1 Historia PowerShella

W tym rozdziale przedstawiony zostanie historia oraz rozwój powłoki Windows PowerShell.

1.1 *Pierwsze kroki*

Okno konsoli w stylu DOS pozostało w niezmiennej formie przez wiele wersji systemu Windows. Wraz z wydaniem Windows PowerShell firma Microsoft wydała jego godnego następcę. PowerShella można śmiało nazwać adaptacją pomysłów zawartych w powłokach systemów UNIX z wykorzystaniem technologii .NET Framework oraz WMI. Tworzenie skryptów w Windows Script Host było zbyt skomplikowane dla wielu administratorów, ponieważ wymagało wiedzy na temat programowania obiektowego oraz Component Object Model (COM). Wiele wyjątków oraz niespójność COM spowodowały, że tworzenie skryptów w WSH z wykorzystaniem bibliotek było trudne do opanowania. Po wydaniu Microsoft .NET Framework w 2002 roku wielu użytkowników oczekiwało rozwiązania nazwanego „WSH.NET". Jednakże firma Microsoft zaprzestała wydawania kolejnych wersji WSH dla .NET Framework, ponieważ wymagałoby to używania języków programowania bazujących na .NET takich jak C# czy Virtual Basic w jeszcze większym stopniu zaawansowania niż dotychczas oraz obszerniejszej wiedzy o programowaniu obiektowym. Firma Microsoft dostrzegła popularność powłok UNIX-owych i zdecydowała aby połączyć potokową koncepcję powłok UNIX-owych z .NET Framework. Rezultatem było wydanie nowej powłoki prostej w użyciu oraz wykorzystującej potencjał .NET. Windows PowerShell po raz pierwszy został przedstawiony na Professional Developers Conference mającej miejsce we październiku 2003 roku i był znany pod nazwą kodową „Monad". W roku 2006 Windows PowerShell wydał wersję 1.0 zapewniającą pełne wsparcie dla Windows XP Service Pack 2, Windows XP Service Pack 3, Windows Server 2003 oraz Windows Vista. Aby zapewnić wsparcie dla systemu Windows Server 2008 należało wprowadzić wiele zmian w rejestrze celem przystosowania systemu do pracy z powłoką. Początkowo nazwą powłoki było Microsoft Shell (MSH), zastąpione później przez Microsoft Command Shell. Finalną nazwę: PowerShell, powłoka przyjęła w maju 2006 roku.

1.2 Windows PowerShell 1.0

Windows PowerShell 1.0 był pierwszym w pełni wydanym produktem Windows PowerShell. Pomimo tego, że dostarczał użytkownikowi wszelkie kluczowe elementy jakie powinna mieć powłoka, proces adaptacji tejże wersji przebiegał bardzo powoli z kilku przyczyn:

- Nie był on preinstalowany w żadnym z istniejących w tym czasie systemów operacyjnych firmy Microsoft, dlatego administratorzy musieli podjąć świadomy wysiłek, aby wdrożyć do systemu powłokę PowerShell.
- Administratorzy, którzy już wcześniej doprowadzili do perfekcji posługiwanie się innym językiem skryptowym nie czuli potrzeby używania nowej powłoki do wykonywania tych samych zadań.
- Dopiero po pewnym czasie społeczność użytkowników korzystających z PowerShella rozrosła się na tyle, aby móc zademonstrować bardziej kreatywne rozwiązania jakie oferuje nowy produkt.

Ostatecznie programiści Microsoftu zaczęli korzystać z Windows PowerShell 1.0 co zaowocowało wprowadzeniem powłoki do produktów nad którymi głównie wtedy pracowano: Microsoft Exchange 2007 i System Center Operations Manager. Następnie Windows PowerShell 1.0 został wydany wraz z Windows Server 2008 jako możliwy do zainstalowania dodatek. Umożliwiło to zaprezentowanie możliwości PowerShella gwarantując tym samym wzrost zainteresowania nową powłoką [2].

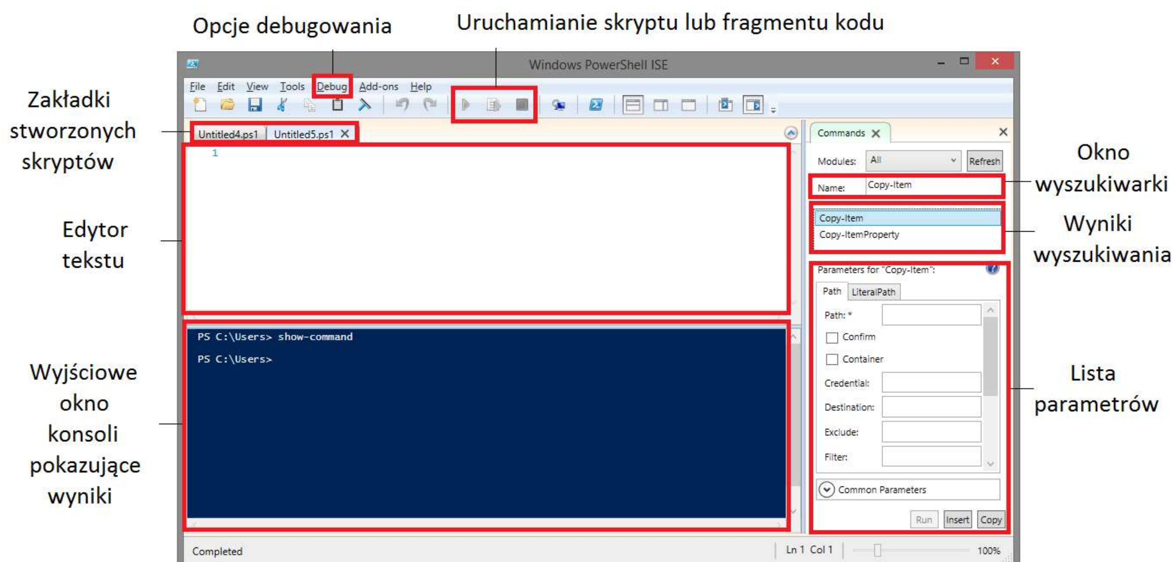
1.3 Windows PowerShell 2.0

Pomimo powolnego przyjmowania się PowerShella w wersji 1.0, szybko rozrastająca się społeczność użytkowników narzuciła odpowiednie tempo rozwoju. Deweloperzy Microsoftu wzięli pod uwagę krytykę oraz opinie użytkowników obiecując tym samym stworzenie jeszcze bardziej użytkowego środowiska jakim miał się stać Windows PowerShell 2.0.

Oto kilka głównym zmian i udogodnień jakie wprowadzono do wersji 2.0:

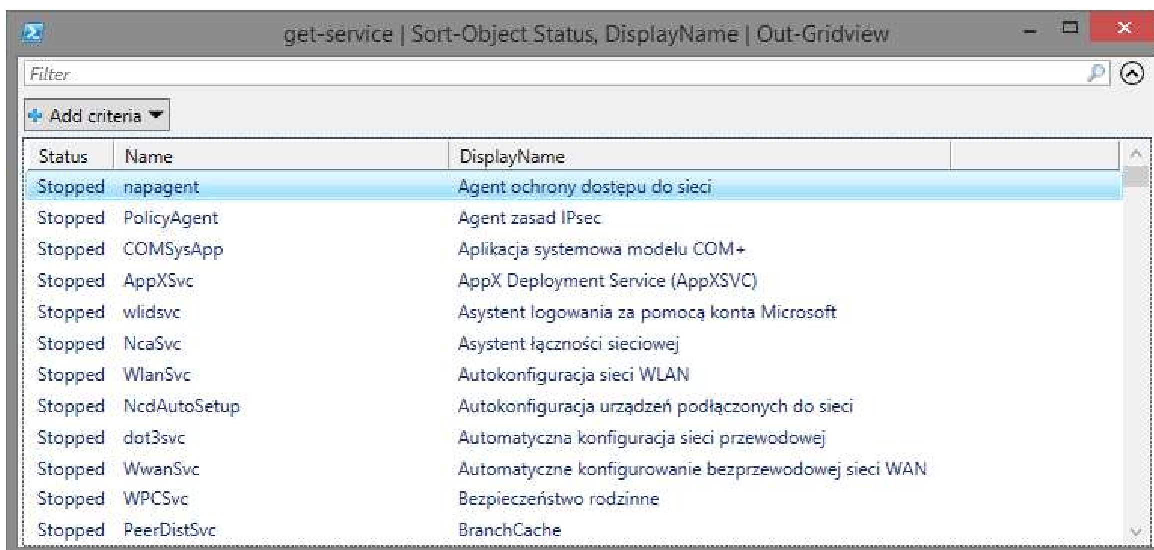
- Obsługa zdalna – pozwala na wykonywanie poleceń Cmdlet oraz skryptów zdalnie.

- Działanie w tle – daje możliwość uruchamiania komend działających w tle, podczas gdy użytkownik może zająć się innymi rzeczami.
- Zaawansowane funkcje – komendy Cmdlet musiały być pisane w C# lub VB.NET, wersja 2.0 pozwala na tworzenie pseudo-komend używając wyłącznie PowerShella.
- Język danych – pozwala na odseparowanie tworzonego kodu od danych czyniąc tym samym skrypt bardziej przenośnym i prostszym do zrozumienia.
- Debugowanie skryptów – możliwość ustawiania pułapek (breakpointów) w skryptach, które zatrzymają ich wykonywanie w określonym momencie pozwalając na dokładną analizę fragmentu kodu.
- Nowe operatory oraz zmienne automatyczne – kilka nowych operatorów aby ułatwić dzielenie i łącznie ciągów znaków.
- Nowe komendy typu cmdlet – w większości wspierające istniejące już rozwiązania.
- Wymuszone środowiska uruchamiania – daje możliwość wymuszenia jakie komendy oraz skrypty Windows PowerShell będzie mógł uruchomić wewnątrz udostępnionego środowiska.
- Zintegrowane środowisko pisania skryptów (ang. Integrated Scripting Environment, ISE) – graficzna wersja powłoki. Poza standardowym oknem konsoli posiada ono kilka dodatkowych opcji oraz ulepszeń. Wśród zastosowanych udogodnień możemy wymienić: okno wyjściowe konsoli, edytor tekstu służący do pisania skryptów (pozwala na pisanie wielu skryptów jednocześnie przechowując każdy z nich w oddzielnych zakładkach), panel boczny z zakładką *Commands*, która posiada spis wszystkich komend oraz wyszukiwarkę poleceń, gdzie po odnalezieniu interesującego nas polecenia wyświetlona zostanie lista parametrów jakie możemy zastosować. Ponadto dysponujemy możliwością uruchomienia właśnie pisanego skryptu z poziomu edytora lub uruchomienia określonej sekcji kodu. Dodatkowo z paska menu możemy uruchomić debugowanie naszego skryptu by krok po kroku analizować przebieg skryptu.



Rysunek 1.1. Integrated Scripting Environment w Windows PowerShell.

- Rzutowanie wyników na siatkę (ang. Out-GridView) – możliwość wyświetlania wyników działania komendy jako interaktywnej tabeli, którą można sortować, przeszukiwać oraz grupować wyniki.



Rysunek 1.2. Zastosowanie Out-GridView.

1.4 Windows PowerShell 3.0

Windows PowerShell w wersji 3.0 był zintegrowany z Windows 8 oraz Windows Server 2012. Ponadto Microsoft umożliwił zainstalowanie wersji 3.0 w systemach Windows 7, Windows Server 2008 i Windows Server 2008 R2 poprzez wykorzystanie rozszerzenia

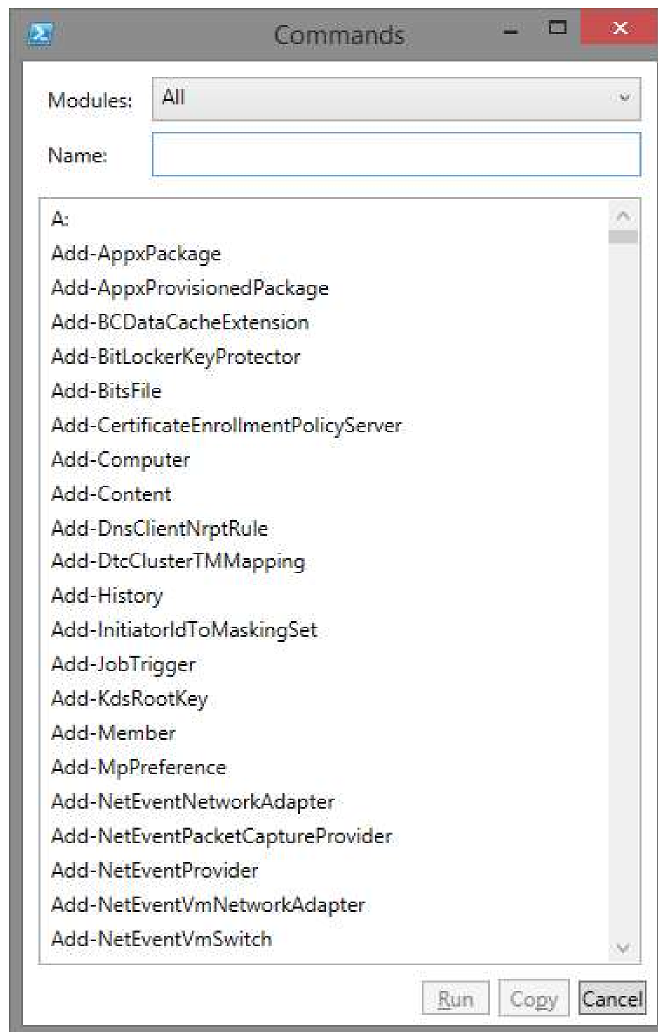
Service Pack 1 do każdej z wersji wyżej wymienionych systemów. Oto kilka nowości jakie oferowała nowa wersja powłoki:

- Organizacja pracy w Windows PowerShell (ang. Windows PowerShell Workflow) - pozwala administratorom na tworzenie oraz wywoływanie zadań, które mogą być: powtarzalne, możliwe do zatrzymania, możliwe do przerwania lub zrestartowania. W przypadku zastosowania tego aspektu w połączeniu z zarządzaniem zdalnym, użytkownik może zarządzać zdalnie wieloma komputerami jednocześnie. Jako potok pracy można rozumieć sekwencje zaprogramowanych kroków działania które pozwalają m. in. na wywoływanie długoterminowych zadań, koordynację wielu etapów działania programów na wielu urządzeniach lub wątkach naraz. Jeśli przyjrzeć się bliżej zasadzie działania takiej organizacji pracy można zauważyć, że to nie Windows PowerShell bezpośrednio ją wykonuje. Windows PowerShell tłumaczy przekazane do wykonania skrypty na język XAML[1], który z kolei jest wykonywany przez Workflow Foundation[2] [7].
- Rozłączane sesje – sesje w PowerShell mogą być rozłączane ze zdalnego komputera oraz przywrócone z dowolnego komputera bez utraty danych, bądź zatrzymywania działających procesów.
- Odnawialne sesje zdalne – elastycznie odpowiadają na błędy sieci próbując odnowić połączenie przez kilka minut od przerwania połączenia. W przypadku niepowodzenia sesja automatycznie zostaje rozłączona tak, aby mogła zostać przywrócona zaraz po odzyskaniu łączności z siecią.
- Zaplanowane zadania – programy oraz skrypty napisane w Windows PowerShell mogą być uruchamiane przez harmonogramem zadań umożliwiając regularną pracę zgodnie z harmonogramem lub w odpowiedzi na konkretne wydarzenie.

1 XAML (ang. Extensible Application Markup Language) – jest to język bazujący na XML, dzięki któremu można opisać interfejs użytkownika. Znajduje on zastosowanie między innymi w takiej technologii jak Windows Presentation Foundation, która jest nieodłącznym składnikiem .NET Framework od wersji 3.0.

2 Windows Workflow Foundation – jest to technologia stworzona przez Microsoft, która dostarcza API (ang. Application Programming Interface) pozwalające na tworzenie ludzkich, bądź systemowych przepływów pracy w aplikacjach w celu zarządzania i wykonywania długoterminowych zadań. Obecna wersja Windows Workflow Foundation jest zawarta w pakiecie .NET Framework 3.0.

- Delegowanie uprawnień – komendy mogą być uruchamiane przez użytkownika z większymi uprawnieniami niż te, które posiada poprzez dołączanie do nich listy uwierzytelniającej od administratora. Pozwala to na tymczasowe nadanie większych uprawnień użytkownikowi w przypadku np. błędów krytycznych i potrzeby natychmiastowego działania.
- Uproszczona składnia języka – dzięki niej komendy oraz skrypty mniej przypominają kod źródłowy oraz są bardziej intuicyjne dla użytkownika.
- Wyszukiwanie komend typu Cmdlet – udoskonalony system podpowiedzi i sugestii pozwala łatwo i szybko odnaleźć każde polecenie zainstalowane na komputerze.
- Polecenie Show-Command – jest to komenda oraz dodatek do ISE (ang. Integrated Scripting Environment), który pomaga użytkownikowi znaleźć właściwą komendę, pokazuje jej parametry w nowym oknie dialogowym oraz umożliwia uruchomienie jej [6].



Rysunek 1.3. Zastosowanie polecenia Show-Command.

1.5 Windows PowerShell 4.0

Kolejna wersja Windows PowerShell była preinstalowana w systemach Windows 8.1 i Windows Server 2012 R2. Istnieje możliwość instalacji nowej wersji powłoki na systemach takich jak Windows 7, Windows Server 2012, Windows Server 2008 R2 poprzez zainstalowanie paczki aktualizacji Windows Management Framework 4.0. Oto kilka nowości jakich doczekała się kolejna wersja:

- Wsparcie dla debugowania skryptów wykonywanych zdalnie.
- Dodanie zmiennej potokowej (ang. PipelineVariable) jako wspólnego parametru. Umożliwiło to odnoszenie się do aktualnej wartości przesyłanej potokowo.

- Udoskonalone wsparcie dla możliwych do pobrania zestawów pomocy, które można uzyskać komendami Save-Help i Update-Help. Szczególnie pomocne w przypadku korzystania z Windows PowerShell w trybie offline.
- Windows PowerShell Desired State Configuration – nowa platforma zarządzania pozwalająca na wprowadzanie oraz zarządzanie konfiguracjami programów i usług. Nie jest to pojedynczy element PowerShella, tylko rozbudowana platforma składająca się z wielu zasobów odpowiadających za różne funkcjonalności.
- Łatki naprawiające pomniejsze błędy oraz poprawiające wydajność [9].

1.6 Windows PowerShell 5.0

Kolejną i jednocześnie najnowszą wersję powłoki Windows PowerShell można pobrać wraz z aktualizacją Windows Management Framework 5.0. Wraz z tą aktualizacją użytkownik otrzymuje nowe możliwości:

- Zarządzanie archiwami .ZIP poprzez nowe komendy Cmdlet.
- Możliwość zarządzania siecią poprzez nowe zestawy poleceń Cmdlet.
- Synchronizacja ustawień na różnych komputerach poprzez wykorzystanie DSC (ang. Desired State Configuration).
- Możliwość porównywania, aktualizacji oraz publikowania konfiguracji DSC [5,12].

Tabela 1. Dostępność kolejnych wersji PowerShella na systemach Windows [2, 3, 12].

Wersja systemu Windows	Wersja Windows PowerShell				
	1.0	2.0	3.0	4.0	5.0
Windows XP	I, SP2	I, SP3			
Windows Server 2003	W	I, SP2			
Windows Vista	W	I, SP1			
Windows Server 2003		I, SP2			
Windows Server 2008	O	I	I, SP2		
Windows 7		W	I	I, SP1	I, SP1
Windows Server 2008 R2		W	I, SP1	I, SP1	I, SP1
Windows Server 2012			W	I	I
Windows Server 2012 R2				W	I
Windows 8			W		
Windows 8.1				W	I, V
Windows 10					W

W – wersja PowerShella wbudowana w system operacyjny.

I – wymaga zainstalowania.

SP1, SP2, SP3 – wymaga zainstalowania dodatku Service Pack 1, 2, 3.

O – opcjonalny komponent.

V – wsparcie tylko dla wersji Professional oraz Enterprise.

2 Podstawy pracy w PowerShellu

W tym rozdziale zostanie przedstawione wiele aspektów związanych bezpośrednio z pracą w konsoli PowerShell. Wyjaśnione zostaną takie zagadnienia jak profile, kwestie bezpieczeństwa, interfejs, czy uzyskiwanie pomocy. Ponadto przedstawione zostaną rodzaje komend, wartości oraz instrukcji począwszy od podstawowych poleceń typu Cmdlet przez wartości stałe oraz zmienne, aliasy, operatory i ich rodzaje, strumienie, potoki, instrukcje warunkowe, pętle, tablice, a kończąc na bardziej skomplikowanych zagadnieniach takich jak funkcje i skrypty. Wszystkie przykłady zostały stworzone w Windows PowerShell wersji 4.0.

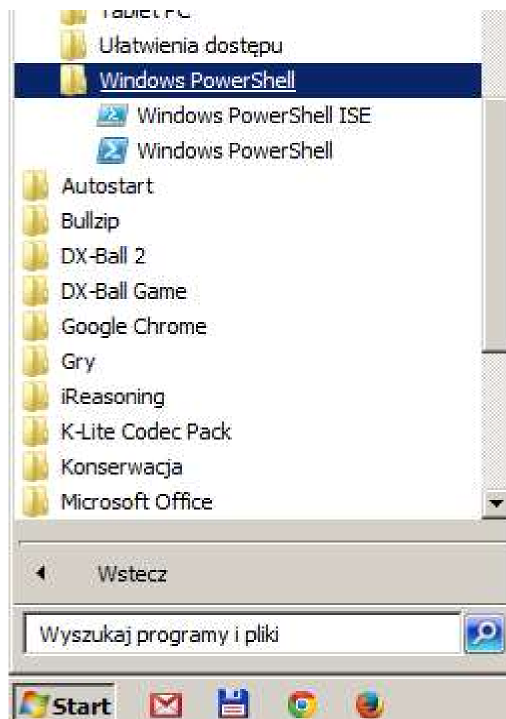
2.1 Uruchamianie PowerShella

W zależności od systemu operacyjnego Windows PowerShell można uruchomić używając do tego różnych sposobów począwszy od otwarcia pliku *powershell.exe* z głównego katalogu PowerShella, a na skrótach skończywszy. We wszystkich wersjach systemów operacyjnych katalogiem głównym, w którym znajduje się Windows PowerShell, jest:

%systemroot%\System32\WindowsPowerShell\v1.0[3]

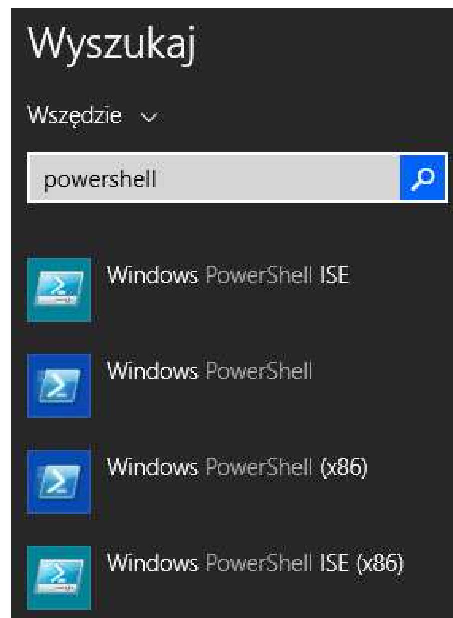
Stała środowiskowa %systemroot% przechowuje lokalizację folderu Windows. Na poniższych rysunkach zostanie przedstawiona lokalizacja skrótów do PowerShella na systemach operacyjnych: Windows 7, Windows 8.1 oraz Windows 10:

3 Niezależnie od zainstalowanej wersji PowerShella, w nazwie katalogu głównego zawsze widnieje wersja 1.0. W wyżej wymienionej lokalizacji znajduje się 64-bitowa wersja PowerShella, natomiast wersję 32-bitową można znaleźć w: %systemroot%\SysWoW64\WindowsPowerShell\v1.0



Rysunek 2.1. W systemie Windows 7 skrót do PowerShella znajduje się w menu start w katalogu Wszystkie Programy\ Akcesoria

Rysunek 2.2. W Windows 8.1 skróty do konsoli PowerShell oraz ISE znajdują się domyślnie jako kafelki na ekranie powitalnym

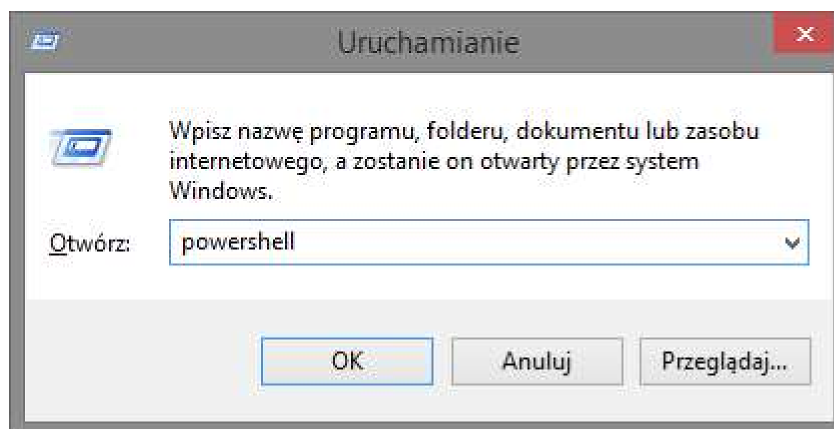


Rysunek 2.3. Po wpisaniu w wyszukiwarkę ekranu powitalnego Windowsa 8.1 słowa *powershell* otrzymujemy skróty do wszystkich interesujących nas plików.

Rysunek 2.4. W systemie Windows 10 skrót do PowerShella znajduje się na pasku startowym w kategorii Wszystkie aplikacje

W każdej wersji systemu Windows istnieje również możliwość uruchomienia konsoli PowerShell przez wpisanie *powershell* w konsolę cmd lub w okno „Uruchom".

Rysunek 2.5. Uruchomienie PowerShella w konsoli cmd



Rysunek 2.6. Wywołanie konsoli PowerShell wykorzystując okno „Uruchom"

Windows PowerShell podobnie jak wiele innych programów może zostać uruchomiony z pewnymi parametrami. W tym celu należy uruchomić wiersz poleceń cmd lub bezpośrednio w konsoli PowerShell wywołać kolejną sesję programu PowerShell dopisując do niego parametry poprzedzone myślnikiem. Można również utworzyć skrót do PowerShella dodając na końcu linii interesujące nas parametry. Oto lista najważniejszych parametrów jakie można zastosować:

Tabela 2. Najważniejsze parametry wywołania konsoli PowerShell.

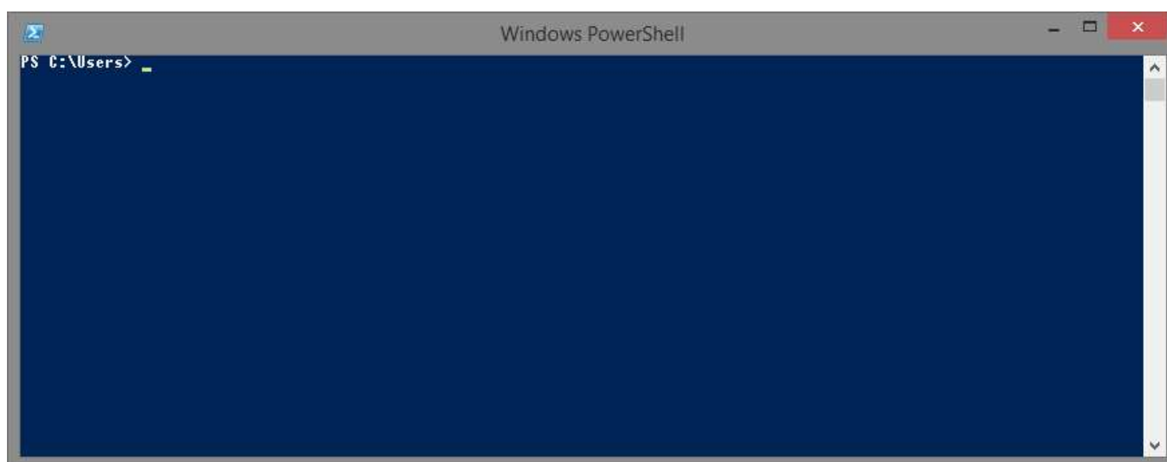
Parametr:	Wartość parametru:	Opis:
-Version	1.0, 2.0, 3.0, 4.0	Uruchamia określoną wersję PowerShella. Uruchomienie wcześniejszej wersji nakłada ograniczenia adekwatne do wersji.
-NoLogo	brak	Ukrywa komunikat o prawach autorskich wyświetlany po uruchomieniu PowerShella.
-Command	<polecenie typu cmdlet>	Uruchamia określoną komendę tak, jakby została wpisana w PowerShellu, następnie zamyka sesję PowerShella
-NoExit	brak	Zapobiega zamknięciu sesji PowerShella po wykonaniu poleceń przez parametr *-Command* .
-NonInteractive	brak	Uruchamia konsolę w trybie nieinteraktywnym. Wprowadzanie danych przez takie komendy jak *Read-Host* jest zablokowane.

-NoProfile	brak	Uruchamia sesję PowerShell z pominięciem ustawień zawartych w profilach.
-WindowStyle	Normal, Minimized, Maximized, Hidden	Ustawia właściwość okna jako: zmaksymalizowane, zminimalizowane, ukryte, normalne.
-ExecutionPolicy	Restricted, Allsigned, RemoteSigned, Unrestricted, Bypass, Undefined	Ustawia politykę zabezpieczeń przed uruchamianiem skryptów w bieżącej sesji.
-File	<ścieżka i nazwa skryptu>	Uruchamia skrypt w zasięgu lokalnym wykorzystując uruchamianie z kropką. Dokładne opisanie tego zagadnienia znajduje się w rozdziale poświęconym skryptom.

Aby wyjść z konsoli należy wcisnąć przycisk *zamknij*, oznaczony jako X na pasku aktywnego okna. Ponadto można opuścić bieżącą sesję konsoli używając polecenia `exit`.

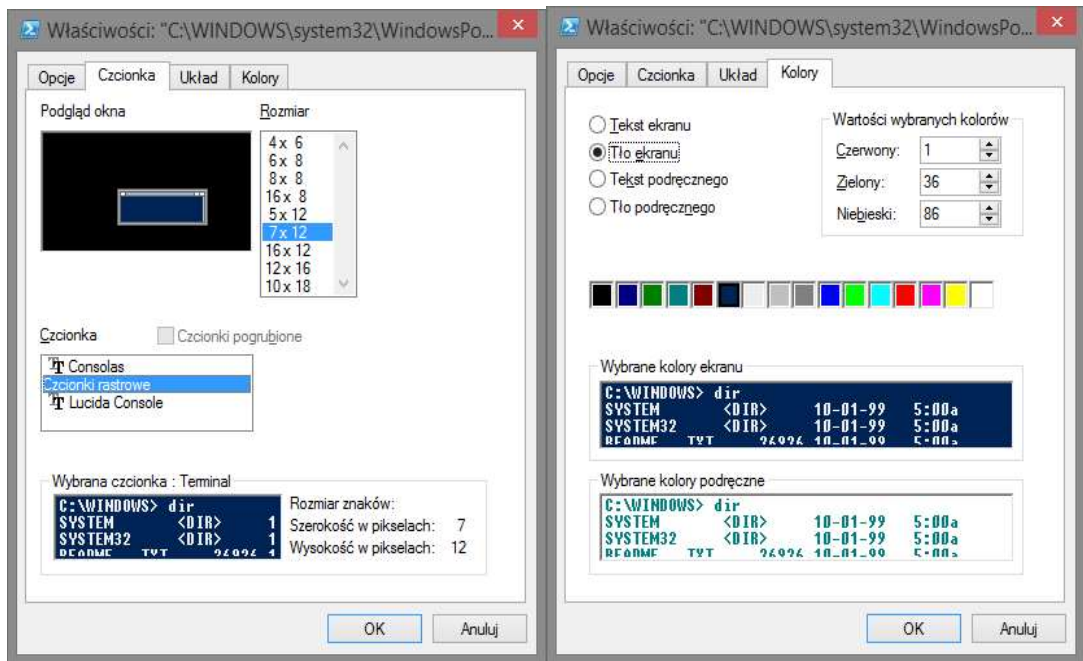
2.2 Interfejs użytkownika oraz profile

Okno konsoli Windows PowerShell nie różni się wiele od okna klasycznego wiersza poleceń.



