Soós Tibor és Szerényi László

Microsoft PowerShell 1.0

rendszergazdáknak – elmélet és gyakorlat

```
PS C:\> Get-Book | Where-Object {$_.Title -match "PowerShell"}
```

Microsoft Magyarország
2008

Első kiadás

A szerzők a könyv írása során törekedtek arra, hogy a leírt tartalom a lehető legpontosabb és naprakész legyen. Ennek ellenére előfordulhatnak hibák, vagy bizonyos információk elavulttá válhattak.

A könyvben leírt programkódokat mindenki saját felelősségére alkalmazhatja. Javasoljuk, hogy ezeket ne éles környezetben próbálják ki. A felhasználásából fakadó esetleges károkért sem a szerzők, sem a kiadó nem vonható felelősségre.

”Aki másra néz, ha válaszra vár, magát sohasem érti meg.
Ám ha tükörképét kérdezi szüntelen, a világ lesz érthetetlenebb.
Aki nem tekint az égre, nem néz rá a napsugár.
Aki nem tekint a földre, az forrásra nem talál.
Aki előremenne, de mindig hátra néz, elvét minden útirányt.
Ám ha célján kívül semmit se lát, magára ölti a magányt.
Aki nem tekint az égre, éjjel vakon imbolyog.
Aki nem tekint a földre, némák annak a csillagok.”

/Laár András: Aki/

Tartalomjegyzék

1. Elmélet

1.1 Előszó

Ha egyetlen mondatban kellene megfogalmaznunk, hogy mi is a PowerShell – amiről ez a könyv szól -, akkor azt mondhatnánk, hogy a PowerShell egy teljesen objektumorientált, a .NET keretrendszerre épülő parancsfeldolgozó- és szkriptkörnyezet, ami gyökeresen új alapokra helyezi (és fogja helyezni) a Windows operációs rendszerekkel és kiszolgálóoldali alkalmazásokkal kapcsolatos felügyeleti feladataink elvégzését.

A Windows rendszerek parancssori felületének viszonylagos gyengesége történelmi okokra vezethető vissza. A Windows fejlesztésének fő vonala a kezdetektől fogva a grafikus felhasználói felület tökéletesítése volt, a parancssor sokáig csak mint kiegészítő eszköz szerepelt, így a lehetőségek többé-kevésbé korlátozottak voltak.

A Microsoft célja alapvetően az, hogy minden parancs vagy lista, ami az MMC konzolon vagy egy webes adminisztrációs felületről elérhető, illetve amit az adott szerveralkalmazás kiajánl magából automatizációhoz, vagy saját felügyeletéhez, az Windows PowerShell parancsok vagy providerek formájában is rendelkezésre álljon. Így a rendszergazda eldöntheti, hogy a grafikus, vagy a parancssoros interfészt használja feladatai elvégzéséhez. A grafikus felület nyilván sokak számára egyszerűbb, azonban a parancssor használatával messze több lehetőségünk adódik; például ugyanazokkal az eszközökkel barangolhatunk a registry, a fájlrendszer, a tanúsítványtár, vagy éppen az Exchange mailboxok között, és egy nagyon korrektül összerakott, típusos, .NET-alapú szkriptnyelvvel dolgozhatunk az ott talált adatokkal.

Ha egy műveletsort egyetlen gépen csak egyetlen egyszer kell végrehajtanunk, akkor a grafikus felület a logikus választás, ebben az esetben a parancssor használata, vagy a megfelelő szkript megírása csupán időigényes (bár szórakoztató☺) hobbinak tekinthető. Egészen más a helyzet azonban, ha az adott műveleteket minden nap el kellene végeznünk (vagy esetleg 300 gépen kell minden nap elvégeznünk). Akinek még így is kényelmesebb a grafikus felület, annak váljon egészségére az egerészés.

Az Exchange Server 2007-ben bár nagyon sok funkció elérhető a hozzá tartozó új MMC konzolról, azonban nem mindent tudunk beállítani onnan. Jó néhány, ritkábban használatos beállítás csak PowerShell parancsokkal végezhető és. És ha automatizálni szeretnénk az Exchange Server 2007 konfigurálását, akkor ezt mindenképpen a Windows PowerShell alapokra épülő Exchange Management Shellel tehetjük meg. Az Exchange 2007, – nagyon hasonlóan az SQL Server 2005-höz –már arra is képes, hogy ha a grafikus felületen módosítunk valamit, azt szkriptként is visszaadja nekünk. Mégpedig egy olyan PowerShell szkript képében, amit azonnal tudunk az új parancssorban, vagy automatizáláshoz használni.

Az pedig, hogy a PowerShell szinte teljesen .NET-alapú, még egy érdekességet mutat: az eddig szinte kizárólag programozóknak szóló osztálykönyvtárak végre elérhetőek egyszerűbb formában a rendszergazdák számára is, így a szerverszoftverek testre szabásához és automatizálásához kevesebb alkalommal lesz szükség a fejlesztő beavatkozására. Ezzel egy időben pedig a rendszergazdák is közelebb kerülhetnek a .NET alapú programozáshoz.

Aki látja a jövőt, az tudhatja, hogy a PowerShell mindent visz. Előbb-utóbb mindenkinek ugyanúgy meg kell tanulnia, mint annak idején a DOS-t – egyszerűen nincs nélküle élet a Windows-világban.

Szerényi László

A könyvet ketten írtuk, Soós Tibor és Szerényi László. Az egyes alfejezeteket vagy egyikünk, vagy másikunk, ez indokolja a könyvben alkalmazott egyes szám első személy alkalmazását. László leginkább a 1.2, 1.4, 1.10 és a gyakorlati rész egyes alfejezeteibe segített be. Annak ellenére, hogy ketten írtuk, bízom benne, hogy az olvasásban és a megértében ez nem okoz zavart, próbáltam azért szerves egésszé összegyúrni a szöveget.

A könyv írását először tanfolyami jegyzetként kezdtem el, majd annyi dolog tolult fel bennem PowerShell témában, hogy egyre csak hízott az anyag, így két hónap után megérlelődött bennem, hogy könyvvé kellene átalakítani. Ennek ellenére végig az lebegett a szemem előtt, hogy ez olyan könyv legyen, amely segítségével meg lehet tanulni a PowerShell használatát. Próbáltam azokat a részeket részletesebben leírni, amelyek anno nekem nehezebben voltak érthetőek.

Természetesen pusztán a könyv elolvasása még nem elegendő ahhoz, hogy profi PowerShell programozók legyünk, sok gyakorlásra is szükség van. A Microsoft Script Center weboldalán telente szoktak hirdetni „Winter Scripting Games” néven szkriptelési versenyt, javaslom a tisztelt olvasónak, hogy nézze meg milyen feladatok voltak az elmúlt években, próbálja meg megoldani azokat, és javaslom, hogy akár nevezzen is be a következő vetélkedőbe, nagyon sokat lehet ebből tanulni!

Ha valaki időt szeretne nyerni, akkor javaslok egy - általam tartott ☺ - tanfolyam elvégzését, amely ennek a könyvnek az anyagára épül. A négynapos tanfolyam során az elmélet elmagyarázása mellett lehetőség van gyakorlásra. A kiadott feladatokat megbeszéljük, megnézzük, hogy a különböző megoldási javaslatok közül vajon melyik tanfolyami résztvevő megoldása a legelegánsabb, legügyesebb. Ezt a fajta tanulást a könyv önmagában nem helyettesíti. Ezen a négy napon hozzávetőlegesen 2 hónap önálló ismeretelsajátításának idejét lehet megspórolni.

A PowerShell számomra szórakozás, szellemi játék is. Bízom benne, hogy a kedves olvasó számára is sikerül ezt az oldalát is megmutatni, és élvezettel próbálja ki mindenki a példaszkripteket és akár el is játszadozik velük, tovább is fejleszti azokat.

A PowerShell 2.0 már készülőben van. Terveim szerint a megjelenése után nem sokkal aktualizálom ezt a könyvet, így a 2. kiadás már az újdonságokat is tartalmazni fogja.

Szívesen fogadom a könyvvel kapcsolatos visszajelzéseket! Akár valami elgépelést, vagy nehezen érthető példát találnak, vagy javasolnának valamit, ami még illene egy ilyen jellegű könyvbe, kérem, írják meg! Címem: soost@IQJB.hu .

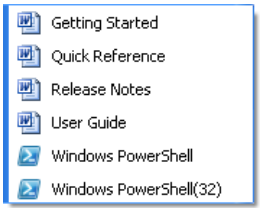
Soós Tibor

1.2 Kezdetek

1.2.1 A PowerShell telepítése

A PowerShell 1.0 telepíthető Windows XP SP2, Windows Server 2003 SP1 és ezeknél újabb operációs rendszerre. A telepítés előfeltétele a .NET Framework 2.0 megléte a gépen. Mind a PowerShell, mind a .NET Framework letölthető a Microsoft weboldaláról. A PowerShell telepítőkészleteihez legegyszerűbben a http://www.microsoft.com/powershell címen megjelenő „Download" szekcióból tudunk eljutni. Az indokolja a többes számot, hogy minden operációs rendszer típushoz és processzor-platformhoz külön telepítőkészlet van, így fontos, hogy a megfelelőt töltsük le és telepítsük.

A PowerShell-t a Start menü *Windows PowerShell 1.0* csoportjában található *Windows PowerShell* parancsikon segítségével, vagy közvetlenül a *powershell.exe* meghívásával indíthatjuk. A 64 bites Windows-okon két PowerShell ikonunk is lesz, egy 64 bites és egy 32 bites parancssori környezet indítására:

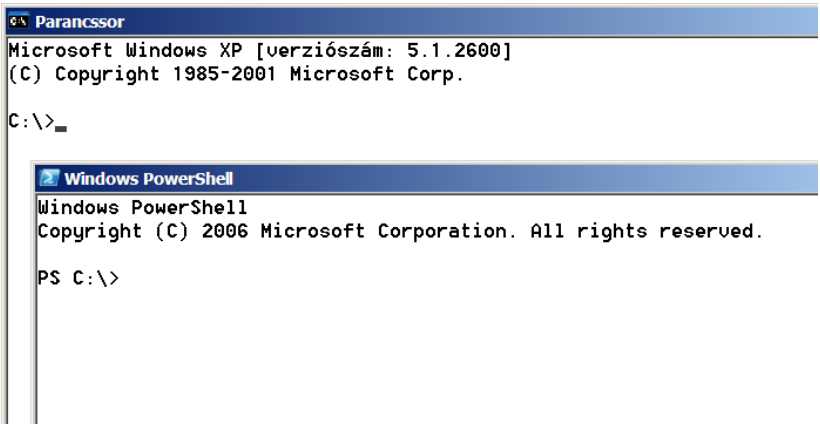


1. ábra PowerShell programcsoport ikonjai (64 bites)

A programcsoportban a PowerShell dokumentációjának ikonjai is megtalálhatók.

1.2.2 Indítsuk el a PowerShellt!

Indítás után a régi, megszokott Parancssorhoz (cmd.exe) hasonló felület jelenik meg. Amint az alábbi ábrán is látható, kísérteties a hasonlóság, nem lesz itt baj, gondosan tartsuk magunktól távol a dokumentációt és próbálkozzunk bátran.



2. ábra, A cmd.exe és a PowerShell

Kezdetnek próbáljuk ki tehát a megszokott parancsokat, vajon mit szól hozzájuk az új shell.

```
PS C:\> dir


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----        2007.06.10.     12:41            Config.Msi
d----        2007.06.13.     21:30            Documents and Settings
d----        2003.10.25.     18:59            Inetpub
d-r--        2007.06.04.     20:36            Program Files
d----        2006.09.13.     18:10            temp
d----        2007.05.31.     19:50            WINDOWS
-a---        2003.08.15.     19:35          0 AUTOEXEC.BAT
-a---        2003.08.15.     19:35          0 CONFIG.SYS
```

Amint látható, nem történt semmi meglepő, kaptunk egy mappa- és fájllistát, nagyjából a megszokott módon.

Eddig rendben, a nehezén túl is vagyunk, ez is csak egy parancssor. Próbálkozzunk valami nehezebbel, írjuk be mondjuk a következőt:

```
PS C:\> dir /s
Get-ChildItem : Cannot find path 'C:\s' because it does not exist.
At line:1 char:4
+ dir  <<<< /s
```

Hát ez már nem jött össze: szép piros hibaüzenet tájékoztat, hogy a `C:\s` mappa nem létezik. Ez így teljesen igaz is, de ki akart oda menni? (Egyébként teljesen korrekt a hibaüzenet, mivel ha mégis van ilyen mappa, akkor azt gond nélkül kilistázza, vagyis a /

karakter a gyökérmappát jelenti.) Mi történik itt? Úgy látszik, mégis segítséget kell kérnünk, de vajon hogyan? Mondjuk, legyen a következő:

```
PS C:\> help dir

NAME
    Get-ChildItem

SYNOPSIS
    Gets the items and child items in one or more specified locations.
...
```

Talán meglepő, de a `help` parancs működik. Talán még meglepőbb, de például a `man` parancs is működik! A válaszból viszont az derül ki, hogy a `dir` egyáltalán nem is létezik, a mappalistát valójában a `Get-ChildItem` nevű parancs produkálta az előbb. Kiderül az is, hogy hogyan kell őt az alkönyvtárakba leküldeni: `-recurse`, vagy egyszerűen: `-r` kapcsolóval.

A `Get-ChildItem` leírását elolvasva joggal kérdezhetjük, hogy miért változott meg az égvilágon minden, ha egyszer ugyanazt a feladatot hajtja végre a `Get-ChildItem`, mint a jó öreg `dir` parancs. A válasz pedig egyszerűen az, hogy nem ugyanazt végzi el, nem ugyanúgy, és nem ugyanazzal az eredménnyel! A `Get-ChildItem` például nemcsak mappákból, hanem számos más adathalmazból is képes listát produkálni, így a regisztrációs adatbázisból, a tanúsítványtárból, stb. is, kimenete pedig korántsem az a lista, amit a képernyőn láttunk.

Ha kiadjuk az alábbi parancsot, szomorúan láthatjuk, hogy `help` parancs sem létezik, az ő igazi neve `Get-Help`.

```
PS C:\> help help
```

Hogy tovább szaporítsuk a nem létező, de ennek ellenére jól működő parancsok számát, próbáljuk ki a `cd` parancsot, majd kérjük el az ő súgólapját is. Láthatjuk, hogy az igazi neve `Set-Location`, és természetesen nemcsak a fájlrendszerben, hanem a regisztrációs adatbázisban, tanúsítványtárban, stb. is remekül működik. Valójában a `cd`**,** és a többi nem létező, de működő parancs csupán egy alias[1], egy becenév, álnév a megfelelő PowerShell parancsra (lásd: `Get-Alias`), amelyek hivatalos neve cmdlet (ejtsd: kommandlet, magyarul parancsocska).

A következőkben felsorolunk a fenti parancsokkal kapcsolatos néhány egészen egyszerű példát, amelyek között részben a „régi” cmd parancsok megfelelőit, részben az új képességek demonstrációját találhatjuk.

- dir *.txt /s

```
PS C:\> Get-ChildItem -Include *.txt -Recurse
```

[1] A `help` és a `man` nem alias, hanem függvény, de ennek a mi szempontunkból most nincs jelentősége.

- A HKLM/Software ág kilistázása a registryből.

```
PS C:\> Get-ChildItem registry::HKLM/Software
```

- dir *.exe

```
PS C:\> Get-ChildItem * -i *.exe
```

- cd q:

```
PS C:\> Set-Location q:
```

- Jelenlegi pozíció elmentése[2].

```
PS C:\> Push-Location
```

- Átállás a HKEY_CURRENT_USER ágra a registryben.

```
PS C:\> Set-Location HKCU:
```

- Visszaugrás egy korábbi elmentett pozícióra.

```
PS C:\> Pop-Location
```

- Átállás a tanúsítványtár „meghajtóra”.

```
PS C:\> Set-Location Cert:
```

Azért volt néhány elég furcsa külsejű parancs, igaz? Tanúsítványtár meghajtó!? Mi lesz még itt?

Kísérletezzünk tovább! Ha már ilyen szépen tudjuk használni a nem létező parancsokat, próbáljunk ki valami olyat, ami tényleg biztosan nem létezhet:

```
PS C:\> 1
1
```

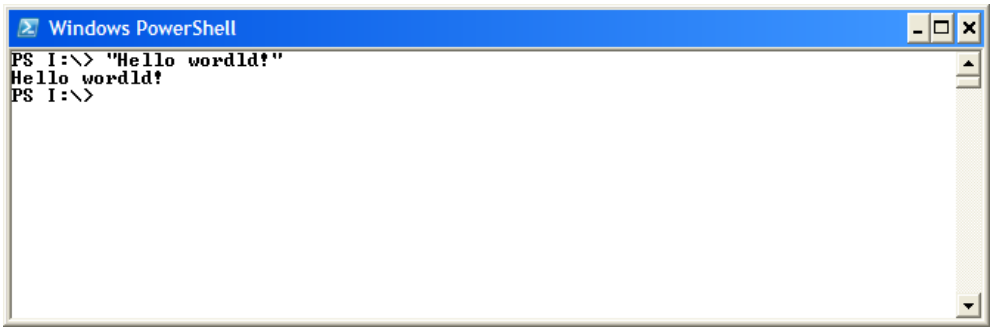
Semmi hibaüzenet, a PowerShell egyszerűen visszaírta a számot a konzolra.

A PowerShell ablakot bezárni az `Exit` paranccsal lehet, de ez nem PowerShell parancs, nem is alias, a help sem ad semmi támpontot.

[2] Ahogyan a név sugallja, több pozíció is elmenthető egymás „tetejére”, vagyis egy verembe. A Pop-Location mindig a legfelül lévőre tér vissza, majd eldobja azt.

1.2.3 Hello World!

Az előzőekben a PowerShell saját parancsaival és DOS-os álnevekkel kísérleteztünk, most kezd kezdjünk el alkotni, saját programot írni! Bjarne Stroustrup óta, azt hiszem, minden programozással foglalkozó könyv azzal kezdődik, hogy olyan programot írunk, amelyik kiírja a képernyőre, hogy „Hello world!”. Nézzük meg, hogy ez hogyan megy PowerShellben:

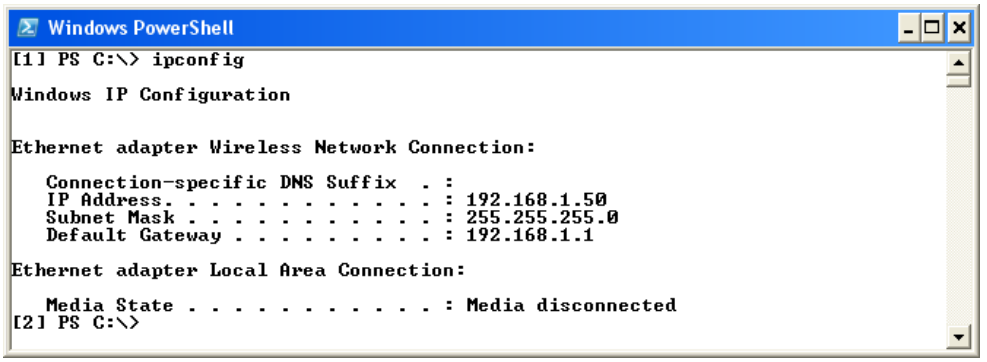


3. ábra Első PowerShell parancsom

Nagyon egyszerű, idézőjelek közt beírjuk a szöveget, Enter, és már ki is írta. Ennél egyszerűbben nem is lehetne ezt!

1.2.4 DOS parancsok végrehajtása

Láttuk, hogy a `Dir` parancs lefutott, de könnyű neki, hiszen a `Dir` egy PowerShell parancs álnéven. De vajon a többi DOS paranccsal mi történik? Azokkal is nyugodtan próbálkozzunk! Például próbáljuk ki az `IPConfig` parancsot:



4. ábra IPConfig a PowerShell ablakban

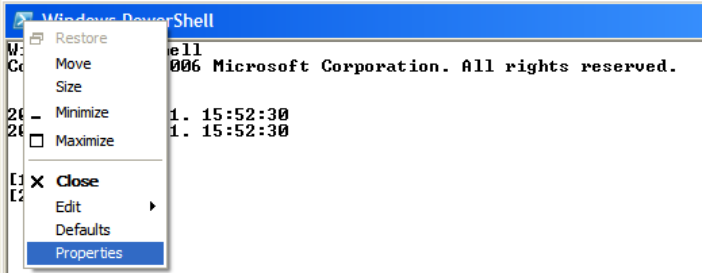
Ez is lefutott, az `ipconfig` ugyanúgy működik, mint a DOS ablakban. Ehhez hasonlóan a többi DOS parancsot is nyugodtan kipróbálhatjuk, azaz akár teljes egészében áttérhetünk a PowerShell környezet használatára, akár a nem PowerShell parancsok futtatásával is.

1.2.5 Gyorsbillentyűk, beállítások

Nagyban meggyorsítja munkánkat, ha ismerjük azokat a gyorsbillentyűket, amelyek elérhetőek a PowerShell ablakban:

TAB	Automatikus kiegészítés, további TAB további találatokat ad
Shift + TAB	Visszalép az előző találatra
F1	1 karakter a történet-tárból
F2	Beírt karakterig másol
F3 vagy ↑	Előző parancs a történet-tárból
F4	Kurzortól az adott karakterig töröl
F5 vagy ↓	Lépked vissza a történet-tárból
F6	hatástalan
F7	Történettárat megjeleníti
F8	Parancstörténet, de csak a már begépelt szövegre illeszkedőeket
F9	Bekéri a történettár adott elemének számát és futtatja
Jobbkattintás	Beillesztés
Egérrel kijelöl + Enter	Másolás
→	Karaktert lépked, üres sorban a parancstörténet-tárból hív elő karaktereket
←	Karaktert lépked
ESC	Törli az aktuális parancssort

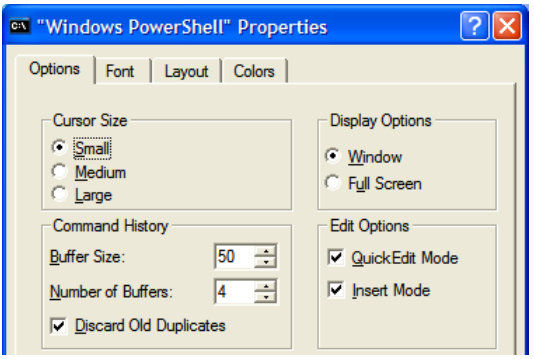
Tovább gyorsítja a munkánkat, ha a PowerShell ablak néhány beállítására is odafigyelünk. Ha jobb egérgombbal kattintunk az ablak bal felső sarkában levő PowerShell ikonra ( ), akkor a következő menü jelenik meg:



5. ábra A PowerShell ablak menüje

Itt minket a „*Properties*” menüpont érdekel, kattintsunk rá. A megjelenő párbeszédpanelben van néhány fontos, munkánkat segítő beállítási lehetőség:

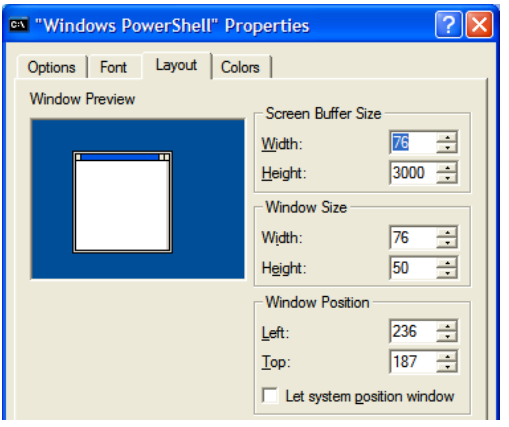
Az „*Options*” fülön a „*QuickEdit Mode*” fontos, hogy be legyen kapcsolva. Ennek birtokában tudjuk az egérrel kijelölni a konzolon megjelenő szövegeket és Enter leütésével a vágólapra helyezni. Szintén ez kell ahhoz, hogy a vágólapra valahonnan kimásolt szöveget a jobb egérgombbal beilleszthessük a kurzor pozíciójától kezdődően.



6. ábra Options fül

A „*Font*” fülön állíthatunk be az igényeinknek és szemünknek esetleg jobban megfelelő betűtípust és méretet.

A „*Layout*” fülön lehet optimálisabb ablakméretet beállítani. Ez főleg az ablak szélessége szempontjából fontos, mert majd látni fogjuk, hogy a PowerShell parancsok kimenete nagyon gyakran táblázat jellegű, és annak jobb, ha minél szélesebb ablakban tudjuk megjeleníteni.



7. ábra Az ablak méretének beállítása

Jelen könyvbe másolt konzoltartalmak miatt én az ablakméretemet levettem 76 karakterre, mert ez pont még megfelelő fontmérettel beilleszthető a könyv lapjaira, de igazi

munkában, amikor jó nagy felbontású monitort használunk, akkor ezt érdemes akár 150 karakterre megnövelni.

Szintén praktikus, ha a látható ablak szélessége és a mögötte levő „*Screen Buffer*" szélessége egyforma, mert akkor nem lesz vízszintes gördítő sávunk, ami jó, ha nincs. Ezzel szemben a függőleges puffer-méret lehet jó nagy, hogy hosszabb listák is gond nélkül teljes terjedelmükkel kiférjenek, és függőleges gördítéssel megtekinthetőek legyenek.

Az utolsó fül a színek beállítására való. Alaphelyzetben mélykék alapon fehér betűkkel nyílik meg a PowerShell ablak. Ha valaki könyvet ír, és a képernyőképek nyomtatásra kerülnek, akkor érdemes festékkímélés céljából inkább fehér alapon fekete betűket használni.

Ha testre szabtuk az ablakot, akkor a párbeszédpanel bezáráskor rákérdez a felület, hogy ezeket a beállítások csak az adott ablakra vonatkozzanak, vagy legközelebb is viszont akarjuk látni a változtatásainkat, ha újra megnyitjuk a Start menüből a PowerShell ablakot. Érdemes ez utóbbi opcióval elmenteni változtatásainkat kipróbálás után.

1.2.6 A TAB-billentyű

Az egyik legtöbbet használt gyorsbillentyű a TAB-billentyű, amit érdemes kicsit részletesebben is tárgyalni, mivel ez rengeteg fölösleges gépeléstől mentheti meg a lustább (ez határozottan pozitív tulajdonság) szkriptelőket. A PowerShell TAB-billentyűje annyira sokrétűen használható, hogy akár olyan információforrásként is tekinthetünk rá, ami gyorsan, könnyen elérhető és mindig kéznél van.

A TAB használata nem igazi újdonság, nyilván sokan használták a *cmd.exe*-be épített változatát, de a PowerShell-ben számos új képességgel is találkozhatunk. Mi a teendő például, ha át akarunk váltani a *C:\Documents and Settings* mappára? Természetesen begépelhetjük a teljes parancsot:

```
PS C:\> Set-Location "c:\documents and settings"
```

Nem is olyan sok ez. Akinek mégis, az csak ennyit írjon:

```
PS C:\> Set-Location c:\d<tab>
```

Ha a c: meghajtó gyökerében nincs más 'd' betűvel kezdődő mappa, akkor készen is vagyunk, a shell kiegészítette a parancsot, még az idézőjeleket is begépelte helyettünk. Ha több mappa is kezdődik 'd'-vel akkor sincs nagy baj, a TAB-billentyű szorgalmas nyomkodásával előbb-utóbb eljutunk a megfelelő eredményig. Még ez is túl sok? Próbálkozzunk tovább! A PowerShell nemcsak a mappa és fájlneveket, hanem a cmdletek nevét is kitalálja helyettünk:

```
PS C:\> Set-L<tab> c:\d<tab>
```

Mi történik vajon, ha csak ezt gépeljük be:

```
PS C:\> Set-<tab>
```

Bizony, a TAB-billentyű nyomkodásával végiglépkedünk az összes `Set-` parancson, így akkor is könnyen kiválaszthatjuk a megfelelőt, ha a pontos név éppen nem jut eszünkbe. A Shift+TAB-bal vissza tudunk lépni az előző találatra, ha túl gyorsan nyomkodva esetleg tovább léptünk, mint kellett volna.

Mi a helyzet a cmdletek paramétereivel? Gépeljük be a következőt:

```
PS C:\> Get-ChildItem -<tab>
```

Talán már nem is meglepő, hogy a lehetséges paraméterek listáján is végiglépkedhetünk, hogy a megfelelőt könnyebben kiválaszthassuk közülük.

A paraméterek megadásával kapcsolatban itt említünk meg egy másik lehetőséget is (bár nem kell hozzá TAB). Egyszerűen annyi a teendőnk, hogy egyáltalán nem adunk meg paramétert, mivel ebben az esetben a parancs indítása előtt a PowerShell szép sorban bekéri a kötelező paramétereket:

```
PS C:\> New-Alias

Supply values for the following parameters:
Name: ga
Value: Get-Alias
```

Hamarosan részletesen megismerkedünk a PowerShell változók rejtelmeivel és az objektumok használatával, most csak egy kis előzetes következik a TAB-billentyű kapcsán. Ha a PowerShell munkamenetben saját változókat definiálunk, a TAB ezek nevének kiegészítésére is képes, amint az alábbi példán látható:

```
PS C:\> $ezegynagyonhosszunevuvaltozo = 1
PS C:\> $ez<tab>
```

Ügyes! És mi a helyzet a .NET vagy COM-objektumok tulajdonságaival és metódusaival? A TAB-billentyű ebben az esetben is készségesen segít:

```
PS C:\> $s = "helló"
PS C:\> $s.<tab>
PS C:\> $excel = new-object -comobject excel.application
PS C:\> $excel.<tab>
```

Aki még arra is kíváncsi, hogy ezt a sok okosságot mi végzi el helyettünk, adja ki a következő két parancsot (természetesen a TAB-billentyű sűrű nyomkodásával):

```
PS C:\> Set-Location function:
PS Function:\> (get-item tabexpansion).definition
```

A képernyőn a PowerShell beépített `TabExpansion` nevű függvényének kódja jelenik meg; ez fut le a TAB-billentyű minden egyes leütésekor. A kód most még talán egy

kicsit ijesztőnek tűnhet, de hamarosan mindenki folyékonyan fog olvasni PowerShell-ül. A kód azonban nemcsak olvasgatásra jó, meg is változtathatjuk azt, sőt akár egy teljesen új, saját függvényt is írhatunk a „gyári" darab helyett (a függvényekről később természetesen még lesz szó).

1.2.7 Promptok, beviteli sor

Látható, hogy a PowerShell is a parancssorok elején, amikor tőlünk vár valamilyen adatbevitelt, akkor egy kis jelző karaktersorozatot, u.n. promptot ír ki. Ez alaphelyzetben így néz ki:

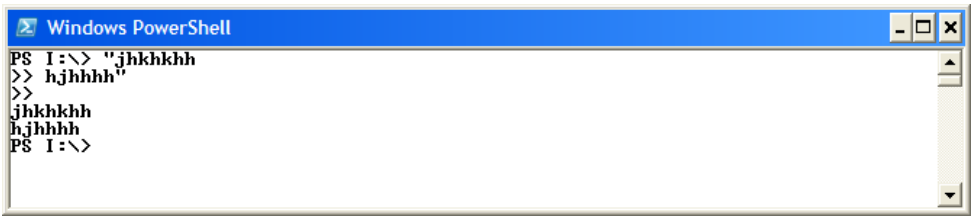
```
PS alaphelyzet szerinti könytár elérési útja\>
```

Az „alaphelyzet szerinti könyvtár" nagy valószínűséggel a `My Documents` könyvtárra mutat. Ez a prompt testre szabható. Itt ebben a könyvben sok helyen ilyen módosított prompt látható a példáimnál:

```
[1] PS I:\>
```

A PS elé biggyesztettem egy növekvő számot szögletes zárójelek között, hogy jobban tudjak hivatkozni az egyes sorokra a magyarázó szövegben.

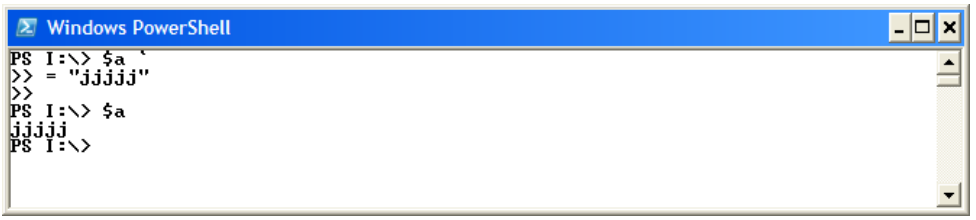
A prompt többszintű, viszont csak 1 szint mélységig jelzi a >> jellel az alapszinttől eltérő mélységet, ennél mélyebb szintekre nincs külön speciális prompt:



8. ábra Többszintű prompt

Ha „érzi", hogy kell még folytatódnia a sornak, >> lesz a prompt. Ha már ki akarunk szállni a mélyebb szintű promptból, akkor azt egy üres sorban leütött Enterrel tehetjük meg.

Ha egy hosszabb sort inkább külön sorba szeretnénk tenni, a „visszafeleaposztróf" (` - AltGr 7) használatos:



9. ábra Sortörés Escape (`) karakter után

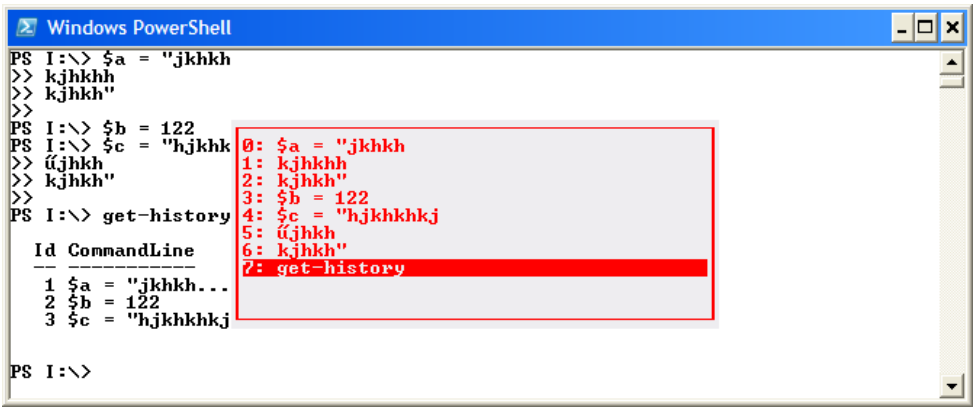
Ez az u.n. *Escape* karakter, ami hatástalanítja a következő speciális karakter (itt pl. az Enter) hatását. Amúgy nyugodtan folytatódhat a parancssorunk a következő sorban, ha túl hosszú.

Ha több utasítást szeretnénk egy sorba beírni, akkor a „;” karakterrel tudjuk azokat elválasztani.

```
[1] PS I:\>"Első utasítás"; 1+3; "Vége"
Első utasítás
4
Vége
```

1.2.8 Parancstörténet

A PowerShell ablak megjegyzi a korábban begépelt parancssorainkat. Ezt a parancstörténet-tárat az F7 billentyűvel tudjuk megjeleníteni. Vigyázat! Kétfajta történet-sorszámozás van:

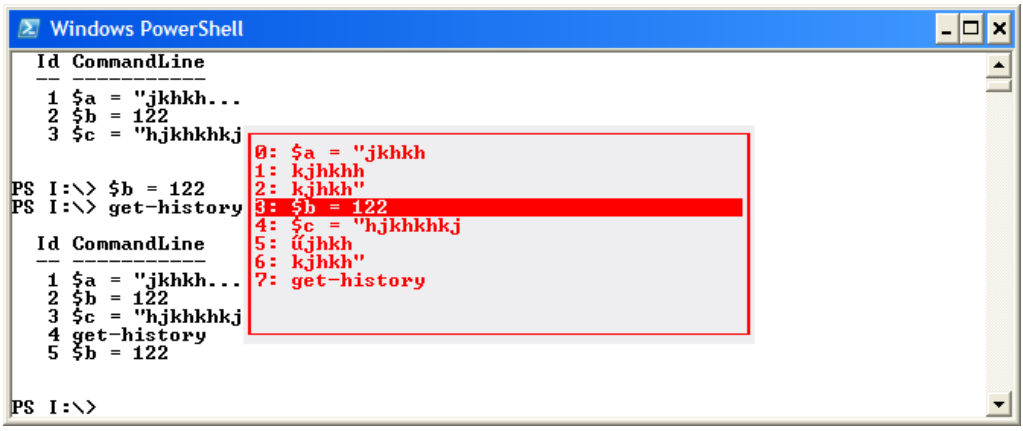


10. ábra A PowerShell parancstörténet-tár

Az F7-re megjelenő történettárban minden tényleges sor (alpromptok is) külön vannak nyilvántartva. Van azonban egy PowerShell cmdletünk is, a `get-history`, amiben

főpromptonként kapunk új sorszámot. Ráadásul a történettárból előhívott parancsok esetében a sorszámok másként viselkednek a kétfajta előhívás során.

Az alábbi ábra azt demonstrálja, hogy előhívtam a 3. parancsot a történettárból. Az F7-es történettár ezek után ezt a parancsot nem jeleníti meg újra, azt mutatja, hogy ezt a 3. parancsot egyszer hajtottam végre. Ezzel szemben a `get-history` kiírta, mint 5. végrehajtást:

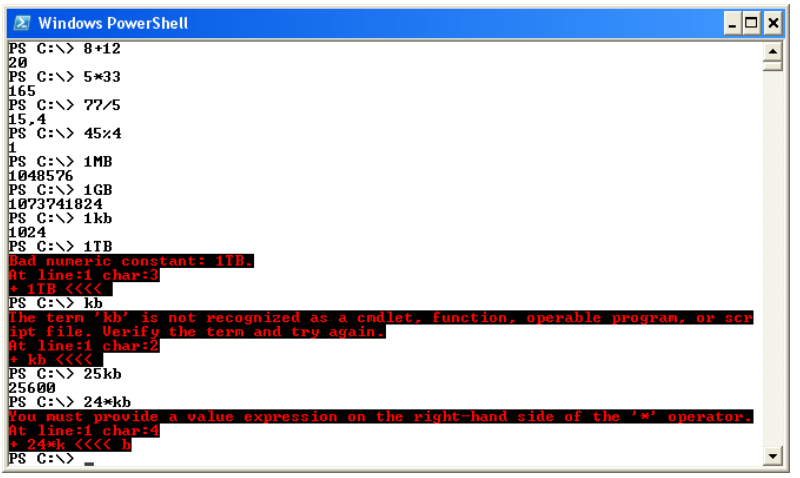


11. ábra A megismételt parancsok nem jelennek meg a történettárban

A parancstörténetből Enterrel lehet végrehajtatni a kijelölt korábbi parancsot, de gyakran arra van szükségünk, hogy egy régebbi parancsot kicsit módosítva hajtsuk újra végre. Ekkor ne Entert, hanem a jobbnyílt nyomjuk meg.

1.2.9 A PowerShell, mint számológép

A PowerShell alaphelyzetben számológépként is működik:



12. ábra Számolási műveletek a PowerShell ablakban

Látszik a fenti ábrán, hogy támogatja az alapműveleteket és a leggyakoribb számítástechnikában előforduló mértékegységeket: `kb`, `mb`, `gb`. Ezeket számmal egybeírva kell használni, különben hibajelzést kapunk. A % a maradékos osztást végzi.

Sajnos a terabájtot nem támogatja, és ahogy a fenti hibajelzésekből is látható, a mértékegységek önállóan nem állhatnak, mindig számmal kell bevezetni, még akkor is, ha egyről van szó.

1.2.10 A konzol törlése

Ha már jól teleírtuk a PowerShell ablakot, és szeretnénk, ha üres lenne újra, akkor futtassuk a `Clear-Host` cmdletet.

Ennek van két gyakran használt álneve is: `cls`, `clear`.

1.2.11 Kis-nagybetű

A PowerShell a legtöbb esetben nem különbözteti meg a kis- és nagybetűket. A parancsokat, kifejezéseket akár kis, akár nagybetűkkel beírhatjuk. De még az általunk létrehozott változónevek tekintetében is figyelmen kívül hagyja ezt:

```
[2] PS I:\>$kis = 1
[3] PS I:\>$KIS
1
[4] PS I:\>get-command get-help
```

```
CommandType     Name                                Definition
-----------     ----                                ----------
Cmdlet          Get-Help                            Get-Help [[-Name] <String>...


[5] PS I:\>GET-COMMAND get-HELP

CommandType     Name                                Definition
-----------     ----                                ----------
Cmdlet          Get-Help                            Get-Help [[-Name] <String>...
```

Az összehasonlításoknál sincs alaphelyzetben szerepe a betű méretének, természetesen itt majd ezt az alapműködést felülbírálhatjuk, hogy a kis-nagybetűket tekintse különbözőnek:

```
[6] PS I:\>"alaphelyzet" -eq "ALAPHELYZET"
True
[7] PS I:\>"kis-nagybetű érzékeny" -ceq "kis-NAGYbetű ÉrZéKeNy"
False
```

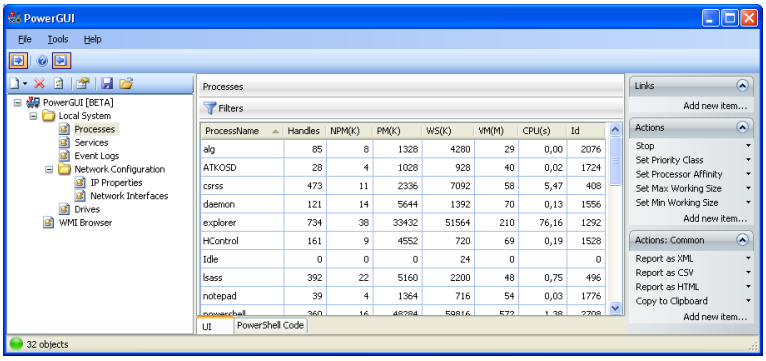
A fenti példában az alaphelyzet szerinti egyenlőséget vizsgáló operátor nem kis-nagybetű érzékeny, ha érzékeny változatot szeretnénk használni, akkor az a „c” előtaggal (azaz a „case-sensitive”) külön jelezni kell. Az összehasonlítás műveletéről az *1.6 Operátorok* fejezetben lehet bővebben olvasni.

1.3 Segédprogramok

Ha kezdők vagyunk PowerShellben, akkor mindenképpen érdemes felszerelkeznünk néhány hasznos segédprogrammal, amelyek megkönnyítik a munkánkat és a tanulási folyamatot. Ebben a fejezetben néhány ilyen programot mutatok be, amelyek többsége ingyenesen letölthető, használható, vagy legalábbis van ingyenes próbaváltozata.

