Windows PowerShell

Cosa dovreste aspettarvi da un'opera così breve - Prefazione

Obiettivi di questo libro

Questo libro contiene una presentazione di Windows PowerShell ed alcuni esempi pratici di utilizzo di questa tecnologia, in maniera tale da introdurre velocemente il lettore anche se quest'ultimo non dovesse avere esperienze di scripting pregresse. Questo libro non è esplicitamente destinato a chi è già un professionista dei linguaggi di scripting; la guida di Windows PowerShell e la vastità di informazioni recuperabili tramite Internet dovrebbero già fornire agli esperti qualsiasi informazione di cui dovessero avere bisogno. I principianti, invece, troveranno in questo testo tutte le informazioni dettagliate per iniziare a sviluppare script e magari imparare velocemente a gestire un computer anche senza mouse.

I testi di partenza da cui questo libro è stato tratto sono principalmente le pubblicazioni Microsoft correlate a Windows PowerShell. Questo libro presenta le informazioni in modo nuovo, appropriato per il target di utenza a cui è stato destinato, tralasciando inizialmente la teoria e prediligendo la quantità di esempi e mini esercizi pratici, così da catturare l'interesse del lettore.

Quando avrete terminato di studiare questo libro e deciso di rendere Windows PowerShell parte integrante della vostra attività IT quotidiana, potrete leggere la documentazione originale di Windows PowerShell, inclusa automaticamente con l'installazione di Windows PowerShell:

- First Steps with Windows PowerShell
- Basic Principles of Windows PowerShell

Per trarre realmente beneficio da questo libro dovreste poter accedere ad un PC dove effettuare gli esercizi proposti: l'unico prerequisito è un PC con installato Windows PowerShell 1.0.
Per maggiori informazioni sul download e la procedura di installazione di Windows PowerShell potrete utilizzare i siti menzionati nel paragrafo successivo.

Risorse aggiuntive online

- Il sito ufficiale di Windows PowerShell, che contiene un'introduzione alla tecnologia e le informazioni per effettuare il download del prodotto, è raggiungibile a questo indirizzo:

  http://www.microsoft.com/PowerShell

  In questo sito potrete trovare anche link ad altro utile materiale, come webcast, libri e forum online.

- Il blog ufficiale del team di sviluppo di Windows PowerShell si trova a questo indirizzo:

  http://blogs.msdn.com/PowerShell/

Potrete reperire informazioni sulle tecniche di scripting e dimostrazioni di utilizzo pratiche.

- Se parlate tedesco, potrete consultare il blog del team ITPro di Microsoft Switzerland, dove troverete webcast e download correlati agli esercizi di questo libro (archivio Marzo/Aprile 2007):

  http://blogs.technet.com/chITPro-DE

- Il sito della community italiana di Windows PowerShell, con la guida in linea di oltre 1000 cmdlet di differenti produttori, una raccolta di articoli e snippet in italiano, un forum gestito da esperti del settore e molto altro ancora è raggiungibile a questo indirizzo:

  http://www.powershell.it

Combinazioni utili di tasti (per le tastiere italiane)

Simbolo	Combinazione	Significato
|	Shift + \	Inoltra l'output di un comando
`	Alt+96	Continua il comando nella riga successiva
{	Ctrl+Alt+Shift+[Alt+123	Inizia una sequenza di comandi (es. dopo una condizione if)
}	Ctrl+Alt+Shift+] Alt+125	Termina una sequenza di comandi
[	AltGr+[	Richiesto a volte per manipolare gli oggetti
]	AltGr +]	Richiesto a volte per manipolare gli oggetti
TAB	Tab	Completa il comando quando necessario Esempio: **GET-HE** (TAB) produce **GET-HELP**

Windows PowerShell è un prodotto sviluppato a Redmond e la sintassi del linguaggio risulta essere ottimale per il layout delle tastiere americane. Alcuni caratteri sono difficili da trovare nelle tastiere italiane, perciò potrebbe essere utile avere a portata di mano una lista di combinazioni equivalenti.

Sommario

WINDOWS POWERSHELL IN PRATICA

Windows PowerShell è un add-on gratuito per sistemi operativi Windows versione XP e superiori, scaricabile dal sito Microsoft all'indirizzo http://www.microsoft.com/powershell. Unico prerequisito è il framework .NET versione 2.0 che – se non installato – richiede un download ed una installazione a parte. Il pacchetto di installazione di Windows PowerShell è di dimensioni relativamente ridotte (circa 1.5 MB) e può essere facilmente installato anche tramite distribuzione automatizzata. L'unico aspetto da considerare è il fatto che esistono differenti versioni di Windows PowerShell, in base sia alla versione di Windows che all'architettura utilizzata (x86, x64, ia64). Terminata l'installazione, Windows PowerShell disporrà di un'icona nel menu Start e sarà accessibile cliccando sul collegamento o digitando "powershell" nel box Esegui di Windows (Windows+R).

Prime impressioni su Windows PowerShell

Per farvi una prima impressione sul prodotto, lanciate sia Windows PowerShell sia CMD - il classico inteprete dei comandi di Windows – dal menu Start. Di primo acchito entrambe le shell appariranno molto simili, se non per i colori distintivi di ciascuna:

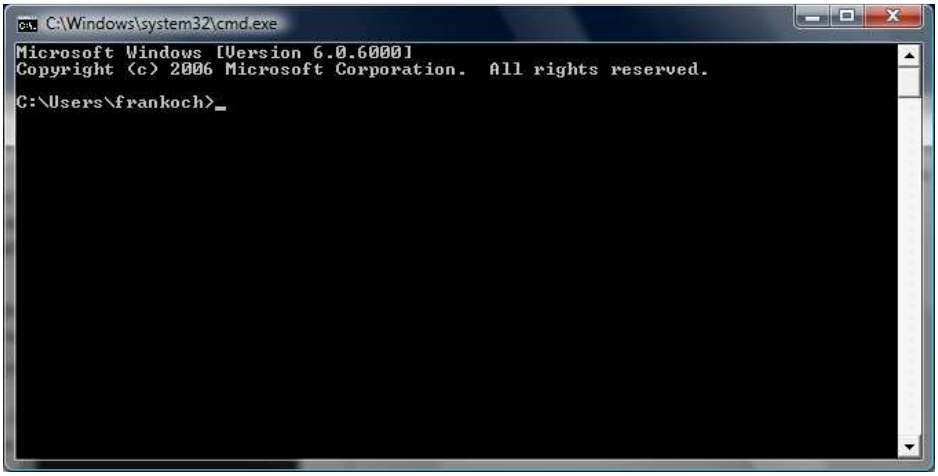

FIGURA 1: CMD, IL CLASSICO INTERPRETE DEI COMANDI WINDOWS

FIGURA 2: WINDOWS POWERSHELL

Questa somiglianza non vi deve sorprendere, perchè entrambe le shell utilizzano il medesimo "contenitore di input". Sfortunatamente, questo significa che anche Windows PowerShell soffre della stessa mancanza di supporto per le operazioni di Copia/Incolla di cui l'interprete dei comandi CMD ha sofferto per anni. Ecco però come superare questo scoglio grazie al supporto di un mouse:

- Selezionate il testo desiderato con il mouse
- Premete il pulsante destro (copia)
- Posizionate il cursore nella posizione desiderata
- Premete il pulsante sinistro (incolla)

Provate voi stessi. Copiate la prima linea di testo visualizzata nella schermata della shell (es. "Windows PowerShell", oppure "Copyright (c) 2006 Microsoft Corporation") e incollatela come fosse una *riga di comando*. Non preoccupatevi dei messaggi di errore che appariranno quando premerete Invio. Fate pratica con questi esercizi: utilizzerete spesso questa tecnica da qui in avanti.

Anche se il "contenitore" è il medesimo, i contenuti e le funzionalità delle shell sono molto differenti. La modalità più semplice per scoprire quali siano queste differenze è quella di dare un'occhiata, ovviamente, alla guida in linea. Appare subito evidente che Windows PowerShell offre molte più funzionalità di CMD; più di 100 comandi, chiamati anche cmdlet (pronunciato "*Commandlet*"). Poichè solo alcuni di comandi erano direttamente contenuti all'interno di CMD (DIR, TYPE, CD, etc), diversi software aggiuntivi di supporto sono stati sviluppati nel corso degli anni (XCOPY, ATTRIB, TELNET, etc). E poichè ciascuno di questi software per CMD utilizza una propria sintassi indipendente dalle altre, un utilizzatore esperto di CMD avrebbe dovuto imparare comandi differenti e logiche di funzionamento e di passaggio di parametri differenti. Per i cmdlet, la sintassi e la logica sono definite chiaramente e da tutti condivise. I comandi di Windows PowerShell seguono delle specifiche regole di nomenclatura:

- I comandi di Windows PowerShell consistono in un verbo ed un nome (sempre singolare) separati da un trattino (-). I comandi sono in lingua inglese. Esempio: *Get-Help* richiama l'help in linea di Windows PowerShell.
- I parametri sono indicate dal prefisso –:
  Get-Help -Detailed
- La shell include poi molti dei comandi più utilizzati nelle altre shell presenti nel mercato, così da rendere più facile agli utenti di queste l'adozione di Windows PowerShell. Esempio:
  Sia G*et-Help*, *help* (da CMD) e *man* (classico delle shell *nix) funzionano allo stesso modo.

Visualizzate il testo di help per ciascuna shell. Digitate *help* in ciascuna shell: potrete osservare il diverso numero di comandi documentati tra CMD e Windows PowerShell.

Al posto di *help* o *man* potete anche utilizzare *Get-Help* all'interno di Windows PowerShell. La sintassi è la seguente:

- *Get-Help* ritorna informazioni su come utilizzare la guida in linea
- *Get-Help ** elenca tutti i comandi disponibili all'interno di Windows PowerShell
- *Get-Help comando* visualizza la guida in linea per il comando indicato

- *Get-Help comando –Detailed* visualizza la guida in linea dettagliata per il comando indicato, comprensiva di esempi

Utilizza il comando Get-Help per recuperare informazioni dettagliate sul comando Get-Help: *Get-Help Get-Help –Detailed*. Consiglio: utilizzate il tasto TAB per completare automaticamente i comandi. Vi permetterà di evitare errori di battitura.

Per ulteriori informazioni sul comando **Get-Help** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Get-Help.aspx

Utilizzi aggiuntivi dell'output: il "piping" degli oggetti

Come menzionato in precedenza, Windows PowerShell è una shell orientata agli oggetti. Ciò significa che l'input e l'output dei comandi sono solitamente oggetti. Poiché gli essere umani non possono "leggere" gli oggetti, Windows PowerShell 'traduce' gli oggetti dell'output in testo, visualizzato successivamente sullo schermo (tramite alcuni comandi è addirittura possibile manipolare l'output di Windows PowerShell affinchè questa "traduzione" avvenga in maniera differente). I comandi collegati tra loro sono rappresentati dal comando 'pipe': |

Potete utilizzare questo collegamento tra comandi per creare il vostro libro personale su Windows PowerShell: *Get-Help * | Get-Help –Detailed* lo farà per voi: *Get-Help ** crea una lista di comandi conosciuti che utilizzerà come input del comando *Get-Help –Detailed*. L'output è decisamente corposo; annullare l'operazione utilizzando la combinazione di tasti CTRL+C.

Per essere in grado di utilizzare il risultato del libro così costruito in futuro, conviene memorizzarne l'output in un file anzichè visualizzarlo a video. Windows PowerShell dispone del comando *Out-File*, meglio conosciuto con il simbolo >.

Create il vostro libro su file ora; digitate questo comando: *Get-Help * | Get-Help –Detailed | Out-File C:\Powershell-Help.txt* oppure
*Get-Help * | Get-Help –Detailed > C:\PowerShell-Help.txt.*
Dovete necessariamente disporre delle autorizzazioni di scrittura relative al percorso in questione (nell'esempio: C:\ ).

Potete aprire il file così creato con Notepad e utilizzarlo come guida per gli esercizi futuri.

Per ulteriori informazioni sul comando **Out-File** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Out-File.aspx

Se cercate informazioni su di un comando, allora *Get-Help* può fare al caso vostro. Se desiderate ordinare un elenco di oggetti, provate a cercare un comando adeguato digitando *Get-Help Sort**. *Get-Help* cercherà così tra i comandi di Windows PowerShell e, poichè tutti i comandi iniziano con un

verbo, sarà possibile strutturare le proprie ricerche facilmente, utilizzando *Get-Help "verbo inglese"**. Per chi non lo sapesse, il simbolo * rappresenta una ricerca wildcard ed indica che dopo il testo specificato può apparire qualsiasi altro testo, di cui al momento della ricerca non siamo a conoscenza o che comunque non ci interessa.

Dopo aver trovato un comando che vi interessa (in questo caso potrebbe essere *Sort-Object*), potete semplicemente richiamare *Get-Help* di nuovo, ma questa volta fornendo il comando di interesse ed il parametro *–Detailed* così da ottenere anche degli esempi di utilizzo:

Get-Help Sort-Object –Detailed

Per ulteriori informazioni sul comando **Sort-Object** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Sort-Object.aspx

Primi esercizi con gli oggetti di Windows PowerShell

Se non avete mai lavorato con gli oggetti prima d'ora, gli esercizi che seguono vi permetteranno di capire la miriade di opportunità offerte da questa tecnologia. Se siete interessati a lavorare con gli oggetti potrete reperirne una documentazione dettagliata presso il sito Microsoft MSDN, all'indirizzo http://msdn.microsoft.com.
Iniziamo a studiare gli oggetti utilizzando l'oggetto "processo" come esempio. Se il termine "processo" non vi dice niente provate a pensare a cosa vedete a video quando richiamate il Task Manager di Windows. Se siete interessati ad approfondire le funzionalità dell'oggetto processo potrete trovare utile la documentazione MSDN menzionata in precedenza.