Rysunek 2.7. Widok konsoli Windows PowerShell

Największa różnica tkwi w strukturze komend oraz sposobie tworzenia skryptów. Konsolę można w pewien sposób modyfikować. Klikając prawym klawiszem myszy na pasku aktywnego okna oraz wybrać z menu kontekstowego pozycję „*Właściwości*".



Rysunek 2.8. Widok przykładowych opcji okna konsoli.

Dostępne opcje pozwalają na konfigurację podstawowych ustawień taki jak: rozmiar okna, rodzaj oraz pogrubienie czcionki, kolory tekstu oraz tła, położenie okna z dokładnością jednego piksela, rozmiar i wielkość bufora.

Windows PowerShell pozwala na personalizowanie wielu ustawień poprzez wykorzystanie profili. Profil w rozumieniu PowerShella to skrypt, który jest wykonywany zaraz po uruchomieniu konsoli. W skrypcie tym, użytkownik może zdefiniować takie wartości jak osobiste aliasy, pule zmiennych oraz stałych, ustawienia wizualne konsoli i wiele innych. Istnieje 6 różnych poziomów profili:

Tabela 3. Profile w Windows PowerShell [15].

Lp.	Dane profili:	
1	Nazwa:	$PROFILE.CurrentUserCurrentHost
	Dotyczy:	Bieżącego użytkownika, bieżącego rodzaju konsoli
	Ścieżka:	$Home\Documents\WindowsPowerShell\Profile.ps1
2	Nazwa:	$PROFILE.CurrentUserAllHosts
	Dotyczy:	Bieżącego użytkownika, wszystkich rodzajów konsoli
	Ścieżka:	$Home\Documents\Profile.ps1
3	Nazwa:	$PROFILE.AllUsersCurrentHost
	Dotyczy:	Wszystkich użytkowników, bieżącego rodzaju konsoli
	Ścieżka:	$PsHome\Microsoft.PowerShell_profile.ps1

4	Nazwa:	$PROFILE.AllUsersAllHost
	Dotyczy:	Wszystkich użytkowników, wszystkich rodzajów konsoli
	Ścieżka:	$PsHome\Profile.ps1
5	Nazwa:	$PROFILE.CurrentUserCurrentHost
	Dotyczy:	Bieżącego użytkownika i bieżącego rodzaju konsoli w Intergrated Script Envorinment
	Ścieżka:	$Home\Documents\WindowsPowerShell\Microsoft.P owerShellISE_profile.ps1
6	Nazwa:	$PROFILE.AllUsersCurrentHost
	Dotyczy:	Wszystkich użytkowników i bieżącego rodzaju konsolii w Intergrated Script Envorinment
	Ścieżka:	$PsHome\Microsoft.PowerShellISE_profile.ps1

Poprzez słowo *Host* w kontekście PowerShella rozumiemy osobne pliki konsoli nie będące tą samą odmianą jednego pliku *.exe. Domyślnie zainstalowanymi różnymi wersjami konsoli są: konsola PowerShell oraz konsola ISE. Profile 5 oraz 6 dotyczą profili PowerShell ISE. Zmienna `$Home` zawiera domyślny katalog użytkownika, zwykle jest to:

```
C:\Users\<nazwa_użytkownika>
```

natomiast zmienna `$PsHome` przechowuje ścieżkę głównego katalogu, w którym jest zainstalowany PowerShell, domyślnie:

```
C:\Windows\System32\WindowsPowerShell\v1.0
```

Domyślnie pliki profili nie istnieją. Aby z nich korzystać należy je najpierw stworzyć. Można do tego użyć wewnętrznej komendy PowerShella lub stworzyć plik o rozszerzeniu *.ps1 zachowując jego odpowiednią nazwę oraz lokalizację. Domyślnym profilem uruchamianym przez użytkownika w momencie włączenia konsoli jest profil *$Profile.CurrentUserCurrentHost*. Aby stworzyć plik profilu należy posłużyć się komendą:

```
New-Item $PROFILE.CurrentUserCurrentHost
-ItemType file -Force
```

Parametr `-Force` powoduje nadpisanie pliku profilu w przypadku gdyby już istniał. Aby rozpocząć edycję swojego profilu należy otworzyć plik z profilem. Można do tego użyć każdego edytora tekstu, na przykład notatnika, chociaż zalecane jest używanie ISE, ponieważ podkreśla składnię, posiada system podpowiadania oraz wiele innych ułatwień. Plik profilu można otworzyć wpisując w konsolę polecenie:

```
ise $Profile.CurrentUserCurrentHost
```

lub:

```
notepad $Profile.CurrentUserCurrentHost
```

To polecenie wywoła sesję Windows PowerShell ISE lub notatnika z otwartym skryptem profilu. We wszystkich profilach można zamieścić specjalną funkcję o nazwie `Prompt`, która służy do personalizacji linii poleceń. Funkcja ta zwraca wartość typu `String`, która umieszczana jest w każdej nowej linii przed znakiem zachęty. Wszystkie profile dotyczące okna konsoli PowerShell, o ile zostały stworzone, wczytywane są jeden za drugim, od profilu o największym zasięgu do profilu o najmniejszym zasięgu. W przypadku edytowania tych samych zmiennych w wielu profilach, PowerShell zastosuje ustawienia zawarte w profilu o najmniejszym zasięgu. Kolejność wczytywania profili konsoli wygląda następująco:

1) Wszyscy użytkownicy, wszystkie rodzaje konsoli.
2) Wszyscy użytkownicy, bieżący rodzaj konsoli.
3) Bieżący użytkownik, wszystkie rodzaje konsoli.
4) Bieżący użytkownik, bieżący rodzaj konsoli.

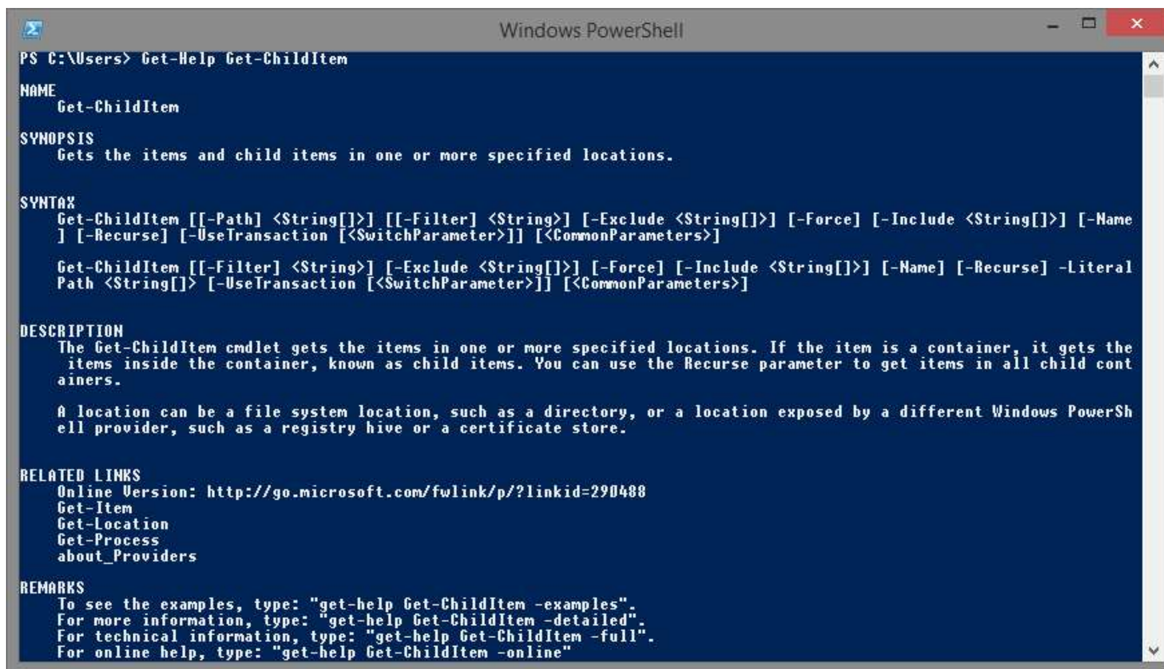
Należy pamiętać, że profile w PowerShell to tak naprawdę skrypty. Aby mieć możliwość uruchomienia skryptów należy zastosować odpowiednią politykę zabezpieczeń, którą szerzej wyjaśnię w podrozdziale poświęconym bezpieczeństwu w Windows PowerShell.

2.3 Uzyskiwanie pomocy w Windows PowerShell

Windows PowerShell dysponuje bardzo obszernym działem pomocy. Można w łatwy sposób dowiedzieć się wszystkiego, czego użytkownik może potrzebować korzystając z polecenia `Get-Help` a następnie wpisując komendę, na temat której chcemy uzyskać pomoc, na przykład polecenie:

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

wyświetli informacje na temat polecenia `Get-ChildItem`.



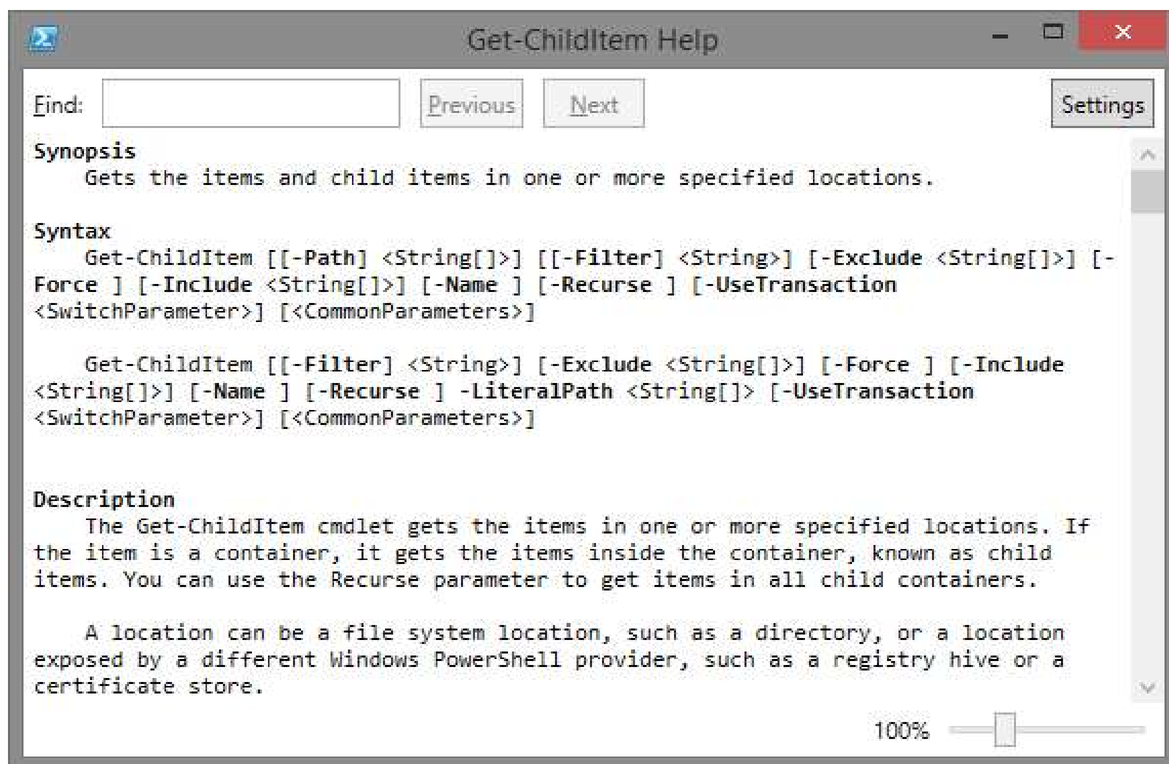
Rysunek 2.9. Standardowe okno pomocy na temat polecenia `Get-ChildItem`

To, czym wyróżnia się pomoc oferowana przez Windows PowerShell na tle innych powłok, jest sposób oraz ilość wyświetlanych informacji. Po wywołaniu komendy otrzymujemy streszczenie działania polecenia, pełną składnię z listą parametrów oraz rodzajem wartości jakie owe parametry przyjmują, pełny opis polecenia, linki do innych, powiązanych plików pomocy mogących zawierać dodatkowe informacje oraz listę parametrów z jakimi można wywołać komendę `Get-Help`, celem otrzymania bardziej szczegółowych informacji. Oto lista parametrów jakich można użyć [4]:

- Detailed – wyświetla rozszerzoną, bardziej szczegółową wersję pomocy wraz z dokładnym opisem możliwych do użycia parametrów oraz listą przykładów opisami ich działania.
- Full – najbardziej rozbudowana wersja pomocy jaką można uzyskać, powiększona o formaty wejściowe i wyjściowe danych z jakich korzysta dane polecenie oraz dokładniejszy opis parametrów jakich można użyć.
- Examples – ten parametr spowoduje wyświetlenie wyłącznie listę przykładów użycia polecenia.
- Online – użycie tego parametru wywoła okno przeglądarki na stronie internetowej firmy Microsoft *www.technet.microsoft.com* wraz ze szczegółową pomocą na temat

poszukiwanego polecenia. Windows PowerShell, aby skorzystać z tego parametru, wymaga połączenia z internetem.

- ShowWindow – ten parametr spowoduje, że cały tekst pomocy, zamiast w oknie konsoli, wyświetlony zostanie w osobnym oknie. Treść pomocy będzie taka sama jak w przypadku użycia parametru `-Full`.



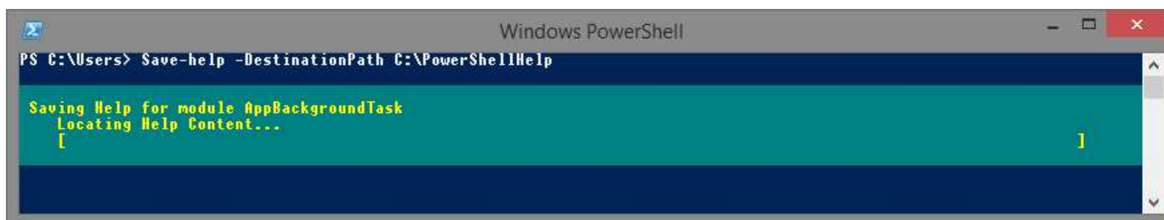
Rysunek 2.10. Tekst pomocy wyświetlony w osobnym oknie

W Windows PowerShell znajduje się domyślnie kompletny zbiór pomocy dotyczący podstawowych modułów, komend i bazowej zawartości dostarczanej z bieżącą wersją powłoki. Od wersji 3.0 istnieje możliwość aktualizowania plików pomocy przez internet bezpośrednio z konsoli. Nowo pobrane wersje plików pomocy zostają zainstalowane w podfolderze *Modules* katalogu głównego. Raz pobrane pliki pomocy można zapisać wskazując inną lokalizację, na przykład wspólny folder sieciowy. Służą do tego polecenia: `Update-Help`, które pobiera pliki pomocy oraz `Save-Help`, które oferuje możliwość zapisania plików pomocy na dysku, gdzie później będzie można ich użyć do aktualizowania pomocy na komputerach w sieci lokalnej bez dostępu do internetu. Oto przykład ich zastosowania:

```
Save-Help -DestinationPath C:\PowerShellHelp
```

Polecenie domyślnie zapisze wszystkie pliki pomocy w lokalizacji C:\PowerShellHelp. Następnie będzie można z tego miejsca zaktualizować pomoc przez użycie komendy [4]:

```
Update-Help -SourcePath C:\PowerShellHelp
```



Rysunek 2.11. Zapisywanie pomocy do pliku przy użyciu polecenia `Save-Help`

2.4 Bezpieczeństwo w PowerShell

Z racji niezwykle rozbudowanych możliwości PowerShella kwestia bezpieczeństwa jest tutaj bardzo istotna. Największym problemem środowisk takich jak PowerShell są skrypty. O ile wykonywanie pojedynczych poleceń zwykle przebiega zgodnie z intencjami użytkownika, o tyle w skryptach, dopóki dokładnie nie przeanalizujemy zawartości, dopóty nie możemy być pewni tego, jaki skutek wywoła wykonanie gotowego zestawu poleceń. Aby zapobiec tego typu wydarzeniom Windows PowerShell posiada wbudowany w powłokę system zabezpieczeń uniemożliwiający uruchomienie jakichkolwiek skryptów bez wcześniejszego zezwolenia użytkownika. Ponadto w PowerShellu możemy odnaleźć pełne zestawy reguł uruchamiania skryptów, które dzielą się w następujący sposób:

- Zastrzeżony (ang. Restricted) – domyślnie ustawiony zestaw reguł dotyczących zabezpieczeń w PowerShell. Jest on najbardziej rygorystyczny, pozwala na uruchamianie tylko wybranych komend.
- Wymagający podpisu cyfrowego (ang. AllSigned) – jest to zestaw, w którym można uruchamiać tylko te skrypty, które zostały podpisane cyfrowo przez zaufanego wydawcę.
- Wymagający podpisu cyfrowego dla skryptów spoza lokalnego komputera (ang. RemoteSigned) – pozwala na wykonywanie skryptów powstałych na lokalnym komputerze, jednak blokuje skrypty z innych źródeł (wymagają one podpisania własnym certyfikatem).

- Niestrzeżony (ang. Unrestricted) – zestaw reguł zapewniający bardzo niski poziom bezpieczeństwa. Pozwala na uruchomienie skryptów dowolnego pochodzenia. Jedynym środkiem ochrony jest stosowny komunikat informujący o niebezpieczeństwach związanych z wykonywaniem skryptów pochodzących z nieznanych źródeł.
- Przechodzący (ang. Bypass) – ten zestaw reguł nie blokuje niczego, nie zwraca też żadnych komunikatów informacyjnych. Stosowany jest do konfiguracji PowerShella oraz czasem jest wykorzystywany w przypadku programów, które do podstawowego działania wykorzystują powłokę PowerShell.
- Niezdefiniowany (ang. Undefined) – niezdefiniowany zestaw reguł, który domyślnie przyjmuje wartości polityki zastrzeżonej [8].

Poza możliwością ustawienia zasad wykonywania skryptów Windows PowerShell umożliwia stosowanie ich na różnych poziomach działania. Dzięki temu użytkownik może określić jaki zestaw reguł będzie obowiązywał w danym zakresie działania, które dzielimy na:

- Zakres komputera (ang. MachinePolicy) – wykorzystując zasady grupy (ang. Group Policy, GPO) zestaw reguł przypisywany jest dla wszystkich użytkowników danego komputera.
- Zakres użytkownika (ang. UserPolicy) – reguły w tym przypadku będą dotyczyć określonego użytkownika.
- Zakres bieżącej sesji (ang. Process) – zestaw reguł dotyczyć będzie wyłącznie aktualnej sesji PowerShella.
- Zakres bieżącego użytkownika (ang. CurrentUser) – podobnie jak w przypadku zakresu użytkownika z tą różnicą, że zasady tyczyć się będą obecnie zalogowanego użytkownika.
- Zakres komputera lokalnego (ang. LocalMachine) – zestaw reguł stosowany w zakresie całej maszyny, dla wszystkich jej użytkowników. Jest to domyślnie stosowana reguła przy nakładaniu zasad wykonywania skryptów [8].

Aby uzyskać informacje dotyczące poleceń związanych z ustawianiem oraz uzyskiwaniem informacji o regułach zabezpieczeń bezpośrednio w konsoli Windows PowerShell należy używać komend z rodziny `ExecutionPolicy`:

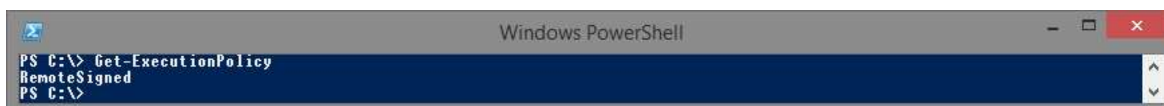
```
Get-Command *ExecutionPolicy
```

Rysunek 2.12. Polecenia służące do zarządzania polityką zabezpieczeń w Windows PowerShell.

Wywołanie komendy:

```
Get-ExexcutionPolicy
```

spowoduje wyświetlenie aktualnie funkcjonującej polityki zabezpieczeń.



Rysunek 2.13. Pozyskiwanie informacji na temat bieżącego systemu zabezpieczeń.

Zastosowanie parametru `-List` pozwala na wyświetlenie aktywnych zestawów reguł na wszystkich możliwych zakresach.

```
Get-ExecutionPolicy -List
```

Rysunek 2.14. Informacje dotyczące reguł bezpieczeństwa w różnych zakresach

2.5 Składnia podstawowych poleceń

Windows PowerShell dysponuje nowym, a zarazem potężnym zestawem komend jakimi są „*Commandlets*". Stosuje się również nazwę skróconą w postaci *cmdlets*. Ten typ poleceń składa się z:

1. Czasownika.
2. Rzeczownika.
3. (Opcjonalnie) Listy parametrów.

Czasownik oraz rzeczownik oddzielone są znakiem „-", natomiast parametry, jeśli występują, oddziela znak spacji. Wielkość liter stosowana do zapisywania poleceń nie jest istotna. Wzór na składnię poleceń wygląda zatem następująco:

```
czasownik-rzeczownik [-parametry]
```

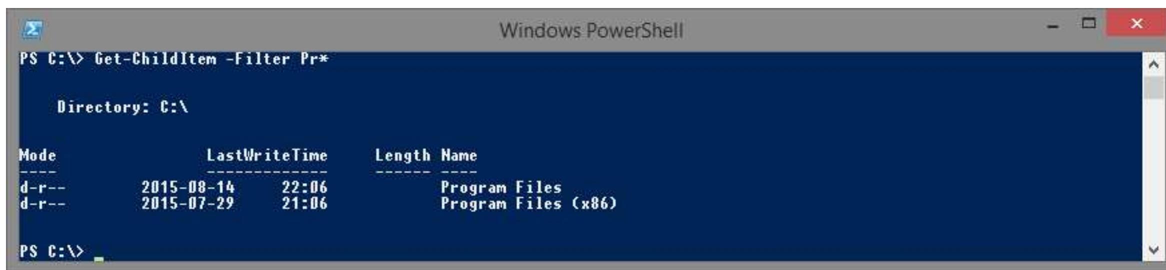
Aby skorzystać z systemu podpowiedzi podczas wywoływania poleceń wystarczy wpisać pierwszą bądź kilka pierwszych liter interesującej nas komendy, a następnie wcisnąć klawisz *„Tab"*. Spowoduje to uzupełnienie polecenia o pierwszy wynik odpowiadający wpisanym wcześniej znakom. W przypadku wciskania klawisza *„Tab"* kilkukrotnie Windows PowerShell będzie sugerował następne komendy, pasujące do wpisanych wcześniej znaków, w kolejności alfabetycznej. Oto przykład zastosowania komendy, dzięki której konsola wyświetli zawartość bieżącego katalogu:

```
Get-ChildItem
```

Rysunek 2.15. Zastosowanie polecenia *Get-ChildItem*.

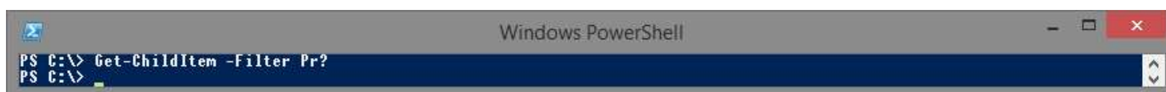
Korzystając z parametrów tego polecenia można między innymi w łatwy sposób przefiltrować wyniki pod kątem poszukiwanych przez nas plików. Przykładem takiego parametru jest `-Filter`, który wyświetli tylko wyniki odpowiadające znakom podanym po parametrze. Jeśli chcemy odszukać wszystkie pliki zaczynające się od liter „Pr" wystarczy użyć polecenia:

```
Get-ChildItem -Filter Pr*
```



Rysunek 2.16. Polecenie `Get-ChildItem` z wykorzystaniem parametru `-Filter`.

Użycie znaku „*" odpowiada wszystkim możliwym kombinacjom ciągu znaków, natomiast „?" odpowiada za jeden dowolny znak.



Rysunek 2.17. Użycie znaku „?" nie zwraca żadnych wyników, ponieważ bieżący katalog nie posiada plików odpowiadających zadanemu warunkowi.

W przypadku wywołania polecenia bez używania nazw parametrów, wartości wymienione za poleceniem przypisywane są parametrom w domyślnej kolejności. W przypadku polecenia `Get-ChildItem` pierwszym parametrem jest ścieżka z której będą wyświetlane pliki, a następnie filtr nazwy pliku. Zatem komenda:

```
Get-ChildItem C:\Windows *.exe
```

działa tak samo jak polecenie

```
Get-ChildItem -Path C:\Windows -Filter *.exe
```

które wyświetli wszystkie pliki typu „.exe" znajdujące się w lokalizacji „C:\Windows". Jeśli zamierzamy używać folderów lub plików które posiadają w nazwie spację, należy umieścić nazwę pliku lub folderu w cudzysłowie, na przykład:

```
Get-ChildItem -Path "C:\Program Files" -Filter *.exe
```

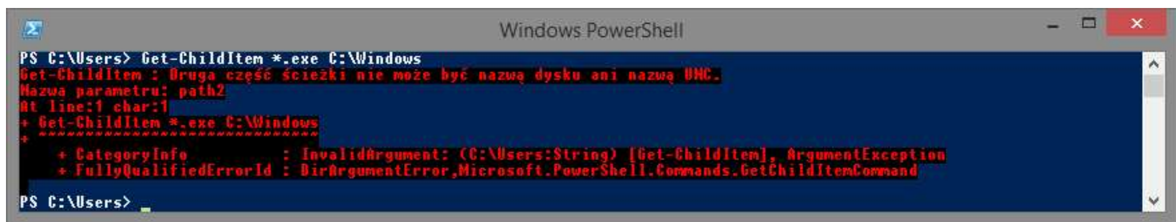
W przypadku używania nazw parametrów przed ich wartościami można zmienić kolejność ich wywoływania:

```
Get-ChildItem -Filter *.exe -Path C:\Windows
```

natomiast jeśli posłużymy się taką kolejnością bez używania nazw parametrów:

```
Get-ChildItem *.exe C:\Windows
```

otrzymamy komunikat informujący o błędnej składni polecenia ponieważ parametry nie zostały podane w domyślnej kolejności [3].



Rysunek 2.18. Użycie parametrów polecenia `Get-ChildItem` w niepoprawnej kolejności

2.6 Aliasy

Aliasy to skrócone wersje poleceń. Używa się ich, aby zaoszczędzić sobie czasu pisania długich i czasem skomplikowanych poleceń. Istnieje określona ilość predefiniowanych bezpośrednio przez Windows PowerShell aliasów. Aby wywoływać ich listę należy użyć polecenia:

```
Get-Alias
```

PowerShell pozwala również na tworzenie własnych aliasów. Można to zrobić korzystając z poleceń *New-Alias* lub *Set-Alias*. Oto przykład ich zastosowania:

```
New-Alias policy Get-ExecutionPolicy
Set-Alias policy Get-ExecutionPolicy
```

Istnieje jedna, aczkolwiek istotna różnica pomiędzy tymi poleceniami: komenda *New-Alias* tworzy nowy alias, jednak w przypadku gdy alias o takiej nazwie już istnieje użytkownik otrzyma komunikat z błędem informującym o zaistniałej sytuacji, natomiast użycie komendy *Set-Alias* tworzy nowy alias lub nadpisuje już istniejący. Aliasy można stosować nie tylko do poleceń, można je również użyć, jako skrót do uruchamiania programów, na przykład:

```
New-Alias kal calc.exe
```

Wyżej wymienione polecenie tworzy alias *kal* który uruchomi systemowy kalkulator. Windows PowerShell nie sprawdza poprawności lokalizacji ani nazwy pliku. Jeśli alias nie został zdefiniowany poprawnie, komunikat o błędzie zostanie wyświetlony dopiero w momencie pierwszego użycia aliasu. Nowo utworzone aliasy ważne są tylko na czas aktywnej sesji PowerShell. Można jednak wyeksportować wszystkie aliasy a następnie wczytać je podczas nowej sesji PowerShell. Spis aliasów można przechowywać w

formacie skryptu Windows PowerShell *.ps1 lub w formacie *.csv[4]. Służą do tego polecenia:

```
Export-Alias [ścieżka docelowa pliku]\[nazwa pliku].csv
```

lub:

```
Export-Alias [ścieżka docelowa pliku]\[nazwa pliku].ps1
-as Script
```

Aby wczytać wcześniej zapisany zestaw aliasów możemy użyć poleceń:

```
Import-Alias [ścieżka docelowa pliku]\[nazwa pliku].csv
```

w przypadku rozszerzenia „.csv" lub:

```
. [ścieżka docelowa pliku]\[nazwa pliku].ps1
```

jeśli formatem pliku docelowego był „.ps1".

Ilość możliwych do przechowania aliasów jest ograniczona. Wartość ta domyślnie wynosi 4096 i jest przechowywana w zmiennej *$MaksimumAliasCount*. Na szczęście można ją zmienić edytując wartość wcześniej wymienionej zmiennej [3].

4 CSV (ang. Comma-Separated Values) – jest to sposób przechowywania danych używając do tego plików tekstowych polegający na oddzielaniu kolejny wartości przecinkami. [10]

Tabela 4. Najważniejsze predefiniowane aliasy w Windows PowerShell [11].

Polecenie w PowerShell	Alias	Odpowiednik w:		Opis
		CMD	Unix	
Get-ChildItem	ls,gci,dir	dir	ls	Wyświetla pliki i foldery w bieżącym katalogu
Get-Content	type,gc,cat	type	cat	Wyświetla zawartość pliku
Get-Command	gcm	help	type, apropos, which	Wyświetla listę dostępnych komend
Get-Help	man,help	help	man	Wyświetla pomoc do wybranej komendy
Clear-Host	clear, cls	cls	clear	Czyści ekran konsoli
Copy-Item	cp, copy, cpi	copy	cp	Kopiuje pliki lub katalogi
Move-Item	move , mv, mi	move	mv	Przenosi plik lub katalog w wybrane miejsce
Remove-Item	del, rmdir, erase, rd, rm, ri	rmdir, erase, rd, del	rm, rmdir	Usuwa wybrany plik lub katalog
Rename-Item	ren, rv, rni	rename, ren	mv	Zmienia nazwę pliku lub katalogu
Get-Location	cd, pwd, gl	cd	pwd	Wyświetla bieżący katalog
Get-Process	ps, gps	tasklist	ps	Wyświetla działające procesy
Stop-Process	kill, spps	taskkill	kill	Zatrzymuje wybrany proces
Select-String	sls	find, findstr	grep	Wyświetla ciąg znaków odpowiadający zadanemu warunkowi
Set-Variable	set, sv	set	export, env, set, setenv	Tworzy i ustawia wartość zmiennej
Write-Host	write, echo	echo	echo	Wyświetla ciąg znaków lub wartości zmiennych

2.7 Zmienne, zmienna potokowa, zmienne automatyczne, stałe oraz znaki specjalne

Wartości występujące w PowerShell pojawiają się jako zmienne lub stałe. Każda z wyżej wymienionych posiada określony typ. Oto lista podstawowych typów stałych oraz zmiennych:

Tabela 5. Podstawowe typy zmiennych.