1.3.1 PowerGUI, PowerGUI Script Editor

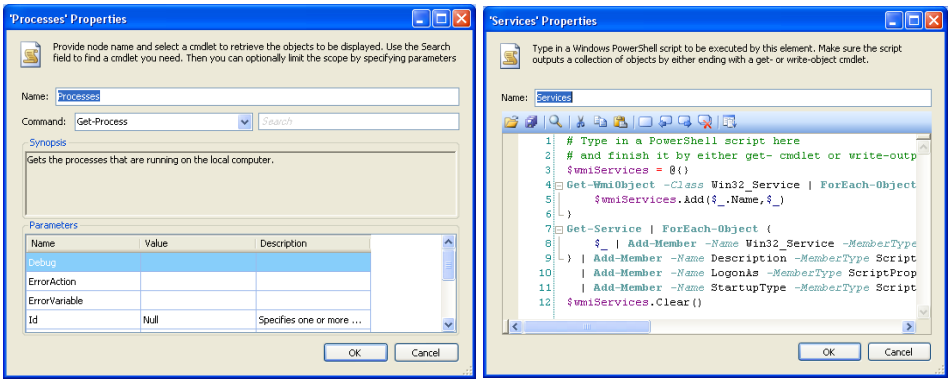
A PowerShellhez az egyik legpraktikusabb segédprogram a Quest Software PowerGUI csomagja. Ez két fő programból áll. Az első maga a PowerGUI, ami jó néhány előre elkészített PowerShell parancsot tartalmaz egy fastruktúrába fűzve:



13. ábra A PowerGUI felülete

A fastruktúra egyébként bővíthető, például Active Directory elemekkel, attól függően, hogy milyen bővítményeket telepítettünk a gépünkre, és mi magunk is készíthetünk hozzá faelemeket.

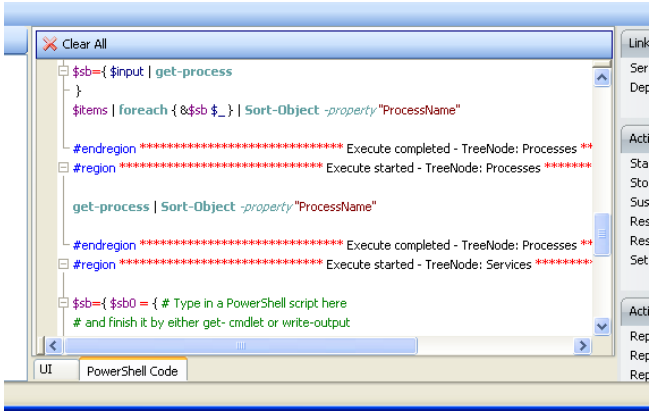
A programban az ikonnal jelzett faelemek mögött PowerShell szkriptek vannak:



14. ábra A PowerGUI faelemek mögötti PowerShell parancsok

Ez lehet egy egyszerű PowerShell parancs, ún. cmdlet, mint a Processes-nél, vagy bonyolultabb szkript, mint a Services-nél.

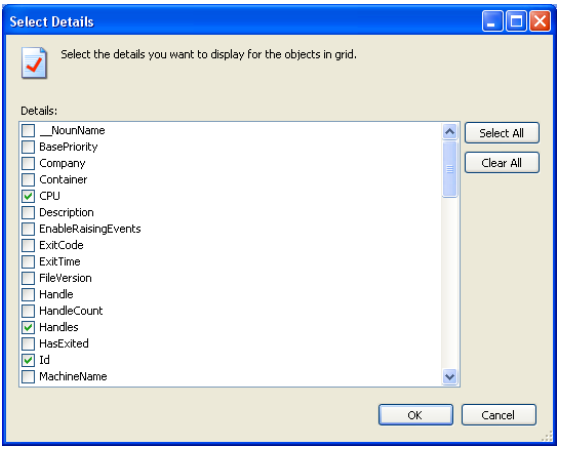
Az ablak alján a „*PowerShell Code*” fülre kattintva meg lehet figyelni, hogy pontosan mit hajt végre a PowerGUI amikor kattintgatunk a felületén:



15. ábra A PowerGUI által végrehajtott parancsok

Például a fenti képen, középtájon látszik, hogy amikor a processzek listáján a „*ProcessName*” oszlopra kattintottam, akkor erre egy `get-process | Sort-Object -property „ProcessName”` parancs hajtódott végre. Kezdőként sokat lehet tanulni ebből.

Illetve az oszlopfejlécen jobb egérgombbal kattintva lehet kérni egy ilyen listát:

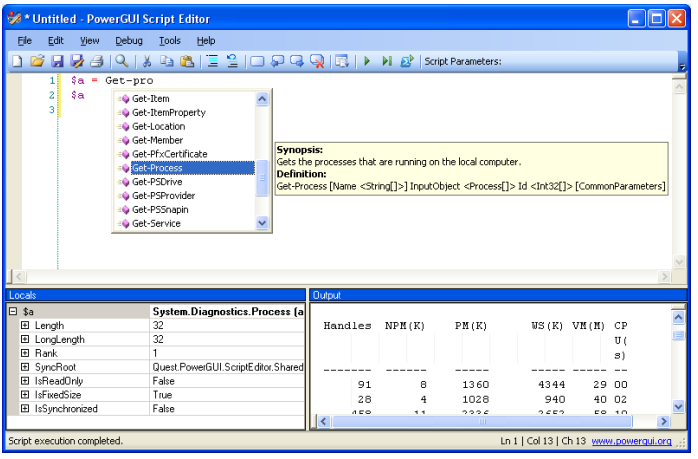


16. ábra Oszlopok, azaz tulajdonságok kiválasztása

Ez mutatja, hogy az adott parancs kimeneteként kapott objektumtípusnak milyen tulajdonságai, „property”-jei vannak.

A másik eszköz a PowerGUI csomagban a Script Editor. Ez nagyon nagy szolgálatot tesz a PowerShell-t tanulóknak, hiszen ahogy gépeljük a parancsokat rögtön szintaxisellenőrzést végez, különböző színnel jelöli a különböző fajtájú szövegelemeket (parancs, kifejezés, változó, sztring, stb.)

Gépelés közben a lehetséges parancsokat egy listában azonnal megjeleníti, a parancsok argumentumlistáját kis felugró ablakban kiírja. F1-re az adott parancs helpjét is megjeleníti.



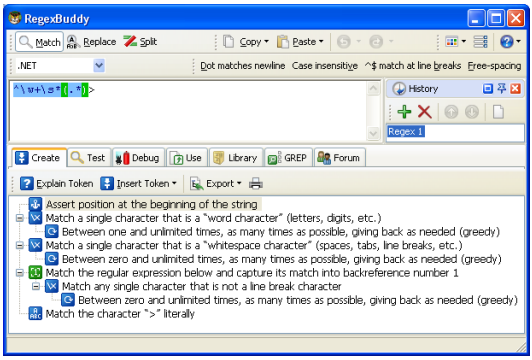
17. ábra Help és parancskiválasztó a PowerGUI Script Editorban

Az ablak felső részében szerkeszthetjük a kódot, bal alsó részben láthatjuk változóinkat és azok értékét, a jobb alsó részben a futtatott kód kimenetét olvashatjuk.

1.3.2 RegexBuddy

Később fogunk foglalkozni a reguláris kifejezésekkel, amelyek a sztringek összehasonlításában, vizsgálatában segítenek. Pl. meg lehet ilyen kifejezések segítségével állapítani, hogy egy szöveg vajon e-mailcímet tartalmaz-e, vagy pl. telefonszámot. Regex kifejezésekkel ki lehet nyerni szövegekből adott formátumú, adott mintára illeszkedő karaktersorozatokat.

Regex kifejezéseket írni nem egyszerű, ebben segít a *RegexBuddy*.



18. ábra Barátunk a RegexBuddy

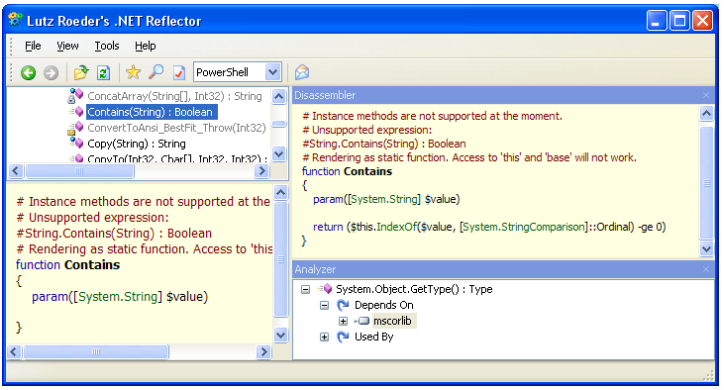
Az ablak felső részén látható, hogy kiválaszthatjuk a regex kifejezésünk „tájszólását", ami a PowerShell esetében .NET. Ez alatt van a szerkesztő felület, legalul meg egy fülrendszeren sok minden. A *Create* fül értelmezését adja a regex kifejezésemnek, és regex elemeket, u.n. tokeneket is hozzá tudok fűzni. Ami még érdekes számunkra a *Test* fül, ahol minta sztringeket tudok begépelni és ellenőrizni tudom, hogy a regex mintám illeszkedik-e arra, amire kellene, illetve nem illeszkedik-e arra, amire nem kell.

Hasznos segítséget nyújt a *Library* fül, ahol sok probléma megoldásához vannak előre elkészített regex kifejezések, amiket segítségül tudunk hívni és testre tudjuk szabni a saját igényeinknek megfelelően.

1.3.3 Reflector

A PowerShell a .NET Frameworkre épül, a keretrendszer osztályaival, objektumaival dolgozik, így fontos tudni, hogy a keretrendszerben mi található. A Visual Studio természetesen az Object Browser-ével segít a .NET osztályok felderítésében, de a PowerShell felhasználók zöme nem fejlesztő, nincs szükségük általában a Visual Studiora, és csak az Object Browser miatt telepíteni azt elég nagy luxus.

Az interneten Lutz Röder jóvoltából elérhető egy Reflector nevű kis ingyenes program (http://www.aisto.com/roeder/dotnet/), amellyel helyettesíteni lehet az Object Browsert. Ráadásul ez tartalmaz egy PowerShell add-in-t, amivel PowerShell szintaxisra lehet lefordítani a .NET-es objektumokat.

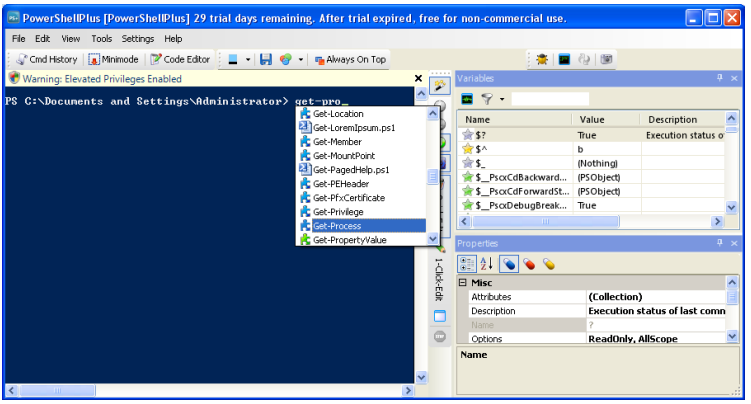


19. ábra Rendszergazdák Object Browsere: .NET Reflector

Ha találtunk olyan .NET osztályt, vagy annak valamilyen tagját, ami érdekel minket, akkor Ctrl+M billentyűkombinációval közvetlenül elő tudjuk hívni az MSDN website megfelelő lapját, ahol részletes leírást kapunk a kiválasztott elemünkről.

1.3.4 PSPlus

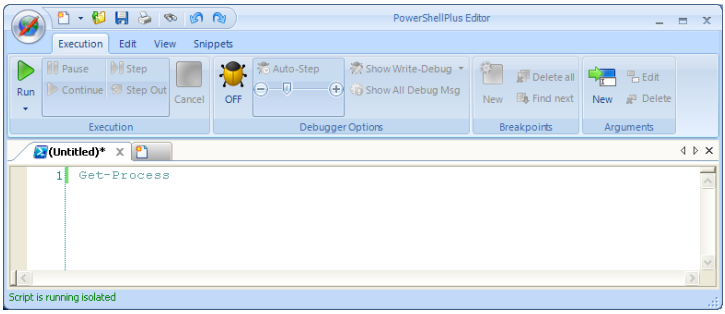
A PowerShell Plus hasonló célokat szolgál, mint a PowerGUI Script Editora, csak jóval összetettebb, több szolgáltatást nyújt.



20. ábra PowerShell Plus felülete

Ez az eszköz integrál magában az eredeti PowerShell konzolablakhoz nagyon hasonló ablakrészt, láthatjuk a változókat és egyéb paramétereket.

Külön szkriptszerkesztője is van:

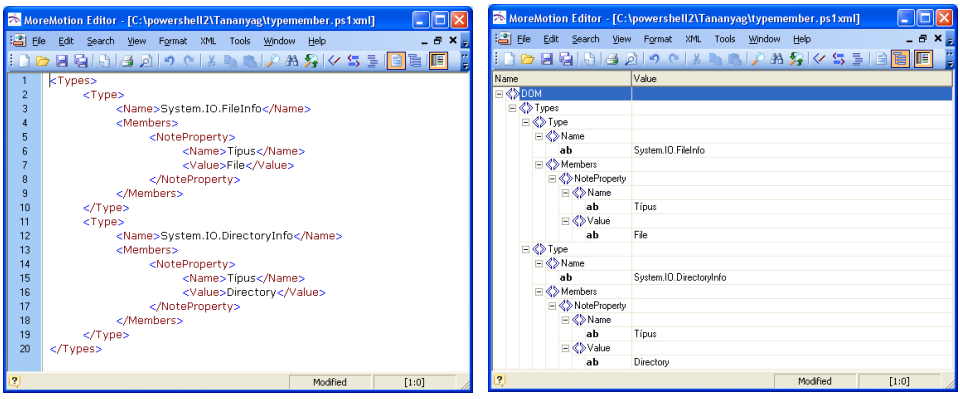


21. ábra PowerShell Plus Script Editora

Ez az Office 2007 stílusú szerkesztőfelület hasonló szolgáltatásokat nyújt, mint a PowerGUI Script Editora.

1.3.5 MoreMotion Editor

A PowerShellben is előfordul az XML adatformátum, így érdemes egy Notepad-nél esetleg okosabb XML szerkesztővel felszerelkezni. Én egy teljesen ingyenes és telepítést nem igénylő programot találtam erre a célra, a MoreMotion Editort:



22. ábra MoreMotion Editor: XML szerkesztő

A fenti képeken látható, hogy két nézete van, az első a forráskód szintű nézet, a második a fastruktúra-nézet, amelyben az XML tag-ek hierarchiáját és adattartalmát sokkal jobban át lehet tekinteni. A program rendelkezik szintaxisellenőrzési funkcióval, automatikus formázási lehetőségekkel is.

1.4 Alapfogalmak

Mi is a PowerShell? Kettős szerepe van: egyrészt a PowerShell egy parancssori környezet, azaz egy olyan program, ahol a felhasználó kapcsolatba léphet a számítógépes rendszerrel (direkt nem korlátozva le az operációs rendszerre!). Ebből a szempontból sok olyan szolgáltatást kell nyújtania, ami a gyors, kényelmes adatbevitelt, azaz a gépelést szolgálja. Másrészt a PowerShell egy programnyelv is, mint ilyennek rendelkeznie kell azokkal a programíráshoz szükséges elemekkel, mint például a ciklus, változók, függvények, adatszerkezetek.

A PowerShell mindkét szempontot próbálja a lehető legjobban kielégíteni, támaszkodik más, elsősorban a Unix és Linux különböző shelljeire és onnan átveszi a legjobb megoldásokat. De természetesen ez egy Windowsos shell, tehát nem próbál kompatibilis lenni a Unixos shellekkel, hanem inkább a Windows adottságaihoz és lehetőségeihez alkalmazkodik, ezen belül is a .NET Framework a legfontosabb alapja.

Ellentétben a DOS shellel, a PowerShell önmagában is nagyon sok parancsot tartalmaz. Például DOS shellben a fájlok másolását a copy.com végzi, maga a parancssori környezet nem tud fájlt másolni. Ezzel szemben a PowerShellnek van egy `copy-item` parancsa, u.n. cmdlet-je, amellyel lehet fájlt (de nem csak fájlt!) másolni.

1.4.1 Architektúra

A PowerShell tehát a .NET Frameworkre épül, az 1.0-ás PowerShellhez a 2.0-ás .NET Framework szükséges. Ez a ráépülés számos előnnyel jár, főleg ha azt is tudjuk, hogy a PowerShell objektumokkal dolgozik. Pl. a `dir` parancsot futtatva nem egyszerűen egy karaktersorozatot kapunk vissza válaszként, hanem a fájlok, alkönyvtárak objektumainak halmazát, u.n. collection-t, gyűjteményt.

Tehát két megjegyzendő fogalom: OBJEKTUM, COLLECTION!

Mi az, hogy objektum? Olyan képződmény, amely tulajdonságokkal (property) és meghívható metódusokkal (method) rendelkezik.

A hagyományos shellek (command.com, cmd.exe, bash, stb.) mindegyikét karakterlánc-orientáltnak nevezhetjük; a parancsok bemenete és kimenete (néhány speciális esetet nem számítva) karakterláncokból áll. Például a cmd.exe esetén a `dir` parancs kimenete valóban és pontosan az a fájl- és mappalistát leíró karaktersorozat, amely a képernyőn megjelenik, se több, se kevesebb. Ez bizonyos szempontból jó, hiszen így azt látjuk, amit kapunk. Másrészről viszont csak azt kapjuk, amit látunk, ez pedig sajnos nem túl sok. A PowerShell adatkezelése azonban alapjaiban különbözik ettől; az egyes parancsok bemenete és kimenete is .NET osztályokon alapuló objektumokból (objektumreferenciákból) áll, vagyis a kiírtakon kívül tartalmazza a .NET objektum számtalan másik tulajdonságát is. Amint láthattuk, a dir parancs kimenete itt is a képernyőre kiírt fájl és mappalistának tűnik, de ez csak egyszerű érzéki csalódás, a kimenet valójában egy `System.IO.DirectoryInfo` és `System.IO.FileInfo` osztályú objektumokból álló gyűjtemény. Mivel azonban az objektumok nem alkalmasak emberi fogyasztásra,

képernyőn való megjelenítésük természetesen szöveges formában, legfontosabb (illetve kiválasztott) jellemzőik felsorolásával történik.

A fentiekből nyilvánvalóan következik az objektumorientált megközelítés két fontos előnye; egyrészt a parancsok kimenete rengeteg információt tartalmaz, amelyeket szükség esetén felhasználhatunk, másrészt nincs szükség szövegfeldolgozó parancsok és eszközök használatára, ha a kimenetként kapott adathalmazból csak egy meghatározott információdarabra van szükségünk, mivel minden egyes adatmező egyszerűen a nevére való hivatkozással, önállóan is lekérdezhető. Képzeljük el például, hogy a `dir` parancs által produkált fenti listából szövegfeldolgozó parancsokkal kell kiválasztanunk azokat a fájlokat (mappákat nem), amelyek 2007. április 12-e után módosultak. Jaj. Persze megoldható a feladat, de helyesen fog lefutni az a szkript egy német nyelvű Windowson is? (És akkor például a kínai nyelvű rendszerekről még nem is beszéltünk.) Ha viszont a kimenet objektumokból áll, akkor természetesen egyáltalán nincs szükség a megjelenő szöveg feldolgozására, mert minden egyes objektum „tudja” magáról, hogy ő fájl, vagy mappa, és az operációs rendszer nyelvétől függetlenül, `DateTime` objektumként adja vissza az utolsó módosításának idejét.

Most egyelőre - különösebb magyarázat nélkül - nézzük, hogy egy fájl milyen objektumjellemzőkkel, tagjellemzőkkel (member) bír, a `get-member` cmdlet futtatásával:

```
[2] PS C:\> get-item szamok.txt | get-member


   TypeName: System.IO.FileInfo

Name                       MemberType     Definition
----                       ----------     ----------
AppendText                 Method         System.IO.StreamWriter AppendTe...
CopyTo                     Method         System.IO.FileInfo CopyTo(Strin...
Create                     Method         System.IO.FileStream Create()
...
Delete                     Method         System.Void Delete()
...
MoveTo                     Method         System.Void MoveTo(String destF...
...
Attributes                 Property       System.IO.FileAttributes Attrib...
CreationTime               Property       System.DateTime CreationTime {g...
CreationTimeUtc            Property       System.DateTime CreationTimeUtc...
Directory                  Property       System.IO.DirectoryInfo Directo...
DirectoryName              Property       System.String DirectoryName {get;}
Exists                     Property       System.Boolean Exists {get;}
Extension                  Property       System.String Extension {get;}
FullName                   Property       System.String FullName {get;}
IsReadOnly                 Property       System.Boolean IsReadOnly {get;...
LastAccessTime             Property       System.DateTime LastAccessTime ...
LastAccessTimeUtc          Property       System.DateTime LastAccessTimeU...
LastWriteTime              Property       System.DateTime LastWriteTime {...
LastWriteTimeUtc           Property       System.DateTime LastWriteTimeUt...
Length                     Property       System.Int64 Length {get;}
Name                       Property       System.String Name {get;}
BaseName                   ScriptProperty System.Object BaseName {get=[Sy...
Mode                       ScriptProperty System.Object Mode {get=$catr =...
```

```
ReparsePoint             ScriptProperty System.Object ReparsePoint {get...
```

Mint ahogy egy fájltól elvárhatjuk, másolhatjuk (`CopyTo`), törölhetjük (`Delete`) és egy csomó jellemzőjét kiolvashatjuk: utolsó hozzáférés idejét (`LastAccessTime`) és hosszát (`Length`). Azaz ezekhez a tevékenységekhez nem kell külső program, még csak külön PowerShell parancs sem, maga az objektum tudja ezeket.

Pl. a törlés:

```
[3] PS C:\> (get-item szamok.txt).Delete()
```

Visszatérve pl. a `dir` parancs futtatására, általában egy könyvtárban nem egy fájl vagy alkönyvtár van, hanem több. Ebben az esetben a PowerShell a parancs végrehajtása során egy u.n. collection-t, azaz egy objektumhalmazt, objektumgyűjteményt vagy más kifejezéssel objektumtömböt kapunk vissza. Ennek a halmaznak ráadásul nem feltétlenül kell egyforma típusú elemekből állnia, mint ahogy egy könyvtárban is lehetnek fájlok is és alkönyvtárak is. A PowerShell az ilyen collection-ök, azaz gyűjtemények feldolgozására nagyon sok praktikus, nyelvi szintű támogatást ad, amit a későbbiekben látni fogunk.

1.4.2 OOP alapok

Az előzőekben már találkozhattunk azzal a kijelentéssel, hogy a PowerShell egy objektumorientált shell. Első hallásra talán furcsának tűnhet ez a szókapcsolat, már csak azért is, mert ilyen egyszerűen nem létezett korábban; a PowerShell az első, és pillanatnyilag az egyetlen ilyen tulajdonságú parancsfeldolgozó. A következőkben áttekintjük, mit is jelent, és milyen következményekkel jár az objektumorientált felépítés.

Nézzük meg részletesebben az objektumorientált programozáshoz kapcsolódó legfontosabb fogalmakat, amelyek pontos ismeretére a későbbiekben feltétlenül szükségünk lesz.

1.4.2.1 Osztály (típus)

Az osztályok az objektumok tervrajzai; tartalmazzák a különféle funkciókat megvalósító programkódot és deklarálják az objektumok adatszerkezeteit. Az osztályok és objektumok viszonya megegyezik a változótípus és konkrét változó viszonyával. Ahogyan a változóhoz tartozó típus meghatározza a változó lehetséges állapotait, és rajta végezhető műveleteket, úgy határozza meg az objektum osztálya a benne tárolható adatokat (tulajdonságokat), és az általa elvégezhető műveleteket (metódusokat). Az osztály definiálja tehát a majdani objektumok adatait, és azokat a műveleteket (eljárások és függvények), amelyek elvégzésére az osztály alapján létrehozott objektum képes.

1.4.2.2 Példány (objektum)

Az adott osztály (tervrajz) alapján létrehozott objektumpéldányok képesek az osztályban definiált változóknak megfelelő információ tárolására, és az osztály által meghatározott műveletek (eljárások és függvények) végrehajtására.

Minden objektum egy meghatározott memóriaterületet foglal el, itt tárolja adatait. Az adatok pillanatnyi értékét az objektum állapotának nevezzük. Két objektum azonban akkor sem azonos, ha állapotuk megegyezik (vagyis valamennyi adatuk rendre egyenlő), mivel az objektumot nem az állapota, hanem az általa elfoglalt memóriaterület kezdőcíme azonosítja. Programjainkban az objektumok tehát memóriacímként (objektumreferencia) jelennek meg.

1.4.2.3 Példányosítás

Az objektum osztálya tehát az objektum viselkedését, képességeit meghatározó kódból, és adatainak típusdefinícióiból áll. Példányosításnak azt a műveletet nevezzük, amelynek során a szükséges memóriaterületet lefoglalva, egy osztály alapján objektumot hozunk létre. Az objektum létrehozásakor elvégzendő műveleteket az adott osztályban definiált speciális függvény, az osztály konstruktora határozza meg. Az osztály alapján létrehozott objektum adatszegmense a konstruktornak átadott paraméterlista, vagy más adatforrás alapján a példányosítás során, vagyis az objektum létrehozása közben töltődik fel. Programunk szempontjából a példányosítás „végterméke” egy memóriacímet tartalmazó változó (objektumreferencia, mutató, pointer); a következőkben ennek segítségével érhetjük el az adott példány adatait, és hívhatjuk meg az osztályban definiált eljárásokat és függvényeket.

Ha egyszerre több azonos osztályú objektumot is használunk, az objektumok kódja csak egyetlen példányban kerül a memóriába (hiszen ez minden azonos osztályba tartozó objektum esetén feltétlenül egyforma), de természetesen az adatszegmens minden objektum esetében önállóan létezik. Minden objektum tartalmaz egy referenciát (`this`), amely az osztályát azonosítja, ennek felhasználásával hívhatja meg az osztályban tárolt metódusokat.

1.4.2.4 Metódusok és változók

Az osztályokban definiált függvényeket és eljárásokat az osztály, illetve az osztály alapján létrehozott objektum metódusainak nevezzük. Ugyanígy, az osztályok változóit statikus változóknak, a

1.4.2.5 Példányváltozó, példánymetódus

Azokat a változókat, amelyek objektum-példányonként külön memóriaterületre kerülnek példányváltozóknak, a példányváltozókat felhasználó metódusokat pedig példánymetódusoknak nevezzük. Ha meghívjuk egy objektum példánymetódusát, akkor a metódus kódja az osztályból származik ugyan, az elvégzett műveletek viszont általában az

objektum példány saját adatait fogják felhasználni, vagyis megváltoztatják az objektum állapotát.

1.4.2.6 Statikus változó, statikus metódus

Bizonyos változók nem egy konkrét objektum-példányra, hanem az egész osztályra jellemzők. Az ilyen közös, az egész osztályra jellemző változókat osztályváltozóknak, vagy statikus változóknak nevezzük. A statikus változók értéke minden példány esetén megegyezik, ezért csak egy helyen kell tárolni, minden példány ezt az egy memóriaterületet éri el.

Osztálymetódusnak, vagy statikus metódusnak nevezzük azokat a metódusokat, amelyek objektum-példány nélkül, közvetlenül az osztályra való hivatkozással futtathatók. A statikus metódusok csak a statikus változókat érik el a példányváltozókat nem. Tehát a példányváltozókat csak a példánymetódusok érik el, míg a statikus változókat a statikus- és példánymetódusok egyaránt használhatják.

1.4.2.7 Változók elrejtése

Az objektumok egyik fontos tulajdonsága, hogy a külvilág számára csak azokat a metódusokat (és ritkán adatokat) teszi elérhetővé, amelyek feltétlenül szükségesek az objektum használatához. Az objektum tehát egy jól meghatározott interfészen keresztül érhető el, amelyet természetesen az osztály készítői igyekeznek a lehető legkisebbre készíteni.

Az osztályokban definiált változók általában csak metódusokon keresztül, vagyis ellenőrzött módon érhetők el. Ezzel megakadályozható az objektum állapotának „elrontása”, azaz minden adatmezőnek csak olyan érték adható, amelynek tárolására azt a programozó szánta. Előnyös továbbá az is, hogy ilyen módon az osztály teljes belső adatszerkezete lecserélhető anélkül, hogy az osztályt használó komponenseknek erről tudniuk kellene.

1.4.2.8 Overloaded metódusok

A metódus szignatúrája a nevén kívül tartalmazza paramétereinek számát és az egyes paraméterek típusát is. A metódusokat a fordítóprogram nem pusztán a nevük, hanem a szignatúrájuk alapján azonosítja, vagyis egy osztályon belül lehet több azonos nevű, de eltérő paraméterlistájú metódus is. A metódus meghívásakor a fordító a név és az aktuálisan átadott paraméterek alapján választja ki azt a metódust, amelyik le fog futni. Ilyen metódusokkal igen gyakran fogunk találkozni a .NET keretrendszer osztályaiban is. A jelenséget „method overloading[3]”-nak, azaz metódus-újratöltésnek, -felültöltésnek, -felülbírálásnak nevezzük.

[3] Sajnos erre a kifejezésre nincsen jó magyar szó, a szokásosan használt „túlterhelt metódus” kifejezés inkább valamiféle elmegyógyászati diagnózisra emlékeztet.

1.4.2.9 Öröklődés

Az öröklődés két osztály között értelmezett kapcsolat, azt fejezik ki, hogy az egyik osztály (az utód) specializált változata a másiknak, az ősosztálynak. A specializálás során egy már meglévő objektum leírásához, tervrajzához, új, egyedi jellemzőket és képességeket adunk hozzá. A specializált osztály tehát örökli az ősosztály adatait és metódusait, de az öröklés során ezekhez újabbakat is hozzáadhatunk, illetve módosíthatjuk a meglévőket.

Egy osztály örökítésekor három lehetőséget használhatunk:

- Új változókat adhatunk hozzá az ősosztályhoz.
- Új metódusokat adhatunk hozzá az ősosztályhoz.
- Felülírhatjuk az ősosztály metódusait.[4]

1.4.3 Mi is az a .NET keretrendszer

A .NET már sok éve velünk van, és a három betű valószínűleg mindenkinek ismerősen cseng, de talán nem fölösleges röviden áttekinteni, hogy pontosan miről is van szó, és miért jó nekünk ez az egész.

A .NET Framework a Windows operációs rendszerekbe egyre szorosabban integrálódó komponens, amely az alkalmazások új generációjának fejlesztését és futtatását teszi lehetővé. A .NET egyszerűen az infrastruktúra szintjére emeli számos általános feladat megoldását, amelyekkel korábban minden programozónak magának kellett jól-rosszul megküzdenie. Aki ismeri és betartja a szabályokat, az használhatja az infrastruktúrát.

A .NET alapú programok már nem közvetlenül veszik igénybe az operációs rendszer szolgáltatásait és nem is közvetlenül a processzor futtatja őket; egy újabb absztrakciós réteg (ez maga a .NET keretrendszer) kapcsolódott be a játékba. Ez természetesen némi sebességcsökkenést okoz a programfuttatás szintjén, de igen jelentős sebességnövekedéssel jár, ha a fejlesztésre fordított időt vesszük figyelembe.

A keretrendszer alapvetően két komponesből áll, ezek a *CLR* (*Common Language Runtime*) és a mögötte álló osztályhierarchia, a *Class Library*.

1.4.3.1 Futtatókörnyezet (Common Language Runtime, CLR)

A .NET keretrendszer felhasználásával készített programok, egy szoftveres virtuális környezetben (CLR) futnak, amely biztosítja a programfuttatáshoz szükséges feltételeket. A CLR egységes futási környezetet biztosít a .NET alapú programok számára, függetlenül attól, hogy azokat milyen programnyelven készítették. Elvégzi a memóriakezelést és biztosítja a programfutáshoz szükséges egyéb alapvető szolgáltatásokat, kezeli a programszálakat, biztonságos futási környezetet ad a programkódnak, és megakadályozza, hogy a

[4] Az ősosztály adatait nem lehet felülírni.

futtatott programok bármiféle általa szabálytalannak ítélt műveletet végezzenek. A CLR fennhatósága alatt futó programokat felügyelt kódnak (managed code) nevezzük.

A CLR biztosítja a programnyelvek közötti teljes együttműködést, így lehetővé válik a különböző nyelveken megírt komponensek problémamentes együttműködése is. A CLR szigorú típusrendszerre épül, amelyhez minden CLR-kompatibilis programnyelvnek alkalmazkodnia kell. Ez azt jelenti, hogy az adott nyelv minden elemének (típusok, struktúrák, elemi adattípusok) a CLR által ismert típusokká konvertálhatónak kell lennie. További feltétel, hogy a fordítóprogramoknak a kódban lévő típusokat és hivatkozásokat leíró metaadatokat kell elhelyezniük a lefordított állományokban. A CLR ezek felhasználásával felügyeli a folyamatokat, megkeresi és betölti a megfelelő osztályokat, elhelyezi az objektum-példányokat a memóriában, stb.

A CLR minden erőforrást az adott folyamat számára létrehozott felügyelt heap-en (halom) helyez el. A felügyelt heap hasonló a hagyományos programnyelvek által használt heap-hez, de az itt létrehozott objektumokat nem a programnak kell megszüntetnie, a memória felszabadítása automatikusan történik, ha az adott objektumra már nincs többé szükség. A .NET egyik fontos szolgáltatása a szemétgyűjtő (Garbage Collector, GC), amely képes a hivatkozás nélkül maradt objektumok felkutatására, és az általuk lefoglalt memóriaterület felszabadítására.

1.4.3.2 Class Library (osztálykönyvtár)

A Class Library egy több ezer(!) osztályból álló gyűjtemény, amelyek segítségével szinte bármilyen feladatot megoldhatunk, lehetővé teszi parancssori, grafikus, vagy webes felületet, hálózati és biztonsági szolgáltatásokat használó alkalmazások fejlesztését. Segítségével gyakorlatilag a Windows rendszerek valamennyi szolgáltatása a korábbinál lényegesen egyszerűbb formában elérhetővé válik. A Class Library lehetővé teszi szinte az összes, eddig csak a Win32 API segítségével megvalósítható szolgáltatás használatát, és még több ezer más feladatra is megoldást ad. Megtalálhatjuk benne a klasszikus algoritmusok (rendezések, keresések, stb.) megvalósítását, valamint rengeteg gyakori feladat szinte kész megoldását is.

A Class Library-t természetesen bármely .NET-képes programnyelvből használhatjuk, a nyelvek közötti különbség így tulajdonképpen szinte jelentéktelen szintaktikai különbséggé válik, a lényeg, az osztálykönyvtár azonos, bármelyik nyelvet (vagy akár a PowerShellt) is használjuk.

1.4.3.3 Programnyelvek

A CLR által nyújtott szolgáltatásokat a Visual Basic, a C#, a Visual C++, és a Jscript programnyelvekből, néhány külső gyártó által fejlesztett nyelvből (például Eiffel, Perl, COBOL stb.), illetve most már a PowerShellből is elérhetjük. A szükséges fordítóprogramok parancssori változatai megtalálhatók a .NET Framework SDK csomagban különféle egyéb eszközökkel együtt (debugger, disassembler stb.). Ilyen módon a .NET alapú programok készítéséhez nincs feltétlenül szükség integrált fejlesztői környezetre (például a

Visual Studiora), elvben bármilyen alkalmazást elkészíthetünk a notepad.exe és a megfelelő fordítóprogram felhasználásával.

1.4.3.4 Programfordítás

A .NET fordítóprogramjai forráskódunkat egy köztes nyelvre (Intermediate Language, IL) fordítják. Az IL olyan processzorfüggetlen kód, amely igen hatékonyan fordítható tovább egy adott platform gépi kódjává. A compilerek kimenete tehát az IL-kód, amit a fordítóprogram exe, vagy dll állományba csomagol. Az elkészített exe állományok a felhasználó szempontjából a szokásos módon futtathatók, a háttérben azonban természetesen egészen más történik.

1.4.3.5 Programfuttatás

Mivel a processzor csak natív programok futtatására képes, az IL állományt végrehajtás előtt gépi kóddá kell fordítani. A gépi kód előállítását a futásidejű fordító (just-in-time compiler, JIT) végzi el, de csak akkor, ha az adott kód valóban le is fog futni. Az IL állományba irányuló minden metódushívás az első alkalommal meghívja a JIT-fordítót, ami gépi kóddá alakítja az adott metódust, ezt futtatja majd a processzor. Ha újra meghívjuk ugyanazt a metódust, már nincs szükség fordításra, közvetlenül az eltárolt gépi kódú változat fog lefutni. Így az egyes metódusokat csak egy alkalommal kell lefordítani, ami jelentősen gyorsíthatja a programok futását. További időmegtakarítást jelent, hogy a nem használt IL-kódot a JIT egyáltalán nem fordítja le. Amikor az alkalmazás leáll, a generált gépi kód automatikusan törlődik, újraindítás után tehát újra szükség van a metódusok futásidőben történő fordítására.

A JIT fordító természetesen minden támogatott processzor-architektúrára rendelkezésére áll, így a fordítóprogramok által készített IL-kód változtatás nélkül futtatható a különböző architektúrájú számítógépeken.[5]

1.4.3.6 Assemblyk (kódkészletek)

A .NET-ben az assemblyk jelentik az alapvető telepítési, verziókövetési és biztonsági egységet. Az assembly több fizikai dll-t és esetleg más fájlokat egy önálló logikai egységben gyűjt össze. Az assemblykben lévő fizikai dll-ek a modulok, amelyekben minden esetben IL-kód található. Az assembly ezen felül önleíró adatokat (manifest) is tartalmaz, ami metaadatok formájában írja le az assemblyben található kódot és egyéb erőforrásokat.

Ha egy assemblyt több alkalmazás is használ, akkor azt egy speciális helyre, a .NET globális assembly gyorsítótárába (Global Assembly Cache, GAC) kell helyeznünk az SDK gacutil.exe programjának segítségével. Az osztott használatú assemblyk nyilvános kulcsú titkosítással biztosítják nevük egyediségét.

[5] Persze csak akkor, ha nincs benne olyan API hívás, amely csak az adott platformon létezik.

1.4.3.7 Érték- és referenciatípusok

A .NET keretrendszerben két alapvetően eltérő típussal találkozhatunk. Az érték típusok (ilyen például az int, a byte, vagy a char) a veremmemóriában jönnek létre, és helyük automatikusan felszabadul, amikor a deklaráló kódblokk véget ér, vagyis nincs szükségük a szemétgyűjtő szolgáltatásaira. Ha értékadásban használjuk őket, akkor nem egy rájuk mutató referencia, hanem tényleges értékük másolódik át. Minden egyes érték típusú változóhoz saját, önálló memóriaterület tartozik, vagyis az egyik változón végzett művelet egyetlen másik változó tartalmát sem változtathatja meg. Saját magunk is létrehozhatunk érték szerinti típusokat a „struct" kulcsszóval. Ilyen struktúra például az int típushoz tartozó `System.Int32` is, amely az osztályokhoz hasonlóan adatmezőket és metódusokat is tartalmazhat.

A referenciatípusok nem magát az adatot, hanem egy memóriacímre való hivatkozást tárolnak. Egy memóriacímre több hivatkozás is mutathat, tehát egy referencián (vagyis az általa mutatott objektumon) végzett műveletek a többi referenciát is érintik. A referenciatípusokat a `new` operátorral hozzuk létre, és azok a szemétgyűjtő (GC) által kezelt memóriába kerülnek (vagyis a felügyelt heap-re). Amikor az értékadásokban referenciatípusokat használunk, mindig csak a referencia értéke másolódik át, a valódi objektumhoz nem nyúlunk hozzá.

Valamennyi érték és referenciatípus elérhető `System.Object` osztályú referencia segítségével. Ha az érték szerinti típust referencián keresztül érjük el, akkor a fordító olyan kódot hoz létre, amely a típusnak megfelelő memóriát a heap-en foglalja le, és átmásolja ide a változó tartalmát a veremből. Ezt az eljárást „Boxing"-nak (dobozolásnak) nevezzük.

1.4.4 COM-objektumok

A Component Object Model (COM) a komponens alapú rendszerek fejlesztésének Microsoft féle szabványa. Lehetővé teszi, hogy a COM-ügyfelek bináris szinten hívjanak meg olyan függvényeket, amelyeket a COM-objektumok számukra elérhetővé tesznek. A COM-objektumok mindegyike egyedileg azonosítható, önálló komponens, a különféle alkalmazások, és más komponensek egy jól meghatározott csatolófelületen keresztül vehetik igénybe az adott komponens szolgáltatásait.

A Microsoft is COM-komponensekkel tette lehetővé a hozzáférést az operációs rendszer függvényeihez, így a rendszerfelügyelettel kapcsolatos szkriptek igen jelentős része használ COM-objektumokat; segítségükkel érhetjük el a rendszer különféle elemeit, újrahasznosíthatjuk a már létező, előre megírt funkciókat. PowerShell esetén gyakran még akkor is COM-objektumot használunk, amikor látszólag nem, mivel a .NET rendszerfelügyelettel kapcsolatos osztályainak legtöbbje titokban szintén a megfelelő COM-objektum metódusait hívogatja (ADSI, WMI, stb.).

A szkriptjeinkben használt COM-objektumok mindegyike tehát olyan funkcionalitásgyűjtemény, amelyet gondos kezek előre elkészítettek, és jól felhasználható formában

összecsomagoltak számunkra. Ezek az objektumok szinte minden esetben bináris formában léteznek, gépünkön egy dinamikusan csatolt könyvtár (dll), vagy Active-X kontroll (ocx) képében találhatók meg. Szerencsére a COM-objektumok belső felépítéséről, adatszerkezeteiről semmit nem kell tudnunk ahhoz, hogy megírhassuk a velük kommunikáló szkriptet, és felhasználhassuk a bennük rejtőző képességeket. Az objektumokkal való kommunikáció metódusaik meghívását és tulajdonságaik beállítását jelenti; a rendszerfelügyeleti szkriptek legfontosabb funkciói ilyen módon valósíthatók meg.

1.4.4.1 Típuskönyvtárak

Típuskönyvtárnak az olyan csomagokat (dll vagy ocx fájlok) nevezzük, amelyek több, általában egymással szoros kapcsolatban álló osztályból állnak. A típuskönyvtár használatával az abban szereplő osztályok mindegyike alapján létrehozhatjuk a megfelelő objektumot, azaz elvégezhetjük az osztály példányosítását.