Lavorare con i processi

Il comando *Get-Process* elenca tutti i processi del vostro sistema. La lista può essere molto lunga. Per ordinare la lista utilizzate un altro cmdlet: *Sort-Object*. La modalità predefinita di ordinamento di *Sort-Object* è ascendente ma è possibile cambiarne il valore tramite il parametro *-Descending*. L'argomento di questo cmdlet sono le proprietà (una o più) dell'oggetto da utilizzare per ordinare il risultato (es. CPU).

A1: Provate ora a generare la lista di tutti i processi ed ad ordinarla inversamente secondo la loro percentuale di utilizzo del processore (proprietà CPU). Avete già imparato tutto ciò di cui avete bisogno per farlo: *Get-Process, Sort-Object* ed il pipe |. Consiglio: CPU non è un parametro di *Sort-Object* ma è un argomento che potete utilizzare per l'ordinamento. Non ha, perciò, il prefisso –.

Per ulteriori informazioni sul comando **Get-Process** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Get-Process.aspx

Nel prossimo esercizio vogliamo fare in modo di limitare un po' la lista per renderla più facile da gestire. Utilizzeremo il comando *Select-Object*. *Select-Object* può accettare diversi parametri (utilizzate *Get-Help* per trovarli), ma in questo caso ci servono solo *–First x* e *–Last y* per recuperare i primi *x* o gli ultimi *y* oggetti in una lista. Es. *Select-Object –First 5.*
Select-Object da solo non serve a nulla ma, anzi, attende che gli venga fornito dell'input tramite pipe.

A2: Generate una lista dei primi dieci processi, ordinati in base alla loro percentuale di utilizzo del processore. Per fare ciò, utilizzate il risultato ottenuto dall'esercizio A1 e aggiungeteci il comando *Select-Object*. Ci sono due modi per raggiungere una soluzione ottimale, a seconda di come la lista viene ordinata. Provate a trovarli entrambi.
Suggerimento: un'opzione utilizza il parametro *–First*, l'altra il parametro *–Last*.

Per ulteriori informazioni sul comando **Select-Object** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Select-Object.aspx

Utilizzeremo il prossimo esercizio come veloce introduzione alle variabili. In parole semplici, le variabili possono memorizzare qualsiasi valore, nonostante tali valori possano anche essere oggetti veri e propri! Se vorrete approfondire questi aspetti meno pratici è conveniente che vi rifacciate alla documentazione indicate all'inizio di questo libro; per ora vi basti sapere che le variabili in PowerShell devono sempre iniziare con il carattere $. Potete memorizzare il risultato dell'esercizio A2 in una variabile, consentendo l'accesso alla lista dei 10 processi in seguito ed in qualsiasi momento, e permettendo eventualmente di confrontarne tali valori con altri recuperati nel corso del tempo, valutando i cambiamenti del sistema. Impostare il valore di una variabile è semplice:

$a = Get-Process | Sort-Object CPU -Desc…

A3: Assegna alla variabile $P la lista dell'esercizio A2. Consiglio: utilizzate il tasto "Freccia su" per richiamare l'ultimo comando utilizzato e il tasto "Home" per spostare il cursore all'inizio della riga; poi inserite i comandi aggiuntivi. Potete visualizzare il contenuto della variabile a video semplicemente digitando *$P* al prompt.

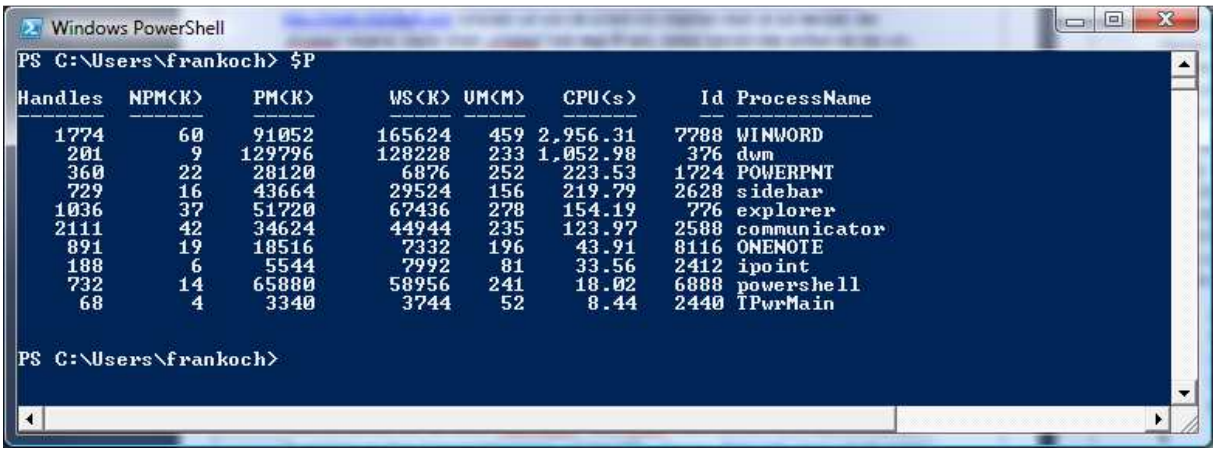

FIGURA 3: OUTPUT DELLA VARIABILE $P

Output in un file TXT, CSV o XML

Di default, Windows PowerShell visualizza il risultato di un blocco di comandi a video. Ogni oggetto è convertito in testo in maniera tale da poterne leggere il contenuto: internamente viene utilizzato il comando *Out-Host*, che Windows PowerShell aggiunge per voi in maniera invisibile ed automatica se il vostro comando non specifica una modalità di output. Ci sono alternative ad *Out-Host*: le troverete facilmente utilizzando il comando *Get-Help Out**.

Per ulteriori informazioni sul comando **Out-Host** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Out-Host.aspx

Riversare il risultato di un blocco di comandi su di un file di testo è semplice: la soluzione è il comando *Out-File filename*. Molte altre shell utilizzano il comando >, supportato anche da Windows

PowerShell. Oltre che riversare l'output su di un file di testo è anche possibile utilizzare nativamente file di tipo CSV o XML, utilizzando rispettivamente i cmdlet *Export-CSV* ed *Export-CLIXML*; entrambi richiedono il nome del file di destinazione come argomento. Eh sì, avete ragione: se potete esportare dovete anche poter importare. Utilizzate *Import-CSV* o *Import-CLIXML* per effettuare il processo inverso a quello appena descritto.

A4: Prendete la variabile $P dall'esercizio A3 e memorizzatene il contenuto in un file con nome "A4.txt". Poi salvate il contenuto di $P in un file CSV chiamato "A4.CSV" e, per finire, in un file XML con nome "A4.XML". Suggerimento: il comando > sostituisce il pipe |, richiesto solo per gli "autentici" cmdlet come *Out-File*, *Export-CSV* etc. Se vorrete dare un'occhiata ai risultati il blocco note dovrebbe essere sufficiente.

Per ulteriori informazioni sui comandi sopra menzionati è possibile consultare la guida di riferimento:

Out-File: http://www.powershell.it/Comandi/v1.0/Out-File.aspx
Export-CSV: http://www.powershell.it/Comandi/v1.0/Export-CSV.aspx
Export-CLIXML: http://www.powershell.it/Comandi/v1.0/Export-CLIXML.aspx
Import-CSV: http://www.powershell.it/Comandi/v1.0/Import-CSV.aspx
Import-CLIXML: http://www.powershell.it/Comandi/v1.0/Import-CLIXML.aspx

Output colorato

A volte conviene evidenziare alcuni risultati per renderli più semplici da leggere. Potete farlo, per esempio, utilizzando un colore. Il comando *Write-Host* accetta diversi parametri come *–ForegroundColor* e *–BackgroundColor*. Ed ora provate ad indovinare, come sarà l'output di questo comando?

Write-Host "Testo rosso su fondo blu" –ForegroundColor red –BackgroundColor blue

Sì, avete proprio indovinato! *Get-Help Write-Host –Detailed* ritorna la lista di tutti i possibili colori. Ci sono anche delle combinazioni di colori predefinite: con *Write-Warning "errore"* potete anche attirare l'attenzione dell'utente. Ora proviamo insieme; con questo comando possiamo visualizzare tutti i processi in un particolare colore. Sarebbe più conveniente, però, se potessimo colorarli a seconda del verificarsi di una determinata condizione; vediamo più da vicino questo problema.

Per ulteriori informazioni sui comandi sopra menzionati è possibile consultare la guida di riferimento:

Write-Host: http://www.powershell.it/Comandi/v1.0/Write-Host.aspx
Write-Warning: http://www.powershell.it/Comandi/v1.0/Write-Warning.aspx

Per semplicità utilizziamo i servizi del vostro PC invece che i processi. Se non sapete cosa sia un servizio vi conviene approfondire – su MSDN, ad esempio – prima di proseguire con la lettura. È possibile vedere la lista dei servizi del proprio PC tramite il ***Pannello di controllo / Strumenti di***

amministrazione / Servizi. Ciascuno di essi appare con lo stato “avviato“ o “arrestato“, proprietà che si presta molto bene alla visualizzazione colorata. Ma prima vediamo come ottenere la lista dei servizi tramite il cmdlet *Get-Service*.

A5: Generate la lista di tutti i servizi ed ordinatela per stato (proprietà *Status*). Consiglio: utilizzate lo stesso metodo che avete utilizzato per ordinare i processi per percentuale di elaborazione ma utilizzate *Get-Service* e “*Status*“ come argomenti per *Sort-Object*.

Per ulteriori informazioni sul comando **Get-Service** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Get-Service.aspx

Ora vogliamo visualizzare l’intera lista in colore rosso e *Write-Host* farà al caso nostro. Sfortunatamente, però, il comando *Get-Service | Write-Host –Forgroundcolor Red* non funziona come sperato; *Write-Host* non è così amichevole come altri cmdlet e richiede una lista di oggetti da visualizzare con il colore specificato. *Write-Host* vuole sapere per ciascun oggetto quali attributi dovranno essere visualizzati.
Dobbiamo quindi aiutare *Write-Host* nel suo compito: prima di tutto scorriamo la lista di oggetti uno ad uno, creando un ciclo. Ci sono diversi tipi di cicli ed ognuno ha il proprio significato e la propria area di utilizzo. Per il nostro scopo utilizziamo il ciclo fornito dal cmdlet *ForEach-Object*, che scorre la lista di oggetti e ne passa uno alla volta al cmdlet successivo. All’interno del ciclo possiamo scegliere gli oggetti di interesse utilizzando la variabile automatica *$_*, una particolare caratteristica di PowerShell che permette di recuperare le variabili presenti in un determinato contesto. È possibile quindi selezionare un particolare attributo di un oggetto utilizzando la sintassi *$_.nome-della-proprietà*.
Vediamo un esempio:

Get-Process | ForEach-Object { Write-Host $_.ProcessName $_.CPU}

Dove *Get-Process* ritorna una lista di processi, di cui in questo esempio sono visibili solo il nome e la percentuale di tempo di elaborazione poichè l’output del primo cmdlet finisce nel pipe (|) e successivamente *Write-Host* visualizza solo quei due attributi per ogni oggetto recuperato. Se ripeterete l’esempio più volte non vi stupite dei risultati! Non tutti i processi hanno un tempo di elaborazione e in certe circostanze una linea di risultato può avere solo il nome del processo.

A6: Generate una lista di servizi, visualizzando di ciascuno solo il nome e lo stato. Utilizzate il ciclo ForEach descritto in precedenza, anche se vi vengono in mente soluzioni alternative.

Per ulteriori informazioni sul comando **ForEach-Object** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/ForEach-Object.aspx

Ora potete utilizzare le opzioni di *Write-Host* per visualizzare l’output colorato.

A7: Generate una lista di servizi e visualizzate di ciascuno solo il nome e lo stato in un colore a vostra scelta. Consiglio: prendete la soluzione A6 ed aggiungete i parametri –- *Foregroundcolor* e *–Backgroundcolor.*

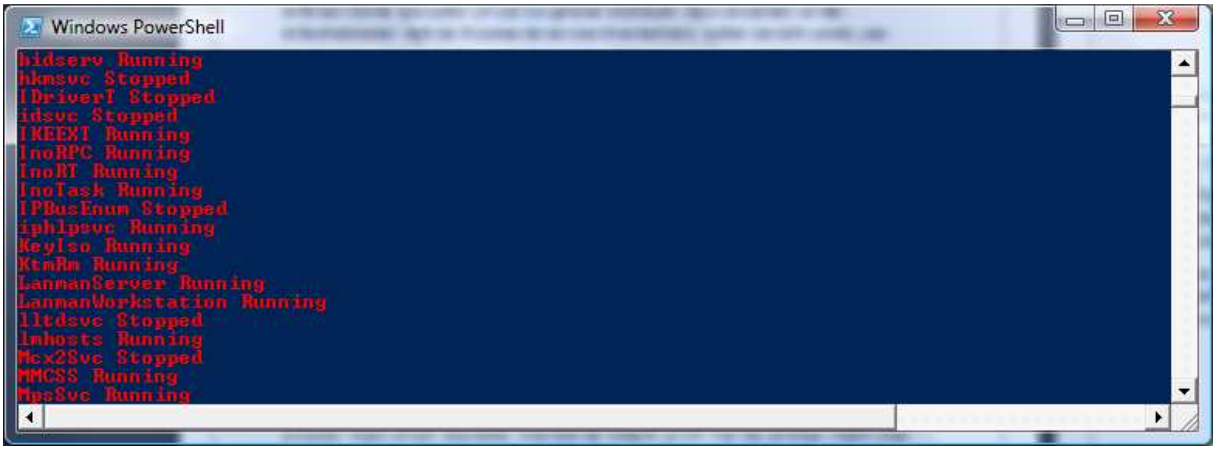

FIGURA 4: VISUALIZZA I NOMI E GLI STATI DEI SERVIZI IN ROSSO

Verificare le condizioni utilizzando il comando *if*

L'ultimo concetto non ancora analizzato consiste nel verificare le condizioni. Vi sono diverse soluzioni possibili per risolvere questo tipo di problema ma ci limiteremo, per ora, al comando *if*. Probabilmente ne conoscete già la sintassi perché è molto simile a quella di altri linguaggi di scripting e di programmazione; ad ogni modo è davvero molto semplice:

```
if (condizione) {comando/i da eseguire}
elseif (condizione2) {comando/i da eseguire}
else {comando/i da eseguire}
```

elseif è opzionale e non sempre necessario. Se volete eseguire più di un comando all'interno dell'area delimitata dalle parentesi graffe {}, potete separare ciascuno utilizzando il punto e virgola o un ritorno a capo. Windows PowerShell attende semplicemente per la parentesi graffa chiusa } alla fine.

