennej systemowej $PATH (wliczając w to pliki tekstowe, wykonywalne jak i biblioteki DLL)
- **Cmdlet** – komendy będące tylko cmdlet'ami (czyli integralną częścią PowerShell'a)
- **ExternalScript** – pliki zawarte w folderach zadeklarowanych w zmiennej $PATH zawierające skrypty PowerShell'a (czyli pliki o rozszerzeniu *.ps1)
- **Filter** – wszystkie filtry PowerShell'a
- **Function** – wszystkie funkcje PowerShell'a
- **Script** – wszystkie bloki skryptów (za wyjątkiem oddzielnych plików, które zwróci typ ExternalScript
- **Workflow** – wszystkie workflow'y dostępne w PowerShell'u

GET-COMMAND –MODULE NAZWA

Wyświetla informacje o komendach, które pochodzą z określonego modułu lub dodatku.

```
Get-Command –Module nazwa_modułu
```

GET-COMMAND –NAME WZÓR

Wyświetla tylko te komendy, które zawierają w sobie konkretną nazwę lub wzór nazwy, w którym można stosować wieloznaczniki. Aby dodatkowo uzyskać listę wszystkich poleceń o podanej nazwie zaleca się stosowanie opcji **–All** z opcja **–Name**

```
Get-Command –all –name wzór
```

GET-COMMAND –NOUN WZÓR

Podaje wszystkie komendy, których nazwy zawierają podany rzeczownik w swojej nazwie. Jako argument można podać klika słów, również stosowanie masek jest dozwolone.

```
Get-Command –Noun rzeczownik
```

GET-COMMAND -SYNTAX

Podaje tylko określone informacje na temat poszukiwanej komendy:

- Dla aliasów – podaje nazwę komendy, którą uruchamia alias
- Dla cmdlet'ów – podaje składnię polecenia
- Dla funkcji i filtrów – wyświetla ich definicję
- Dla skryptów i aplikacji – wyświetla ścieżkę i nazwę pliku, który zostanie uruchomiony

```
Get-Command –Syntax wzór
```

GET-COMMAND -TOTALCOUNT

Zwraca określoną ilość komend. Parametr ten jest wykorzystywany, aby ograniczyć ilość zwróconych komend:

```
Get-Command –TotalCount ilość [dalsze parametry]
```

GET-COMMAND -VERB

Zwraca wszelkie możliwe parametry, których nazwa zawiera podany czasownik. Można stosować kilka czasowników jak również maski w podawanych słowach.

```
Get-Command –Verb czasownik
```

GET-COMMAND -LISTIMPORTED

Podaje tylko te komendy, które dostępne w bieżącej sesji. Parametr ten jest przydatny, gdyż od PowerShell 3.0 domyślnie zwracane są wszystkie dostępne komendy, nie tylko te z bieżącej sesji.

```
Get-Command –ListImported
```

GET-COMMAND -PARAMETERNAME

Zwraca te komendy dostępne w danej sesji, które posiadają podane parametry. Można podawać nazwy parametrów lub ich aliasy. Obsługiwane są również maski w słowach.

```
Get-Command –ParameterName nazwa_parametru
```

GET-COMMAND –PARAMETERNAMETYPE

Jako wynik otrzymujemy listę komend dostępnych w aktualnej sesji, które mogą zostać wywołane z parametrami określonego typu. Można stosować całe lub częściowe nazwy typów. Dozwolone jest również stosowanie masek.

```
Get-Command –ParameterType typ_parametru
```

Oczywiście polecenie **Get-Command** oferuje jeszcze inne możliwe parametry, z którymi można je wywołać, takie jak**: -Verbose, -Debug, -ErrorAction, -ErrorVariable, -OutBuffer** czy **–OutVariable**.

Jednak są one na tyle rzadko stosowane, że nie ma potrzeby wspominać o nich w kursie podstaw PowerShell'a. Jeśli jednak ktoś jest zainteresowany działaniem wymienionych parametrów, może zawsze skorzystać z pomocy PowerShell'a.

GET-MEMBER

Wiedząc już jak znaleźć polecenie wśród gąszczu funkcjonalności zdarza się, że nie do końca wiemy jakie własności czy metody posiada obiekt na którym działamy. Z pomocą przyjdzie nam na tutaj polecenie **Get-Member**. Służy ono, jak z resztą wskazuje nazwa, to podpowiedzenia użytkownikowi jakie elementy składowe posiada obiekt wykorzystywanej klasy.

```
Get-Member paramerty
```

CIĄGI KOMEND, CZYLI STRUMIENIE

PowerShell, jako że nie zwraca jako efektu działania swoich komend tekstu, tylko obiekt umożliwia w prosty sposób obróbkę danych. Oczywiście ten model użytkowania PowerShell'a doceniony będzie głównie przez administratorów, gdyż wreszcie otrzymujemy dostęp do strumieni, które są zupełnie naturalne dla użytkowników systemu Unix i jego pochodnych oraz dla programistów. Dzięki temu nie trzeba poddawać parsowaniu wyników tekstowych i wykorzystania ich jako argument, ale można bezpośrednio przesłać wynik działania jednej komendy do drugiej.

W pierwszej kolejności, zanim jeszcze przejdziemy do przekazywania danych między poleceniami, warto wspomnieć o operatorze **>** i **>>**. Pierwszy z nich, **>**, powoduje przeniesienie standardowego wyjścia komendy do pliku. Za to **>>** spowoduje przekierowanie standardowego wyjścia do pliku z tą tylko różnicą, że plik nie zostanie zapisany od początku (ujmując to w bardziej profesjonalny sposób – nadpisany) , tylko dane zostaną do niego dopisane tuż przed znakiem końca pliku.

Użytkownicy Linux'owego terminala na pewno odczuli właśnie ulgę. Aby uspokoić ich do końca dodam, że operator **2>** powoduje przekierowanie standardowego wyjścia błędów do pliku. Jeśli komenda nie zwróciła żadnych błędów, zostanie utworzony plik pusty. Analogicznie działa operator **2>>**, który przekierowuje standardowe wyjście błędów do pliku.

Istnieje oczywiście możliwość równoległego przekierowania zarówno wyniku działania komendy jak i strumienia błędów do pliku. Można to zrobić w następujący sposób:

```
Komenda > plik 2> &1
```

Efektem tego działania będzie plik, w którym zostanie zapisany efekt działania komendy oraz strumień błędów. Można oczywiście rozdzielić oba strumienie i wykonać dwie operacje jedną komendą:

```
Komenda > plik 2> plik_błędu
```

Przejdźmy jednak do sedna. Operatorem przekierowania strumienia wyjściowego jednej aplikacji do innej aplikacji jest **|**. Dla przykładu:

```
Get-Process | Sort-Object ID
```

Spowoduje przekazanie listy procesów do posortowania względem ID procesu i wyświetlenie danych na ekranie. Więcej informacji na temat obróbki danych w dalszej części tutoriala.