Typ:	Opis:
[int] lub [int32]	32-bitowa liczba całkowita
[long] lub [int64]	64-bitowa liczba całkowita
[string]	Ciąg znaków, łańcuch
[char]	16-bitowy znak zapisany w kodowaniu Unicode
[byte]	8-bitowa liczba całkowita bez znaku
[bool]	Zmienna typu prawda/fałsz
[single]	32-bitowa liczba zmiennoprzecinkowa, pojedynczej precyzji
[double]	64-bitowa liczba zmiennoprzecinkowa, podwójnej precyzji
[decimal]	128-bitowa liczba zmiennoprzecinkowa, poczwórnej precyzji
[xml]	Typ xml
[array]	Typ tablicy
[hastable]	Tablica mieszająca[5]

Zmienne

Zmienne pozwalają przechowywać pewne wartości, z których będzie można skorzystać później. Nazwa zmiennej w Windows PowerShell zawsze poprzedzana jest znakiem „$". Przykład definiowania zmiennej:

```
$wynik = 4
```

Po wywołaniu zmiennej otrzymamy wartość jaką przechowuje:

```
$wynik
4
```

Zmienne mogą mieć w swoim zapisie cyfry, mogą zaczynać się od cyfr oraz posiadać polskie znaki w zapisie:

```
$1Koło = "Koło"
```

5 Tablica mieszająca (ang. Hashtable) - specjalny rodzaj tablicy przechowujący dane w postaci pary: unikatowego klucza oraz wartości. Pozwala na odniesienie się do wartości poprzez podanie klucza w indeksie tablicy.

```
$1Koło
Koło
```

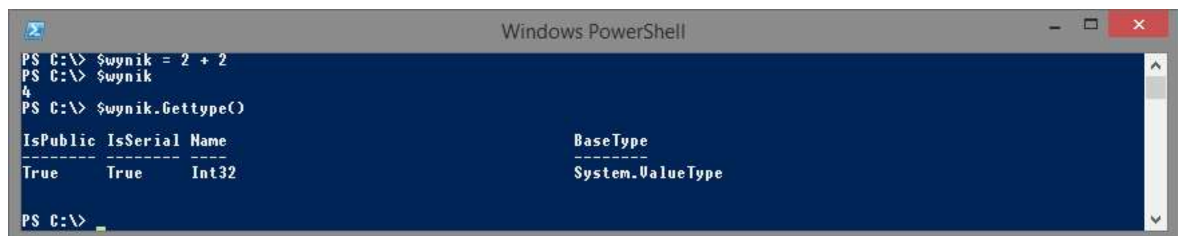
Typ zmiennej można zdefiniować na kilka sposobów. Najprostszym a zarazem najbardziej intuicyjnym rozwiązaniem jest automatyczne przypisywanie typu w momencie podawania wartości zmiennej, może to być liczba całkowita, liczba zmiennoprzecinkowa, ciąg znaków lub obiekt. Jeżeli użytkownik chce mimo wszystko definiując zmienną od razu nadać jej typ należy zapisać typ zmiennej w nawiasach kwadratowych przed jej definicją.

```
[Double] $A
[Double] $A = 2
```

Można również skorzystać z rzutowania wartości, aby nadać jej odpowiedni typ w momencie przypisywania wartości. Cały proces jest dokładnie wyjaśniony w podrozdziale poświęconym operatorom. Oto przykład takiego zastosowania:

```
$A = 2 -as [Int32]
```

Aby uzyskać dane na temat typu zmiennej należy wywołać jej metodę „`.GetType()`".



Rysunek 2.19. Przypisywanie wartości zmiennej oraz pozyskiwanie informacji o jej typie.

Windows PowerShell bardzo inteligentnie operuje z wykorzystaniem różnych typów zmiennych. Potrafi, w pewnym zakresie, rzutować typy zmiennych zgodnie z życzeniem użytkownika. Aby zobrazować tę funkcjonalność zdefiniujmy 3 zmienne różnych typów:

```
$zmienna1 = 2
$zmienna2 = "12"
$zmienna3 = 6.5
```

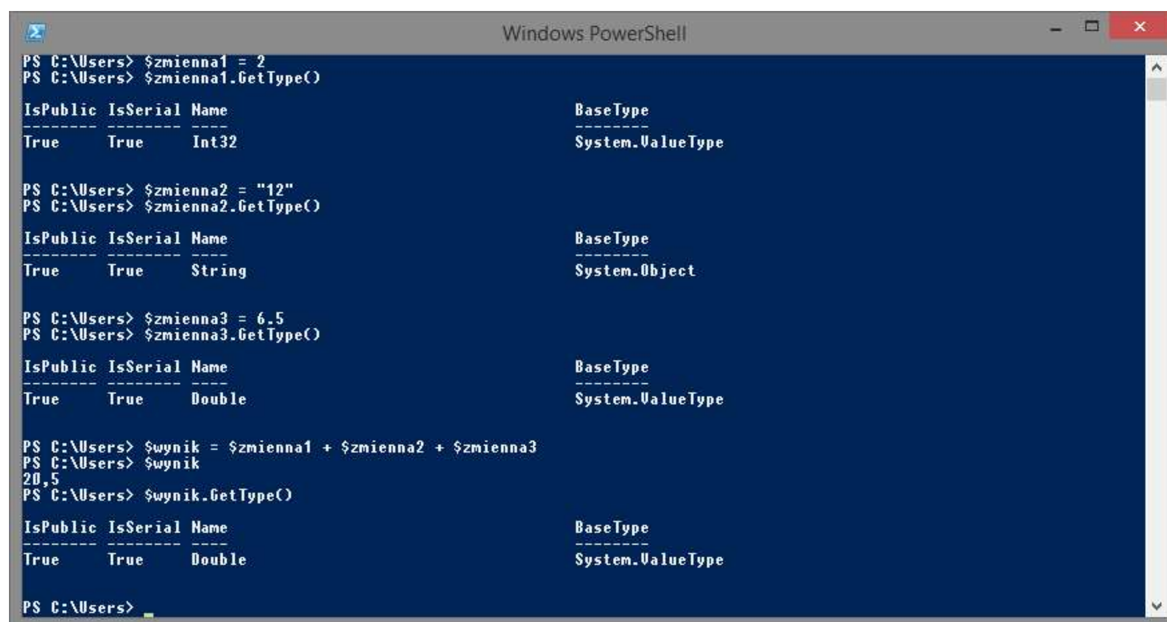
Pierwsza zmienna przyjmie typ *Int32*, ponieważ jej wartość to liczba całkowita, druga zmienna zostanie określona jako *String*, gdyż jej wartością jest ciąg znaków (wartość podana w cudzysłowie odbierana jest jak ciąg znaków), natomiast trzecia zmienna będzie typu *Double*, z racji że jej wartość to liczba zmiennoprzecinkowa. Jeśli po zdefiniowaniu zmiennych będziemy chcieli je zsumować, kompilator w innych językach programowania zwykle powita nas błędem informującym o nie pasujących do siebie zmiennych.

PowerShell jednak wykona automatycznie rzutowanie typów zwracając poprawny wynik. Stąd też komenda:

```
$wynik = $zmienna1 + $zmienna2 + $zmienna3
$wynik
```

zwróci prawidłowy wynik:

```
20,5
```



Rysunek 2.20. Przykład wykorzystania automatycznego rzutowania typu zmiennych

Zmienne można również tworzyć używając do tego polecenia `New-Variable`. Używając takiej składni możemy określić nazwę zmiennej, jej wartość, parametry, widoczność czy zasięg działania.

```
New-Variable -Name Wzrost -Value 180 -Description "Wzrost
podany w  centymetrach"
```

Oprócz podstawowych typów, zmienna może przechowywać bardziej rozbudowane wartości, jak na przykład efekt wywołania poleceń. Można zatem stworzyć zmienną, która na przykład przechowa listę plików i folderów znajdujących się bezpośrednio na dysku C: używając polecenia:

```
$dir = Get-ChildItem -Path C:\
```

```
Windows PowerShell
PS C:\Users> $dir = Get-ChildItem -Path C:\
PS C:\Users> $dir


    Directory: C:\


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        2014-08-19     22:55            AMD
d----        2015-06-25     13:01            Fraps
d----        2014-11-28     12:33            Keyboard
d----        2013-08-22     17:22            PerfLogs
d----        2015-07-22     10:54            PowerShellHelp
d-r--        2015-07-03     13:29            Program Files
d-r--        2015-07-15     22:14            Program Files (x86)
d----        2015-06-08     12:30            Speed
d-r--        2015-07-25     16:05            Users
d----        2015-07-09     15:34            Windows


PS C:\Users>
```

Rysunek 2.21. Przypisanie efektu wywołania polecenia `Get-ChildItem` do zmiennej

Windows PowerShell poza listą zmiennych jakie użytkownik może stworzyć, posiada również listę predefiniowanych zmiennych środowiskowych. Przechowują one informacje dotyczące uruchamianych programów, domyślnych ścieżek, nazw komputera, użytkownika i wiele innych. Aby uzyskać listę wszystkich zmiennych środowiskowych z jakich korzysta Windows PowerShell, należy użyć polecenia:

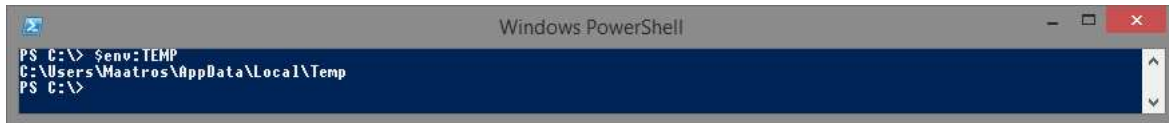
```
Get-ChildItem env:
```

```
Windows PowerShell
PS C:\> Get-ChildItem env:

Name                           Value
----                           -----
ALLUSERSPROFILE                C:\ProgramData
APPDATA                        C:\Users\Admin\AppData\Roaming
CommonProgramFiles             C:\Program Files\Common Files
CommonProgramFiles(x86)        C:\Program Files (x86)\Common Files
CommonProgramW6432             C:\Program Files\Common Files
COMPUTERNAME                   MAATROS-PC
ComSpec                        C:\WINDOWS\system32\cmd.exe
FP_NO_HOST_CHECK               NO
HOMEDRIVE                      C:
HOMEPATH                       \Users\Admin
LOCALAPPDATA                   C:\Users\Admin\AppData\Local
LOGONSERVER                    \\MAATROS-PC
NUMBER_OF_PROCESSORS           4
OS                             Windows_NT
Path                           C:\ProgramData\Oracle\Java\javapath;C:\Program Files (x86)\NVIDIA Corporation\PhysX\C...
PATHEXT                        .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.CPL
PROCESSOR_ARCHITECTURE         AMD64
PROCESSOR_IDENTIFIER           Intel64 Family 6 Model 58 Stepping 9, GenuineIntel
PROCESSOR_LEVEL                6
PROCESSOR_REVISION             3a09
ProgramData                    C:\ProgramData
ProgramFiles                   C:\Program Files
ProgramFiles(x86)              C:\Program Files (x86)
ProgramW6432                   C:\Program Files
PSModulePath                   C:\Users\Admin\Documents\WindowsPowerShell\Modules;C:\Program Files\WindowsPowerShell...
PUBLIC                         C:\Users\Public
SESSIONNAME                    Console
SystemDrive                    C:
SystemRoot                     C:\WINDOWS
TEMP                           C:\Users\Admin\AppData\Local\Temp
TMP                            C:\Users\Admin\AppData\Local\Temp
USERDOMAIN                     Maatros-PC
USERDOMAIN_ROAMINGPROFILE      Maatros-PC
USERNAME                       Admin
USERPROFILE                    C:\Users\Admin
windir                         C:\WINDOWS


PS C:\>
```

Rysunek 2.22. Lista zmiennych środowiskowych w Windows PowerShell

Aby wywołać wartość zmiennej środowiskowej należy jej nazwę poprzedzić przedrostkiem „`env:`" oraz tak jak w przypadku wszystkich zmiennych poprzedzić całość znakiem „$". Oto przykład wywołania zmiennej środowiskowej *TEMP*, która określa położenie folderu, w którym przechowywane są pliki tymczasowe użytkownika:

```
$env:TEMP
```



Rysunek 2.23. Wyświetlenie wartości zmiennej środowiskowej *TEMP*

Aby usunąć wartość zmiennej w Windows PowerShell należy użyć polecenia:

```
Clear-Variable -Name <nazwa_zmiennej>
```

lub ustawić jej wartość na *null*:

```
$<nazwa_zmiennej> = $null
```

Jeśli natomiast chcemy całkowicie usunąć zmienną należy użyć komendy:

```
Remove-Variable -Name <nazwa_zmiennej>
```

Zmienna potokowa

PowerShell dysponuje bardzo użyteczną funkcjonalnością jaką jest zmienna potokowa. Zapis zmiennej potokowej wygląda następująco: „$_". Pozwala ona na odniesienie się do aktualnej wartości przesyłanej przez potok danych. Upraszcza to w wielu przypadkach formę zapisu. Dokładniejsze opisanie zmiennej potokowej znajduje się w podrozdziale poświęconym strumieniowym oraz potokowym przesyłaniu danych.

Zmienne automatyczne

W PowerShellu istnieje pula z góry zdefiniowanych zmiennych, noszących nazwę zmiennych automatycznych, które przechowują informacje o PowerShellu i zostały stworzone specjalnie na jego potrzeby. Kilka najważniejszych zmiennych automatycznych zostanie przedstawione w tabeli poniżej:

Tabela 6. Zmienne automatyczne.

Zmienna:	Opis:
$?	Sprawdza poprawność wykonania ostatniego polecenia, zwraca true jeśli ostatnie polecenie było wykonane poprawnie lub false jeśli się nie powiodło
$$	Zawiera ostatni wyraz ostatnio wprowadzanej sekwencji do konsoli
$^	Zawiera pierwszy wyraz ostatnio wprowadzanej sekwencji do konsoli
$_	Zmienna potokowa należy od puli zmiennych automatycznych
$Error	Przechowuje tablicę błędów powstałych w bieżącej sesji konsoli. Indeks [0] przechowuje ostatnio zaistniały błąd.
$False	Przechowuje wartość false
$True	Przechowuje wartość true
$ForEach	Przechowuje licznik aktualnego przebiegu pętli ForEach
$Home	Przechowuje adres katologu domowego
$Null	Przechowuje wartość null
$PID	Przechowuje identyfikator procesu aktywnej konsoli PowerShella
$Profile	Przechowuje ścieżkę do profilu dla bieżącego użytkownika oraz bieżącego rodzaju konsoli
$PsCulture	Przechowuje informacje o bieżącym regionie i języku wybranym w komputerze użytkownika
$PsHome	Przechowuje ścieżkę katalogu głównego PowerShella
$PsVersionTable	Przechowuje dokładne informacje o wersji bieżącej sesji konsoli
$Pwd	Przechowuje obiekt typu *Path* zawierający ścieżkę do bieżącego katalogu

Zmiennymi, którym warto poświęcić trochę więcej czasu są `$?` oraz `$error`. Przechowują one informacje na temat błędów zaistniałych podczas wykonywania poleceń. Zmienna $error przechowuje tylko błędy wywołane tylko przez wewnętrzne polecenia PowerShella, jednakże zmienna `$?` zwróci `false` jeśli działanie zewnętrznego programu zakończy się niepowodzeniem.

Aby zobrazować działanie tych zmiennych zaprezentowane zostaną 2 przykłady, w których nastąpi próba kopiowanie pliku do nieistniejącej lokalizacji. Pierwszy przykład będzie wykorzystywał wewnętrzne polecenie PowerShella, natomiast drugi będzie wykorzystywał zewnętrzny program:

```
PS C:\> Copy-Item C:\Windows Z:\
```

```
PS C:\> Copy-Item C:\Windows Z:\
Copy-Item : Cannot find drive. A drive with the name 'Z' does not exist.
At line:1 char:1
+ Copy-Item C:\Windows Z:\
+ ~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (Z:String) [Copy-Item], DriveNotFoundException
    + FullyQualifiedErrorId : DriveNotFound,Microsoft.PowerShell.Commands.CopyItemCommand

PS C:\> $?
False
PS C:\> $Error
Copy-Item : Cannot find drive. A drive with the name 'Z' does not exist.
At line:1 char:1
+ Copy-Item C:\Windows Z:\
+ ~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (Z:String) [Copy-Item], DriveNotFoundException
    + FullyQualifiedErrorId : DriveNotFound,Microsoft.PowerShell.Commands.CopyItemCommand

PS C:\> _
```

Rysunek 2.24. Błąd wywołany kopiowaniem obiektu do nieistniejącego katalogu korzystając z polecenia PowerShellowego

W tym przykładzie zmienna `$Error` przechowuje identyczny komunikat błędu jak ten wyświetlony zaraz po wywołaniu polecenia. Ponadto zmienna `$?` zwróci wartość false, ponieważ ostatnia komenda nie została wykonana poprawnie.

Drugi przykład wygląda następująco:

```
PS C:\> robocopy C:\Windows Z:\
```

```
PS C:\> robocopy C:\Windows Z:\

-------------------------------------------------------------------------------
   ROBOCOPY     ::     Robust File Copy for Windows
-------------------------------------------------------------------------------

  Started : 1 września 2015 15:15:34
2015/09/01 15:15:34 ERROR 3 (0x00000003) Getting File System Type of Destination Z:\
System nie może odnaleźć określonej ścieżki.

   Source : C:\Windows\
     Dest - Z:\

    Files : *.*

  Options : *.* /DCOPY:DA /COPY:DAT /R:1000000 /W:30

------------------------------------------------------------------------------

2015/09/01 15:15:34 ERROR 3 (0x00000003) Creating Destination Directory Z:\
System nie może odnaleźć określonej ścieżki.
PS C:\> $?
False
PS C:\> $Error
PS C:\>
```

Rysunek 2.25. Błąd wywołany kopiowaniem obiektu do nieistniejącego katalogu korzystając z zewnętrznego programu

W drugim przykładzie zewnętrzny program jakim jest `robocopy` wygeneruje błąd, lecz nie zostanie on przechwycony do zmiennej `$Error`. Mimo to zmienna `$?` nadal zwróci `False` jeśli sprawdzimy poprawność ostatnio wykonywanego polecenia.

Stałe

Windows PowerShell nie posiada swojej puli wartości stałych. Istnieje możliwość tworzenia własnych wartości stałych poprzez stworzenie zmiennej i nadanie jej właściwości `Constant` lub `Readonly`. Te dwie wersje parametru `-Option` mają jedną zasadniczą różnicę: zmienna posiadająca właściwość `Readonly` może zostać nadpisana jeśli użyjemy w poleceniu przypisania nowej wartości parametru `-Force`, natomiast w przypadku właściwości `Constant`, nie ma możliwości edytowania wartości zmiennej. Z właściwości `Constant` należy jednak korzystać rozważnie, ponieważ nie ma możliwości usunięcia utworzonej zmiennej. Zmienna będzie aktywna do końca aktywnej sesji PowerShella.

```
New-Variable -Name Temperatura -Description "Temperatura ↲
ciała człowieka" -Value 36.6 -Option Constant
```

Powyższa konstrukcja polecenia wartość stałą o nazwie *Temperatura* posiadającą wartość 36.6. Do PowerShella wydano wiele modułów oraz klas .NET, które po zainstalowaniu udostępniają użytkownikowi korzystanie z różnego rodzaju stałych. Jedną z preinstalowanych klas jest klasa *Math*, pozwalająca na korzystanie z wielu operacji matematycznych, a ponadto oferuje takie stałe wartości jak liczbę Eulera oraz Pi.

Aby odwołać się do klasy *Math*, należy umieścić nazwę klasy w nawiasach kwadratowych, a następnie wywołać określoną metodę/funkcję poprzedzając ją dwoma dwukropkami na przykład:

```
[math]::e
```

lub:

```
[math]::Pi
```

Dokładniejsze opisanie możliwości jakie daje klasa *Math* zostanie przedstawione w podrozdziale poświęconym funkcjom oraz skryptom.

Znaki specjalne

W PowerShellu występuje pula znaków specjalnych, które mają różne zastosowania. Tabela znaków specjalnych wygląda następująco:

Tabela 7. Znaki specjalne.

Znak:	Opis:
*	Oznacza dowolny ciąg znaków
?	Oznacza jeden dowolny znak
[]	Zapis [A-Z] oznacza zakres liter
[]	Zapis [AZ] oznacza jedną literę z wypisanej puli, (w tym przypadku A lub Z)
#	Komentarz w jednej linii
<#	Początek komentarza zawartego w wielu liniach
#>	Koniec komentarza zawartego w wielu liniach
&	Operator wywołania, wywołuje zewnętrzne programy
„ ”	W cudzysłowie podwójnym zapisywane są ciągi znaków. Odniesienie się do zmiennej wewnątrz cudzysłowu przez zapis *$zmienna* spowoduje wyświetlenie wartości zmiennej
' '	W cudzysłowie pojedynczym zapisywane są ciągi znaków. Nie ma możliwości odniesienia się do zmiennych wewnątrz cudzysłowu pojedynczego. Zapis '*$zmienna*' spowoduje wyświetlenie napisu *$zmienna*
;	Oddziela polecenia w jednej linii

2.8 Operatory

Windows PowerShell posiada kilka rodzajów operatorów. Wymienić możemy między innymi [1]:

- operatory arytmetyczne
- operatory przypisania
- operatory porównujące
- operatory logiczne
- operatory binarne
- pozostałe operatory

Warto nadmienić, że nie ma wymogu stosowania znaku spacji pomiędzy wartościami, między którymi został użyty operator. Dzięki temu można stosować krótszą formę zapisu poleceń.

Tabela 8. Tabela operatorów arytmetycznych [1]:

Operator	Opis	Przykład	
		Polecenie	Wynik
+	Operator dodawania; w przypadku użycia na liczbach: zwraca sumę, w przypadku ciągu znaków: tworzy nowy ciąg znaków dopisując drugi na końcu pierwszego, w przypadku użycia na tablicach: dopisuje drugą tablicę do pierwszej	2+5	7
		„A"+„C"	„AC"
		(1..3+6..8)	1 2 3 6 7 8
-	Operator odejmowania; w przypadku użycia na liczbach: zwraca różnicę, nie działa na ciągach znaków ani na tablicach	12-7	5
*	Operator mnożenia; w przypadku użycia na liczbach: zwraca iloczyn, w przypadku użycia na ciągu znaków: tworzy nowy ciąg znaków przez napisanie podanego ciągu znaków *n* razy, w przypadku użycia na tablicy: tworzy nową tablicę przez dopisanie *k* razy zawartości tablicy np. wartości od 1 do *m*	4 * 6	24
		(„C"*5)	„CCCCC"
		(1..3*2)	1 2 3 1 2 3
/	Operator dzielenia; w przypadku użycia na liczbach: zwraca ich iloraz, nie działa na ciągach znaków ani na tablicach	20/5	4
		21/4	5.25
%	Operator modulo, reszta z dzielenia; w przypadku użycia na liczbach: zwraca resztę z dzielenia, nie działa na ciągach znaków ani na tablicach	12%5	2
++	Operator inkrementacji, zwiększa podaną wartość o 1	2++	3
--	Operator dekrementacji, zmniejsza wybraną wartość o 1	5--	4

Tabela 9. Tabela operatorów przypisania [1]:

Operator	Opis	Przykład dla: $A=8;$B=2	
		Polecenie:	Wynik: $A
=	Operator przypisania, przypisuje wartość do zmiennej	$A=10	10
+=	Operator dodaje wartości (wartość po lewej stronie musi być zmienną), wynik przechowuje w zmiennej z lewej strony	$A+=5	13
		$A+=$B	10
-=	Operator odejmuje wartość z prawej strony od wartości z lewej strony (wartość po lewej stronie musi być zmienną), wynik przechowuje w zmiennej z lewej strony	$A-=7	1
		$A-=$B	6
*=	Operator mnoży wartości (wartość po lewej stronie musi być zmienną), wynik działania przechowuje w zmiennej z lewej strony	$A*=9	72
		$A*=$B	16
/=	Operator dzieli wartość z lewej strony przez wartości z prawej strony (wartość po lewej stronie musi być zmienną), wynik przechowuje w zmiennej z lewej strony	$A/=4	2
		$A/=$B	4
%=	Operator dzieli wartość z lewej strony przez wartości z prawej	$A%=3	1

	strony (wartość po lewej stronie musi być zmienną), następnie resztę z dzielenia przechowuje w zmiennej z lewej strony	$A%=$B	0

Tabela 10. Tabela operatorów porównujących [1]:

Operator	Opis	Przykład	
		Polecenie	Wynik
-eq	Operator równości (ang. equal); w przypadku użycia na liczbach: zwraca *true* gdy lewa wartość jest równa prawej, w przypadku użycia na tablicy: zwraca wszystkie elementy tablicy które są równe podanej wartości	5 -eq 5	true
		(1..4) -eq 4	4
		$A=3;$A -eq 2	false
		$A=(1..4); $A -eq 2	2
-ne	Operator nierówności (ang. not-equal); w przypadku użycia na liczbach: zwraca *true* gdy lewa wartość nie jest równa prawej, w przypadku użycia na tablicy: zwraca wszystkie elementy tablicy które nie są równe podanej wartości	4 -ne 6	true
		(1..4) -ne 2	1 3 4
		$A=3;$A -ne 3	false
		$A=(1..4); $A -ne 4	1 2 3
-ge	Operator większe lub równe (ang. greater or equal); w przypadku użycia na liczbach: zwraca *true* gdy lewa wartość jest większa lub równa prawej, w przypadku użycia na tablicy: zwraca wszystkie elementy tablicy które są większe lub równe podanej wartości	7 -ge 8	false
		(1..4) -ge 2	2 3 4
		$A=5;$A -ge 3	true
		$A=(1..4); $A -ge 4	4
-gt	Operator większe niż (ang. greater than); w przypadku użycia na liczbach: zwraca *true* gdy lewa wartość jest większa od prawej, w przypadku użycia na tablicy: zwraca wszystkie elementy tablicy które są większe od podanej wartości	9 -gt 7	true
		(1..4) -gt 2	3 4
		$A=6;$A -gt 4	true
		$A=(1..4); $A -gt 1	2 3 4
-lt	Operator mniejsze niż (ang. lower than); w przypadku użycia na liczbach: zwraca *true* gdy lewa wartość jest mniejsza od prawej, w przypadku użycia na tablicy: zwraca wszystkie elementy tablicy które są mniejsze od podanej wartości	10 -lt 9	false
		(1..4) -lt 3	1 2
		$A=4;$A -lt 6	true
		$A=(1..4); $A -lt 4	1 2 3
-le	Operator mniejsze lub równe (ang. lower or equal); w przypadku użycia na liczbach: zwraca *true* gdy lewa wartość jest mniejsza lub równa prawej, w przypadku użycia na tablicy: zwraca wszystkie elementy tablicy które są mniejsze lub równe podanej wartości	7 -le 7	true
		(1..4) -le 2	1 2
		$A=4;$A -le 8	true
		$A=(1..4); $A -le 3	1 2 3
-like	Operator porównujący; zwraca *true* jeśli lewa wartość powiela się ze wzorcem lub wartością po prawej stronie dokładnie co do pojedynczego znaku. Dopuszczalne symbole wieloznaczne we wzorcu to: ? - dowolny znak, * - ciąg dowolnych znaków bądź zero znaków, [a-d] – zakres liter od a do d, [ac] – wybrany znak , w tym przypadku a lub c	5 -like 5	true
		5 -like 10	false
		„witaj" -like „wit"	false
		„witaj"-like„[w][a-z]?*"	true
-notlike		5 -notlike 5	false

	Negacja operatora porównującego *-like*, neguje wynik, jaki zwróciłoby użycie operatora *-like*	5 -notlike 10	true
		„witaj" -notlike „wit"	true
		„witaj"-notlike „[w][a-z]*"	false
-match	Operator porównujący; zwraca *true* jeśli prawa wartość lub wzorzec częściowo lub całkowicie zawiera się w lewej wartości.	12345 -match 34	true
		12345 -match 13	false
		„witaj" -match „it"	true
		„witaj" -match „wj"	false
-notmach	Negacja operatora porównującego *-match*, neguje wynik, jaki zwróciłoby użycie operatora *-match*	12345 -notmatch 34	false
		12345 -notmatch 13	true
		„pal"-notmatch„al"	false
		„witaj" -notmatch „wj"	true
-contains	Operator zawiera; zwraca *true* jeśli wartość lub tabela zawiera w sobie wartość po prawej stronie.	5 -contains 5	true
		120 -contains 12	false
		„witaj" -contains „*"	false
		(1..9) -contains 5	true
-notcontains	Negacja operatora zawiera *-contains*, neguje wynik, jaki zwróciłoby użycie operatora *-contains*	5 -notcontains 5	false
		120 -notcontains 12	true
		„witaj" -notcontains „*"	true
		(1..9) -notcontains 5	false
-is	Operator porównania typu; zwraca *true* jeśli typ wartości po lewej stronie odpowiada typowi zmiennej podanemu po prawe stronie	5 -is [int32]	true
		„oko" -is [double]	false
		$A=4.5;$A -is [double]	true
		$A="oko";$A -is [string]	true
-isnot	Negacja operatora porównania typu *-is*, neguje wynik, jaki zwróciłoby użycie operatora *-is*	5 -isnot [int32]	false
		„oko" -isnot [double]	true
		$A=4.5;$A -isnot [double]	false
		$A="As";$A -isnot [string]	false

Tabela 11. Tabela operatorów logicznych [1]:

Operator	Opis	Przykład dla: $A=$true; $B=$true $C=$false; $D=$false; $E=true;	
		Polecenie	Wynik
-and	Operator iloczynu logicznego; zwraca *true* jeśli wszystkie zmienna posiadają wartość *true*, można stosować z więcej niż dwoma zmiennymi	$A -and $B -and $C	false
		$A -and $B -and $E	true
		$A -and $C	false
-or	Operator sumy logicznej; zwraca *true* jeśli chociaż jedna zmienna posiada wartość *true*, można stosować z więcej niż dwoma zmiennymi	$A -or $B -or $C	true
		$A -or $B -or $E	true
		$C -or $D	false
-xor	Operator alternatywy wykluczającej; zwraca *true* jeśli jedna z dwóch zmiennych ma wartość true, jednak zwraca *false* jeśli obie zmienne mają tę samą wartość.	$A -xor $C	true
		$A -xor $B	false
		$C -xor $D	false
-not	Negacja; zwraca *true* wyłącznie w przypadku gdy zmienna ma wartość *false*	-not $A	false
		-not $C	true

Tabela 12. Tabela operatorów binarnych

Operator	Opis	Polecenie	Wynik
-band	Zwraca jedynkę na pozycji, jeśli bit w obu liczbach na danej pozycji był jedynką	$A=[Convert]::ToInt32(1100,2)	12
		$B=[Convert]::ToInt32(1010,2)	10
		$C = $A -band $B	8
		[Convert]::ToString($C,2)	1000
-bor	Zwraca jedynkę na pozycji, jeśli przynajmniej jeden bit, w obu liczbach na danej pozycji był jedynką	$A=[Convert]::ToInt32(1100,2)	12
		$B=[Convert]::ToInt32(1010,2)	10
		$C = $A -bor $B	8
		[Convert]::ToString($C,2)	1110
-bxor	Zwraca jedynkę na pozycji, jeśli bity w obu liczbach na danej pozycji miały różne wartości	$A=[Convert]::ToInt32(1100,2)	12
		$B=[Convert]::ToInt32(1010,2)	10
		$C = $A -bxor $B	8
		[Convert]::ToString($C,2)	0110
-bnot	Zwraca jedynkę w pozycjach gdzie występowało zero, i na odwrót. Działa tylko z jedną wartością.	$A=[Convert]::ToInt32(1100,2)	12
		$C = -bnot $A	-
		[Convert]::ToString($C,2)	1111111111111111 1111111111110011

Tabela 13. Tabela pozostałych operatorów [1]:

Operator	Opis	Przykład	
		Polecenie	Wynik

-replace	Operator zamiany; w przypadku stosowania na liczbach lub ciągach znaków: zamiany znaki wedle zadanego wzorca	„rower" -replace „r","j"	jowej
		12230 -replace 2,4	14430
		(9..13) -replace 1,2	9 20 22 22 23
-f	Operator formatu danych; formatuje ciągi znaków typu *string* wedle zadanego wzorca. Wzorce zaczerpnięte są bezpośrednio z klasy .NET String.FormatMethod. Najważniejsze formaty można znaleźć w dodatku B.	"{0:n0}" -f 123456789	123 456 789
		"{0:M}"-f [datetime] "12/24/2015"	24 grudnia
		"{0:P2}"-f 0.25467	25,47%
-as	Operator konwertowania; konwertuje, jeśli jest to możliwe, zmienną na inny typ w celu wykorzystania jej do innych działań. W przypadku, gdy zmiennej nie da się skonwertować, przesyła jej wartość jako *null*. Cała operacja nie narusza wartości zmiennej.	$B=(12 -as [String]) + „on"	12on
		$A=2.6;$A -as [Int32]	3
		$A="Ala";$A -as [Double]	$null

2.9 Strumienie i potoki danych

Windows PowerShell pozwala na dynamiczne przesyłanie efektu polecenia lub skryptu do pliku tekstowego używając w tym celu strumieni. W zależności od tego w jaki zostaną użyte, można zdecydować czy zewnętrzny plik tekstowy, do którego zapisywany jest efekt wywołania polecenia, ma być za każdym razem nadpisywany, czy też kolejne wywołania polecenia będą dopisywać zawartość na końcu pliku nie naruszając jego poprzedniej zawartości. Ponadto można umieszczać w pliku wyjściowym informacje z niestandardowych źródeł. Oto zasada działania strumieni w PowerShell:

```
polecenie > plik.txt
```

lub:

```
polecenie 1> plik.txt
```

umieszcza w pliku tekstowym efekt wywołania *polecenia*. Za każdym razem gdy taka konstrukcja zostanie użyta, *plik.txt* zostanie nadpisany. Następnie:

```
polecenie >> plik.txt
```

lub:

```
polecenie 1>> plik.txt
```

w przypadku istnienia pliku tekstowego, dopisuje efekt wywołania polecenia nie naruszając jego poprzedniej zawartości. Jeśli plik jeszcze nie został utworzony działa tak samo jako konstrukcja wcześniej. Kolejny zapis:

```
polecenie 2> plik.txt
```

lub:

```
polecenie 2>> plik.txt
```

przekierowuje komunikat błędu powstałego na skutek użycia *polecenia* do pliku tekstowego dopisując lub nadpisując już istniejący plik. Konstrukcja:

```
polecenie 3> plik.txt
```

lub:

```
polecenie 3>> plik.txt
```

przekierowuje komunikat ostrzeżenia powstałego na skutek użycia *polecenia* do pliku tekstowego dopisując lub nadpisując już istniejący plik. Zapis:

```
polecenie 4> plik.txt
```

lub:

```
polecenie 4>> plik.txt
```

przekierowuje strumień komunikatów uzupełniających do pliku tekstowego dopisując lub nadpisując już istniejący plik. Konstrukcja:

```
polecenie 5> plik.txt
```

przekierowuje komunikat z debugowania danego polecenia do pliku tekstowego dopisując lub nadpisując już istniejący plik. Ostatni zapis:

```
polecenie *> plik.txt
```

przekierowuje dane ze wszystkich źródeł: efekt wywołania polecenia, błędy, ostrzeżenia, komunikaty informacyjne oraz komunikaty debugowania. Istnieje możliwość przekierowania kilku strumieni do strumienia wynikowego, a ten z kolei można przekierować do pliku tekstowego. W tym celu stosujemy zapis:

```
polecenie 2>&1> plik.txt
```

który w tym przypadku dopisuje strumień komunikatów błędów do strumienia wyjściowego a następnie całość zapisuje do pliku tekstowego. Komunikaty pojawiają się w miejscu ich wystąpienia w programie [14].