Windows PowerShell accetta diversi operatori di confront all'interno delle condizioni: iniziano tutti con il trattino "–" e ciascuno consiste tipicamente in un'abbreviazione di due lettere di un termine inglese: ad esempio, -eq sta per 'equals' e verifica l'uguaglianza tra due termini. Gli operatori di confronto più importanti sono elencati di seguito:

-eq	**Equal** Uguaglianza tra due termini
-match	**Match** Match tra due termini, non così limitato come Equals
-ne	**Not equal** Disuguaglianza tra due termini

-notmatch	**Not match** Non match tra due termini
-gt –ge	**Greater than / Greater than or equal to** Maggiore di/Maggiore di o uguale a tra due termini
-lt –le	**Less than / Less than or equal to** Minore di/Minore di o uguale a tra due termini

Per ulteriori informazioni sui concetti sopra esposti è possibile consultare la guida di riferimento:

If/else/elseif: http://www.powershell.it/Snippet/Il-blocco-if.aspx
Operatori di confronto: http://www.powershell.it/Snippet/Come-utilizzare-gli-operatori-di-confronto.aspx

L'ultimo esercizio di questa sezione combina tutti i punti del problema iniziale del monitoraggio di un sistema utilizzando PowerShell. Tutti i servizi di un sistema sono dapprima ordinati in base al proprio status e successivamente visualizzati a video, ciascuno colorato in base al proprio stato: i servizi arrestati in rosso, quelli avviati in verde.

A8: Recuperate la lista di tutti i servizi. Ordinate la lista per stato, poi visualizzate a video il nome di ciascun servizio utilizzando il colore rosso o verde a seconda dello del stato del servizio, come descritto in precedenza. Suggerimento: utilizzate dapprima il cmdlet *Sort-Object* come negli esercizi precedenti. Poi utilizzate un ciclo *ForEach* ma, anzichè sfruttare direttamente *Write-Host*, aggiungete un blocco *if*. Potete recuperare lo stato di un servizio all'interno del ciclo utilizzando come di consueto *$_.Status*; i valori di interesse che questa proprietà può assumere sono "*Stopped*" (servizio arrestato) o "*Running"* (servizio avviato). Ricordate la sintassi: la condizione del blocco *if* è inserita all'interno di parentesi tonde () ed il comando da eseguire tra parentesi graffe {}. Ignorate le altre opzioni e, per semplicità, evitate il blocco opzionale *elseif*. Non dimenticate la parentesi graffa finale } per il cmdlet *ForEach*! Quando raggiungete la fine di una linea che inizia per >>, chiudete il blocco con un paio di ritorni a capo per eseguire lo script che avete creato.

Questa volta eseguite nuovamente lo script ma omettete il cmdlet *Sort-Object.* Utilizzate le frecce cursore in maniera tale da non dover reinserire tutti i comandi di nuovo.

ForEach-Object può essere abbreviato in *ForEach*. Potreste renderlo ancora più breve ma dobreste ricordarvi a memoria il significato di ciascuna sigla ogni volta. E così non sarebbe più tanto semplice. Quindi, per ora continuiamo ad utilizzare *ForEach-Object* o *ForEach.*

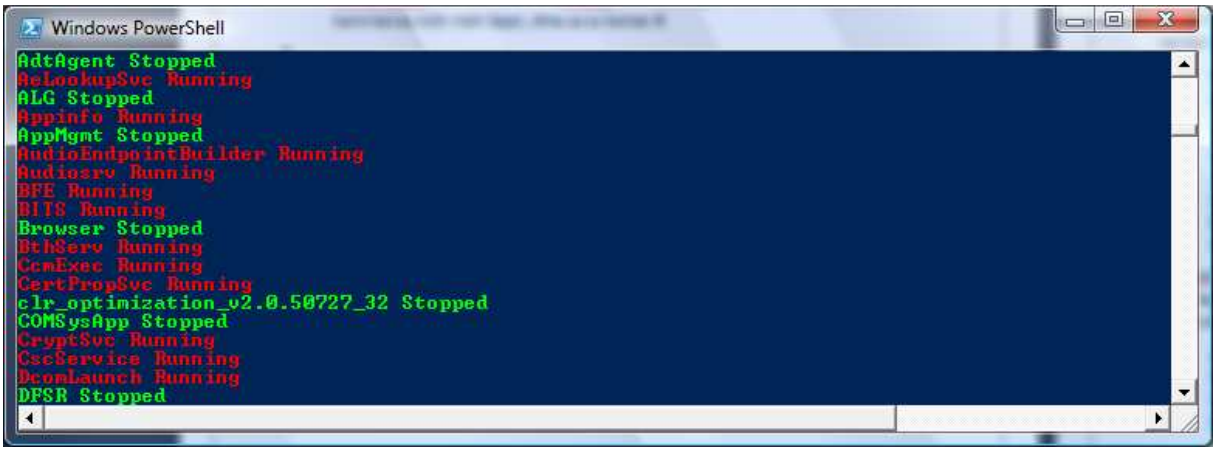

FIGURA 5: OUTPUT A COLORI DEI SERVIZI

Output in HTML

L'esempio A8 potrebbe essere utilizzato per monitorare un server e sarebbe magnifico se fosse più semplice leggerne l'output. Sapete già come memorizzare l'output in formato CSV e XML; allora vi presento un altro formato di output che potrebbe a volte tornarvi ancora più utile: HTML. E il cmdlet *ConvertTo-HTML* è utilizzato per trasformare il vostro output in questo formato. L'output non consiste più in un file ma, come con la maggior parte degli altri cmdlet, è in una forma che è possibile modificare direttamente utilizzando un pipe. Alla fine potete comunque memorizzare il testo prodotto da questo cmdlet in un file, in maniera da consentirne, ad esempio, la visualizzazione utilizzando un browser web.

Per ulteriori informazioni sul comando **ConvertTo-HTML** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/ConvertTo-HTML.aspx

Ora useremo una serie di mini-esempi per esplorare le varie opzioni rese disponibili dal cmdlet *ConvertTo-HTML*.

A9: Convertite l'output prodotto da *Get-Service* in formato HTML. Utilizzate il cmdlet *ConvertTo-HTML*, che può lavorare direttamente con una lista di oggetti. Suggerimento: se la lista impiega troppo tempo ad essere generata annullate l'elaborazione con CTRL+C.

A10: Infine utilizzate i comandi che conoscete per memorizzare l'output dell'esercizio precedente in un file di nome ".\A10.html" e date un'occhiata al contenuto prodotto. Suggerimento: Potete usare *Invoke-Item .\a10.html* per lanciare il vostro browser web predefinito ed aprire il file direttamente da PowerShell. Non dimenticate di indicare il percorso corretto di A10.html! Se preferite, aprite il file anche da Esplora risorse.

Per ulteriori informazioni sul comando **Invoke-Item** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Invoke-Item.aspx

ConvertTo-HTML consente di limitare l'output così da renderlo più leggibile. Usate *ConvertTo-HTML* ed indicate con il parametro *–Property* la lista delle proprietà da includere nell'output HTML, es. … | *ConvertTo-HTML –property name, status*.

A11: Continua dall'esercizio A10: Generate una pagina web più concisa e riportate solo il nome e lo stato di ciascun servizio. Potete anche ordinare l'output per stato **prima** della conversione. Suggerimento: la vostra linea di comando utilizzerà quattro comandi distinti: lista di tutti i servizi, ordina per stato, conversione in HTML, memorizzazione su file.

Poichè *ConvertTo-HTML* crea del testo HTML, l'output può essere modificato facilmente se si ha un po' di esperienza con questo formato. Non è un compito tipicamente svolto utilizzando Windows PowerShell, ma la shell lo consente. Provate ad indovinare quale sarà il risultato dello script che segue prima di copiarlo ed eseguirlo in una finestra di Windows PowerShell:

```
Get-Service | ConvertTo-HTML -Property Name,Status | foreach {

if ($_ -like "*<td>Running</td>*") {$_ -replace "<tr>", "<tr bgcolor='green'>"}

else {$_ -replace "<tr>", "<tr bgcolor='red'>"}} > .\Get-Service.html
```

L'output dovrebbe essere simile a quello visualizzato nella figura che segue. L'esempio funziona con lo stesso principio dell'output del cmdlet *Write-Host* ma, in questo caso, le singole linee del file HTML sono rielaborate: viene impostato il markup HTML per le righe della tabella in maniera tale che abbiano uno sfondo di colore verde (*bgcolor green*) o rosso (*bgcolor red*). Poiché non tutti conoscono HTML, questo esempio è fornito, al contrario degli altri, con la soluzione completa.

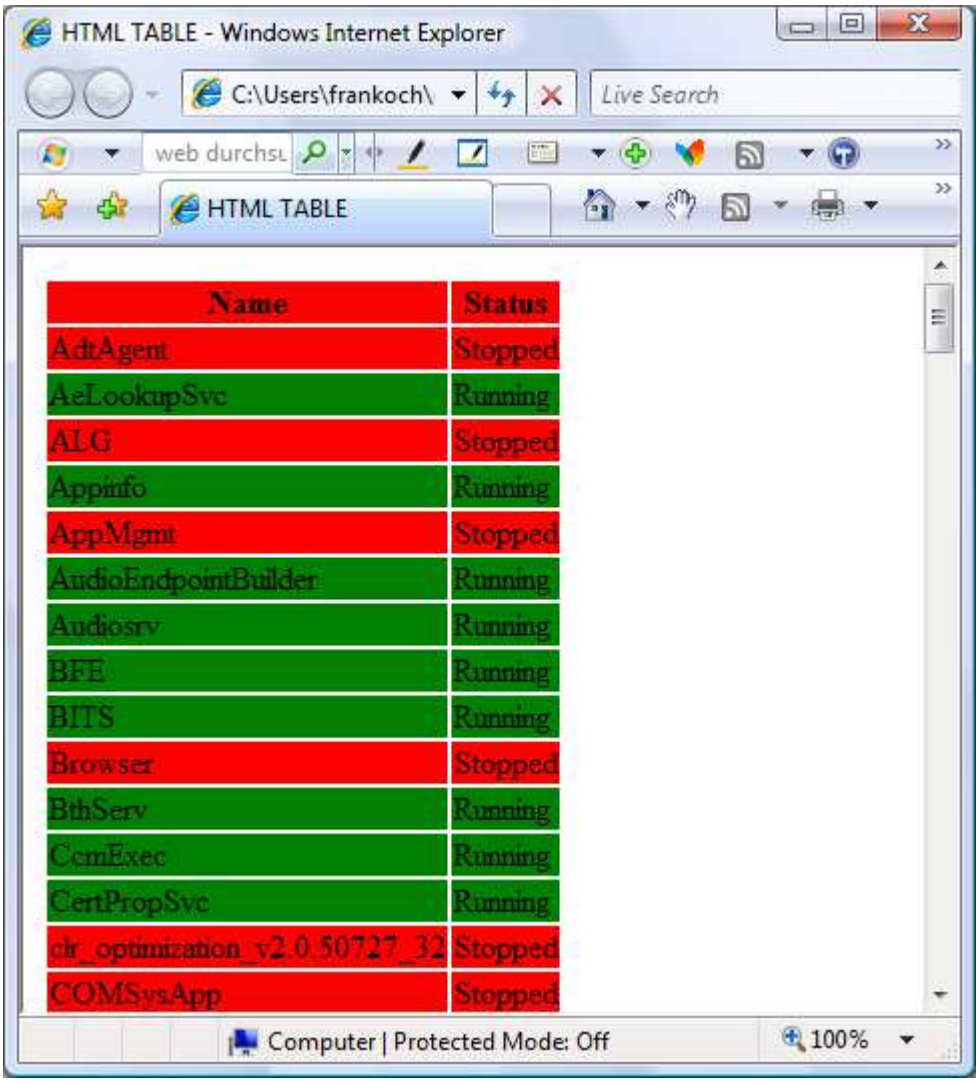

FIGURA 6: COLORAZIONE DELL'OUTPUT HTML UTILIZZANDO LA MANIPOLAZIONE DELL'HTML

Lavorare con i file

Negli esercizi che seguono lavoreremo con i file. Se questo libro vi è stato consegnato come materiale di un workshop, chiedete per piacere al vostro istruttore di consegnarvi una chiavetta di memoria con dei file di test. Se state effettuando gli esercizi per conto vostro, create una cartella destinata agli esercizi e copiaci dentro un po' di file differenti (40-50). Se non trovate niente di meglio potete copiare i file della cache del vostro browser web. Assicuratevi, per piacere, che vi siano almeno due tipi differenti di file ma più ce ne sono meglio è.

Lavorare con i file in Windows PowerShell è davvero intuitivo. Comandi popolari come *dir* o *ls* possono essere utilizzati direttamente. Per *cd*, ad ogni modo, dovete considerare lo spazio obbligatorio che separa il comando dall'argomento: ad esempio, anziché cd.. dovete utilizzare cd .. !

Windows PowerShell considera tutti i file come fossero oggetti. La dimensione di un file può essere recuperata direttamente e non deve essere estrapolata da una stringa. Windows PowerShell, inoltre, non lavora con il file system 'classico'; utilizzando il cmdlet *Get-PSDrive* visualizzate tutti i percorsi a cui Windows PowerShell consente di accedere direttamente. I drive sono individuabili dal fatto che terminano con il carattere due punti (:).