DODATKOWE MODUŁY

A co jeśli brakuje funkcjonalności, która wydaje się być dla nas niezbędna? Oczywiście można stworzyć optymalne rozwiązanie na własną rękę, pod postacią skryptu czy modułu. Jednak może okazać się, że wkład pracy będzie niewspółmierny do czasu i pracy, które włożymy w jego wykonanie. Wtedy z pomocą przychodzi nam kilka przydatnych w pracy programisty/administratora systemowego stron www, na których znajdziemy gotowe rozwiązania. Są to między innymi strony projektu codeplex, przykładowe kody na MSDN czy strony Microsoft TechNet.

Jednak nie wszystkie moduły należy stworzyć lub pobrać z Internetu. Zgodnie polityką tworzenia nowych produktów, Microsoft uwzględnia dostarczanie przydatnych cmdet'ów, które mogą okazać się niezbędne do kontroli produktu z linii komend, do każdego z tworzonych narzędzi. Dodatkowo Microsoft zwraca uwagę, aby większość prac można było wykonać bez wykorzystania interfejsu graficznego.

Skoro PowerShell nie posiada od razu wszystkich modułów, a nie zawsze istnieje potrzeba pobrania nowych plików, to w takim razie skąd wziąć potrzebne cmdlet'y? Większość z potrzebnych narzędzi zostanie dodanych w trakcie instalacji danej funkcjonalności (np. przy instalacji IIS, do PowerShell'a zostanie dodany moduł WebAdministration).

Oczywiście pozostawiono użytkownikowi możliwość zarządzania poszczególnymi modułami. Służą ku temu trzy cmdlet'y:

- **`Get-Module`**
- **`Import-Module`**
- **`Remove-Module`**

Najważniejsze według mnie cmdlet'y, które mogą być przydatne w trakcie pracy zebrane zostały w tabeli Tab. 1. Oczywiście nie są to wszystkie polecenia. Pełną dokumentację i listę poleceń można znaleźć w helpie. I to właśnie Get-Help będzie nam towarzyszył w pracy z PowerShell'em. Umiejętność wyszukania polecenia, które jest nam potrzebne zaowocuje w przyszłości i spowoduje, że wcale nie trzeba poświęcić długiego czasu na opanowanie komend.

Windows PowerShell (Cmdlet)	Windows PowerShell (Alias)	cmd.exe / COMMAND.COM (MS-DOS, Windows, OS/2, etc.)	Bash (Unix, BSD, GNU/Linux, etc.)	Opis
Set-Location	sl, cd, chdir	cd, chdir	cd	Zmienia aktualną lokalizację (katalog, gałąź rejestru, repozytorium certyfikatów)
Clear-Host	cls, clear	cls	clear	Czyści ekran
Copy-Item	cpi, copy, cp	copy	cp	Kopiuje obiekty (np. pliki) lub fragment struktury (np. drzewo katalogów)
Get-Help	help, man	help	man	Wyświetla pomoc do komend
Remove-Item	ri, del, rmdir, rd, rm	del, rmdir, rd	rm, rmdir	Usuwa obiekt (plik, katalog itp.)
Rename-Item	rni, ren	ren	mv	Zmienia nazwę obiektu (pliku, katalogu itp.)
Get-ChildItem	gci, dir, ls	dir	ls	Zwraca wszystkie obiekty w bieżącej lokalizacji. (Na przykład pliki w aktualnym katalogu)
Write-Output	echo, write	echo	echo	Wyświetla łańcuchy, zmienne itd na ekranie
Pop-Location	popd	popd	popd	Zamienia aktualną lokalizację na lokalizację ostatnio przesuniętą na stos
Push-Location	pushd	pushd	pushd	Przesuwa aktualną lokalizację na stos

Set-Variable	sv, set	set	set	Wyświetla wartość zmiennej/Tworzy zmienną
Get-Content	gc, type, cat	type	cat	Wyświetla zawartość obiektu (np. pliku)
Get-Process	gps, ps	tlist, tasklist	ps	Wypisuje aktualnie uruchomione procesy
Stop-Process	spps, kill	kill, taskkill	kill	Zatrzymuje uruchomiony proces
Tee-Object	tee	?	tee	Tuneluje wejście do pliku lub zmiennej, przenosi wejście wzdłuż tunelu

Źródło: Wikipedia

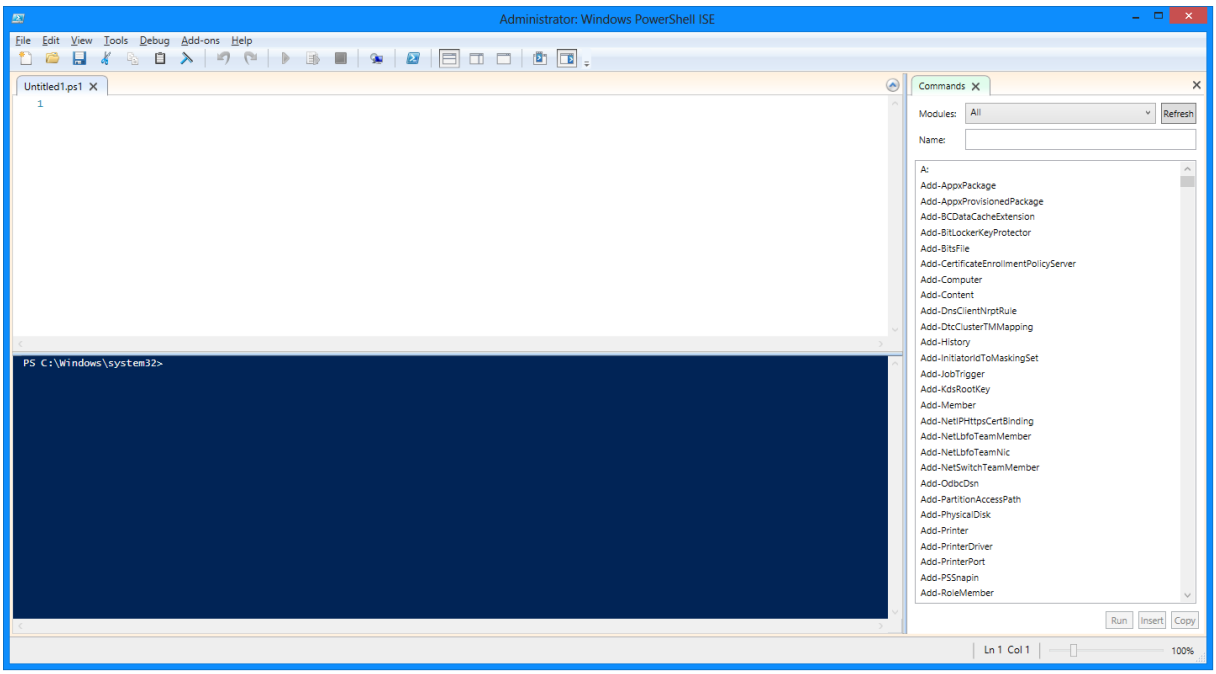
PROGRAMOWANIE W POWERSHELL'U

EDYTOR

Podstawowym narzędziem programisty jest IDE, z poziomu którego może tworzyć, testować i debuggować kod. Oczywiście w przypadku pisania skryptów w PowerShell'u przyda się również edytor. Idąc tym tropem można oczywiście stwierdzić, że wystarczy tutaj najprostszy edytor tekstu. Jest w tym trochę racji. Można oczywiście wykorzystać Notatnik lub podobny program. Można również pisać w Visual Studio.