1.4.5 Command és cmdlet

A PowerShellben a parancsokat cmdlet-nek (kommandlet) hívjuk. Hogy miért? Valószínű megkülönböztetésül a DOS-os parancsoktól, amelyek valójában külső programok meghívását jelentik (pl.: netsh → netsh.exe, attrib → attrib.exe). Lényeg, hogy a cmdletek a PowerShell „sajátjai", egységes szerkezetet alkotnak „ige-főnév" formában. Ezeket a `get-command` cmdlettel ki is listázhatjuk:

```
[4] PS C:\> get-command

CommandType     Name                                Definition
-----------     ----                                ----------
Cmdlet          Add-Content                         Add-Content [-Path] <Strin...
Cmdlet          Add-History                         Add-History [[-InputObject...
Cmdlet          Add-Member                          Add-Member [-MemberType] <...
Cmdlet          Add-PSSnapin                        Add-PSSnapin [-Name] <Stri...
Cmdlet          Clear-Content                       Clear-Content [-Path] <Str...
Cmdlet          Clear-Item                          Clear-Item [-Path] <String...
...
Cmdlet          Write-Debug                         Write-Debug [-Message] <St...
Cmdlet          Write-Error                         Write-Error [-Message] <St...
Cmdlet          Write-GZip                          Write-GZip [-Path] <String...
Cmdlet          Write-Host                          Write-Host [[-Object] <Obj...
Cmdlet          Write-Output                        Write-Output [-InputObject...
Cmdlet          Write-Progress                      Write-Progress [-Activity]...
Cmdlet          Write-Tar                           Write-Tar [-Path] <String[...
Cmdlet          Write-Verbose                       Write-Verbose [-Message] <...
Cmdlet          Write-Warning                       Write-Warning [-Message] <...
Cmdlet          Write-Zip                           Write-Zip [-Path] <String[...
```

Említettem, hogy a cmdletek a PowerShell sajátjai, tehát nem külső programok, de ez természetesen nem azt jelenti, hogy külső programokat nem lehet meghívni. És azt sem jelenti, hogy csak a PowerShell telepítésével létrejövő cmdleteket használhatjuk, ilyene-

ket mi is írhatunk vagy különböző gyártók által (pl.: PowerShell Community Extensions: http://www.codeplex.com/PowerShellCX) készített cmdleteket is beépíthetünk a PowerShellbe, de innentől kezdve ezek a cmdletek is a PowerShell sajátjaivá válnak.

Természetesen arra is lehetőség van, hogy magunk hozzunk létre cmdleteket, de ez olyan mélyen bevinne minket az igazi programozók veszélyekkel teli földjére, hogy ebben könyvben nem foglakozunk a témával. Akit mégis érdekel a dolog, és elegendő bátorságot is érez magában, az például elolvashatja a *Professional Windows PowerShell Programming - Snap-ins, Cmdlets, Hosts, and Providers* című könyvet, ami a Wiley Publishing, Inc. gondozásában jelent meg 2008-ban angol nyelven.

Ezen kívül a PowerShellben vannak kifejezések és egyéb nyelvi elemek, amelyekkel a következőkben foglalkozom.

1.4.6 Segítség! (Get-Help)

A PowerShell kiterjedt belső súgórendszerrel rendelkezik, a súgótémák teljes listáját a `get-help` paranccsal kérhetjük le:

```
PS C:\> get-help *

Name                      Category                  Synopsis
----                      --------                  --------
ac                        Alias                     Add-Content
asnp                      Alias                     Add-PSSnapin
clc                       Alias                     Clear-Content
cli                       Alias                     Clear-Item
clp                       Alias                     Clear-ItemProperty
...
```

A megjelenő listában szerepel valamennyi alias, cmdlet, provider és számos olyan súgótéma, amely nem konkrét utasításhoz, hanem valamely PowerShell-beli fogalomhoz köthető. A lista egyes elemeihez tartozó súgótéma a következő paranccsal kérhető le:

```
PS C:\> get-help <alias, cmdlet, provider, helpfile>
```

Ha a `Get-Help` helyett a `help` függvényt használjuk, akkor a szöveget oldalakra tördelve láthatjuk.

Ha nem tudjuk a keresett funkcióhoz tartozó cmdlet teljes nevét, megtippelhetjük a kéttagú kifejezés egyik felét, vagy bármilyen töredékét, és ez alapján is kérhetünk segítséget. Az alábbi utasítás például kilistázza valamennyi Service végződésű parancsot:

```
PS C:\> get-help *-service

Name                      Category                  Synopsis
----                      --------                  --------
Get-Service               Cmdlet                    Gets the services on ...
Stop-Service              Cmdlet                    Stops one or more run...
Start-Service             Cmdlet                    Starts one or more st...
```

```
Suspend-Service          Cmdlet                Suspends (pauses) one...
Resume-Service           Cmdlet                Resumes one or more s...
Restart-Service          Cmdlet                Stops and then starts...
Set-Service              Cmdlet                Changes the display n...
New-Service              Cmdlet                Creates a new entry f...
```

A megjelenő listából a pontos név és a rövid leírás alapján már könnyen kiválaszthatjuk a megfelelőt. Az egyes cmdletek súgóoldalai négy különböző nézetben is megjeleníthetők. A nézetet a `Get-Help` paramétereivel választhatjuk ki. Alapértelmezés szerint egy rövidített, gyors áttekintésre alkalmas változat jelenik meg. A `-detailed` paraméter használatával egy bővebb változat, az `-example` paraméter hatására csak a mintapéldák (ezek nagyon hasznosak lehetnek), a `-full` paraméter segítségével pedig teljes súgótéma jeleníthető meg.

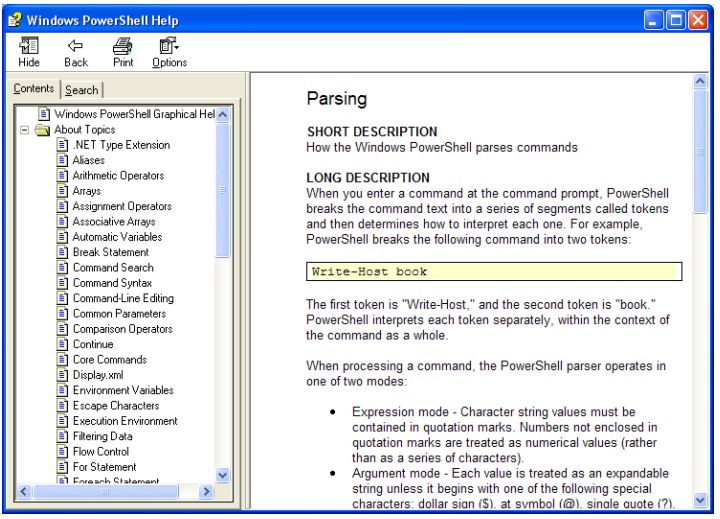
A PowerShellben nagyon sok fogalom is van, szerencsére ezekre is van súgó-téma. Ezeket `about_` előtaggal látták el:

```
[6] PS C:\> get-help about *

Name                      Category                Synopsis
----                      --------                --------
about_alias               HelpFile                Using alternate names...
about_arithmetic_opera... HelpFile                Operators that can be...
about_array               HelpFile                A compact data struct...
about_assignment_opera... HelpFile                Operators that can be...
about_associative_array   HelpFile                A compact data struct...
about_automatic_variables HelpFile                Variables automatical...
about_break               HelpFile                A statement for immed...
about_command_search      HelpFile                How the Windows Power...
about_command_syntax      HelpFile                Command format in the...
about_commonparameters    HelpFile                Parameters that every...
about_comparison_opera... HelpFile                Operators that can be...
about_continue            HelpFile                Immediately return to...
about_core_commands       HelpFile                Windows PowerShell co...
about_display.xml         HelpFile                Controlling how objec...
about_environment_vari... HelpFile                How to access Windows...
about_escape_character    HelpFile                Change how the Window...
about_execution_enviro... HelpFile                Factors that affect h...
about_filter              HelpFile                Using the Where-Objec...
about_flow_control        HelpFile                Using flow control st...
about_for                 HelpFile                A language command fo...
about_foreach             HelpFile                A language command fo...
about_function            HelpFile                Creating and using fu...
about_globbing            HelpFile                See Wildcard
about_history             HelpFile                Retrieving commands e...
about_if                  HelpFile                A language command fo...
about_line_editing        HelpFile                Editing commands at t...
about_location            HelpFile                Accessing items from ...
about_logical_operator    HelpFile                Operators that can be...
about_method              HelpFile                Using methods to perf...
about_namespace           HelpFile                Namespaces maintained...
about_object              HelpFile                Working with objects ...
about_operator            HelpFile                Types of operators su...
about_parameter           HelpFile                Working with Cmdlet p...
```

```
about_parsing              HelpFile                How the Windows Power...
about_path_syntax          HelpFile                Full and relative pat...
about_pipeline             HelpFile                Combining commands in...
about_property             HelpFile                Using object properti...
about_provider             HelpFile                Windows PowerShell pr...
about_pssnapins            HelpFile                A Windows PowerShell ...
about_quoting_rules        HelpFile                Rules for setting the...
about_redirection          HelpFile                Redirecting output fr...
about_ref                  HelpFile                How to create and use...
about_regular_expression   HelpFile                Using regular express...
about_reserved_words       HelpFile                Words in the Windows ...
about_scope                HelpFile                The visibility a func...
about_script_block         HelpFile                Grouping statements a...
about_shell_variable       HelpFile                Variables that are cr...
about_signing              HelpFile                Describes the Windows...
about_special_characters   HelpFile                The special character...
about_switch               HelpFile                Using a switch to han...
about_system_state         HelpFile                Data maintained by th...
about_types                HelpFile                Extending the .NET ty...
about_where                HelpFile                Filter objects based ...
about_while                HelpFile                A language statement ...
about_wildcard             HelpFile                Using wildcards in Cm...
```

Érdemes ezeket is elolvasni, mert gyakran olyan referenciainformációk is fellelhetők bennük, amelyek ilyen „tanító” jellegű könyvekben nem szerepelnek.

Ha valaki nem szeret karakteres konzolon olvasgatni, akkor elkészítették ezen súgótémák grafikus változatát is, ami letölthető a PowerShell weboldalról. Ezzel nem csak jobban olvasható szöveget kapunk, hanem a teljes szöveges keresés és a témák közti navigáció is sokkal egyszerűbbé válik.



23. ábra PowerShell külön letölthető grafikus súgója

1.4.7 Ki-mit-tud (Get-Member)

Korábban már megállapítottuk, hogy a PowerShellben minden objektum, és mint ilyen, lekérdezhető tulajdonságok és meghívható metódusok jól meghatározott készletével rendelkezik. Az egyes objektumok képességeinek felderítésére a `Get-Member` cmdlet szolgál. Korábban, a nem létező parancsok kipróbálásakor már találkozhattunk az alábbi példával:

```
PS C:\> 1
1
```

Mi is történt itt? A beírt szám ebben az esetben nyilvánvalóan nem parancs, hanem mi is lehet? Természetesen egy integer típusú konstans. Ha ezt a konstanst odaadjuk a `Get-Member` cmdletnek, az megmondja, hogy az integer (ami egy érték-típus) milyen tulajdonságokkal és képességekkel rendelkezik (bármilyen szám jó☺) :

```
PS C:\> 1 | Get-Member


   TypeName: System.Int32

Name         MemberType Definition
----         ---------- ----------
CompareTo    Method     System.Int32 CompareTo(Int32 value), System.Int3...
Equals       Method     System.Boolean Equals(Object obj), System.Boolea...
GetHashCode Method     System.Int32 GetHashCode()
GetType      Method     System.Type GetType()
GetTypeCode Method     System.TypeCode GetTypeCode()
ToString     Method     System.String ToString(), System.String ToString...
```

A megjelenő listából látható, hogy a típust tartalmazó struktúra neve `System.Int32`, és ugyan nem rendelkezik tulajdonságokkal (egyetlen tulajdonsága az értéke), de van néhány meghívható metódusa: vissza tudja adni a típusát, át tudja alakítani magát karakterlánccá, összehasonlítható egy másik számmal, stb. A listában szerepelnek az adott metódus különféle paraméterezéssel meghívható (overloaded) változatai, és az egyes változatok által visszaadott érték típusa is.

Próbáljuk ki, mit tud egy karakterlánc!

```
PS C:\> "Helló" | Get-Member


   TypeName: System.String

Name             MemberType            Definition
----             ----------            ----------
Clone            Method                System.Object Clone()
CompareTo        Method                System.Int32 CompareTo(Object va...
Contains         Method                System.Boolean Contains(String v...
CopyTo           Method                System.Void CopyTo(Int32 sourceI...
EndsWith         Method                System.Boolean EndsWith(String v...
```

```
Equals          Method                System.Boolean Equals(Object obj...
...
```

Ez a lista már jóval bővebb, csak győzzünk válogatni. Akinek esetleg feltűnt korábban, hogy a PowerShellben egyetlen karakterlánc-feldolgozásra képes utasítás sincsen, most megkaphatja a magyarázatot: nincs szükség ilyesmire, mert a .NET Framework `String` osztálya mindent tud, ami ebben a témában felmerülhet.

A következőkben kipróbálunk néhány egyszerű karakterlánc-műveletet.

- Bővítsük a beírt karakterláncot (vagyis a „Helló” karakterláncra hívjuk meg a `String` osztály `Insert` metódusát, első paraméter a pozíció, második a beszúrandó karakterlánc):

```
PS C:\> "Helló".Insert(5, " világ!")
Helló világ!
```

- Cseréljünk ki benne egy betűt egy másikra (a karaktereket idézőjelek vagy aposztrófok között adhatjuk meg):

```
PS C:\> "Helló".Replace('e', 'a')
Halló
```

- Alakítsuk át a karakterláncot csupa nagybetűssé:

```
PS C:\> "Helló".ToUpper()
HELLÓ
```

- Számoljuk meg a karakterlánc betűit (nem metódus, hanem tulajdonság, így nem kell zárójel):

```
PS C:\> "Helló".Length
5
```

A `Get-Member` lehet a segítségünkre akkor is, ha egy adott cmdlet kimenetét szeretnénk közelebbről megvizsgálni, megmondja milyen típusú objektumokból áll a kimenet, és azoknak milyen tulajdonságai, metódusai vannak. Ha a kimenet nem egyetlen objektum, hanem gyűjtemény (legtöbbször ez a helyzet), akkor abban akár több különböző típusú objektum is előfordulhat, de a `Get-Member` ebben az esetben sem jön zavarba; a gyűjteményben lévő valamennyi típust kilistázza. A mappákra kiadott `Get-ChildItem` például szokásosan `FileInfo` és `DirectoryInfo` objektumokat is tartalmaz, ekkor a `Get-Member` kimenetében mindkét típus megtalálható.

```
PS C:\> Get-ChildItem | Get-Member


   TypeName: System.IO.DirectoryInfo
```

```
Name                        MemberType      Definition
----                        ----------      ----------
Create                      Method          System.Void Create(), System.V...
CreateObjRef                Method          System.Runtime.Remoting.ObjRef...
...
   TypeName: System.IO.FileInfo

Name                        MemberType      Definition
----                        ----------      ----------
AppendText                  Method          System.IO.StreamWriter AppendT...
CopyTo                      Method          System.IO.FileInfo CopyTo(Stri...
...
```

A .NET objektumokkal, metódusokkal, tulajdonságokkal és adattípusokkal kapcsolatos információt a .NET Framework SDK dokumentációjából kaphatunk, itt igen részletes leírást találhatunk valamennyi osztály valamennyi elemével kapcsolatban. Egyszerűbb esetben azonban nem érdemes a rettenetes mennyiségű információ közötti keresgéléssel sokkolni magunkat, némi segítség közvetlenül a PowerShellből is hozzáférhető az alábbi módszer használatával (egy karakterlánc objektumra meghívható `Replace()` metódus definícióját kérjük le, a kimenetben látható a visszaadott érték, és a lehetséges paraméterlisták):

```
PS Run:\> ("Helló" | get-member replace).definition
System.String Replace(Char oldChar, Char newChar), System.String Replace(
String oldValue, String newValue)
```

Ugyanez (legalábbis az eleje) persze megjelenik egy „közönséges” `Get-Member` hívás kimenetében is („`Definition`” oszlop), de ott a szűkös hely miatt a hosszabb szövegek vége általában már nem látható.

1.4.8 Alias, becenév, álnév

Ahogy a fejezet bevezetőjében említettem, a PowerShell kettős célt szolgál: jó shell és jó programnyelv próbál lenni. A jó shellre az jellemző, hogy kényelmes, könnyen kezelhető, keveset kell gépelni, „fél szavakból” is megérti a felhasználót. Ennek eszköze a néhány betűből álló alias (becenevek, álnevek) használatának lehetősége. Álnevet kapcsolhatunk (akár többet is) különböző PowerShell elemekhez. Az alias bármilyen parancsban, kifejezésben teljes értékű helyettesítője gazdájának, így nincs szükség az eredeti név begépelésére. Az álnevek természetesen szkriptekben is korlátozás nélkül használhatók, de alkalmazásuk jelentősen ronthatja a szkript olvashatóságát[6].

Álnevet a következő elemekhez rendelhetünk:

- PowerShell függvények
- PowerShell szkriptek

[6] Ugyancsak a jobb olvashatóság miatt nem nagyon használunk aliasokat a könyvben szereplő mintapéldákban és szkriptekben sem.

- Fájlok
- Bármilyen végrehajtható állomány (exe, com, cmd, vbs, stb.)

A PowerShell számos beépített álnévvel is rendelkezik, ezek közül néhánnyal (`dir`, `cd`, stb.) már a korábbi fejezetekben is találkozhattunk. Az álnevek teljes listáját a `Get-Alias` cmdlet jeleníti meg:

```
PS C:\> Get-Alias

CommandType     Name                            Definition
-----------     ----                            ----------
Alias           ac                              Add-Content
Alias           asnp                            Add-PSSnapin
Alias           clc                             Clear-Content
Alias           cli                             Clear-Item
Alias           clp                             Clear-ItemProperty
Alias           clv                             Clear-Variable
...
```

Rendelkezésünkre áll továbbá az „`Alias:`” meghajtó is, amely a már megismert módon teszi hozzáférhetővé a beceneveket:

```
PS C:\> Set-Location alias:
PS Alias:\> Get-ChildItem
```

Ha nem a teljes listára, hanem csak egy konkrét aliasra vagyunk kíváncsiak, akkor a következő parancsot használhatjuk:

```
PS C:\> Get-Alias -name dir

CommandType     Name                             Definition
-----------     ----                             ----------
Alias           dir                              Get-ChildItem
```

Kissé bonyolultabb a helyzet, ha az egy adott parancshoz használható aliasokat szeretnénk felderíteni (természetesen ilyenből több is lehet). Az alábbi parancs a `Set-Location` cmdlethez tartozó álneveket listázza ki[7]:

```
PS C:\> Get-Alias * | Where-Object {$ .Definition -eq "Set-Location"}

CommandType     Name                             Definition
-----------     ----                             ----------
Alias           sl                               Set-Location
Alias           cd                               Set-Location
Alias           chdir                            Set-Location
```

A már ismert `Get-Alias` cmdleten kívül az alábbi listában látható négy további parancs is az aliasokkal kapcsolatos különféle műveletek elvégzésére szolgál.

[7] A parancs egyes részeinek jelentéséről a következő fejezetben lesz szó.

```
PS C:\> Get-Help *-alias

Name                    Category                Synopsis
----                    --------                --------
Export-Alias            Cmdlet                  Exports information a...
Get-Alias               Cmdlet                  Gets the aliases for ...
Import-Alias            Cmdlet                  Imports an alias list...
New-Alias               Cmdlet                  Creates a new alias.
Set-Alias               Cmdlet                  Creates or changes an...
```

A `New-Alias` cmdlet segítségével (meglepetés!) új álneveket definiálhatunk, paraméterként az alias nevét és a vele helyettesítendő parancsot kell megadnunk. Az alábbi parancs a „`word`” aliast hozza létre, amely a továbbiakban a szövegszerkesztő egyszerű indítását teszi lehetővé (természetesen a `winword.exe` valódi helyét kell megadnunk):

```
PS C:\> New-Alias -name word -Value "c:\program files\microsoft
office\office10\winword.exe"
PS C:\> word
```

A meghajtókhoz hasonlóan a létrehozott aliasok is csak az adott munkameneten belül élnek, a rendszeresen használni kívánt darabokat a PowerShell profilba kell felvennünk (lásd később). Az `Export-Alias` cmdlet segítségével a teljes alias listát fájlba menthetjük, a fájl pedig (például egy másik számítógépen) az `Import-Alias` parancs segítségével tölthető vissza.

1.4.9 PSDrive

Mivel a PowerShellnek kiterjedt saját parancskészlete van ezért logikus az elképzelés, hogy egy-egy parancs hasonló objektumokon is ugyanabban a formában futtatható lehessen. Például a fájlrendszer és a registry is hasonló mappastruktúra-szerűen épül fel. Ebből jöhetett a PowerShell alkotóinak az az ötlete, hogy kiterjesztették a meghajtó fogalmát, PSmeghajtó lett így a fájlrendszeren kívül a registry, a tanúsítványtár, de a függvényeinket és a környezeti és saját változóinkat is ilyen PSDrive-okon keresztül is megnézhetjük.

Ezekhez a különböző PSDrive-okhoz tartozik egy-egy provider (ami egy .NET alapú program), ami biztosítja az egységes felületet, amelyhez a PowerShell cmdletjei csatlakozhatnak. A meghajtók (illetve a mögöttük álló providerek) teszik lehetővé, hogy a `Set-Location`, a `Get-ChildItem`**,** stb. parancsok a fájlrendszerben, a registryben, a tanúsítványtárban és még számos más helyen is teljesen azonos módon működhessenek. A provider által támogatott cmdletek mindegyike használható a providerre alapuló meghajtókon, illetve vannak speciálisan egy meghatározott providerhez készült cmdletek is. Az egyes providerek úgynevezett dinamikus paramétereket is adhatnak a cmdletekhez, amelyek csak akkor használhatók, ha a cmdletet az adott provider segítségével létrehozott adatforráson használjuk. A PowerShell rendszerrel kapott providerek készlete tovább bővíthető, így újabb adatforrások (például az Active Directory!) is elérhetővé tehetők a szokásos cmdletek számára (lásd később).

A PowerShell meghajtók tehát olyan logikai adattárak, amelyek a megfelelő provider közreműködésével a fizikai meghajtókhoz hasonló módon érhetők el. A PowerShell meghajtók semmilyen módon nem használhatók a shell környezeten kívülről, de természetesen a fizikai és hálózati meghajtók csatlakoztatását, illetve eltávolítását a shell is érzékeli.

A PSmeghajtók kilistázásához használhatjuk a `Get-PSDrive` cmdletet, egy ilyen drive-on belül érvényes például a `get-childitem` parancs, azaz alias `dir`:

```
[1] PS C:\> Get-PSDrive

Name       Provider      Root                                  CurrentLocatio
                                                                       n
----       --------      ----                                  --------------
Alias      Alias
C          FileSystem    C:\
cert       Certificate   \
D          FileSystem    D:\
E          FileSystem    E:\
Env        Environment
F          FileSystem    F:\
Function   Function
G          FileSystem    G:\
HKCU       Registry      HKEY_CURRENT_USER
HKLM       Registry      HKEY_LOCAL_MACHINE
Variable   Variable

[2] PS C:\> dir hkcu:


   Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER

SKC  VC Name                           Property
---  -- ----                           --------
  2   0 AppEvents                     {}
  0  32 Console                      {ColorTable00, ColorTable01, Colo...
 23   1 Control Panel                 {Opened}
  0   2 Environment                   {TEMP, TMP}
  1   6 Identities                    {Identity Ordinal, Migrated5, Las...
  4   0 Keyboard Layout               {}
  3   1 Printers                      {DeviceOld}
 37   0 Software                      {}
  0   0 UNICODE Program Groups        {}
  0   1 SessionInformation            {ProgramCount}
  0   4 Volatile Environment          {LOGONSERVER, CLIENTNAME, SESSION...

[3] PS C:\> dir 'HKCU:\Keyboard Layout' -Recurse


   Hive: Microsoft.PowerShell.Core\Registry::HKEY CURRENT USER\Keyboard Lay
out

SKC  VC Name                           Property
---  -- ----                           --------
  1   0 IMEtoggle                     {}
```

```
   Hive: Microsoft.PowerShell.Core\Registry::HKEY CURRENT USER\Keyboard Lay
out\IMEtoggle

SKC  VC Name                           Property
---  -- ----                           --------
  0   0 scancode                       {}


   Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Keyboard Lay
out

SKC  VC Name                           Property
---  -- ----                           --------
  0   1 Preload                        {1}
  0   0 Substitutes                    {}
  0   3 Toggle                         {Hotkey, Language Hotkey, Layout ...
```

1.4.9.1 Meghajtók létrehozása és törlése (New-PSDrive, Remove-PSDrive)

PSDrive létrehozásával a gyakran használt helyek elérését jelentősen megkönnyíthetjük.

? Feladat: Készítsünk egy Run nevű PS meghajtót, amely a registry HKLM/Software/Microsoft/Windows/CurrentVersion/Run helyére mutat!

A `New-PSDrive` paramétereként az új meghajtó nevét, a használandó providert és a gyökér elérési útját kell megadnunk az alábbiak szerint, és máris használhatjuk a frissen létrehozott meghajtót:

```
PS C:\> New-PSDrive -Name Run -PSProvider Registry -root HKLM:\Software\Mi
crosoft\Windows\CurrentVersion\Run

Name        Provider      Root                                  CurrentLocat
                                                                      ion
----        --------      ----                                  ------------
Run         Registry      HKEY_LOCAL_MACHINE\Software\Micr...


PS C:\> Set-Location Run:
PS Run:\> Get-ChildItem


   Hive: Microsoft.PowerShell.Core\Registry::HKEY LOCAL MACHINE\Software\
Microsoft\Windows\CurrentVersion\Run

SKC  VC Name                           Property
---  -- ----                           --------
  3   0 OptionalComponents             {}
```

A létrehozott meghajtó csak az adott PowerShell munkamenetben használható, kilépéskor egyszerűen elvész. A rendszeresen használt meghajtókat mégsem kell minden egyes alkalommal újra létrehoznunk, a megfelelő parancsok beilleszthetők a PowerShell indításakor automatikusan lefutó profil-szkriptbe is (lásd később). Ha a munkameneten belül törölni szeretnénk a létrehozott meghajtót, ezt nagyon egyszerűen megtehetjük, csupán a nevét kell paraméterként megadnunk a `Remove-PSDrive` cmdletnek:

```
PS C:\> Remove-PSDrive Run
```

Korábban említettük, hogy a PSmeghajtók csak a PowerShellből érhetők el, a Windows, és más alkalmazások nem ismerik fel az ilyen módon létrehozott útvonalakat. Ha azonban egy külső programot a shellből indítunk el, akkor mégis van mód a korlátozás megkerülésére. Hozzuk létre a „`C:\documents and settings\scripts`” mappát és készítsünk benne egy hello.txt nevű szöveges fájlt tetszőleges tartalommal, majd rendeljük a mappához a scripts nevű PowerShell-meghajtót!

```
PS C:\> New-PSDrive -name scripts -PSProvider FileSystem -Root 'C:\Documents
and Settings\scripts'

Name        Provider      Root                                    CurrentLoc...
----        --------      ----                                    -------------
scripts     FileSystem    C:\Documents and Settings\scripts
```

Próbáljuk notepad segítségével megnyitni a hello.txt fájlt!

```
PS C:\> notepad scripts:\hello.txt
```

Elég rosszul néz ki, igaz? Az a baj, hogy ilyen esetben a beírt útvonalat nem a PowerShell, hanem az elindított alkalmazás értelmezi (illetve esetünkben nem értelmezi). Az eredmény csakis egy szép hibaüzenet lehet[8]. Azt kéne megoldanunk, hogy a PowerShell (aki persze tudja az igazságot), ne a beírt karakterláncot, hanem az annak alapján megfejtett igazi útvonalat adja oda szegény, buta notepadnak, hiszen az még soha nem is hallhatott a „`scripts:`” meghajtó létezéséről. A PowerShell belső használatú útvonalainak megfejtésére, feloldására a `Resolve-Path` cmdlet szolgál, ami az alábbi módon használható:

```
PS C:\> (Resolve-Path scripts:\hello.txt).ProviderPath
C:\Documents and Settings\scripts\hello.txt
```

Az így átalakított útvonalat már bármely külső alkalmazásnak, így a notepadnak is odaadhatjuk.

```
PS C:\> notepad (Resolve-Path scripts:/hello.txt).ProviderPath
```

[8] Nem a PowerShell, hanem a notepad fogja a hibaüzenetet megjeleníteni.

1.4.10 Változók, konstansok

Változóknak az olyan memóriaterületeket nevezzük, amelyeket szkriptünkben vagy programunkban névvel jelölünk meg, és a programnak lehetősége van az adott memóriaterület értékét futása során megváltoztatni. A névvel ellátott memóriaterület tartalmazhatja közvetlenül a változó értékét (érték típusú változók), illetve tartalmazhat egy hivatkozást (memóriacímet), ami az adatok valódi helyét azonosítja (referencia típusú változók). Minden létrehozott változó számára program adatszegmensén lefoglalódik a megfelelő nagyságú memóriaterület, amelyet ezután a programból a névre való hivatkozással érhetünk el, kiolvashatjuk vagy beállíthatjuk annak értékét, illetve referencia esetén a név segítségével érhetjük el a mutatott objektumot is. Az érték- és referenciatípusok a PowerShellben gyakorlati szempontból nem különböznek egymástól lényegesen.

A típusok közötti konverzió minden esetben a .NET szabályai szerint történik, ahogyan a későbbi példákban látható. Mi tárolható tehát egy PowerShell változóban? Bármi, amit a .NET a memóriában tárolhat, méghozzá szigorúan típusos formában.

A PowerShell változónevei minden esetben a $ karakterrel kezdődnek (ha önmagában használjuk őket és nem valamilyen változókat kezelő cmdlet segítségével), betűket, számokat és speciális karaktereket is tartalmazhatnak.

Változókat PowerShellben legegyszerűbben a következő formában tudunk létrehozni, használni:

```
PS C:\> $a = "bcdefghijklmnopqrstuvwxyz"
PS C:\> $b = 12
PS C:\> $c = Get-Location
PS C:\> $a
bcdefghijklmnopqrstuvwxyz
PS C:\> $b
12
PS C:\> $c

Path
----
C:\
```

Látszik, hogy a változók értékadásához nem kell semmilyen kulcsszót használni, az egyenlőségjel elég az értékadáshoz. A változókhoz nem feltétlenül kell típust rendelni, azaz bármilyen típust, objektumosztályt tartalmazhatnak.

Azt, hogy egy változó éppen milyen típusú értéket tartalmaz a `GetType()` metódus meghívásával kérdezhetjük le:

```
PS C:\> $a.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object


PS C:\> $b.GetType()
```

```
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int32                                    System.ValueType


PS C:\> $c.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    PathInfo                                 System.Object
```

Látszik, hogy a `$c` változónk nem egyszerűen egy szöveget tartalmaz, hanem egy `PathInfo` típusú objektumot.

A változók típusa nem rögzül, azaz egy `String` változónak később adhatunk `Int32` értéket minden további hiba nélkül, a változó típusa az értékének megfelelően módosul.

```
PS C:\> $a
bcdefghijklmnopqrstuvwxyz
PS C:\> $a = 22
```

Ez a megoldás persze nagyon kényelmes, de az automatikus változódeklaráció komoly problémákkal is járhat; minden elgépelésből új, rejtélyes változók születnek, érdekes, de legalábbis nehezen felderíthető futási hibákat és memóriaproblémákat okozva. Szkriptek esetén mindenképpen célszerű legalább a hibakeresés fázisában változtatni az alapértelmezett viselkedésen a következő módon:

```
PS C:\> Set-PSDebug -strict
```

Ezután a PowerShell hibaüzenetet ad, ha olyan változónevet használunk, amihez korábban még nem rendeltünk értéket:

```
PS C:\> $a = 2
PS C:\> $a + $b
The variable $b cannot be retrieved because it has not been set yet.
At line:1 char:7
+ $a + $b <<<<
```

Az alapértelmezett viselkedés a következő parancs használatával állítható vissza:

```
PS C:\> Set-PSDebug -off
```

A változók gyakran nem egyszerű elemeket tartalmaznak, hanem collection-öket (gyűjteményeket, vegyes elemű tömböket) vagy hagyományos tömböket:

```
PS C:\> $alias = get-alias a*
PS C:\> $alias

CommandType     Name                               Definition
-----------     ----                               ----------
```

```
Alias           ac                                  Add-Content
Alias           asnp                                Add-PSSnapin
Alias           apv                                 Add-PathVariable
Alias           adl                                 Add-DirectoryLength
Alias           asp                                 Add-ShortPath
Alias           apropos                             Get-HelpMatch
```

A fenti példában az „a” kezdetű aliasok gyűjteménye lesz a `$alias` változó tartalma. A gyűjteményekkel és tömbökkel később foglalkozom részletesebben.

Változóknak még precízebben is tudunk értéket adni. A `set-variable` cmdlet nagyon sok opciót biztosít erre:

```
PS C:\> Set-Variable -Name PI -Value 3.14159265358979 -Option Constant
PS C:\> $PI
3.14159265358979
```

A fenti példában például a PI nevű változót konstansként hoztuk létre, ami azt jelenti, hogy nem engedi felülírni más értékkel, illetve bárhonnan látható ez a változó (scope-pal, azaz láthatósággal később foglalkozom). Ezt még törölni sem lehet a `Remove-Variable` cmdlettel.

A `Set-Variable` cmdletnek van még egy érdekessége is, magyarázatot lehet adni ezzel a módszerrel a változóknak:

```
PS C:\> Set-Variable -Name Nevem -Value "Soós Tibor" -Description "Ez az én
nevem"
PS C:\> $nevem
Soós Tibor
```

De vajon hogyan olvashatjuk ki később ezt a magyarázó szöveget? Hát így nem:

```
PS C:\> $nevem.description
```

Ehhez a `get-variable` cmdletet kell használni:

```
PS C:\> (get-variable nevem).Description
Ez az én nevem
```

A `get-variable` kimeneteként nem maga a változó tartalma jön vissza, hanem egy `PSVariable` típusú objektum, aminek már kiolvasható a `Description` tulajdonsága is.

```
PS C:\> (get-variable nevem).gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    PSVariable                               System.Object
```

A korábban már mutatott `Set-Variable` cmdlethez hasonlóan a `new-variable` is használható új változók definiálásához, a két parancs között csak az a különbség, hogy a `set-variable` segítségével már meglevő változók tartalmát is meg lehet változtatni, a `new-variable` erre nem alkalmas:

```
[35] PS I:\>$x="Körte"
[36] PS I:\>New-Variable x -Value 55
New-Variable : A variable with name 'x' already exists.
At line:1 char:13
+ New-Variable  <<<< x -Value 55
```

Ha meguntunk egy változót, akkor törölhetjük is a `Clear-Variable` cmdlet segítségével. Ez különösen akkor fontos, ha például egy nagyméretű gyűjtemény kerül bele a változóba, ami aztán szükségtelenné válik. Ha nem töröljük, akkor az továbbra is foglalja a memóriát. Ez magát a változót nem szünteti meg, csak az értékét törli:

```
[29] PS I:\>$x = "alma"
[30] PS I:\>Get-Variable x

Name                           Value
----                           -----
x                              alma


[31] PS I:\>Clear-Variable x
[32] PS I:\>Get-Variable x

Name                           Value
----                           -----
x


[33] PS I:\>Remove-Variable x
[34] PS I:\>Get-Variable x
Get-Variable : Cannot find a variable with name 'x'.
At line:1 char:13
+ Get-Variable  <<<< x
```

A [32]-es promptnál látjuk, hogy a `Clear-Variable` után az `$x` változóm még létezik, csak nincs értéke. A `Remove-Variable` után a [34]-es promptban viszont már nem létezik `$x`.

Megjegyzés

Változónévnek mindenféle csúnyaságot is használhatunk, de lehetőség szerint ne tegyünk ilyet, mert a szkriptünk, parancssorunk értelmezhetőségét nagyon megnehezíthetjük:

```
[37] PS I:\>$1 = 5
[38] PS I:\>$1
5
```

```
[39] PS I:\>$1+2
7

[40] PS I:\>${kód}="sor"
[41] PS I:\>"${kód}sor"
sorsor

[42] PS I:\>$$ = "Dollár"
[43] PS I:\>$$
Dollár

[44] PS I:\>$ a="bla"
[45] PS I:\>$_a
bla
```

1.4.11 Idézőjelezés, escape használat

Az idézőjelek használata arra szolgál, hogy a PowerShell számára egyébként valamiféle speciális jelentéssel bíró szövegdarabokat egyszerű karakterláncként olvastassuk be. A PowerShellben kétfajta idézőjel létezik. Az egyik a macskaköröm (”), a másik az aposztróf (’). Ez utóbbi nem tévesztendő össze a „visszafele aposztrófal” (`- AltGr + 7), ami a korábban említett Escape, azaz „hatástalanító” karakter.

```
PS C:\> $szöveg = "karaktersorozat"
PS C:\> $szöveg
karaktersorozat
PS C:\> "Ez itt egy $szöveg."
Ez itt egy karaktersorozat.
PS C:\> 'Ez itt egy $szöveg'
Ez itt egy $szöveg
PS C:\>
```

A fenti példában látszik, hogy a macskaköröm (”) közti szövegben a PowerShell észreveszi a változókat (a $ jel alapján) és behelyettesíti a változó értékét. Az aposztróf (’)-nál nem értékel ki semmit sem, minden karaktert szó szerint értelmez.

Hogyan kell például macskaköröm (”) között akkor $ jelet kiíratni?

```
PS C:\> "Ez most itt egy `$jel"
Ez most itt egy $jel
```

Itt jön jól az Escape karakter, ami hatástalanítja a $ alaphelyzet szerinti hatását.

Vigyázat, a $ jel hatása macskakörmök között csak az első szóelválasztó karakterig érvényes! Azaz nem működik így egy adott szöveg hossztulajdonságának kiíratása:

```
PS C:\> $szó = "ablak"
PS C:\> "Az ablak szó $szó.Length karakter hosszú."
Az ablak szó ablak.Length karakter hosszú.
```

Hogyan lehet az egész kifejezést kiértékeltetni a macskakörmök között?

```
PS C:\> "Az ablak szó $($szó.Length) karakter hosszú."
Az ablak szó 5 karakter hosszú.
```

Azaz a problematikus részt bezárójelezzük, és felhívjuk a PowerShell figyelmét, hogy a változóknál megszokott kiértékelés szerint járjon el ez egész zárójeles kifejezés esetében, ezért elé teszünk még egy $ jelet.

Többsoros szöveget is egyszerűen be tudunk írni:

```
PS C:\> $hosszú = "1. sor
>> 2. sor
>> 3. sor"
>>
PS C:\> $hosszú
1. sor
2. sor
3. sor
```

Egyszerűen Entert nyomunk a sortörésnél. A parancssor értelmező beindul és észreveszi, hogy egy idézőjel után vagyunk, így átvált egy alpromptra és várja a szöveget a következő idézőjelig.

1.4.12 Sortörés, többsoros kifejezések

A PowerShell nagyon rugalmasan kezeli a többsoros kifejezéseket. Ha egy nyitó karakter (macskaköröm, aposztróf, minden fajta zárójel) után új sort kezdünk, akkor kapunk egy u.n. „nested prompt”-ot, azaz beágyazott promptot, melyet két > jel jelez, és ahol folytathatjuk a parancssorunk írását. A macskaköröm példáját már előzőleg mutattam, most nézzünk pár egyéb példát:

```
PS C:\> (get-service
>> ).count
>>
91
```

Szögletes zárójel:

```
PS C:\> $a = get-service
PS C:\> $a[
>> 1]
>>

Status   Name               DisplayName
------   ----               -----------
Stopped  Alerter            Alerter
```

Minden egyéb esetben, amikor tehát nem egyértelmű a PowerShell számára, hogy folytatódik a sor, ott segítsünk neki a már említett (`) karakterrel.

```
PS C:\> get-service -name `
>> Alerter
>>

Status   Name               DisplayName
------   ----               -----------
Stopped  Alerter            Alerter
```

A szövegek többsoros tárolására van még egy lehetőség, az u.n. „here string” alkalmazása, amit egy önálló sorban levő @” vagy @’ karakterpárral vezetünk be és egy önálló ”@ vagy ’@ karakterpárral fejezünk be. Ennél - hasonlóan a korábbiakhoz - a macskakörmös változat kiértékelődő, az aposztrófos pedig annyira „erős” idézőjelezésnek számít, hogy a köztes `, ” és ’ jelet sem veszi figyelembe:

```
PS C:\> $szöveg = @'
>> Itt aztán lehet bármi, sortörés
>> Escape karakter `
>> Aposztróf '
>> Macskaköröm "
>> '@
>>
PS C:\> $szöveg
Itt aztán lehet bármi, sortörés
Escape karakter `
Aposztróf '
Macskaköröm "
```

1.4.13 Kifejezés- és parancsfeldolgozás

A PowerShell két különböző üzemmódban képes értelmezni a beírt szöveget – kifejezés és parancs üzemmódban. Kifejezés üzemmódban a shell a legtöbb magas szintű programnyelvhez hasonlóan viselkedik: a számokat számnak tekinti, a karakterláncokat idézőjelek közé kell tennünk, stb. Kifejezések például a következők:

```
2+2
"Helló" + " világ!"
```

Parancs üzemmódban a karakterláncokhoz nincs szükség idézőjelekre, mivel a shell a változókon és a zárójelben lévő kifejezéseken kívül mindent karakterláncnak tekint. Ilyen módon értelmezi a parancsfeldolgozó az alábbi utasításokat:

```
write-host 2+2
copy-item egyik.txt masik.txt
```

Most már csak azt kellene tudnunk, hogy pontosan mi az, aminek hatására a PowerShell egyik vagy másik üzemmódot választja a parancsunk értelmezésére. Az üzemmódot az első beolvasott token (vagyis értelmezhető parancsdarab) fogja meghatározni az alábbiaknak megfelelően:

- Kifejezés üzemmódba vált a PowerShell, ha a beírt szöveg számmal (vagy egy pont karaktert követő számmal), idézőjelek közé tett karakterlánccal, vagy $ jellel kezdődik.
- Parancs üzemmódba vált a PowerShell, ha a beírt szöveg bármilyen betűvel, a `&` karakterrel, egy pont utáni szóközzel, vagy pont utáni betűvel kezdődik.