Per ulteriori informazioni sul comando **Get-PSDrive** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Get-PSDrive.aspx

Visualizzate tutti i drive di Windows PowerShell. Passate (*cd*) al drive HKLM: e digitate *cd software*. Digitate *dir*. Dove siete ora? Passate al drive ENV: e recuperatene il contenuto con *ls*, come se fosse una cartella utilizzata da una shell Unix. Infine, passate al drive CERT: e recuperatene il contenuto utilizzando il cmdlet *Get-ChildItem*.

Vedrete che in Windows PowerShell quasi tutte le informazioni disponibili nel vostro computer saranno facilmente disponibili. I comandi *dir, ls* e *Get-ChildItem* hanno la stessa funzionalità, ovunque. Significa che potete utilizzare quello che preferite.

Per ulteriori informazioni sul comando **Get-ChildItem** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Get-ChildItem.aspx

Aggiungo solo una piccola nota in merito all'utilizzo del registro di Windows (registry); se avete effettuato qualche test con il recupero delle informazioni dal registry probabilmente vi sarete accorti che i comandi *dir* e *cd* funzionano sì come vi sareste aspettati ma non riuscite a visualizzare gli effettivi valori delle chiavi. In effetti ciò accade perchè i valori delle chiavi del registry sono essi stessi proprietà degli oggetti del registry, proprio come *Size* e *Date last accessed* sono attributi di un file. Per recuperare i valori delle chiavi del registry dovete utilizzare il cmdlet *Get-ItemProperty*. Questo comando consente anche, infatti, di recuperare i valori delle chiavi di registro.

Per ulteriori informazioni sul comando **Get-ItemProperty** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Get-ItemProperty.aspx

Per rendervi più facile l'accesso ai file di test creeremo un nuovo "drive" in Windows PowerShell, che punterà alla cartella di test sopramenzionata. Per farlo utilizzeremo il cmdlet *New-PSDrive*.

Create un nuovo drive con il comando che segue, cambiando il path specificato alla fine in maniera che punti alla tua cartella contenente i file di test:

New-PSDrive –name FK –psprovider FileSystem –root c:\CartellaDiTest

Utilizzate poi *cd FK:* per passare a quella cartella e verificate di essere nella cartella giusta. Se non lo siete, potete rimuovere il drive appena creato utilizzando *Remove-PSDrive FK* e riprovate.

Per ulteriori informazioni sui comandi sopra menzionati è possibile consultare la guida di riferimento:

New-PSDrive: http://www.powershell.it/Comandi/v1.0/New-PSDrive.aspx
Remove-PSDrive: http://www.powershell.it/Comandi/v1.0/Remove-PSDrive.aspx

Gli esercizi che seguono dovrebbero consentirvi di apprendere le potenzialità di Windows PowerShell. Naturalmente possiamo solo vedere una piccola parte di ciò che è possibile effettuare ma dovrebbe bastarvi per farvi capire i princìpi a supporto degli script. Se necessario, modificate le estensioni dei file dell'esempio affinchè combacino con quelle disponibili nella vostra cartella di test.

Passate al vostro drive di test utilizzando cd fk: (utilizzando la sintassi PowerShell, ad ogni modo, dovremmo usare Set-Location *fk:*). Visualizzatene il contenuto utilizzando *Get-ChildItem*. Escludete ora tutti i file temporanei: *Get-ChildItem * –exclude *.tmp, *.temp*

Per ulteriori informazioni sul comando **Set-Location** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Set-Location.aspx

B1: Visualizzate solo il nome del file e la dimensione, escludendo sempre i file temporanei con estensione tmp o temp. Suggerimento: usate la tecnica impiegata per i processi e i servizi.

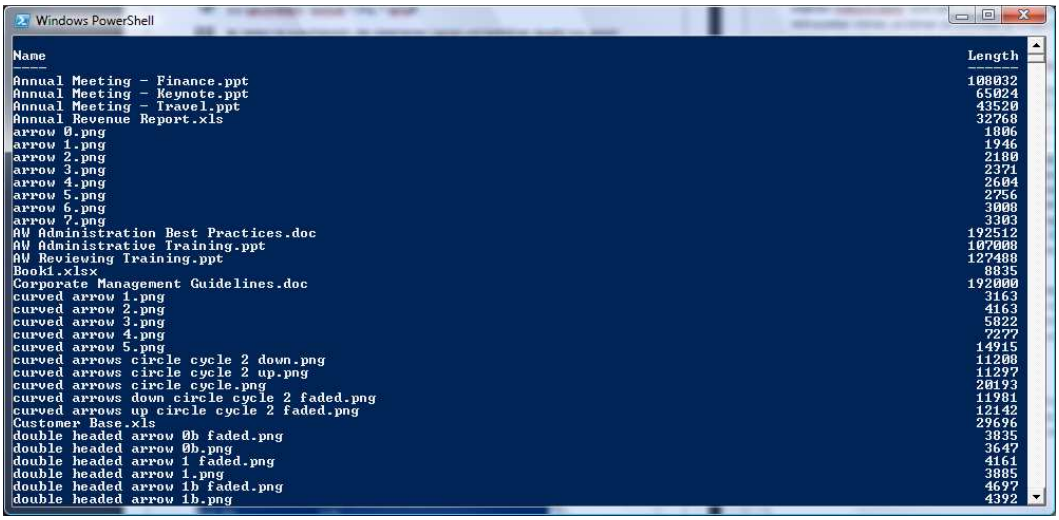

FIGURA 7: VISUALIZZA CIASCUN NOME E DIMENSIONE DI FILE ESCLUDENDO I FILE TEMPORANEI

Per ridurre il quantitativo di informazioni da digitare, Windows PowerShell offre diversi metodi per abbreviare i comandi.
Digitate *Get-Alias | Sort-Object Definition* e vedrete una lista di possibili varianti riconosciute per i comandi. Per i parametri, inoltre, utilizzate un certo numero di lettere iniziali affinchè il parametro sia completato automaticamente. Quindi:

*Get-ChildItem * –Exclude *.tmp | Select-Object Name, Length*

Può diventare:

*ls * –ex *.tmp | select n, le*

B2: Ordinate i file per dimensione (length), e per nome (name). Suggerimento: usate lo stesso metodo impiegato per i processi negli esempi precedenti.

Per ulteriori informazioni sul comando **Get-Alias** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Get-Alias.aspx

Recuperare informazioni sugli oggetti usando Get-Member

Usate *Get-Member* per ottenere una panoramica su tutte le proprietà ed i metodi di un oggetto. Per usare questa funzionalità passate a *Get-Member* un oggetto tramite pipe. Potete anche fornire a *Get-Member* una lista di oggetti simili.

B3: Create una lista di tutti i possibili attributi per un file utilizzando il cmdlet *Get-Member*. Ordinate tutti i file per la data di ultimo accesso. Suggerimento: usa il risultato del cmdlet *Get-Member* e prova ad indovinare l'attributo da usare dalla lista delle proprietà.

Per ulteriori informazioni sul comando **Get-Member** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Get-Member.aspx

Il cmdlet *Group-Object* può suddividere una lista di oggetti in gruppi. Per farlo dovete usare uno degli attributi dell'oggetto come argomento. *Get-Service | Group-Object status* genera perciò una nuova lista contenente due (o più) elementi. È utile anche il fatto che venga visualizzato il numero di servizi ed il loro stato corrispondente:

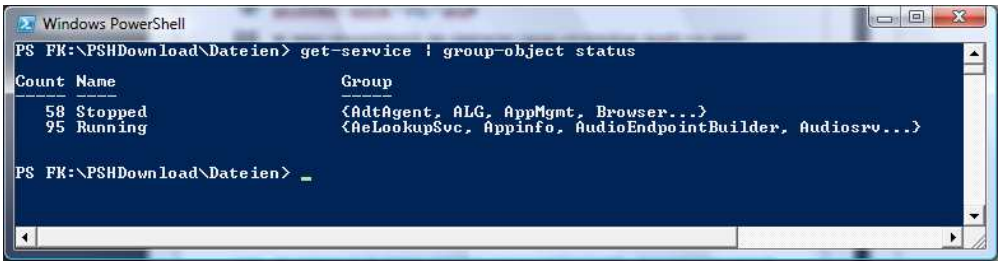

FIGURA 8: RISULTATI DEL CMDLET GROUP-OBJECT

B4: Raggruppate i file in base alla loro estensione. Poi ordinate il risultato in base al numero di file per ciascuna estensione. Suggerimento: recuperate i file, raggruppateli e poi ordinate la nuova lista in base al numero di elementi trovati (usate l'argomento *Count*).

Per ulteriori informazioni sul comando **Group-Object** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Group-Object.aspx

Esiste un altro utile cmdlet, al di là di *Get-Member*, per recuperare informazioni sugli oggetti: *Measure-Object*. Anche se non possiamo entrare nel dettaglio di tutte le opzioni disponibili per *Measure-Object*, possiamo almeno intravederne le possibilità usando qualche esempio. Provate ad indovinare quale sarà il risultato di questa sequenza di comandi:

Get-ChildItem | Measure-Object Length –Average –Sum –Maximum -Minimum

Avrete probabilmente notato che lo script potrebbe funzionare anche con altri cmdlet di partenza o inserendo caratteri wildcard.

Per ulteriori informazioni sul comando **Measure-Object** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Measure-Object.aspx

B5: Determinate la dimensione totale di tutti i file con estensione TMP. In uno step successivo, visualizzate a video SOLO la dimensione totale. Suggerimento: dopo il primo tentativo, inserite i cmdlet all'interno di parentesi tonde (). Dopo aver eserguito la catena di comandi, ripetete i comandi con l'aggiunta di *Get-Member* per visualizzare tutti gli attributi dei risultati (ricordate? Windows PowerShell lavora con gli oggetti e li converte in testo per permetterci di leggerli!). Trovate la proprietà che corrisponde al risultato desiderato e posizionatela all'interno di parentesi tonde (): in questo esempio stiamo cercando l'attributo "Total". Ricordate l'esempio del ciclo ForEach e come abbiamo recuperato la proprietà "Status"? Esattamente: "*Object.Status*". Ma qui stiamo cercando il totale ("Total") e non lo stato ("Status"), di conseguenza dovrete cambiare il comando.

Eliminare file

Windows PowerShell dispone di tutti i comandi necessari per eliminare file. Usando il cmdlet *Remove-Item* potete addirittura eliminare molto di più dei semplici file e funziona allo stesso modo di *Get-ChildItem*.
A questo punto sarebbe una buona idea fare una copia di backup della vostra cartella di test, così da permettervi un recupero veloce nel caso di eliminazioni accidentali.

B6: Eliminate tutti i file TMP usando il cmdlet *Remove-Item* con gli argomenti corretti!

Per ulteriori informazioni sul comando **Remove-Item** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Remove-Item.aspx

A volte potreste avere la necessità di eliminare tutti i file maggiori di una certa dimensione. In questo caso utilizzate il cmdlet *Where-Object*. Come per i blocchi condizionali *if*, potete definire una condizione che gli oggetti di una lista devono soddisfare prima di essere selezionati. Vediamo un esempio utilizzando i servizi. Usando:

Get-Service | Where-Object {$_.Status –eq "Stopped"}

verranno visualizzati a video solo i servizi arrestati.

B7: Ora eliminate tutti i file di dimensione maggiore di 2 MB, considerando che 2MB rappresentano indicativamente 2000000 di byte. Suggerimento: create il vostro script passo

dopo passo. Prima create la lista di tutti i file e poi filtratela per dimensione (…*Length –gt 2000000*). Ciò vi fornirà una nuova lista con cui lavorare usando un ciclo. Poi fornite in output solo il nome di ciascun file *($_.FullName*). Avrete bisogno di questi nomi di file per l'esecuzione del cmdlet finale, *Remove-Item*. Ogni tanto potete anche lavorare con le variabili se non volete far diventare le vostre righe di codice eccessivamente lunghe!

Per ulteriori informazioni sul comando **Where-Object** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Where-Object.aspx

Ad ogni modo, non avete bisogno di inserire il valore di 2 MB come 2000000 (e, dopo tutto, quest'ultima sarebbe stata solo un'approssimazione del valore corretto). Sarebbe meglio indicare direttamente 2MB come dimensione; e, fortunatamente, Windows PowerShell accetta anche questo valore senza alcun problema. Potete anche chiedere alla shell di calcolare la somma di 512KB + 512KB. Per i calcoli avete solo bisogno di digitare le operazioni all'interno della shell, senza bisogno di alcun cmdlet particolare.

Creare cartelle

Ora proviamo a fare un po' di ordine nel caos dei nostri file. Creeremo una sottocartella separata per ogni tipo di file e vi sposteremo dentro i relativi file. Per farlo abbiamo bisogno di un cmdlet in grado di creare nuovi elementi all'interno del file system[1]: *New-Item*. Questo cmdlet accetta il nome dell'elemento come argomento ed il tipo come parametro, come *Directory* per una cartella. Create una nuova cartella di nome "Test" come segue:

```
New-Item .\test   –type Directory
```

Per ulteriori informazioni sul comando **New-Item** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/New-Item.aspx

Per rendervi la vita più facile, torniamo brevemente di nuovo sul comando di ordinamento: *Get-Service | Sort-Object Status* lo conoscete già, quindi ora provate:

```
Get-Service | Sort-Object Status –Unique
```

Questo comando restituisce solo un elemento rappresentativo di ciascuno stato. Provatelo ora: disponete di tutto ciò di cui avete bisogno per creare cartelle.

[1] I file systems come FAT or NTFS non sono veri e propri sistemi orientati agli oggetti. Questo è il motivo per cui utilizziamo il cmdlet *New-Item* anzichè *New-Object*.

B8: Create una sotto-cartella separata per ciascuna estensione di file presente nella vostra cartella di test. Suggerimento: create una lista di tutti i file e recuperatene solo l'estensione (attributo "Extension"). Ora ordinatela utilizzando il parametro *–Unique*. Vedrete una lista di estensioni di file, ma ciascuna comparirà solo una volta. Quando sarete pronti, potrete assegnare questa lista ad una variabile e passare al prossimo step: usando un ciclo, ad ogni iterazione create una sottocartella con il nome di ciascuna estensione (*.Extension*). Ricordate che deve essere un percorso completo, che includa almeno un simbolo di backslash (\)! Se avete problemi con il percorso, utilizzate ("*.\New" + $_.Extension*) come argomento. Non dimenticate di fornire al cmdlet il tipo di elemento corretto (*Directory*) per creare una cartella!