Jednak o wiele wygodniej korzystać z dodatków, które powodują, że praca ze skryptami jest przyjemniejsza – mamy podpowiedzi, sprawdzanie składni i jej kolorowanie.

Istnieje kilka narzędzi, które ułatwią nam pracę. Na początku warto wspomnieć o edytorze PowerShell ISE (PowerShell Integrated Scripting Environment). Oferuje ono zarówno przejrzysty edytor jak i PowerShell'a wewnątrz siebie, dzięki czemu nie ma potrzeby przełączania się pomiędzy oknami. Dodatkowo środowisko to, oferuje nam debugger i spis komend, które można użyć. Dzięki „podpowiadaczowi" komend można przejrzeć parametry poszczególnych komend i po prostu „wyklikać" je w okienkach budując przy tym komendę z parametrami, którą można użyć lub komendę, którą można bezpośrednio wykonać. PowerShell ISE dostarcza nam pełną platformę skryptową do obsługi PowerShell'a.

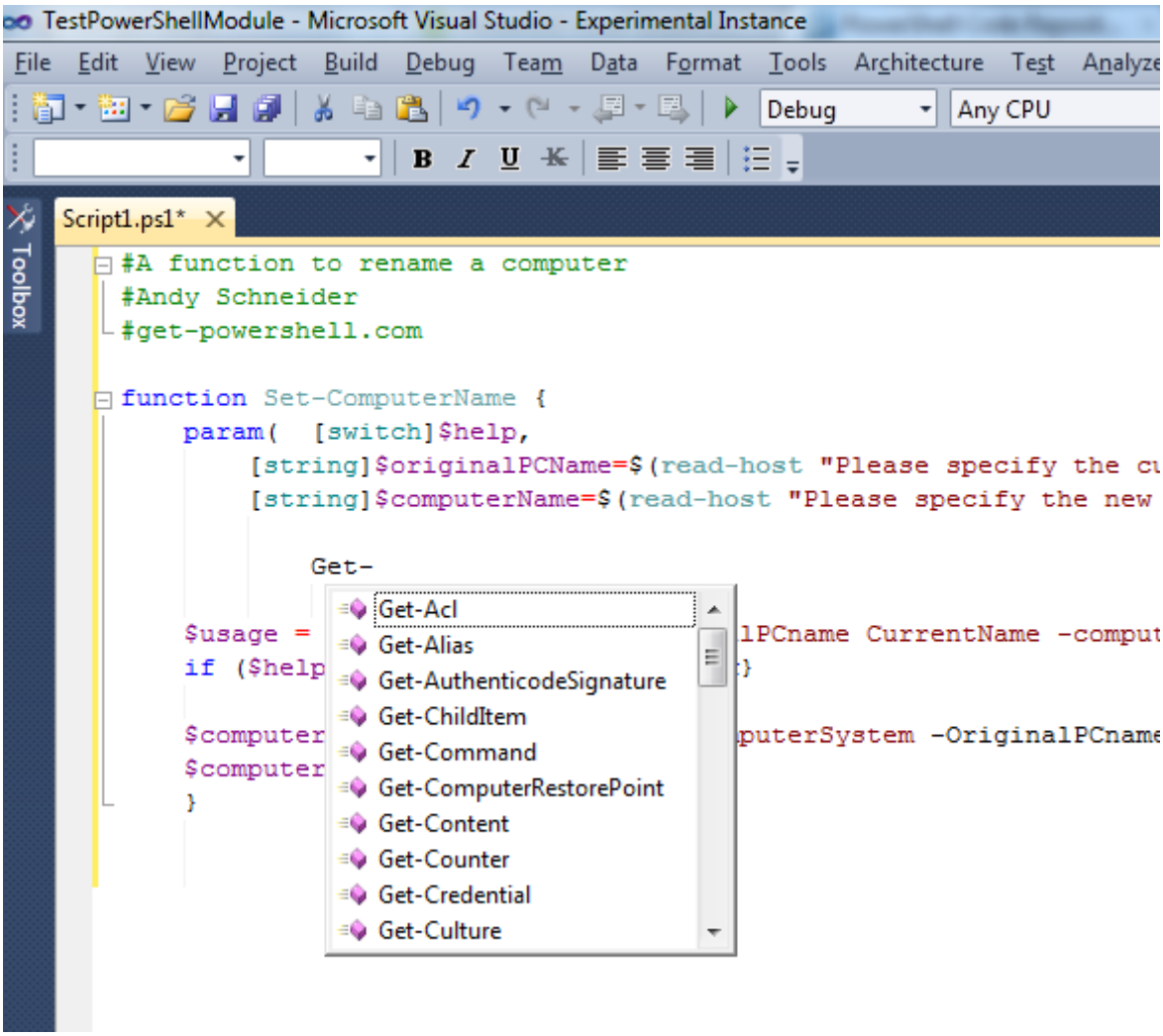


Rys. 3. – Edytor Windows PowerShell ISE

Wspomniałem również o Visual Studio. Z jego poziomu można bez problemu tworzyć skrypty, a po doinstalowaniu dodatku PowerGUI Visual Studio Extension otrzymujemy pełnego IntelliSense'a wspierającego PowerShell'a dla Visual Studio. Dla programistów, którzy na co dzień korzystają z tego narzędzia będzie to na pewno jedna z najwygodniejszych dróg tworzenia skryptów. Aby zapoznać się ze wszystkimi możliwościami, które daje PowerGUI warto odwiedzić stronę projektu na CodePlex'ie:

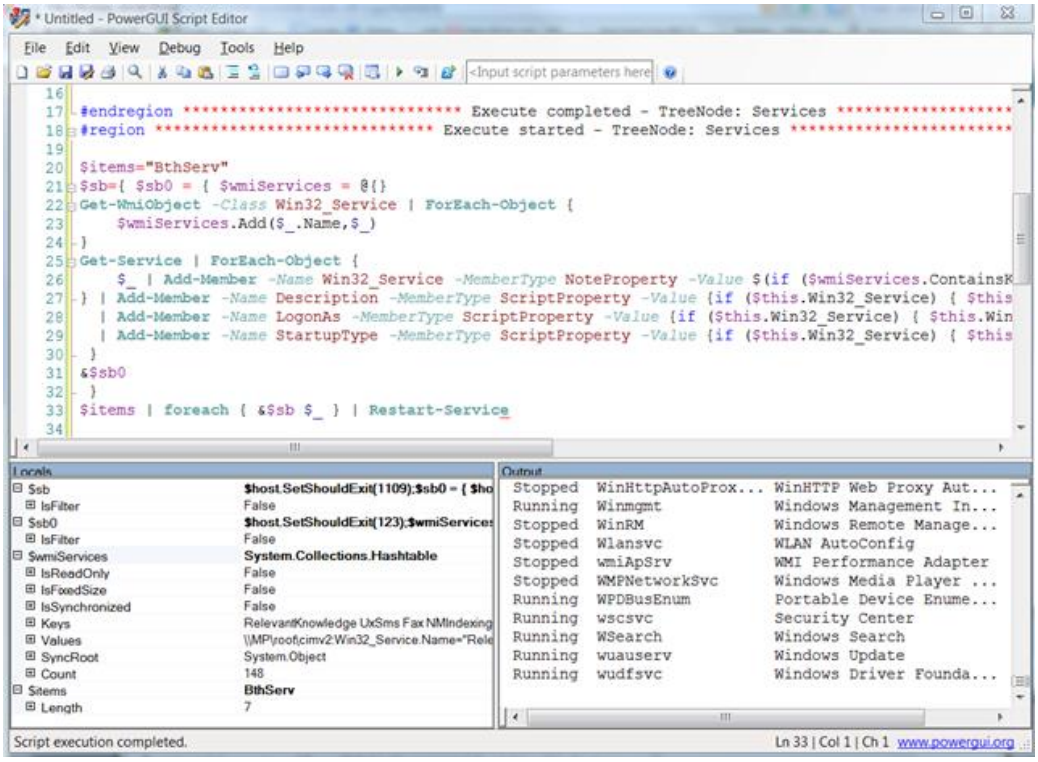
powerguivsx.codeplex.com. Stąd można przenieść się do bloga twórców oraz go repozytorium GitHub, gdzie można śledzić rozwój projektu.

Jak widać po instalacji pakietu można swobodnie korzystać z Visual Studio jako z edytora skryptów PowerShell:



Rys. 4. – Dodatek do Visual Studio robi z niego potężne narzędzie do tworzenia skryptów

Sam PowerShellGUI posiada narzędzie Script Editor, które również zawiera w sobie PowerShell'a i wygląda bardzo podobnie do PowerShell ISE, co widać na zrzutach ekranu:



Rys. 5. – PowerGUI jako oddzielny edytor

W takcie pracy nad podstawowymi skryptami i na potrzeby tego tutoriala korzyrzystałem z PowerShell ISE, ze względu na bardzo prosty interfejs. Oczywiście narzędzie, przy pomocy którego pracować będzie użytkownik zależy tylko od niego.

Na temat innych edytorów graficznych informacje znaleźć można na stronach Microsoft Technet w kategorii The Windows PowerShell Toolbox.

http://technet.microsoft.com/en-us/scriptcenter/ee861518.aspx

Kiedy wybraliśmy już edytor dla siebie, możemy przejść do tworzenia skryptów

UMOŻLIWIENIE WYKONYWANIA SKRYPTÓW

Zanim przejdziemy do jakiegokolwiek pisania kodu, powinniśmy skonfigurować PowerShell'a tak, aby można w nim było wykonywać skrypty. Domyślnie konfigurowany PowerShell nie pozwoli nam na uruchomienie stworzonego przez nas kodu.

Są na to dwa sposoby – pierwszym z nich jest wykorzystanie narzędzia Włącz wykonywanie skryptów w usłudze katalogowej Active Directory a drugim wykorzystanie PowerShell'a i gotowego cmdlet'a. Pierwszy ze sposobów jest banalnie prosty, gdy nasz komputer ma skonfigurowane AD lub gdy korzystamy z wersji Windows dla serwerów. Drugi zaś przydatny jest na zwykłych PC-tach.

W tym przypadku pokażę jak włączyć możliwość wykonywania skryptów z poziomu PowerShell'a.

Jednak zanim to zrobimy – kilka słów o poleceniu, które zostanie wykorzystane – Set-ExecutionPolicy. Polityka wykonywania skryptów może przyjmować jedną z czterech wartości:

- **Restricted** – uniemożliwia uruchamianie skryptów
- **All Signed** – umożliwia uruchomienie skryptów, które podpisane są przez wiarygodnego twórcę
- **Remote Signed** – skrypty lokalne mogą być uruchamiane, skrypty ściągnięte muszą być cyfrowo podpisane
- **Unrestructed** – można uruchamiać wszelkie skrypty, nawet te niepodpisane cyfrowo i od niepewnych wydawców.

Jako, że będziemy potrzebowali pełnego dostępu do PowerShell'a, wykonajmy komendę:

```
Set-ExecutionPolicy Unrestricted
```

Dzięki temu możemy korzystać zarówno z własnych skryptów jak i z przykładów, które znajdziemy w sieci.

PODSTAWY TWORZENIA SKRYPTÓW

ARGUMENTY

Argumenty uruchomieniowe skryptu, które podane zostały w linii komend można przeglądać dzięki zmiennej **$args**. **$args** jest tablicą, w której znajdują się parametry. Tablica ta indeksowana jest od 0. Aby dostać się bezpośrednio do ostatniego elementu tablicy, można wykorzystać indeks -1.

Wywołując skrypt:

```
PS> skrypt.ps1 –pierwszy –drugi –trzeci
```

Do parametrów dostać można się w następujący sposób:

- **$args[0] – pierwszy**
- **$args[1] – drugi**
- **$args[2] – trzeci**
- **$args[-1] – trzeci**

KOMENTOWANIE KODU

W czasie pracy bardzo często zdarzy nam się komentować kod – zarówno wstawiając komentarze na przyszłość, aby pamiętać o tym, dlaczego zrobiliśmy coś tak a nie inaczej, ale również celem wykluczenia pewnej części kodu z wykonania.

Znakiem komentarza w PowerShell'u jest znak #.

```
#

# Te trzy linie są zakomentowane

#
```

ZNAKI SPECJALNE FORMATOWANIA LINII

Bardzo często zdarza się, że potrzeba złamać linię lub złączyć klika instrukcji w jednej, aby kod był czytelny.

Do oddzielenia następujących po sobie instrukcji w PowerShell'u służy znak **;**

Oto przykład jego wykorzystania:

```
$a = 1; $b = 2; $c = $a + $b;
```

Istnieje również możliwość złamania linii z wykorzystaniem znaku `.

```
Write-Host `
  „KONTYNUACJA”
```

Podobne działanie ma znak **|**, który oddziela poszczególne polecenia, przekazując zarówno strumień obiektu z jednego cmdlet'a do drugiego:

```
Get-ChildItem | Sort-Object Size

   Get-ChildItem |

                  Sort-Object Size
```

Które wylistuje bieżący katalog i posortuje wyniki zgodnie z ich rozmiarem.