A két üzemmód vegyes használatát zárójelezéssel érhetjük el, a zárójelek közötti szövegre minden esetben újra megtörténik az üzemmód meghatározása a fenti szempontok szerint.

```
PS C:\> 2+2
4
PS C:\> Write-Host 2+2
2+2
PS C:\> Write-Host (2+2)
4
PS C:\> (Get-Date).Year + 3
2010
PS C:\> "Get-Date"
Get-Date
PS C:\> &"Get-Date"

2007. július 11. 15:37:51
```

A fenti példákon jól látható a két üzemmód közötti különbség és az üzemmód váltás kényszerítése. Az első utasítást a PowerShell kifejezés üzemmódban értelmezte, elvégezte a műveletet és kiírta az eredményt. A második sor egy cmdlet nevével, vagyis betűvel kezdődik, ennek hatására aktiválódik a parancs üzemmód, és a shell már nem adja össze a két számot, egyszerű karakterláncként (pedig nincs idézőjelek között!) olvassa be. A harmadik sor betűvel kezdődik (parancs üzemmód), de a zárójelek miatt később újra megtörténik az üzemmód meghatározása, és a zárójelben lévő számok hatására a PowerShell az összeadás erejéig kifejezés üzemmódra vált. Az negyedik sorban a zárójeles szövegre parancs üzemmód indul, így a cmdlet kimenete kerül a teljes kifejezésbe. Az ötödik sorban egy karakterláncot adunk meg (ami éppen egy cmdlet neve), de az idézőjelek miatt a PowerShell kifejezés üzemmódra vált, így nem ismeri fel a cmdletet, a nevet egyszerű karakterláncnak tekinti. Az utolsó sorban a & karakter segítségével lefuttatjuk a karakterláncot, vagyis parancs üzemmódra kényszerítjük a shellt.

? **Feladat:** Számoljuk meg, hogy hány parancs (cmdlet) van a PowerShell-ben!

A cmdletek listáját a `Get-Command` adja vissza egy objektumcsoport képében, a .NET-ben pedig minden objektumcsoporthoz tartozik egy `Count` tulajdonság, amely az elemek számát adja vissza. Vagyis:

```
PS C:\> get-command.Count
The term 'get-command.Count' is not recognized as a cmdlet, function, oper
able program, or script file. Verify the term and try again.
```

```
At line:1 char:17
+ get-command.Count <<<<
```

A hibaüzenetből megtudhatjuk, hogy nem sikerült a „`Get-Command.Count`” utasítást végrehajtani, ahogyan az várható is volt. Természetesen nem egyszerre kellene végrehajtani az egész parancsot, hanem elsőként csak a `Get-Command`-nak kellene lefutnia, ezután pedig a kimeneten megjelenő gyűjteményre kellene `Count`-ot kérni. Ezt az alábbi szintaktika szerint érhetjük el:

```
PS C:\> (get-command).Count
129
```

Láttuk a fenti példákban, hogy a PowerShell a cmdletek paramétereit alapvetően szövegként értelmezi (parancsmód), kivéve a változókat:

```
PS C:\> $bla="PowerShell"
PS C:\> Write-Output $bla
PowerShell
```

Sőt! Ilyenkor még a változó metódusaira és tulajdonságaira is lehet hivatkozni különösebb trükközés nélkül:

```
PS C:\> Write-Output $bla.Length
10
PS C:\> Write-Output $bla.split("S")
Power
hell
```

1.4.14 Utasítások lezárása

A PowerShell utasításainak lezárására két karakter is használható: az egyik a pontosvessző (`;`), a másik pedig az újsor karakter. Az újsor karakter azonban csak akkor jelenti az utasítás végét, ha az adott utasítást a PowerShell szintaktikailag teljesnek és befejezettnek tekinti, ellenkező esetben a shell megpróbálja bekérni a hiányzó befejezést.

Nézzünk egy-egy példát ezekre. Elsőként az egy sorban több kifejezés megadására:

```
PS C:\> 1+2; 54/12; "Ezek voltak az eredmények egy sorban kiszámolva"
3
4,5
Ezek voltak az eredmények egy sorban kiszámolva
```

A PowerShellnek hiányérzete van, alpromptot nyit a kifejezés korrekt lezárásához:

```
PS C:\> 2 +
>> 2
>>
4
PS C:\>
```

Ha a lezártnak is tekinthető utasításokat mégis folytatni szeretnénk, akkor ismét a korábban már megismert Escape (`) karaktert kell használnunk:

```
PS C:\> write-host `
>> Helló `
>> világ `
>>
Helló világ
```

1.4.15 Csővezeték (Pipeline)

A hagyományos shellekhez hasonlóan a PowerShellnek is fontos eleme a csővezeték (pipeline), azzal a lényeges különbséggel, hogy a PowerShell-féle csővezetékben komplett objektumok (illetve objektumreferenciák) közlekednek, így a csővezeték minden eleme megkapja az előtte álló parancs által generált teljes adathalmazt, függetlenül attól, hogy az adott kimenet hogyan jelenne meg a képernyőn.

De mi is az a csővezeték, és hogyan működik? Talán nem is szerencsés a csővezeték elnevezés, hiszen ha valamit beöntünk egy csővezeték elején, az általában változatlan formában bukkan fel a végén, a PowerShell-féle csővezeték lényege pedig éppen a benne utazó dolgok (objektumok) átalakítása, megfelelő formára hozása. Sokkal szemléletesebb, ha csövek helyett egy futószalagot képzelünk el, amelyen objektumok utaznak, a szalag mellett álló munkásaink (a cmdletek) pedig szorgalmasan elvégzik rajtuk a megfelelő átalakításokat, mindegyikük azt, amihez éppen ő ért a legjobban. Egyikük lefaragja és eldobja a fölösleget, a következő kiválogatja a hibás darabokat, egy másik kiválogatja, és szépen becsomagolja az egybetartozókat, stb. A szalag végén pedig mi megkapjuk a készterméket, ha ügyesek voltunk (vagyis a megfelelő munkásokat állítottuk a szalag mellé, és pontosan megmondtuk nekik, hogy mit kell csinálniuk), akkor éppen azt, és olyan formában, amire és ahogyan szükségünk volt.

A csövezhetőség azzal is jár, hogy az egyik cmdlet kimenete a következő cmdlet bemeneteként megy tovább, anélkül hogy nekünk az első kimenetet el kellene mentenünk változóba. Ezt a paraméterátadást a PowerShell helyettünk elvégzi. Ez - annak figyelembevételével - különösen praktikus, hogy a különböző cmdletek kimeneteként többfajta, előre nem biztos, hogy pontosan meghatározható típusú kimenet, általánosan gyűjtemény, azaz *collection* lehet.

Ráadásul, ha ilyen gyűjtemény a kimenet, akkor lehet, hogy annak első tagja hamar előáll. Ha ügyesen van megírva a következő csőszakasz helyén álló cmdlet, akkor az már a rendelkezésre álló első gyűjteményelemet el is kezdheti feldolgozni, sőt, akár tovább is adhatja az ő utána következő csőszakasznak. Ez jóval hatékonyabb feldolgozást tesz lehe-

tővé memória-felhasználás tekintetében, hiszen nem kell bevárni az utolsó elem megérkezését és eltárolni az addigi elemeket.

Nézzünk egy nagyon egyszerű példát, a már többször alkalmazott `get-member` cmdlet példáján. Látható, hogy a csőszakaszok csatlakozását a | jel (Alt Gr + W) jelenti:

```
[4] PS C:\> "sztring" | get-member


   TypeName: System.String

Name             MemberType            Definition
----             ----------            ----------
Clone            Method                System.Object Clone()
CompareTo        Method                System.Int32 CompareTo(Object val...
Contains         Method                System.Boolean Contains(String va...
CopyTo           Method                System.Void CopyTo(Int32 sourceIn...
EndsWith         Method                System.Boolean EndsWith(String va...
Equals           Method                System.Boolean Equals(Object obj)...
...
```

A beírt „sztring” kimenete maga a „sztring”, ezt küldjük tovább a `get-member` cmdletnek, amely kilistázza a sztring objektum tagjait.

De vajon honnan tudja a PowerShell, hogy a `get-member` számos paramétere közül melyik legyen a csőből kieső kimenet?

Ehhez az adott cmdlet helpje (itt most kicsit megkurtítva) ad segítséget:

```
[5] PS C:\> get-help get-member -full

NAME
    Get-Member

SYNOPSIS
    Gets information about objects or collections of objects.

SYNTAX
    Get-Member [[-name] <string[]>] [-inputObject <psobject>] [-memberType
    {<AliasProperty> | <CodeProperty> | <Property> | <NoteProperty> | <Scri
    ptProperty> | <Properties> | <PropertySet> | <Method> | <CodeMethod> |
    <ScriptMethod> | <Methods> | <ParameterizedProperty> | <MemberSet> | <A
    ll>}] [-static] [<CommonParameters>]


DETAILED DESCRIPTION
    Gets information about the members of objects. Get-Member can accept in
...

PARAMETERS
    -name <string[]>
        Specifies the member names to retrieve information about.

        Required?                    false
        Position?                    1
        Default value                *
        Accept pipeline input?       false
```

```
        Accept wildcard characters?  true

    -inputObject <psobject>
        Specifies the objects to retrieve information about. Using this par
        ameter to provide input to Get-Member results in different output t
        han pipelining the same input. When you pipeline input to Get-Membe
        r, if the input is a container, the cmdlet returns information abou
        t each unique type of element in the container. If you provide the
        same input by using the InputObject parameter, the cmdlet returns i
        nformation about the container object itself. If you want to use pi
        pelining to retrieve information about a container, you must procee
        d the pipelined input by a comma (,). For example, if you informati
        on about processes stored in a variable named $process, you would t
        ype ,$process | get-member to retrieve information about the contai
        ner.

        Required?                    false
        Position?                    named
        Default value
        Accept pipeline input?       true (ByValue)
        Accept wildcard characters?  false
...
```

Tehát a cmdlet azon paramétere, amely fogadja a csővezetéken érkező adatot az általában „`inputObject`” névre hallgat, illetve a súgóban fel van tüntetve, hogy „`Accept pipeline input? true`”.

Az *1.8 Függvények* fejezetben még részletesebben ismertetem a csövezést, hiszen ott majd mi magunk is írunk ilyen „csőképes” függvényt.

1.4.16 Kimenet (Output)

Az előző alfejezetben bemutattam a csövezés lehetőségét. Ez nem egyszerűen csak paraméterátadás, hanem el is kaphatjuk az éppen átadott paramétert, és annak tulajdonságaival, metódusaival is játszhatunk. Az így átadott paramétert a `$_` speciális, automatikusan generálódó változón keresztül érhetjük el.

Fontos fogalom még a PowerShellben az *output*, azaz a kimenet fogalma. Majdnem minden cmdletnek van kimenete. Ha mégsem lenne, akkor is van, merthogy ha nem készítünk explicit kimenetet, akkor az implicit módon alaphelyzetben az outputra adódik át. Ez az output lesz a következő csőszakaszban elérhető `$_` változó tartalma.

Nézzünk erre példát:

```
[1] PS C:\> "szöveg" | ForEach-Object{$_.Length}
6
[2] PS C:\> write-output "szöveg" | ForEach-Object{$ .Length}
6
[3] PS C:\> write-host "szöveg" | ForEach-Object{$ .Length}
szöveg
```

Az első és a második promptnál ugyanazt az eredményt kapjuk, azaz az első csőszakaszban megszületik a „szöveg” objektum, amit átadunk a következő csőszakasznak. Az első esetben implicit módon adjuk át, a második esetben pedig explicit módon.

A `write-output` cmdletet igazából nem „emberi fogyasztásra” szánják, ez tisztán a csőszakaszok közti paraméterátadásra van szánva, nem pedig a csicsás, színes-szagos konzolos adatmegjelenítésre. Más kérdés, hogy ha a csővezeték legvégén nem rendelkezünk a csővezeték tartalmáról, akkor az „kifolyik” a konzol ablakba, azaz kiíródik a képernyőre.

Nézzük meg a `write-output` helpjéből a szintaxist:

```
[7] PS C:\> (get-help write-output).syntax

Write-Output [-inputObject] <PSObject[]> [<CommonParameters>]
```

Nincs túl sok extra lehetőségünk.

A fenti [3]-as promptnál viszont nem az outputra küldöm a „`szöveg`”-et, hanem a `write-host` cmdlettel a konzolra (képernyőre), így a következő csőszakasz nem kap semmit és ezért nem is jelenik meg semmilyen hosszadat, csak a `write-host` által kiírt szöveg.

Nézzük meg a `write-host` szintaxisát is:

```
[8] PS C:\> (get-help write-host).syntax

Write-Host [[-object] <Object>] [-noNewLine] [-separator <Object>] [-foregr
oundcolor {<Black> | <DarkBlue> | <DarkGreen> | <DarkCyan> | <DarkRed> | <D
arkMagenta> | <DarkYellow> | <Gray> | <DarkGray> | <Blue> | <Green> | <Cyan
> | <Red> | <Magenta> | <Yellow> | <White>}] [-backgroundColor {<Black> | <
DarkBlue> | <DarkGreen> | <DarkCyan> | <DarkRed> | <DarkMagenta> | <DarkYel
low> | <Gray> | <DarkGray> | <Blue> | <Green> | <Cyan> | <Red> | <Magenta>
| <Yellow> | <White>}] [<CommonParameters>]
```

Ez már sok olyan lehetőséget is tartalmaz, amelyek tényleg az kiírt adatok élvezeti értékét fokozzák: háttér- és betűszínt lehet beállítani, valamint azt, hogy ne nyisson új sort a kiírás végén.

Amúgy számos egyéb helyekre is tudjuk irányítani a kimenetet:

```
[10] PS C:\> get-command write* -CommandType cmdlet

CommandType     Name                              Definition
-----------     ----                              ----------
Cmdlet          Write-Debug                       Write-Debug [-Message] <St...
Cmdlet          Write-Error                       Write-Error [-Message] <St...
Cmdlet          Write-Host                        Write-Host [[-Object] <Obj...
Cmdlet          Write-Output                      Write-Output [-InputObject...
Cmdlet          Write-Progress                    Write-Progress [-Activity]...
Cmdlet          Write-Verbose                     Write-Verbose [-Message] <...
Cmdlet          Write-Warning                     Write-Warning [-Message] <...
```

A fenti egyéb `write-…` cmdleteknek a szkriptek hibakeresésénél van elsődlegesen jelentőségük, így majd a hibakeresés fejezetben fogok ezzel részletesen foglalkozni.

1.4.17 Egyszerű formázás

Az output fogalmához tartozik hozzá, hogy három alapvető formázási módról beszéljek. Említettem, hogy a PowerShell objektumokat kezel, amelyeknek különböző kiolvasható tulajdonságaik (property) vannak. Egy-egy objektum típusnak nagyon sok ilyen tulajdonsága is lehet, így probléma, hogy akkor egy ilyen objektumot hogyan is jelenítsünk meg a képernyőn, hiszen ez az elsődleges interakciós felületünk a PowerShellel.

Nézzünk a problémára egy példát, listázzuk ki a gépen futó W-vel kezdődő szolgáltatásokat:

```
[47] PS C:\> get-service w*

Status   Name               DisplayName
------   ----               -----------
Running  W32Time            Windows Time
Running  WebClient          WebClient
Stopped  WinHttpAutoProx... WinHTTP Web Proxy Auto-Discovery Se...
Running  winmgmt            Windows Management Instrumentation
Stopped  WmdmPmSN           Portable Media Serial Number Service
Stopped  Wmi                Windows Management Instrumentation ...
Stopped  WmiApSrv           WMI Performance Adapter
Running  wscsvc             Security Center
Running  WSearch            Windows Search
Running  wuauserv           Automatic Updates
Running  WZCSVC             Wireless Configuration
```

Látszik, hogy alaphelyzetben táblázatos nézetben látjuk a szolgáltatások három tulajdonságát: `Status, Name, DisplayName`. A „`DisplayName`” oszlopban nem minden szöveg fér ki, ezért van egy másik listázási lehetőségünk is:

```
[48] PS C:\> get-service w* | format-list


Name                  : W32Time
DisplayName           : Windows Time
Status                : Running
DependentServices     : {}
ServicesDependedOn    : {}
CanPauseAndContinue   : False
CanShutdown           : True
CanStop               : True
ServiceType           : Win32ShareProcess

Name                  : WebClient
DisplayName           : WebClient
Status                : Running
DependentServices     : {}
ServicesDependedOn    : {MRxDAV}
```

```
CanPauseAndContinue : False
CanShutdown         : True
CanStop             : True
ServiceType         : Win32ShareProcess

Name                : WinHttpAutoProxySvc
DisplayName         : WinHTTP Web Proxy Auto-Discovery Service
Status              : Stopped
DependentServices   : {}
ServicesDependedOn  : {Dhcp}
CanPauseAndContinue : False
CanShutdown         : False
CanStop             : False
ServiceType         : Win32ShareProcess
…
```

Itt már jól kiférnek a hosszabb feliratok is, ezt a formázást hívjuk lista nézetnek, és az objektumok ilyen jellegű megjelenítését a `format-list` cmdlet végzi. Ezt olyan gyakran használjuk, hogy ennek rövid alias nevét érdemes mindenképpen megjegyezni: `fl`.

Van, hogy pont az ellenkezőjére van szükség, a listanézet helyett szeretnénk táblázatos nézetet. Ennek cmdletje a `format-table`, röviden az `ft`.

Mindkét esetben kapunk egy alaphelyzet szerinti tulajdonságlistát. Azaz nem szabad megijedni, ha pont azokat az információkat nem látjuk, amire szükségünk lenne. Először érdemes minden objektum esetében a `get-member`-rel ellenőrizni a tulajdonságok listáját és utána ezekre hivatkozhatunk a `format-` parancsoknál:

```
[11] PS C:\> Get-Service w* | Get-Member -MemberType properties


   TypeName: System.ServiceProcess.ServiceController

Name                MemberType    Definition
----                ----------    ----------
Name                AliasProperty Name = ServiceName
CanPauseAndContinue Property      System.Boolean CanPauseAndContinue {get;}
CanShutdown         Property      System.Boolean CanShutdown {get;}
CanStop             Property      System.Boolean CanStop {get;}
Container           Property      System.ComponentModel.IContainer Conta...
DependentServices   Property      System.ServiceProcess.ServiceControlle...
DisplayName         Property      System.String DisplayName {get;set;}
MachineName         Property      System.String MachineName {get;set;}
ServiceHandle       Property      System.Runtime.InteropServices.SafeHan...
ServiceName         Property      System.String ServiceName {get;set;}
ServicesDependedOn  Property      System.ServiceProcess.ServiceControlle...
ServiceType         Property      System.ServiceProcess.ServiceType Serv...
Site                Property      System.ComponentModel.ISite Site {get;...
Status              Property      System.ServiceProcess.ServiceControlle...


[12] PS C:\> Get-Service w* | Format-Table Name, Status, CanStop

Name                                    Status                  CanStop
----                                    ------                  -------
```

```
W32Time                                    Running                    True
W3SVC                                      Running                    True
WebClient                                  Running                    True
WinHttpAutoProxySvc                        Stopped                   False
winmgmt                                    Running                    True
WmdmPmSN                                   Stopped                   False
Wmi                                        Stopped                   False
WmiApSrv                                   Stopped                   False
wscsvc                                     Running                    True
wuauserv                                   Running                    True
WZCSVC                                     Running                    True
```

Az [11]-es promptnál lekérdezem a szolgáltatások tulajdonságait. Látszik, hogy okos a `Get-Member`, hiszen több szolgáltatás-objektumot kap a bemeneteként, de mivel mindegyik egyforma típusú, teljesen egyforma a tulajdonságaik vannak, így csak egyszer adja meg a tulajdonságlistát.

Ezután a [12]-es promptban kiválasztottam, hogy a `Name, Status, CanStop` tulajdonságok kellenek nekem, ezeket átadom paraméterként a `Format-Table` cmdletnek és megkapom a kívánt táblázatot.

Ezt még szebbé lehet tenni az `-AutoSize` kapcsolóval, amellyel csak a szükséges minimális táblázatszélességet használja, és nem húzza szét az egészet a képernyőn:

```
[13] PS C:\> Get-Service w* | Format-Table Name, Status, CanStop -AutoSize

Name                 Status CanStop
----                 ------ -------
W32Time             Running    True
WebClient           Running    True
WinHttpAutoProxySvc Stopped   False
winmgmt             Running    True
WmdmPmSN            Stopped   False
Wmi                 Stopped   False
WmiApSrv            Stopped   False
wscsvc              Running    True
WSearch             Running    True
wuauserv            Running    True
WZCSVC              Running    True
```

A `Format-Table` olyan gyakran használatos, hogy nézzük meg néhány ügyes további lehetőségét.

Például nézzük az „a” és „b” betűvel kezdődő nevű szolgáltatásokat:

```
[14] PS C:\> Get-Service [a-b]*

Status   Name               DisplayName
------   ----               -----------
Running  AeLookupSvc        Application Experience Lookup Service
Stopped  Alerter            Alerter
Running  ALG                Application Layer Gateway Service
Stopped  AppMgmt            Application Management
Stopped  aspnet_state       ASP.NET State Service
Running  AudioSrv           Windows Audio
```

```
Stopped  BITS                Background Intelligent Transfer Ser...
Stopped  Browser             Computer Browser
```

Csoportosítsuk ezeket `Status` szerint:

```
[15] PS C:\> Get-Service [a-b]* | ft -GroupBy Status


   Status: Running

Status   Name                DisplayName
------   ----                -----------
Running  AeLookupSvc         Application Experience Lookup Service


   Status: Stopped

Status   Name                DisplayName
------   ----                -----------
Stopped  Alerter             Alerter


   Status: Running

Status   Name                DisplayName
------   ----                -----------
Running  ALG                 Application Layer Gateway Service
...
```

Hoppá! Ez nem valami jó. Itt tetten érhetjük a csövezést. Ugye a `format-table` szépen kapja egymás után a szolgáltatásokat és nyit egy aktuális csoportot a `Status` alapján. Ha a következő szolgáltatás is ugyanilyen státusú, akkor szépen mögé biggyeszti, de ha nem, akkor nyit egy új csoportot. Azaz kicsit összevissza az eredményünk.

A megoldás az lenne, hogy ideiglenesen összegyűjtenénk a csővezetéken átmenő összes objektumot egy pufferben, és utána berendezzük az elemeket `Status` alapján, majd utána jelenítjük meg a csoportosított nézetet. Ezt a pufferelést és sorbarendezést végzi a `Sort-Object` cmdlet:

```
[16] PS C:\> Get-Service [a-b]* | sort-object status, name |ft -GroupBy
Status


   Status: Stopped

Status   Name                DisplayName
------   ----                -----------
Stopped  Alerter             Alerter
Stopped  AppMgmt             Application Management
Stopped  aspnet_state        ASP.NET State Service
Stopped  BITS                Background Intelligent Transfer Ser...
Stopped  Browser             Computer Browser
```

```
   Status: Running

Status   Name               DisplayName
------   ----               -----------
Running  AeLookupSvc        Application Experience Lookup Service
Running  ALG                Application Layer Gateway Service
Running  AudioSrv           Windows Audio
```

Ha pedig a hosszú nevű szolgáltatások neveit is szeretnénk látni teljesen, akkor a `-wrap` kapcsolót érdemes használni:

```
[17] PS C:\> Get-Service [a-b]* | sort-object status, name |ft -GroupBy
Status -wrap


   Status: Stopped

Status   Name               DisplayName
------   ----               -----------
Stopped  Alerter            Alerter
Stopped  AppMgmt            Application Management
Stopped  aspnet_state       ASP.NET State Service
Stopped  BITS               Background Intelligent Transfer Servic
                            e
Stopped  Browser            Computer Browser


   Status: Running

Status   Name               DisplayName
------   ----               -----------
Running  AeLookupSvc        Application Experience Lookup Service
Running  ALG                Application Layer Gateway Service
Running  AudioSrv           Windows Audio
```

Megjegyzés:

Kicsit ízlelgessük ezt a kifejezésrészt: „`Format-Table Name, Status, CanStop`”. Most ez itt mi? Miért van itt vessző? Nem elég a szóköz? Majd később nézzük részletesen a kifejezések argumentumait, elöljáróban annyit, hogy a `Format-Table` itt egy háromelemű tömböt kap argumentumként, ezért kell vessző.
Melyik argumentuma kaphat tömböt?
A súgóból ez látszik:

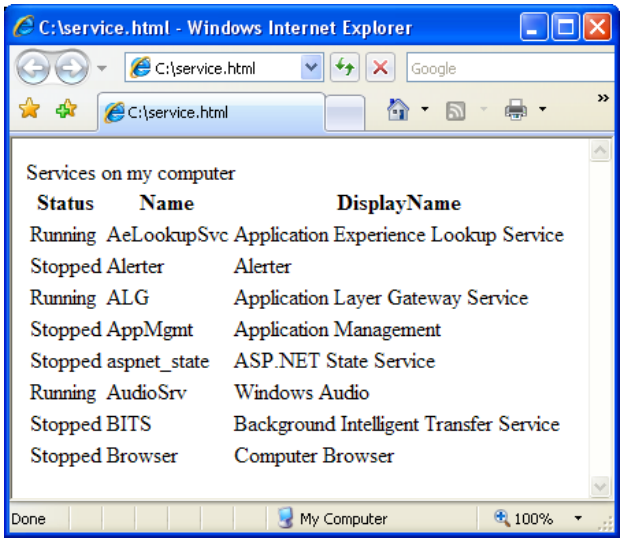
```
PARAMETERS

-property <Object[]>

Specifies the object properties that appear in the display and the order in
which they appear. Wildcards are permitted.
```

Az <Object[]> jelzés végén a szögletes zárójelpár jelöli a tömböt. Azaz a PowerShellben nagyon fontos, hogy a több paramétert nem a vessző jelzi, hanem a szóköz! Viszont a tömbök esetében az elemeket vesszővel kell elválasztani.

1.4.18 HTML output

Ha már ilyen szép táblázatokat tudunk készíteni, akkor jó lenne ezeket valamilyen még szebb formában megjeleníteni. Ezt szolgálja a PowerShell beépített HTML támogatása:

```
[7] PS C:\> Get-Service [a-b]* | ConvertTo-Html -Property
Status,Name,DisplayName -head "Services on my computer" > service.html
```



24. ábra Convert-HTML eredménye böngészőben

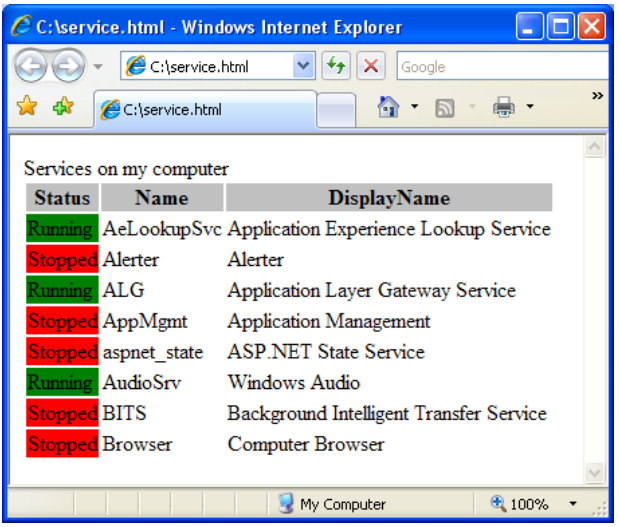
Természetesen itt a kimenetet nem a konzolon érdemes nézni, mert ott a html forráskódot látjuk, hanem érdemes átirányítani egy fájlba, amit a böngészőben lehet megnézni.

Látjuk, hogy szép táblázatos formátumot ad, kis ügyességgel lehet ezt még fokozni. Itt most még nem célom a teljes kód elmagyarázása, csak egy kis kedvcsinálóként írom ide, a PowerGUI Script editorából kimásolva:

```
Get-Service [a-b]* | ConvertTo-Html -Property Status,Name,DisplayName `
-head "Services on my computer" |
foreach {
       if ($_ -like "*<td>Running</td>*")
              {$_ -replace "<td>Running", "<td bgcolor=green>Running"}
```

```
        elseif ($  -like "*<td>Stopped</td>*")
               {$  -replace "<td>Stopped", "<td bgcolor=red>Stopped"}
        else
               {$_ -replace "<tr>", "<tr bgcolor=#C0C0C0>"}
        } > c:\service.html
```

És a végeredmény:



25. ábra Kicsit kibővített HTML kimenet a böngészőben

(Ugyan a könyvben nem látszik, de a futó szolgáltatások Status-a zöld, a leálltaké pedig piros hátterű.)

1.5 Típusok

A PowerShellben a típusoknak (vagy ha valaki inkább a programozós terminológiát szereti, akkor osztályoknak) nagyon nagy jelentősége van, hiszen a parancsok kimenete általában nem egyszerű sztring, hanem valamilyen egyéb objektum. Ennek ellenére az a cél, hogy a típuskezelés minél egyszerűbb legyen, azaz hogy a PowerShell alkalmas legyen hatékony parancssori felhasználásra.

Ebben a fejezetben a PowerShell által kezelt legfontosabb típusokat és azok kezelését mutatom be.

1.5.1 Típusok, típuskezelés

Láthattuk, hogy ha nem adunk meg konkrét típust, akkor a PowerShell a változónak adott érték alapján önállóan határozza meg, hogy milyen típusú változót is hoztunk létre. A parancssorba beirkált néhány utasítás esetén ez teljesen rendben is van, a (látszólagos) típustalanság rendszerint nem okoz különösebb problémát. Más a helyzet azonban hosszabb, bonyolultabb szkriptek esetén. Ekkor a kódolást, de még inkább a hibakeresést nagymértékben megkönnyíti, ha biztosak lehetünk benne, hogy milyen típusú változókat is használunk. A változó típusának meghatározása a következőképpen történik:

```
PS C:\> [System.Int32]$szam = 1
PS C:\> $szam = "akarmi"
Cannot convert value "akarmi" to type "System.Int32". Error: "Nem megfelel
ő a bemeneti karakterlánc formátuma."
At line:1 char:6
+ $szam  <<<< = "akarmi"
```

A `$szam` változót `Int32` típusként hozzuk létre, ezután csak számot (vagy számként értelmezhető karakterláncot) adhatunk neki értékként, más típusú változók, vagy objektumreferenciák már nem kerülhetnek bele. Az alábbi értékadás „típustalan” esetben egy karakterláncot eredményezne, de így a karakterlánc szám megfelelője lesz a változó értéke.

```
PS C:\> [System.Int32]$szam = "123"
PS C:\> $szam | get-member

   TypeName: System.Int32
```

Értékadás nélkül is létrehozhatunk változót a következő módon:

```
PS C:\> $b = [System.Int32]
```

A PowerShell változóinak típusaként tehát a .NET valamennyi érték-, illetve referenciatípusát megadhatjuk.

Most tegyük egy változóba a `Get-ChildItem` cmdlet kimenetét:

```
PS C:\> $lista = Get-ChildItem
```

Tudjuk, hogy a `Get-ChildItem` kimenete objektumokból áll, de vajon a változóba ez milyen formában kerül? Lehet, hogy csak a képernyőre kiírt lista van benne egyszerű szövegként? Hogy biztosak lehessünk a dologban, hívjuk segítségül a rendszergazda legjobb barátját, a `get-member` cmdletet:

```
PS C:\> $lista | get-member


   TypeName: System.IO.DirectoryInfo

Name                      MemberType     Definition
----                      ----------     ----------
Create                    Method         System.Void Create(), System.V...
CreateObjRef              Method         System.Runtime.Remoting.ObjRef...
CreateSubdirectory        Method         System.IO.DirectoryInfo Create...
...
```

Láthatjuk, hogy a változó teljesen az eredeti formában, `System.IO.DirectoryInfo` és `System.IO.FileInfo` objektumok alakjában tartalmazza a cmdlet kimenetét, még az sem okozott problémát, hogy a gyűjtemény két különböző típust is tartalmazott.

Interaktív üzemmódban a `Get-Member` tökéletesen alkalmas a típusok felderítésére, de a típus meghatározására nem csak itt, hanem szkriptek kódjában is szükség lehet. Ebben az esetben több megoldás közül is választhatunk, talán a legegyszerűbb az `-is` és `-isnot` operátorok használata. Kérdezzük meg, hogy milyen objektumok alkotják a `Get-Process` kimenetét:

```
PS C:\> $a = get-process
PS C:\> $a -is "System.Diagnostics.Process"
False
```

Nem `Process` objektumok?! Persze azok, csak nem jól kérdeztünk. A `Get-Process` kimenete ugyanis egy gyűjtemény, a `Process` objektumok pedig ennek belsejében vannak. A `Get-Member` ezek szerint, bár teljesen érthető okok miatt, de mégis hamis eredményt ad. Hogyan tudhatjuk meg az igazi típust? A `GetType()` metódus az `Object` osztályból öröklődik, vagyis minden elképzelhető objektum és változó esetén meghívható:

```
PS C:\> $a = get-process
PS C:\> $a.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

A kimenet tehát egyszerűen egy tömb, amelynek elemei `Object` típusú objektumok, vagyis bármi beletehető. Ha beleindexelünk a tömbbe, akkor kivehetjük belőle az egyik `Process` objektumot, hogy annak típusát is lekérdezhessük (a `Get-Member` ezt előzékenyen megteszi helyettünk):

```
PS C:\> $a = get-process
PS C:\> $a[0] -is "System.Diagnostics.Process"
True
```

Mit kell tennünk akkor, ha korlátozni szeretnénk ugyan a változóba kerülő típusok körét, de olyan módon, hogy mégis több különböző (bár hasonló) típus is beleférjen? A megoldást az osztályok közötti öröklődés környékén kell keresnünk. Minden objektum ugyanis a saját konkrét típusán kívül valamennyi ősosztály típusába is beletartozik, mindent „tud" amit az ősei, de ezen felül van még néhány speciális tulajdonsága és képessége is. Egy `FileInfo` osztályú objektum tehát nemcsak `FileInfo`, hanem `FileSystemInfo` (az ősosztály) és `Object` (mindenki ősosztálya) is egyben. A változó típusát tehát úgy kell meghatároznunk, hogy az a tárolni kívánt objektumok közös őse legyen. Ha ez csak az `Object` osztály, akkor nincs mód korlátozásra, a klubnak mindenki tagja lehet. Az alábbi változóba például csak és kizárólag `DirectoryInfo` objektumot tehetünk:

```
PS C:\> [System.IO.DirectoryInfo]$mappa = Get-Item c:\windows
PS C:\> $mappa

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----       2007.07.25.      8:06            windows
```

Ha például `FileInfo`-val próbálkozunk (aki pedig elég közeli rokon☺), csak hibaüzenetet kaphatunk:

```
PS C:\> [System.IO.DirectoryInfo]$mappa = Get-Item c:\windows\notepad.exe
Cannot convert "C:\windows\notepad.exe" to "System.IO.DirectoryInfo".
At line:1 char:32
+ [System.IO.DirectoryInfo]$mappa  <<<< = Get-Item c:\windows\notepad.exe
```

Ha mindkét típust tárolni szeretnénk (de semmi mást!), akkor közös őstípusú változót kell létrehoznunk:

```
PS C:\> [System.IO.FileSystemInfo]$mappa = Get-Item c:\windows
PS C:\> [System.IO.FileSystemInfo]$mappa = Get-Item c:\windows\notepad.exe
```

Honnan tudhatjuk meg, hogy egy adott osztálynak mi az őse? Természetesen a .NET SDK dokumentációjából bármikor, de szerencsére nem kell feltétlenül ilyen messzire mennünk. Az ősosztály egyszerűen a PowerShellből is lekérdezhető a következő módon (`BaseType` oszlop):

```
PS C:\> [System.IO.FileInfo]

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     FileInfo                                 System.IO.FileSystemInfo
```

1.5.2 Számtípusok

Nézzük akkor részletesebben a típusokat, azok közül is a leggyakoribb számtípusokat és azok jelölését:

Példa (értéktartomány)	.NET teljes típusnév	PowerShell rövid név
12 (±2147483648)	System.Int32	[int]
3.12 (±1.79769313486232e308)	System.Double	[double]
12345678901 (±9223372036854775807)	System.Int64	[long]
15d (±79228162514264337593543950335)	System.Decimal	[decimal]

Az informatikában még gyakran alkalmazunk hexadecimális számokat. Erre külön nincs PowerShellben típus, viszont használhatunk egy nagyon egyszerű jelölést:

```
[20] PS C:\> 0x1000
4096
[21] PS C:\> 0xbaba
47802
[22] PS C:\> 0xfababa
16431802
```

1.5.3 Tömbök

A programnyelvek egyik legalapvetőbb adattípusa a tömb, ami ugye egy olyan változó, amely értékek egy halmazát tartalmazza. Nézzük meg az egyszerű tömböktől kezdve a többdimenziós tömbökig a lehetőségeket!

1.5.3.1 Egyszerű tömbök

A PowerShell nagyon rugalmasan, akár a parancssorban gyorsan begépelhető módon kezeli a tömböket:

```
[17] PS C:\> $egésztömb = 1,2,11,22,100
[18] PS C:\> $egésztömb
1
2
11
22
100
[19] PS C:\> $egésztömb.gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

Látszik, hogy a tömbök létrehozásához és adatainak megadásához legegyszerűbben a vessző (,) karakter használatos.

A PowerShellben a tömbök nem csak egyforma típusú elemeket tartalmazhatnak:

```
[21] PS C:\> $vegyestömb = "szöveg", 123, 666d, 3.1415
[22] PS C:\> $vegyestömb
szöveg
123
666
3,1415
```

Hogyan lehet egyelemű tömböt létrehozni?

```
[23] PS C:\> $nemegyelemű = "elem"
[24] PS C:\> $nemegyelemű
elem
[25] PS C:\> $nemegyelemű.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object


[26] PS C:\> $egyelemű = ,"elem"
[27] PS C:\> $egyelemű
elem
[28] PS C:\> $egyelemű.gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

A fenti példában a [23]-as promptnál látszik, hogy egyszerűen egy tagot megadva természetesen – ahogy korábban is láttuk – nem jön létre egyelemű tömb. Ennek legegyszerűbb megadásához a [26]-os promptban alkalmazott trükköt érdemes használni, azaz az elem elé egy vesszőt kell rakni.

Mi van akkor, ha üres tömböt akarunk létrehozni, mert majd később, egy ciklussal akarjuk feltölteni elemekkel? Ehhez ezt a formátumot lehet használni:

```
[29] PS C:\> $ürestömb = @()
[30] PS C:\> $ürestömb.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

Ezt a „`@()`” jelölést természetesen egy- és többelemű tömbök létrehozására is felhasználhatjuk:

```
[31] PS C:\> $eet = @(1)
[32] PS C:\> $eet
1
[33] PS C:\> $eet.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array


[34] PS C:\> $tet = @(1,2,3,4)
[35] PS C:\> $tet
1
2
3
4
[36] PS C:\> $tet.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

Ha már ennyit foglalkoztunk tömbökkel, nézzük meg, hogy milyen tulajdonságaik és metódusaik vannak:

```
[37] PS C:\> $tet | Get-Member


   TypeName: System.Int32

Name        MemberType Definition
----        ---------- ----------
CompareTo   Method     System.Int32 CompareTo(Object value), System.Int3...
Equals      Method     System.Boolean Equals(Object obj), System.Boolean...
GetHashCode Method     System.Int32 GetHashCode()
GetType     Method     System.Type GetType()
GetTypeCode Method     System.TypeCode GetTypeCode()
ToString    Method     System.String ToString(), System.String ToString(...
```

Hoppá! Ez valahogy nem jó! A négy számot tartalmazó tömbünk esetében a `Get-Member` nem magának a tömbnek, hanem a tömb egyes elemeinek adta meg a tagjellemzőit. Hogyan lehetne rábírni, hogy magának a tömbnek a tagjellemzőit adja vissza? Segíteni kell az előbb látott, egyelemű tömbre vonatkozó (`,`) trükkel:

```
[38] PS C:\> ,$tet | Get-Member


   TypeName: System.Object[]

Name                MemberType     Definition
----                ----------     ----------
Count               AliasProperty  Count = Length
Address             Method         System.Object& Address(Int32 )
Clone               Method         System.Object Clone()
CopyTo              Method         System.Void CopyTo(Array array, Int32 i...
Equals              Method         System.Boolean Equals(Object obj)
Get                 Method         System.Object Get(Int32 )
GetEnumerator       Method         System.Collections.IEnumerator GetEnume...
GetHashCode         Method         System.Int32 GetHashCode()
GetLength           Method         System.Int32 GetLength(Int32 dimension)
GetLongLength       Method         System.Int64 GetLongLength(Int32 dimens...
GetLowerBound       Method         System.Int32 GetLowerBound(Int32 dimens...
GetType             Method         System.Type GetType()
GetUpperBound       Method         System.Int32 GetUpperBound(Int32 dimens...
GetValue            Method         System.Object GetValue(Params Int32[] i...
get_IsFixedSize     Method         System.Boolean get_IsFixedSize()
get IsReadOnly      Method         System.Boolean get IsReadOnly()
get IsSynchronized  Method         System.Boolean get IsSynchronized()
get_Length          Method         System.Int32 get_Length()
get_LongLength      Method         System.Int64 get_LongLength()
get_Rank            Method         System.Int32 get_Rank()
get SyncRoot        Method         System.Object get SyncRoot()
Initialize          Method         System.Void Initialize()
Set                 Method         System.Void Set(Int32 , Object )
SetValue            Method         System.Void SetValue(Object value, Int3...
ToString            Method         System.String ToString()
IsFixedSize         Property       System.Boolean IsFixedSize {get;}
IsReadOnly          Property       System.Boolean IsReadOnly {get;}
IsSynchronized      Property       System.Boolean IsSynchronized {get;}
Length              Property       System.Int32 Length {get;}
LongLength          Property       System.Int64 LongLength {get;}
Rank                Property       System.Int32 Rank {get;}
SyncRoot            Property       System.Object SyncRoot {get;}
```

Vajon miért nem ez az alapértelmezett működése a `Get-Member`-nek? A PowerShell alkotói próbáltak mindig olyan megoldásokat kitalálni, ami a gyakoribb felhasználási esetekre ad jó megoldást, márpedig inkább az a gyakoribb, hogy egy tömb elemeinek keressük a tagjellemzőit, nem pedig magának a tömbnek.

A fenti tagjellemzőkből nézzük meg a fontosabbakat:

```
[46] PS C:\> $tet.count # elemszám
4
[47] PS C:\> $tet.length # elemszám
4
[48] PS C:\> $tet.rank # dimenzió
1
[49] PS C:\> $tet.isfixedsize # fix méretű?
True
```

A fenti listában látjuk, hogy kétféle szintaxissal is lekérhetjük a tömb elemszámát, lekérhetjük, hogy hány dimenziós a tömb és hogy bővíthetjük-e a tömbünk elemszámát. Ez utóbbi szomorú eredményt ad, hiszen azt mondja, hogy ez fixméretű tömb, nem lehet elemeket hozzáadni. Vajon tényleg?