Per spostare i file nella loro posizione finale possiamo utilizzare il cmdlet *Move-Item*. Come argomento, il cmdlet usa il percorso completo dell'oggetto iniziale ed il percorso di destinazione, es. *Move-Item .\test.txt .\newfolder*

B9: Spostate tutti i file dalla cartella di test alle sotto-cartelle appena create. Suggerimento: l'output di *Get-ChildItem* per la cartella di partenza ora include anche le nuove sotto-cartelle, che ora dovete escludere. Create un'altra lista di tutti gli elementi (verificate attentamente la lista prima di continuare) e filtratela utilizzando un operatore di confronto sul tipo di elemento (…Type *–notmatch "d")*. Poi avete bisogno di un ciclo per la lista, che ora contiene solo file. Lo step finale è semplice: per ciascun oggetto, trovate la cartella di destinazione corretta utilizzando l'estensione del file e spostate il file in quella cartella. Per memorizzare i risultati temporanei utilizzate delle variabili.

Per ulteriori informazioni sul comando **Move-Item** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Move-Item.aspx

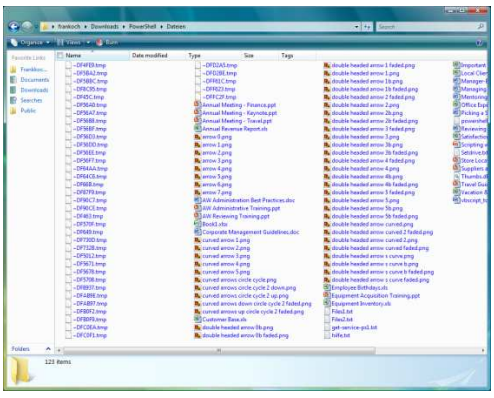

FIGURA 9: LA CARTELLA DI TEST PRIMA DEL RIORDINO

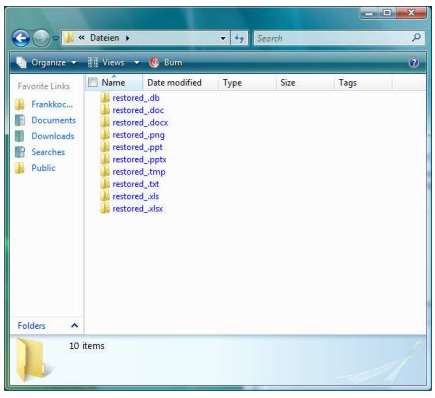

FIGURA 10: LA CARTELLA DI TEST DOPO DEL RIORDINO

Infine recuperiamo nuovamente tutti i file. Il percorso originale dovrebbe ora esserne privo direttamente ma contenere diverse sotto-cartelle, con diversi elementi ciascuna. Il comando *Get-ChildItem –Recurse* ci mostrerà il risultato nel dettaglio. Salviamo ora il risultato in un file TXT così da poterlo studiare utilizzando il Blocco note di Windows.

B10: Memorizzate il risultato dell'esercizio precedente incluse tutte le sotto-cartelle in un file di testo e salvatelo con nome FinalOutput.txt.

B11: Se avete i file originali dell'esercizio, potete fare ancora una cosa: per rendervi la vita ancora più semplice, possiamo resettare l'attributo di sola lettura per ciascun documento Word. Per ottenere questo risultato andate nella sotto-cartella dei file .doc/.docx e recuperate tutti gli oggetti. L'attributo da impostare per ciascun oggetto è chiamato *IsReadOnly* e deve essere impostato a 0 (numero zero). Suggerimento: usate due comandi. Create dapprima la lista di tutti gli oggetti, poi usate un ciclo per iterare tutti gli oggetti, come avete già fatto in diversi esercizi fino ad ora.

Windows PowerShell può essere d'aiuto anche nell'impostazione delle ACL, le impostazioni di sicurezza. Tramite i cmdlet *Get-ACL* e *Set-ACL* potete facilmente copiare le ACL di un oggetto e trasferirle ad un altro o anche generarne di nuove. Questa attività, ad ogni modo, è al di là degli scopi di questo workshop.

Per ulteriori informazioni sui comandi sopra menzionati è possibile consultare la guida di riferimento:

Get-ACL: http://www.powershell.it/Comandi/v1.0/Get-ACL.aspx
Set-ACL: http://www.powershell.it/Comandi/v1.0/Set-ACL.aspx

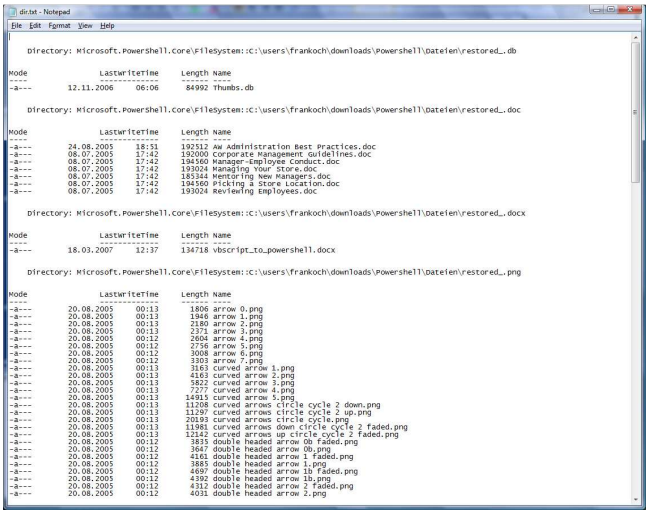

FIGURA 11: OUTPUT DEI CONTENUTI CON SOTTO-CARTELLE

Se vi rimane ancora un po' di tempo…

Torniamo per un attimo all'inizio e ai cmdlet *Export-CSV* e *Export-CLIXML*. Se è ancora disponibile, usate la variabile $P, altrimenti digitate questo comando all'interno di Windows PowerShell:

$p = Get-Process

Ora memorizzate la variabile in un file CSV e in un file CliXML:

$p | Export-CSV .\test.csv

$p | Export-CLIXML .\test.xml

Ora importate questi valori in due nuove variabili:

$p1 = Import-CSV .\test.csv

$p2 = Import-CLIXML .\test.xml

C1: Calcolate l'utilizzo medio, massimo e minimo del processore. Suggerimento: usate il comando *Measure-Object* come negli esempi precedenti. Fatelo separatamente per ciascuna delle tre variabili: $p, $p1 $p2.

C2: Ora ordinate le variabili, che sono mostrate come una lista, per utilizzo del processore e selezionatene i primi 5 elementi. Fatelo di nuovo per ciascuna delle tre variabili: $p, $p1 e $p2.
I risultati sono tutti uguali? Quale variabile si rivela diversa dalle altre in certe condizioni? Cosa c'è di sbagliato qui?

La soluzione è relativamente semplice. Durante l'esportazione di un file CSV e la seguente importazione, Windows PowerShell perde tutte le informazioni contestuali dei valori. Ciò significa che quando ordinate la lista i numeri diventano stringhe e 8.0 viene posizionato prima di 800. Nei file XML questa informazione è salvata ed è perciò disponibile per essere ordinata correttamente. Diventa perciò facile calcolarne il valore medio, la somma, etc.; in questo caso 8.0 + 800 dà come risultato 808.0 e quindi le valutazioni di *Measure-Object* sono corrette, ma non lo sono per gli ordinamenti. Prestate attenzione al formato in cui memorizzate le vostre variabili, quando le salvate su file; XML può essere il formato più sicuro!

LAVORARE CON ALTRI OGGETTI

Windows PowerShell: un elaboratore generico di oggetti

Con Windows PowerShell non siete limitati a lavorare con gli oggetti che avete creato voi stessi; avete la possibilità, infatti, di accedere ad un intero mondo di oggetti che include WMI, .NET e anche COM. Questi contenuti meriterebbero un workshop a sé stante ciascuno e richiederebbero del training a parte; nella bibliografia troverete diverso materiale per approfondire. Ci limiteremo, perciò, a rendervi una panoramica di ciò che è possibile effettuare e forniremo solo alcuni esempi, nella speranza di alimentare il vostro interesse su ciascun argomento.

Oggetti WMI

Probabilmente conoscete già gli oggetti WMI perchè avete lavorato con Windows Scripting Host (WSH) e VBScript. Se non è così, benvenuti in questa sezione ma, per favore, non aspettatevi qui nessuna discussione approfondita su WMI. Ci concentreremo solo sull'integrazione lato Windows PowerShell.

Potete generare un oggetto WMI in Windows PowerShell tramite l'apposito cmdlet *Get-WmiObject*; il fatto che ci sia un cmdlet dedicato dovrebbe dimostrare quanto WMI sia importante. Aprite una finestra di Windows PowerShell e digitate questo comando:

Get-WmiObject –Class Win32_ComputerSystem

Vedrete alcune informazioni di base sul vostro sistema. Al contrario di VBScript e altri linguaggi, Windows PowerShell non richiede l'utilizzo di una sintassi complessa, riducendo l'input al minimo assoluto. Avete solo bisogno:

- Del cmdlet *Get-WmiObject* per definire che vuoi lavorare con WMI
- Della classe WMI con cui vuoi lavorare (es. *-Class Win32_ComputerSystem*)

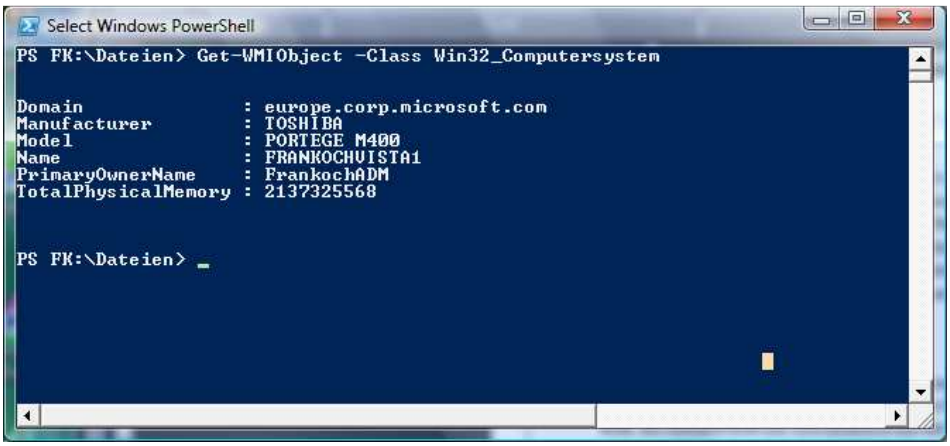

FIGURA 12: OUTPUT WMI DELLA CLASSE WIN32_COMPUTERSYSTEM

L'output è naturalmente solo una piccola porzione dei dati disponibili per l'oggetto; usando il cmdlet *Get-Member* potete visualizzare la lista di tutti gli attributi supportati.

D1: Visualizzate l'attributo del nome utente ("User name") per il vostro sistema. Suggerimento: partite dall'esempio iniziale su WMI e determinate l'attributo appropriato per il nome utente.

Per ulteriori informazioni sul comando **Get-WmiObject** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Get-WmiObject.aspx

WMI racchiude, come abbiamo detto, un mondo a sé stante. In WMI non solo potete recuperare le informazioni ma potete anche modificarle! E questa possibilità è estesa anche al di là del limite della propria macchina, ai sistemi esterni raggiungibili tramite rete, a patto che sia possibile autenticarvisi. Per approcciare il mondo di WMI potete utilizzare uno dei vari browser grafici di oggetti, come CIM Studio. Reperirete tutte le informazioni di cui avete bisogno nella bibliografia su questo argomento.

Ora vediamo alcuni esempi. Le classi WMI utilizzate di seguito sono spesso utilizzate nei task IT svolti più di frequente:

Recuperare informazioni sul desktop del PC:

Get-WmiObject -Class Win32_Desktop -ComputerName .

Il punto finale fa parte del comando ed indica che volete recuperare i dati del computer locale. In altri casi è possibile utilizzare il nome di un altro computer (server1, server2.mycompany.it, etc.).

Informazioni sul BIOS del sistema:

Get-WmiObject –Class Win32_Bios

Se desiderate lavorare sul computer locale, il parametro "*–ComputerName .*" è opzionale.

Elencare tutti gli hotfix installati nel sistema:

Get-WmiObject –Class Win32_QuickfixEngineering

Oppure, con un output più conciso:

Get-WmiObject –Class Win32_QuickfixEngineering –Property HotfixId |

Select-Object –Property HotfixId

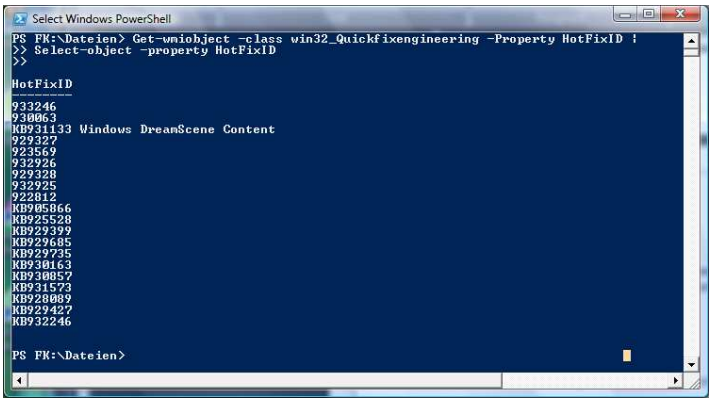

FIGURA 13: OUTPUT DEGLI HOTFIX INSTALLATI PER NUMERO DI KB

Modificare i dati all'interno degli oggetti WMI richiede lo stesso procedimento di tutti gli altri oggetti di Windows PowerShell. Utilizzate *Get-Member* per vedere attributi e metodi di ciascun oggetto e saprete subito se ciascun attributo sia fornito in sola lettura (*get*) o sia anche modificabile (*set*). L'attributo *"Visible"* nell'ultimo esempio COM è sia leggibile che modificabile.