INTERAKCJA Z UŻYTKOWNIKIEM

WYŚWIETLENIE DANYCH NA KONSOLI

Podstawowym sposobem na interakcję z użytkownikiem jest pobieranie i wyświetlanie danych w konsoli. Aby wyświetlić dane wykorzystamy cdlet'a Write-Host. Znany wszystkim programistom „Hello World!" w przypadku PowerShell'a wygląda następująco:

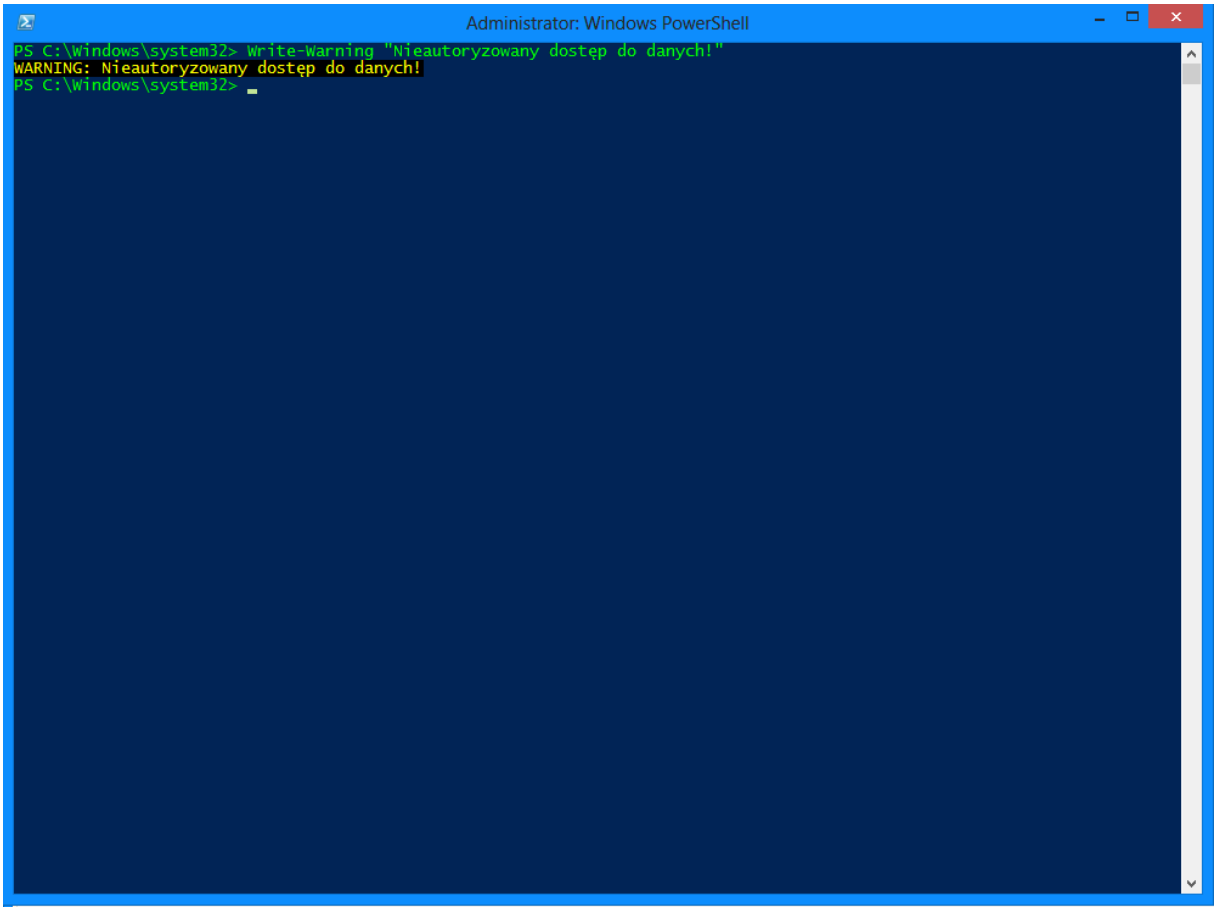
```
Write-Host „Hello world"
```

Oczywiście PowerShell daje nam kilka „wodotrysków", które można wykorzystać przy interakcji z użytkownikiem. Chodzi tutaj o zmianę kolorów - zarówno tła jak i czcionki.

Najprostszym sposobem na zmianę wizualną stylu wyświetlania komunikatów jest wykorzystanie trybu ostrzegawczego konsoli:

```
Write-Warnig „Nieautoryzowany dostęp do danych!"
```

W przypadku ustawień zastosowanych na początku tego tutorialu okno, efekt, który zobaczymy po wykonaniu tych operacji powinien wyglądać następująco:



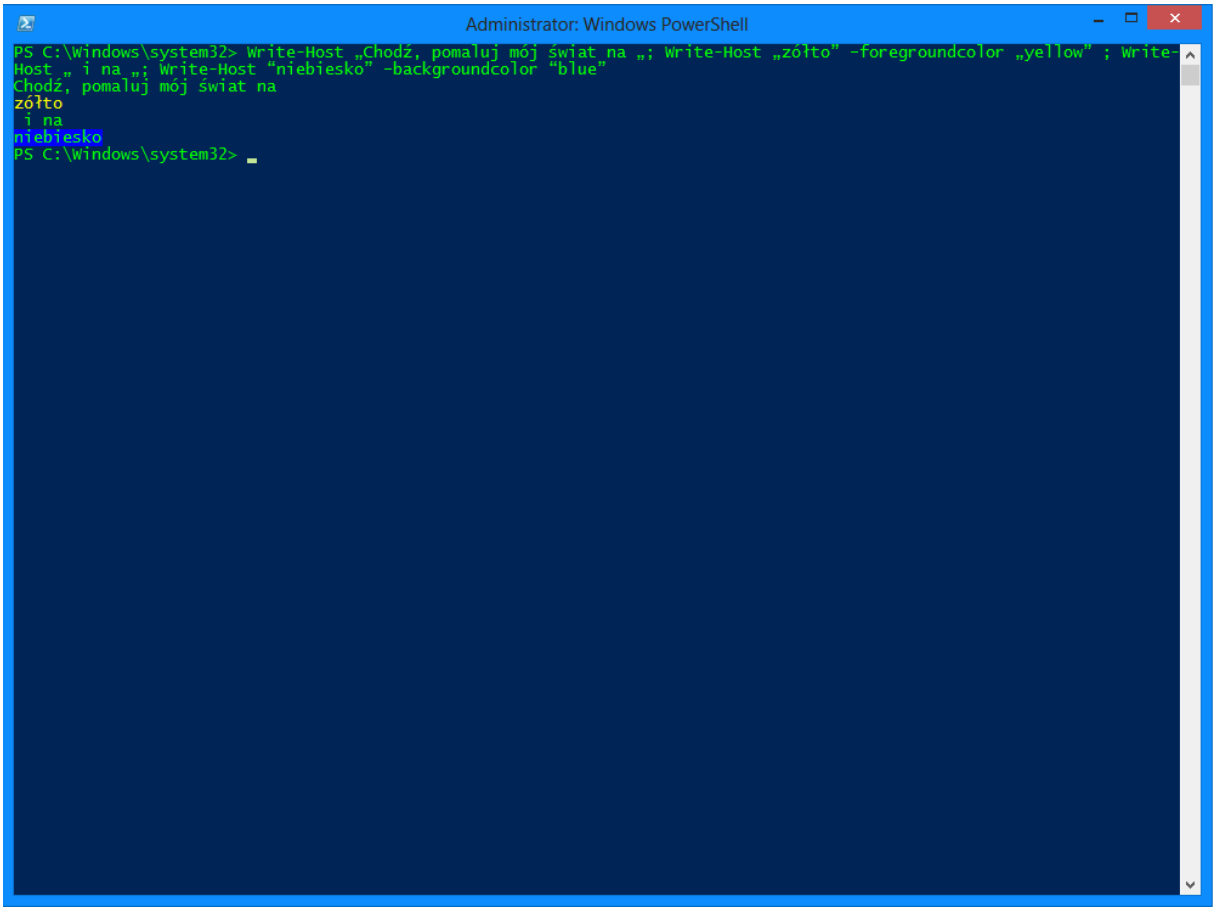
Rys. 6. – ostrzeżenie wygenerowane z poziomu PowerShell'a

Aby wykorzystać dowolny kolor z dostępnej palety, można zastosować argument –foregroundcolor, który dla cmdlet'a Write-Host zmieni kolor pierwszoplanowy oraz cmdlet'a –backgroundcolor, który zmieni kolor drugoplanowy.