Tabela 14. Rodzaje strumieni wyjściowych w Windows PowerShell

Rodzaj strumienia:	Przykład:	
	nadpisaniem wyników w pliku:	z dopisaniem wyników do pliku

Standardowy	Get-Process>proc.txt	Get-Process>>proc.txt
Komunikatów o błędach	Write-Error -Message „Błąd” 2>błędy.txt	Write-Error -Message „Błąd” 2>>błędy.txt
Ostrzeżeń	Write-Warning -Message „Ostrzeżenie” 3>ost.txt	Write-Warning -Message „Ostrzeżenie” 3>>ost.txt
Komunikatów informacyjnych	Write-Verbose –Message „Informacja” 4>info.txt	Write-Verbose –Message „Informacja” 4>>info.txt
Komunikatów debugowania	Write-Debug -Message „Debugowanie” 5>debug.txt	Write-Debug -Message „Debugowanie” 5>>debug.txt

Polecenia użyte w powyższej tabeli sztucznie wygenerują komunikaty na określonych strumieniach wyjściowych. Ma to na celu zaprezentowanie sposobu przesyłania różnych strumieni wyjściowych do pliku.

Istnieje również możliwość przesłania danych potokowo z pliku tekstowego. W tym celu można użyć polecenia Get-Content <ścieżka_i_nazwa_pliku>, które wyświetli zawartość pliku .txt, którą to będzie można przekierować potokowo do innego polecenia lub funkcji.

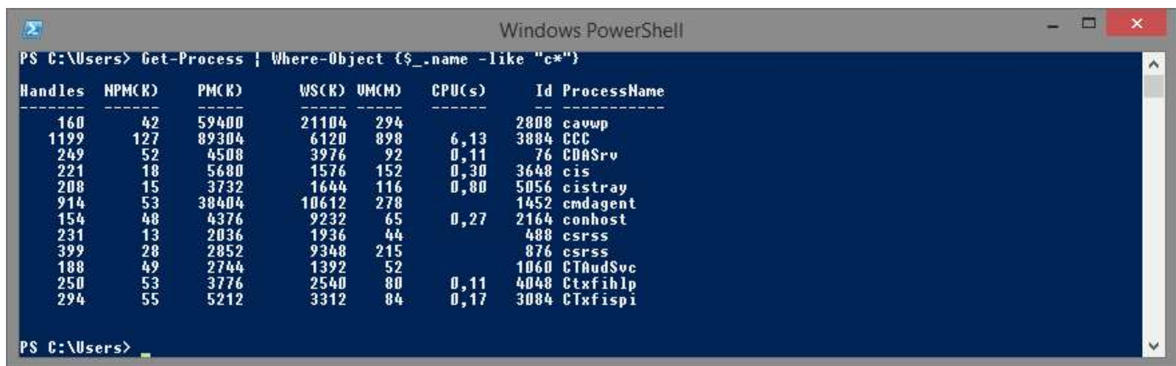
W PowerShellu polecenia oraz dane można ze sobą łączyć za pomocą potoków (ang. pipeline). Dzięki temu dane wyjściowe jednego polecenia mogą być użyte jako dane wejściowe kolejnego. Daje to użytkownikowi bardzo wiele możliwości zastosowania takiego rozwiązania.

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

    Directory: C:\


Name           : AMD
CreationTime   : 2014-08-18 21:26:58
LastWriteTime  : 2014-08-19 22:55:55
LastAccessTime : 2014-08-19 22:55:55

Name           : Fraps
CreationTime   : 2015-06-25 13:01:22
LastWriteTime  : 2015-06-25 13:01:23
LastAccessTime : 2015-06-25 13:01:23

Name           : Keyboard
CreationTime   : 2014-11-28 12:33:03
LastWriteTime  : 2014-11-28 12:33:03
LastAccessTime : 2014-11-28 12:33:03

Name           : PerfLogs
CreationTime   : 2013-08-22 17:36:30
LastWriteTime  : 2013-08-22 17:22:35
LastAccessTime : 2013-08-22 17:22:35

Name           : PowerShellHelp
CreationTime   : 2015-07-22 10:52:16
LastWriteTime  : 2015-07-22 10:54:48
LastAccessTime : 2015-07-22 10:54:48

Name           : Program Files
CreationTime   : 2013-08-22 15:36:15
LastWriteTime  : 2015-07-03 13:29:02
LastAccessTime : 2015-07-03 13:29:02

Name           : Program Files (x86)
CreationTime   : 2013-08-22 15:36:15
LastWriteTime  : 2015-07-15 22:14:15
LastAccessTime : 2015-07-15 22:14:15

Name           : Speed
CreationTime   : 2014-08-18 21:18:49
LastWriteTime  : 2015-06-08 12:30:12
LastAccessTime : 2015-06-08 12:30:12

Name           : Users
CreationTime   : 2013-08-22 15:36:15
LastWriteTime  : 2015-07-25 16:05:43
LastAccessTime : 2015-07-25 16:05:43

Name           : Windows
CreationTime   : 2013-08-22 15:36:15
LastWriteTime  : 2015-07-09 15:34:46
LastAccessTime : 2015-07-09 15:34:46
```

Rysunek 2.26. Dane z polecenia `Get-ChildItem` zostały przesłane potokowo do polecenia `Format-List`, które formatuje wprowadzone dane.

Poza formatowaniem tekstu, przesyłania potokowego można w łatwy sposób użyć aby przefiltrować wynik innego polecenia. Aby tego dokonać należy użyć polecenia `Where-Object`, które wybiera obiekty wynikowe podane wcześniej kryteria. Przykładem takiego zastosowania jest szukanie określonego procesu:

```
Get-Process | Where-Object {$_.name -like "c*"}
```

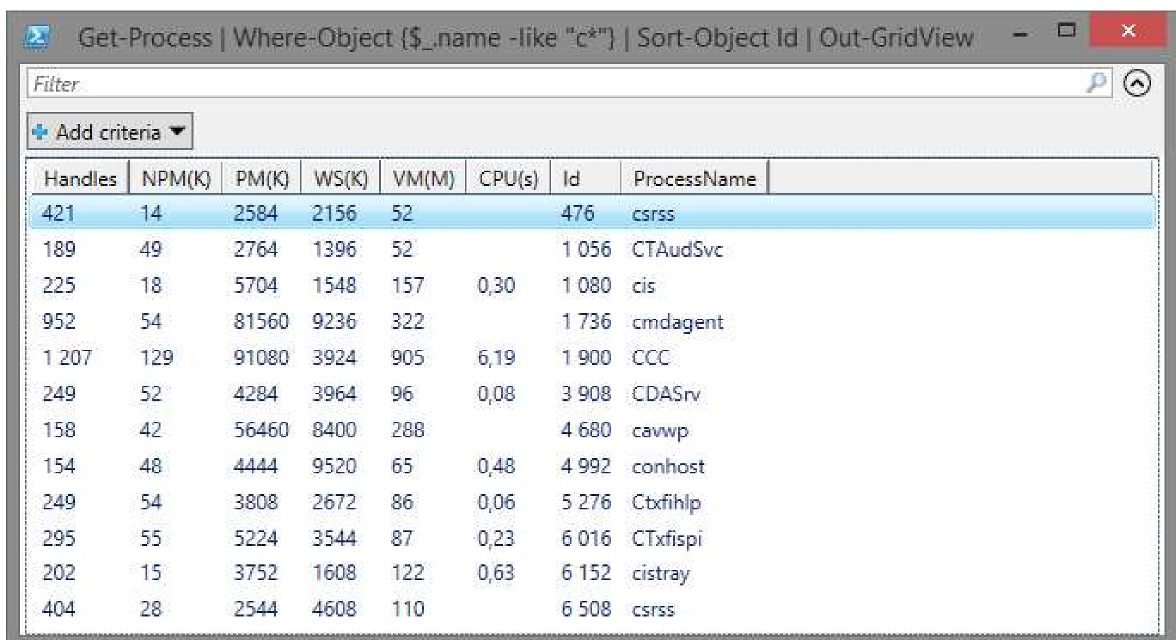


Rysunek 2.27. Przefiltrowany wynik polecenia `Get-Process` zwracający jedynie nazwy procesów zaczynające się na literę „C"

Można również w łatwy sposób używać przesyłania potokowego wielokrotnie. Aby to zaprezentować zmodyfikuje poprzednie polecenie wzbogacając je o nową funkcjonalność:

```
Get-Process | Where-Object {$_.name -like "c*"} |
Sort-Object Id, ProcessName | Out-GridView
```

Efektem wywołania takiego polecenia będzie lista procesów, których nazwa zaczyna się na literę „C", następnie wynik zostanie posortowany, pierwszym kryterium będzie ID procesu, drugim jego nazwa. Całość zostanie wyświetlona w nowym oknie w postaci tabeli dzięki zastosowaniu polecenia `Out-GridView`.



Rysunek 2.28. Rozbudowana wersja polecenia `Get-Process`

Jeśli użyjemy przesyłania potokowego na poleceniu `Get-ChildItem` do kolejnego polecenia `Get-ChildItem`, w efekcie dla każdego elementu wyniku polecenia, w tym przypadku: każdego pliku oraz folderu wyświetlonego w bieżącym katalogu, zostanie zastosowane kolejne polecenie `Get-ChildItem` wyświetlające zawartość katalogu. Można w ten sposób przeszukać drzewo katalogów w poszukiwaniu interesującego nas pliku jednocześnie wyświetlając zawartość każdego z katalogów.

Istnieje kilka strumieni wyjściowych, do których użytkownik może przekierować dane stosując w tym celu polecenia cmdlet:

- Out-Default – przesyła wynik na domyślny strumień wyjściowy
- Out-Null – przesyła wynik do strumienia NULL, dane nie zostaną wyświetlone w żadnym miejscu
- Out-File – przesyła wynik do pliku
- Out-GridView – wyświetla wyniki jako tabelę generowaną w nowym oknie
- Out-Host – przesyła dane na strumień wyjściowy konsoli
- Out-String – wyniki polecenia zostają zapisane jako ciąg znaków
- Out-Printer – przesyła dane do domyślnej drukarki uruchamiając proces drukowania

Aby skorzystać z wyżej wymienionych funkcji należy przekierować polecenie za pomocą potoku do funkcji. Przykład:

```
Get-ChildItem | Out-Printer
```

Ten zapis przekieruje wynik polecenia `Get-ChildItem` do domyślnej drukarki.

Aby pobrać dane z zewnętrznego źródła jakim jest na przykład plik tekstowy, należy użyć polecenia `Get-Content`, które wyświetli zawartość pliku. Można taką zawartość przypisać do zmiennej wewnątrz PowerShella bądź potokowo przesłać do innych poleceń. Dokładne opisanie możliwości i operacji z wykorzystaniem polecenia `Get-Content` znajduje się w rozdziale 3.1 poświęconym operacjom na danych typu `string`.

2.10 Instrukcje warunkowe i pętle

Windows PowerShell posiada sposób na kontrolowanie wykonywania poleceń oraz skryptów wykorzystując do tego pętle oraz instrukcje warunkowe.

Instrukcje warunkowe

Wśród instrukcji warunkowych, wykorzystywanych w PowerShell możemy wymienić instrukcję `If` oraz `Switch`. W obu przypadkach kluczową rolę pełni zadany warunek. Deklarowanie warunku odbywa się przez zastosowanie operatorów pomiędzy wartościami, natomiast bloki instrukcji, które mają być wykonane w przypadku spełnienia, lub nie spełnienia warunku, umieszcza się w nawiasach klamrowych {}.

Instrukcja warunkowa If

Oto podstawowa budowa instrukcji If :

```
If (warunek)
{ blok instrukcji }
```

W najprostszej wersji, blok instrukcji wykonywany jest w przypadku gdy spełniony zostanie warunek. Istnieją też bardziej rozbudowane wersje:

```
If (warunek)
{ blok instrukcji 1 }
Else
{ blok instrukcji 2 }
```

W tym przypadku wykonany zostanie zestaw poleceń określony jako `blok instrukcji 1`, jeśli warunek zostanie spełniony, w przeciwnym razie wykonany zostanie zestaw poleceń przedstawiony jako `blok instrukcji 2`. Istnieje jeszcze jedna wersja tej instrukcji:

```
If (warunek1)
{ blok instrukcji 1 }
Else
{
   If (warunek2)
   { blok instrukcji 2 }
   Else
   { blok instrukcji 3 }
```

```
}
```

Lub jej skrócona wersja:

```
If (warunek1)
{ blok instrukcji 1 }
ElseIf (warunek2)
{ blok instrukcji 2 }
Else
{ blok instrukcji 3 }
```

W wyżej wymienionej konstrukcji warunkowej instrukcji `If` najpierw sprawdzany jest `warunek1`, jeśli jest spełniony wykonane zostaną `blok instrukcji 1`, jeśli nie przechodzimy do sprawdzenia drugiego warunku, jeżeli ten zostaje spełniony, wykonane będą polecenia zawarte w `bloku instrukcji 2`. Natomiast jeśli żaden z warunków nie został spełniony, wykonany zostanie `blok instrukcji 3`. Oto przykład zastosowania takiej konstrukcji:

```
# To polecenie sprawdza aktualny dzień tyogdnia
# i wyświetla jego nazwę.
if ((Get-Date -UFormat %A) -like "poniedziałek")
{ Write-Host "Dzisiaj jest poniedziałek"}
elseif ((Get-Date -UFormat %A) -like "wtorek")
{ Write-Host "Dzisiaj jest wtorek"}
elseif ((Get-Date -UFormat %A) -like "środa")
{ Write-Host "Dzisiaj jest środa"}
elseif ((Get-Date -UFormat %A) -like "czwartek")
{ Write-Host "Dzisiaj jest czwartek"}
elseif ((Get-Date -UFormat %A) -like "piątek")
{ Write-Host "Dzisiaj jest piątek"}
elseif ((Get-Date -UFormat %A) -like "sobota")
{ Write-Host "Dzisiaj jest sobota"}
else
{ Write-Host "Dzisiaj jest niedziela"}
```

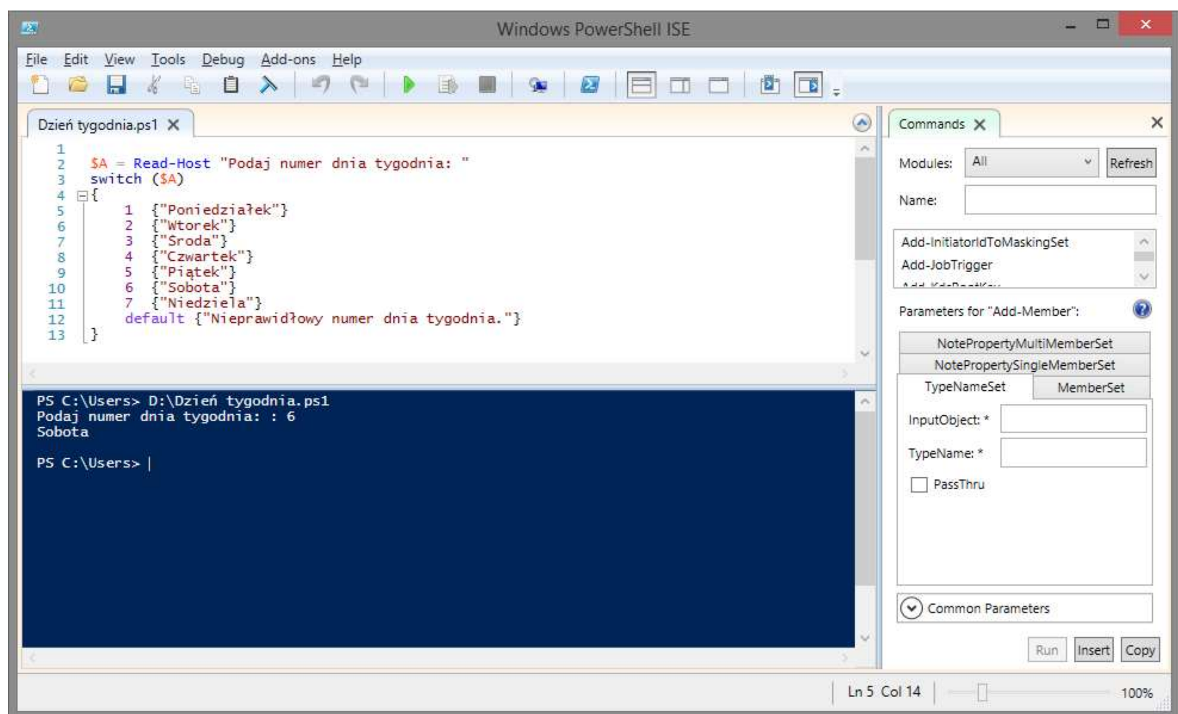
Polecenie `Get-Date` wyświetla pełną datę. Korzystając z parametru `-UFormat` możemy wybrać w jakim formacie data ma być wyświetlana, natomiast wartość „`%A`" wybiera z formatu tylko dzień tygodnia.

Instrukcja warunkowa Switch

Kolejną instrukcją warunkową jaką chce przedstawić jest instrukcja *Switch*:

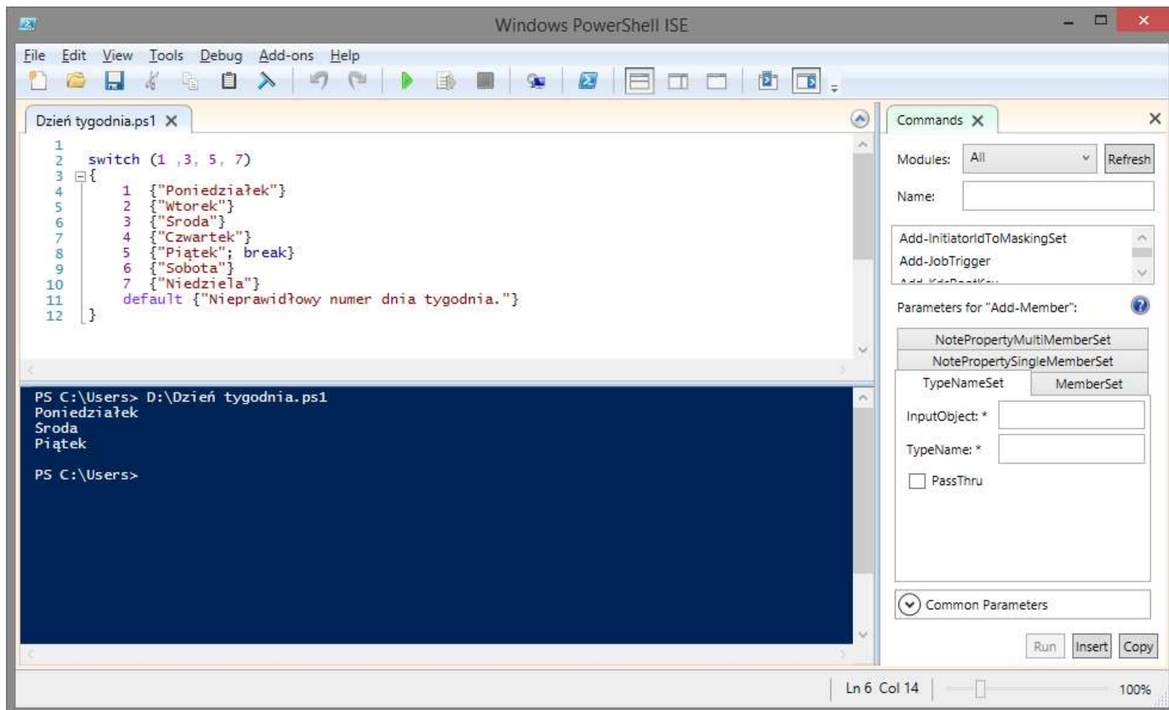
```
Switch (przełącznik)
{
   <wartość przełącznika> { blok instrukcji 1 }
   <wartość przełącznika> { blok instrukcji 2 }
   <wartość przełącznika> { blok instrukcji 3 }
   default { blok instrukcji 4 }
}
```

Na początku instrukcji podawany jest przełącznik, którego wartość będzie porównywana z kolejnymi możliwościami w instrukcji. Na końcu tych możliwości opcjonalnie może znaleźć się wartość `default`, do której to przypisany blok instrukcji zostanie wykonany gdy wartość przełącznika nie będzie spełniała żadnego kryterium.



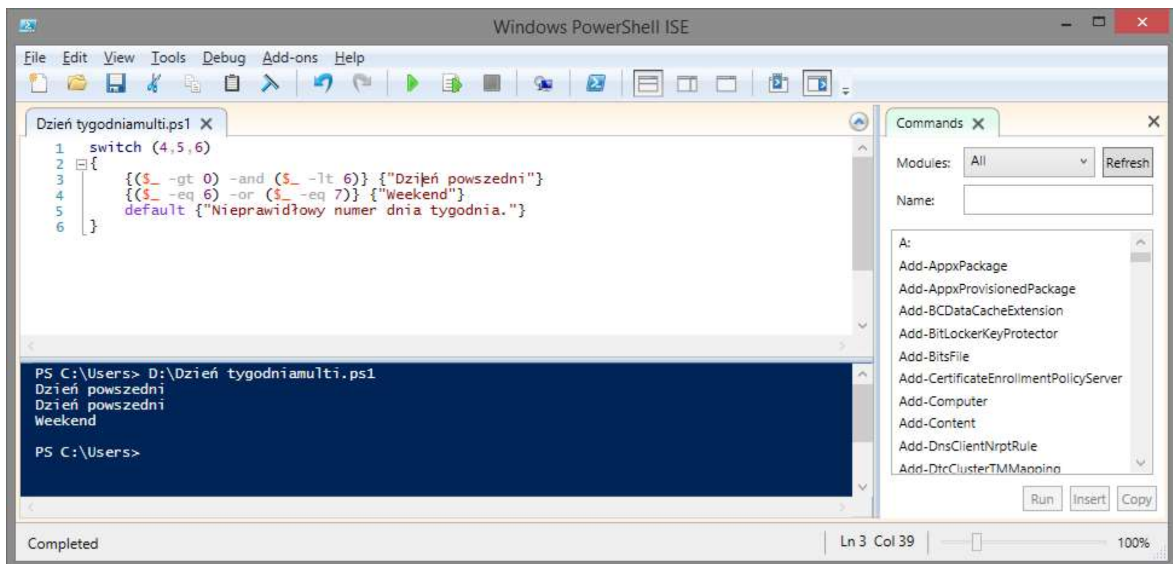
Rysunek 2.29. Przykład zastosowania instrukcji warunkowej `Switch`, wykorzystana została tutaj komenda `Read-Host` do odczytania wartości podanej w konsoli przez użytkownika.

PowerShell umożliwia podanie kilku wartości przełącznika jako warunek, wtedy wszystkie zestawy poleceń spełniające kryterium zostaną wykonane o ile jeden z nich nie posiada polecenia `break`, które przerywa działanie całej instrukcji.



Rysunek 2.30. Wywołanie instrukcji z wieloma wartościami przełącznika, polecenie `break` przerywa działanie instrukcji i nie pozwala na wyświetlenie bloku instrukcji przypisanego do wartości 7

Istnieje możliwość utworzenia jednego bloku instrukcji w konstrukcji `Switch`, który będzie reagował na kilka wartości. W tym celu należy zapisać w nawiasach klamrowych poprzedzających blok instrukcji, warunki jakie muszą spełniać wartości przełącznika, aby wykonać polecenia w określonym bloku instrukcji. W tym celu należy użyć zmiennej potokowej `$_`, która odnosi się w trakcie przebiegu instrukcji do aktualnej wartości przełącznika. Aby określić warunki jakie ma spełniać wartość przełącznika, należy użyć operatorów porównujących. Nic nie stoi na przeszkodzie, żeby połączyć ze sobą kilka warunków, w tym celu należy użyć operatorów logicznych, a same warunki uprzednio umieścić w zwykłych nawiasach.



Rysunek 2.31. Przykład, w którym jeden blok instrukcji odpowiada na różne wartości

Pętle

Poza instrukcjami warunkowymi w PowerShellu dostępne są też pętle. Wyróżniamy trzy pętle: for, foreach oraz while.

Pętla for

Pętla for posiada w swym zapisie deklarację licznika, określenie warunku, dla którego pętla będzie wykonywana oraz działanie na liczniku. Zwykle jest to inkrementacja lub dekrementacja. Konstrukcja pętli for wygląda następująco:

```
for(deklaracja licznika;warunek;operacja na licznika)
{ blok instrukcji }
```

przykład:

```
for($i=10;$i-gt0;$i--)
{ Write-Host "Odliczanie: $i" }
```

Pętla foreach

Pętla foreach wykonywana jest jeden raz dla każdego elementu przekazanego do pętli. Konstrukcja pętli foreach wygląda następująco:

```
foreach(<element> in <obiekt>)
{ blok instrukcji }
```

Przykład zastosowania pętli foreach:

```
$lista = Get-ChildItem -Path C:\
foreach ($pozycja in $lista)
{ Write-Host $pozycja.Name }
```

Pętla While

Instrukcje zawarte w pętli while wykonywane są dopóty, dopóki spełniony jest warunek. Pętla while posiada dwa warianty:

```
while (warunek)
{ blok instrukcji }
```

oraz:

```
do
{ blok instrukcji }
while (warunek)
```

Pierwszy zapis najpierw sprawdza warunek, jeśli jest spełniony, wykonuje blok instrukcji, natomiast druga wersja zawsze wykona blok instrukcji przynajmniej raz, ponieważ warunek sprawdzany jest dopiero po pierwszym przebiegu pętli. Oto przykład:

```
Do
{
   Write-Host "Pracowniku, czy chcesz podwyżkę ?"
   $wybór = Read-Host
}
While ($wybór -notlike "nie")
Write-Host "Dziękujemy za podjęcie słusznej decyzji"
```

Instrukcja Break

Polecenie `break` służy do natychmiastowego opuszczenia bieżącej pętli. W przypadku użycia poza pętlą powoduje przerwanie działania całej funkcji lub skryptu, stąd nie zaleca się używać jej w ten sposób. Polecenia break można używać w połączeniu z etykietami. Jest to jedyny sposób na wykorzystanie etykiet w PowerShell. Służą one do przerwania więcej niż jednej pętli w przypadku zagnieżdżenia pętli. Etykietę zapisuje się poprzez podanie jej nazwy poprzedzając ją dwukropkiem:

```
:etykieta
```

Etykietę należy zapisać przed poleceniem definiującym pętlę:

```
:pierwszy for($i=0;$i-lt10;$i++)
    {
        Write-Host "Pętla zewnętrzna: $i"
        :drugi for($i=0;$i-lt10;$i++)
        {
            Write-Host "Pętla wewnętrzna: $i"
            break pierwszy
        }
    }
```

Polecenie `break pierwszy` zamiast opuścić tylko wewnętrzną pętlę, wedle zasady działania komendy `break`, opuści również zewnętrzną pętlę oznaczoną etykietą `pierwszy`, ponieważ na nią wskazuje polecenie `break`.

Z racji, że Windows PowerShell jest zaawansowanym językiem skryptowym nie umieszczono w nim polecenia `goto`, ani żadnego jego odpowiednika, dlatego też etykiety mogą być użyte tylko w jednym, określonym przypadku przedstawionym na przykładzie powyżej.

2.11 Funkcje

Funkcje pozwalają na zapisanie wielu poleceń oraz bloków instrukcji w postaci jednej funkcji. Ponadto funkcje pozwalają na przekazywanie do niej parametrów dzięki czemu można dynamicznie zmieniać wartości na jakich funkcja będzie operować. Każda funkcja powinna zwracać jakąś wartość z operacji, które wykonuje. Do zwracania wartości z funkcji służy polecenie `return`. Jeżeli funkcja nie zwraca wartości przez `return`, wtedy każde wypisanie wartości może zostać użyte jako wartość zwracana. Wzór funkcji wygląda następująco:

```
function <nazwa funkcji>(argument1, argument2...)
{
  blok instrukcji
  return <wynik>
}
```

Jeżeli wywołamy funkcję z większą ilością parametrów niż zostało zdefiniowane, możemy odnieść się do tych dodatkowych parametrów wewnątrz ciała funkcji poprzez zmienną $args.

Zmienna $args jest tablicą, która przechowuje wszystkie niezdefiniowane parametry wywołania funkcji lub skryptu. Zapis $args[0] odnosi się do pierwszego, dodatkowego parametru, $args[1] do drugiego dodatkowego parametru i tak dalej. Zmienna $args ma zasięg lokalny. Użyta wewnątrz skryptu odnosi się do parametrów skryptu, użyta wewnątrz funkcji odnosi się do parametrów funkcji.

Przykład funkcji wykorzystującej argumentu dodatkowe:

```
function Argumenty
{
    Write-Host "Wyświetlam argumenty od końca:"
    for($i=$($args.Count -1);$i-ge0;$i--)
    {Write-Host "Argument $i : $($args[$i])"}
}
```

oraz efekt jej wywołania:

```
Argumenty "a" "b" "c"
Wyświetlam argumenty od końca:
Argument 2 : c
Argument 1 : b
Argument 0 : a
```

Argumenty w funkcji można definiować na dwa sposoby. Jednym z nich jest zdefiniowanie ich w nawiasach za nazwą funkcji, oddzielając kolejne argumenty przecinkami, natomiast drugim sposobem jest wpisanie polecenia `param()` z argumentami wewnątrz nawiasów, również oddzielając je przecinkami:

```
function <nazwa funkcji>
{
  param(argument1, argument2,...)
  blok instrukcji
  return <wynik>
}
```

Poza typami parametrów można zmienić ich zaawansowane ustawienia. W tym celu należy korzystać z instrukcji `parameter`, a wewnątrz niej określić wartości false lub true dla różnych właściwości na przykład:

```
function Typ_zmiennej
{
   param([parameter(Mandatory=$true)]$Zmienna)
   Process
   { Write-Host Zmienna typu: $Zmienna.GetType().Name }
}
```

Tabela 15. Najważniejsze właściwości możliwe do nadania parametrowi funkcji

Właściwość:	Wartość:	Opis:
Mandatory	$true lub $false	Określa czy paramatr jest wymagany do wywołania funkcji
ValueFromPipeline	$true lub $false	Określa czy można przesyłać wartości do parametru potokowo z innego źródła
Position	Liczba całkowita	Określa czy parametr musi mieć określoną pozycję zapisie wywołania funkcji
HelpMessage	<string>	Ustawia treść pomocy dla tego parametru
AllowNull	$true lub $false	Zezwala na podanie wartości `$null` jako wartości parametru
AllowEmptyString	$true lub $false	Zezwala na pusty ciąg znaków jako wartość parametru
AllowEmptyCollection	$true lub $false	Zezwala na pusty zbiór kolekcji jako wartość parametru
ValidateLength	(min,max)	Nakłada wymóg określonej długości łańcuchu znaków parametru

Powyższy zapis definiuje funkcję, która posiada parametr Zmienna, który jest wymagany do każdego użycia funkcji i należy również podać jego wartość:

```
$A = 2.5
Typ_zmiennej -Zmienna $A
Zmienna typu: Double
```

Przykład funkcji która zapisuje liczbę jako wartość bezwzględną:

```
function W_absolutna($p)
{
   if($p -lt 0)
   { $p = $p * -1 }
   return $p
}
```

Następnie aby wywołać funkcję należy napisać jej nazwę wraz z parametrem:

```
W_absolutna (-2)
2
W_absolutna 3
3
```

Funkcja ta oblicza wartość bezwzględną z podanej liczby. Jeśli podana liczba jest mniejsza od zera, zmienia jej znak na przeciwny wykorzystują mnożenie przez -1. W pozostałych przypadkach liczba zostaje przepisywana. Taka funkcja prawidłowo spełnia swoje obowiązki dopóki argumentem będzie liczba, z której można obliczyć wartość bezwzględną. W przypadku, gdy jako argument podamy ciąg znaków, warunek zawarty w funkcji nie zostanie spełniony, a sam ciąg zostanie przepisany. Aby tego uniknąć, należy sprawdzić typ przekazywanego argumentu. Zmodyfikowana wersja wyżej wymienionej funkcji wygląda następująco:

```
 function W_absolutna_nowa($p)
{
    if(($p.GetType().Name -like "Int32") -or
       ($p.GetType().Name -like "Double"))
    {
        if($p -lt 0)
        { $p = $p * -1 }
        return $p
    }
    else
    {"Podano zły format zmiennej !"}
}
```

Wykorzystując metodę `GetType().Name` pobieramy nazwę typu, następnie porównujemy do `Int32` lub `Double`. Te dwa warunki połączone są ze sobą za pomocą operatora logicznego `-or`, który zwraca wartość `true` w przypadku, gdy chociaż jeden z warunków zostaje spełniony. Jeśli typ parametru to `Int32` lub `Double`, sprawdzany jest kolejny warunek. Jeżeli podana liczba jest mniejsza od zera, wykonywana jest operacja obliczająca wartość bezwzględną, natomiast kiedy argument jest większy lub równy zeru, jest on po prostu przepisywany. W przypadku gdy argument ma inny typ, funkcja wypisuje stosowny komunikat o błędzie.

Wywołując funkcję, jej parametry podajemy oddzielając je spacją. Wyjątkiem są liczby ujemne, które należy umieścić w nawiasach, każdą wartość ujemną w osobnych nawiasach:

```
function Suma($A, $B)
   {
      $wynik = $A + $B
      return $wynik
   }
   Suma (-6) 2
   -4
```

Można również podać jako argument całe działanie umieszczone w nawiasach. PowerShell najpierw dokona obliczeń w nawiasie, a dopiero potem przekaże wynik jako parametr do funkcji:

```
Suma (2+3) (1+2)
8
$C = 3
$D = 7
Suma 3 ($C-$D)
-1
```

Z racji, że w definicji funkcji nie zostały określone typy parametrów, a operator dodawania działa nie tylko na liczbach, można użyć tej funkcji do łączenia ciągów znaków:

```
Suma "Dzień " "dobry"
Dzień dobry
```

Wartości do funkcji można również przesyłać potokowo. W tym celu w zapisie funkcji należy użyć zmiennej potokowej $_. Tak wygląda struktura takiej funkcji [13]:

```
Function <nazwa_funkcji>
{
  Begin
  {
   # Blok instrukcji wykonanywany jeden raz przed
   # przetworzeniem pierwszej wartości w potoku
  }
```

```
  Process
  {
   # Blok instrukcji wykonywany raz dla każdej wartości
   # w potoku. Odwołujemy się do wartości przez zmienną
   # potokową $_
  }
  End
  {
   # Blok instrukcji wykonanywany jeden raz po
   # przetworzeniu ostatniej wartości w potoku
  }
}
```

Należy pamiętać, że konstrukcje `Begin`, `Process` i `End` są całkowicie opcjonalne i można stosować je oddzielnie. Jako przykład zastosowania instrukcji `process`, przedstawiona zostanie funkcja mnożąca wartość przez 3:

```
function razy3
{
  process{$_ * 3}
}
```

oraz sposób jej użycia na liczbie:

```
12.3 | razy3
36.9
```

lub tablicy:

```
(1..3) | razy3
3
6
9
```

> Z racji, że PowerShell jest interpreterem poleceń, całe funkcje można zdefiniować poprzez wpisanie ich w konsolę linijka po linijce, bez konieczność zapisywania ich w osobnym skrypcie. Funkcje takie będą aktywne na czas bieżącej sesji PowerShella.

2.12 Tablice

Windows PowerShell pozwala na tworzenie oraz operowanie na tablicach. Tablice można podzielić na dwa rodzaje: zwykłe oraz mieszające.

Tablice zwykłe

Zwykłe tablice, o ile wcześniej nie zostanie zdefiniowany ich typ, mogą przechowywać dowolne typy wartości jednocześnie. Tablice indeksowane są od 0. Aby odnieść się do dowolnego elementu tablicy należy zapisać nazwę tablicy, a następnie w nawiasach kwadratowych umieścić indeks wskazujący na wybraną pozycję w tablicy. Deklarowanie zwykłej tablicy wygląda następująco:

```
$A = @()
$B = New-Object "Int32[]" 10
```

`"Int32[]"` deklaruje obiekt jako tablicę typu `Int32` natomiast wartość `10` określa wielkość tablicy. Można również od razu zdefiniować wartości przechowywane przez tablicę:

```
$A = (10..20)
$A[3]
13
```

Jeżeli jako indeks tablicy podamy całkowitą liczbę ujemną, będzie on wskazywać na określony element od końca tablicy:

```
$A[-2]
19
```

Podczas próby wywołania elementu spoza zakresu tablicy, użytkownik otrzymuje wartość `$null`, natomiast jeśli spróbuje przypisać wartość do takiego elementu, PowerShell wyświetli komunikat o błędzie przekroczenia zakresu tablicy.

Zmienne, które przechowują efekty wywołania natywnych poleceń PowerShella posiadają typ tablicy. Można to wykorzystać do wyświetlenia kolejnych elementów tablicy wedle życzenia użytkownika:

```
$A = Get-ChildItem -Path C:\
$A | Sort-Object LastWriteTime |
ForEach {Write-Host $_}
```

Następuje definicja zmiennej `$A` jako efekt polecenia `Get-ChildItem` w lokalizacji C:\, następnie przesłanie potokowo zmiennej do funkcji sortującej według ostatniej daty modyfikacji, następnie potokowo przekazanie posortowanych danych do funkcji `ForEach`, która wykonuje akcję zawartą w nawiasach klamrowych dla każdego elementu tablicy - w tym przypadku wyświetla wartość zmiennej potokowej `$_`, czyli bieżącego elementu tablicy. Efektem będzie wyświetlenie samych nazw plików i folderów w lokalizacji C:\ posortowane według daty ostatniej modyfikacji.

Tablica mieszająca

Tablica mieszająca jest specjalnym rodzajem tablicy. Przechowuje ona parę wartości: klucz oraz wartość. Aby odnieść się do wartości należy podać w nawiasach kwadratowych zamiast indeksu tablicy, klucz do wartości jaka nas interesuje. Deklaracja tablicy mieszającej wygląda następująco:

```
$A = @{}
```

Wartości do tablicy mieszającej możemy dodawać wykorzystując metodę `Add`. Dodawanie kolejnych obiektów wygląda zatem tak:

```
$A.Add(<klucz>,<wartość>)
```

Przykład:

```
$A.Add("Kolor","Czerwony")
$A["Kolor"]
Czerowny
```

Istotną informacją jest fakt, że klucze w tablicy mieszającej są unikatowe. Nie ma możliwości dodania kolejnej pozycji wykorzystując klucz, który znajduję się już w tablicy.

Tablice wielowymiarowe

PowerShell pozwala również na tworzenie wielowymiarowych tablic. Tablice wielowymiarowe można podzielić na zwykłe oraz postrzępione.

Tablice wielowymiarowe, zwykłe

Tablice zwykłe mają określony rozmiar oraz wymagają deklaracji typu. Typ ten definiuje jakiego rodzaju wartości będzie można wprowadzać do tablicy, są to na przykład `Int32`, `Double`, `String`. Jednakże zastosowanie typu `Array` pozwala na dodawanie różniących się od siebie typów wartości do jednej tablicy zwykłej.

Zwykła tablica dwuwymiarowa:

```
$tablica1 = New-Object "int32[,]" 3,4
```

Zwykła tablica trójwymiarowa:

```
$tablica2 = New-Object "int32[,,]" 2,3,4
```

Aby odwołać się do określonego elementu tablicy należy podać w nawiasach kwadratowych jego współrzędne oddzielając je przecinkiem.

```
$tablica1[1,2]
$tablica2[0,1,2]
```

Tablica wielowymiarowa, postrzępiona

Tablica postrzępiona składa się z wielu jednowymiarowych list. Przykład tablicy postrzępionej:

```
$tablica3 = @( (1,2,3,4),("A","B","C") )
```

Jak widać na przykładzie, tablica ta składa się z dwóch tablic jednowymiarowych rozdzielonych przecinkiem, jedna z nich posiada 4 elementy, natomiast druga tylko 3. Aby odwołać się do dowolnego elementu należy podać każdą współrzędną w osobnych nawiasach kwadratowych: pierwsza współrzędna określa którą tablicę jednowymiarową wybieramy, natomiast druga określa element wybranej tablicy:

```
$tablica3[1][2]
C
```

Tablica zagnieżdżona

PowerShell umożliwia również tworzenie zagnieżdżonych tablic postrzępionych. Proces ten polega na umieszczeniu kolejnej jednowymiarowej tablicy jako pojedynczego obiektu innej tablicy. Jeżeli nadpiszemy wartość przechowywaną w wyżej wymienionej `tablicy 3`, do której odnosimy się poprzez indeksy [1][2] kolejną tablicą:

```
$tablica3[1][2] = ("X","Y","Z")
```

To aby odnieść się do wartości „X" należy zastosować taki zapis:

```
$tablica3[1][2][0]
```

Zgodnie z panującą zasadą, wedle której tablice indeksowane są od zera, pierwszy indeks wskazuje na drugą tablicę, drugi indeks na trzeci element drugiej tablicy będący

jednocześnie kolejną jednowymiarową tablicą, natomiast trzeci indeks odnosi się do pierwszego elementu nowo zdefiniowanej tablicy.

2.13 Skrypty

Skrypty są najbardziej rozbudowaną formą wykonywania złożonych poleceń. Dają największe możliwości pozwalając na kontrolowanie przebiegu pracy na wiele różnych sposobów oraz wykorzystują duże ilości poleceń, które razem ciężko byłoby zapisać w postaci jednej długiej komendy czy chociażby funkcji. Warto wspomnieć o kilku ważnych zasadach, jakich warto przestrzegać podczas tworzenia skryptu:

- Wszystkie definicje zaleca się umieścić na początku skryptu. Pozwala to uniknąć problemu jakim jest wywołanie niezdefiniowanej funkcji.
- Jeżeli nazwy zmiennych oraz funkcji chociaż w minimalnym stopniu będą opisywały zadanie jakie wykonują, znacznie ułatwi to analizowanie kodu po dłuższym czasie oraz ułatwi jego zrozumienie przez inne osoby.
- Używanie wcięć oraz spacji pozwala w przejrzysty sposób pokazać zasięgi funkcji, pętli bądź innych instrukcji

W dalszej części rozdziału poprzez wykorzystanie skryptów zostaną zaprezentowane różne sposoby działania mniej lub bardziej skomplikowanych poleceń w Windows PowerShell.

Instrukcja `Try{}Catch{}`

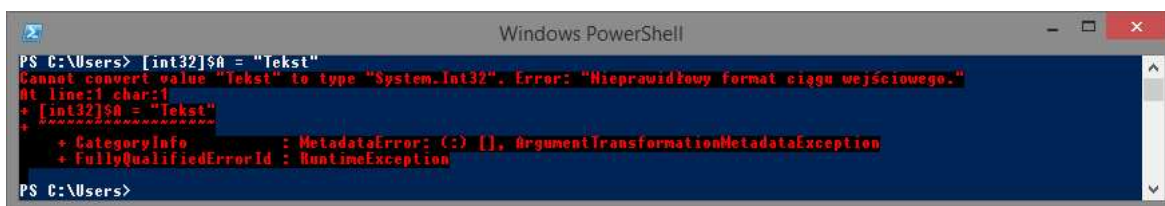
W PowerShellu istnieje zapis instrukcji `Try{} Catch{}`. Jej wykorzystanie daje te same możliwości jak w zaawansowanych językach programowania np.: C# lub C++. Instrukcja ta pozwala na ręczną obsługę błędów pojawiających się w w trakcie wykonywania poleceń. Aby poprawnie z niej skorzystać należy w nawiasach klamrowych następujących po słowie `Try` umieścić fragment kodu, w którym chcemy zastosować obsługę wyjątku. Bezpośrednio za nawiasem klamrowym zamykającym blok `Try` należy umieścić polecenie `Catch{}`, w którego nawiasach klamrowych umieszczamy akcję, jaką program ma wykonać w przypadku wystąpienia błędu. W razie zastosowania instrukcji `Try{} Catch{}` należy pamiętać, że konsola nie zwróci domyślnego komunikatu błędu jaki powinien się pojawić, zastępując go poleceniami umieszczonymi w nawiasach klamrowych za komendą `Catch`.

Oto wzór instrukcji:

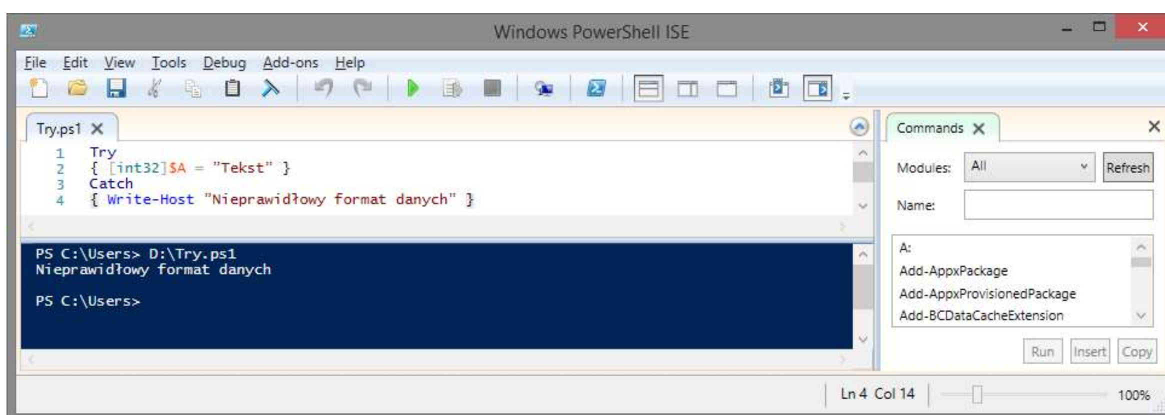
```
Try
{ Blok instrukcji }
Catch
{ Polecenia obsługi wyjątku }
Finally
{ Polecenia które zostaną wykonane w każdym przypadku }
```

oraz przykład zastosowania:

```
Try
{ [int32]$A = "Tekst" }
Catch
{ Write-Host "Nieprawidłowy format danych" }
```



Rysunek 2.32. Wygląd standardowego komunikatu błędu w przypadku próby przypisania wartości różniącej się typem do zmiennej



Rysunek 2.33. Obsługa tego samego błędu za pomocą konstrukcji `Try{} Catch{}`

Domyślnie w instrukcji zostanie zastosowany ten sam komunikat dla każdego rodzaju błędu. Można temu zapobiec definiując określone komunikaty w zależność od zaistniałego błędu wykorzystując w tym celu elementy klas *System* i *System.Management.Automation*. Wśród elementów tej klasy możemy wyróżnić kilka typów błędów:

Tabela 16. Przykładowe rodzaje błędów występujących w PowerShell.

Rodzaje błędów:	Opis:
System.Management.Automation.ApplicationFailedException	Błąd aplikacji
System.Management.Automation.CmdletInvocationException	Błąd wywołania polecenia cmdlet
System.Management.Automation.JobFailedException	Błąd wykonywania pracy w tle
System.Management.Automation.LoopflowException	Błąd pracy pętli
System.Management.Automation.MethodInvocationException	Błąd wywołania metody
System.Management.Automation.PSArgumentNullException	Błąd w postaci argumentu typu $null
System.Management.Automation.RemoteException	Błąd połączenia zdalnego
System.DivideByZeroException	Błąd dzielenia przez zero
System.BadImageFormatException	Błąd formatu obrazka
System.InvalidOperationException	Błąd niepoprawnego użycia operacji
System.IndexOutOfRangeException	Błąd wyjścia indeksu poza zakres

Aby podejmować akcje zależne od rodzaju zaistniałego błędu należy zastosować zapis:

```
Try
{
   <blok instrukcji>
}
Catch [System.DivideByZeroException]
{
   <akcje wykonane w przypadku dzielenia przez zero>
}
   Catch [System.Management.Automation.LoopflowException]
{
   <akcje wykonane w przypadku błędu działania pętli>
}
```

Zadania wykonywane w tle

PowerShell oferuje nową funkcjonalność jaką jest możliwość wykonywania zadań w tle (ang. background jobs). Pozwala to uruchomienie czasochłonnego zadania, pozostawiając tym samym aktywną konsolę w rękach użytkownika. Każde zadanie działające w tle posiada swój unikalny numer ID, nazwę, status zadania oraz lokalizację w której jest wykonywane. Aby zarządzać zadaniami w tle należy posłużyć się określonym zestawem poleceń przedstawionym w tabeli poniżej.

Tabela 17. Tabela poleceń zarządzających zadaniami wykonywanymi w tle.

Polecenie:	Opis:
Start-Job	Polecenie rozpoczynające wykonywanie zadania w tle
Get-Job	Wyświetla listę wszystkich prac w tle wraz z ich danymi
Receive-Job	Przenosi wyniki pracy w tle na aktywną sesję
Stop-Job	Zatrzymuje prace wykonywane w tle
Wait-Job	Blokuje konsolę do czasu ukończenia zadania działającego w tle
Suspend-Job	Tymczasowo zatrzymuje zadania działające w tle z możliwością ich wznowienia
Resume-Job	Wznawia wcześniej zatrzymane zadania działające w tle
Remove-Job	Usuwa zadanie działające w tle. Można usuwać jedynie zatrzymane bądź ukończone zadania.

Uruchamianie skryptów

Skrypty można uruchamiać wykorzystując do tego kilka sposobów. Pierwszym z nich jest wywołanie skryptu poprzez podanie jego pełnej lokalizacji wraz z rozszerzeniem np.:

```
C:\Skrypt.ps1
```

lub jeśli skrypt znajduje się w bieżącym katalogu można wykorzystać jeden z zapisów:

```
.\Skrypt.ps1
./Skrypt.ps1
```

Kolejnym sposobem jest użycie operatora wywołania &. Służy on do otwierania zewnętrznych programów, funkcji oraz skryptów:

```
& "C:\Nowy folder\Skrypt.ps1"
```

Rozwiązuje on problem spacji w nazwach folderów lub pliku w ścieżce: wystarczy ująć całą ścieżkę w znak cudzysłowu, zamiast każdy folder, w którym występuje spacja, osobno. Ścieżkę do skryptu można również przypisać do zmiennej, a następnie wywołać wykorzystując parametr & np.:

```
$A = C:\Windows\System32\notepad.exe
& $A
```

Domyślnie wszystkie wartości: zmienne, funkcje, aliasy itd. utworzone w skrypcie przestają istnieć wraz z zakończeniem skryptu. Dzieje się tak, ponieważ podczas wywołania skryptu Windows PowerShell tworzy nowy, wewnętrzny zasięg, w którym to operują nowe wartości. Wraz z końcem skryptu zasięg ten jest usuwany. Aby uniknąć takiej sytuacji należy uruchomić skrypt poprzez wywołanie z kropką:

```
. C:\Skrypt.ps1
```

```
. .\Skrypt.ps1
```

Spowoduje to uruchomienie skryptu wykorzystując bieżący zasięg konsoli, przez co wszystkie nowe wartości zdefiniowane wewnątrz skryptu nadal będą istnieć i będzie można się do nich odnieść nawet po zakończeniu działania skryptu.

Przykładowe skrypty:

Pierwszym przykładem będzie skrypt obliczający silnię z podanego argumentu:

```
If($args.Count -ne 1)
{
   Write-Host "Podano nieprawidłową ilość paramterów."
   Break
}
Elseif($args[0].GetType().Name -ne "Int32")
{
   Write-Host "Podano zły format parametru."
   Break
}
Else
{    [int32]$wynik = 1
     for ($i=$args[0];$i-gt0;$i--)
         { $wynik *= $i }
     "$wynik"
}
```

Wywołanie skryptu, gdy znajduje się na przykład na dysku C:, wygląda następująco:

```
C:\silna.ps1 5
```

Wartość, z której będzie obliczana silnia podawana jest w argumencie. Skrypt posiada zabezpieczenia przed dwoma przypadkami, dla których wykonanie obliczeń byłoby niemożliwe. Metoda `Count` zmiennej `$args` sprawdza ilość argumentów, a następnie wyświetla komunikat oraz wychodzi z instrukcji warunkowej w przypadku, gdy ilość jest inna niż 1. Metoda `GetType().Name` zmiennej `$args[0]`, która w takim zapisie odnosi się bezpośrednio do pierwszego argumentu, sprawdza typ argumentu, po czym wyświetla komunikat oraz wychodzi z instrukcji warunkowej w przypadku, gdy typ argumentu jest inny niż liczba całkowita. Jeżeli podany argument spełnia wymagania, następuje obliczenie silni wykorzystując w tym celu pętlę `for`, a na koniec wyświetlenie wyniku.

Do wykonywania bardziej skomplikowanych operacji matematycznych można wykorzystać klasę `math`. Lista najważniejszych operacji jakie można wykonać z wykorzystaniem tej klasy wygląda następująco:

Tablica 18. Lista operacji możliwych do wykonania z użyciem klasy `math`

Metoda:	Opis:	Przykład:	Wynik:
Acos	Arcus cosinus	[Math]::Acos(0.5)	1,0471975511966
Asin	Arcus sinus	[Math]::Asin(0.5)	0,523598775598299
Atan	Arcus tangens	[Math]::Atan(0.5)	0,463647609000806
Ceiling	Zaokrąglenie w górę, tzw. sufit	[Math]::Ceiling(2.3)	3
Cos	Cosinus	[Math]::Cos(0.5)	0,877582561890373
Floor	Zaokrąglenie w dół, tzw. podłoga	[Math]::Floor(2.7)	2
Sin	Sinus	[Math]::Sin(0.5)	0,479425538604203
Tan	Tangens	[Math]::Tan(0.5)	0,54630248984379
Round	Zaokrąglenie do ilości miejsc po przecinku	[Math]::Round(100.23, 1)	100.2
Truncat	Zaokrąglenie przez obcięcie liczb po przecinku	[Math]::Truncat(100.23)	100
Sqrt	Pierwiastek	[Math]::Sqrt(4)	2
Log	Logarytm naturalny	[Math]::Log(10)	2,30258509299405
Log10	Logarytm dziesiętny	[Math]::Log10(1000)	3
Pow	Potęgowanie	[Math]::Pow(3,3)	27
Abs	Wartość absolutna	[Math]::Abs(-4)	4
Min	Wartość minimalna	[Math]::Min(1,2,3,4)	1
Max	Wartość maksymalna	[Math]::Max(1,2,3,4)	4
Sign	Wartość znaku, zwraca 1 gdy liczba jest dodatnia, 0 gdy liczba jest zerem i -1 gdy liczba jest ujemna	[Math]::Sign(-3)	-1

Klasa math pozwala na różnego rodzaju operacje wykorzystując wartości stałe predefiniowane w klasie. Jedną z takich wartości jest liczba Eulera:

```
$E = [math]::E
$A = [math]::Pow($E,2)
$A
7,38905609893065
$X = Read-Host -Prompt "Podaj wartość"
$A = [math]::Pow($E,$X)
```

Aby przedstawić użyteczność klasy `math` zaprezentowany zostanie skrypt korzystający z jej możliwości:

```
Write-Host "Podaj wartość: " -NoNewLine
$X = Read-Host
Write-Host "1. Sinus"
Write-Host "2. Cosinus"
Write-Host "3. Tangens"
Write-Host "4. Arcus Sinus"
Write-Host "5. Arcus Cosinus"
Write-Host "6. Arcus Tangens"
Write-Host "Wybierz operację: " -NoNewLine
$Wybor = Read-Host

try{
switch($Wybor)
{
    1 {[math]::Sin($X)}
    2 {[math]::Cos($X)}
    3 {[math]::Tan($X)}
    4 {[math]::Asin($X)}
    5 {[math]::Acos($X)}
    6 {[math]::Atan($X)}
    default {Write-Warning -Message "Wybrano nieprawidłową
        operację"}
}
}
catch{Write-Warning -Message "Podano nieprawidłową
        wartość"}
```

Skrypt pobiera wartość, która posłuży do obliczeń. Następnie pojawia się pytanie o rodzaj operacji. Obsługa błędów stosowana jest dla dwóch przypadków: gdy wartość podana do obliczeń nie jest liczbą oraz gdy użytkownik wybierze zły indeks operacji. Pierwszy przypadek obsługiwany jest przez wartość `default` instrukcji `switch`, natomiast drugi przez instrukcję `Try{}Catch{}`.

Kolejnym skryptem, jaki zostanie zaprezentowany w tym rozdziale, jest skrypt transponujący macierz dwuwymiarową:

```
do
```

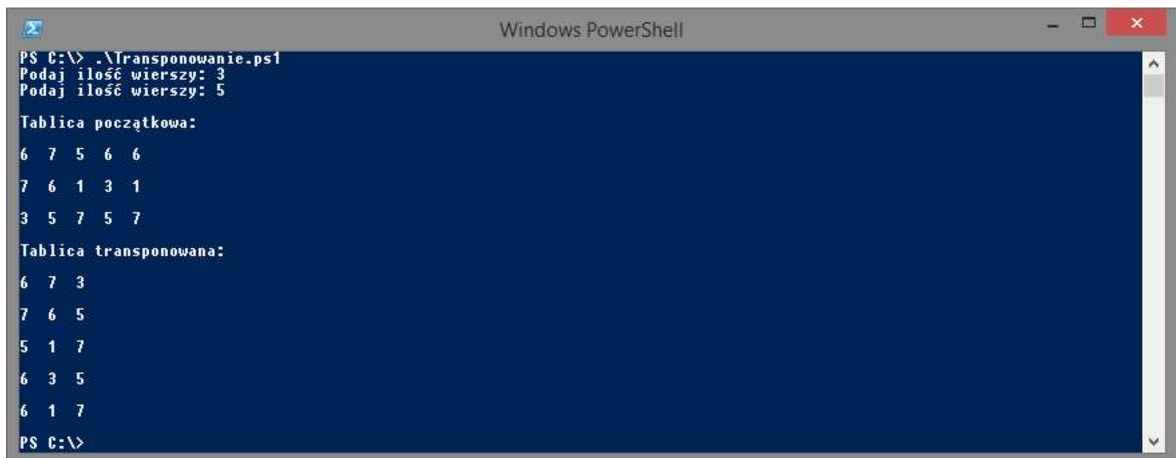
```
{
    Write-Host -NoNewLine "Podaj ilość wierszy: "
    try { [int32]$a = Read-Host }
    catch {"Nieprawidłowy format danych";$a=$null}
}
while($a -like $null)

do
{
    Write-Host -NoNewLine "Podaj ilość kolumn: "
    try { [int32]$b = Read-Host }
    catch {"Nieprawidłowy format danych";$b=$null}
}
while($b -like $null)

$tablica = New-Object "Int32[,]" $a,$b

for($i=0;$i -lt $a;$i++)
{
    for($j=0;$j -lt $b;$j++)
    { $tablica[$i,$j] = Get-Random -Min 1 -Max 9 }

}

function transponowanie([int32[,]]$tab , [int32]$y ,
[int32]$x)
{
    $tablica_nowa = New-Object "Int32[,]" $b,$a

    for($i=0;$i -lt $x;$i++)
    {
        for($j=0;$j -lt $y;$j++)
        {
            $tablica_nowa[$i,$j] = $tab[$j,$i]
        }
    }
```

```
        return ,$tablica_nowa
    }

    function wyswietl([array]$tab , [int32]$x , [int32]$y)
    {
        for($i=0;$i-lt$x;$i++)
        {
            for($j=0;$j-lt$y;$j++)
            { Write-Host -NoNewLine $tab[$i,$j]" " }
            "`n"
        }
    }
    "`nTablica początkowa: `n"
    wyswietl $tablica $a $b
    $wynik = transponowanie $tablica $a $b
    "Tablica transponowana: `n"
    wyswietl $wynik $b $a
```

Na początku skryptu użytkownik pytany jest o liczbę wierszy oraz kolumn tablicy początkowej. Poprzez instrukcję `Try{} Catch{}` zapytanie będzie ponawiane w momencie gdy użytkownik wpisze na przykład ciąg znaków zamiast liczby. Na podstawie podanych wartości tworzona jest tablica początkowa, po czym w pętli `for` zostaje zapełniona losowymi wartościami całkowitymi z zakresu od 1 do 9. Następnie do funkcji `transponowanie` przekazane zostają: tablica początkowa oraz ilości wierszy i kolumn. Na podstawie przekazanych wartości utworzona zostaje nowa tablica, w której ilość kolumn odpowiada ilości wierszy tablicy początkowej, a ilość wierszy odpowiada ilości kolumn tablicy początkowej, po czym następuje proces transponowania. Na koniec z funkcji `transponowanie` zwracana jest nowa, transponowana tablica. Tablica ta nie jest zwracana po prostu przez `return`, ponieważ to spowodowałoby zwrócenie tablicy nie jako jednego obiektu, lecz wszystkich elementów tablicy naraz. Aby temu zapobiec należy zwrócić dwuelementową tablicę, w której pierwszy element jest elementem pustym a drugim jest transponowana tablica. Dzięki temu traktowana jest jako cały obiekt, który potem można prawidłowo przypisać do zmiennej. Dlatego w skrypcie został zastosowany zapis:

```
return ,$tablica_nowa
```

Funkcja `wyświetl` wypisuje zawartość tablicy.



Rysunek 2.34. Skrypt transponujący macierz.

Kolejnym przedstawionym skryptem będzie skrypt działający w tle, który przeszukuje pliki w na dysku:

```
Write-Host -Nonewline "Podaj nazwę: "
[String]$Obiekt = Read-Host
$Wynik = @{}

Start-Job -name Wyszukiwanie {param($Nazwa) Get-ChildItem ↵
-Path C:\ -Recurse -Filter "*$Nazwa*"} -Arg $Obiekt|Out-Null
Write-Host -NoNewline "Trwa wyszukiwanie."
Do
{
    Write-Host -NoNewline "."
    sleep -Seconds 1
}
while($(Get-Job -name Wyszukiwanie).JobStateInfo -notlike
      "Completed")

$Efekt = Receive-Job -name Wyszukiwanie
Remove-Job -name Wyszukiwanie

foreach($Linijka in $Efekt)
{ $Wynik.Add($Linijka.FullName, $Linijka.Name) }
```

```
switch ($Wynik.Count)
{
1 {"`nZnaleziono $($Wynik.Count) plik: "; break} ↵
{($_ -eq "2")  -or ($_ -eq "3")  -or ($_ -eq "4")} ↵
{"`nZnaleziono $($Wynik.Count) pliki: "; break}
{($_ -like "*2") -or ($_ -like "*3") -or ($_ -like "*4")} ↵
{"`nZnaleziono $($Wynik.Count) pliki: "; break}
default {"`nZnaleziono $($Wynik.Count) plików: "; break}
}