```
[54] PS C:\> $tet
1
2
3
4
[55] PS C:\> $tet += 11
[56] PS C:\> $tet
1
2
3
4
11
```

Azt láthatjuk, hogy az eredetileg négyelemű tömböt minden nehézség nélkül tudtuk öteleműre bővíteni a += operátorral. Ez azonban valójában nem ilyen egyszerűen ment végbe a felszín alatt, hanem a PowerShell létrehozott egy új, üres tömböt és szépen átmásolta az eredeti tömbünk elemeit, majd hozzábiggyesztette az új tagot. Természetesen ezt az új tömböt továbbra is a régi név alatt érjük el, de ez már valójában nem ugyanaz a tömb. Ez akkor érdekes, ha nagyon nagyméretű tömbökkel dolgozunk, hiszen akkor az új tömb felépítése során a művelet végrehajtásának végéig ideiglenesen kétszer is tárolódik a tömb, ami jelentős memória-felhasználást igényelhet.

A .NET Frameworkben van ennél „okosabb” tömb is, azt is használhatjuk PowerShellben, ez pedig a `System.Collections.ArrayList` típus. Nézzük meg ennek a tagjellemzőit:

```
[60] PS C:\> $scal = New-Object system.collections.arraylist
[61] PS C:\> ,$scal | Get-Member


   TypeName: System.Collections.ArrayList

Name               MemberType            Definition
----               ----------            ----------
Add                Method                System.Int32 Add(Object value)
AddRange           Method                System.Void AddRange(ICollectio...
BinarySearch       Method                System.Int32 BinarySearch(Int32...
Clear              Method                System.Void Clear()
Clone              Method                System.Object Clone()
Contains           Method                System.Boolean Contains(Object ...
CopyTo             Method                System.Void CopyTo(Array array)...
Equals             Method                System.Boolean Equals(Object obj)
GetEnumerator      Method                System.Collections.IEnumerator ...
GetHashCode        Method                System.Int32 GetHashCode()
GetRange           Method                System.Collections.ArrayList Ge...
GetType            Method                System.Type GetType()
get_Capacity       Method                System.Int32 get_Capacity()
```

```
get Count           Method                 System.Int32 get Count()
get IsFixedSize     Method                 System.Boolean get IsFixedSize()
get_IsReadOnly      Method                 System.Boolean get_IsReadOnly()
get_IsSynchronized Method                  System.Boolean get_IsSynchroniz...
get Item            Method                 System.Object get Item(Int32 in...
get SyncRoot        Method                 System.Object get SyncRoot()
IndexOf             Method                 System.Int32 IndexOf(Object val...
Insert              Method                 System.Void Insert(Int32 index,...
InsertRange         Method                 System.Void InsertRange(Int32 i...
LastIndexOf         Method                 System.Int32 LastIndexOf(Object...
Remove              Method                 System.Void Remove(Object obj)
RemoveAt            Method                 System.Void RemoveAt(Int32 index)
RemoveRange         Method                 System.Void RemoveRange(Int32 i...
Reverse             Method                 System.Void Reverse(), System.V...
SetRange            Method                 System.Void SetRange(Int32 inde...
set_Capacity        Method                 System.Void set_Capacity(Int32 ...
set Item            Method                 System.Void set Item(Int32 inde...
Sort                Method                 System.Void Sort(), System.Void...
ToArray             Method                 System.Object[] ToArray(), Syst...
ToString            Method                 System.String ToString()
TrimToSize          Method                 System.Void TrimToSize()
Item                ParameterizedProperty  System.Object Item(Int32 index)...
Capacity            Property               System.Int32 Capacity {get;set;}
Count               Property               System.Int32 Count {get;}
IsFixedSize         Property               System.Boolean IsFixedSize {get;}
IsReadOnly          Property               System.Boolean IsReadOnly {get;}
IsSynchronized      Property               System.Boolean IsSynchronized {...
SyncRoot            Property               System.Object SyncRoot {get;}
```

A [60]-as promptban létrehozok egy új objektumot a `new-object` cmdlettel, melynek típusa a `System.Collections.ArrayList`, majd ennek kilistázom a tagjellemőit. Itt már egész fejlett lehetőségeket találunk. Van itt `Add` metódus, amivel lehet elemet hozzáadni, meg van `Contains`, amivel lehet megvizsgálni, hogy egy adott érték benne van-e a tömbben, az `Insert` metódussal be lehet szúrni elemeket, a `Remove`-val el lehet távolítani, a `Reverse`-zel meg lehet fordítani az elemek sorrendjét, a `Sort`-tal sorba lehet rendezni:

```
[67] PS C:\> $scal = [system.collections.arraylist] (1,2,3,4,5)
[68] PS C:\> $scal
1
2
3
4
5
[69] PS C:\> $scal.Contains(4)
True
[70] PS C:\> $scal.add(1000)
5
[71] PS C:\> $scal
1
2
3
4
5
```

```
1000
[72] PS C:\> $scal.insert(3, 200)
[73] PS C:\> $scal
1
2
3
200
4
5
1000
[74] PS C:\> $scal.reverse()
[75] PS C:\> $scal
1000
5
4
200
3
2
1
[76] PS C:\> $scal.sort()
[77] PS C:\> $scal
1
2
3
4
5
200
1000
```

Az `Add`-nél vigyázni kell, hogy kimenetet ad, méghozzá azt a számot, amelyik elemként tette hozzá az éppen hozzáadott elemet. Ha ez nem kell, akkor el lehet „fojtani” a kimenetet az alábbi két módszer valamelyikével:

```
[78] PS C:\> [void] $scal.add(1234)

[79] PS C:\> $scal.add(2345) > $null
```

A [78]-es promptban átkonvertáljuk a kimenetet `[void]` típussá, azaz semmivé. A [79]-as sorban pedig átirányítjuk a semmibe a kimenetet.

Most már csak egy dolgot nem mutattam meg, hogy hogyan lehet hivatkozni a tömbelemekre:

```
[91] PS C:\> $scal[0] # első elem
1
[92] PS C:\> $scal[2..5] # harmadiktól hatodik elemig
3
4
5
200
[93] PS C:\> $scal[5..2] # hatodiktól harmadik elemig
200
5
4
```

```
3
```

A fenti példákban látható, hogy a tömbök első elemére a 0-s indexszel lehet hivatkozni. Egyszerre több egymás utáni elemre a (`..`) *range*, azaz tartomány operátorral. Ugyan operátorokkal később foglalkozom, de ez a „range” operátor annyira kötődik a tömbökhöz, hogy érdemes itt tárgyalni. Az alábbi példa mutatja az alapvető működését:

```
[94] PS C:\> 1..10
1
2
3
4
5
6
7
8
9
10
```

A `range` segítségével egy másik, érdekes módon is lehet hivatkozni a tömbelemekre, de az csak a „hagyományos” `array`-re működik:

```
[95] PS C:\> $a=1,2,3,4,10,8
[96] PS C:\> $a
1
2
3
4
10
8
[97] PS C:\> $a[-1..-3]
8
10
4
```

A [97]-es prompt azt mutatja, hogy értelmezett a negatív index, amit a PowerShell a tömb utolsó elemétől visszafele számol. A -1. elem az utolsó elem, -2. az utolsó előtti és így tovább.

Ha több elem kellene egyszerre? Adjunk meg bátran több indexet egyszerre! Ebben az esetben természetesen a visszakapott érték is egy tömb lesz.

```
PS C:\> $b = 1..100
PS C:\> $b[5,7,56]
6
8
57
```

Az már csak hab a tortán, hogy nemcsak konkrét indexet, hanem intervallumot (sőt akár több intervallumot) adhatunk meg ebben az esetben is.

```
PS C:\> $b = 1..100
PS C:\> $b[2..4 + 56..58]
3
4
5
57
58
59
```

1.5.3.2 Többdimenziós tömbök

Természetesen egy tömbnek nem csak egyirányú kiterjedése lehet, tudunk többdimenziós tömböket is létrehozni. Az alábbi példában úgy érem el a kétirányú kiterjedést, hogy a `$tábla` tömb elemeiként szintén tömböket teszek:

```
[1] PS C:\> $tábla = (1,2,3,4),("a","b","c","d")
[2] PS C:\> $tábla
1
2
3
4
a
b
c
d
[3] PS C:\> $tábla[0][0]
1
[4] PS C:\> $tábla[0][1]
2
[5] PS C:\> $tábla[1][0]
a
```

Látszik, hogy ilyenkor két indexszel hivatkozhatunk a „dimenziókra”. Sőt! Nem csak egyforma „hosszú” sorokból állhat egy „kvázi” kétdimenziós tömb:

```
[7] PS C:\> $vegyes = (1,2,3),("a","b"),("egy","kettő","három","négy")
[8] PS C:\> $vegyes[0][0]
1
[9] PS C:\> $vegyes[0][2]
3
[10] PS C:\> $vegyes[0][3]
[11] PS C:\> $vegyes[1][1]
b
[12] PS C:\> $vegyes[1][2]
[13] PS C:\> $vegyes[2][3]
négy
```

Az „igazi” többdimenziós tömböt az alábbi szintaxissal lehet hivatalosan létrehozni, és ilyenkor másként kell hivatkozni a tömbelemekre:

```
[20] PS C:\> $igazi = new-object 'object[,]' 3,2
[21] PS C:\> $igazi[2,1]="kakukk"
```

Természetesen nem csak kettő, hanem akárhány dimenziós lehet egy tömb, de ilyet valószínű csak a robottechnikában használnak. Példa egy tízdimenziós tömbre:

```
[23] PS C:\> $igazi = new-object 'object[,,,,,,,,,]' 8,3,7,5,6,7,8,9,10,3
```

1.5.3.3 Típusos tömbök

Tudunk létrehozni típusos tömböket, amelyek csak az adott típusú elemeket tartalmazhatnak:

```
[1] PS C:\> $t = New-Object int[] 20
[2] PS C:\> $t[1]="szöveg"
Array assignment to [1] failed: Cannot convert value "szöveg" to type "Syst
em.Int32". Error: "Input string was not in a correct format.".
At line:1 char:4
+ $t[1 <<<< ]="szöveg"
```

A fenti példában látszik, hogy létrehozunk előre egy 20 elemű `int` típusú tömböt, amibe ha szöveget akarunk betölteni, akkor hibát kapunk. Azonban ha új elemet biggyesztünk hozzá, akkor az már lehet akármilyen típusú.

1.5.4 Szótárak (hashtáblák) és szótártömbök

A strukturált adatszerkezetek esetében nagyon praktikusan használható adattípus a hashtable, vagy magyarul talán szótárnak vagy asszociatív tömbnek lehetne hívni.

```
[9] PS C:\> $hash = @{ Név = "Soós Tibor"; Cím = "Budapest"; "e-mail"="soost
ibor@citromail.hu"}
[10] PS C:\> $hash

Name                           Value
----                           -----
Név                            Soós Tibor
e-mail                         soostibor@citromail.hu
Cím                            Budapest
```

A hashtábla jelölése tehát egy kukac-kapcsos zárójel pár (`@{`) jellel és egy lezáró kapcsos zárójellel (`}`) jellel történik. Belül *kulcs=érték* párokat kell elhelyezni. A kulcs nevét csak akkor kell idézőjelezni, ha az valami speciális karaktert (pl. szóköz, kötőjel, stb.) tartalmaz. Az érték megadásánál az eddig megszokott formákat kell alkalmazni.

Hogyan tudok vajon egy újabb személyt felvenni ebbe a hashtáblába? Nézzük meg ehhez a hashtábla tagjellemzőit:

```
[12] PS C:\> $hash | Get-Member | ft -wrap


   TypeName: System.Collections.Hashtable
```

```
Name                MemberType            Definition
----                ----------            ----------
Add                 Method                System.Void Add(Object key, Object
                                           value)
Clear               Method                System.Void Clear()
Clone               Method                System.Object Clone()
Contains            Method                System.Boolean Contains(Object key
                                          )
ContainsKey         Method                System.Boolean ContainsKey(Object
                                          key)
ContainsValue       Method                System.Boolean ContainsValue(Objec
                                          t value)
CopyTo              Method                System.Void CopyTo(Array array, In
                                          t32 arrayIndex)
Equals              Method                System.Boolean Equals(Object obj)
GetEnumerator       Method                System.Collections.IDictionaryEnum
                                          erator GetEnumerator()
GetHashCode         Method                System.Int32 GetHashCode()
GetObjectData       Method                System.Void GetObjectData(Serializ
                                          ationInfo info, StreamingContext c
                                          ontext)
GetType             Method                System.Type GetType()
get_Count           Method                System.Int32 get_Count()
get_IsFixedSize     Method                System.Boolean get_IsFixedSize()
get_IsReadOnly      Method                System.Boolean get_IsReadOnly()
get_IsSynchronized  Method                System.Boolean get_IsSynchronized(
                                          )
get_Item            Method                System.Object get_Item(Object key)
get_Keys            Method                System.Collections.ICollection get
                                           Keys()
get_SyncRoot        Method                System.Object get_SyncRoot()
get_Values          Method                System.Collections.ICollection get
                                          _Values()
OnDeserialization   Method                System.Void OnDeserialization(Obje
                                          ct sender)
Remove              Method                System.Void Remove(Object key)
set_Item            Method                System.Void set_Item(Object key, O
                                          bject value)
ToString            Method                System.String ToString()
Item                ParameterizedProperty System.Object Item(Object key) {ge
                                          t;set;}
Count               Property              System.Int32 Count {get;}
IsFixedSize         Property              System.Boolean IsFixedSize {get;}
IsReadOnly          Property              System.Boolean IsReadOnly {get;}
IsSynchronized      Property              System.Boolean IsSynchronized {get
                                          ;}
Keys                Property              System.Collections.ICollection Key
                                          s {get;}
SyncRoot            Property              System.Object SyncRoot {get;}
Values              Property              System.Collections.ICollection Val
                                          ues {get;}
```

Láthatjuk, az `Add` metódust, így nézzük meg, azzal mit kapunk:

```
[13] PS C:\> $hash.Add("Név","Fájdalom Csilla")
Exception calling "Add" with "2" argument(s): "Item has already been added.
 Key in dictionary: 'Név'  Key being added: 'Név'"
At line:1 char:10
+ $hash.Add( <<<< "Név","Fájdalom Csilla")
```

Hát erre bizony hibajelzést kaptam, mert nem szereti ez az adatszerkezet, ha ugyanolyan kulccsal még egy értéket akarok felvenni. Csak megváltoztatni tudom az adott kulcshoz tartozó értéket:

```
[17] PS C:\> $hash.set_Item("Név","Fájdalom Csilla")
[18] PS C:\> $hash

Name                           Value
----                           -----
e-mail                         soostibor@citromail.hu
Cím                            Budapest
Név                            Fájdalom Csilla


[19] PS C:\> $hash["Név"]="Beléd Márton"
[20] PS C:\> $hash

Name                           Value
----                           -----
e-mail                         soostibor@citromail.hu
Cím                            Budapest
Név                            Beléd Márton


[21] PS C:\> $hash.Név = "Pandacsöki Boborján"
[22] PS C:\> $hash

Name                           Value
----                           -----
e-mail                         soostibor@citromail.hu
Cím                            Budapest
Név                            Pandacsöki Boborján
```

A fenti példában látszik, hogy három különböző szintaxissal is lehet módosítani értékeket ([17], [19] és [21]-es sorok).

Lekérdezni az értékeket is a fenti lehetőségekhez hasonlóan lehet:

```
[27] PS C:\> $hash.Név
Pandacsöki Boborján
[28] PS C:\> $hash["Név"]
Pandacsöki Boborján
[29] PS C:\> $hash.get_item("Név")
Pandacsöki Boborján
```

Szóval lehetőségek elég széles tárháza áll rendelkezésre. Külön lekérhetjük a hashtábla kulcsait és értékeit is:

```
[32] PS C:\> $hash.keys
e-mail
adat
Cím
Név
[33] PS C:\> $hash.values
soostibor@citromail.hu
12345
Budapest
Pandacsöki Boborján
```

De visszatérve a problémámhoz, hogyan lehet még egy ember adatait berakni ebbe a hashtáblába? Nagyon egyszerűen, hashtábla-tömböt kell létrehozni, méghozzá abból a `System.Collections.ArrayList` fajtából, amit tudunk bővíteni:

```
[37] PS C:\> $névjegyek = New-Object system.collections.arraylist
[38] PS C:\> $névjegyek.Add(@{Név="Soós Tibor";
"e-mail"="soostibor@citromail.hu";Cím="Budapest"})
0
[39] PS C:\> $névjegyek.Add(@{Név="Fájdalom Csilla";
"e-mail"="fcs@citromail.hu";Cím="Zamárdi"})
1
[40] PS C:\> $névjegyek

Name                           Value
----                           -----
e-mail                         soostibor@citromail.hu
Cím                            Budapest
Név                            Soós Tibor
e-mail                         fcs@citromail.hu
Cím                            Zamárdi
Név                            Fájdalom Csilla


[41] PS C:\> $névjegyek.count
2
[42] PS C:\> $névjegyek[0]

Name                           Value
----                           -----
e-mail                         soostibor@citromail.hu
Cím                            Budapest
Név                            Soós Tibor


[43] PS C:\> $névjegyek[1]

Name                           Value
----                           -----
e-mail                         fcs@citromail.hu
Cím                            Zamárdi
Név                            Fájdalom Csilla


[44] PS C:\> $névjegyek[1].Név
```

```
Fájdalom Csilla
[45] PS C:\> $névjegyek[1]."e-mail"
fcs@citromail.hu
```

Így ezzel a hashtábla-tömbbel adatbázis-szerű alkalmazási területek adatleképezési igényeit is nagyon jól ki lehet elégíteni.

1.5.5 Dátumok ([datetime], Get-Date, Set-Date)

A dátumok kezelésével kapcsolatban egy alapszabályt mindenképpen be kell tartanunk: **soha** ne kezdjünk el kézihajtány módszerrel dátumokat faragni, a beépített metódusok és tulajdonságok használata nem csak egyszerűbb megoldáshoz vezet, de a számtalan hibalehetőség elkerülésének érdekében egyenesen kötelező.

Mit tud a PowerShell a dátumokkal kapcsolatban? Igazság szerint nem túl sokat, mindössze két egyszerű cmdletet kapunk: A `Get-Date` segítségével az aktuális rendszerdátumot és időt kérdezhetjük le, a `Set-Date` cmdlet pedig ezek beállítását képes elvégezni. A háttérben azonban mindig ott van a .NET `DateTime` osztálya; ennek segítségével már bármit megtehetünk.

? **Feladat:** Készítsünk dátumot a következő karakterláncból: „2007. május 12.”, és alakítsuk át „2007.05.12” alakra (természetesen úgy, hogy bármilyen dátumra működjön)!

Dátumot leíró karakterláncok beolvasására a `DateTime` osztály statikus `Parse()` metódusa szolgál. A `Parse()` egy mindenevő metódus, ismeri és beolvassa az összes szokásos dátumformátumot:

```
PS C:\> PS C:\> [DateTime]::Parse("2007. május 12.")

2007. május 12. 0:00:00
```

A fenti kifejezés egy `DateTime` objektumot ad vissza (aki nem hiszi, annak `Get-Member` segít), ennek egy metódusát kell meghívnunk, hogy az egyszerűbb alakot előállítsuk (az eredmény már nem dátum, hanem karakterlánc!):

```
PS C:\> PS C:\> ([DateTime]::Parse("2007. május 12.")).ToShortDateString()
2007.05.12.
```

Megjegyzés

A `[datetime]` konstruktora is képes némi dátumértelmezésre, azonban nem olyan okos, mint a `Parse` metódus:

```
[11] PS C:\> $d = [datetime] "2007.05.12."
[12] PS C:\> $d
```

```
2007. május 12. 0:00:00
[13] PS C:\> $d = [datetime] "2007. május 12."
Cannot convert value "2007. május 12." to type "System.DateTime". Error: "T
he string was not recognized as a valid DateTime. There is a unknown word s
tarting at index 6."
At line:1 char:16
+ $d = [datetime]  <<<< "2007. május 12."
```

A [11]-es sorban létrehozott `[datetime]` típusú változóban az ottani formában szövegként megadott dátumot képes volt értelmezni, de a [13]-as sor szövegét már nem.

? **Feladat:** Állapítsuk meg, hogy milyen napra esik az aktuális dátum 13 év múlva!

A 13 évnyi időutazás a PowerShellben nem lehet probléma: egy `Get-Date` eredményére meg kell hívnunk az `AddYears()` metódust. A hét napjának nevét pedig meglepő módon a `DayOfWeek` tulajdonság adja vissza.

```
PS C:\> (get-date).AddYears(13).DayOfWeek
Saturday
```

A `Get-Date` kicsit használható `[datetime]` objektumok konstruktoraként is:

```
[14] PS C:\> $d = Get-Date -year 2007 -month 5 -day 12
[15] PS C:\> $d

2007. május 12. 14:02:26
```

A fenti példában a `Get-Date` cmdlettel majdnem ugyanazt értem el, mint a megjegyzés [11]-es sorában. Egy fontos különbség van: míg a korábbi példában a nem megadott óra, perc, másodperc érték nulla lett, addig a `Get-Date` használatával az aktuális óra, perc, másodperc érték helyettesítődik be, ami esetleg nem kívánatos a programunk működése szempontjából.

? **Feladat:** Listázzuk ki azokat a folyamatokat, amelyek az elmúlt 1 órán belül indultak el a számítógépen!

Először is le kell gyártanunk az egy órával ezelőtti időt, a változatosság kedvéért használjuk most a `DateTime` osztály statikus `Now` tulajdonságát. (Statikus tagokkal részletesebben majd a *1.5.8 .NET típusok, statikus tagok* fejezetben lesz szó.) A második sor azokat a `Process` objektumokat válogatja le, amelyeknek `StartTime` tulajdonságában ennél későbbi időpont szerepel.

```
PS C:\> $ora = [DateTime]::Now.AddHours(-1)
PS C:\> Get-Process | Where-Object {$_.StartTime -ge $ora}

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
```

```
    37       2     2016         40    30     0,09   3144 cmd
   780      20    67944       6456   198    19,06   2496 iexplore
   172       8    10012       2280    52     0,91   2884 mstsc
    38       2     1004        848    30     0,25   3128 notepad
    78       3     1428       2668    33     3,25   1256 taskmgr
```

A második sorban az `$ora` változó helyére természetesen beírhattuk volna magát a kifejezést is, a két parancs csak a jobb áttekinthetőség miatt került külön sorba.

? **Feladat:** Állítsuk vissza a számítógép óráját 10 perccel!

A rendszeridőt a `Set-Date` cmdlet segítségével tologathatjuk, paraméterként `TimeSpan` objektumot (lásd később), vagy az adott területi beállítások mellett időintervallumként értelmezhető karakterláncot is megadhatunk. A megoldás tehát:

```
PS C:\> Set-Date -adjust -0:10:0

2007. július 11. 21:10:16
```

1.5.5.1 Időtartam számítás (New-TimeSpan)

A `New-TimeSpan` cmdlet segítségével dátum, illetve időintervallumokat adhatunk meg. A alábbi parancs például a 2006. december 31-e óta eltelt időt adja vissza:

```
PS C:\> New-TimeSpan (Get-Date) (Get-Date -month 12 -day 31 -year 2006)


Days              : -192
Hours             : -23
Minutes           : -59
Seconds           : -59
Milliseconds      : -984
Ticks             : -166751999843750
TotalDays         : -192,999999819155
TotalHours        : -4631,99999565972
TotalMinutes      : -277919,999739583
TotalSeconds      : -16675199,984375
TotalMilliseconds : -16675199984,375
```

A cmdletnek két `DateTime` objektumot kell paraméterként adnunk, a visszaadott érték pedig egy `TimeSpan` objektum, amely a két dátum közötti különbséget tartalmazza. A `TimeSpan` objektumtól természetesen egyesével is elkérhetjük a fenti értékek bármelyikét a megfelelő tulajdonság nevére való hivatkozással:

```
PS C:\> $d=New-TimeSpan (Get-Date) (Get-Date -month 12 -day 31 -year 2006)
PS C:\> $d.Ticks
-166752000000000
```

A `Ticks` igényel talán egy kis magyarázatot, ez az 1600. január 1. 0:00 óta eltelt 100 ms-okban jelzi az eltelt időt. (1600. előtti eseményeket nem nagyon tudunk egyszerűen se a PowerShellel, se a .NET keretrendszerrel kezelni.)

1.5.6 Automatikus típuskonverzió

Korábban már láttuk, hogy a PowerShell megpróbálja automatikusan megváltoztatni az objektumok típusát, ha szükséges, a minél kényelmesebb, egyszerűbb parancsbevitel érdekében:

```
[1] PS I:\>1+2.0+"3"
6
[2] PS I:\>(1+2.0+"3").GetType().FullName
System.Double
```

Az [1]-es promptban látszik, hogy össze tudok adni egy egész számot, egy lebegőpontos számot egy „szöveg” formátumú számmal anélkül, hogy nekem kellene típuskonverziót végezni. A PowerShell ezt helyettem elvégzi. Megnézi, hogy a művelet tagjait vajon át lehet-e alakítani olyan típusra, amellyel egyrészt a művelet elvégezhető, másrészt nem történik adatvesztés. Erre a célra ebben az esetben a `System.Double` típus alkalmas, így a PowerShell minden tagot erre konvertál, illetve a végeredményt is ilyen formában adja meg.

Ez nem csak a matematikai műveletekre igaz, hanem az összehasonlításokra is:

```
[12] PS I:\>15 -eq 15d
True
[13] PS I:\>15.0 -eq 15d
True
[14] PS I:\>15 -eq "15"
True
[15] PS I:\>(15).GetType().FullName
System.Int32
[16] PS I:\>(15d).GetType().FullName
System.Decimal
[17] PS I:\>(15.0).GetType().FullName
System.Double
[18] PS I:\>("15").GetType().FullName
System.String
```

Azaz az egyenlőségvizsgálat (`-eq`) nem alkalmas arra, hogy a nem egyforma típusú objektumokat kiszűrhessük segítségével, hiszen a PowerShell típuskonverziót végezhet. Részletesebben az összehasonlítási lehetőségekkel az *1.6.3 Összehasonlító operátorok* fejezetben foglalkozom.

Az automatikus típuskonverziónál fontos észben tartani, hogy a kifejezések szigorúan balról jobbra értékelődnek ki, így nem mindegy, hogy milyen sorrendben adjuk meg a műveleteink paramétereit:

```
[20] PS C:\> 1+"2"
3
[21] PS C:\> "2"+1
21
```

A [20]-as sorban először egy `int` típusú számmal találkozik a parancselemző, ehhez próbálja hozzáigazítani a szöveges formátumban megadott „2”-t és így végzi el a műveletet, amelynek eredménye 3 lett.

A [21]-es sorban a szöveges „2”-höz igazítja az 1-et, amit szintén szöveggé alakít, és a két szöveg összeadásának, azaz összeillesztésének eredményét adja meg, ami „21” lett. Láthatjuk, hogy ennek eredménye tényleg sztring típusú:

```
[22] PS C:\> ("2"+1).gettype().fullname
System.String
```

1.5.7 Típuskonverzió

Előzőekben mutattam, hogy a PowerShell a kifejezésekben megpróbálja a triviális típuskonverziókat elvégezni, olyan típusúvá próbálja konvertálni a tagokat, amely nem jár információveszteséggel.

Ha mi magunk akarunk típuskonverziót elvégezni és nem a PowerShell automatizmusára bízni a kérdést, akkor erre is van lehetőség:

```
[1] PS C:\> $a = "1234"
[2] PS C:\> $b = [int] $a
[3] PS C:\> $b.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int32                                    System.ValueType


[4] PS C:\> $c = 4321
[5] PS C:\> $d = [string] $c
[6] PS C:\> $d.gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object
```

Ennél trükkösebb dolgokat is lehet típuskonverzió segítségével elvégezni:

Például az Exchange Server 2007-ben van egy „rejtélyes” karaktersorozat az Administrative Group objektumnál. Próbáljuk ezt megfejteni típuskonverzió segítségével. Elsőként alakítsuk át a sztringet karaktertömbbé:

```
[10] PS C:\> $s = "FYDIBOHF23SPDLT"
[11] PS C:\> [char[]] $s
F
```

```
Y
D
I
B
O
H
F
2
3
S
P
D
L
T
```

Majd ebből csináljunk egy egésztömböt, amely a karakterkódokat tartalmazza:

```
[12] PS C:\> [int[]][char[]] $s
70
89
68
73
66
79
72
70
50
51
83
80
68
76
84
```

Majd ezt a számsort töltsük bele egy csővezetékbe és vonjunk ki minden eleméből egyet:

```
[13] PS C:\> [int[]][char[]] $s | ForEach-Object{$  -1}
69
88
67
72
65
78
71
69
49
50
82
79
67
75
83
```

Az így megkapott számsort alakítsuk vissza karaktersorozattá:

```
[14] PS C:\> [char[]]([int[]][char[]] $s | ForEach-Object{$  -1})
E
X
C
H
A
N
G
E
1
2
R
O
C
K
S
```

Itt már az éles szeműek felismerik a megoldást, a többieknek segítsünk azzal, hogy a karaktersorozatból sztringet gyúrunk össze:

```
[15] PS C:\> [string][char[]]([int[]][char[]] $s | ForEach-Object{$_ -1})
E X C H A N G E 1 2 R O C K S
```

Ez majdnem tökéletes, azzal a különbséggel, hogy felesleges a sok szóköz.

Megjegyzés

Alaphelyzetben a PowerShell, amikor egy karaktertömbből sztringet rak össze, akkor az összefűzött elemek közé egy szóközt tesz elválasztó karakterként. De szerencsére ez testre szabható a gyári `$ofs` változóval, ami az `Output Field Separator`. Ha ennek a változónak adunk egy üres sztring értéket (“”), akkor ez pont célra vezet:

```
[16] PS C:\> $ofs=""
[17] PS C:\> [string][char[]]([int[]][char[]] $s | ForEach-Object{$ -1})
EXCHANGE12ROCKS
```

Az `$ofs` változó – ellentétben a többi, „igazi” automatikus változóval –alaphelyzetben nem létezik. Ha létrehozzuk, akkor a benne tárolt sztring lesz az elválasztó karakter.

Még a dátum típussal kapcsolatban szoktunk gyakran típuskonverziót végezni. Például szeretnék készíteni egy olyan függvényt, ami kiszámolja, hogy hány nap van még a születésnapomig:

```
[9] PS I:\>function szülinap ([string] $mikor)
>> {
>>  ([datetime] ([string]((get-date).Year) + "-$mikor") - (get-date)).Days
>> }
>>
[10] PS I:\>szülinap "10-14"
```

```
178
```

A függvény működése: sztring formátumban beírom a születési hó, nap értékét kötőjelesen (`10-14`), ehhez az aktuális dátum (`Get-Date`) sztringgé alakított évszám részét hozzátoldom, ebből az új, teljes dátumot kiadó sztringből csinálok egy dátumot `[datetime]`, ebből a dátumból kivonom az aktuális dátumot, majd ennek az egésznek veszem a napokban kifejezett értékét (`.Days`).

1.5.8 .NET típusok, statikus tagok

A következő táblázatban összefoglaltam a PowerShell által rövid névvel is hivatkozott típusokat és azok .NET Frameworkbeli típusmegnevezését:

PowerShell rövid név	.NET típusnév
[int]	System.Int32
[long]	System.Int64
[string]	System.String
[char]	System.Char
[bool]	System.Boolean
[byte]	System.Byte
[double]	System.Double
[decimal]	System.Decimal
[float]	System.Single
[single]	System.Single
[regex]	System.Text.RegularExpressions.Regex
[array]	System.Array
[xml]	System.Xml.XmlDocument
[scriptblock]	System.Management.Automation.ScriptBlock
[switch]	System.Management.Automation.SwitchParameter
[hashtable]	System.Collections.Hashtable
[psobject]	System.Management.Automation.PSObject
[type]	System.Type
[datetime]	System.DateTime
[void]	System.Void

Ezen kívül természetesen használható más típus is a .NET Frameworkből, de akkor a típus hivatkozásánál a teljes névvel kell hivatkozni, mint ahogy ezt tettük korábban a `System.Collections.ArrayList` típusnál. Az adott típusú (vagy más szóval osztályba tartozó) objektumot, azaz annak a típusnak egy példányát a `new-object` cmdlettel tudjuk létrehozni.

Ezen kívül nagyon sok olyan típus van a .NET Frameworkben, amelyek estében nem csak akkor profitálhatunk az osztály metódusaiból, ha azok egy példányát, objektumát hozzuk létre, hanem maga az osztály (típus) is rendelkezik metódusokkal, tagjellemzőkkel. Például ilyen a `[math]` osztály. Ebből nem csinálunk tényleges objektumot, hanem magának a `[math]` típusnak hívjuk meg a metódusait, tulajdonságait. Ehhez speciális szintaxist kell alkalmazni:

```
[1] PS C:\> [math]::pi
3,14159265358979
```

A fenti példában a `[math]` osztály `PI` tulajdonságát olvasom ki. Ehhez a (`::`) u.n. static member hivatkozást kell alkalmazni.

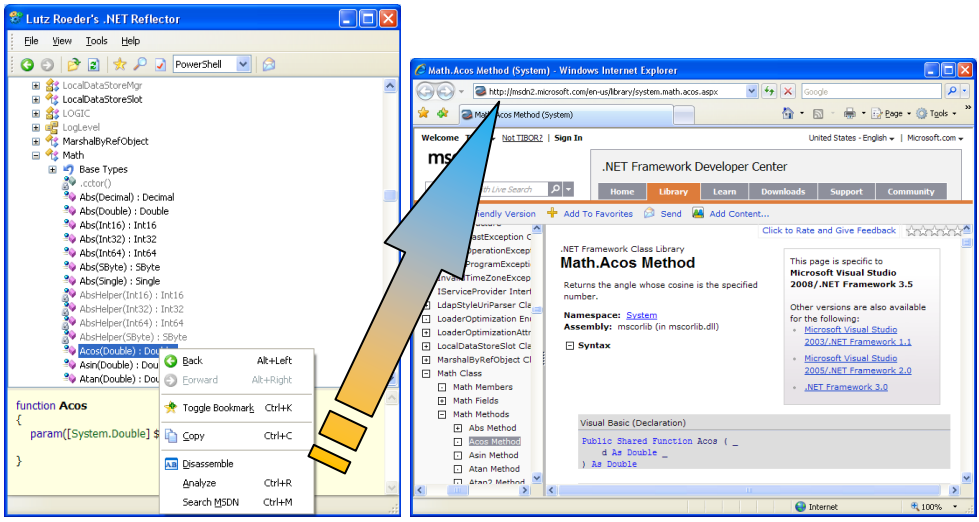
Nézzük meg, hogy hogyan lehet egy statikus metódust meghívni:

```
[2] PS C:\> [math]::Pow(3,2)
9
```

Itt a hatványozás statikus metódust hívtam meg: $3^2 = 9$.

1.5.9 A .NET osztályok felderítése

Az osztályok (típusok) felderítéséhez, a használatuk magyarázatához nagyon jól használhatjuk a *1.3.3* fejezetben említett Reflector programot:



26. ábra A Reflector közvetlenül megnyitja a .NET osztály magyarázatát

Látható, hogy a helyi menüben rögtön meg tudja hívni nekünk az adott osztály vagy metódus magyarázatát az MSDN weboldalról.

Természetesen a `get-member` cmdlettel is kilistázhatók a statikus tagjellemzők a `-static` kapcsolóval. Alább például a `[system.convert]` osztály `ToInt32` metódusa látszik:

```
[19] PS C:\> [system.convert] | get-member ToInt32 -MemberType methods -stat
ic | ft -wrap


   TypeName: System.Convert

Name    MemberType Definition
----    ---------- ----------
ToInt32 Method     static System.Int32 ToInt32(String value, Int32 fromBase
                   ), static System.Int32 ToInt32(Object value), static Sys
                   tem.Int32 ToInt32(Object value, IFormatProvider provider
                   ), static System.Int32 ToInt32(Boolean value), static Sy
                   stem.Int32 ToInt32(Char value), static System.Int32 ToIn
                   t32(SByte value), static System.Int32 ToInt32(Byte value
                   ), static System.Int32 ToInt32(Int16 value), static Syst
                   em.Int32 ToInt32(UInt16 value), static System.Int32 ToIn
                   t32(UInt32 value), static System.Int32 ToInt32(Int32 val
                   ue), static System.Int32 ToInt32(Int64 value), static Sy
                   stem.Int32 ToInt32(UInt64 value), static System.Int32 To
                   Int32(Single value), static System.Int32 ToInt32(Double
                   value), static System.Int32 ToInt32(Decimal value), stat
                   ic System.Int32 ToInt32(String value), static System.Int
                   32 ToInt32(String value, IFormatProvider provider), stat
                   ic System.Int32 ToInt32(DateTime value)
```

Ezzel a metódussal tetszőleges számrendszer számait lehet egésszé konvertálni:

```
[14] PS C:\> [system.convert]::ToInt32("F", 16)
15
[15] PS C:\> [system.convert]::ToInt32("111111", 2)
63
[16] PS C:\> [system.convert]::ToInt32(111111, 2)
63
```

Vagy még egy praktikus .NET osztály a véletlen szám generátor. Érdekes módon itt nem statikus metódusokkal játszhatunk, hanem ténylegesen létre kell hozni az adott osztály egy példányát és annak kell meghívni a számgeneráló metódusát:

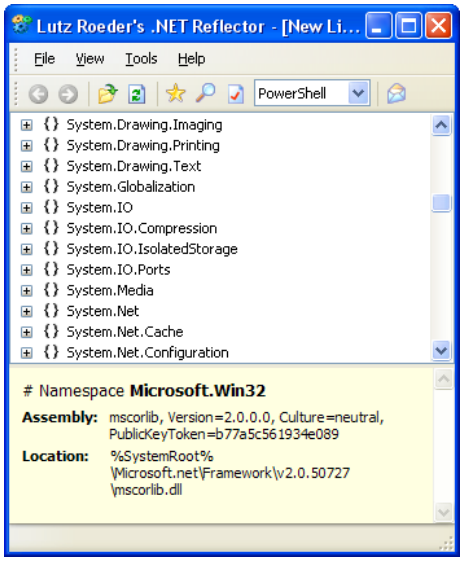
```
[30] PS C:\> $rnd = new-object random
[31] PS C:\> $rnd.Next(100)
9
[32] PS C:\> $rnd.NextDouble()
0,780635437825991
```

A fenti példában látható, hogy vagy egy adott egészig terjedő egész véletlen számot generáltatunk a `Next` metódussal, vagy egy 0 és 1 közötti lebegőpontos számot a `NextDouble` segítségével.

Megjegyzés

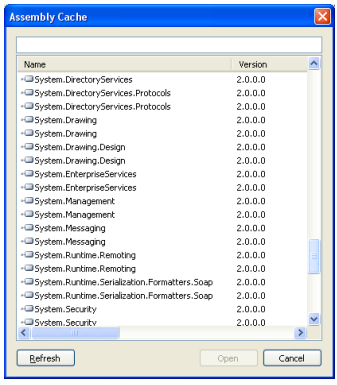
A legtöbb esetben nem is kell kiírni a .NET osztály teljes nevét. Ha a rövid név alapján is egyértelmű, hogy melyik osztályra is gondoltunk, akkor az is elég.

Nézzük kicsit jobban meg a Reflectorral a különböző .NET osztályokat! Keressük például meg a PowerShell osztályait, amelyeket a `System.Management.Automation` névtérben keresendők:



27. ábra Hol a System.Management.Automation névtér?

A .NET osztályok jó része gyorsítótárba töltődik, próbáljuk meg ezt kihasználni, és a Reflector *File* menüjének *Open Cache...* parancsát hívjuk meg:

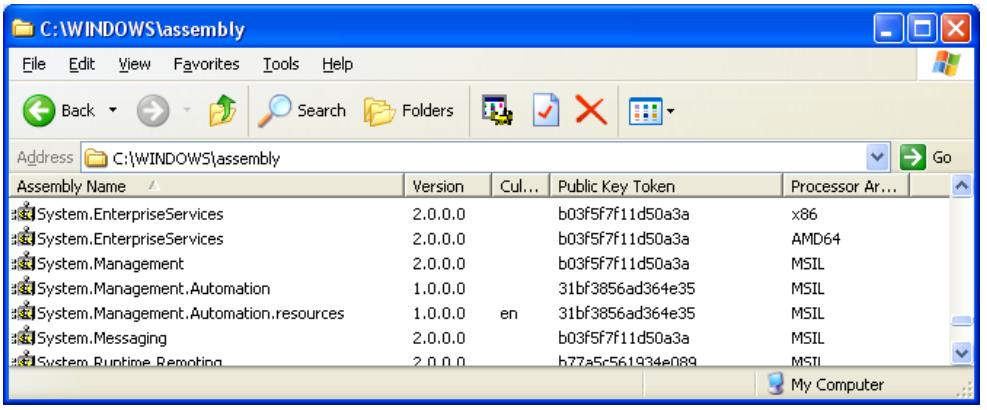


28. ábra Open Cache...

Ez sem elég, bár van *System.Management* névtér, de abban a dll-ben nincs *Automation*, tehát az sem jó. Hol van akkor a nekünk megfelelő dll? Hiába keresem a PowerShell telepítő könyvtárában:

```
[1] PS C:\> $pshome
C:\WINDOWS\system32\WindowsPowerShell\v1.0
```

Ott sincsen ilyen dll. Akkor hol van? Gonosz módon a .NET építőelemek (assembly) tárolási helyét a Windows kicsit elrejti a kíváncsi szemek elől.



29. ábra Az Assembly-tár

Itt rá is lelünk a keresett *System.Management.Automation* névtérre, de vajon hogyan lehet ezt megnyitni a Reflectorral? Mert az ottani *File/Open* menüvel mindenképpen .exe, .dll vagy .mcl kiterjesztésű fájlt akarna megnyitni. Márpedig ha a *C:\WINDOWS\assembly* útra böngészünk, akkor ott nem látunk dll-eket! Na ez a rejtegetés! Márpedig a dll ott van, csak épp a Windows Explorer rejtegeti előlünk, de a PowerShell ablakban már célhoz tudunk jutni:

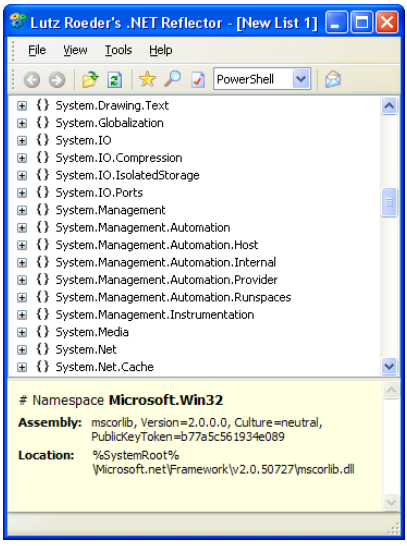
```
[16] PS C:\> cd C:\WINDOWS\assembly\GAC_MSIL
C:\WINDOWS\assembly\GAC_MSIL
[17] PS C:\WINDOWS\assembly\GAC MSIL> dir