Przykładowe użycie wygląda następująco:

```
Write-Host „Chodź, pomaluj mój świat na „; Write-Host „zółto” –
foregroundcolor „yellow” ; Write-Host „ i na „; Write-Host
“niebiesko” –backgroundcolor “blue”
```

Efekt działania jest następujący:



Rys. 7. – Kolorowy tekst w konsoli PowerShell

Jak łatwo zauważyć, wpisywany tekst jest łamany dopiero po zakończeniu polecenia. Aby złamać go wcześniej wystarczy zawrzeć w miejscu podziału ciąg: `n, tak jak w poniższym przykładzie:

```
Write-Host „To jest linia pierwsza. `n A to już linia druga.”
```

POBIERANIE DANYCH Z KONSOLI

Równie banalne jest pobieranie danych z konsoli. Służy ku temu polecenie Read-Host (po raz kolejny zwracam uwagę na bardzo logiczną nomenklaturę stosowaną przy nazewnictwie cmdlet'ów).

```
$zmienna = Read-Host „Podaj swoje imię tajemniczy użytkowniku:”
```

Polecenie przedstawione powyżej spowoduje wyświetlenie na konsoli monitu o podanie danych i wczytanie danych do zmiennej **$zmienna.**

PORÓWNYWANIE ZMIENNYCH

Jeśli korzystamy ze zmiennych, będziemy je chcieli zapewne porównać. Oczywiście część z cmdlet'ów posiada już zaimplementowaną obsługę operatorów porównania (np. **Where-Object**).

W PowerShell'u można wykorzystać następujące operatory porównania:

- **-lt** – mniejsze niż
- **-le** – mniejsze lub równe
- **-gt** – większe niż
- **-ge** – większe bądź równe
- **-eq** – równe
- **-ne** – nierówne
- **-like** – podobne do (wykorzystuje symbole wieloznaczne i maski)
- **-notlike** – niepodobne do (również wykorzystuje symbole wieloznaczne i maski)

Dodatkowo operatory mogą rozróżniać wielkie i małe litery. Aby włączyć tą funkcjonalność należy przed symbolem operatora porównania użyć litery c. Np. operator **–ceg** to operator równości z uwzględnieniem wielkości liter.

CZYTANIE DANYCH Z PLIKÓW

PoweShell umożliwia również czytanie danych z pliku, przy wykorzystaniu polecenia Get-Content. Jego składnia wygląda następująco:

```
$wczytany_plik = Get-Cotent C:\Users\PiotrAblewsi\pliczek.txt
```

Każda z linii w pliku taktowana jest jako oddzielna zmienna, która znajduje się w tablicy. Dostać można się do niej poprzez numer wiersza w pliku. Zasady numeracji są takie same jak w przypadku argumentów wejściowych skryptu - **$wczytany_plik[0]** to linia pierwsza, **$wczytany_pli[10]** to linia jedenasta, a **$wczytany_plik[-1]** to ostania linia w pliku.

Aby określić ilość linii, słów lub znaków w pliku, należy zastosować polecenie:

```
Get-Content ścieżka_do_pliku | Measure-Object –line –word –character
```

ZAPISYWANIE DO PLIKU

Skoro umiemy już otwierać pliki, warto zapoznać się ze sposobami na zapisanie danych do pliku. Oczywiście dostępne są wcześniej wspomniane strumienie plikowe znane jeszcze z czasów DOSa, czyli **>**, który nadpisuje plik i **>>**, który dopisuje do pliku.

Jednak stworzony został cmdlet, który umożliwia zapis do pliku. Jest to **Out-File**.

Świetnym przykładem zapisu do pliku jest wylistowanie procesów uruchomionych na maszynie, tak więc w PowerShell'u można zrealizować to na dwa sposoby:

Pierwszy, znany już z czasów DOS'a:

```
Get-Process > plik
```

Celem utworzenia nowego pliku, lub

```
Get-Process >> plik
```

Celem dopisania treści do istniejącego już pliku.

Drugim sposobem jest zastosowanie cmdlet'a Out-File:

```
Get-Process | Out-File plik
```

Celem nadpisania pliku, lub

```
Get-Process | Out-File plik –Append
```

Celem dopisania danych do istniejącego pliku.

Oczywiście dane można zapisać w bardziej przyjaznym do obróbki formacie CSV, stosując polecenie **Export-CSV**:

```
Get-Process | Export-CSV plik
```

DRUKOWANIE DANYCH

Dane w PowerShell'u można przedstawiać nie tylko na standardowym wyjściu czy w pliku, ale również można wysłać je na domyślną drukarkę poprzez polecenie **Out-Printer**:

```
Get-Process | Out-Printer
```

Spowoduje to wydrukowanie wszystkich procesów uruchomionych na naszej maszynie na domyślnej drukarce.

PĘTLE I INSTRUKCJE WARUNKOWE

PowerShell, jako że jest pełnoprawnym językiem programowania, posiada szereg pętli i instrukcji, które można wykorzystać przy tworzeniu kodu. Postaram się omówić ich strukturę w sposób na tyle jasny, ze nie będzie problemem ich wykorzystanie w trakcie pisania skryptów.

IF … ELSEIF .. ELSE

Podstawową instrukcją warunkową wykrzystywana przez PowerShell jest poczciwy IF. Jego konstrukcja powinna być znana wszystkim porgramistom, w takiej formie jak w tym przypadku – programistom tworzącym skrypty pod powłoki systemu Linux/UNIX.

```
if (warunek1)
{
   Instrukcje…
}
elseif ( warunek2)
{
   Instrukcje…
}
else
{
   Instrukcje…
}
```

Opcje **elseif** i **else** są oczywiście opcjonalne i najprostsza konstrukcja zakłada wykorzystanie samej opcji **if**.