$Wynik | Format-Table Value,Name
```

Skrypt ten, na początku swego działania pobiera z konsoli ciąg znaków, który później będzie wyszukiwany na dysku C:\. Ponadto następuje deklarowanie tablicy mieszającej, która będzie przechowywać wyniki. Proces szukania zostaje uruchomiony jako zadanie w tle. Nazwa tego zadania to `Wyszukiwanie`, dodawany jest również parametr `$Nazwa` do którego zostanie przekazany wyszukiwany ciąg znaków. Następnie zapisywana jest składnia polecenia przeszukującego dysk C:\ z filtrowaniem obiektów po nazwach wedle zadanego ciągu znaków. Parametr `-Arg` odpowiada za przekazanie ciągu znaków pobranego na początku skryptu do zadania w tle, jako argument `Wyszukiwania`. Cała komenda przekierowywana jest potokowo do strumienia wyjściowego `Null`, aby nie wyświetlać na bieżąco wyników. Następnie w pętli `while` zaprojektowana jest animacja w postaci wyświetlania co sekundę kolejnej kropki po napisie *"Trwa wyszukiwanie."* do momentu, aż zadanie nie zostanie ukończone. Gdy przeszukiwanie zostaje zakończone, wyniki zadania `Wyszukiwanie` zwracane są do zmiennej `$Efekt` w postaci zwykłej tablicy. Za pomocą pętli `foreach` następuje przepisanie wyników ze zwykłej tablicy to tablicy mieszającej. Kluczem tablicy mieszającej staje się nazwa znalezionego pliku z pełną ścieżką dostępu, ponieważ klucze w tablicy mieszającej muszą być unikatowe, a plik o takiej samej nazwie może znajdować się w kilku miejscach. Instrukcja `switch` generuje napis podsumowujący wyszukiwanie w zależność od ilości wyników. Na końcu następuje wyświetlenie tablicy z wynikami w taki sposób, by w pierwszej kolumnie pojawiła się sama nazwa pliku, a w drugiej pełna nazwa ze ścieżką.

2.14 Moduły

W PowerShellu już w wersji 2.0 dodano możliwość podłączania modułów do konsoli. Moduły te przechowują różnego rodzaju wartości: stałe, zmienne, funkcje, nowe komendy cmdlet. Pozwalają na rozszerzenie bazowej zawartości konsoli o nowe rozwiązania i funkcjonalności. W PowerShellu istnieje pewna liczba modułów preinstalowanych. Moduły mają domyślne rozszerzenie pliku .psm1. Znajdują się one w lokalizacji: „%systemroot%\System32\WindowsPowershell\v1.0\Modules”. Moduły te nie są domyślnie załadowane do każdej sesji konsoli, należy ręcznie je podpiąć aby uzyskać dostęp do ich zawartości. Aby uzyskać listę dostępnych modułów w podkatalogu *Modules* należy użyć polecenia:

```
Get-Module -ListAvailable
```

Wyświetli ono wszystkie dostępne moduły, które można załadować poleceniem:

```
Import-Module <nazwa_modułu>
```

Warto wspomnieć o tym, że taka konstrukcja działa tylko w przypadku modułów znajdujących się w podkatalogu *Modules*, głównego katalogu PowerShella. Aby podłączyć do konsoli moduły z innej lokalizacji, należy użyć polecenia `Import-Module` podając zamiast samej nazwy interesującego nas modułu, nazwę ze ścieżką pliku.

Tworzenie modułów

Poza gotowymi modułami dostępnymi w katalogu PowerShella istnieje możliwość tworzenia modułów na własne potrzeby. Aby tego dokonać należy stworzyć plik o rozszerzeniu .psm1, a następnie otworzyć go za pomocą dowolnego edytora tekstu, chociaż zalecane jest korzystanie z PowerShell ISE. Tworząc moduły działamy w ten sam sposób, jak w przypadku tworzenia skryptu: definiujemy potrzebne wartości oraz funkcje, a następnie tworzymy kod skryptu. Aby całość funkcjonowała jako moduł należy zakończyć ją poleceniami `Export-ModuleMember`, które odpowiadają za poinformowanie bieżącej sesji konsoli o tym, gdzie szukać definicji wszystkich wartości i danych utworzonych w module. Polecenie to, ma kilka parametrów w zależności od rodzaju eksportowanych danych: `Alias`, `Variable`, `Cmdlet`[6], `Function`. Jeżeli plik z modułem nie posiada w

6 Tworzenie własnych poleceń typu cmdlet jest procesem skomplikowanym. Wymagany do niego jest pakiet PowerShell Software Development Kit oraz narzędzie do tworzenia klas typu cmdlet takiego jak Microsoft Visual Studio, dlatego też proces tworzenia własnych poleceń cmdlet zostanie w tej pracy pominięty.

swoim zapisie tej komendy, zaimportowane zostaną tylko zdefiniowane w module funkcje. Pozostałe rodzaje danych zostaną pominięte. Przykład definiowania modułu:

```
[double]$stopa = 30.48
[double]$yard = 91.44
function stopy_na_centymetry($ilość)
{ [double]$wynik = $ilość * $stopa
   return $wynik }
function miary
{ Write-Host "Stopa: 30,48cm"
  Write-Host "Yard: 91.44cm" }
function Get-Miary {
   param (
         [parameter(Mandatory=$true,ValueFromPipeline=$true
)]
         [string[]]$Miara
     )
     PROCESS
     { Write-Host "Typ: $($Miara.GetType().Name)"
       Write-Host "Wartość: $Miara" }
}
New-Alias snc stopy_na_centymetry
Export-ModuleMember -Function stopy_na_centymetry, miary,
Get-Miary -Alias snc -Variable stopa,yard
```

Jeśli plik o formacie .psm1, znajdujący się bezpośrednio na dysku C:, zawierałby wyżej wymienioną zawartość, można go zaimportować do konsoli jako moduł używając polecenia:

```
Import-Module C:\<nazwa_pliku>.psm1
```

Powyższe polecenie załaduje do konsoli funkcje: `stopy_na_centymetry`, `miary` oraz `get-miary`, alias: `snc`, zmienne: `stopa` i `yard`. Należy również zaznaczyć, że zaimportowane moduły ważne są tylko na czas bieżącej sesji PowerShell.

3 Zaawansowane operacje w PowerShell

Oprócz prostych operacji jakie można wykonać oraz skryptów jakie można stworzyć, PowerShell oferuje bardziej zaawansowane możliwości. Wśród nich możemy znaleźć operowanie na zmiennych typu `string`, wykorzystywanie elementów graficznego interfejsu jakim są obiekty klasy `System.Windows.Forms` oraz bezpośrednia ingerencja w inne programy pozwalająca na wywoływanie zaawansowanych funkcji zewnętrznego programu przez odpowiedni zapis w PowerShellu. Istnieje bardzo wiele innych rozwiązań, lecz przedstawienie ich wszystkich byłoby zbyt czasochłonne, stąd w tym rozdziale uwaga czytelnika będzie skupiona na wyżej wymienionych.

3.1 Zaawansowane operacje na zmiennych typu string

Pierwszym rodzajem bardziej skomplikowanych operacji, jakie zostaną omówione w tym rozdziale, są operacje na plikach tekstowych. Są to między innymi: importowanie pliku tekstowego do konsoli, wyszukiwanie określonego ciągu znaków, zamiana określonego ciągu znaków na inny, dopisywanie nowej zawartości do pliku testowego.

Aby wyświetlić w zawartość pliku tekstowego w konsoli PowerShell należy użyć polecenia `Get-Content`:

```
Get-Content <ścieżkapliku>\<nazwa pliku>
```

Dzięki takiemu zapisowi na standardowe wyjście konsoli zostanie wypisana zawartość pliku tekstowego. W celu przestawienia kolejnych przykładów zostanie utworzony plik tekstowy *Planety* na przykład na dysku D:, z nazwami planet układu słonecznego:

```
"Merkury" > D:\planety.txt
"Wenus"  >> D:\planety.txt
"Ziemia" >> D:\planety.txt
"Mars"   >> D:\planety.txt
"Jowisz" >> D:\planety.txt
"Saturn" >> D:\planety.txt
"Uran"   >> D:\planety.txt
"Neptun" >> D:\planety.txt
"Pluton" >> D:\planety.txt
```

Aby odczytać zawartość pliku używamy polecenia `Get-Content` wedle wcześniejszego schematu:

```
Get-Content D:\planety.txt
```

Można również przypisać całą zawartość pliku tekstowego do zmiennej używając zapisu:

```
$A = Get-Content D:\planety.txt
$A
Merkury
Wenus
Ziemia
Mars
Jowisz
Saturn
Uran
Neptun
Pluton
```

Zmienna $A przybrała typ tablicy ponieważ każdy wczytywany wyraz znajdował się w innej linii i został potraktowany jako kolejny obiekt tablicy. Na potrzeby dalszych przykładów zawartość pliku *Plantety.txt* zostanie przypisana do zmiennej typu `string`:

```
[string]$B = Get-Content D:\planety.txt
$B
Merkury Wenus Ziemia Mars Jowisz Saturn Uran Neptun Pluton
```

Mając zawartość pliku testowego w zmiennej można łatwo odnaleźć poszukiwane słowo lub zwrot używając metody `Contains` oraz `IndexOf`. Metoda `Contains` zwraca wartość `true` jeżeli zmienna zawiera w sobie zadaną wartość, natomiast użycie metody `IndexOf` zwróci numer znaku od którego zaczyna podany ciąg znaków po raz pierwszy:

```
$B.Contains("Wenus")
True
$B.IndexOf("Wenus")
8
```

Z tego zapisu można wywnioskować, że w pliku *Planety.txt* znajduje się wyraz *Wenus*, a jego pierwsza litera to 8 znak całego ciągu znaków przypisanego do zmiennej `$B`. Należy pamiętać, że identycznie jak w przypadku tablic, ciągi znaków indeksowane są od 0, czyli pierwsza litera w pliku tekstowym będzie miała indeks 0. Ponadto metoda `IndexOf` bierze

pod uwagę wielkość liter. Jeżeli ciąg znaków, którego poszukujemy występuje wielokrotnie w pliku możemy użyć metody `IndexOf` wzbogaconej o parametr określający początek poszukiwań.

```
$B.IndexOf("ur")
4
```

Kombinacja znaków *ur* występuje w pliku po raz pierwszy w 4 znaku, czyli w wyrazie *Merkury*. Jeżeli chcemy poszukać następnej takiej kombinacji znaków należy użyć polecenia:

```
$B.IndexOf("ur", 5)
36
```

Nowy parametr określa, od którego znaku metoda ma rozpocząć poszukiwania. Jeżeli poprzedni ciąg znaków *ur* zaczynał się w czwartym znaku całego pliku, to należy rozpocząć kolejne poszukiwania od znaku piątego. Kolejna kombinacja znaków pojawia się w 36 znaku całego pliku w wyrazie *Saturn*. Znak spacji liczy się jak każdy inny oraz tak samo jak reszta znaków, posiada indeks.

Można również rzutować ciąg znaków, jaki zawiera zmienna $A na tablicę, która w każdej komórce będzie zawierała jeden wyraz. Jednym ze sposobów, aby to zrobić jest użycie parametru `-replace`, który zamieni określone ciągi znaków na inne wedle schematu, na przykład:

```
$B -replace " ",","
```

Polecenie to zamieni każdy znak spacji na przecinek. W połączeniu z zapisem:

```
$C = $B -replace " ",","
```

utworzymy tablicę jednowymiarową, która w swoich komórkach będzie posiadała kolejne wyrazy ciągu znaków. W tym przypadku wykorzystując metodę `IndexOf` otrzymamy indeks całego wyrazu zamiast indeksu znaku:

```
$C.IndexOf("Mars")
3
```

Mars jest czwartą planetą układu słonecznego. Pamiętając o indeksowaniu od zera, otrzymujemy prawidłowy indeks wyrazu jakim jest 3. Parametrem `-replace` można również zamieniać całe ciągi znaków. Można w ten sposób na przykład zamienić nazwę *Jowisz* na jej angielski odpowiednik *Jupiter*.

```
$B -replace "Jowisz" , "Jupiter"
Merkury Wenus Ziemia Mars Jupiter Saturn Uran Neptun ↵
Pluton
```

Kolejną ważną metodą, używaną na zmiennych typu string jest metoda `Insert`. Metoda ta służy do wstawiania ciągu znaków w określonego miejsce. Jej parametrami są: znak od którego należy zacząć wstawianie oraz ciąg znaków jaki chcemy wstawić. Możemy ją wykorzystać, aby dodać Planetoidy do naszego Układu Słonecznego pomiędzy Marsem a Jowiszem. W tym celu należy najpierw odnaleźć indeks, na którym zaczyna się wyraz Jowisz:

```
$B.IndexOf("Jowisz")
26
$B = $B.Insert(26, "Planetoidy ")
$B
Merkury Wenus Ziemia Mars Planetoidy Jowisz Saturn Uran ↵
Neptun Pluton
```

Zmienna $A, do której dodano Planetoidy, wygląda jak wyżej. Można teraz usunąć Planetoidy z ciągu znaków używając w tym celu metody `remove`. Metoda ta posiada dwa argumenty: indeks początkowy, od którego znaki będą usuwane oraz ilość znaków jakie użytkownik chce usunąć:

```
$B = $B.Remove(26, 10)
Merkury Wenus Ziemia Mars  Jowisz Saturn Uran Neptun ↵
Pluton
```

Wyraz *Planetoidy* ma 10 liter. Po usunięciu 10 znaków począwszy od 26 miejsca w ciągu znaków, pozostała dodatkowa spacja między *Marsem* a *Jowiszem*. W tym celu można wykorzystać funkcję, którą sprawdza ilość dodatkowych znaków spacji w ciągu znaków oraz usuwa te nadmiarowe:

```
function usun_spacje([string]$zmienna)
{
    [int32]$indeks=0

    Write-Host "Ciąg znaków ma $($zmienna.Length) liter."
    Write-Host "Przed usunieciem nadmiarowych spacji:"
    Write-Host "$zmienna"
```

```
        Write-Host "Po usunięciu nadmiarowych spacji:"

        while($indeks -lt $zmienna.Length)
        {
            if($zmienna[$indeks] -like " ")
            {
                $spacja = $indeks
                while($zmienna[$spacja+1] -like " ")
                { $zmienna = $zmienna.remove($spacja+1,1) }
                $indeks = $spacja
            }
            $indeks++
        }
        return $zmienna
    }
Ciąg znaków ma 59 liter.
Przed usunieciem nadmiarowych spacji:
Merkury Wenus Ziemia Mars  Jowisz Saturn Uran Neptun Pluton
Po usunięciu nadmiarowych spacji:
58
Merkury Wenus Ziemia Mars Jowisz Saturn Uran Neptun Pluton
```

W przedstawionym układzie słonecznym nadal znajduje się błąd. Okazuje się bowiem, że Pluton nie jest planetą. Można go zatem wyciąć i zapisać do pliku o nazwie *planety_karłowate.txt*. Aby tego dokonać należy użyć metody `substring` służącej do wycinania określonego fragmentu tekstu. Jej składnia wygląda identycznie jak w przypadku metody remove:

```
$B.IndexOf("Pluton")
53
$B.Substring(53,6)
Pluton
$B.Substring(53,6) | Out-File D:\planety_karłowate.txt
$B = $B.Remove(53,6)
```

Na koniec należy zapisać zmienną $B przechowującą układ słoneczny do pliku, jednakże zmienna $B przechowuje nazwy planet w jednej linii oddzielając kolejne wyrazy spacją. W pliku natomiast każda planeta zapisana była w nowej linii. Żeby zapisać dane do pliku w

identyczny sposób jak wcześniej, czyli każdą planetę w nowej linii, należy użyć metody `split`. Jako wartość parametru trzeba podać ciąg znaków, który zostanie zamieniony na znak nowej linii:

```
$B.split(" ")
Merkury
Wenus
Ziemia
Mars
Jowisz
Saturn
Uran
Neptun
```

Zapis ten zamieni każdy znak spacji na nową linię. Wykorzystując taki zapis można przekierować dane do pliku korzystając z jednego z rozwiązań:

```
$B.split(" ") | Out-File D:\planety.txt
$B.split(" ") > D:\planety.txt
```

Każde zadanie można zapisać na wiele sposobów. Ten zaprezentowany powyżej ma na celu przedstawienie kilku metod jakich można używać na danych typu string oraz plikach tekstowych. Całą operację wraz z zapisem do nowego pliku można ująć w jednym poleceniu:

```
Get-Content D:\planety.txt | ForEach {If($_ -like ↵
"Pluton"){}Else{Write-Output $_}} | Out-File ↵
D:\planety_nowa.txt
```

Aby dowiedzieć się więcej o metodach z jakich można korzystać w przypadku operowania na plikach tekstowych w PowerShellu należy użyć polecenia, które wyświetli wszystkie elementy składowe podanej wartości, w tym przypadku pustego elementu klasy string:

```
[string]::Empty | Get-Member
```

3.2 Operacje na plikach

PowerShell podobnie jak pozostałe interpretery poleceń oferuje możliwość różnorakich operacji na plikach. Zwykle nie są to skomplikowane operacje i służą one do zarządzania strukturą plików oraz katalogów. W tym dziale zostanie przedstawiona podstawowa lista

możliwych operacji. Jako lokalizacja początkowa zostanie wykorzystana lokalizacja *C:\Pliki*.

Podstawowym poleceniem służącym do wyświetlenia zawartości bieżącego katalogu jest `Get-ChildItem`.

```
      PS C:\pliki> Get-ChildItem
          Directory: C:\pliki
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----         2015-08-23     12:33           Katalog1
d----         2015-08-23     12:34           Katalog2
```

Polecenie to wyświetli zawartość bieżącego katalogu czyli *C:\Pliki*. Można zauważyć że wewnątrz znajdują się dwa foldery.

```
      PS C:\pliki> Get-ChildItem Katalog1
          Directory: C:\pliki\Katalog1
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         2015-08-23     12:34        26 PlikTekstowy.txt

      PS C:\pliki> Get-ChildItem Katalog2
          Directory: C:\pliki\Katalog2
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----         2015-08-23     12:34           Podkatalog
-a---         2015-08-23     12:34         0 PlikTekstowy2.txt

      PS C:\pliki> Get-ChildItem Katalog2\Podkatalog
          Directory: C:\pliki\Katalog2\Podkatalog
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-ar--         2015-08-23     12:34         0 PlikTekstowy3.txt
```

Struktura drzewa katalogów prezentuje się jak powyżej. Do kopiowania plików służy polecenie Copy-Item:

```
Copy-Item Katalog2\Podkatalog\PlikTekstowy3.txt ↲
Katalog2\PlikTesktowy3.txt
```

Podczas kopiowania można zmienić nazwę pliku docelowego. Następnie można usunąć plik zawarty w *Podkatalogu*. Jednak nie można tego dokonać ponieważ *Pliktekstowy3* posiada atrybut *Tylko od odczytu*. Aby tego dokonać należy przypisać plik do zmiennej korzystając z polecenia `Get-Item`:

```
$A = Get-Item Katalog2\Podkatalog\PlikTekstowy3.txt
$A.Attributes
Archive
Hidden
```

Po przypisaniu pliku do zmiennej można użyć metody `Attributes` aby wyświetlić bądź zmieniać atrybuty pliku:

```
$A.Attributes = "Archive"
$A.Attributes
Archive
```

Atrybuty można zmienić używając również polecenia `attrib`:

```
attrib -r Katalog2\Podkatalog\PlikTekstowy3.txt
```

Parametr `r` określa atrybut *ReadOnly* czyli *Tylko do odczytu*, natomiast znak + lub – określa czy otrzymuje dany atrybut czy go traci. Po tej czynności można usunąć plik tekstowy używając polecenia `Remove-Item`.

```
Remove-Item Katalog2\Podkatalog\PlikTekstowy3.txt
```

Można również użyć aliasu:

```
Rm Katalog2\Podkatalog\PlikTekstowy3.txt
```

W przypadku, gdy PowerShell zwraca komunikat błędu o braku odpowiednich uprawnień do wykonywania operacji na dysku C:, należy użyć parametru -`Force`

Aby zmienić nazwę pliku lub katalogu należy użyć polecenia `Rename-Item`:

```
Rename-Item Katalog2\Podkatalog Katalog3
```

lub skorzystać a aliasu tego polecenia:

```
Ren Katalog2\Podkatalog Katalog3
```

natomiast do przenoszenia plików lub katalogów służy polecenie Move-Item:

```
Move-Item C:\Pliki\Katalog2 C:\Pliki\Katalog1
```

Pozostała pula poleceń służących do operowania na plikach i folderach znajduje się w dodatku A tej pracy.

3.3 Interfejs graficzny

Jednym z wielu elementów wyróżniających PowerShella od pozostałych interpreterów poleceń jest możliwość tworzenia graficznego interfejsu. Do tworzenia własnych interfejsów graficznych należy skorzystać z obiektów klasy `System.Windows.Forms`. Ilość obiektów znajdujących się w tej klasie jest bardzo duża, stąd przedstawione zostaną tylko podstawowe jej elementy.

Tabela 19. Podstawowe rodzaje kontrolek i ich atrybuty.

Rodzaj obiektu:	Obiekt klasy System.Windows .Forms:	Podstawowe atrybuty:	Wartość atrybutu:
Okno	Form	Szerokość: Width	Wartość w pikselach
		Wysokość: Height	Wartość w pikselach
		Automatyczny rozmiar: AutoScale	$True lub $False
		Tekst: Text	<String>
Etykieta	Label	Szerokość: Width	Wartość w pikselach
		Wysokość: Height	Wartość w pikselach
		Tekst: Text	<String>
		Kolor tekstu: ForeColor	System.Drawing.Color
		Kolor tła: BackColor	System.Drawing.Color
Pole tekstowe	TextBox	Szerokość: Width	Wartość w pikselach
		Wysokość: Height	Wartość w pikselach
		Tekst: Text	<String>
Przycisk	Button	Szerokość: Width	Wartość w pikselach
		Wysokość: Height	Wartość w pikselach
		Tekst: Text	<String>
		Efekt kliknięcia: Add_Click	{<blok poleceń>}
Okno z zaznaczeniem	Checkbox	Czy zaznaczone ?: Checked	$True lub $False
Okno grupy[7]	GroupBox	Rozmiar: Size	System.Windows.Drawing(wartość w pikselach, wartość w pikselach)
		Tekst: Text	<String>
Przycisk wyboru	RadioButton	Czy zaznaczone ?:Checked	$True lub $False
		Tekst: Text	<String>

Dokładną listę na temat wszystkich możliwych do wykorzystania metod oraz właściowści danej klasy można uzyskać poprzez użycie polecenia:

7 Służy do grupowania przycisków wyboru tak, aby nie można było wybrać więcej niż jednego naraz.

```
[<klasa_obiektu>] | Get-Memeber
```

Na przykład:

```
[System.Windows.Forms.Form] | Get-Member
```

Natomiast listę wszystkich klas zawartych w przestrzeni System.Windows.Forms można odnaleźć na stronie Microsoftu [23]:

https://msdn.microsoft.com/en-us/library/system.windows.forms(v=vs.110).aspx

Tworzenie interfejsu graficznego należy rozpocząć od dodania klasy `Forms` do aktywnej sesji konsoli, ponieważ domyślnie nie jest ona załadowana. W tym celu należy zastosować polecenie:

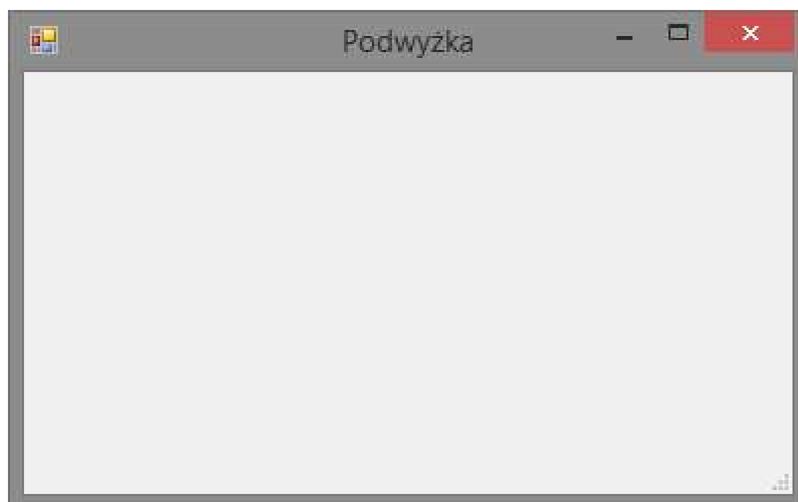
```
Add-Type -AssemblyName System.Windows.Forms
```

Podstawowe rodzaje kontrolek zostaną przedstawione w procesie tworzenia programu z interfejsem graficznym.

Na początku należy utworzyć główne okno będące podstawowym elementem interfejsu. Aby to zrobić należy stworzyć obiekt typu `Form`:

```
$Okno = New-Object System.Windows.Forms.Form
$Okno.Text = "Podwyżka"
$Okno.Width = 400
$Okno.Height = 250
$Okno.ShowDialog()
```

Wykorzystując podstawowe parametry utworzonego obiektu można zmienić między innymi: tytuł okna oraz jego rozmiar. Metoda `ShowDialog()` wywołuje zdefiniowane wcześniej okno programu.

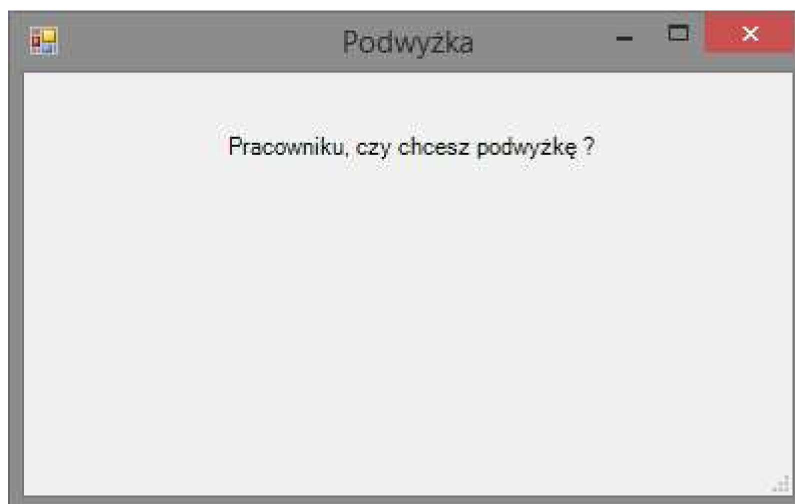


Rysunek 3.1. Bazowe okno interfejsu graficznego.

Kolejnym krokiem będzie zdefiniowanie etykiety, której celem jest wyświetlanie tekstu wewnątrz okna.

```
$Pytanie = New-Object System.Windows.Forms.Label
$Pytanie.Location = New-Object System.Drawing.Size(100,30)
$Pytanie.Width = 200
$Pytanie.Text = "Pracowniku, czy chcesz podwyżkę ?"
$Okno.Controls.Add($Pytanie)
$Okno.ShowDialog()
```

Pozycję etykiety definiować należy przez skorzystanie z obiektu `System.Drawing.Size()`. Wewnątrz nawiasów znajdują się oddzielone przecinkiem współrzędne X i Y lewego, górnego wierzchołka obiektu, którego pozycję określa metoda `Location`. Żeby obiekt znajdował się wewnątrz wcześniej utworzonego okna należy użyć metody `Controls.Add()`.



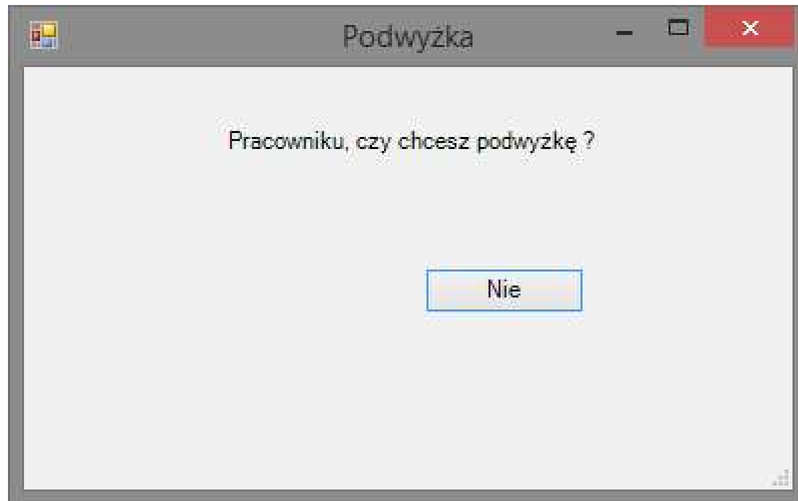
Rysunek 3.2. Okno z wyświetloną etykietą.

Następnymi elementami, które trzeba będzie zdefiniować są przyciski służące do wyboru odpowiedzi na pytanie przedstawione za pomocą etykiety.