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS\assembly\GA
    C_MSIL


Mode            LastWriteTime       Length Name
----            -------------       ------ ----
...
d----   2008.01.28.    16:53        <DIR> System.Management
d----   2008.02.22.    22:56        <DIR> System.Management.Automation
d----   2008.02.22.    22:56        <DIR> System.Management.Automation.reso
                                          urces
d----   2008.01.28.    16:53        <DIR> System.Messaging
```

```
d----    2008.01.28.     16:53         <DIR> System.Runtime.Remoting
...
```

Belépve a *System.Management.Automation* könyvtárba és egy kicsit még beleásva a mappastruktúrába megtaláljuk a keresett dll-t:

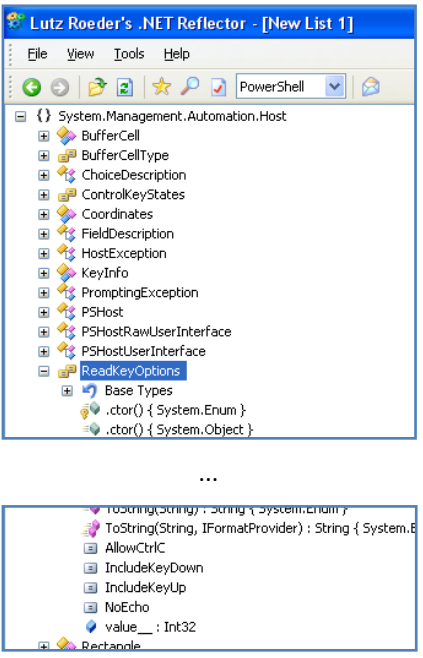
```
[21] PS C:\WINDOWS\assembly\GAC MSIL\System.Management.Automation\1.0.0.0  3
1bf3856ad364e35> dir


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS\assembly\GA
    C MSIL\System.Management.Automation\1.0.0.0  31bf3856ad364e35


Mode            LastWriteTime       Length Name
----            -------------       ------ ----
-a---    2008.02.22.     22:56      1564672 System.Management.Automation.dll
```

Ezt az elérési utat kell beadni a Reflectornak, és akkor már feltárulnak a PowerShell .NET osztályai:



30. ábra A Reflector, immár a PowerShell névtereivel bővítve

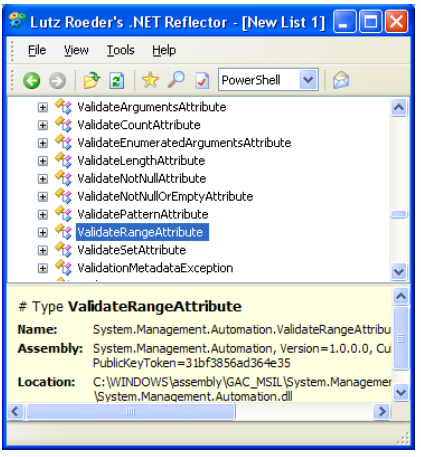
Vajon mire jó ez a sok trükközés? Igazából nem gyakran kell ide járkálni, néhány típus belső világáról azonban gyűjthetünk hasznos információkat:



31. ábra A ReadKeyOption felderítése

A fenti példában például meg tudjuk nézni, hogy a `ReadKey` metódusnak milyen opciói vannak. (A `ReadKey` metódussal a *2.1.3 Lépjünk kapcsolatba a konzolablakkal ($host)* fejezetben lesz részletesen szó.)

Vagy a függvények paramétereinek ellenőrzését lehetővé tevő osztályokat deríthetjük fel:



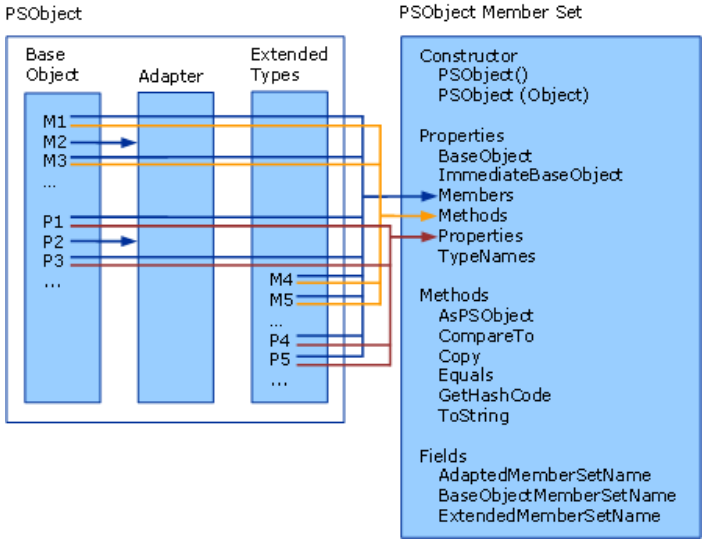
32. ábra Validálási osztályok felderítése

Ezekről részletesen a *1.8.2.8 Paraméterek, változók ellenőrzése (validálás)* fejezetben lesz szó.

1.5.10 Objektumok testre szabása, kiegészítése

Elöljáróban annyit meg kell jegyezni, hogy természetesen a PowerShell parancssori értelmezőjétől nem várhatjuk el, hogy magát a .NET osztályokat módosítsa. A PowerShell csak a saját „*generic*" osztályát, a `PSObject` osztályt és annak példányait képes módosítani.

Vegyünk egy általános objektumot, hívjuk őt „Base Object"-nek. A PowerShell elképzelhető, hogy nem minden metódust és tulajdonságot tesz elérhetővé, ezeket egy adapter réteggel elfedi. Ha bármilyen módosítást, kiegészítést teszünk ehhez az objektumhoz, például definiálunk mi magunk valamilyen tulajdonságot vagy metódust hozzá, akkor azt egy újabb, „Extended Types" rétegben tesszük meg. Így alakul ki a végleges tulajdonság- és metóduslistánk, amit az alábbi ábra illusztrál:



33. ábra PowerShell adaptált objektummodellje
(forrás: http://msdn2.microsoft.com/en-us/library/cc136098(VS.85).aspx)

Az így elérhető objektum kettős jelleggel bír: egyrészt hordozza magában az eredeti „Base Object" jellegét, de valójában ez már egy új, `PSObject` típusú objektum lesz.

Ennek megértéséhez nézzük meg az alábbi példát, melyben az `Add-Member` cmdlettel bővítek ki egy objektumot újabb tulajdonsággal, vagy egyéb tagjellemzővel. Ilyen tagjellemzők lehetnek:

Tagjellemző típus	Leírás
AliasProperty	Álnév egy már meglevő tulajdonságra.
All	Minden lehetséges taglehetőség típus.
CodeMethod	Olyan metódus, amellyel hivatkozunk egy .NET osztály statikus metódusára.
CodeProperty	Olyan tulajdonság, amellyel hivatkozunk egy .NET osztály statikus tulajdonságára.
MemberSet	Tulajdonságok és metódusok halmaza egy közös néven.
Method	A PSObject alapjaként szolgáló objektumosztály egyik metódusa.
Methods	Minden metódus.
NoteProperty	Tulajdonságnév és tulajdonságérték páros, nem számított, hanem fix tag.
ParameterizedProperty	Olyan tulajdonság, ami paraméterezhető.
Properties	Minden tulajdonság.
Property	A PSObject alapjaként szolgáló objektumosztály egyik tulajdonsága.
PropertySet	Tulajdonsághalmaz.
ScriptMethod	Szkripttel megfogalmazott metódus.
ScriptProperty	Szkripttel megfogalmazott tulajdonság.

Ezek közül leggyakrabban `NoteProperty`, `ScriptProperty`, `ScriptMethod` testre szabási lehetőséget alkalmazzuk.

Akkor nézzük a példát:

```
[30] PS C:\> $a = 23
[31] PS C:\> $a.GetType().FullName
System.Int32
[32] PS C:\> $a -is [PSObject]
False
[33] PS C:\> $a | add-member -memberType Scriptproperty -Name Dupla -value
{$this*2}
[34] PS C:\> $a.dupla
[35] PS C:\> $a -is [PSObject]
True
[36] PS C:\> $a| Get-Member -MemberType scriptproperty
[37] PS C:\> $a
23
[38] PS C:\> $a.dupla
[39] PS C:\>
```

Nézzük mi történt: a [30]-ban van egy egyszerű `Int32` változóm, [32]-ben rákérdezek, hogy vajon nem `PSObject`-e a szegény? Válasz az, hogy nem.

[33]-ban hozzáadok egy szkript alapján kiszámolódó tulajdonságot `Dupla` néven. Az érték (`value`) úgy generálódik, hogy az adott objektum értékét (`$this` változó tartalmazza a szkript számára majd ezt futási időben) megszorzom 2-vel.

Ezután ki is próbálom, lekérem a [34]-ben a dupla tulajdonságot, de nem kapok vissza semmit. A [35]-ben meg is vizsgálom, és hiába lett a `$a` változóm már `PSObject`, az `add-member` a csővezeték végén nem hat vissza az `$a` változóra. Szegény PowerShell annyit csinált csak, hogy érzékelte, hogy itt az `$a` tulajdonságainak bővítése történik, ezért az automatikus típuskonverzió miatt `[PSObject]`-té tette, de ténylegesen az új tulajdonságot nem integráltuk bele az `$a`-ba. Ezért [36]-ban hiába kérdezem le, hogy mi az `$a scriptproperty`-je, nem kapok választ. És efauldis

természetesen ugyanilyen némaság a válasz a nem létező tulajdonság lekérdezésére is a [38]-ban.

Nézzük akkor meg, hogy hogyan lesz jó:

```
[40] PS C:\> $a = add-member -inputobject $a -memberType Scriptproperty -
Name Dupla -value {$this*2} -PassThru
[41] PS C:\> $a.dupla
46
[42] PS C:\> $a| Get-Member -MemberType scriptproperty


   TypeName: System.Int32

Name  MemberType     Definition
----  ----------     ----------
Dupla ScriptProperty System.Object Dupla {get=$this*2;}


[43] PS C:\> $a -is [PSObject]
True
```

[40]-ben már helyesen alkalmazom az `add-member` cmdletet. Értékadással kombinálom, `$a` vegye fel az `add-member` kimenetén kijövő értéket. De baj van! Az `add-member`-nek alaphelyzetben nincs kimenete! Ezért találták ki a `-PassThru` kapcsolót, hogy mégis legyen. Az esetek zömében ugyanis nem az alap .NET osztályok objektumait szoktuk testre szabni, hanem `[PSObject]` osztály objektumait, ott meg nem kell semmit visszaadnia az `add-member`-nek, hiszen „helyben” hozzá tudja adni a tulajdonságot, metódust. Itt viszont az új, kibővített `PSObject` objektumot vissza kell tölteni az eredeti objektumba.

[41]-ben már jól működik a `Dupla` tulajdonság, a [42]-ben meg azt látjuk, hogy ott a `ScriptProperty`-nk, és annak ellenére, hogy látjuk `$a` egy `TypeName: System.Int32`, emellett a [42]-ben azt is látjuk még `[PSObject]` is!

Nézzünk egy olyan példát is, ahol nincs szükség ennyi trükközésre, hanem egy már eleve `PSObject` objektumot módosítok. Először tehát érdemes mindig megvizsgálni, hogy egy adott objektum már eleve `PSObject`-e vagy sem, hiszen ha nem az, akkor másként kell eljárni, mintha igen.

Vegyünk egy fájlt:

```
[2] PS C:\> $f = Get-Item C:\filemembers.txt
[3] PS C:\> $f


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\


Mode            LastWriteTime       Length Name
----            -------------       ------ ----
-a---    2008.03.01.    21:09           10 filemembers.txt


[4] PS C:\> $f -is [PSObject]
True
[5] PS C:\> $f.gettype().FullName
System.IO.FileInfo
```

Látszik, hogy amellett, hogy ez egy `PSObject`, mellette még `System.IO.FileInfo` típusú objektum.

Így erre már közvetlenül alkalmazhatjuk az `add-member` cmdletet:

```
[6] PS C:\> $f | Add-Member -MemberType scriptproperty -Name Típus -Value
{if($this -is "System.IO.FileInfo"){"File"} else{"Directory"}}
[7] PS C:\> $f.Típus
File
```

A hiányossága az `add-member`-rel kibővített objektumoknak, hogy a bővítményüket „elfelejtik” mihelyst új értéket adunk nekik:

```
[11] PS C:\> $f = get-item c:\old
[12] PS C:\> $f.Típus
[13] PS C:\> $f | Add-Member -MemberType scriptproperty -Name Típus -Value
{if($this -is "System.IO.FileInfo"){"File"} else{"Directory"}}
[14] PS C:\> $f.Típus
Directory
```

A [12]-es promptban hiába kérem le újra a `Típus` tulajdonságot, nem kapok választ, csak ha újra hozzáadom ezt a tagjellemzőt.

Erre megoldást jelent magának az osztálynak (típusnak) a testre szabása, amit a következő fejezetben mutatok be.

1.5.11 Osztályok (típusok) testre szabása

Az előző fejezetben láttuk, hogy egy osztály egy konkrét objektumpéldányának hogyan adhatunk újabb tagjellemzőket. Ez nagyon jó lehetőség, csak az a baja – mint ahogy láttuk is – hogy minden újabb objektum példánynál újra létre kell hozni saját tagjellemzőinket. Milyen jó lenne, ha magát az osztályt (típust) tudnánk módosítani és akkor az adott osztály minden objektuma már eleve rendelkezne az általunk definiált tagjellemzővel. Szerencsére ezt is lehetővé teszi a PowerShell!

Az objektumtípusok a PowerShell számára egy `types.ps1xml` fájlban vannak definiálva a `C:\WINDOWS\system32\windowspowershell\v1.0` könyvtárban. Azt senki sem ajánlja, hogy ezt átszerkesszük, de hasonló fájlokat mi is készíthetünk, amelyekkel mindenféle dolgot tehetünk az objektumtípusokhoz: új tulajdonságokat, metódusokat és még azt is, hogy mely tulajdonságait mutassa meg magáról az objektum alapban. Ez utóbbi is nagyon fontos, mert engem zavart, hogy például a `get-services` cmdlet alapban miért pont a `Status`, `Name` és `DisplayName` tulajdonságokat adja ki? Hiszen van neki jó néhány egyéb tulajdonsága is:

```
PS C:\Documents and Settings\SoosTibi> Get-Service

Status   Name               DisplayName
------   ----               -----------
Stopped  Alerter            Alerter
Running  ALG                Application Layer Gateway Service
Running  AppMgmt            Application Management
Stopped  aspnet_state       ASP.NET State Service
...
```

Erre válasz ez az előbb említett types.ps1xml file. Keressük meg benne a szolgáltatások `System.ServiceProcess.ServiceController` adattípusát:

```
...
    <Type>
        <Name>System.ServiceProcess.ServiceController</Name>
        <Members>
            <MemberSet>
                <Name>PSStandardMembers</Name>
                <Members>
                    <PropertySet>
                        <Name>DefaultDisplayPropertySet</Name>
                        <ReferencedProperties>
                            <Name>Status</Name>
                            <Name>Name</Name>
                            <Name>DisplayName</Name>
                        </ReferencedProperties>
                    </PropertySet>
                </Members>
            </MemberSet>
            <AliasProperty>
                <Name>Name</Name>
                <ReferencedMemberName>ServiceName</ReferencedMemberName>
            </AliasProperty>
        </Members>
    </Type>
...
```

A kiemelésben látszik, hogy azért ezeket a tulajdonságokat mutatja meg alapban a `get-service`, mert ezek vannak `DefaultDisplayPropertySet`-ként definiálva.

Na, de minket most nem ez érdekel, hanem hogy hogyan tudok típust módosítani.

Létrehoztam egy `typemember.ps1xml` fájlt:

```
<Types>
        <Type>
                <Name>System.IO.FileInfo</Name>
                <Members>
                        <NoteProperty>
                                <Name>Tipus</Name>
                                <Value>
                                        File
                                </Value>
                        </NoteProperty>
                </Members>
        </Type>
        <Type>
                <Name>System.IO.DirectoryInfo</Name>
                <Members>
                        <NoteProperty>
                                <Name>Tipus</Name>
                                <Value>
                                        Directory
                                </Value>
                        </NoteProperty>
                </Members>
        </Type>
</Types>
```

A szerkezet magáért beszél. Definiáltam két különböző típusban is egy-egy `NoteProperty` tulajdonságot, hiszen itt magából a típusból következik, hogy fájlról vagy könyvtárról van szó, így nem kell futási időben kiszámolni semmit sem.

Most már csak be kell etetni a rendszerbe az én típusmódosításomat és már nézhetjük is az eredményt:

```
[1] PS C:\> Update-TypeData C:\powershell2\Tananyag\typemember.ps1xml
[2] PS C:\> $f = Get-Item C:\filemembers.txt
[3] PS C:\> $f.tipus
File
[4] PS C:\> $f = Get-Item C:\old
[5] PS C:\> $f.tipus
Directory
```

> Megjegyzés:
>
> Sajnos ebben az XML fájlban nem használhatunk ékezetes karaktereket, így a definiált tulajdonság nem `Típus`, hanem `Tipus` lett.

Természetesen ez csak egy kis ízelítő volt az osztályok, típusok testre szabásából, a gyakorlati részben visszatérek majd erre gyakorlatiasabb példákkal.

1.5.11.1 PSBase, PSAdapted, PSExtended

Mint ahogy az *1.5.10 Objektumok testre szabása, kiegészítése* fejezet elején bemutattam, a PowerShell igazából a .NET osztályokat nem közvetlenül kezeli, hanem néha kicsit

átalakítja annak érdekében, hogy még egységesebb, egyszerűbb, bizonyos esetekben biztonságosabb legyen ezen objektumok kezelése.

Különböző nézetek segítségével mi is láthatjuk azt, hogy milyen „csalafintaságokat” követett el ezeken az osztályokon a PowerShell:

Nézet neve	Nézet tartalma
PSBASE	A .NET-es osztály eredeti állapotban
PSADAPTED	A PowerShell által adaptált nézet (ezt látjuk alaphelyzetben)
PSEXTENDED	Csak a kibővített tagok
PSOBJECT	Magának az adapternek a nézete

Nézzünk ezekre néhány példát. Elsőként az XML adattípust mutatom, mert ott elég jól láthatóak ezen nézetek közti különbségek. Nézzük meg egy XML adat tagjellemzőit:

```
[20] PS C:\> $x = [xml] "<elem>érték<szint1><szint2>mélyadat</szint2></szint
1></elem>"
[21] PS C:\> $x | Get-Member


   TypeName: System.Xml.XmlDocument

Name                        MemberType            Definition
----                        ----------            ----------
ToString                    CodeMethod            static System.String X...
add_NodeChanged             Method                System.Void add_NodeCh...
add_NodeChanging            Method                System.Void add_NodeCh...
...
Validate                    Method                System.Void Validate(V...
WriteContentTo              Method                System.Void WriteConte...
WriteTo                     Method                System.Void WriteTo(Xm...
Item                        ParameterizedProperty System.Xml.XmlElement ...
elem                        Property              System.Xml.XmlElement ...
```

Nagyon sok jellemzője van, az egyszerűbb áttekinthetőség miatt kicsit megvágtam a közepén. A legutolsó jellemző egy Property típusú, `elem` nevű tag. Hát ilyet biztos nem tettek bele a .NET keretrendszerbe. Erről meg is győződhetünk:

```
[22] PS C:\> $x.psbase | Get-Member


   TypeName: System.Management.Automation.PSMemberSet

Name                        MemberType            Definition
----                        ----------            ----------
add_NodeChanged             Method                System.Void add_NodeCh...
add_NodeChanging            Method                System.Void add_NodeCh...
...
ChildNodes                  Property              System.Xml.XmlNodeList...
DocumentElement             Property              System.Xml.XmlElement ...
DocumentType                Property              System.Xml.XmlDocument...
FirstChild                  Property              System.Xml.XmlNode Fir...
```

```
HasChildNodes               Property             System.Boolean HasChil...
...
Value                       Property             System.String Value {g...
XmlResolver                 Property             System.Xml.XmlResolver...
```

A fenti listában tényleg nincs `elem` nevű tulajdonság. Miért tették bele ezt az elem tulajdonságot vajon a PowerShell alkotói? Azért, hogy egyszerűen lehessen hivatkozni az XML adathalmaz különböző elemeire, hiszen az XML egy hierarchikus felépítésű adattípus, így könnyen adódik az ötlet az ilyen jellegű hivatkozási lehetőségre:

```
[23] PS C:\> $x.elem

#text                                szint1
-----                                ------
érték                                szint1


[24] PS C:\> $x.elem.szint1

szint2
------
mélyadat


[25] PS C:\> $x.elem.szint1.szint2
mélyadat
```

Nézzük, hogy a PS1XML fájlban történt-e típusbővítés az XML típus esetében?

```
[53] PS C:\> $x.psextended | Get-Member


   TypeName: System.Management.Automation.PSMemberSet

Name     MemberType Definition
----     ---------- ----------
ToString CodeMethod static System.String XmlNode(PSObject instance)
```

Egyetlen egy `CodeMethod` lett csak definiálva, amellyel sztringgé alakíthatjuk az XML adatot.

Nézzük meg az általam létrehozott `Tipus` tulajdonságot a fájl és könyvtár objektumoknál:

```
[27] PS C:\> $fo = Get-Item C:\powershell2\demo\demo1.ps1
[28] PS C:\> $fo.Tipus
File
[29] PS C:\> $fo.psextended | Get-Member


   TypeName: System.Management.Automation.PSMemberSet

Name          MemberType     Definition
```

```
----            ----------     ----------
PSChildName     NoteProperty   System.String PSChildName=demo1.ps1
PSDrive         NoteProperty   System.Management.Automation.PSDriveInfo PS...
PSIsContainer   NoteProperty   System.Boolean PSIsContainer=False
PSParentPath    NoteProperty   System.String PSParentPath=Microsoft.PowerS...
PSPath          NoteProperty   System.String PSPath=Microsoft.PowerShell.C...
PSProvider      NoteProperty   System.Management.Automation.ProviderInfo P...
Tipus           NoteProperty   System.String Tipus=File
BaseName        ScriptProperty System.Object BaseName {get=[System.IO.Path...
Mode            ScriptProperty System.Object Mode {get=$catr = "";...
ReparsePoint    ScriptProperty System.Object ReparsePoint {get=if($this.At...
```

Ott látható a listában az általam, a PS1XML fájlon keresztül történt típusbővítésnek a nyoma.

Nézzünk még egy példát a `PSBase` nézet használatára:

```
[30] PS C:\> $user = [ADSI] "WinNT://asus/administrator"
[31] PS C:\> $user.name
Administrator
[32] PS C:\> $user | Get-Member


   TypeName: System.DirectoryServices.DirectoryEntry

Name                        MemberType Definition
----                        ---------- ----------
AutoUnlockInterval          Property   System.DirectoryServices.PropertyV...
BadPasswordAttempts         Property   System.DirectoryServices.PropertyV...
Description                 Property   System.DirectoryServices.PropertyV...
FullName                    Property   System.DirectoryServices.PropertyV...
HomeDirDrive                Property   System.DirectoryServices.PropertyV...
HomeDirectory               Property   System.DirectoryServices.PropertyV...
LastLogin                   Property   System.DirectoryServices.PropertyV...
LockoutObservationInterval  Property   System.DirectoryServices.PropertyV...
LoginHours                  Property   System.DirectoryServices.PropertyV...
LoginScript                 Property   System.DirectoryServices.PropertyV...
MaxBadPasswordsAllowed      Property   System.DirectoryServices.PropertyV...
MaxPasswordAge              Property   System.DirectoryServices.PropertyV...
MaxStorage                  Property   System.DirectoryServices.PropertyV...
MinPasswordAge              Property   System.DirectoryServices.PropertyV...
MinPasswordLength           Property   System.DirectoryServices.PropertyV...
Name                        Property   System.DirectoryServices.PropertyV...
objectSid                   Property   System.DirectoryServices.PropertyV...
Parameters                  Property   System.DirectoryServices.PropertyV...
PasswordAge                 Property   System.DirectoryServices.PropertyV...
PasswordExpired             Property   System.DirectoryServices.PropertyV...
PasswordHistoryLength       Property   System.DirectoryServices.PropertyV...
PrimaryGroupID              Property   System.DirectoryServices.PropertyV...
Profile                     Property   System.DirectoryServices.PropertyV...
UserFlags                   Property   System.DirectoryServices.PropertyV...
```

Egy helyi felhasználót betöltöttem a `$user` nevű változómba, szépen le is tudtam kérdezni a nevét. Majd amikor kilistázom a felhasználóm tagjellemzőit, meglepődve lát-

hatjuk, hogy nincs köztük egy metódus sem! Márpedig nehezen hihető el, hogy tényleg semmit nem tud egy felhasználói fiók magával kezdeni. Nézzünk az objektumunk mögé:

```
[34] PS C:\> $user.psbase | Get-Member


   TypeName: System.Management.Automation.PSMemberSet

Name                      MemberType Definition
----                      ---------- ----------
add_Disposed              Method     System.Void add_Disposed(EventHandl...
...
get Parent                Method     System.DirectoryServices.DirectoryE...
get Path                  Method     System.String get Path()
get Properties            Method     System.DirectoryServices.PropertyCo...
...
Rename                    Method     System.Void Rename(String newName)
set_AuthenticationType    Method     System.Void set_AuthenticationType(...
set ObjectSecurity        Method     System.Void set ObjectSecurity(Acti...
set Password              Method     System.Void set Password(String value)
set Path                  Method     System.Void set Path(String value)
set_Site                  Method     System.Void set_Site(ISite value)
set_UsePropertyCache      Method     System.Void set_UsePropertyCache(Bo...
set_Username              Method     System.Void set_Username(String value)
ToString                  Method     System.String ToString()
AuthenticationType        Property   System.DirectoryServices.Authentica...
Children                  Property   System.DirectoryServices.DirectoryE...
Container                 Property   System.ComponentModel.IContainer Co...
Guid                      Property   System.Guid Guid {get;}
Name                      Property   System.String Name {get;}
NativeGuid                Property   System.String NativeGuid {get;}
NativeObject              Property   System.Object NativeObject {get;}
ObjectSecurity            Property   System.DirectoryServices.ActiveDire...
Options                   Property   System.DirectoryServices.DirectoryE...
Parent                    Property   System.DirectoryServices.DirectoryE...
Password                  Property   System.String Password {set;}
Path                      Property   System.String Path {get;set;}
Properties                Property   System.DirectoryServices.PropertyCo...
SchemaClassName           Property   System.String SchemaClassName {get;}
SchemaEntry               Property   System.DirectoryServices.DirectoryE...
Site                      Property   System.ComponentModel.ISite Site {g...
UsePropertyCache          Property   System.Boolean UsePropertyCache {ge...
Username                  Property   System.String Username {get;set;}
```

Hoppá! Mindjárt más a helyzet. A metódusok zöme természetesen tartományi környezetben használható, de például a `rename()` vagy a `set_password()` metódus helyi gépen is praktikus szolgáltatás.

Vajon ezek a metódusok miért nincsenek alaphelyzetben adaptálva a PowerShell környezetre? Erre az igazi választ nem tudom, valószínű ez egy biztonsági megfontolás volt, hogy a szkriptelők ne írogassanak felelőtlenül olyan szkripteket, amelyekkel a felhasználói objektumokat módosítanak. Vagy az is lehet a magyarázat, hogy egy másik interfészt szánt volna igazából a Microsoft a felhasználó menedzsment céljaira, amin keresztül módosítani lehetett volna, de ez az 1.0-ás PowerShell verzióba már nem fért bele.

1.5.12 Objektumok mentése, visszatöltése

A PowerShell lehetőséget biztosít objektumok elmentésére és visszatöltésére. Ez nagyon praktikus, hiszen ha épp a konzolon a változóim jól fel vannak töltve mindenféle objektummal és nekem valami miatt be kell csuknom a PowerShell ablakot, akkor a változóim törlődnek és legközelebb újra elő kellene állítanom őket.

Az `Export-CliXML` cmdlettel ki tudom ezeket menteni egy fájlba, bezárhatom a PowerShell ablakot, majd később vissza tudom importálni a változóim értékét az `Import-CliXML` cmdlettel.

Nézzünk erre egy példát:

```
[13] PS C:\> $p = Get-Process w*
[14] PS C:\> $p

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    123       7     2836       7924    74     0,09   2464 WindowsSearch
    655     159    10748       2976    97    11,77    440 winlogon
    700      49    40684      78780   280    50,63   2316 WINWORD
    215      12     4708       8584    79     0,11   2196 WLLoginProxy
    161       7     2704       7728    39     0,19   1812 wmiprvse
     30       3     1020       3088    50     0,02    504 wscntfy


[15] PS C:\> $p | get-member


   TypeName: System.Diagnostics.Process

Name                           MemberType     Definition
----                           ----------     ----------
Handles                        AliasProperty  Handles = Handlecount
Name                           AliasProperty  Name = ProcessName
NPM                            AliasProperty  NPM = NonpagedSystemMemorySize
PM                             AliasProperty  PM = PagedMemorySize
VM                             AliasProperty  VM = VirtualMemorySize
WS                             AliasProperty  WS = WorkingSet
add_Disposed                   Method         System.Void
add_Disposed(Event...
...

[16] PS C:\> $p | Export-Clixml c:\export.xml
[17] PS C:\> Clear-Variable p
[18] PS C:\> $p
[19] PS C:\> $p = Import-Clixml C:\export.xml
[20] PS C:\> $p

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    123       7     2836       7924    74     0,09   2464 WindowsSearch
    655     159    10748       2976    97    11,77    440 winlogon
    700      49    40684      78780   280    50,63   2316 WINWORD
    215      12     4708       8584    79     0,11   2196 WLLoginProxy
    161       7     2704       7728    39     0,19   1812 wmiprvse
```

```
     30       3     1020       3088    50     0,02    504 wscntfy

[21] PS C:\> $p | get-member


   TypeName: Deserialized.System.Diagnostics.Process

Name                        MemberType     Definition
----                        ----------     ----------
Handles                     AliasProperty  Handles = Handlecount
Name                        AliasProperty  Name = ProcessName
NPM                         AliasProperty  NPM = NonpagedSystemMemorySize
PM                          AliasProperty  PM = PagedMemorySize
VM                          AliasProperty  VM = VirtualMemorySize
WS                          AliasProperty  WS = WorkingSet
__NounName                  NoteProperty   System.String __NounName=Process
BasePriority                Property       System.Int32 {get;set;}
Container                   Property        {get;set;}
EnableRaisingEvents         Property       System.Boolean {get;set;}
...
```

A fenti példában a `$p` változó megkapta a W-vel kezdődő processz-objektumok tömbjét. A [15]-ben látszik, hogy ez tényleg jól nevelt `System.Diagnostics.Process` objektumokat tartalmaz, a rájuk jellemző tulajdonságokkal és metódusokkal együtt.

Ezután kiexportáltam `$p`-t, töröltem majd visszaimportáltam. [20]-ban megnézem, mi van `$p`-ben, szemre teljesen ugyanaz, mint korábban. Azonban a [21] `get-member`-je felfedi, hogy azért ez mégsem 100%-osan ugyanaz az objektum, hiszen elveszítette a metódusait és típusa is más lett:

`Deserialized.System.Diagnostics.Process`.

Mindenesetre, ha a statikus adataival szeretnénk csak foglalkozni ez bőven elég, éles, „real time” adatokkal meg úgysem biztos, hogy tudnánk dolgozni, hiszen az export óta lehet, hogy megszűntek már a korábban futó processzek.

1.6 Operátorok

Minden programnyelv alapvető elemei az operátorok. A PowerShell alkotói az operátorok esetében is arra törekedtek, hogy minél „hatásosabbak” legyenek, azaz a lehető legtömörebb kifejezésekkel lehessen a legtöbb eredményt elérni.

Ebben a fejezetben a PowerShell operátorait mutatom be.

1.6.1 Aritmetikai operátorok

A legtriviálisabb operátorok – a kifejezés eredeti jelentése alapján – valós számokon végezhető műveletekhez kötődik. A PowerShell ezen jóval túlmutat, a matematikai tanulmányainkban megszokott műveletek ki vannak terjesztve egyéb objektumokra: többek között a tömbökre, sztringekre, fájlokra.

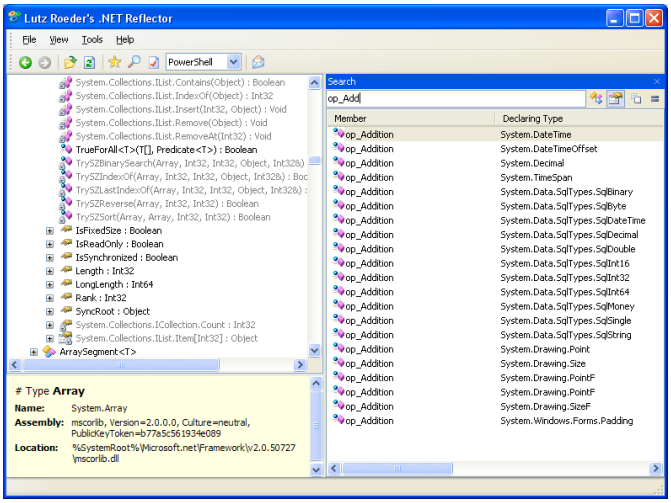
1.6.1.1 Összeadás

Rögtön nézzünk is példákat az összeadásra:

```
[1] PS C:\> 5+4
9
[2] PS C:\> "ab" + "lak"
ablak
[3] PS C:\> "egy", "kettő", "három" + "négy", "öt"
egy
kettő
három
négy
öt
```

Látszik, hogy akár az egészekre, sztringekre vagy a tömbökre is értelmezett az összeadás. Ebben jelentős szerepe van a .NET Framework osztálydefinícióinak, hiszen valójában az ő érdemük, az ő megfelelő összeadást végző metódusuké, hogy mi is történik ténylegesen az összeadás hatására. A PowerShell „csak” abban ludas, hogy megtalálja az adott metódust és átalakítsa a kifejezést, esetlegesen típuskonverziót végezzen az operandusok között.

A Reflector programmal ezeket meg is tudjuk nézni, a számoknál az `op_Addition` metódust kell keresni:



34. ábra Az összeadás metódusainak felderítése a Reflectorral

A sztringeknél talán az `Append` metódust, de ezzel nekünk nem kell törődni, ezt a PowerShell helyettünk elvégzi.

Ezen kívül még dátumokat és esetlegesen hashtáblákat szoktunk még összeadni:

```
[4] PS I:\>$hash1 = @{Első = 1; Második = 2}
[5] PS I:\>$hash2 = @{Harmadik = 3; Negyedik = 4}
[6] PS I:\>$hash1 + $hash2

Name                           Value
----                           -----
Második                        2
Harmadik                       3
Negyedik                       4
Első                           1
```

Dátumoknál kicsit zűrösebb a helyzet:

```
[12] PS I:\>get-date

2008. március 11. 9:53:26


[13] PS I:\>(get-date) + 5000000000000

2008. március 17. 4:47:00
```

Jó nagy számot kellett a dátumhoz adnom, hogy valami látható legyen az eredményen, mert valójában a dátum u.n. „tick”-ekben, „ketyegésekben” számolódik, ami 100 ms-os felbontású időtárolást tesz lehetővé.

Épp ezért a dátumoknál már eleve van mindenféle praktikusabb `Add…` metódus:

```
[18] PS I:\>get-date | Get-Member


   TypeName: System.DateTime

Name                 MemberType     Definition
----                 ----------     ----------
Add                  Method         System.DateTime Add(TimeSpan value)
AddDays              Method         System.DateTime AddDays(Double value)
AddHours             Method         System.DateTime AddHours(Double value)
AddMilliseconds      Method         System.DateTime AddMilliseconds(Double
v...
AddMinutes           Method         System.DateTime AddMinutes(Double value)
AddMonths            Method         System.DateTime AddMonths(Int32 months)
AddSeconds           Method         System.DateTime AddSeconds(Double value)
AddTicks             Method         System.DateTime AddTicks(Int64 value)
AddYears             Method         System.DateTime AddYears(Int32 value)
...

[19] PS I:\>get-date

2008. március 11. 9:57:55


[20] PS I:\>(get-date).AddDays(5)

2008. március 16. 9:58:10
```

A fenti példában először kilistáztam a `get-date` által visszaadott `[DateTime]` típus metódusait, majd a [20]-as sorban 5 napot adtam az aktuális dátumhoz.

1.6.1.2 Többszörözés

Az összeadáshoz hasonlóan a szorzás is jelentős általánosításon esett át:

```
[10] PS C:\> 6*7
42
[11] PS C:\> "w"*4
wwww
[12] PS C:\> ("BO","CI")*3
BO
CI
BO
CI
BO
CI
[13] PS C:\> "BO","CI"*3
BO
CI
BO
CI
BO
CI
[14] PS I:\>("BO","CI"*3).length
```

```
6
```

Láthatjuk, hogy a szöveget is lehet szorozni, ez annyiszor teszi össze a szöveget, ahány a szorzásjel után áll.

Tömbnél is hasonlóan többszörözi a tagokat. A [13]-as promptnál látszik, hogy nem is kell zárójelet használni! Ez a PowerShell alkotóinak az érdeme.

Megjegyzés

Érdekes módon a [14]-es promptban látszik, hogy a kéttagú tömb háromszorozása után nem háromelemű lett (3 darab kételemű), hanem hatelemű!

Nézzük meg, hogy mi történik, ha felcseréljük a szorzás operandusit:

```
[15] PS C:\> 3*"BO","CI"
Cannot convert "System.Object[]" to "System.Int32".
At line:1 char:3
+ 3*" <<<< BO","CI"
[16] PS C:\> 3*("BO","CI")
Cannot convert "System.Object[]" to "System.Int32".
At line:1 char:3
+ 3*( <<<< "BO","CI")
```

Azt láthatjuk, hogy nem mindegy, hol van a szorzó és a tényező. A szorzónak kell mindig jobb oldalon állnia, ha nem, akkor hibát kaphatunk, mint ahogy az a [15]-es és [16]-ös sor után is kaptunk, hiszen a PowerShell az automatikus típuskonverziót balról jobbra végzi, azaz az [int] típust veszi alapnak, és ehhez próbálja igazítani a szorzás következő tényezőit, ami itt – szövegek és kételemű tömb esetén - nem sikerült.

1.6.1.3 Osztás, maradékos osztás

Természetesen osztás is van a PowerShellben:

```
[32] PS I:\>15/5
3
[33] PS I:\>(15/5).GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int32                                    System.ValueType


[34] PS I:\>15/6
2,5
[35] PS I:\>(15/6).GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Double                                   System.ValueType
```

```
[36] PS I:\>2,4/1,2
Method invocation failed because [System.Object[]] doesn't contain a method
nam
ed 'op Division'.
At line:1 char:5
+ 2,4/1 <<<< ,2

[37] PS I:\>(15/6)/(125/100)
2
```

A [32]-es prompt egyértelmű, 15-öt elosztunk 5-tel, 3-at kapunk, nem is akármilyen 3-at: `Int32` típusút! Szemben a 15/6-tal, aminek eredménye természetesen [35]-ben `Double`.

Egy kicsit időzzünk el itt! Hogyan is néz ki a [34] eredményeként kapott `2,5`? A tizedes elválasztására vesszőt használt a PowerShell! Viszont a [36]-ban nem szerette ezt, mert a `(,)` általában tömb jelzésére szolgáló operátor. Akkor vajon fog tudni dolgozni ezzel az általa kiadott `2,5`-gyel? A [37]-ben azt látjuk, hogy természetesen igen! És itt jön ki a PowerShell objektumalapúságának nagy előnye! Annak ellenére, hogy a megjelenítésnél használja az aktuális gép területi beállításait, és ennek megfelelően jeleníti meg a számokat, amikor ő számol tovább egy objektummal, akkor területi beállítás-mentesen kezeli ezt. Ha sztring alapú lenne a PowerShell, hasonlóan a Linuxos/Unixos shellekhez, akkor vagy a kimenet nem venné figyelembe a területi beállításokat, vagy egy csomó sztring-hókuszpókuszt kellene elvégezni ahhoz, hogy ilyen egyszerű műveletek működhessenek. Ha mi adunk be számokat a parancssor számára, akkor mindig ponttal jelezzük a tizedesjelet, nem a területi beállítás alapján kell beírni a számokat.

Ebbe a fejezetbe tartozik még a maradékos osztás művelete, a %:

```
[41] PS I:\>21%5
1
[42] PS I:\>1.4%1.1
0,3
```

Ezzel nincs sok trükk, hiszen ez az operátor csak számokra van értelmezve, viszont nem csak egészekre, ahogy ez a [42]-ben látszik.

1.6.2 Értékadás

Értéket már számtalanszor adtam, így az alapeset nem fog senkinek meglepetést okozni:

```
[49] PS I:\>$a=5
[50] PS I:\>$a
5
```

Azonban nem csak ilyen módon lehet értéket adni.

```
[51] PS I:\>$a+=8
[52] PS I:\>$a
13
[53] PS I:\>$a-=4
[54] PS I:\>$a
9
```

Az [51]-től kezdődően látszik, hogy ha ugyanannak a változónak az aktuális értékével szeretnénk műveletet végezni és utána ezt visszatölteni a változónkba, akkor ennek tömör formáját is alkalmazhatjuk az adott műveleti jel és az egyenlőségjel kombinációjával.

Nézzük a trükkösebb formákat! Ha ugyanolyan értéket szeretnék adni több változónak is, akkor ezt tömören is megtehetem:

```
[59] PS I:\>$x=$y=$z=81
[60] PS I:\>$x
81
[61] PS I:\>$y
81
[62] PS I:\>$z
81
```

Vigyázzunk a vesszővel!

```
[63] PS I:\>$x, $y, $z, $w = 1,2,3
[64] PS I:\>$x
1
[65] PS I:\>$y
2
[66] PS I:\>$z
3
[67] PS I:\>$w
[68] PS I:\>
```

A [63]-as sorban vesszővel soroltam fel a változóimat, amelyeknek szintén vesszővel elválasztott értékeket adok. Ahelyett, hogy mind a négy változóm felvenné a háromelemű tömböt értékként, az első változóm megkapta az első tagot, a második a másodikat és így tovább. Szegény `$w`-nek nem jutott érték, így ő üres maradt.

Ha fordított a helyzet, azaz a változók vannak kevesebben, mint az értékek, akkor az utolsó megkapja a maradékot egy tömbként:

```
[69] PS I:\>$x, $y = 1,2,3
[70] PS I:\>$y
2
3
```

Ez a változók felsorolásának lehetősége jól jöhet szövegek feldolgozásakor.