SWITCH

Dzięki instrukcji **Switch** można w prosty sposób uniknąć tworzenia wielu instrukcji warunkowych **if** po sobie.

```
switch (zmienna)
{
   wartość_1 {instrukcje}
   wartość_2{instrukcje}
   …
   wartość_n {instrukcje}
   default {instrukcje wykonane, gdy warunki nie pasują}
}
```

FOR

Jeśli pętla ma wykonać się dla określonej wartości zmiennych, najlepszym wyborem będzie **FOR**

```
for(wartość_początkowa; warunek_zakończenia; inkrementacja_zmiennej)
{
Instrukcje;
}
```

FOREACH

Jeśli będziemy operować na większym zbiorze danych, jak na przykład na obiekcie, bardzo pomocna okaże się pętle **Foreach**:

```
foreach ($i in lista_parametrów)
{
     Instrukcje zależne od zmiennej $i;
}
```

PĘTLA WHILE – STEROWANA Z GÓRY

Jeśli zależy nam na wykonywaniu się kodu do osiągnięcia pewnego warunku, można wykorzystać pętlę **While.** Wykonuje się ona dopóty, dopóki warunek jest spełniony:

```
while {warunek}
{
    Instrukcje;
}
```

PĘTLA DO WHILE – STEROWANA Z DOŁU

Jeśli instrukcje mają zostać najpierw wykonane, a później ma zostać sprawdzony warunek (czyli wykonanie instrukcji nastąpi na pewno minimum raz), możemy wykorzystać pętlę **Do While**:

```
do
{
    Instrukcje;
}
while (warunek)
```

DO … UNTIL

Ciekawostką jest również fakt, że do łask wróciła pętla **until** sterowana z dołu, a więc instrukcja wykona się przynajmniej raz i będzie wykonywana dopóty, dopóki warunek nie będzie spełniony:

```
do
{
    Instrukcje;
}
until (warunek)
```

BARDZIEJ ZAAWANSOWANE FUNCJE POWERSHELL'A DLA PROGRAMISTÓW (W SZCZEGÓLNOŚCI .NET)

Znając już postawy można przejść do dalszej, bardziej zaawansowanej pracy. Oczywiście pełnej wprawy można nabrać dopiero po kilku godzinach pracy z PowerShell'em, ale postawy, które tutaj przedstawię powinny dać pewien pogląd na to jak w prosty sposób okiełznać zaawansowanego PowerShella (w szczególności będąc programistą).

OBIEKTY COM

PowerShell umożliwia pracę z obiektami **COM**. Tworzy się je przy użyciu polecenia **New-Object** z parametrem **–comobject**, gdzie jako argument parametru podawany jest **ProgID.**

Przykładowo Uruchomienie Internet Explorera z poziomu PowerShell'a wygląda następująco:

```
$obj = New-Object –comobject „InternetExplorer.Application”

$obj.Visible = $true;
```

Należy pamiętać, że aby korzystać z obiektów COM potrzebne są przywileje administratora.

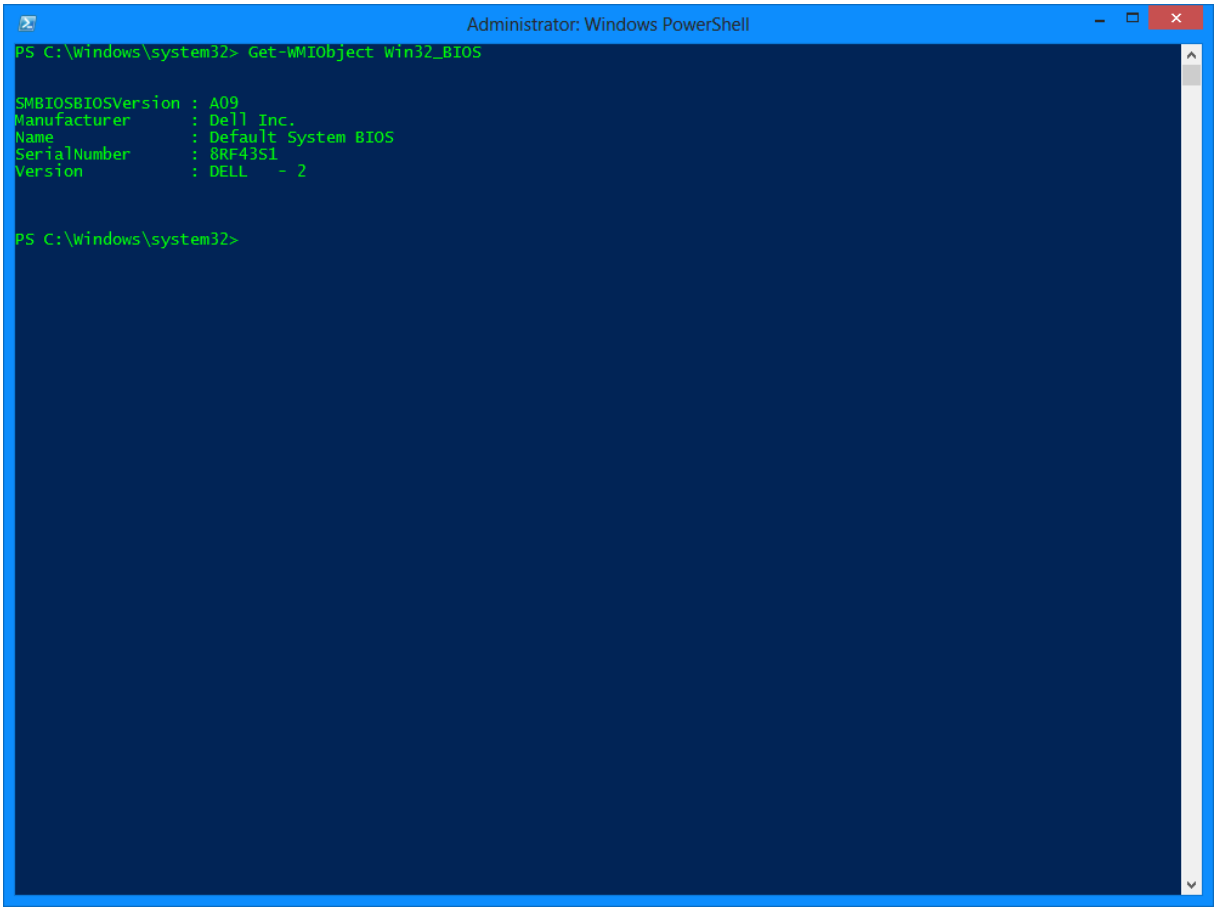
Aby zobaczyć listę możliwych do wykorzystania obiektów należy skorzystać z pomocy PowerShell'a.

WMI

Aby uzyskiwać dane przy pomocy WMI (Windows Management Instruction) należy skorzystać z polecenia **Get-WMIObject**, jako parametr podając nazwę klasy. Aby zilustrować działanie WMI pod PowerShell'em, wykorzystajmy podstawową klasę WMI – **Win32_BIOS**:

```
Get-WMIObject Win32_BIOS
```

Efekt działania jest następujący:



Rys.8. – Efekt działania na obiektach WMI

Jeśli klasa nie występuje w wykorzystywanej przestrzeni nazw **(cimv2),** konieczne będzie wykorzystanie parametru **–namesace.**

```
Get-WMIObject nazwa_klasy -namespace przestrzeń_nazw
```

Podobnie sytuacja wygląda, gdy korzystamy z połączenia zdalnego. Wtedy należy wykorzystać parametr **–computername**:

```
Get-WMIObject klasa –computername nazwa_komputera_zdalnego
```

Oczywiście możne nie interesować nas kompletny zestaw informacji zwracanych w obiekcie. Dlatego warto jest wykorzystać zapytania WQL (SQL for WMI). Można to zrobić poprzez parametr **–query**, w którym jako argument podaje się zapytanie w SQL, np.:

```
Get-WMIObject -query "Select * From Win32_Service Where State =
'Running'"
```

Polecenie wylistuje uruchomione procesy.