Lavorare con oggetti .NET ed XML

Le potenzialità di impiego di .NET e XML sono impressionanti, alla stregua di WMI. Nell'appendice troverete due esempi: il primo si connette ad un sito web online, scarica un feed RSS e ne legge argomenti ed URL, come fosse il vostro reader RSS personale. Questo script è straordinariamente corto ed è inserito anche qui di seguito come dimostrazione; salteremo una discussione dettagliata sull'argomento per non andare al di là dello scopo di questo libro. Nel fantastico libro su Windows PowerShell intitolato Windows PowerShell in Action di Bruce Payette, uno dei padri fondatori di Windows PowerShell, troverete spiegazioni approfondite su entrambi gli script.

```
([xml](New-Object net.webclient).DownloadString(
"http://blogs.msdn.com/powershell/rss.aspx")).rss.channel.item |
Format-Table title,link
```

Sì, avete visto bene: l'intero script consiste in due sole linee di codice.

La sintassi è di nuovo simile a WMI. Per il momento non prestate attenzione alle parentesi quadre [XML]. C'è un semplice cmdlet – *New-Object* – che potete utilizzare voi stessi per creare nuovi oggetti; poiché abbiamo specificato esplicitamente il tipo desiderato (*Net.WebClient*), Windows PowerShell è subito in grado di capire con cosa ha a che fare: un oggetto .NET di tipo *WebClient*.

Per ulteriori informazioni sul comando **New-Object** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/New-Object.aspx

Utilizziamo poi il metodo *DownloadString(url)* di questo oggetto per recuperare il documento dall'URL specificato all'interno del sito web.

Il resto è solo un trucchetto: poichè sappiamo cosa si cela dietro l'URL specificata non andremo a controllare l'intero documento ma, invece, daremo per assunto che si tratti di un feed RSS e perciò recupereremo gli attributi che sappiamo esserci in un documento di questo tipo. Se specificassimo un URL diverso lo script andrebbe in errore, a meno di non puntare ad un altro feed RSS.

Format-Table è un altro cmdlet comunemente utilizzato ed il suo compito consiste nel visualizzare una lista di oggetti in un layout a tabella. Potete specificare direttamente le colonne di cui avete bisogno per il titolo (*title*) e l'URL (*link*) del feed RSS.

Per ulteriori informazioni sul comando **Format-Table** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Format-Table.aspx

Il secondo script di esempio è disponibile nell'appendice ed è indicato dal numero 4: questo script dimostra, tra l'altro, la possibilità di includere oggetti .NET complessi come finestre e controlli Windows all'interno di uno script di Windows PowerShell. Il risultato dovrebbe essere abbastanza impressionante da stimolare la tua sete di conoscenza!

Per maggiori informazioni sullo sviluppo mediante il framework Microsoft.NET e gli oggetti.NET consulta il sito MSDN o uno qualsiasi dei libri pubblicati sull'argomento.

Lavorare con gli oggetti COM

L'ultimo esempio proposto in questo testo illustra le possibilità offerte dall'integrazione con COM; nonostante gli esercizi che seguono utilizzino Excel, nel caso in cui non doveste averlo installato nel vostro PC potrete comunque prendere visione della versione alternativa che utilizza Internet Explorer. In questa introduzione, ad ogni modo, ci limiteremo a verificare le potenzialità di questo tipo di integrazione. Per approfondire le tematiche legate alla tecnologia COM consultate uno degli ottimi libri disponibili sull'argomento o il sito MSDN.

Prima di tutto è necessario indicare a Windows PowerShell che vogliamo lavorare con un oggetto COM, assegnando un nuovo oggetto ad una nuova variabile; questo primo passaggio ci assicurerà di lavorare con l'oggetto COM del tipo indicato. Non esiste un cmdlet specifico per utilizzare gli oggetti COM, come avviene per quelli WMI: per creare un oggetto COM si utilizza il cmdlet generico per la creazione di nuovi oggetti: *New-Object*.

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

Come argomento viene specificato il tipo di oggetto COM da utilizzare. In questo caso è Excel, indicato dall'argomento *Excel.Application*. Come presto imparerete, ci sono oggetti COM nel vostro sistema il cui nome esatto non è così ovvio, per cui conviene sempre consultare MSDN o uno dei molti libri disponibili sull'argomento.

Così come potremmo utilizzare Excel come collettore di dati per generare report, nell'attività di amministrazione IT quotidiana potremmo anche decidere di impiegare Visio per visualizzare graficamente dei valori del nostro sistema, utilizzando l'automazione resa possibile da Windows PowerShell. Se desiderate approfondire, ci sono diversi libri che trattano l'architettura degli oggetti COM, editi da Microsoft Press.

Ma torniamo ora all'esempio di Excel.

Per aggiungere dati abbiamo ora bisogno di un workbook. Poiché siamo entrati nel mondo di COM dobbiamo ora utilizzare la sintassi COM per raggiungere il nostro obiettivo. Windows PowerShell rende, fortunatamente, tutto il più semplice possibile e possiamo recuperare la lista dei metodi e degli attributi supportati dall'oggetto utilizzando *$a | Get-Member*. Sì, questo cmdlet funziona anche per gli oggetti COM: indicativamente, però, ci saranno più metodi disponibili per un applicativo come Excel che non per uno dei piccoli oggetti con cui abbiamo lavorato fino ad ora.
Il metodo *Workbooks.Add()* crea un nuovo workbook per noi; sono disponibili, inoltre, metodi per caricare un workbook esistente, specificando il nome ed il percorso di un file.

Per ora limitiamoci a creare un nuovo workbook:

```
$b = $a.Workbooks.Add()
```
[2]

[2] È possibile che in seguito a questo comando appaia un messaggio di errore simile a "Errore: 0x80028018 (-2147647512) Descrizione: Formato vecchio o Libreria di tipo non valida" durante la creazione di un workbook Excel. L'articolo della Knowledge Base Microsoft #320369 descrive il problema; di solito l'errore compare quando Excel è installato in una lingua differente (es. English (US)) da quella delle impostazioni internazionali di Windows (es. English (UK)).

I fogli di lavoro (worksheet) appartengono automaticamente al workbook. Prendiamo il primo:

$c = $b.Worksheets.Item(1)

Se vogliamo scrivere nel foglio di lavoro dobbiamo farlo non direttamente in quest'ultimo ma in una linea (in una cella, più precisamente) del foglio stesso. Ciò significa che è necessario utilizzare un comando di questo tipo:

$c.Cells.Item(1,1) = "powershell.it è un sito fantastico!"

Ecco fatto, i nostri dati in Excel sono pronti.
Non mi credete? Ok, allora eseguite questo comando:

$a.Visible = $True

Come per magia, impostando l'attributo "Visible" del vostro oggetto Excel a $true, farete apparire a video l'interfaccia del software con i dati immessi.

Così come abbiamo utilizzato l'attributo "Visible", possiamo anche richiamare i metodi (funzioni) di Excel ed utilizzarli per i nostri scopi. Potete sicuramente indovinare cosa effettua questa linea di codice, ne sono sicuro:

$b.SaveAs(".\Test.xls")

E, nel caso ve lo steste chiedendo: no, non stiamo salvando l'oggetto Excel in questo caso ma, invece, stiamo salvando il workbook così come faremmo utilizzando l'interfaccia di Excel. Siamo interessati al file XLS prodotto, non al programma in sé.

Questa demo, naturalmente, è incompleta; non vi ho mostrato nel dettaglio come ottenere le informazioni che vi consentono di manipolare Excel: come abbiamo fatto, ad esempio, a sapere che le cartelle di lavoro di Excel sono chiamate "workbook"? E come abbiamo fatto a conoscere il comando "SaveAs" e la sua sintassi? L'interfaccia di Excel è disponibile come oggetto COM e potete utilizzare qualsiasi fonte di informazioni su COM per imparare questi dettagli ed utilizzarli, successivamente, all'interno di Windows PowerShell; *Get-Member* è sempre di grande aiuto in questi casi. Poiché stiamo qui discutendo di Windows PowerShell, ci siamo limitati ad utilizzare le informazioni e non mostrare le fonti utilizzate per acquisirle.

Alla fine dovremo effettuare il clean up e chiudere Excel, se non l'avete già fatto voi manualmente:

$a.Quit()

Se lo desiderate, provate a risolvere questo piccolo esercizio:

D2: Create una lista di tutti i servizi e memorizzatene nome e stato di ciascuno in un foglio di lavoro Excel. Suggerimento: utilizzate l'esempio precedente per creare un oggetto Excel e assegnatelo a una variabile. Per gestire le linee all'interno del foglio Excel potete utilizzare una variabile separata, *$i*. Prendete lo script per l'output colorato dei servizi e sostituiteci il comando per l'output con *$c.Cells.Item($i,1)*. Non dimenticate di incrementare *$i* dopo ogni

linea, utilizzando *$i = $i + 1*. Potete inserire più comandi in un'unica linea utilizzando il punto e virgola ';'. Infine, salvate il risultato dell'Excel in un file XLS utilizzando, per piacere, l'automazione dell'oggetto COM di Excel e non procedendo manualmente dal menu.

Nel file che ne risulterà noterete che lo stato dei servizi è visualizzato come un numero all'interno di Excel. Windows PowerShell, invece, ci semplifica la vita e visualizza tale valore come "Running" o "Stopped", in maniera tale che sia più semplice leggere le informazioni.

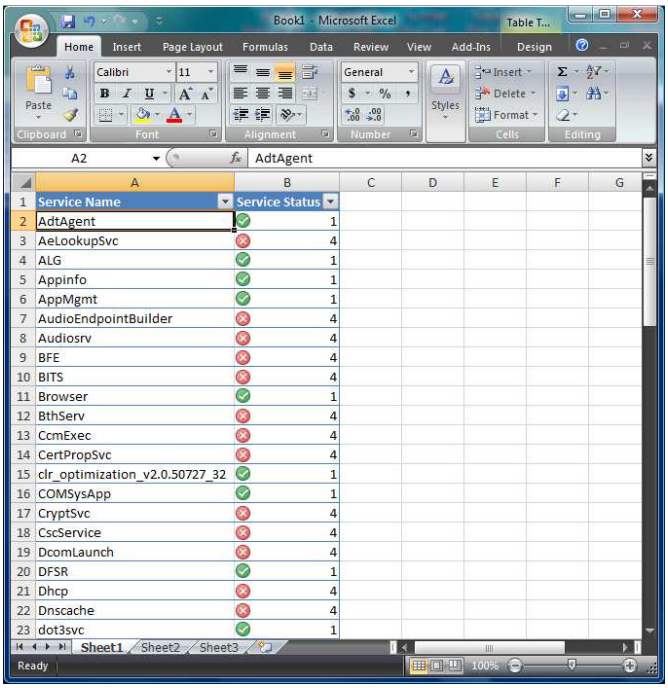

FIGURA 14: OUTPUT DEI SERVIZI E DEL LORO STATO IN EXCEL 2007. QUI VENGONO UTILIZZATE, INOLTRE, LE FUNZIONALITÀ SPECIFICHE DI EXCEL 2007 PER VISUALIZZARE LO STATO DEI SERVIZI COME ICONA INVECE CHE COME NUMERO.

Se non avete installato Excel nel vostro PC, potete effettuare un esercizio differente utilizzando Internet Explorer: invece che inserire dei dati in un foglio di lavoro navigherete in un sito web. Una volta combinato questo meccanismo con il reader RSS menzionato in precedenza, avrete solo bisogno di riuscire a scaricare un feed RSS automaticamente, ricercare un testo nel titolo e, una volta trovato l'elemento di interesse, aprirne automaticamente la relativa pagina.
Navigare dal vostro divano non è mai stato così facile!

Lo script inizia in maniera simile a quello di Excel, richiamando *New-Object:*

$ie = New-Object –ComObject InternetExplorer.Application

Anche qui il nostro nuovo oggetto è inizialmente invisibile e, come in Excel, la soluzione è molto semplice:

$ie.Visible = $True

Per trovare tutte le funzionalità esposte da Internet Explorer utilizziamo il cmdlet Get-Member:

$ie | Get-Member

Per semplificare le cose possiamo navigare in un sito ben conosciuto:

$ie.Navigate("http://www.powershell.it")

Se ne avete voglia, potete combinare questo script con quello del reader RSS; se aveste a disposizione un numero sufficiente di elementi, potreste prendere la lista dei titoli, cercare un determinato testo e, trovato un risultato, aprirne il collegamento correlato in Internet Explorer.

In questo workshop non è stata analizzata la funzionalità di ricerca di testi presente all'interno di Windows PowerShell ma vi esorto ad utilizzare l'help in linea per ricercare maggiori informazioni.

Lavorare con il registro degli eventi

Per completare gli esercizi pratici compresi all'interno di questo libro, discuteremo ora brevemente del registro degli eventi di Windows.
Windows PowerShell consente l'accesso al registro degli eventi utilizzando diversi metodi: all'interno di WMI, .NET e COM, infatti, sono presenti alcuni oggetti utili a questo scopo ma anche Windows PowerShell stesso mette a disposizione alcuni cmdlet in grado di aiutarci in questo compito. Il più importante di questi è *Get-EventLog.*

Il comando *Get-EventLog –List* visualizza tutti i registri degli eventi presenti nel sistema. Per accedere ad uno specifico registro possiamo utilizzare il cmdlet *Where-Object*. Potete dunque accedere al registro degli eventi di sistema (*System*) come segue:

```
Get-EventLog -List | Where-Object {$_.LogDisplayName -eq "System"}
```

Tuttavia, deve esserci un modo più semplice per farlo, altrimenti questa linea di codice sarebbe troppo lunga per essere utilizzata quotidianamente. Ed effettivamente, se volete recuperare le righe da uno specifico registro degli eventi, potete richiamare semplicemente *Get-EventLog*, aggiungendovi il nome del registro degli eventi desiderato. I risultati possono a volte essere molto lunghi e, per annullare l'operazione, potete premere CTRL+C in qualsiasi momento. Se volete visualizzare solo le 20 righe più recenti del registro utilizzate il parametro *–Newest 20*. Naturalmente potete sostituire il 20 con un numero qualsiasi.

```
Get-EventLog System -Newest 3

Get-EventLog System -Newest 3 | Format-List
```

Per ulteriori informazioni sul comando **Get-EventLog** è possibile consultare la guida di riferimento, a questo indirizzo:

http://www.powershell.it/Comandi/v1.0/Get-EventLog.aspx

Le righe recuperate dal registro degli eventi sono disponibili per l'elaborazione, come qualsiasi altro oggetto all'interno di Windows PowerShell e possono essere di conseguenza ordinate e raggruppate.