```
[71] PS I:\>$sor = "Soós Tibor Budapest"
[72] PS I:\>$vezetéknév, $keresztnév, $város = $sor.Split()
[73] PS I:\>$vezetéknév
Soós
```

```
[74] PS I:\>$keresztnév
Tibor
[75] PS I:\>$város
Budapest
```

A [72]-ben látható `Split()` metódus a sztringek gyakran felhasznált metódusa, ami a paramétereként átadott karakter mentén feldarabolja a sztringet. Ha üresen hagyjuk a paraméterét (mint most), akkor a normál szóelválasztó karakterek mentén tördel. Kimeneteként az így széttördelt sztringekből álló tömböt adja vissza, amit szépen betöltök a változóimba.

1.6.3 Összehasonlító operátorok

Az összehasonlítás műveletét végző összehasonlító operátorok jelzésére a PowerShell alkotói nem a hagyományos jeleket (=, <, >, <>, !=, stb.) választották, hanem külön jelölést vezettek be, rögtön két különböző szériát a kis-nagybetű érzéketlen és érzékeny változatra:

Operátor	Leírás
-eq	egyenlő
-ne	nem egyenlő
-gt	nagyobb
-ge	nagyobb egyenlő
-lt	kisebb
-le	kisebb egyenlő

Kis-nagybetű érzékeny operátor	Leírás
-ceq	egyenlő
-cne	nem egyenlő
-cgt	nagyobb
-cge	nagyobb egyenlő
-clt	kisebb
-cle	kisebb egyenlő

Megjegyzés

„Érzéketlen” változatként használhatjuk az operátorok „i” betűs megfelelőit is: `-ieq`, `-ine`, `-igt`, stb., de ezek az alap változattal teljesen ekvivalensen működnek.

Értelemszerűen az „érzékeny” változatot elsődlegesen szövegek összehasonlításakor használjuk, de a számoknál se ad hibát alkalmazásuk. Nézzünk akkor néhány példát mindezek alkalmazására:

```
[79] PS I:\>1 -ceq 1
True
[80] PS I:\>1 -eq 2
False
[81] PS I:\>1 -eq 1
True
[82] PS I:\>1 -ceq 1
True
[83] PS I:\>"ablak" -eq "ABLAK"
```

```
True
[84] PS I:\>"ablak" -ceq "ABLAK"
False
[85] PS I:\>5 -gt 2
True
```

Sokat azt hiszem nem is kell magyarázni.

Megjegyzés

A kis-nagybetű érzékenységnek van egy speciális esete, amikor nem sztringeket, hanem karaktereket hasonlítunk össze:

```
[36] PS C:\> "a" -eq "A"
True
[37] PS C:\> [char] "a" -eq "A"
False
[38] PS C:\> [char] "a" -like "A"
True
```

A [36] az alapeset sztringgel, a [37]-ben karaktereket hasonlítok össze, ott már számít a kis-nagybetű. Ha ezt nem tekintjük különbözőnek, akkor használhatjuk a `-like` operátort is (lásd később).

Talán azt érdemes megnézni, hogy a szövegeket hogyan tudom „nagyság” szempontjából összehasonlítani:

```
[86] PS I:\>"ablak" -gt "ajtó"
False
[87] PS I:\>"baba" -gt "ablak"
True
[88] PS I:\>"ablak" -lt "állat"
True
[89] PS I:\>"asztal" -lt "állat"
False
[90] PS I:\>"á" -lt "a"
False
[91] PS I:\>"ab" -cgt "Ab"
False
```

Látszik, hogy itt a területi beállításoknak megfelelő szótár-sorrend alapján dől el, hogy melyik sztring nagyobb a másiknál.

Megjegyzés

Némi „bug” van a dologban:

```
[92] PS I:\>"áb" -gt "ab"
True
[93] PS I:\>"ác" -gt "ab"
True
[94] PS I:\>"áb" -gt "ac"
```

```
False
```

Itt a [94]-es sorban helytelenül mondja, hogy nem nagyobb az „áb” az „ac”-tól. Ez a Windows XP-m magyar nyelvi beállításának sorba rendezési alapbeállításának köszönhető. Ha ezt átváltjuk „technical”-ra, akkor helyes, szótár szerinti összehasonlítást és sorrendet fogunk kapni.

Vigyázat! A PowerShell összehasonlítás során is végez automatikus típuskonverziót:

```
[1] PS I:\>"1234" -eq 1234
True
[2] PS I:\>1234 -eq "1234"
True
```

Persze nem mindenkor tud okos lenni, ha nagyon akarjuk, akkor átverhetjük:

```
[3] PS I:\>"0123" -eq 123
False
```

Ugye itt a 0-val kezdődő, idézőjelek közé tett szám esetén azt tartja valószínűbbnek, hogy ez szöveg, és akkor a jobb oldalon levő részt is szöveggé konvertálja magában, márpedig a „0123” nem egyenlő „123”-mal.

Nézzük meg, hogy vajon tömbök esetében hogyan működnek ezek az összehasonlító operátorok?

```
[10] PS I:\>1,2,3,4,1,2,3,1,2 -eq 1
1
1
1
[11] PS I:\>1,2,3,4,1,2,3,1,2 -eq 2
2
2
2
[12] PS I:\>1,2,3,4,1,2,3,1,2 -lt 3
1
2
1
2
1
[13] PS I:\>1,2,3,4,1,2,3,1,2 -eq 4
4
[14] PS I:\>1,2,3,4,1,2,3,1,2 -eq 5
```

Hoppá! Az első, amit láthatunk tömbök esetében, hogy nem `True` vagy `False` értéket kapunk, hanem azokat az elemeket, amelyekre igaz az adott összehasonlító művelet. Sőt! Ahogy a [14]-es promptban is látható, ha nincs egyezés, akkor sem kapunk `False`-t, hanem „semmi” a válasz.

Ha a jobb oldalon van tömb, akkor nem igazán kapunk egyenlőséget semmilyen esetben sem:

```
[19] PS I:\>(1,2),3 -eq 1,2
[20] PS I:\>1,2 -eq 1,2
[21] PS I:\>1 -eq 1,2
False
```

1.6.4 Tartalmaz (-contains, -notcontains)

Az előző fejezetben azt láthattuk, hogy az összehasonlító `-eq` elég furcsa módon működik tömbök esetében. Ezért, hogy ha csak arra vagyunk kíváncsiak, hogy egy tömb tartalmaz-e valamilyen elemet, akkor erre a `-contains` operátor használható, illetve ennek negáltja, a `-notcontains`:

```
[24] PS I:\>1,2,3,4,1,2,3,1,2 -contains 3
True
[25] PS I:\>1,2,3,4,1,2,3,1,2 -notcontains 5
True
```

Sajnos itt is vigyázni kell, ha a jobb oldalon tömb áll, mert nem ad találatot:

```
[26] PS I:\>(1,2),3 -contains (1,2)
False
```

1.6.5 Dzsóker-minták (-like)

Ha valaki a számítógépekkel kezd el foglalkozni, azon belül a fájlokkal, akkor viszonylag hamar találkozik a (*) karakterrel, mint dzsoli-dzsókerrel. A PowerShellben nagyon kiterjedt lehetőségeket adnak az ilyen jellegű dzsoli-dzsókerek, akár a hagyományosnak tűnő „DOS” parancsokkal is:

```
[2] PS C:\scripts> dir


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\scripts


Mode            LastWriteTime       Length Name
----            -------------       ------ ----
d----   2008.02.22.    22:47        <DIR> EntryForm
-a---   2008.01.15.    20:21          709 alice.txt
-a---   2008.01.11.    11:36          235 coffee.txt
-a---   2008.02.12.    11:07          382 DebugMe.pl
-a---   2008.02.12.    11:07          253 DebugMe.ps1
-a---   2008.02.12.    11:06          823 DebugMe.vbs
-a---   2008.02.08.    20:47           32 lettercase.txt
-a---   2008.02.08.    20:55           22 numbers.txt
-a---   2008.02.11.     8:36        42496 Password Checklist.doc
-a---   2008.01.14.     8:16       229376 pool.mdb
-a---   2008.02.09.    21:15          726 presidents.txt
-a---   2008.02.12.    11:34         1760 readme.txt
```

```
-a---   2008.02.05.    13:52         3366 skaters.txt
-a---   2007.01.03.     8:00         2139 songlist.csv
-a---   2008.02.08.    20:48           80 symbols.txt
-a---   2008.02.08.    20:46           46 vertical.txt
-a---   2007.01.03.     8:00        60358 votes.txt
-a---   2007.01.03.     8:00       328620 wordlist.txt


[3] PS C:\scripts> dir *.txt


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\scripts


Mode           LastWriteTime       Length Name
----           -------------       ------ ----
-a---   2008.01.15.    20:21          709 alice.txt
-a---   2008.01.11.    11:36          235 coffee.txt
-a---   2008.02.08.    20:47           32 lettercase.txt
-a---   2008.02.08.    20:55           22 numbers.txt
-a---   2008.02.09.    21:15          726 presidents.txt
-a---   2008.02.12.    11:34         1760 readme.txt
-a---   2008.02.05.    13:52         3366 skaters.txt
-a---   2008.02.08.    20:48           80 symbols.txt
-a---   2008.02.08.    20:46           46 vertical.txt
-a---   2007.01.03.     8:00        60358 votes.txt
-a---   2007.01.03.     8:00       328620 wordlist.txt


[4] PS C:\scripts> dir s*.txt


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\scripts


Mode           LastWriteTime       Length Name
----           -------------       ------ ----
-a---   2008.02.05.    13:52         3366 skaters.txt
-a---   2008.02.08.    20:48           80 symbols.txt


[5] PS C:\scripts> dir [ad]*.txt


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\scripts


Mode           LastWriteTime       Length Name
----           -------------       ------ ----
-a---   2008.01.15.    20:21          709 alice.txt


[6] PS C:\scripts> dir [a-r]*.txt


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\scripts
```

```
Mode           LastWriteTime       Length Name
----           -------------       ------ ----
-a---   2008.01.15.    20:21          709 alice.txt
-a---   2008.01.11.    11:36          235 coffee.txt
-a---   2008.02.08.    20:47           32 lettercase.txt
-a---   2008.02.08.    20:55           22 numbers.txt
-a---   2008.02.09.    21:15          726 presidents.txt
-a---   2008.02.12.    11:34         1760 readme.txt


[7] PS C:\scripts> dir ?o*.txt


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\scripts


Mode           LastWriteTime       Length Name
----           -------------       ------ ----
-a---   2008.01.11.    11:36          235 coffee.txt
-a---   2007.01.03.     8:00        60358 votes.txt
-a---   2007.01.03.     8:00       328620 wordlist.txt
```

Azt hiszem, a fenti példák magukért beszélnek. Kiemelném az [a-r] formulát, tehát ez az „a” és „r” közti összes betűt helyettesíti.

Ilyen jellegű „dzsókeres” kifejezéseket lehet használni a `-like`, `-notlike` vizsgálatokkal, és a kis-nagybetű érzékeny változataikkal: `-clike`, `-cnotlike`:

```
[11] PS C:\scripts> "ablak" -like "[a-f]lak"
False
[12] PS C:\scripts> "ablak" -like "[a-f]blak"
True
[13] PS C:\scripts> "ablak" -clike "[A-F]blak"
False
[14] PS C:\scripts> "blak" -clike "[A-F]blak"
False
[15] PS C:\scripts> "blak" -like "[a-f]blak"
False
```

Hasonlóan bánik el a tömbökkel is a `-like` és a többi dzsókeres operátorunk, mint ahogyan az `-eq` is:

```
[17] PS C:\scripts> "ablak", "abrosz", "alabástrom", "baba" -like "ab[l-r]*"
ablak
abrosz
```

Természetesen dzsókeres kifejezés csak az operátor jobb oldalán állhat.

1.6.6 Regex (-match, -replace)

A dzsókerek nagyon praktikusak, de egy csomó mindenre nem jók. Például szeretnénk megvizsgálni, hogy egy szöveg e-mail cím formátumú-e? Vagy van-e a szövegben valahol egy webcím, vagy telefonszám? Mivel ilyen jellegű vizsgálatok nagyon gyakoriak a számítástechnikában, ezért erre külön „tudományág” alakult, matematikai alapját pedig a formális nyelvek képezik.

Az ilyen jellegű szövegminta vizsgálatok kifejezéseit hívják Regular Expression-nek, vagy röviden Regex-nek. Miután ez ténylegesen külön tudomány, Regular Expression kifejezésekről számtalan könyv látott napvilágot, ezért itt most nem vállalkozom arra, hogy még csak közelítő részletességgel tárgyaljam ezt a témát, de azért a főbb elveket és jelöléseket megpróbálom bemutatni.

1.6.6.1 Van-e benne vagy nincs?

Elsőként nézzünk a legegyszerűbb esetet, amikor egy szövegről el akarjuk dönteni, hogy van-e benne általunk keresett mintára hasonlító szövegrész:

```
[12] PS C:\> "Benne van a NAP szó" -match "nap"
True
[13] PS C:\> "Benne van a NAPfény szó" -match "nap"
True
```

A fenti két példa még nem igazán mutatja meg a regex erejét, akár a `-like` operátor is használható lett volna:

```
[14] PS C:\> "Benne van a NAP szó" -like "*nap*"
True
```

De vajon mit tennénk, ha a „napfény”-t nem tekintenénk a mintánkkal egyezőnek, csak az önálló „nap”-ot? Az önállóságot persze sokfajta karakter jelölheti: szóköz, pont, zárójel, stb. A regex-szel ez nagyon egyszerű, használni kell a „szóvég” jelölő „\b” karakterosztály szimbólumot a mintában:

```
[15] PS C:\> "Benne van a NAPfény szó" -match "\bnap\b"
False
[16] PS C:\> "Benne van a NAP. Fény is" -match "\bnap\b"
True
[17] PS C:\> "Benne van a (NAP)" -match "\bnap\b"
True
```

Azaz keresek egy olyan mintát, amely áll egy szóelválasztó karakterből, a „nap”-ból, és megint egy szóelválasztó karakterből.

Megjegyzés:

Mint ahogy láttuk, egy regularáris kifejezésben a „\b” szóhatárt jelöl, de ha ezt szögletes zárójelek között „[]” alkalmazzuk (lásd később a karakterosztályokat), akkor a „\b” a visszatörlés, azaz a „backspace” karaktert jelenti. A cserénél (`-replace`, lásd szintén később) a „\b” mindig visszatörlés karakter jelent.

Vagy keresem, hogy van-e a szövegben szám:

```
[26] PS C:\> "Ebben van 1 szám" -match "\d"
True
```

A \d a számjegy jelölője. Ezen jelölőknek van „negáltjuk” is, \B – a nem szóelválasztó karakter, \D – a nem számjegy.

Vagy keresem a Soós nevűeket, akik lehetnek „Sós”-ok is, de azért „Sooós” ne legyen:

```
[23] PS C:\> "Soós" -match "So?ós"
True
[24] PS C:\> "Sós" -match "So?ós"
True
[25] PS C:\> "Sooós" -match "So?ós"
False
```

A „?” 0 vagy 1 darabot jelöl az azt megelőző regex kifejezésből. A * bármennyi darabot jelent, akár 0-t is. A + legalább 1 darabot jelöl. Tetszőleges darabszámot jelölhetünk {*min,max*} formában, ahol a maximum akár el is hagyható.

Kereshetem az „s”-sel kezdődő, „s”-sel végződő háromkarakteres szavakat:

```
[26] PS C:\> "Ez egy jó találat 'sas' nekem" -match "\bs.s\b"
True
[27] PS C:\> "Ez nem jó találat 'saras' nekem" -match "\bs.s\b"
False
```

Alaphelyzetben a „.” minden karaktert helyettesít, kivéve a sortörést.

Egyszerűen vizsgálhatom, hogy van-e a szövegben ékezetes betű:

```
[28] PS C:\> "Ékezet van ebben több is" -match "[áéíóöőüű]"
True
[29] PS C:\> "Ekezetmentes" -match "[áéíóöőüű]"
False
```

A [] zárójelpár közti betűk vagylagosan kerülnek vizsgálat alá, ezt is úgy hívjuk, hogy karakterosztály (meg a \d, \w, stb. kifejezéseket is). Ezzel karakter tartományokat is meg lehet jelölni:

```
[30] PS C:\> "A" -match "[a-f]"
True
[31] PS C:\> "G" -match "[a-f]"
```

```
False
```

Természetesen ezeket az alapelemeket kombinálni is lehet. Keresem a legfeljebb négyjegyű hexadecimális számot:

```
[32] PS C:\> "Ebben van négyjegyű hexa: 12AB" -match "\b[0-9a-f]{1,4}\b"
True
[33] PS C:\> "Ebben nincs: 12kicsiindián" -match "\b[0-9a-f]{1,4}\b"
False
[34] PS C:\> "Ebben sincs: 12baba" -match "\b[0-9a-f]{1,4}\b"
False
```

A fenti karakterosztályokban használhatunk negált változatot is. Például tartalmaz olyan karaktert, ami nem szóköz jellegű karakter:

```
[35] PS C:\> "Van nem szóköz is benne" -match "[^\s]"
True
[36] PS C:\> "     " -match "[^\s]"
False
```

A „nem szóköz”-nek van egyszerűbb jelölése is:

```
[37] PS C:\> "Van nem szóköz is benne" -match "[\S]"
True
[38] PS C:\> "     " -match "[\S]"
False
```

Ennek analógiájára van „nem szókarakter” - \W, „nem számjegy” - \D.

A szögletes zárójelek között csak karaktereket használhatok „vagylagos” értelemben. Ha azt akarom vizsgálni, hogy a szövegben vagy „PowerShell” vagy „PS” található, azt így vizsgálhatom:

```
[39] PS C:\> "Ebben PowerShell van" -match "(PowerShell|PS)"
True
[40] PS C:\> "Ebben PS van" -match "(PowerShell|PS)"
True
[41] PS C:\> "Ebben egyik sem" -match "(PowerShell|PS)"
False
```

Hogy egy kicsit trükkösebb legyen, nézzük az alábbi két példát:

```
[42] PS C:\> "ab" -match "[^ab]"
False
[43] PS C:\> "ab" -match "[^a]|[^b]"
True
```

A [42]-es sorban azért kaptunk hamis eredményt, mert kerestünk egy nem „a”, de nem is „b” karaktert, de az „ab”-ben csak ilyen van, így hamis eredményt kaptam. A [43]-as sorban kerestem egy vagy nem „a”, vagy nem „b” karaktert, márpedig az „ab” első „a”-jára igaz, hogy az nem „b”, tehát van találat!

1.6.6.2 Van benne, de mi?

Az eddigiekben a `-match` operátorral arra kaptunk választ egy `$true` vagy `$false` formájában, hogy a keresett minta megtalálható-e a szövegünkben vagy sem. De egy bonyolultabb mintánál, ahol „vagy” feltételek vagy karakterosztályok is vannak, egy `$true` válasz esetében nem is tudjuk, hogy most mit is találtunk meg. Nézzük meg ezt, az utolsó példa kapcsán:

```
[43] PS C:\> "ab" -match "[^a]|[^b]"
True
[44] PS C:\> $matches

Name                           Value
----                           -----
0                              a
```

Mint látható, a PowerShell automatikusan generál egy `$matches` változót, amely tartalmazza, hogy mit is találtunk a `-match` operátor alkalmazása során. Jelen esetben megtalálhattuk volna a „b”-t is, hiszen arra is igaz, hogy „nem a”, de a `-match` operátor szigorúan balról jobbra halad, így először az „a” karakterre vizsgálja a mintát. Ha ott sikert ér el, akkor nem is halad tovább, megelégszik az eredménnyel és abbahagyja a keresgélést.

A `$matches` igazából nem is egy tömböt, hanem egy hashtáblát tartalmaz, amelynek 0-s kulccsal hivatkozható eleme a megtalált sztring. Nézzünk erre néhány példát:

```
[22] PS C:\> "Ez itt egy példa: szöveg" -match ":.*"
True
[23] PS C:\> $matches[0]
: szöveg
[24] PS C:\> $matches.gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Hashtable                                System.Object
```

Kerestük a kettőspontot és utána tetszőleges számú karaktert, illetve látható, hogy a `$matches` tényleg hashtábla.

1.6.6.3 A mohó regex

A `-match` mögött meghúzódó regex motor „mohó”, azaz ha rá bízzuk a mennyiségeket, akkor ő alapesetben annyit vesz belőle, amennyit csak tud a `-match` kielégítése mellett. Ez nem biztos, hogy nekünk mindig jó:

```
[26] PS C:\> "Keresem ezt: xTalálatx, az x-ek közti szöveget" -match "x.*x"
True
[27] PS C:\> $matches[0]
xTalálatx, az x
```

Mi történt itt? Ugye `xTalálatx`-re számítottunk, de „mohóság” miatt az első x megtalálása után a „.*” minta a szövegem végéig kielégíti a feltételt. Csak az utolsó karakter elérése után döbben rá a regex motor, hogy lemaradt az utolsó feltétel, a második x teljesítése, és ekkor kezd visszafele lépkedni, hogy hátha még ezt is tudja teljesíteni valahol. A visszafele lépkedés során eljut a harmadik x-ig, és ott áll meg. Mit lehetett volna tenni, hogy a kívánt találatot kapjuk meg? A regex „mohóságát” ki is lehet kapcsolni, ehhez egy „?”-et kell tenni az ilyen meghatározatlan számú többszörözők után. Ekkor csak a minimális darabszámot veszi a mintából, és ha az nem elég, akkor növeli azt:

```
[32] PS C:\> "Keresem ezt: xTalálatx, az x-ek közti szöveget" -match "x.*?x"
True
[33] PS C:\> $matches[0]
xTalálatx
```

Vagy pontosíthatjuk a „.*” mintát, hiszen ha például a két *x* között szókarakterek lehetnek csak, akkor pontosabb ez a minta:

```
[41] PS C:\> "Keresem ezt: xTalálatx, az x-ek közti szöveget" -match "x\w*x"
True
[42] PS C:\> $matches[0]
xTalálatx
```

Vagy ez is lehet megoldás:

```
[43] PS C:\> "Keresem: xTalálatx, az x-ek közti szöveget" -match "x[^x]+x"
True
[44] PS C:\> $matches[0]
xTalálatx
```

Itt azzal tettem a regexet szerényebbé, hogy az *x* utáni „nem x” karaktereket keresem x-ig. Ez azért is jó, mert jóval hatékonyabb ennek az ellenőrzése, nincs felesleges ide-oda járkálás a mintában. Nagytömegű feldolgozásnál sokkal gyorsabban eredményre jutunk.

1.6.6.4 Escape a Regex-ben

Ezen utóbbi példáknál gondot jelenthet, ha az „x”-nél valami ésszerűbb szeparátor szerepel a keresett szövegrészünk határaként. Ilyen elválasztó karakter szokott lenni mindenfajta zárójel, perjel, stb. Ezek a regexben is legtöbbször funkcióval bírnak, így ha rájuk keresünk, akkor „escape” karakterrel kell megelőzni ezeket. A regexben az „escape” karakter a visszafele-perjel (\), nem pedig a PowerShellben megszokott visszafele aposztróf (`)!

A fenti „x”-es példát cseréljük () zárójel-párra:

```
[45] PS C:\> "A zárójelek közti (szöveget) keresem" -match "\([^)]*\)"
True
[46] PS C:\> $matches[0]
(szöveget)
```

Megjegyzés:

Ne csak a `$matches` változó tartalmára hagyatkozzunk egy `–match` vizsgálat során, mert ha nincs találatunk, attól még a `$matches` tartalmazza egy korábbi `–match` eredményét:

```
[62] PS C:\> "baba" -match "b"
True
[63] PS C:\> $matches[0]
b
[64] PS C:\> "baba" -match "c"
False
[65] PS C:\> $matches[0]
b
```

1.6.6.5 Tudjuk, hogy mi, de hol van?

Most már tudjuk, hogy van-e a mintánknak megfelelő szövegrészünk, azt is tudjuk, hogy mi az, csak azt nem tudjuk, hogy hol találta a `–match`. Ez főleg akkor érdekes, ha több találatunk is lehet.

Az alapműködést már megbeszéltük, a regex balról jobbra elemez:

```
[3] PS C:\> "Minta az elején, végén is minta" -match "minta"
True
[4] PS C:\> $matches[0]
Minta
```

Erről meg is győződhetünk, hiszen a fenti találat nagy „M”-mel kezdődik, az pedig a szövegünk legeleje.

A regex lehetőséget biztosít, hogy a szövegünk végén keressük a mintát:

```
[7] PS C:\> "Minta az elején, végén is minta" -match "minta$"
True
[8] PS C:\> $matches[0]
minta
```

Itt a találat kis „m” betűs lett, így ez tényleg a szövegem vége. Tehát a szöveg végét egy „$” jel szimbolizálja a mintában. Ez azonban nem jelent fordított működést, azaz itt is balról jobbra történik a kiértékelés. A „$” jelet úgy lehet felfogni, mintha az az eredeti szövegemben is ott lenne rejtett módon, és így a minta tényleg csak a sor végén illeszkedik a szövegre. Például a szöveg utolsó szava:

```
[10] PS C:\> "Utolsó szót keresem" -match "\b\w+$"
True
[11] PS C:\> $matches[0]
keresem
```

A szöveg elejére is lehet hivatkozni a „kalap” (^) jellel. Tehát az a „kalap” jel más funkciójú a szögletes [] zárójelek között (negálás), mint a „sima” regex mintában, ahol szöveg elejét jelenti:

```
[15] PS C:\> "1234 Elején van-e szám?" -match "^\d+"
True
[16] PS C:\> $matches[0]
1234
[17] PS C:\> "Elején van-e 4321 szám?" -match "^\d+"
False
```

Az előző példában csak a szöveg elején található számot tekintjük a mintával egyezőnek.

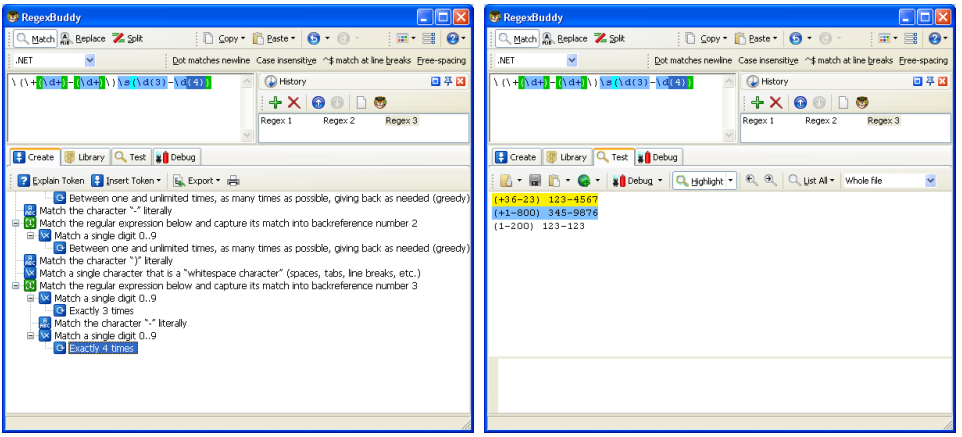
Most már tudjuk, hogy az elején vagy a végén találtuk-e a mintának megfelelő szöveget, de vajon egy köztes szöveget, például szóközök közti szöveget hogyan tudunk úgy megtalálni, hogy magát a szóközöket ne tartalmazza a találat? Erre - egyik megoldásként - alkalmazhatjuk a csoportosítást a mintában:

```
[25] PS C:\> "Ez szóközök közötti" -match "\s(.+?)\s"
True
[26] PS C:\> $matches

Name                           Value
----                           -----
1                              szóközök
0                               szóközök
```

Itt a mintában található „(\.+?)” rész a gömbölyű zárójelek között egy „lusta” többszörözött (a ? lustítja el a + hatását) szókarakter al-minta a teljes minta részeként, más szóval csoport. A `$matches[0]` továbbra is a teljes találatot tartalmazza, de most már van egy `$matches[1]` is, ami az első találati csoport eredményét tartalmazza. Természetesen több csoportot is elkülöníthetünk, például számjegyek csoportjait:

```
[28] PS C:\> "T.: (+36-1) 123-4567" -match "\(\+(\d+)-(\d+)\)\s(\d{3}-
\d{4})"
True
[29] PS C:\> $matches

Name                           Value
----                           -----
3                              123-4567
2                              1
1                              36
0                              (+36-1) 123-4567
```

A fenti példában egy telefonszámból választom ki külön az országkódot, a körzetszámot és a helyi számot. Természetesen egy ilyen mintát összerakni nem egyszerű segédeszköz nélkül. Erre használhatunk olyan eszközöket, mint például a *1.3 Segédprogramok* fejezetben említett RegexBuddy:



35. ábra Barátunk megint segít: értelmez és tesztel

Ez a segédprogram a mintánk különböző részeit színkódokkal jelzi, és a „*Create*” fülön részletes magyarázatot is kapunk az általunk beírt mintáról, észre tudjuk venni, hogy esetleg kifelejtettünk egy „escape” karaktert. A „*Test*” fülön meg ellenőrizni tudjuk, hogy megkapjuk-e találatként a helyes szöveget, illetve azt is, hogy a nem jó szöveg tényleg nem találat-e.

Ezeket a csoportokat nevesíthetjük is a jobb azonosíthatóság érdekében:

```
[47] PS C:\> "Telefon: (+36-1) 123-4567" -match "\(\+(?<ország>\d+)-(?<körze
t>\d+)\)\s(?<helyi>\d{3}-\d{4})"
True
[31] PS C:\> $matches

Name                           Value
----                           -----
körzet                         1
helyi                          123-4567
ország                         36
0                              (+36-1) 123-4567
```

Tehát a csoportot jelző nyitó zárójel mögé tett „?” és „<>” kacsacsőrök közötti címkével tudjuk nevesíteni a csoportot. Itt nyer értelmet, hogy a `$matches` miért hashtábla, és miért nem egyszerű tömb.

Mi van akkor, ha egy csoportra nincs szükségünk? Erre használhatjuk a „(?:)”, u.n. „nem rögzülő” csoportot:

```
[36] PS C:\> "123-4567" -match "(?<eleje>\d+)(?:-)(?<vége>\d+)"
True
[37] PS C:\> $matches

Name                           Value
----                           -----
eleje                          123
```

```
vége                            4567
0                               123-4567
```

A fenti példában a kötőjelet nem rögzülő csoportba helyeztem, így az nem is szerepel külön találatként, de természetesen a teljes találatban (`$matches[0]`) benne van.

Természetesen a csoportokra is alkalmazhatjuk a „?” többszörözés-jelölőt, ami jelenti ugye a 0 vagy 1 darabot, azaz kezelhetjük azokat a telefonszámokat is, ahol nincs országkód és körzetszám:

```
[42] PS C:\> "Telefon: (+36-1) 123-4567" -match "(\(\+(?<ország>\d+)-
(?<körzet>\d+)\)\s)?(?<helyi>\d{3}-\d{4})"
True
[43] PS C:\> $matches

Name                            Value
----                            -----
körzet                          1
helyi                           123-4567
ország                          36
1                               (+36-1)
0                               (+36-1) 123-4567
```

Ilyenkor – a [42]-es példában látható módon – az egymásba ágyazott csoportok külön találatot adnak a listában, jelen esetben 1-es számmal. Nézzük mi van akkor, ha nincs országkód és körzetszám:

```
[44] PS C:\> "Telefon: 123-4567" -match "(\(\+(?<ország>\d+)-
(?<körzet>\d+)\)\s)?(?<helyi>\d{3}-\d{4})"
True
[45] PS C:\> $matches

Name                            Value
----                            -----
helyi                           123-4567
0                               123-4567
```

Most térjünk át a `-match` egy nagy korlátjához: csak az első találatig keres. Bár az előző példában használtuk a „?”-et, más többszörözések nem úgy működnek, ahogy szeretnénk. Vegyünk egy olyan mintát, amellyel el szeretnénk különíteni a szavakat, kezdjünk egyszerűség kedvéért három szóval:

```
[76] PS C:\> "egy kettő három" -match "(\w+)\s(\w+)\s(\w+)"
True
[77] PS C:\> $matches

Name                            Value
----                            -----
3                               három
2                               kettő
1                               egy
0                               egy kettő három
```

Meg is kaptuk a várt csoportokat. Vajon egyszerűsíthetjük-e ezt a kifejezést, egyszersmind felkészíthetjük-e a mintánkat változó számú szóra?

```
[84] PS C:\> "egy kettő három" -match "((?<szó>\w+)\W){3}"
True
[85] PS C:\> $matches

Name                           Value
----                           -----
szó                            három
1                              három
0                              egy kettő három
```

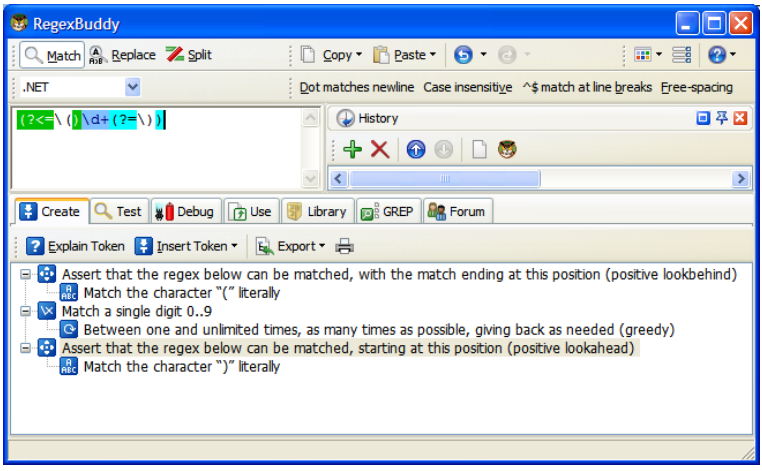
Már a fix számú vizsgálatot se tudtam elérni, a `$matches` nem jegyzi meg egy adott pozíciójú csoport korábbi értékeit. Itt csak a legutolsó szó, a „három” került bele a hashtáblába. Azaz hiába igaz az, hogy csak három szó egyidejű jelenléte teljesíti a feltételt, a végeredménybe csak az utolsó szó kerül bele. Ezt a problémát kicsit később oldom meg.

1.6.6.6 Tekintsünk előre és hátra a mintában

Egyelőre kanyarodjunk oda vissza, hogy a minta helyét valamilyen elválasztó karakter szabja meg. Ha magát a mintát nem akarjuk a találatban viszontlátni, akkor egyik megoldásként alkalmazhatjuk a csoportokat. Egy másik lehetőség a mintában az előretekintés és visszatekintés lehetősége. Például nézzük csak azokat a számokat, amelyek zárójelek között vannak:

```
[86] PS C:\> "sima: 12, nem jó: 11), ez sem: (22, ez jó: (46)" -match
>> "(?<=\()\d+(?=\))"
>>
True
[87] PS I:\>$matches

Name                           Value
----                           -----
0                              46
```

Na, ez megint kezd kicsit hasonlítani az egyiptomi hieroglifákhoz. Nézzük meg, hogy hogyan vizualizálja ezt a RegexBuddy:



36. ábra RegexBuddy értelmezi a mintát

A „(?<=xxx)” szerkezet „visszatekint”, azaz megvizsgálja, hogy az aktuális vizsgálati pozíció előtt az „xxx” minta megtalálható-e. Ha nem, akkor már meg is bukott a vizsgálat. A minta végén található „(?=xxx)” előretekint, azaz megvizsgálja, hogy az aktuális vizsgálati pozíció után az „xxx” minta megtalálható-e. Bár mindkét kitekintő minta zárójeles, de ahogy [87]-ben láthatjuk, nem adnak vissza találatot.

Ezen kitekintő mintáknak is van negatív változatuk, azaz pont azt keressük, hogy a mintánk előtt vagy után ne legyen valamilyen minta:

```
[22] PS I:\>"Árak: Banán:207 Alma:88" -match "(?<!Banán:)\d+"
True
[23] PS I:\>$matches[0]
07
```

Hoppá! Ez nem is jó. Mit is szerettem volna? A „nem banán” gyümölcs árát szerettem volna kiszedni. Azonban a mintám teljesül már a banán áránál is a 2-es után, hiszen a „2” az nem „Banán:”. Azaz ebből az a tanulság, hogy igen gyakran, a negatív kitekintés mellé pozitívat is kell rakni, mert a nem egyezés az nagyon sokféleképpen teljesülhet. A mi esetünkben tehát a jó megoldás:

```
[24] PS I:\>"Árak: Banán:207 Alma:88" -match "(?<!Banán:)(?<=:)\d+"
True
[25] PS I:\>$matches[0]
88
```

Azaz itt most a „pozitív” rész a „:” kettőspont megléte. Ez a két feltétel együttesen már kielégíti a keresési feladatot.

Megjegyzés:

Felmerülhet a tisztelt olvasóban, hogy van-e értelme egykarakteres esetben a negatív körültekintésnek, hiszen használhatnánk negatív karakterosztályt is. Nézzük, hogy mi a különbség a kettő között. Keresem a nem számjeggyel folytatódó „a” betűt:

```
[26] PS C:\> "CAS" -match "a[^\d]"
True
[27] PS C:\> $matches[0]
AS
[28] PS C:\> "CAS" -match "a(?!\d)"
True
[29] PS C:\> $matches[0]
A
[30] PS C:\> "CA" -match "a[^\d]"
False
[31] PS C:\> "CA" -match "a(?!\d)"
True
[32] PS C:\> $matches[0]
A
```

A [27]-es és [29]-es sorokban látható a különbség az eredményben, de ennél súlyosabb a helyzet akkor, ha nincs az „a” betű után „nem számjegy”, hiszen ekkor a negatív karakterosztály fals eredményt ad, mert ő a minta azon helyére mindenképpen akar valamit passzítani, míg a negatív előretekintés esetében a mintának nincs hiányérzete, ha nincs ott semmi már.

1.6.6.7 A mintám visszaköszön

Térjünk vissza a csoportokhoz! Hogyan lehetne felismerni azokat a szavakat, amelyek ugyanolyan elválasztó karakterek között vannak:

```
[16] PS C:\> "-valami-" -match "(\W)\w+\1"
True
[17] PS C:\> "/valami/" -match "(\W)\w+\1"
True
```

Azaz egy csoport eredményét már magában a mintában is felhasználhatom. A csoportra történő hivatkozás formátuma „\1”, „\2”, stb.

Fontos!

Ha egy csoportot ilyen visszahivatkozásban akarjuk szerepeltetni, akkor az a csoport nem lehet „nem rögzülő” csoport, azaz a „?:” jelölést nem használhatjuk.

Természetesen nevesített csoportokra is lehet hivatkozni, ennek formája: „\k<név>”:

```
[28] PS C:\> "A kétszer kétszer rossz" -match "(?<szó>\b\w+\b).+\k<szó>"
True
```

```
[29] PS C:\> $matches.szó
kétszer
```

1.6.6.8 Változatok a keresésre

Az eddigi példákban azt láthattuk, hogy a `-match` operátor, hasonlóan a többi szöveges operátorhoz – nem kis-nagybetű érzékeny. De – mint ahogy a többinél is – ennek is van kis-nagybetű érzékeny változata is, a `-cmatch`:

```
[1] PS I:\>"Ebben Kis Nagy van" -cmatch "kis"
False
```

Ha pont a nem egyezőséget akarjuk vizsgálni, arra a `-notmatch` vagy a kis-nagybetű érzéketlen `-cnotmatch` használható:

```
[2] PS I:\>"Nem szeretem a kacsamajmot!" -notmatch "kacsamajom"
True
[3] PS I:\>$matches
[4] PS I:\>
```

Természetesen a `$true` válasz ellenére itt a `$matches` nem ad találatot.

A `-match` még ezt is lehetővé teszi, hogy egy keresési mintán belül váltogassuk a különböző opciókat. Erre a speciális üzemmód jelölő ad lehetőséget:

(?imnsx-imnsx) **(?imnsx-imnsx:)**	**Jelentése** **(a mintára),** **(vagy az adott csoportra vonatkozóan)**
i	kis-nagybetű érzéketlen
m	többsoros üzemmód, a ^ és $ metakarakterek a sorvégekre is illeszkednek
n	„explicit capture”, azaz csak a nevesített csoportok rögzülnek, a névnélküli csoportok nem
s	egysoros üzemmód, a „.” metakarakter illeszkedik a sorvégkarakterre
x	szóközt figyelmen kívül hagyja a mintában, ilyenkor csak a \s-sel hivatkozhatunk a szóközre

A fenti táblázat fejlécében található jelzés egy kis magyarázatra szorul. A (?imnsx-imnsx) jelzés, a végén kettőspont nélkül jelzi, hogy ez az egész mintára vonatkozik. Természetesen nem használjuk egyszerre az össze opciót bekapcsolt és kikapcsolt módon. Például:

```
"szöveg" -match "(?im-sx)minta"
```

Ebben a példában az „i” és „m” opció be van kapcsolva, az „s” és az „x” meg ki van kapcsolva.

Ha egy mintán belül többfajta üzemmódot szeretnénk, akkor jön a kettőpontos változata ezen kapcsolók használatának:

```
"szöveg" -match "minta(?-i:M)"
```

A fenti példában csak az „M” betűre, mint részmintára vonatkozik a kis-nagybetű érzéketlenség kikapcsolása.

Nézzünk ezek használatára gyakorlati példákat!