OBIEKTY .NET FRAMEWORK

Obiekty .NET Framework są najciekawszą, z punktu widzenia programisty, szczególnie .NET częścią PowerShell'a. To dzięki nim właśnie czujemy się tutaj jak w domu.

Do obiektów .NET Framework można dostać się poprzez nazwę klasy zawartą w nawiasach kwadratowych (z uwzględnieniem wszystkich przestrzeni nazw!) oraz metodę. Obie części należy oddzielić podwójnym dwukropkiem (**::**).

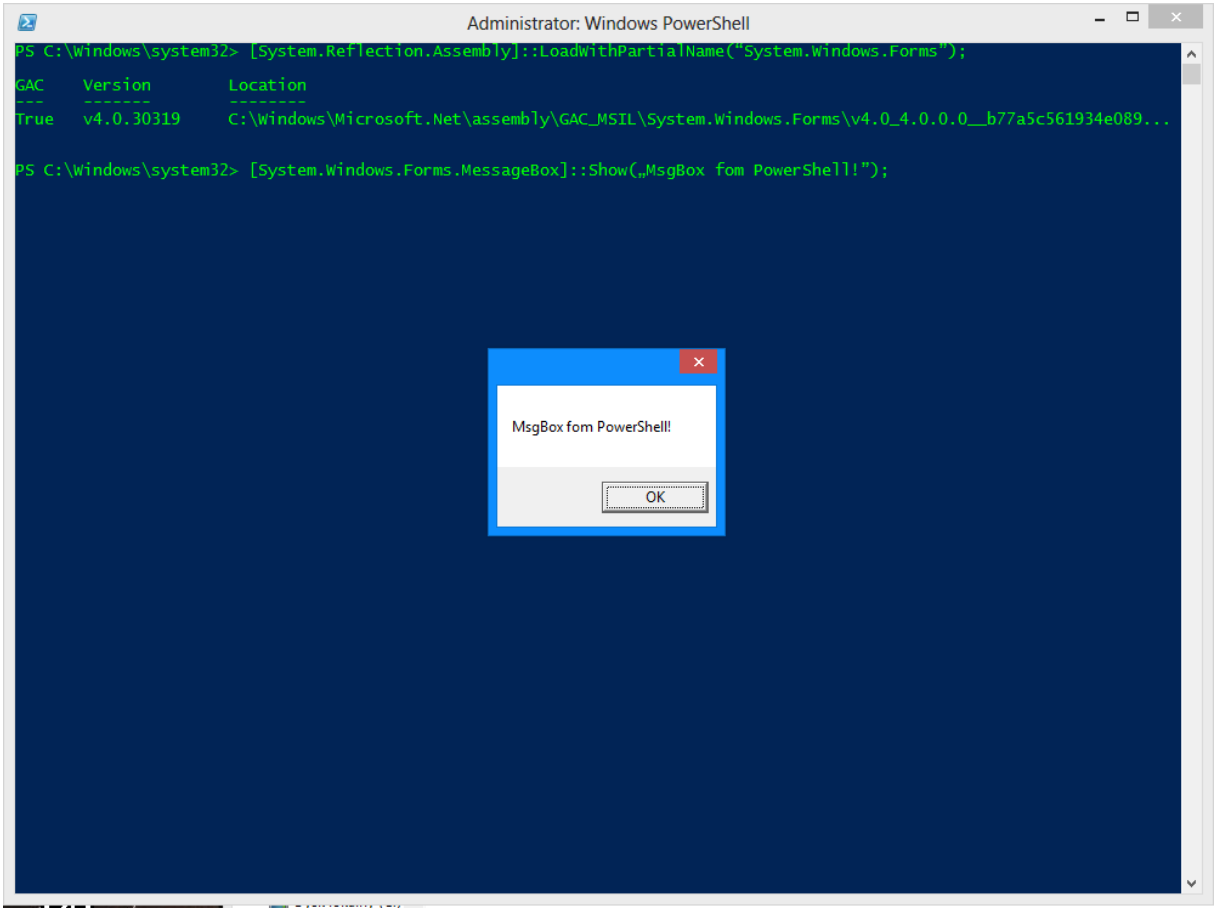
Przykładowo:

```
[System.Windows.Forms.MessageBox]::Show(„MsgBox fom PowerShell!”);
```

Efekt nie wygląda tak jak byś się tego spodziewał? Spokojnie. W pierwszej kolejności należy załadować przestrzeń nazw. W tym przypadku będzie to wyglądać następująco:

```
[System.Reflection.Assembly]::LoadWithPartialName(“System.Windows.Forms”);
```

Dzięki powtórzeniu poprzedniego polecenia powinniśmy ujrzeć następujące okno:



Rys. 9 – MessageBox uruchomiony z poziomu PowerShell'a.

Niestety nie wszystkie obiekty .NET są wspierane przez PowerShell i przed ich użyciem należy zapoznać się z dokumentacją techniczną.

KILKA CIEKAWYCH SZTUCZEK – ZABAWA Z POWERSHELL'EM

MARSZ IMPERIALNY – POWERSHELL W ROLI MUZYKA ☺

```
[console]::beep(440,500)
[console]::beep(440,500)
[console]::beep(440,500)
[console]::beep(349,350)
[console]::beep(523,150)
[console]::beep(440,500)
[console]::beep(349,350)
[console]::beep(523,150)
[console]::beep(440,1000)
[console]::beep(659,500)
[console]::beep(659,500)
[console]::beep(659,500)
[console]::beep(698,350)
[console]::beep(523,150)
[console]::beep(415,500)
[console]::beep(349,350)
[console]::beep(523,150)
[console]::beep(440,1000)
```

MÓWIĄCY POWERSHELL – COM OBJECT

```
$voice = New-Object –COMObject “SAPI.SPVoice”

$voice.speak(“Latam, gadam – pełny serwis”)
```

Ale jak widać język angielski znacznie lepiej naszemu komputerowi wychodzi:

```
$voice.speak(“Hello World! I’m PowerShell! I can speak to you!”)
```

PODSUMOWANIE

Tutorial ten jest wstępem PowerShell'a, i każdy, kto będzie chciał nauczyć się go porządnie – spędzi nad tym narzędziem bardzo dużo czasu. Będzie to czas spędzony na odkrywaniu możliwości szukaniu nowych dróg.

Polecam w szczególności lekturę manuala, czyli częste wykorzystywanie polecenia Get-Help i szukanie potrzebnych informacji na TechNet'cie, który jest potężną skarbnicą wiedzy.