E1: Ricercate il nome del registro degli eventi di Windows PowerShell. Raggruppate gli eventi recuperati per identificativo (*Event ID*), poi ordinateli per nome. Poi recuperate solo gli eventi con identificativo pari a 403. Suggerimento: se il registro degli eventi contiene uno spazio allora specificatene il nome utilizzando i doppi apici.

E2: Ordinate i 15 eventi più recenti del registro degli eventi di sistema in ordine decrescente per identificativo (*Event ID*). Suggerimento: rileggete questo libro un'altra volta se non siete in grado di trovare una soluzione immediata.

SOLUZIONI DEGLI ESERCIZI

Script delle soluzioni degli esercizi proposti nel libro

A1

```
Get-Process | Sort-Object CPU
```

A2a

```
Get-Process | Sort-Object CPU –Descending | Select-Object –First 10

Get-Process | Sort-Object CPU | Select-Object –Last 10
```

A3

```
$P = Get-Process | Sort-Object CPU –Descending | Select-Object –First 10
```

A4

```
$P > .\A4.txt

$P | Export-CSV .\a4.csv

$P | Export-CLIXML .\a4.xml
```

A5

```
Get-Service | Sort-Object Status
```

A6

```
Get-Service | ForEach-Object{ Write-Host $_.Name $_.Status}
```

A7

```
Get-Service | ForEach-Object{ Write-Host –F Yellow –B Red $_.Name $_.Status}
```

Al posto di –F potete anche scrivere –ForegroundColor è anzichè –B potete scrivere –BackgroundColor.

A8

```
Get-Service | ForEach-Object{

if ($_.Status -eq "Stopped") {Write-Host -F Green $_.Name $_.Status}`

else{ Write-Host -F Red $_.Name $_.Status}}
```

A9

```
Get-Service | ConvertTo-HTML
```

A10

```
Get-Service | ConvertTo-HTML > .\a10.html
```

A11

Get-Service | Sort-Object Status | ConvertTo-HTML Name, Status > .\a11.html

B1

```
Get-ChildItem * -Exclude *.tmp | Select-Object Name, Length
```

B2

```
Get-ChildItem * -Exclude *.tmp | Select-Object Name, Length | Sort-Object Length, Name
```

B3

```
Get-ChildItem | Get-Member
```

B4

```
Get-ChildItem | Group-Object Extension | Sort-Object Count
```

B5

```
(Get-ChildItem .\*.tmp | Measure-Object Length –Sum).Sum
```

B6

```
Remove-Item .\*.tmp
```

B7

```
Get-ChildItem | Where-Object {$_.Length –gt 2000000} | ForEach-Object {Remove-Item
$_.Fullname}
```

B8

```
Get-ChildItem | Select-Object Extension | Sort-Object Extension -Unique `

| ForEach-Object {New-Item (".\New" + $_.Extension) -Type Directory}
```

B9

```
Get-ChildItem | Where-Object {$_.Mode -notmatch "d"} | ForEach-Object {$b= ".\New" + `

 $_.Extension; Move-Item $_.Fullname $b}
```

B10

```
Get-ChildItem –Recurse > .\finaloutput.txt
```

B11

```
Get-ChildItem *.doc | ForEach-Object {$_.IsReadOnly = 0}
```

C1

```
$p | Measure-Object CPU -Min -Max -Average
```

C2

```
$p | Sort-Object CPU -Descending | Select-Object -First 5
```

D1

```
(Get-WmiObject -Class Win32_ComputerSystem).UserName
```

D2

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

$a.Visible = $True

$b = $a.Workbooks.Add()

$c = $b.Worksheets.Item(1)

$c.Cells.Item(1,1) = "Service Name"

$c.Cells.Item(1,2) = "Service Status"

$i = 2

Get-Service | ForEach-Object{ $c.cells.item($i,1) = $_.Name; $c.cells.item($i,2) = $_.Status; $i=$i+1}

$b.SaveAs("C:\Test.xls")

$a.Quit()
```

E1

```
Get-EventLog "Windows PowerShell" | Group-Object Eventid | Sort-Object Name

Get-EventLog "Windows PowerShell" | Where-Object {$_.EventId -eq 403}
```

E2

```
Get-EventLog System -Newest 15 | Sort-Object Eventid -Descending
```

APPENDICE
SCRIPT DI ESEMPIO

Esempi di Windows PowerShell – dal semplice al complesso

Potete semplicemente copiare gli script che trovate qui di seguito ed eseguirli direttamente all'interno di Windows PowerShell; ciascuno dimostra le possibilità e le potenzialità di PowerShell ma rende evidente anche il fatto che una breve introduzione come quella rappresentata da questo libro non può ricoprire in nessun modo tutti gli aspetti di questo prodotto eccezionale.
Per copiare gli script, evidenziate le linee di testo al di sotto del primo simbolo > all'interno dei commenti (escludendo questi ultimi, naturalmente). Questi esercizi sono principalmente tratti dal libro "Windows PowerShell in Action", scritto da Bruce Payette, uno dei padri fondatori di Windows PowerShell, dove troverete anche un'analisi più dettagliata di ciascuno esempio, nel caso abbiate bisogno di maggiori informazioni.

Esempio 1: Output diretto di una stringa

Il più breve programma "Hello world" mai realizzato: >

```
"Hello world"
```

Commenti:
Windows PowerShell riconosce le stringhe e può effettuarne direttamente l'output.

Esempio 2: Analisi dei file di log

Create una lista di tutti i file di log nella cartella di Windows (variabile d'ambiente "Windir"); cercate all'interno dei file di log la parola "Error", visualizzate in output il nome del file di log e la linea con l'errore: >

```
dir $env:windir\*.log | Select-String -List Error | Format-Table Path,LineNumber –AutoSize
```

Commenti:
Ci potrebbero essere dei messaggi di errore dovuti a problemi di autorizzazione di accesso alle cartelle. Potete ignorare questi messaggi.

Esempio 3: Il vostro reader di feed RSS

Recuperate da un dato indirizzo un feed RSS e visualizzatene i post, riportando il link di ciascuno: >

```
([xml](New-Object net.webclient).DownloadString(
"http://blogs.msdn.com/powershell/rss.aspx")).rss.channel.item | Format-Table Title,Link
```

Commenti:
Talvolta la prima connessione con un alcuni siti potrebbe essere lenta.

Esempio 4: Aggiungere finestre Windows agli script

Create una finestra Windows per visualizzare graficamente delle informazioni (o altro): >

```
[void][reflection.assembly]::LoadWithPartialName("System.Windows.Forms")

$form = New-Object Windows.Forms.Form

$form.Text = "La mia prima form"

$button = New-Object Windows.Forms.Button

$button.text="Clicca qui!"

$button.Dock="fill"

$button.add_click({$form.close()})

$form.controls.add($button)

$form.Add_Shown({$form.Activate()})