```
$Nie = New-Object System.Windows.Forms.Button
$Nie.Location = New-Object System.Drawing.Size(200,100)
$Nie.Width = 80
$Nie.Text = "Nie"
$Nie.Add_Click({$Okno.Close()})
```

```
$Okno.Controls.Add($Nie)
$Okno.ShowDialog()
```

Powyższe instrukcje tworzą przycisk z wyświetlonym tekstem *Nie*. Zdefiniowany również zostaje efekt kliknięcia na przycisk, w tym przypadku powoduje on zamknięcie okna.

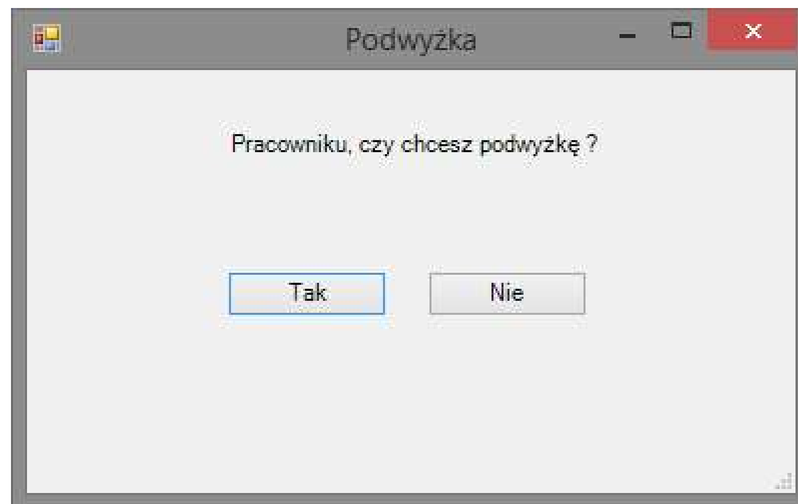


Rysunek 3.3. Dodano przycisk do puli obiektów wewnątrz okna.

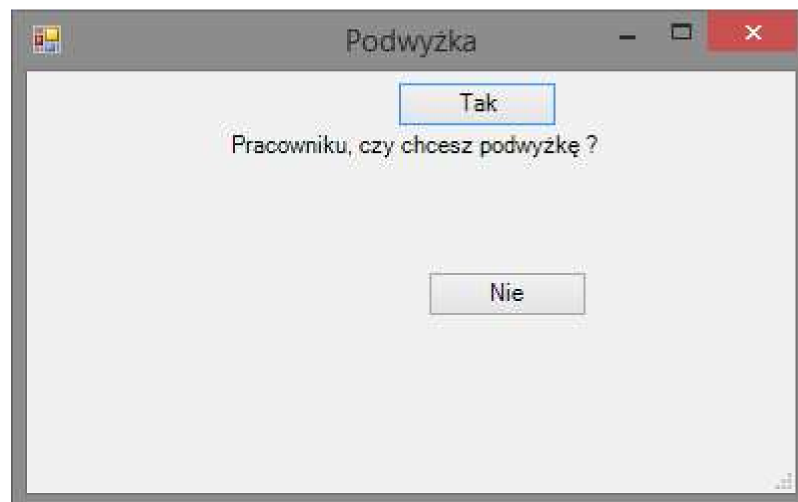
Następnie należy zdefiniować drugi przycisk:

```
$Tak = New-Object System.Windows.Forms.Button
$Tak.Location = New-Object System.Drawing.Size(100,100)
$Tak.Width = 80
$Tak.Text = "Tak"
$Tak.Add_Click({
$X = Get-Random -Min 0 -Max 300
$Y = Get-Random -Min 0 -Max 180;
$Tak.Location = New-Object System.Drawing.Size($X,$Y)})
```

Drugi przycisk będzie pełnił specjalną rolę. Jego efekt kliknięcia generuje nową pozycję klawisza. Dzięki temu wciśnięcie klawisza *Tak* spowoduje zmianę jego pozycji:



Rysunek 3.4. Finalny widok okna programu

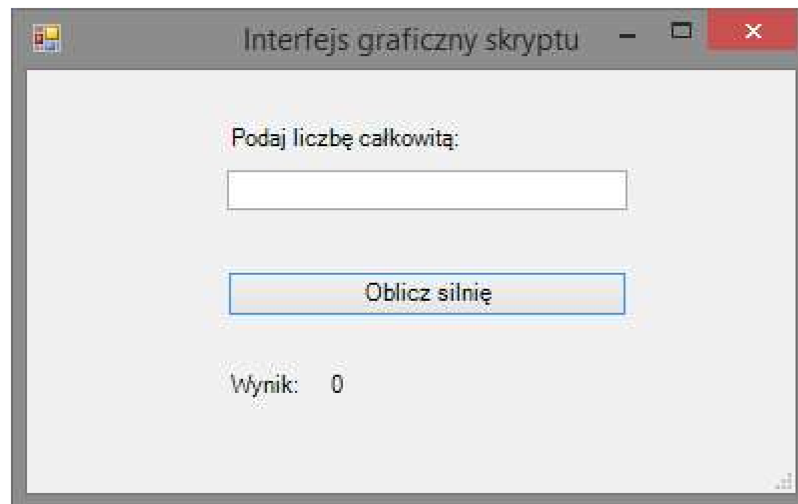


Rysunek 3.5. Przykład pozycji klawisza *Tak* po naciśnięciu go.

Następnym przykładem prezentującym zastosowania kontrolek interfejsu graficznego będzie unowocześniona wersja skryptu obliczającego silnię:

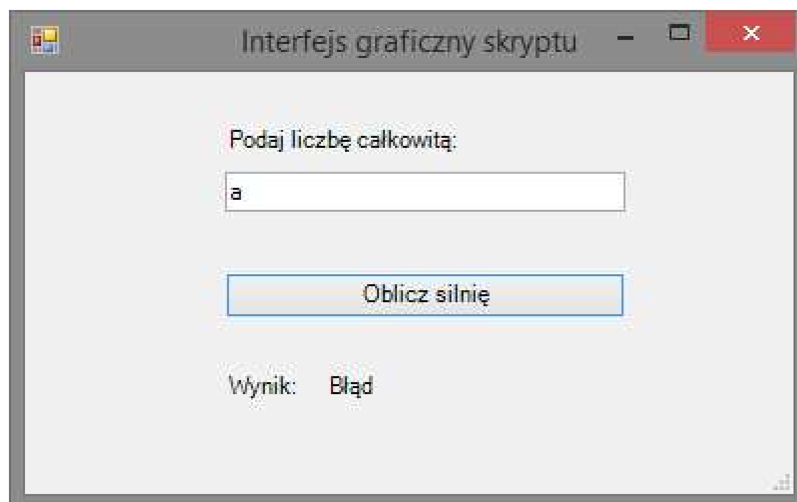
```
$Okno = New-Object System.Windows.Forms.Form
$Okno.Text = "Interfejs graficzny skryptu"
$Okno.Width = 400
$Okno.Height = 250

$PoleTekstowe = New-Object System.Windows.Forms.TextBox
$PoleTekstowe.Location = New-Object
System.Drawing.Size(100,50)
$PoleTekstowe.Width = 200
```

```
$Podaj = New-Object System.Windows.Forms.Label
$Podaj.Location = New-Object System.Drawing.Size(100,27)
$Podaj.Width = 200
$Podaj.Text = "Podaj liczbę całkowitą:"

$Wynik = New-Object System.Windows.Forms.Label
$Wynik.Location = New-Object System.Drawing.Size(100,150)
$wynik.Width = 50
$Wynik.Text = "Wynik: "

$Wyswietlenie = New-Object System.Windows.Forms.Label
$Wyswietlenie.Location = New-Object
System.Drawing.Size(150,150)
$Wyswietlenie.Text = "0"

Function Oblicz
{
    $Zmienna = [System.Convert]::ToInt32($PoleTekstowe.Text)
    return  D:\silnia.ps1 $Zmienna
}

$Przycisk = New-Object System.Windows.Forms.Button
$Przycisk.Location = New-Object System.Drawing.Size(100,100)
$Przycisk.Width = 200
$Przycisk.Text = "Oblicz silnię"
$Przycisk.Add_Click({
Try{ $Wyswietlenie.Text = Oblicz }
Catch{$Wyswietlenie.Text = "Błąd";}})

$Okno.Controls.Add($Przycisk)
$Okno.Controls.Add($PoleTekstowe)
$Okno.Controls.Add($Wynik)
$Okno.Controls.Add($Podaj)
$Okno.Controls.Add($Wyswietlenie)
$Okno.ShowDialog()
```



Rysunek 3.6. Graficzna wersja skryptu obliczającego silnię

Na początku skryptu definiowane jest główne okno, jego rozmiar oraz tytuł. Następnie tworzone są kolejne elementy okna: pole tekstowe pobierające wartość do obliczenia silni, napis „*Podaj liczbę całkowitą*", napis „*Wynik*", przycisk wywołujący obliczenia oraz napis, który na początku przedstawia *0*, a po kliknięciu na przycisk *Oblicz silnię* zmieni swą wartość na wynik obliczeń. Oprócz elementów okna, zdefiniowana jest również funkcja `Oblicz`, wywołująca skrypt obliczający silnię (zawarty w podrozdziale 2.13). W funkcji tej, wykorzystywane zostało konwertowanie wartości typu `string` z pola tekstowego na typ `Int32`, który posłuży jako argument wywołania skryptu obliczającego silnię. Metoda `Add_Click` przycisku wywołuje blok poleceń podany w nawiasach klamrowych za każdym razem, gdy zostaje wciśnięty przycisk *Oblicz silnię*. W razie podania niepoprawnego argumentu lub przekroczenia zakresu zmiennej typu `Int32` zamiast wyniku wyświetlony zostanie błąd.



Rysunek 3.7. Wynik zamienia się w komunikat o błędzie w przypadku złego parametru.

Poza podstawowymi elementami jakie można zastosować w interfejsie graficznym można również korzystać z bardziej zaawansowanych obiektów. Są to między innymi różnego rodzaju menu służące na przykład do otwierania i zapisywania plików, wyboru koloru z palety barw albo wyboru czcionki.

Tabela 20. Wybrane, zaawansowane obiekty typu *System.Windows.Forms* i ich najważniejsze parametry.

Obiekt klasy System.Windows.Forms.	Parametry	Opis
ColorDialog	AllowFullOpen	Zezwala na rozszerzenie okna o paletę barw RGB
	AnyColor	Sprawdza czy został wybrany dowolny kolor
	Color	Przechowuje aktualnie wybrany kolor
	FullColor	Powoduje uruchomienie okna od razu w trybie rozszerzonym
FontDialog	Color	Przechowuje informacje o kolorze czcionki
	Font	Przechowuje informacje o czcionce.
	ShowColor	Dodaje/usuwa do standardowego panelu wyboru czcionki możliwość wyboru koloru
	ShowEffects	Dodaje/usuwa do standardowego panelu wyboru czcionki możliwość wyboru efektów czcionki
	MaxSize	Określa minimalną wielkość czcionki
	MinSize	Określa maksymalną wielkość czcionki
OpenFileDialog	FileName	Nazwa pliku
	Title	Tytuł okna
	Filter	Filtr rozszerzeń plików np. „EXE (*.exe) | *.exe”

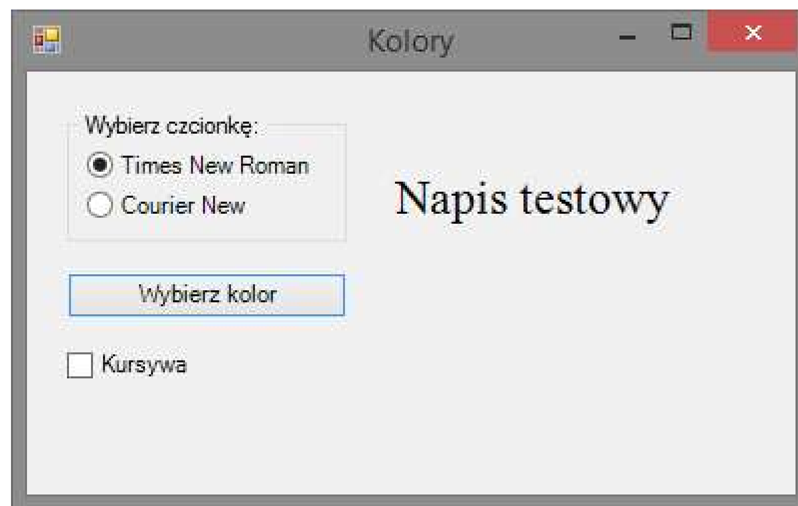
	InitialDirectory	Katalog domyślny po otwarciu menu np. „C:\Windows”
SaveFileDialog	FileName	Nazwa pliku
	Title	Tytuł okna
	Filter	Filtr rozszerzeń plików np. „EXE (*.exe) | *.exe”
	InitialDirectory	Katalog domyślny po otwarciu menu np. „C:\Windows”
FolderBrowserDialog	SelectedPath	Wybrany katalog

Kolejnym przykładem będzie skrypt zmieniający właściwości wyświetlanego napisu:

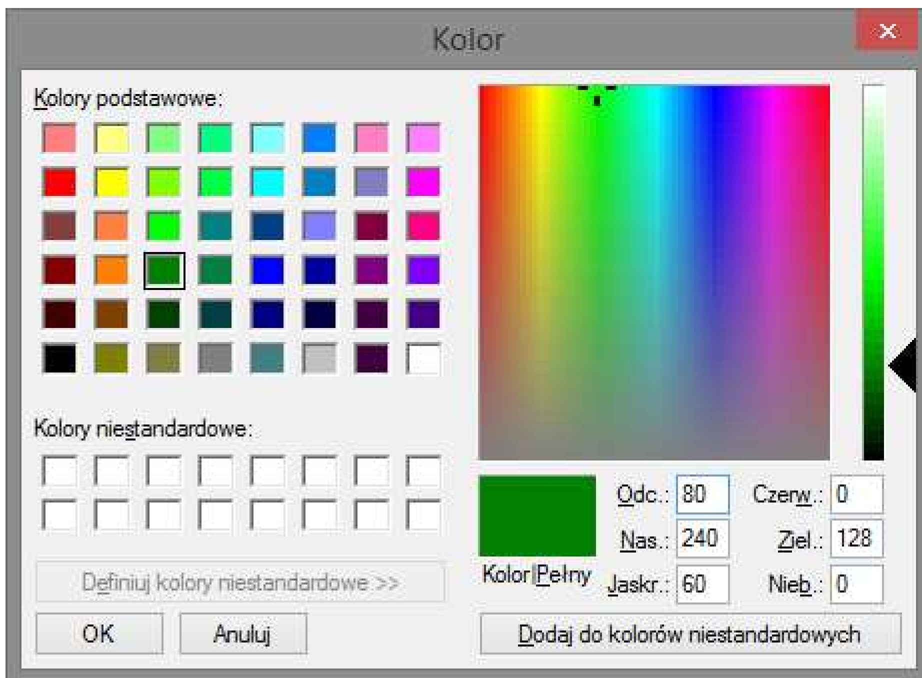
```
$Okno = New-Object System.Windows.Forms.Form
$Okno.Text = "Kolory"
$Okno.Width = 400
$Okno.Height = 250
$WybórKoloru = New-Object System.Windows.Forms.ColorDialog

$Napis = New-Object System.Windows.Forms.Label
$Napis.Location = New-Object System.Drawing.Size(180,50)
$Napis.Width = 300
$Napis.Height = 200
$Napis.Text = "Napis testowy"
$Napis.Font = New-Object System.Drawing.Font("Times New
Roman",18)

$Kursywa = New-Object System.Windows.Forms.CheckBox
$Kursywa.Location = New-Object System.Drawing.Size(20,135)
$Kursywa.Text = "Kursywa"
$Kursywa.Width = 150

$TNR = New-Object System.Windows.Forms.RadioButton
$TNR.Text = "Times New Roman"
$TNR.Location = New-Object System.Drawing.Size(10,15)
$TNR.Width = 120
$TNR.Checked = $true

$Courier = New-Object System.Windows.Forms.RadioButton
$Courier.Text = "Courier New"
$Courier.Location = New-Object System.Drawing.Size(10,35)
```

```
$Courier.Width = 120

$Wybierz = New-Object System.Windows.Forms.GroupBox
$Wybierz.Location = New-Object System.Drawing.Size(20,20)
$Wybierz.Size = New-Object System.Drawing.Size(140,65)
$Wybierz.Text = "Wybierz czcionkę: "
$Wybierz.Controls.Add($Courier)
$Wybierz.Controls.Add($TNR)

$Przycisk = New-Object System.Windows.Forms.Button
$Przycisk.Location = New-Object System.Drawing.Size(20,100)
$Przycisk.Width = 140
$Przycisk.Text = "Wybierz kolor"
$Przycisk.Add_Click({$WybórKoloru.ShowDialog();
$Napis.ForeColor = $WybórKoloru.Color})

Function załaduj
{
If($TNR.Checked -eq $true)
{
If($Kursywa.Checked -eq $True)
{ $Napis.Font = New-Object System.Drawing.Font("Times New ↵
      Roman",18,[System.Drawing.FontStyle]::Italic)}
else
{ $Napis.Font = New-Object System.Drawing.Font("Times New ↵
      Roman",18)}
}
If($Courier.Checked -eq $true)
{
If($Kursywa.Checked -eq $True)
{ $Napis.Font = New-Object System.Drawing.Font("Courier ↵
      New",18,[System.Drawing.FontStyle]::Italic)}
else
{ $Napis.Font = New-Object System.Drawing.Font("Courier ↵
      New",18)}
}
```

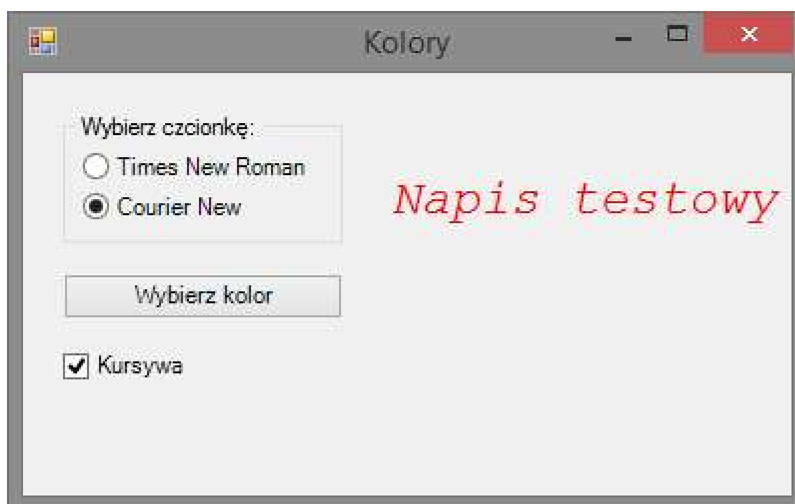
```
}
$TNR.Add_Click({załaduj})
$Courier.Add_Click({załaduj})
$Kursywa.Add_Click({załaduj})

$Okno.Controls.Add($Przycisk)
$Okno.Controls.Add($wybierz)
$Okno.Controls.Add($Kursywa)
$Okno.Controls.Add($Napis)
$Okno.ShowDialog()
```



Rysunek 3.4. Graficzny interfejs skryptu



Rysunek 3.5. Okno palety kolorów



Rysunek 3.6. Napis po modyfikacjach

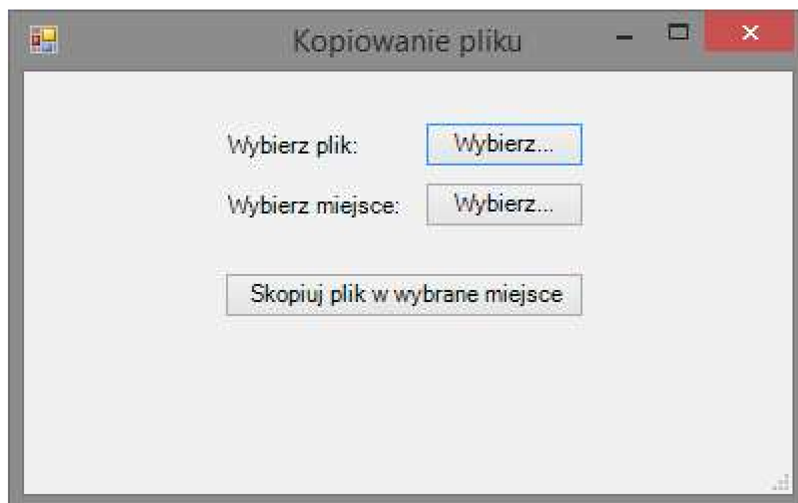
Ostatnim przykładem będzie kopiowanie plików z wykorzystaniem menu kontekstowych wyboru pliku:

```
$Okno = New-Object System.Windows.Forms.Form
$Okno.Text = "Kopiowanie pliku"
$Okno.Width = 400
$Okno.Height = 250
```

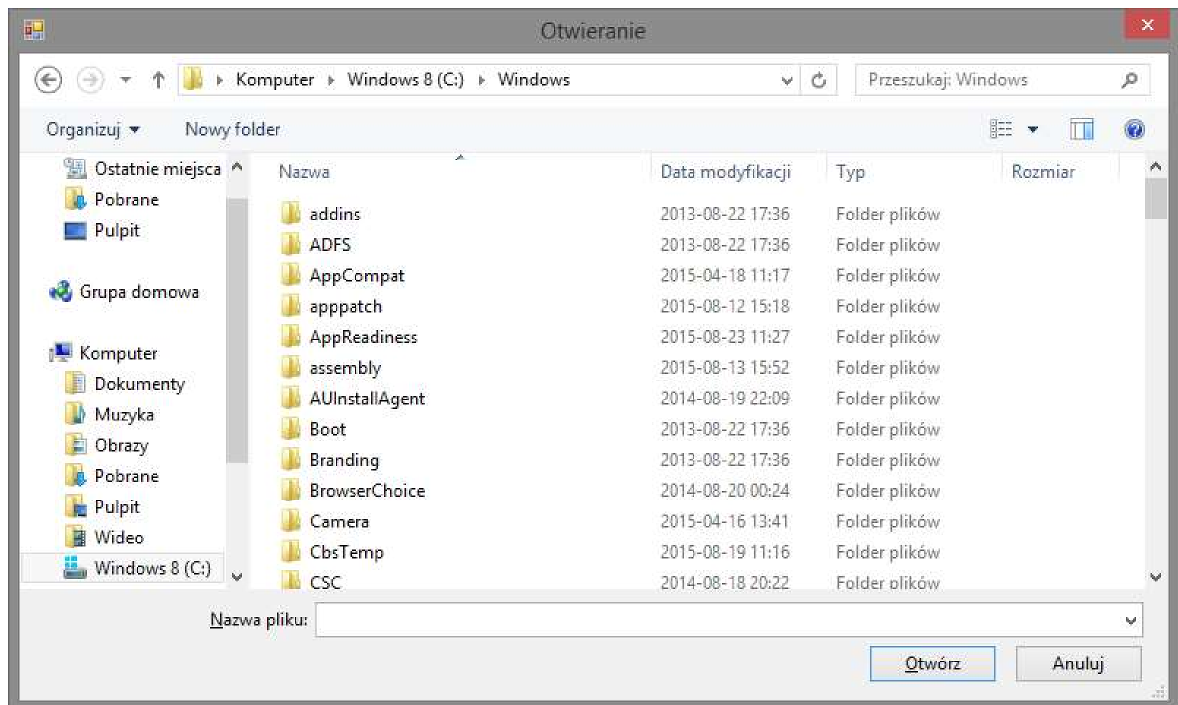
```
$Plik = New-Object System.Windows.Forms.OpenFileDialog
$Miejsce = New-Object System.Windows.Forms.SaveFileDialog

$Napis = New-Object System.Windows.Forms.Label
$Napis.Location = New-Object System.Drawing.Size(100,30)
$Napis.Width = 200
$Napis.Text = "Wybierz plik:"

$Napis2 = New-Object System.Windows.Forms.Label
$Napis2.Location = New-Object System.Drawing.Size(100,60)
$Napis2.Width = 200
$Napis2.Text = "Wybierz miejsce:"

$Napis3 = New-Object System.Windows.Forms.Label
$Napis3.Location = New-Object System.Drawing.Size(100,180)
$Napis3.Width = 200
$Napis3.Text = "Operacja zakończona powodzeniem"
$Napis3.Visible = $false

$Wybierz_plik = New-Object System.Windows.Forms.Button
$Wybierz_plik.Location = New-Object System.Drawing.Size(200,25)
$Wybierz_plik.Width = 80
$Wybierz_plik.Text = "Wybierz..."
$Wybierz_plik.Add_Click({$Plik.ShowDialog()})

$Zapisz_plik = New-Object System.Windows.Forms.Button
$Zapisz_plik.Location = New-Object System.Drawing.Size(200,55)
$Zapisz_plik.Width = 80
$Zapisz_plik.Text = "Wybierz..."
$Zapisz_plik.Add_Click({$Miejsce.ShowDialog()})

$Kopiuj = New-Object System.Windows.Forms.Button
$Kopiuj.Location = New-Object System.Drawing.Size(100,100)
$Kopiuj.Width = 180
$Kopiuj.Text = "Skopiuj plik w wybrane miejsce"
$Kopiuj.Add_Click({
```

```
Copy-Item $Plik.FileName $Miejsce.FileName
$Napis3.Visible = $true
})

$Okno.Controls.Add($Kopiuj)
$Okno.Controls.Add($Wybierz_plik)
$Okno.Controls.Add($Zapisz_plik)
$Okno.Controls.Add($Napis)
$Okno.Controls.Add($Napis2)
$Okno.Controls.Add($Napis3)

$Okno.ShowDialog()
```

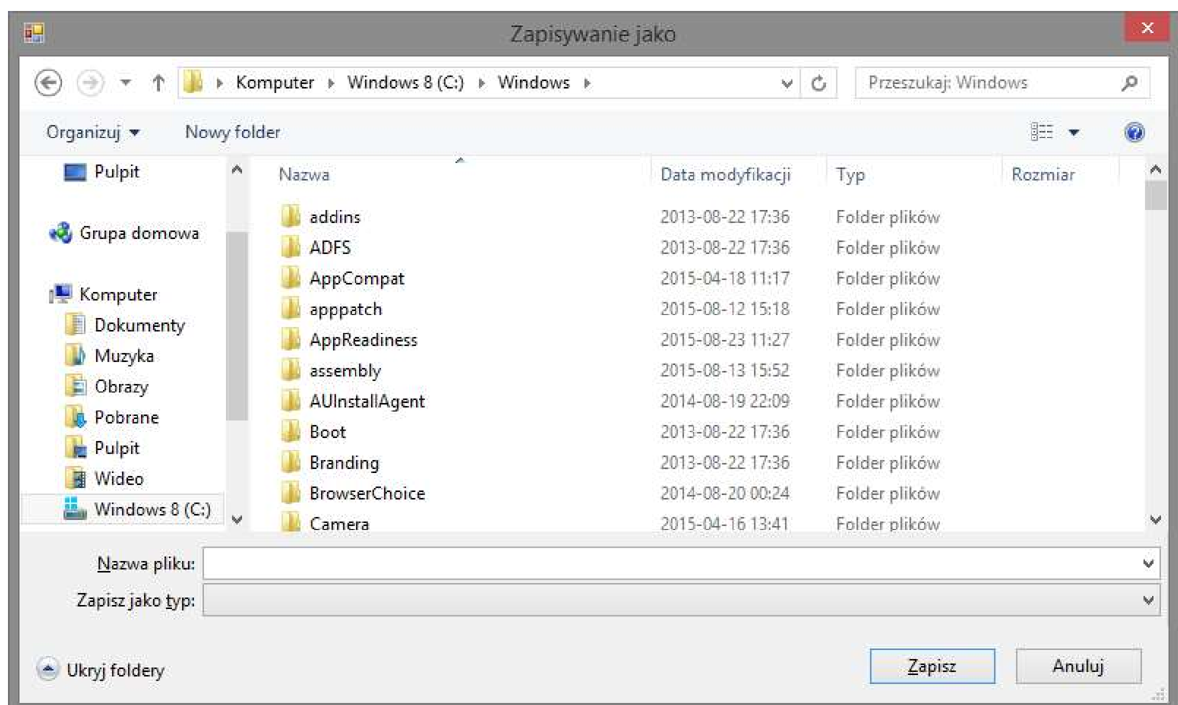
Skrypt ten działa bardzo prosty sposób. Wciśniecie klawisza `$Wybierz_plik` wywołuje menu wyboru pliku, klawisz `$Zapisz_plik` otwiera menu zapisu. Wciśnięcie klawisza `$Kopiuj` wywołuje PowerShellową komendę `Copy-Item`, do której parametrami stają się wcześniej pobrane ścieżki. Poza kopiowaniem, użycie tego klawisza zmienia status widoczności napisu powiadamiającego o pomyślnej operacji na `$true`.



Rysunek 3.8. Główne okno programu



Rysunek 3.9. Okno wyboru pliku do kopiowania



Rysunek 3.10. Okno wyboru folderu docelowego oraz nazwy pliku

3.4 Operacje na obiektach typu COM

Windows PowerShell pozwala na uruchamianie innych aplikacji w sposób spersonalizowany. Umożliwia również modyfikowanie aplikacji bezpośrednio przez komendy zawarte w konsoli. Prostym przykładem, na którym można przedstawić tę funkcjonalność jest na przykład program WordPad:

```
$WordPad = New-Object -Com wscript.shell
$WordPad.run("C:\windows\system32\write.exe")
while($WordPad.AppActivate("WordPad") -ne $true)
{Start-Sleep -Milliseconds 500}
$WordPad.SendKeys("Napis wyświetlajacy się po otwraciu
programu WordPad przez PowerShella")
Start-Sleep -Milliseconds 500
(Get-Process -name Wordpad).CloseMainWindow()
Start-Sleep -Milliseconds 500
$WordPad.SendKeys("z")
Start-Sleep -Milliseconds 500
$WordPad.SendKeys("~")
```

Klasa `wscript.shell` posiada kilka metod które można wykorzystywać na uruchomionych aplikacjach. Powyższy przykład tworzy obiekt typu `wscript.shell`, następnie uruchamia z odpowiedniej lokalizacji program *Wordpad*. Po uruchomieniu sprawdza czy aplikacja już działa, jeśli nie to oczekuje na jej otwarcie 0.5 sekundy po czym znów następuje sprawdzenie. Warto zwrócić uwagę na nazwę według jakiej następuje sprawdzenie czy aplikacja została otwarta. Nie jest to nazwa procesu tylko nazwa aplikacji. W momencie gdy aplikacja jest już uruchomiona, wywoływana jest metoda `SendKeys`, która generuje wprowadzenie tekstu podanego w nawiasie. Następnie wysyłane jest żądanie zamknięcia aplikacji. Powoduje ono wyświetlenie okna z zapytaniem czy użytkownik chce zapisać plik. Metoda `SendKeys` wysyła klawisz *z* jako wybór opcji *Zapisz plik*, po czym pojawia się okno wyboru miejsca zapisu. Wysłanie znaku ~ odpowiada za wciśnięcie klawisza *Enter*, co powoduje zapisanie pliku korzystając z domyślnej nazwy oraz lokalizacji. Aby wszystkie operacje zdążyły wykonać się poprawnie, odseparowane są interwałem czasowym w postaci 0.5 sekundy.

Innym, na którym zostanie przestawiona ta funkcjonalność będzie przeglądarka internetowa Internet Explorer [19]. Na początku należy zdefiniować zmienną, która przyjmie typ obiektu COM. Do tej zmiennej zostanie przypisana sesja Internet Explorera:

```
$P = New-Object -ComObject InternetExplorer.Application
```

Zmienna, do której przypisana jest sesja przeglądarki podsiada wiele metod i właściowości, które można edytować. Najważniejszą z nich jest właściwość `Visible`, która po ustawieniu wartości na `$true` uruchomi przeglądarkę.

```
$P.Visible = $true
```

Zanim jednak przeglądarka zostanie uruchomiona pora zmienić jej ustawienia. Można zacząć od podstawowych właściwości takich jak rozmiar:

```
$P.Width = 800
$P.Height = 600
```

odległość od krawędzie ekranu w trybie okienkowym:

```
$P.Left = 100
$P.Top = 100
```

uruchomienie z wykorzystaniem trybu pełnoekranowego:

```
$P.Fullscreen = $true
```

Aby przejść na początku do określonej strony należy użyć metody `Navigate`:

```
$P.Navigate("www.pcz.pl")
```

Jeżeli kolejne działanie wymaga w pełni załadowanej strony, można skorzystać z metody `Busy`, która określa zajętość przeglądarki. W połączeniu z pętlą `while` otrzymujemy blokadę dalszej części skryptu dopóki strona nie zostanie w pełni załadowana:

```
while($P.Busy){Start-Sleep -Seconds 1}
```

jeśli uruchamiany przeglądarkę z określoną stroną startową, można wyłączyć pasek narzędzi wyłączając możliwość wpisywania adresu strony:

```
$P.ToolBar = $false
```

Istnieje możliwość interakcji z elementami strony wyświetlanej w przeglądarce. Cały proces zostanie przedstawiony na przykładzie strony www.pcz.pl:

```
$Internet = New-Object -ComObject
InternetExplorer.Application
$Internet.FullScreen = $true
```

```
$Internet.Navigate("https://www.pcz.pl/pl/user/ ↵
login?destination=node")

Write-Host "Logowanie do strony www.pcz.pl"
$Login = Read-Host -Prompt "Podaj nazwę użytkownika: "
$Hasło = Read-Host -AsSecureString -Prompt "Podaj hasło: "

$Internet.Visible = $true

while($Internet.Busy){Start-Sleep -Seconds 1}

$Dokument = $Internet.document

$User = $Dokument.getElementByID("edit-name")
$User.Value = $Login
$Pass = $Dokument.getElementByID("edit-pass")
$Pass.Value = $Haslo

$Klawisz = $Dokument.getElementByID("edit-submit")
$Klawisz.Click()
```

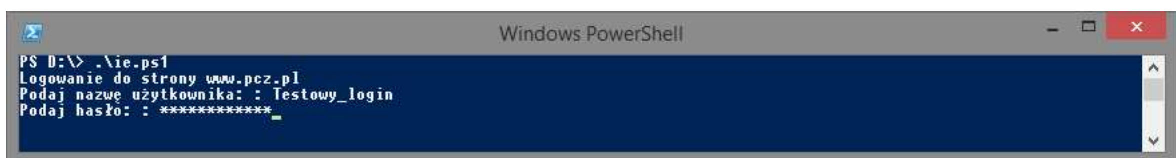
Skrypt ten w pierwszej linii przypisuje do zmiennej `$Internet` sesję przeglądarki *Internet Explorer*. Następnie ustawia parametr przeglądarki na uruchomienie w trybie pełnoekranowym. Kolejnym krokiem jest przejście do podstrony Politechniki Częstochowskiej służącej do logowania. Po tym wyświetlane są zapytania w konsoli z prośbą o podanie loginu i hasła, które później zostaną użyte do logowania. Parametr `-AsSecureString` powoduje, że znaki będą ukryte podczas wpisywania. Następnie rozpoczyna się wywołanie przeglądarki z wcześniejszymi ustawieniami poprzez ustawienie wartości parametru `Visible` na `$true`. W pętli `while` następuje oczekiwanie na pełne załadowanie strony sprawdzając co 1 sekundę status `Busy` przeglądarki. Kolejnym krokiem jest stworzenie zmiennej `$Dokument`, do której przypisany jest dokument na stronie, w którym znajdują się wszystkie elementy strony. Po tym następuje odnalezienie właściwych elementów strony i ich ID. W tym przypadku są to pola tekstowe, do których wpisywane są login i hasło oraz klawisz *Zaloguj*. Zawartość strony można

przejrzeć dzięki wyświetleniu źródła strony. Domyślnym skrótem klawiszowym do pokazania źródła strony jest klawisz F12.