$form.ShowDialog()
```

Commenti:
Per uscire è sufficiente fare click sul pulsante “Clicca qui!” del form appena creato dallo script (potrebbe essere nascosto dalla finestra di Windows PowerShell).

APPENDICE
PRINCIPI TEORICI DI WINDOWS POWERSHELL

Windows PowerShell – Una breve introduzione

I precedenti tentativi di realizzare una shell a riga di comando effettuati da Microsoft nel passato sono stati insoddisfacenti; il vecchio command.com poteva essere adeguato per le prime versioni di MS DOS ma il crescente numero di funzionalità offerte dal sistema operativo ne ha presto determinato il superamento delle capacità. La shell cmd.exe, introdotta a partire da Windows NT offriva molte più possibilità; tuttavia, paragonate alle popolari shell disponibili in ambiente Unix, come Bash, le shell a riga di comando di Microsoft non erano all'altezza della situazione.

Con Windows PowerShell (in precedenza nota come Monad Shell, MSH) Microsoft ha completamente ribaltato la situazione, offrendo agli amministratori di sistema la possibilità di scrivere script in grado di sfruttare e controllare tutte le funzionalità dei nostri sistemi. Windows PowerShell è basata su di un nuovo concetto, completamente diverso da quello utilizzato dalle shell orientate al testo, come Bash.

Obiettivi di Windows PowerShell

Windows PowerShell è una nuova shell a riga di comando per Windows, sviluppata specificatamente per gli amministratori di sistema. La shell consiste in una riga di comando interattiva e in un ambiente di esecuzione per script, in grado di essere utilizzati singolarmente o insieme. In contrasto con la maggioranza delle shell che accettano e ritornano testo, Windows PowerShell è basata sul modello ad oggetti reso disponibile dal framework Microsoft .NET 2.0. Questo cambiamento fondamentale nell'ambiente ha permesso di utilizzare strumenti e metodi completamente nuovi per gestire e configurare Windows.

Windows PowerShell introduce l'idea dei cmdlet (pronunciato "commandlet"). Un cmdlet è un semplice strumento utilizzabile dalla riga di commando, integrato nella shell e in grado di eseguire una sola funzione. Nonostante sia possibile utilizzare i cmdlet singolarmente, le loro potenzialità si esprimono al meglio utilizzandoli in combinazione tra di loro per svolgere compiti complessi. Windows PowerShell contiene diverse centinaia di cmdlet di base ed è possibile scrivere i propri cmdlet e fornirli ad altri affinché siano utilizzati.

Come molte altre shell, Windows PowerShell consente di accedere al file system del computer e, grazie al concetto di provider, anche ad altri contenitori di dati e documenti, come il registry o gli store dei certificati digitali.

Testo, parser e oggetti

PowerShell è completamente orientato agli oggetti. Confrontato con le altre shell, però, tratta l'output di ciascun commando non come un testo ma come un oggetto (questo concetto verrà

approfondito nel prosieguo). E, ancora, alla stregua delle altre shell, PowerShell utilizza la *pipeline*, che consente di passare il risultato di un comando al successivo. La sola differenza è che i valori di input e quelli di output sono oggetti anziché testo.

Gli oggetti di PowerShell non sono molto differenti da quelli impiegati nei programmi sviluppati in C++ o in C#: è possibile immaginare un oggetto come un'unità contenente dati, con attributi e metodi. Questi ultimi corrispondono alle azioni che possono essere eseguire sull'oggetto.

Se, ad esempio, richiamate un *servizio* in Windows PowerShell, state in realtà utilizzando l'oggetto che lo rappresenta; e se visualizzate informazioni su di un elemento di quel tipo, state visualizzando gli attributi del relativo oggetto. E se avviate un servizio, impostandone l'attributo **Status** pari a **Started**, state usando un metodo dell'oggetto relativo al servizio. A mano a mano che aumenterà la vostra esperienza, capirete meglio i vantaggi dell'impiego degli oggetti e potrete sfruttarne le potenzialità.

La tecnologia orientata agli oggetti di PowerShell rende ormai superati i parser delle comuni shell per Linux, così come le informazioni basate su testo, che tali comandi forniscono e accettano. Per rendere le cose più chiare forniremo un esempio concreto.
Supponete di voler recuperare la lista di tutti i processi del vostro sistema a cui sono associati più di 100 handle; con una tradizionale shell Linux probabilmente richiamereste il commando per visualizzare i processi (ps -A). Il commando ritorna una lista di testo, dove ogni linea contiene informazioni su di un processo, separate da spazi. Dovreste effettuare prima il parsing di queste righe utilizzando un tool, recuperare l'identificativo dei processi e richiedere poi, con un altro programma, il numero di handle per ciascun processo. Dovreste, in seguito, portare a termine il parsing di quest'altro risultato testuale, recuperare le linee di interesse e, infine, visualizzare gli elementi che stiamo cercando.

A seconda di quanto bene funziona il meccanismo di recupero e il filtro delle informazioni dalle liste di testo, questo approccio è più o meno affidabile. Ma se, per esempio, il titolo di una colonna nell'output testuale dovesse cambiare o il nome di un processo fosse troppo lungo andreste probabilmente incontro a qualche problema.

PowerShell utilizza un approccio completamente diverso. Anche in questo caso l'operazione comincia con il comando *Get-Process*, che ritorna tutti i processi in esecuzione nel sistema operativo. Solo che qui sono restituiti come *una lista di oggetti*, ciascuno dei quali descrive un processo. Questi oggetti possono poi essere esaminati per recuperarne gli attributi supportati e richiederne direttamente i valori, evitando, quindi, di esaminare le linee di un file di testo e recuperare le colonne di un output testuale. Questo argomento verrà approfondito ulteriormente nel prosieguo.

Un nuovo linguaggio di scripting

Windows PowerShell non utilizza un linguaggio esistente ma ne impiega uno nuovo, creato per queste ragioni:

- Windows PowerShell avrebbe avuto bisogno di un liguaggio per gestire gli oggetti .NET.
- Il linguaggio avrebbe dovuto supportare elaborazioni complesse senza però rendere quelle semplici complicate da effettuare.

- Il linguaggio avrebbe dovuto essere compatibile con gli standard e le convenzioni degli altri linguaggi supportati da .NET, come C#.

Ogni linguaggio ha i propri comandi; in Windows PowerShell ci si è assicurati che tutti questi comandi seguissero una logica specifica per quanto riguarda architettura e nomenclatura. Un *cmdlet* è un comando specializzato che lavora attraverso gli oggetti all'interno di Windows PowerShell. Potete riconoscere i cmdlet dai loro nomi: un verbo ed un nome, sempre in forma singolare, separati da un trattino (-), ad esempio *Get-Help, Get-Process* e *Start-Service*. In Windows PowerShell la maggior parte dei cmdlet sono molto semplici e sono stati pensati per essere in grado di funzionare insieme tra loro. Di conseguenza, per esempio, i cmdlet 'Get' si limitano ad ottenere dei dati, i cmdlet 'Set' generano o modificano i dati, i cmdlet 'Format' li formattano e i cmdlet 'Out' si limitano a dirottare l'output verso una destinazione specifica.

Comandi di Windows e programmi di servizio

Con il tempo vi sarete probabilmente abituati ad alcuni comandi forniti dalle vostre shell preferite; un nuovo linguaggio che non tenga in considerazione le abitudini dei propri utilizzatori sarebbe destinato a fallire nel suo compito, velocemente. Ecco perché in Windows PowerShell potete eseguire anche la maggior parte dei comandi standard di Windows e lanciare eseguibili dotati di interfaccia grafica, come Blocco note o Calcolatrice. Anche in PowerShell, inoltre, così come avveniva in **Cmd.exe**, è possibile importare il testo di output generato da programmi esterni ed utilizzarlo all'interno della shell. Anche se comandi come *dir, ls* o *cd* non seguono la sintassi ufficiale di Windows PowerShell, funzionano e possono essere utilizzati senza problemi.

Un ambiente interattivo

Così come altre shell, anche Windows PowerShell supporta un ambiente completamente interattivo. Se viene inserito un comando al prompt, questo viene elaborato e ne viene mostrato il risultato (output) nella finestra della shell. Potete spedire l'output da un comando ad un file o ad una stampante o utilizzare l'operatore pipeline (|) per dirottarlo ad un altro comando.

Supporto script

Se eseguite spesso gli stessi blocchi di comandi, vi raccomandiamo di non digitarli ogni volta che vi occorrono ma di salvarli su di un file ed eseguire semplicemente quest'ultimo al bisogno. Un file che contiene comandi è chiamato script.

Al di là dell'interfaccia interattiva, Windows PowerShell offre anche un supporto completo per gli script. È possibile eseguire uno script digitandone il nome del file al prompt della shell; l'estensione dei file di script di Windows PowerShell è **.ps1** ma è opzionale quando si eseguono all'interno della shell.

Nonostante gli script siano estremamente utili, potrebbero anche essere sfruttati per diffondere codice in grado di provocare danni. Proprio per questo è possibile definire all'interno di Windows

PowerShell delle policy di sicurezza (chiamate anche policy di esecuzione) per specificare quali script possano essere eseguiti e se questi debbano essere firmati digitalmente. Per evitare possibili rischi, non è consentita in alcuna policy l'esecuzione uno script facendo doppio click sul relativo file, così come accadeva, ad esempio, per i vecchi file .bat, .cmd o .vbs.

CMD, WScript o PowerShell? Quale scegliere?

A partire da Windows XP ci sono tre shell disponibili: la buona vecchia shell CMD, Windows Scripting Host per i vostri script (VBScript o JScript), ed ora Windows PowerShell.
Non abbiate timore, non dovete scegliere quale shell utilizzare o preoccuparvi dell'eventuale obsolescenza di alcune tra quelle menzionate: anche nelle più recenti versioni di Windows, come Vista o Windows Server 2008, è infatti possibile trovarle tutte e tre. Utilizzate quella che preferite, in base ai vostri gusti. Potete addirittura scegliere di volta in volta la shell che meglio si addice all'elaborazione di un task specifico.
Se non avete scritto molti script fino ad oggi forse vale la pena di iniziare con Windows PowerShell, così da apprendere la tecnologia più recente e più semplice da utilizzare. Avendo studiato le sezioni di utilizzo pratico di questo libro dovreste aver notato quanto intuitivi e potenti possono essere degli script realizzati utilizzando questa tecnologia, senza aver avuto bisogno di recuperare e studiare quei tomi degli anni '60 e '70 che trattano la programmazione batch…

Windows PowerShell 1.0 e 2.0

Anche se la versione ufficiale corrente di Windows PowerShell è la 1.0, la qualità di questo prodotto è estremamente convincente e ne raccomando l'utilizzo anche in ambienti di produzione, poiché questa shell è semplicemente un'altra modalità di utilizzo del framework .NET 2.0, maturo ed affidabile.

Solo osservando le funzioni disponibili ci si può rendere conto che Windows PowerShell non contiene ancora tutti i comandi per gestire ogni aspetto del proprio sistema; Windows PowerShell può già accedere direttamente al file system, al log degli eventi, al registry e agli oggetti ed alle interfacce .NET, WMI e ADSI; ad ogni modo ad oggi non è ancora possibile, ad esempio, gestire tramite remoting un sistema remoto mediante PowerShell. Fortunatamente, la versione 2.0 di questo prodotto, disponibile a breve, permetterà anche di superare questo gap.

APPENDICE
SCRIPT E SICUREZZA

Sicurezza nell'utilizzo degli script

Attraverso l'inclusione di Windows Scripting Host (WSH) all'interno di Windows 2000, Microsoft ha introdotto un motore di scripting nuovo e potente. Questo motore era così potente che fu presto usato come campo di battaglia per gli autori di virus. Gli utenti senza esperienza che ricevevano le prime email con delle immagini allettanti e ne aprivano gli allegati non vedevano nulla, perchè ciò che avevano aperto era in realtà uno script VBScript che infettava il loro sistema. Windows PowerShell è progettato per difendere l'utente da questo tipo di minacce.

Le impostazioni di sicurezza di base di Windows PowerShell indicano perciò di non eseguire alcuno script. Questa funzionalità deve essere, infatti, attivata esplicitamente da un amministratore di sistema.
L'attivazione è organizzata secondo un certo numero di livelli, disponibili grazie alla firma digitale applicata agli script. In aggiunta, l'estensione di Windows PowerShell (.ps1) è collegata al Blocco note. Anche se il vostro ambiente consente l'esecuzione di script, quindi, un doppio click su di un allegato o su di un file effettuato inavvertitamente su di uno script aprirebbe semplicemente il Blocco note e ve ne mostrerebbe il codice sorgente. E, infine, Windows PowerShell richiede sempre di specificare il percorso ".\" per i file che risiedono nella cartella corrente, cercando quindi di prevenire l'esecuzione di programmi che non si vorrebbero realmente lanciare.

Per essere in grado di eseguire script è necessario modificare le impostazioni di sicurezza di Windows PowerShell. Vi sono due cmdlet realizzati per questo scopo: *Get-ExecutionPolicy* e *Set-ExecutionPolicy*. Con *Get-ExecutionPolicy* è possibile recuperare le impostazioni correnti. Esistono quattro livelli di sicurezza:

Policy	**Script eseguibili**
Restricted (*predefinita*)	Nessuno
AllSigned	Solo gli script con firma digitale
RemoteSigned	Gli script creati localmente e tutti gli altri purchè siano firmati digitalmente
Unrestricted	Tutti

Per modificare queste impostazioni, un amministratore di sistema dovrebbe eseguire, ad esempio, il comando:

Set-ExecutionPolicy RemoteSigned

Microsoft fornisce anche un modello per Policy di gruppo in grado di effettuare questa impostazione automaticamente all'interno dei domini Active Directory.

Per ulteriori informazioni sui comandi sopra menzionati è possibile consultare la guida di riferimento:

Get-ExecutionPolicy: http://www.powershell.it/Comandi/v1.0/Get-ExecutionPolicy.aspx
Set-ExecutionPolicy: http://www.powershell.it/Comandi/v1.0/Set-ExecutionPolicy.aspx

CONCLUSIONI

Frank Koch desidera ringraziare sua moglie Petra per il suo amore e la sua pazienza: è stata costretta a rinunciare a diversi fine settimana e a diverse serate in compagnia del marito, mentre quest'ultimo passava il suo tempo al computer anzichè con lei.

Efran Cobisi ringrazia la sua compagna Laura, perché l'esistenza dell'edizione italiana di questo libro è merito esclusivo della sua pazienza e della sua amorevole comprensione. E forse anche delle cene deliziose che lei sa preparare… Grazie, amore.

Non sarebbe stato possibile scrivere questo libro senza gli sviluppatori di Windows PowerShell, e così a loro va il nostro ringraziamento più vivo; un ringraziamento speciale va anche al corso di Bruce Payette, che ci ha dato la forza per scrivere le prime bozze grazie al suo libro "Windows PowerShell in Action". Vivamente raccomandato, leggetelo se potete!

Non tutti i contenuti di questo libro sono stati realizzati da noi, in particolare le sezioni teoriche che sono state riprese dai contenuti pubblicati sulle pagine di Microsoft MSDN e dalla guida in linea di Windows PowerShell. Lì troverete maggiori informazioni tecniche, che vi incoraggio a leggere.

Se desiderate contattare l'autore, **Frank Koch**, potete farlo via email (in lingua inglese o tedesca). Potrebbe non essere in grado di rispondere a tutti di persona, ma sarà molto grato a chi vorrà complimentarsi o criticare il suo lavoro in maniera costruttiva: frankoch@microsoft.com

Se desiderate contattare il responsabile dell'edizione italiana, **Efran Cobisi**, utilizzate l'apposito form all'interno del sito della community italiana di Windows PowerShell, powershell.it, a questo indirizzo: http://www.powershell.it/Supporto/Contatta-il-team.aspx

PowerShell Cheat Sheet

Comandi essenziali

Per accedere alla guida su qualsiasi cmdlet utilizzate Get-Help

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

Per visualizzare tutti i cmdlet disponibili usate Get-Command

```
Get-Command
```

Per ottenere proprietà e metodi di un oggetto usate Get-Member

```
Get-Service | Get-Member
```

Impostare le policy di sicurezza

Visualizzate e modificate le policy di esecuzione con Get-ExecutionPolicy e Set-ExecutionPolicy

```
Get-ExecutionPolicy
Set-ExecutionPolicy RemoteSigned
```

Per eseguire uno script

```
powershell.exe -noexit &"c:\myscript.ps1"
```

Variabili

Il nome inizia con $

```
$a = 32
```

Possono essere tipizzate

```
[int]$a = 32
```

Array

Per inizializzarli

```
$a = 1,2,4,8
```

Per leggerli

```
$b = $a[3]
```

Funzioni

I parametri sono separati da spazio.
Ritornare un valore è opzionale.

```
function sum ([int]$a,[int]$b)
{
      return $a + $b
}
sum 4 5
```

Costanti

Create senza $

```
Set-Variable -name b -value 3.142 -option constant
```

Lette con $

```
$b
```

Creare oggetti

Per creare un'istanza di un oggetto COM

New-Object -comobject <ProgID>

```
$a = New-Object -comobject "wscript.network"
$a.username
```

Per creare un'istanza di un oggetto del framework .Net. Possono essere forniti i parametri obbligatori del costruttore, se richiesto.

New-Object –type <.Net Object>

```
$d = New-Object -Type System.DateTime 2006,12,25
```

Visualizzare dati a console

Nome della variabile

```
$a
```

oppure

```
Write-Host $a -foregroundcolor "green"
```

Leggere l'input inserito dall'utente

Utilizzate Read-Host per leggere l'input dell'utente

```
$a = Read-Host "Digita il tuo nome"
Write-Host "Ciao " $a
```

Passare argomenti tramite riga di comando

Usate degli spazi per passarli agli script

```
myscript.ps1 server1 benp
```

Utilizzate $args per recuperarli

```
$servername = $args[0]
$username = $args[1]
```

Varie

Andate a capo con una riga utilizzando `

```
Get-Process | Select-Object `
name, ID
```

Commentate con #

```
# testo non eseguito
```

Unite le righe di comando con ;

```
$a=1;$b=3;$c=9
```

Pipe di un comando verso un altro con |

```
Get-Service | Get-Member
```

Ciclo Do While

Ripete un blocco di comandi mentre una condizione è soddisfatta

```
$a=1
Do {$a; $a++}
While ($a –lt 10)
```

Ciclo Do Until

Ripete un blocco di comandi fino a quando una condizione è soddisfatta

```
$a=1
Do {$a; $a++}
Until ($a –gt 10)
```

Ciclo For

Ripete un blocco di comandi un certo numero di volte

```
For ($a=1; $a –le 10; $a++)
{$a}
```

ForEach – Iterazione di una collezione di oggetti

Itera attraverso una collezioni di oggetti

```
Foreach ($i in Get-ChildItem c:\windows)
{$i.name; $i.creationtime}
```

Blocco If

Esegue un blocco di codice se una condizione è soddisfatta

```
$a = "bianco"
if ($a -eq "rosso")
      {"Il colore è rosso"}
elseif ($a -eq "bianco")
      {"Il colore è bianco"}
else
      {"Colore sconosciuto"}
```

Blocco Switch

Altro metodo per eseguire un blocco di codice specifico, date determinate condizioni

```
$a = "rosso"
switch ($a)
{
      "rosso" {"Il colore è rosso"}
      "bianco"{"Il colore è bianco"}
      default{"Colore sconosciuto"}
}
```

Leggere da un file

Usate Get-Content per creare un array di linee. Poi recuperate gli elementi da quest'ultimo

```
$a = Get-Content "c:\servers.txt"
foreach ($i in $a)
{$i}
```

Scrivere un file (semplice)

Usate Out-File o > per file di testo semplici

```
$a = "Ciao a tutti"
$a | out-file test.txt
```

O usate > per salvare l'output dello script su file

```
.\test.ps1 > test.txt
```

Scrivere su di un file HTML

Usate ConvertTo-HTML e >

```
$a = Get-Process
$a | ConvertTo-HTML -property Name,Path,Company > test.htm
```

Scrivere su di un file CSV

Usate Export-Csv e Select-Object per filtrare il risultato

```
$a = Get-Process
$a| Select-Object Name,Path,Company | Export-Csv -path test.csv
```