```
▼<div class="form-item form-type-textfield form-item-name">
  ▶<label for="edit-name">…</label>
   <input type="text" id="edit-name" name="name" value size="60" maxlength="60" class="form-text required">
   <div class="description">Nazwa użytkownika witryny Politechnika Częstochowska.</div>
 </div>
▼<div class="form-item form-type-password form-item-pass">
  ▶<label for="edit-pass">…</label>
   <input type="password" id="edit-pass" name="pass" size="60" maxlength="128" class="form-text required">
   <div class="description">Hasło powiązane z nazwą użytkownika.</div>
 </div>
 <input type="hidden" name="form_build_id" value="form-IJVH4uK3NJe3taUU9N7Xg4GM4uf6LVsArEVvB40upD8">
 <input type="hidden" name="form_id" value="user_login">
▼<div class="form-actions form-wrapper" id="edit-actions">
   <input type="submit" id="edit-submit" name="op" value="Zaloguj" class="form-submit">
```

Rysunek 3.11. Wartości ID elementów strony internetowej, które zostały przypisane jako obiekty w skrypcie.

Po przypisaniu elementów strony do zmiennych, należy ustawić ich wartości na pobrane wcześniej login i hasło. Ostatnim krokiem w skrypcie jest wywołanie metody odpowiedzialnej za wciśniecie klawisza *Zaloguj* na stronie. Podany skrypt działa w Internet Explorer tylko do wersji 8. W wyższej wersji przeglądarki Internet Explorer należy wykorzystać metodą pobierania ID obiektu na stronie w inny sposób.



Rysunek 3.12. Działanie skryptu przed wywołaniem przeglądarki

4 Możliwości zdalne oraz administracja z wykorzystaniem Windows PowerShell

Dzięki szerokim możliwościom PowerShella, poza tworzeniem skryptów oraz wykorzystywaniem ich na bieżącym komputerze istnieje możliwość zainicjowania połączenia zdalnego oraz wykonywania skryptów na komputerze zdalnym. W znacznym stopniu ułatwia to administrację komputerów w sieci zdalnie oraz otwiera wiele nowych możliwości. Dzięki PowerShellowi można zarządzać zdalnie przebiegami pracy, które odpowiedzialne są za długie i skomplikowane operacje, konfigurować maszyny wirtualne, zarządzać ustawieniami *Active Directory* oraz wiele innych. Nie jest możliwym przestawienie wszystkich rozwiązań jakie oferuje PowerShell w dziedzinie administracji stąd też w tym rozdziale zostanie przedstawiony schemat nawiązywania połączenia zdalnego z innym komputerem, wykonywanie skryptów na urządzeniach zdalnych oraz zostanie zaprezentowana możliwość administrowania kontami użytkowników zaczynając od ich utworzenia, a kończąc na przypisywaniu im odpowiednich uprawnień.

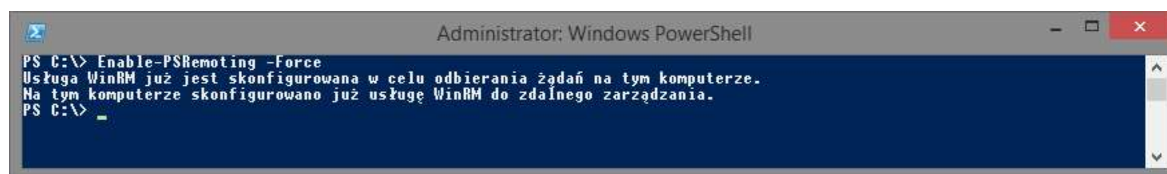
4.1 Nawiązywanie zdalnego połączenia

Każdy komputer posiadający zainstalowanego PowerShella domyślnie blokuje wszystkie przychodzące połączenia zdalne. Jest to forma zabezpieczenia komputera przed ingerencją z zewnątrz.

Aby rozpocząć przygodę z ze zdalną sesją PowerShella należy najpierw odblokować taką możliwość na komputerze docelowym. Cały proces inicjowania połączenia zdalnego przedstawiony zostanie na przykładzie sieci lokalnej. Aby umożliwić zdalne połączenia z wykorzystaniem PowerShella należy użyć polecenia:

```
Enable-PsRemoting -Force
```

Wcześniej jednak PowerShell musi zostać uruchomiony z uprawnieniami administratora.



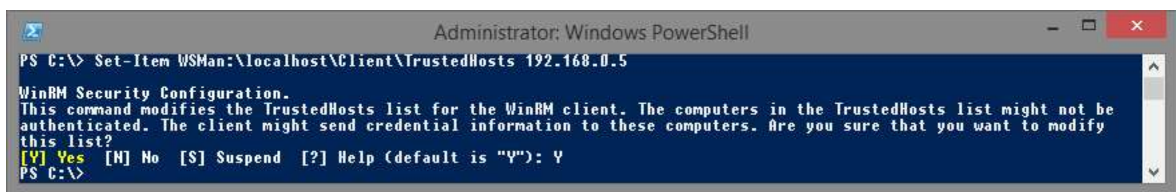
Rysunek 4.1. Odblokowywanie możliwości zdalnego łączenia się z komputerem.

W przypadku gdy komputery, które będą się ze sobą łączyć nie należą do jednej domeny należy podjąć kilka dodatkowych kroków. Pierwszym z nich jest uruchomienie możliwości połączeń zdalnych również na komputerze, z którego użytkownik zamierza zainicjować połączenie. Polecenie to nie zadziała jeśli typ sieci ustawiony jest na publiczny. Aby zmienić typ sieci na prywatny należy użyć polecenia [20]:

```
PS C:\> Set-NetConnectionProfile -name "<nazwa_sieci>" ↵
-NetworkCategory private
```

Kolejnym krokiem jaki należy wykonać w celu utworzenia połączenia zdalnego jest dodanie adresów IP do zaufanej listy urządzeń. Urządzenia można dodawać wykorzystując ich nazwę lub adres IP. Kolejne wartość trzeba odseparować przecinkiem. Należy użyć w tym celu polecenia:

```
Set-Item wsman:\localhost\client\trustedhosts <adres_IP>[8]
```

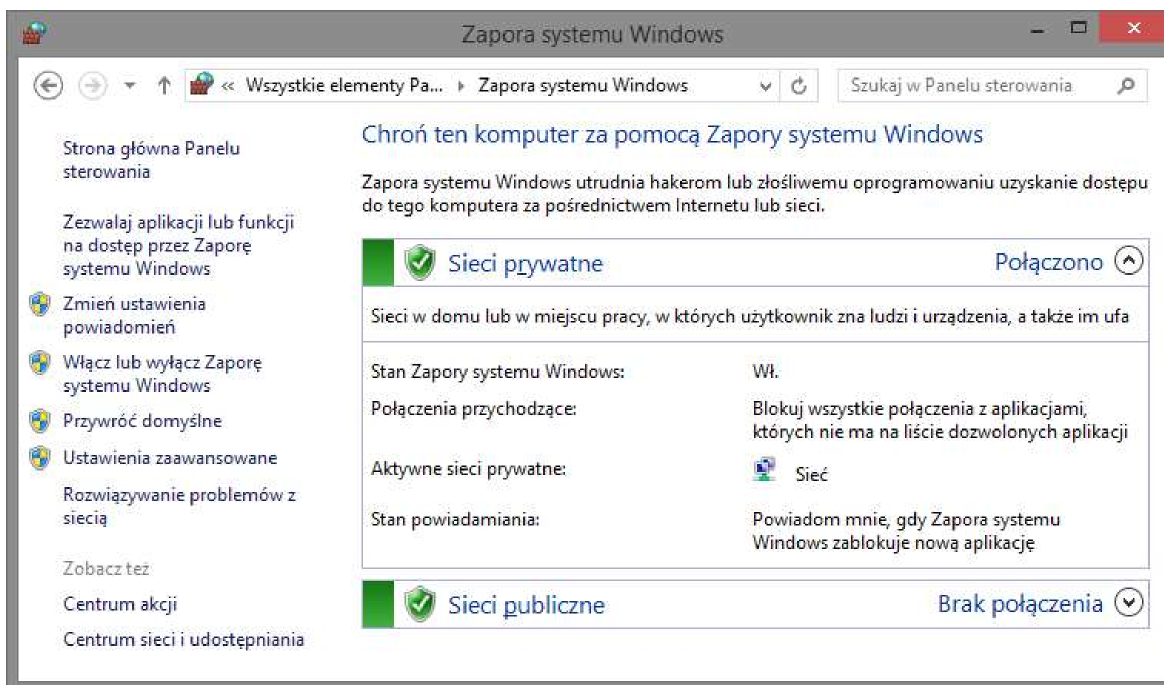


Rysunek. 4.2. Dodanie adresu IP komputera, z którego rozpocznie się połączenie zdalne do puli zaufanych urządzeń

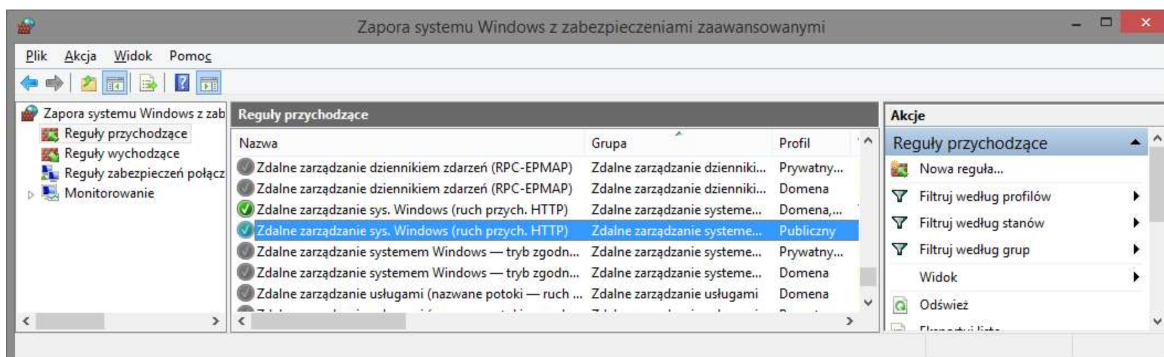
Taką samą operację trzeba wykonać na komputerze początkowym dodając do zaufanych urządzeń adres IP lub nazwę komputera docelowego.

Następną rzeczą jaką trzeba wykonać to uruchomienie reguły zezwalającej na zdalne zarządzanie systemem Windows. Regułę tę znaleźć można w ustawieniach zaawansowanych Zapory systemu Windows.

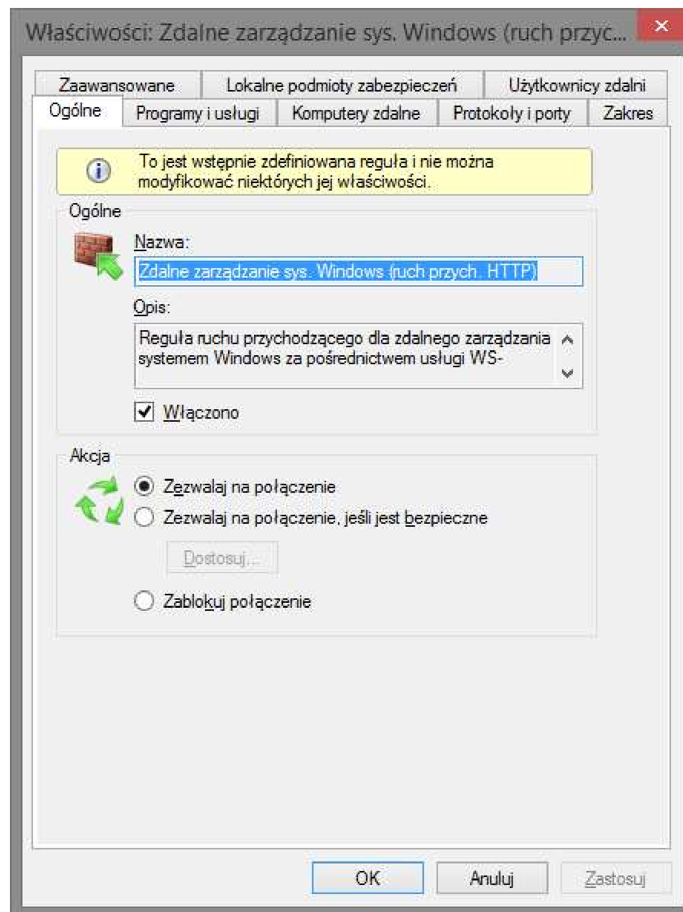
8 W tym miejscu można również podawać nazwy komputerów odseparowane przecinkami.



Rysunek 4.3. W ustawieniach *Zapory* systemu Windows należy wybrać *Ustawienia zaawansowane*



Rysunek 4.4. Następnie w *Regułach przychodzących* odnaleźć *Zdalne zarządzenie systemem Windows (ruch przych. HTTP)*



Rysunek 4.5. Ostatnim krokiem jest włączenie reguły i zezwolenie na połączenie.

Aby upewnić czy połączenie jest poprawnie skonfigurowane można skorzystać z polecenia Test-WSMan:

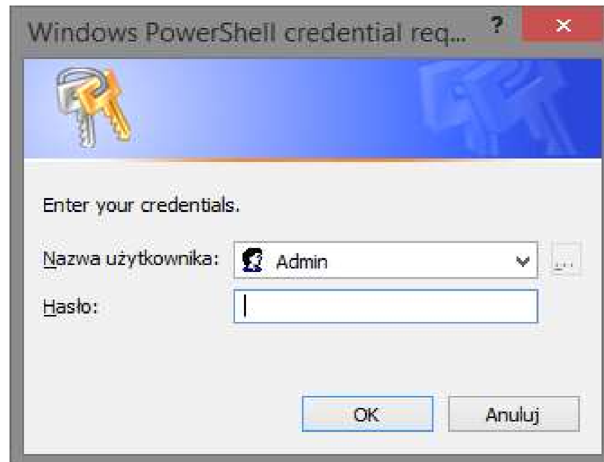
```
PS C:\> Test-WSMan 192.168.0.12
wsmid:http://schemas.dmtf.org/wbem/wsman/identity/1/ ↵
    wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

Po poprawnym skonfigurowaniu ustawień potrzebnych do połączenia zdalnego można połączyć się do drugiego komputera poleceniem `Enter-PSSesion`:

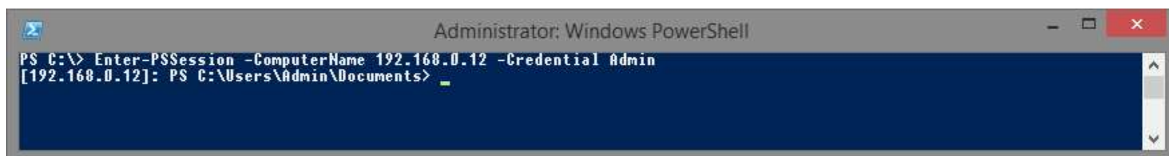
```
Enter-PSSession -ComputerName 192.168.0.12 -Credential Admin
```

Parametr `-Credential` określa konto użytkownika na które PowerShell się zaloguje. Należy pamiętać o tym, że kontom na które użytkownik chce się zalogować musi posiadać

uprawnienia administratora oraz ustawione hasło. W parametrze -`ComputerName` można umieścić zarówno nazwę komputera jak i numer IP.



Rysunek 4.6. Okno wyświetlane podczas inicjowania połączenia zdalnego z pytaniem o hasło do konta komputera docelowego.



Rysunek 4.7. Okno konsoli zdalnie zalogowanej do drugiego komputera w sieci lokalnej

4.2 Zdalne wykonywanie poleceń

Wykorzystując PowerShella nie ma potrzeby inicjowania połączenia, aby móc wykonać krótkie polecenie. Można do tego wykorzystać komendę, która szybko połączy się z komputerem zdalnym, wykona zadane polecenia lub blok poleceń po czym rozłączy się z komputerem zdalnym.

Aby skorzystać z takiej możliwość warto najpierw skonfigurować ustawienia zdalnego połączenia, a następnie zapisać je do zmiennej:

```
$Zdalny = New-PSSession -ComputerName 192.168.0.12 ↵
-Credential Admin
```

Zmienna `$Zdalny` będzie przechowywać teraz dane wymagane do rozpoczęcia połączenia zdalnego, w tym numer IP komputera oraz wskazanie na które konto nastąpi logowanie. Po utworzeniu zmiennej przechowującej dane do połączenia można użyć polecenia `Invoke-`

`Command`, które umożliwia wywołanie polecenia w określonych warunkach na przykład w wybranej sesji, która będzie połączeniem zdalnym:

```
Invoke-Command -Session $Zdalny -Scriptblock {<polecenia>}
```

W nawiasach klamrowych za parametrem `-Scriptblock` znajduje się blok poleceń, który zostanie wykonany na komputerze zdalnym, a następnie jego wyniki zostaną przesłane do podłączonego komputera na przykład polecenie:

```
Invoke-Command -Session $Zdalny -Scriptblock {Get-Process}
```

wyświetli listę aktywny procesów na komputerze zdalnym po czym zakończy połączenie.

4.3 Tworzenie oraz konfiguracja nowych użytkowników

W PowerShellu istnieje możliwość tworzenia nowych użytkowników na komputerze lokalnym wykorzystując wyłącznie polecenia w konsoli. Zaawansowane operacje dotyczące uprawnień użytkownika mogą być nadawane przez korzystanie z *Active Directory*, jednakże w tej pracy zaprezentowany zostanie jedynie proces tworzenia, modyfikacji oraz usuwania nowego użytkownika w systemie Windows. Nawet do tych podstawowych operacji wymagane będzie korzystanie z klasy *Active Directory* [25].

Pierwszym krokiem będzie sprawdzenie nazwy komputera, na którym utworzony zostanie nowy użytkownik. W tym celu można się posłużyć zmienną środowiskową:

```
$env:ComputerName
```

Aby w dalszej części procesu łatwo korzystać z tej nazwy zostanie ona przypisana do zmiennej:

```
$NazwaKomputera = $env:ComputerName
```

Kolejnym krokiem będzie utworzenie zmiennej przechowującej obiekt klasy ADSI (ang. Active Directory Services Interface), dzięki której możliwe są operacje takie jak tworzenie nowych użytkowników:

```
$Komputer = [ADSI]"WinNT://$NazwaKomputera"
```

Następnie należy stworzyć użytkownika korzystając z metody `Create`, utworzonej zmiennej `$Komputer`

```
$Użytkownik = $komputer.Create("User","Testowy")
```

Parametrami metody `Create` są: typ obiektu jaki zostaje utworzony (na przykład: `User`, `Group`), oraz nazwę użytkownika. Użytkownik po utworzeniu otrzymuje domyślną

lokalizację katalogu domowego. Po utworzeniu zmiennej przechowującej dane użytkownika należy ustawić jego hasło. Służy do tego metoda `SetPassword`:

```
$Użytkownik.SetPassword("12345")
```

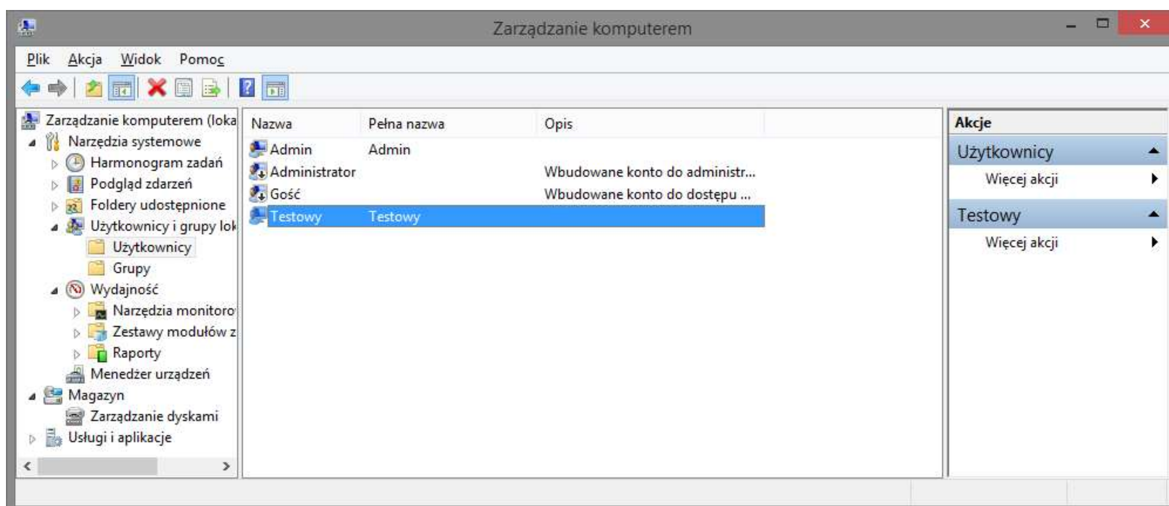
Dobrym sposobem, aby nie pobierać hasła jako jawny ciąg znaków jest zastosowanie polecenia `Read-Host` z parametrem `-AsSecureString` oraz przypisanie pobranego ciągu znaków, który posłuży później jako hasło:

```
Write-Host "Podaj hasło użytkownika: " -NoNewLine
$Hasło = Read-Host -AsSecureString
$Uzytkownik.SetPassword($Hasło)
```

Po ustawieniu hasła dla utworzonego użytkownika należy zastosować wszystkie wprowadzone zmiany poprzez użycie metody `SetInfo()`:

```
$Użytkownik.SetInfo()
```

Po tym kroku proces tworzenia użytkownika zostaje zakończony, jednak użytkownika nadal nie można znaleźć w rubryce *Konta użytkowników* w *Panelu sterowania*, ani tym bardziej zalogować się na jego konto. Znaleźć go za to można w panelu *Zarządzanie komputerem*. Okazuje się bowiem, że nowo utworzonego użytkownika należy jeszcze dodać do grupy użytkowników o nazwie *Użytkownicy*.



Rysunek 4.8. Nowo utworzony użytkownik widoczny w panelu *Zarządzanie komputerem*.

Aby to zrobić, zostanie wykorzystane podobne działanie jak w przypadku tworzenia użytkownika. Pierwszym krokiem będzie zapisanie grupy *Użytkownicy* do zmiennej, aby móc korzystać z jej metod:

```
$Grupa = [ADSI]"WinNT://$NazwaKomputera/Użytkownicy, group"
```

Po przypisaniu grupy do zmiennej można skorzystać z metody `Add` pozwalającej na dodanie użytkownika:

```
$Grupa.Add("WinNT://$NazwaKomputera/$($Użytkownik.Name)")
```

Po wykonaniu tego kroku, użytkownik zostaje dodany do grupy oraz staje się widzialny w *Panelu sterowania*. Można również zalogować się na utworzone konto wpisując hasło podane w procesie tworzenia użytkownika. Aby usunąć użytkownika z grupy należy skorzystać z metody `Remove` zmiennej przechowującej grupę *Użytkownicy*:

```
$Grupa.Remove("WinNT://$NazwaKomputera/$($Użytkownik.Name)")
```

Natomiast, aby usunąć całe konto użytkownika należy skorzystać z metody `Delete` należącej do obiektu klasy ADSI , w tym przypadku przechowywanego przez zmienną `$Komputer`:

```
$Komputer.Remove("User","Testowy")
```

Podobnie jak w przypadku metody służącej do tworzenia użytkownika, pierwszy parametr określa rodzaj usuwanego obiektu, natomiast drugi to nazwa użytkownika.

PowerShell wykorzystując *Active Directory* pozwala na operacje tworzenia i modyfikowania użytkowników, tworzenie nowych grup, oraz dokładną konfigurację każdego z kont użytkowników poprzez nakładanie różnego rodzaju ograniczeń oraz przydzielanie uprawnień. Szersze wytłumaczenie możliwości jakie oferuje *Active Directory Services Interface* można odnaleźć na oficjalnej stronie Microsoftu [24]:

https://msdn.microsoft.com/en-us/library/aa746512(VS.85).aspx

Dodatek A. Tabela poleceń użytych w tej pracy

Tabela 21. Polecenia użyte w tej pracy oraz ich najważniejsze parametry.

Polecenie:	Opis:	Parametr:	Wyjaśnienie:
New-Item	Tworzy nowy obiekt	-ItemType	Typ obiektu
		-Path	Ścieżka obiektu
		-Value	Wartość obiektu
Copy-Item	Kopiuje obiekty	-Path	Ścieżka kopiowanego obiektu
		-Destination	Ścieżka docelowa obiektu
Move-Item	Przenosi obiekty	-Path	Ścieżka przenoszonego obiektu
		-Destination	Ścieżka docelowa obiektu
Remove-Item	Usuwa obiekty	-Path	Ścieżka usuwanego obiektu
		-Force	Wymusza usunięcie obiektu
Rename-Item	Zmienia nazwę obiektu	-Path	Określa ścieżkę obiektu
		-Force	Wymusza zmianę nazwy
Set-Item	Zmienia właściwości obiektu	<właściwość obiektu>	<brak>
New-Object	Tworzy nowy obiekt zaawansowanego typu	<typ obiektu>	<brak>
Get-Content	Pobiera zawartość pliku i wyświetla w konsoli	-Path	Określa ścieżkę obiektu
Clear-Host	Czyści konsolę	<brak>	<brak>
Get-Location	Przechodzi do wybranej lokacji	<lokacja>	<brak>
Get-Process	Wyświetla listę aktywnych procesów	-Name	Filtr nazwy procesu
		-Id	Filtruje procesy po numerze ID
Stop-Process	Zatrzymuje wybrany proces	-Name	Filtr nazwy procesu
		-Id	Filtruje procesy po numerze ID
Write-Host	Wyświetla tekst w konsoli	-NoNewLine	Kursor nie przechodzi do nowej linii w przypadku kolejnego polecenia Write-Host
Get-ChildItem	Wyświetla zawartość bieżącego katalogu	-Filter	Filtr nazwy
		-Path	Ścieżka w której zostanie wywołane polecenie
		-Recurse	Wyświetlanie z uwzględnieniem podfolderów
		-Force	Wyświetla pliki ukryte

Get-Help	Wywołuje okno pomocy na temat określonego polecenia	-Detailed	Wyświetla zaawansowaną wersję pomocy
		-Full	Wyświetla pełną wersję pomocy
		-ShowWindow	Wyświetla pełną wersję pomocy w nowym oknie
		-Examples	Wyświetla przykłady użycia polecenia
Update-Help	Aktualizuje pomoc PowerShella	-SourcePath	Ścieżka do pliku z nowszą wersją pomocy
Save-Help	Zapisuje pobrane pliki pomocy w innej lokalizacji	-DestinationPath	Ścieżka docelowa, do której zostaną zapisane pliki pomocy
Get-Command	Wyświetla polecenia zgodne ze schematem	<polecenie>	Wyświetla wszystkie polecenie o zadanej budowie
Get-ExecutionPolicy	Wyświetla aktualny zestaw reguł bezpieczeństwa	-List	Wyświetla zakresy i przypisane im zestawy reguł bezpieczeństwa
Set-ExecutionPolicy	Ustawia określony zestaw reguł bezpieczeństwa	<brak>	
Get-Alias	Wyświetla polecenie przypisane do aliasu	<alias>	Wyświetla polecenie przypisane do aliasu
New-Alias	Tworzy nowy alias	-Value	Polecenie przypisywane do aliasu
		-Description	Opis aliasu
Set-Alias	Tworzy nowy alias, jeśli istnieje, zostanie nadpisany	-Value	Polecenie przypisywane do aliasu
		-Description	Opis aliasu
Export-Alias	Eksportuje zestaw aliasów do pliku	-as	Parametr służący do zapisania pliku z aliasami jako skrypt (-as Script)
Read-Host	Odczytuje znaki z konsoli	-AsSecureString	Zamienia jawny ciąg znaków na gwiazdki, używane np.: do wprowadzania haseł
		-Prompt	Wyświetla zapytanie poprzedzające pobranie ciągu znaków
New-Variabale	Tworzy nową zmienną	-Name	Określa nazwę zmiennej
		-Value	Określa wartość zmiennej
		-Option	Określa dodatkowe parametry zmiennej
Clear-Variable	Ustawia wartość zmiennej na $null	-Name	Czyści wartość zmiennej o zadanej nazwie
Remove-Variable	Usuwa zmienną	-Name	Usuwa zmienną o zadanej nazwie
Write-Error	Wypisuje komunikat na strumień błędów	-Message	Określa treść komunikatu
Write-Warning	Wypisuje komunikat na strumień ostrzeżeń	-Message	Określa treść komunikatu

Write-Verbose	Wypisuje komunikat na strumień komunikatów informacyjnych	-Message	Określa treść komunikatu
Write-Debug	Wypisuje komunikat na strumień debugowania	-Message	Określa treść komunikatu
Format-List	<brak>	<brak>	Polecenie określające format wyświetlenia obiektów przesłane potokowo
Where-Object	<brak>	<brak>	Polecenie wykonujące akcje z wybranymi obiektami w potoku
Sort-Object	<brak>	<brak>	Polecenie sortujące obiekty w potoku
Out-GridView	<brak>	<brak>	Polecenie wyświetlające obiekty przesłane potokowe w interaktywnej tabeli w nowym oknie
Get-Module	Wyświetla załadowane moduły	-ListAviable	Pobiera listę możliwych do załadowania modułów
Import-Module	Ładuje określony moduł do bieżącej sesji konsoli	-Name	Określa pełną ścieżkę wraz z nazwą modułu
Remove-Module	Usuwa moduł z bieżącej sesji konsoli	-Name	Określa nazwę modułu, który ma zostać usunięty z bieżącej sesji
Export-ModuleMember	Eksportuje element modułu	-Function	Określa eksportowane w module funkcje
		-Alisas	Określa eksportowane w module aliasy
		-Variable	Określa eksportowane w module zmienne
		-Cmdlet	Określa eksportowane w module polecenia typu cmdlet
Start-Job	Rozpoczyna zadanie wykonywane w tle	-Name	Określa nazwę zadania
		-Arg	Określa listę argumentów przekazanych do zadania
		-Id	Określa numer Id zadania
Get-Job	Pobiera listę zadań działających w tle	-Name	Określa nazwę zadania
		-Id	Określa numer Id zadania
Receive-Job	Pobiera wyniki zakończonego zadania działającego w tle	-Name	Określa nazwę zadania
		-Id	Określa numer Id zadania
Stop-Job	Zatrzymuje zadanie wykonywane w tle	-Name	Określa nazwę zadania
		-Id	Określa numer Id zadania

Wait-Job	<brak>	<brak>	Blokuje konsolę do czasu wykonania polecenia działającego w tle
Suspend-Job	Zatrzymuje zadanie wykonywane w tle z możliwością wznowienia	-Name	Określa nazwę zadania
		-Id	Określa numer Id zadania
Resume-Job	Wznawia zatrzymane wcześniej zadanie	-Name	Określa nazwę zadania
		-Id	Określa numer Id zadania
Remove-Job	Usuwa zakończone zadania	-Name	Określa nazwę zadania
		-Id	Określa numer Id zadania
Get-Member	Wyświetla właściwości i metody obiektów	<brak>	<brak>
Get-Random	Losuje wartość losową	-Min	Wartość minimalna
		-Max	Wartość maksymalna
Enter-PSSession	Uruchamia sesję zdalną PowerShella	-Computername	Określa adres IP lub nazwę komputera, z którym nastąpi połączenie
		-Credential	Określa konto użytkownika na które nastąpi logowanie zdalne
Invoke-Command	Wywołanie polecenia	-Session	Określa połączenie, w które zostanie nawiązane w celu wywołania komendy
		-Scriptblock	Określa blok wykonywanych poleceń
Start-Sleep	Zamraża pracę konsoli na określony okres czasu	-Miliseconds	Określa podawaną wartość w milisekundach
		-Seconds	Określa podawaną wartość w sekundach

Dodatek B. Tabela najważniejszych formatów wartości z wykorzystaniem klasy .NET String.FormatMethod

Tabela XX. Tabela najważniejszych formatów wartości z wykorzystaniem klasy .NET

Format	Opis	Przykład
D	Zapis długiego formatu daty	„{0:D}" -f [DateTime] „01/02/2015" 2 stycznia 2015
d	Zapis krótkiego formatu daty	„{0:d}" -f [DateTime] „01/02/2015" 2015-01-02
F lub f	Zapis określonej ilości miejsc po przecinku z zaokrągleniem	„{0:f3}" -f 1234.5678 1234.568
P lub p	Zapis procentowej wartości z określoną dokładnością po przecinku dla liczby pomiędzy 0 a 1	„{0:P2}" -f 0.28 28.00%
N lub n	Zapis liczby z określeniem ilości cyfr pomiędzy separatorami	„{0:N2}" -f 123456789 123 456 789,00