```
Tegyük kis-nagybetű érzékennyé a vizsgálatot:
[1] PS C:\> "a NAGY betűst keresem" -match "(?-i)NAGY"
True
[2] PS C:\> "a NAGY betűst keresem" -match "(?-i)nagy"
False
```

Kikapcsoltam az érzéketlenséget, azaz kis-nagybetű érzékennyé tettem a vizsgálatot, és csak a nagybetűs minta esetében kaptam találatot.

Nézzük a többsoros szöveg vizsgálatának opcióit:

```
[5] PS C:\> $szöveg="első sor1
>> második sor2"
>>
[6] PS C:\> $szöveg -match "sor\d$"
True
[7] PS C:\> $matches[0]
sor2
[8] PS C:\> $szöveg -match "(?m)sor\d$"
True
[9] PS C:\> $matches[0]
sor1
```

Létrehoztam egy kétsoros szöveget, keresem a sorvégi „sor” szót és egy számjegyet. A két sorvégnél a számmal teszek különbséget, hogy a találatból kiderüljön, hogy melyiket is találtuk meg. A [6]-os sorban nem szóltam a `-match`-nek, hogy lelkileg készüljön fel a többsoros szövegek vizsgálatára, így a sztringvég karakter csak a szöveg végére illeszkedik, így a találatunk a sor2 lett. Ha szólok neki, hogy a szövegem több soros, akkor az első sor vége is találatot ad. A „szólás” a `(?m)` többsoros üzemmód-kapcsolóval történt.

Nézzünk egy példát a nem nevesített csoportok eldobására:

```
[10] PS I:\>"Eleje vége" -match "(?n)(\w+)\s(?<ketto>\w+)"
True
[11] PS I:\>$matches

Name                           Value
----                           -----
ketto                          vége
0                              Eleje vége
```

```
[12] PS I:\>"Eleje vége" -match "(?-n)(\w+)\s(?<ketto>\w+)"
True
[13] PS I:\>$matches

Name                           Value
----                           -----
ketto                          vége
1                              Eleje
0                              Eleje vége
```

A fenti példában látható, hogy amikor a [10]-es sorban bekapcsoltam az „n” kapcsolót, akkor külön csoportként nem szerepel az első, név nélküli csoport (de természetesen a teljes találat részét képezi), míg a kikapcsolt, alaphelyzet szerinti működés mellett (12]-es sor) az első, nevesítetlen csoport is létrejött.

Nézzük az egysoros kapcsolót. Kicsit zavaró a „többsoros”, „egysoros” elnevezés, hiszen ezek egymás mellett is megférnek, mert más metakarakter működését szabályozzák. Szóval nézzünk egy példát az egysoros üzemmódra:

```
[14] PS C:\> $szöveg="Eleje közepe
>> újsor vége"
>>
[15] PS C:\> $szöveg -match "eleje.+vége"
False
[16] PS C:\> $szöveg -match "(?s)eleje.+vége"
True
```

A [14]-es sorban vizsgálom, hogy illeszkedik-e a kétsoros szövegem olyan mintára, amely egy „eleje” karaktersorozattal kezdődik, és egy „vége” karaktersorozattal végződik. Alapesetben nem kaptam találatot, hiszen a „.+” felakad a sor végén. Míg ha „kierőltetem” az egysoros értelmezést, ahogy az a [16]-os sorban tettem, akkor a „.+” keresztülgázol a sorvégkarakteren is és megkapom a találatomat.

Szóköz mintaként való alkalmazása:

```
[17] PS C:\> "Valami más" -match "(?x)i m"; $matches[0]
False

[18] PS C:\> "Valami más" -match "(?-x)i m"; $matches[0]
True
i m
```

Ebben a példában azt láthatjuk, hogy ha lusták vagyunk a szóköz `\s`-ként való mintában való hivatkozásához, akkor használhatjuk az igazi szóköz karaktert is, ha ezt az üzemmódot bekapcsoljuk. Lehetőség szerint ennek használatát azért kerüljük, mert sok regex változat nem ismeri ezt a lehetőséget, meg a minták olvashatósága sem biztos, hogy a legjobb lesz.

1.6.6.9 Tudjuk, hogy mi, de hányszor?

Az eddigi példákban azt tapasztaltuk, hogy a mintánkat az első találatig keresi a `-match` és változatai. Még a csoportok használatával is csak akkor tudjuk megkeresni a többszörös találatot, ha mi azt a mintát erre eleve felkészítjük, méghozzá meghatározott darabszámmal.

A PowerShell saját kulcsszavaival nem is tudunk többszörös találatot előcsiholni, de szerencsére – mint ahogy már többször láttuk – a .NET Framework ebben is segítségünkre van:

```
[41] PS C:\> $minta = [regex] "\w+"
```

Itt most a `[regex]` típusjelölővel hozok létre mintát. Nézzük meg ennek tagjellemzőit:

```
[42] PS C:\> $minta | gm


   TypeName: System.Text.RegularExpressions.Regex

Name                MemberType Definition
----                ---------- ----------
Equals              Method     System.Boolean Equals(Object obj)
GetGroupNames       Method     System.String[] GetGroupNames()
GetGroupNumbers     Method     System.Int32[] GetGroupNumbers()
GetHashCode         Method     System.Int32 GetHashCode()
GetType             Method     System.Type GetType()
get_Options         Method     System.Text.RegularExpressions.RegexOptio...
get_RightToLeft     Method     System.Boolean get_RightToLeft()
GroupNameFromNumber Method     System.String GroupNameFromNumber(Int32 i)
GroupNumberFromName Method     System.Int32 GroupNumberFromName(String n...
IsMatch             Method     System.Boolean IsMatch(String input), Sys...
Match               Method     System.Text.RegularExpressions.Match Matc...
Matches             Method     System.Text.RegularExpressions.MatchColle...
Replace             Method     System.String Replace(String input, Strin...
Split               Method     System.String[] Split(String input), Syst...
ToString            Method     System.String ToString()
Options             Property   System.Text.RegularExpressions.RegexOptio...
RightToLeft         Property   System.Boolean RightToLeft {get;}
```

A metódusokat nézve mindent tud, amit eddig láttunk, de van egy sokat ígérő `Matches()` metódusunk is, ami többes számot sejtet (reméljük nem egyes szám harmadik személyt! ☺):

```
[43] PS C:\> $eredmény = $minta.matches("Ez a szöveg több szóból áll")
[44] PS C:\> $eredmény


Groups   : {Ez}
Success  : True
Captures : {Ez}
Index    : 0
```

```
Length    : 2
Value     : Ez

Groups    : {a}
Success   : True
Captures  : {a}
Index     : 3
Length    : 1
Value     : a

Groups    : {szöveg}
Success   : True
Captures  : {szöveg}
Index     : 5
Length    : 6
Value     : szöveg

Groups    : {több}
Success   : True
Captures  : {több}
Index     : 12
Length    : 4
Value     : több

Groups    : {szóból}
Success   : True
Captures  : {szóból}
Index     : 17
Length    : 6
Value     : szóból

Groups    : {áll}
Success   : True
Captures  : {áll}
Index     : 24
Length    : 3
Value     : áll
```

Egészen érdekes és sokat sejtető eredményt kaptunk. Nézzük meg a `get-member` cmdlettel, hogy ez mi:

```
[45] PS C:\> Get-Member -InputObject $eredmény


   TypeName: System.Text.RegularExpressions.MatchCollection

Name               MemberType            Definition
----               ----------            ----------
CopyTo             Method                System.Void CopyTo(Array array,...
Equals             Method                System.Boolean Equals(Object obj)
GetEnumerator      Method                System.Collections.IEnumerator ...
GetHashCode        Method                System.Int32 GetHashCode()
GetType            Method                System.Type GetType()
get Count          Method                System.Int32 get Count()
get_IsReadOnly     Method                System.Boolean get_IsReadOnly()
get_IsSynchronized Method                System.Boolean get_IsSynchroniz...
```

```
get Item           Method                 System.Text.RegularExpressions....
get SyncRoot       Method                 System.Object get SyncRoot()
ToString           Method                 System.String ToString()
Item               ParameterizedProperty System.Text.RegularExpressions....
Count              Property               System.Int32 Count {get;}
IsReadOnly         Property               System.Boolean IsReadOnly {get;}
IsSynchronized     Property               System.Boolean IsSynchronized {...
SyncRoot           Property               System.Object SyncRoot {get;}
```

Ez pont az, amit szerettünk volna: egy tömb, ami tartalmazza az összes találatot! Hogyan lehet ezt az eredményt kezelni? Például egy ciklussal kiírathatom az egyes találati elemeket:

```
[46] PS C:\> foreach($elem in $eredmény){$elem.value}
Ez
a
szöveg
több
szóból
áll
```

Megjegyzés:

Vigyázzunk, hogy a [regex] típusnak van egy `match()` metódusa is, de az szintén csak egy találatod ad vissza, hasonlóan a `-match` operátorhoz.

Ahogy a [43]-as sor kimenetében láttuk, a `[regex]` objektumok rendelkeznek `replace()` metódussal is, de van `split()` is, amellyel a minta találati helyeinél lehet feltördelni a szöveget, amelyre alkalmazzuk. Nézzünk erre egy gondolatébresztő példát:

```
[49] PS C:\> $minta = [regex] '\W+'
[50] PS C:\> $minta.split("Ebben {van}, minden-féle (elválasztó) [jel]")
Ebben
van
minden
féle
elválasztó
jel
```

Ebben a példában feldaraboltam a szövegemet mindenféle szóelválasztó karakternél. Ugye ugyanezt az eredményt nem tudtam volna elérni a sztringek `split()` metódusával:

```
[51] PS C:\> ("Ebben {van}, minden-féle (elválasztó) [jel]").split()
Ebben
{van},
minden-féle
(elválasztó)
[jel]
```

1.6.6.10 Csere

Sok esetben azért keresünk egy mintát, hogy az illeszkedő részeket kicseréljük valami másra. Ez a valami más akár lehet egy találati csoport is. Például cseréljük ellenkező sorrendűre egy IP címben szereplő számokat:

```
[64] PS C:\> "Ezt kellene megfordítani: 192.168.1.2 reverse zónához" -replac
e "(\d{1,3})\.(\d{1,3})\.(\d{1,3}).(\d{1,3})", '$4.$3.$2.$1'
Ezt kellene megfordítani: 2.1.168.192 reverse zónához
```

Fontos!

A `-replace` operátornál a minta és a csereszabály között vessző van és a csereszabály nem macskakörmök, hanem aposztrófok között kell hogy álljon, egyébként a csoporthivatkozásokat ($1, $2,...) a PowerShell változóként értékelné, holott ezek nem változók, hanem a `[regex]` formai elemei!

Látjuk, hogy itt a mintacsoportok résztalálataira $1, $2,... formában tudunk hivatkozni, az indexek nem nullától indulnak, hanem 1-től. A `-replace`-en kívül nem lehet hozzáférni a $1 és a többi változóhoz, és a `-replace` által végrehajtott illeszkedésvizsgálat eredménye sem olvasható ki, nem keletkezik `$matches` változó sem.

A `-replace` szerencsére már alaphelyzetben a mintának megfelelő összes szövegrészt kicseréli:

```
[65] PS C:\> "Szóköz kicsrélése aláhúzásra" -replace "\s", '_'
Szóköz_kicsrélése_aláhúzásra
```

Azonban ez nem rekurzív, azaz ha a csere után is marad cserélni való, akkor arra nekünk kell újra meghívni a `-replace`-t:

```
[66] PS C:\> "Felesleges     szóközök    eltávolítása" -replace "\s\s", ' '
Felesleges   szóközök  eltávolítása
```

A fenti példában látszik, hogy a sok egymás melletti szóközt szeretném egyre cserélni azáltal, hogy kettőből csinálok egyet. A fenti módszerrel ez nem működik, hiszen csak egyszer szalad végig, a megoldásban még bőven marad több egymás melletti szóköz. Erre egy jobb minta alkalmazása a megoldás:

```
[67] PS C:\> "Felesleges    szóközök    eltávolítása" -replace "\s{2,}", ' '
Felesleges szóközök eltávolítása
```

Mint ahogy a `-match`-nek is, a `-replace`-nek is van kis-nagybetű érzékeny változata, a `-creplace`:

```
[11] PS C:\> "Csak a nagy 'A'-t cserélem" -creplace "A", 'a'
Csak a nagy 'a'-t cserélem
```

1.6.7 Logikai és bitszintű operátorok

Logikai operátorok segítségével (`-and, -or, -xor`) össze tudunk fűzni több logikai kifejezést, és a szokásos „igazság táblák” alapján kapjuk meg az összetett kifejezésünk igaz vagy hamis értékét:

```
[25] PS I:\>$false -eq $false -and $true -eq $true
True
[26] PS I:\>"ablak" -ne "ajtó" -or 5 -eq 6
True
[27] PS I:\>1 -eq 1 -xor 2 -gt 1
False
[28] PS I:\>-not $true
False
```

A [28]-ban látható, hogy az egytagú `-not` operátor negálja az eredményt.

Kényelmi szolgáltatásként nem kell `$true` vagy `$false`-ra redukálni a logikai operátorok operandusait, a triviális leképezéseket elvégzi helyettünk a PowerShell. Általában minden 0-tól, `$null`-tól, vagy üres kifejezéstől eltérő értékeket `$true`-nak kezel:

```
[29] PS I:\>7 -and $true
True
[30] PS I:\>"valami" -and $true
True
[31] PS I:\>$null -and $true
False
[32] PS I:\>"" -and $true
False
[33] PS I:\>0 -and $true
False
```

Megjegyzés

Az `-and, -or` használatakor vigyázni kell, mert nem értékelődik ki a jobboldaluk, ha már a baloldal alapján el tudja dönteni a végeredményt:

```
[35] PS I:\>$false -and ($a=25)
False
[36] PS I:\>$a
[37] PS I:\>($a=25) -and $false
False
[38] PS I:\>$a
25
[39] PS I:\>$true -or ($b="valami")
True
[40] PS I:\>$b
[41] PS I:\>($b="valami") -or $true
True
[42] PS I:\>$b
valami
```

Tehát nem érdemes egyéb műveleteket végeztetni a logikai operátorok operandusaiban, mert ha nem kellő figyelemmel választjuk ki az oldalakat, akkor esetleg nem úgy működik a programunk, ahogy vártuk.

A fenti logikai operátorok tehát vagy `$false`, vagy `$true` értéket adnak vissza. Ha „igazi” bit szintű logikai műveleteket akarunk elvégezni, akkor ehhez külön logikai operátorok vannak: `-band, -bor, -bxor, -bnot`:

```
[44] PS I:\>126 -bor 1
127
[45] PS I:\>(128+8+4+1) -bor 132
141
[46] PS I:\>(128+8+4+1) -band (128+16+8)
136
[47] PS I:\>(128+8+4+1) -bxor (128+16+8)
21
[48] PS I:\>-bnot 255
-256
[49] PS I:\>-bnot 1
-2
```

Itt természetesen mindig kiértékelődik a művelet mindkét oldala.

1.6.8 Típusvizsgálat, típuskonverzió (-is, -as)

Az 1.5.7 Típuskonverzió fejezetben variáltam a típusokkal. Ebbe a témakörbe tartozik a típusok vizsgálata is. Ha nem lekérni akarjuk egy objektum típusát, hanem rákérdezni, hogy valami valamilyen típusú-e, akkor az `-is` operátort lehet használni:

```
[52] PS I:\>"szöveg" -is [string]
True
[53] PS I:\>"szöveg" -is [object]
True
[54] PS I:\>"szöveg" -is [array]
False
[55] PS I:\>1 -is [double]
False
[56] PS I:\>1.1 -is [double]
True
[57] PS I:\>1.1 -is [float]
False
[58] PS I:\>$true -is [int]
False
[59] PS I:\>$true -is [bool]
True
```

Minden „dolog” a PowerShellben objektum, így akár a szöveg, vagy a szám is, de akár a típusjelölő is objektum:

```
[64] PS I:\>[string] -is [object]
```

```
True
```

És egy majdnem[9] örök igazság:

```
[70] PS I:\>$a -is $a.GetType()
True
```

A már korábban látott típuskonverziós szintaxis mellett használhatunk direkt típus-konvertáló operátort is, ami az `-as`:

```
[73] PS I:\>"01234" -as [int]
1234
[74] PS I:\>1.1234 -as [int]
1
[76] PS I:\>1.1234 -as [double]
1,1234
[77] PS I:\>112345678 -as [float]
1,123457E+08
[78] PS I:\>112345678 -as [double]
112345678
[79] PS I:\>"szöveg" -as [int]
[80] PS I:\>
```

A [79]-es sornál látszik, hogy mi a különbség a korábbi típuskonverziós szintaxis és az `-as` operátor használata között: ez utóbbinál, ha nem tudja végrehajtani a műveletet, akkor nem ad hibajelzést, hanem `$null` értéket ad vissza.

1.6.9 Egytagú operátorok (+, -, ++, --, [típus])

A + jel, ha csak mögötte van objektum, egyszerűen számmá konvertáló operátorként is felfogható, a − jel ugyancsak számmá konvertál, de még negál is:

```
[81] PS I:\>-"01234"
-1234
[82] PS I:\>+"0123"
123
```

A duplázott ++ és −− jel meg inkrementálást és dekrementálást végez, csakhogy két lehetőségünk is van ezek elhelyezésére:

```
[88] PS I:\>$a = 10
[89] PS I:\>$b = $a++
[90] PS I:\>$b
10
[91] PS I:\>$a
11
[92] PS I:\>$c = 10
```

[9] kivéve az `$a=$null` esetet

```
[93] PS I:\>$d = --$c
[94] PS I:\>$d
9
[95] PS I:\>$c
9
```

A [89]-es sorban előbb kapja meg `$b` az `$a` értékét ahhoz képest, hogy `$a`-t megnöveltem, a [93]-ban pedig előbb csökkentettem `$c`-t, és már ezt a csökkentett értéket vette át a `$d`.

Az 1.5.7 Típuskonverzió fejezetben már láthattuk, hogy a *`[típusnév]`* is egy egytagú operátor, ez a típuskonverziós operátor.

1.6.10 Csoportosító operátorok

Sokfajta zárójelet használ a PowerShell, szedjük ezeket össze most ide egy helyre.

1.6.10.1 Gömbölyű zárójel: ()

Elsőként nézzük a gömbölyű zárójelet:

Zárójel	Használat, magyarázat
()	Egyszerű kifejezések szeparálása. Egy csővezeték „összetorlasztása”, egy egységként kezelése. „Elfojtott” visszatérési értékű kifejezések visszatérési értékének megjeleníttetése

Példák:

```
[1] PS C:\>(1+2)*4
12
[2] PS C:\>(Get-Location).Drive

Name        Provider      Root                                CurrentLocatio
                                                                      n
----        --------      ----                                --------------
C           FileSystem    C:\


[3] PS C:\>(get-childitem).count
42
[4] PS C:\>(1,2,3,-5,4,-6,0,-11 | where-object {$  -lt 0}).count
3
```

Az [1]-es sor egyértelmű, meghatározom, hogy mely műveletet végezze előbb. A [2]-es sorban előbb kell elvégeztetni a `Get-Location` parancsot, hogy az ő kimenetének megnézhessem a `Drive` tulajdonságát. A [3]-as sorban a `Get-ChildItem` cmdlettel előbb létre kell hozni azt a gyűjteményt, aminek aztán megnézhetjük a számosságát. A

[4]-es sorban látható, hogy a zárójelek között a csővezetékkel mindenféle műveletet is végezhetünk, például beirányítani egy következő cmdletbe.

Van még egy érdekes jelenség a PowerShellben, nézzünk egy egyszerű értékadást:

```
[21] PS I:\>$a = 1111
[22] PS I:\>
```

Nem ad semmilyen visszatérési értéket. A PowerShell alkotói úgy vélték, hogy ez a gyakoribb igény, azaz nem akarjuk viszontlátni az éppen most átadott értéket. Ezek az u.n. „voidable” kifejezések, amelyek esetében el lehet rejteni a visszatérési értéket. Ettől függetlenül, ha akarjuk, akkor meg lehet jeleníttetni a visszatérési értéket:

```
[28] PS I:\>($a = 1111)
1111
```

1.6.10.2 Dolláros gömbölyű zárójel: $()

Nézzük a következő csoportosítási lehetőséget!

Zárójel	Használat, magyarázat
$()	Több kifejezés szeparálása. Idézőjelek közti kifejezés kifejtése.

Példák:

```
[3] PS I:\>($a ="ab"; $b="lak"; $a+$b).length
Missing closing ')' in expression.
At line:1 char:10
+ ($a ="ab"; <<<<  $b="lak"; $a+$b).length
```

Ha a sima zárójelbe több kifejezést akarnánk beírni, akkor hibát kapunk, mint ahogy a [3]-as sor után látható. Ha ezt nem szeretnénk, akkor a `$()` változatot kell használni:

```
[4] PS I:\>$($a ="ab"; $b="lak"; $a+$b).length
5
```

Megjegyzés:

Itt, a [4]-es sorban egyébként pont jól jön nekünk az, hogy az értékadás nem ad alaphelyzetben kimenetet, hiszen így csak az `$a+$b`-re vonatkozóan kapjuk meg a hosszt, ami pont kell nekünk. Kényszerítsük például az első értékadást arra, hogy legyen kimenete:

```
[30] PS I:\>$(($a ="ab"); $b="lak"; $a+$b).length
2
```

Itt 2-t kaptunk kimenet gyanánt, merthogy ez már egy kételemű tömb, nem pedig egy valamekkora hosszúságú sztring.

Már korábban is tapasztaltuk ennek a speciális zárójelezésnek a jótékony hatását az idézőjeleknél:

```
[6] PS I:\>$g = "szöveg"
[7] PS I:\>"Ez a `$g hossza: ($g.length)"
Ez a $g hossza: (szöveg.length)
[8] PS I:\>"Ez a `$g hossza: $($g.length)"
Ez a $g hossza: 6
```

Ugye itt a [7]-ben nem tettem `$` jelet, csak sima zárójelpárt, az idézőjel miatt csak a `$g` fejtődött ki, ezzel szemben a [8]-ban az egész kifejezés kiértékelődött.

1.6.10.3 Kukacos gömbölyű zárójel: @()

Zárójel	Használat, magyarázat
@()	Tömbkimenetet garantál.

A harmadik zárójelezés típus azt biztosítja, hogy mindenképpen tömb legyen a zárójelek közti művelet eredménye, ha csak egy elem amúgy a kimenet, akkor ezzel egy egyelemű tömböt fogunk kapni. Miért jó ez? Nézzünk erre egy példát:

```
[32] PS I:\>$tömb = "egy", "kettő", "három"
[33] PS I:\>$tömb.length
3
[34] PS I:\>$tömb = "egy", "kettő"
[35] PS I:\>$tömb.length
2
[36] PS I:\>$tömb = "egy"
[37] PS I:\>$tömb.length
3
```

Van egy `$tömb` változónk, amely változó számú elemet tartalmaz. Szükségünk van az elemszámra valamilyen művelet elvégzéséhez, ezért lekérjük a `Length` tulajdonságát. A [33]-ban és a [35]-ben jó eredményt kapunk, a [37]-ben, amikor már csak egyelemű a tömböm, akkor váratlanul nem 1-et, hanem 3-at kapok! Vajon miért? Mert ekkor „hirtelen” a `$tömb`-öm már nem is tömb, hanem már egy egyszerű sztring. Mivel ennek is van `Length` tulajdonsága, ezért nem kapunk hibát, viszont már más az egésznek a jelentése. Ha nem akarunk ilyen jellegű „hibát”, akkor rákényszeríthetjük a kifejezésünket arra, hogy mindenképpen tömböt adjon vissza:

```
[38] PS I:\>$tömb = "egy", "kettő", "három"
[39] PS I:\>@($tömb).length
3
[40] PS I:\>$tömb = "egy", "kettő"
[41] PS I:\>@($tömb).length
2
```

```
[42] PS I:\>$tömb = "egy"
[43] PS I:\>@($tömb).length
1
```

Így már teljesen konzisztens a megoldásunk.

1.6.10.4 Kapcsos zárójel: {} (bajusz)

A következő zárójeltípus a kapcsos zárójel (vagy bajusz):

Zárójel	Használat, magyarázat
{}	Szkriptblokk

Ezzel kapcsolatosan lesz egy külön fejezet, de elöljáróban annyit, hogy kódrészleteket tudunk ezzel csoportosítani, illetve ténylegesen kódként kezelni:

```
[46] PS C:\scripts> $script = {$dátum = get-date; $dátum}
[47] PS C:\scripts> $script
$dátum = get-date; $dátum
[48] PS C:\scripts> $script | Get-Member


   TypeName: System.Management.Automation.ScriptBlock

Name             MemberType Definition
----             ---------- ----------
Equals           Method     System.Boolean Equals(Object obj)
GetHashCode      Method     System.Int32 GetHashCode()
GetType          Method     System.Type GetType()
get_IsFilter     Method     System.Boolean get_IsFilter()
Invoke           Method     System.Collections.ObjectModel.Collection`1[...
InvokeReturnAsIs Method     System.Object InvokeReturnAsIs(Params Object...
set IsFilter     Method     System.Void set IsFilter(Boolean value)
ToString         Method     System.String ToString()
IsFilter         Property   System.Boolean IsFilter {get;set;}


[49] PS C:\scripts> $script.invoke()

2008. április 22. 23:33:35
```

A [46]-ban úgy néz ki, mintha a `$()`-t használtuk volna, de a [47]-ben kiderült, hogy mégsem ugyanaz a helyzet, hiszen a `$script`-változónk nem a zárójeles kifejezés értékét kapta meg, hanem magát a kifejezést. De mégsem sztringgel van dolgunk, hiszen a `Get-Member`-rel nem a sztringekre jellemző tagjellemzőket kapjuk meg, hanem a típusnál is látjuk, hogy ez egy `ScriptBlock`. Ennek fő jellemzője, hogy van neki egy `invoke` metódusa, amellyel végre is lehet hajtatni, mint ahogy a [49]-ben látjuk.

1.6.10.5 Szögletes zárójel: []

Az utolsó zárójeltípus ugyan nem csoportosít, de ha már itt vannak a zárójelek, akkor tegyük ezt is ide:

Zárójel	Használat, magyarázat
[]	Tömbindex Típusoperátor

Példák:

```
[5] PS I:\>(1,2,3,4,5)[2]
3
[6] PS I:\>(1,2,3,4,5)[0]
1
[7] PS I:\>[int] "00011"
11
[8] PS I:\>[datetime]::now

2008. március 14. 10:28:39
```

Az [5]-ös és [6]-os sorokban tömbindex szerepben látjuk a szögletes zárójelet, a [7]-es és [8]-as sorban típusjelöléshez használtam.

1.6.11 Tömboperátor: „,”

Az előző részben volt szó a `@()` csoportosító operátorról, amit különösen akkor tudunk kihasználni, ha az egyelemű tömböket is tömbként akarjuk kezelni. Erre a célra egy másik operátort, a tömboperátort (**,**) is felhasználhatjuk:

```
[11] PS I:\>(,"szöveg").GetType().Fullname
System.Object[]
[12] PS I:\>(,1).GetType().Fullname
System.Object[]
[13] PS I:\>(@(1)).GetType().Fullname
System.Object[]
[14] PS I:\>(,(1,2,3,4)).count
1
[15] PS I:\>(@(1,2,3,4)).count
4
```

A [12]-es és [13]-as sorokban látszik, hogy egyelemű tömbök esetében teljesen ugyanúgy működik a kétfajta operátor. Azonban, ha már többelemű tömböt kezelünk, akkor nagy különbséget láthatunk. A tömboperátor (,) használatával egy egyelemű tömböt kapunk a [14]-es sorban, ahol ez az egy elem egy tömb. A [15]-ös sorban pedig visszakapjuk az eredeti tömbünket.

1.6.12 Tartomány-operátor: „..”

Viszonylag gyakran kell számsorozatokkal dolgoznunk. Ennek megkönnyítésére van egy nagyon praktikus operátor:

```
[17] PS I:\>1..5
1
2
3
4
5
[18] PS I:\>$a=3
[19] PS I:\>1..$a
1
2
3
```

A [19]-es sorban látható, hogy nem csak statikus lehet egy ilyen tartomány vége, hanem változók is lehetnek benne, így még szélesebb körű a felhasználhatósága:

```
[20] PS I:\>$s = "hétfő", "kedd", "szerda", "csütörtök", "péntek",
"szombat", "vasárnap"
[21] PS I:\>$n1 = 2
[22] PS I:\>$n2 = 4
[23] PS I:\>$s[$n1..$n2]
szerda
csütörtök
péntek
```

A fenti példában például tömbök indexelésére használtam.
Vagy lehet fordított sorrendet is kérni, illetve negatív tartományba is lehet menni:

```
[26] PS I:\>3..-2
3
2
1
0
-1
-2
```

Ez az operátor csak egész számokkal működik. Ha valami egyebet (tört szám, sztring formátumban szám) adunk be, akkor a PowerShell automatikusan egésszé konvertálja.

1.6.13 Tulajdonság, metódus és statikus metódus operátora: „.”

Ilyet is már mutattam sokszor, a pontról (`.`)-ról és a kettőspontról (`::`)-ról van szó:

```
[42] PS I:\>[string]::compareordinal("ac", "ab")
1
```

```
[43] PS I:\>$a = new-object random
[44] PS I:\>$a.Next()
1418203324
[45] PS I:\>(get-date).Year
2008
```

A [42]-ben statikus metódusra hivatkoztam a (**::**)-tal, [44]-ben metódusra, [45]-ben tulajdonságra hivatkoztam a (**.**) segítségével.

1.6.14 Végrehajtás

Úgy látszik, szűkösen állunk írásjelekkel, mert a (**.**)-nak van egy másik jelentése. Amikor egytagú operátorként használjuk, akkor végrehajtási operátor a funkciója, azaz az operandusát végrehajtandó kódként tekinti és végrehajtja azt:

```
[55] PS I:\>$a= {get-date}
[56] PS I:\>.$a

2008. március 14. 14:11:14

[57] PS C:\> $a= "get-date"
[58] PS C:\> .$a

2008. március 14. 14:12:21
```

A (**.**)-hoz hasonlóan az (`&`) jel is végrehajt:

```
[59] PS C:\scripts> &$a

2008. március 14. 14:13:51
```

Természetesen a (**.**) és a (`&`), mint végrehajtási operátor paraméterként valamilyen futtatható objektumot várnak: cmdletet, függvényt, szkriptet vagy szkriptblokkot. Akármilyen, számunkra futtathatónak tűnő sztring nem jó nekik:

```
[60] PS I:\>$a = "(1+2)"
[61] PS I:\>.$a
The term '(1+2)' is not recognized as a cmdlet, function, operable program,
or script file. Verify the term and try again.
At line:1 char:2
+ .$ <<<< a
```

Ha mégis ilyen sztringet akarunk végrehajtatni, akkor van szerencsére erre egy PowerShell cmdletünk, az `Invoke-Expression`:

```
[62] PS I:\>Invoke-Expression "1+2"
3
```

1.6.15 Formázó operátor

Láttuk az *1.4.16 Kimenet (Output)* fejezetben, hogy a `write-host` cmdlet alkalmas arra, hogy színesen írjunk a képernyőre, de hogyan lehet mondjuk egy szép táblázatot kiíratni?

Az alábbi kis szkriptben – felhasználva a korábban már bemutatott tartomány-operátort – azt szeretném, hogy a számokat írja ki a PowerShell a nevükkel együtt szép, táblázatszerű módon. A [24]-es promptban készítek egy tömböt, ami tartalmazza a számokat egytől tízig, a [26]-os promptban pedig a korábban bemutatott tartomány-operátor segítségével kiírom a számokat és a nevüket.

```
[24] PS C:\> $tomb= "egy", "ketto", "harom", "negy", "ot", "hat", "hét",
"nyolc", "kilenc", "tíz"
[25] PS C:\> $tomb
egy
ketto
harom
negy
ot
hat
hét
nyolc
kilenc
tíz
[26] PS I:\>0..9 | foreach-object {($_+1), $tomb[$_]}
1
egy
2
ketto
3
harom
4
negy
5
ot
6
hat
7
hét
8
nyolc
9
kilenc
10
tíz
```

Látjuk, hogy ez nem túl szép kimenet, hiszen egymás alatt vannak a számok és neveik, jó lenne, ha ezek egymás mellett lennének. Ugyan használhatnánk a `write-host` cmdletet a `-nonewline` kapcsolóval, vagy az `$OFS` változót, de ennél van egyszerűbb megoldás: a `-f` formázó operátor:

```
[27] PS I:\>0..9 | foreach-object {'{0} {1}' -f ($ +1), $tomb[$ ]}
1 egy
2 ketto
3 harom
4 negy
5 ot
6 hat
7 hét
8 nyolc
9 kilenc
10 tíz
```

Ennek a működése a következő:

A `-f` előtt kell a formátumot leíró sztringet beírni, a `-f` után meg a formázandó kifejezést. Mivel ez utóbbiból több is lehet (a példámban a `($_+1)` és a `$tomb[$_]`), ezért a formázó sztringben is mindkét elemre hivatkozni kell. A hivatkozás formája pedig {0} az első formázandó kifejezést szimbolizálja, a {1} a másodikat és így tovább.

De nem csak egyszerűen hivatkozhatunk a formázandó kifejezésekre, hanem sokkal bonyolultabb műveleteket is végezhetünk velük:

```
[1] PS C:\> "Szöveg elöl {0,-15} és nem hátul" -f "itt van"
Szöveg elöl itt van         és nem hátul
[2] PS C:\> "Szöveg nem elöl {0,15} hanem hátul" -f "van"
Szöveg nem elöl             van hanem hátul
[3] PS C:\> "Ez most az óra: {0:hh}" -f (get-date)
Ez most az óra: 10
[4] PS C:\> "Ez most a forint: {0:c}" -f 11
Ez most a forint: 11,00 Ft
[5] PS C:\> "Ez most a szép szám: {0:n}" -f 1234568.9
Ez most a szép szám: 1 234 568,90
[6] PS C:\> "Ez most a százalék: {0:p}" -f 0.561
Ez most a százalék: 56,10 %
[7] PS C:\> "Ez most a hexa szám: {0:x}" -f 3000
Ez most a hexa szám: bb8
[8] PS C:\> "Ez most a 8 számjegyű hexa szám: {0:x8}" -f 3000
Ez most a 8 számjegyű hexa szám: 00000bb8
[9] PS C:\> "Ez most a forint, fillér nélkül: {0:c0}" -f 11
Ez most a forint, fillér nélkül: 11 Ft
[10] PS C:\> "Ez most rövid dátum: {0:yyyy. M. d}" -f (get-date)
Ez most rövid dátum: 2008. 4. 18
[11] PS C:\> "Ez most hosszú dátum: {0:f}" -f (get-date)
Ez most hosszú dátum: 2008. április 18. 23:10
```

Aztán számokkal lehet akármilyen mintázatot is kirakni:

```
[22] PS C:\> "Ez most a telefonszám: {0:(##) ###-####}" -f 303116867
Ez most a telefonszám: (30) 311-6867
```

Megjegyzés:

A legtöbb formázási művelet elvégezhető a `ToString` metódussal (ami nagyon sok típusnál megtalálható):

```
[23] PS C:\> (303116867).ToString("(##) ###-####")
(30) 311-6867
```

Fontos hangsúlyozni, hogy ezeknek a formázási műveleteknek a kimenete mindig sztring, azaz a formázás után elveszítik az eredeti tulajdonságaikat. Azaz csak egy műveletsor legvégén érdemes formázni, amikor már csak a megjelenítés van hátra, egyéb feldolgozási műveleteken már túljutottunk. Amúgy még sokkal több formázási lehetőség van, javaslom az MSDN honlapján utánanézni.

Visszatérve a számok és neveik kiírásához, még szebben rendezve:

```
[24] PS C:\> 0..9 | foreach-object {'{1,-6} : {0,2}' -f ($ +1), $tomb[$ ]}
egy    :  1
ketto  :  2
harom  :  3
negy   :  4
ot     :  5
hat    :  6
hét    :  7
nyolc  :  8
kilenc :  9
tíz    : 10
```

Ugye ez már ezek után érthető, előre vettem a tömbelemet, hat karakter szélességben, balra igazítva, utána a tömbindexet, két karakter szélességben, jobbra igazítva.

Megjegyzés

Végezetül egy meglepő kifejezés:

```
[26] PS I:\>Get-Date -f MMddyyHHmmss
052708131411
```

Vigyázat! Ez nem formátum operátor, hanem a `get-date`-nek a `format` paramétere rövidítve! Ennek használatáról bővebben a http://msdn.microsoft.com/en-us/library/system.globalization.datetimeformatinfo.aspx oldalon lehet olvasni.

1.6.16 Átirányítás: „>”, „>>”

Az utolsó operátorhoz érkeztünk, ez pedig a kimenet átirányítása a (>) és a (>>) jelekkel. Nézzünk erre példát az előző fejezet tízelemű tömbjének felhasználásával:

```
[41] PS C:\> 0..3 | foreach-object {'{1,-6} : {0,2}' -f ($ +1), $tomb[$ ]} >
 c:\szamok.txt
[42] PS C:\> Get-Content C:\szamok.txt
egy    :  1
ketto  :  2
harom  :  3
negy   :  4
[43] PS C:\> "még egy sor" >> C:\szamok.txt
[44] PS C:\> Get-Content C:\szamok.txt
egy    :  1
ketto  :  2
harom  :  3
negy   :  4
még egy sor
```

A [41]-es sorban átirányítottam a kimenetet egy szöveges fájlba. A szimpla (>) jel új fájlt kezd. A fájl tartalmának kiírásához a `get-content` cmdletet használtam.

A [43]-as sor dupla (>>) jelével hozzáfűztem egy újabb sort a fájlhoz.

```
[45] PS C:\> "új sor" > C:\szamok.txt
[46] PS C:\> Get-Content C:\szamok.txt
új sor
```

A [45]-ös sorban, mivel megint szimpla (>) jelet használtam, ezért elveszettem a fájl addigi tartalmát.

1.7 Vezérlő utasítások

Bár az eddigi példákban és a csővezeték jellegű feldolgozás miatt sokszor bonyolultabb esetben sincs szükség a feldolgozási sorrend megváltoztatására, természetesen egy komoly programnyelv nem nélkülözheti a vezérlő utasításokat sem.

1.7.1 IF/ELSEIF/ELSE

Az `IF-ELSEIF-ELSE` elágazás az egyik legegyszerűbb lehetőség. Be lehet írni akár egysoros kifejezésekbe is, de akkor a kicsit nehezen értelmezhető. Viszont szkriptekben praktikusan használható:

```
[47] PS C:\> $a = 25
[48] PS C:\> if($a -gt 10)
>> { "nagyobb, mint 10" }
>> elseif($a -lt 10)
>> { "kisebb, mint 10" }
>> else
>> { "pont 10" }
>>
nagyobb, mint 10
```

Természetesen nem kötelező az IF mellett ELSEIF és ELSE használata, bármelyik, akár mindkettő is elhagyható.

1.7.2 WHILE, DO-WHILE

A `WHILE` segítségével ciklust szervezhetünk. Kétfajta változata is van, a `WHILE` elöl tesztelő és a `DO-WHILE` a hátul tesztelő. A hátul tesztelő mindenképpen lefut egyszer, az elöl tesztelő akár a belsejének lefuttatása nélkül is továbbléphet, ha az alkalmazott feltétel rögtön hamis. Nézzünk pár példát:

```
[13] PS C:\> $i = 3
[14] PS C:\> while($i -lt 6)
>> {
>>     "*" * $i
>>     $i++
>> }
>>
***
****
*****
```

Amíg az `$i` kisebb, mint 6, addig `$i` darabszámú csillagot írok ki, és növelem az `$i`-t. A ciklus 5 csillagnál lép ki, mivel a 6.-nál már nem igaz az, hogy kisebb, mint 5.

És egy hátul tesztelő:

```
[16] PS C:\> $a=4
[17] PS C:\> do
>> {
>>     $a++
>>     $a
>> }
>> while ($a -lt 6)
>>
5
6
```

Itt meg a ciklus 6-nál lépett ki, hiszen a feltétel nem teljesülését csak akkor vette észre, amikor már a ciklus magja lefutott.

1.7.3 FOR

Miután az előző ciklusokban is láttuk, hogy viszonylag gyakori szerkezet az, hogy a ciklus elején van egy értékadás (inicializálás), aztán van a feltétel, aztán van egy értékváltoztatás része, így erre van egy tömörebb forma is, a `FOR` ciklus:

```
[18] PS C:\> for($a=0; $a -lt 3; $a++)
>> {
>>     [char] (65+$a)
>> }
>>
A
B
C
```

Az *1.6.10 Csoportosító operátorok* fejezetben ismertetett módon a `$()` felhasználásával egyszerre több kifejezést is tehetünk a `FOR` ciklus különböző tevékenységet végző pozícióiba:

```
[24] PS C:\> for($($i=1; $j=10); $i -lt 4; $($i++; $j--)){"{0,2} {1,2}" -f
$i, $j}
 1 10
 2  9
 3  8
```

1.7.4 FOREACH

Mivel a PowerShellben nagyon sokszor gyűjteményekkel (collection) dolgozunk, így az alkotók praktikusnak találták ezek elemein történő műveletvégzést megkönnyítő ciklust is készíteni. Köszönet ezért nekik! A `FOREACH` kulcsszó segítségével olyan ciklust tudunk létrehozni, ahol nem nekünk kell nyilvántartani, számlálni és léptetni a ciklusváltozót, hanem a PowerShell ezt megteszi helyettünk:

```
[25] PS C:\> $egyveleg = 1,"szöveg",(get-date),@{egy=2}
[26] PS C:\> foreach($elem in $egyveleg){$elem.gettype()}

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int32                                    System.ValueType
True     True     String                                   System.Object
True     True     DateTime                                 System.ValueType
True     True     Hashtable                                System.Object
```

Látszik a ciklus működési elve: az `$elem` változóba a PowerShell mindig betölti az aktuális tömbelemet az `$egyveleg` tömbből egészen addig, amíg van elem, és minden elem mellett a sor végén látható szkriptblokkot végrehajtja. Ugyanezt `FOR` ciklussal is meg tudnánk csinálni, de mennyivel többet kell gépelni, és eggyel több változóra is szükségünk van, plusz még az olvashatósága is sokkal nehézkesebb:

```
[27] PS C:\> $egyveleg = 1,"szöveg",(get-date),@{egy=2}
[28] PS C:\> for($i=0;$i -lt $egyveleg.length;$i++){$egyveleg[$i].GetType()}
...
```

Láthattunk a *1.5.3.1 Egyszerű tömbök* fejezetben, hogy a PowerShell képes automatikusan „kifejteni” a tömböket (gyűjteményeket) például a `get-member` cmdletbe való becsövezéskor. A `FOREACH` ennek ellenkezőjét végzi, azaz egy skaláris (nem tömb) paraméterből képes automatikusan egyelemű tömböt készíteni, ha erre van szükség:

```
[29] PS I:\>$skalár = 1
[30] PS I:\>foreach($szám in $skalár){"Ez egy szám: $szám"}
Ez egy szám: 1
```

A fenti példában nem okozott a PowerShellnek problémát `foreach` ciklust futtatni egy nem tömb típusú változóra, automatikusan egy elemű tömbként kezelte.

1.7.4.1 $foreach változó

A `foreach` ciklus belsejében a PowerShell automatikusan létrehoz egy `$foreach` változót, aminek a `$foreach.current` tulajdonsága az éppen aktuális elemet tartalmazza, így akár a [26]-os sorban levő példát másként is írhatnánk:

```
[53] PS C:\> foreach($elem in $egyveleg) {$foreach.current.gettype()}

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int32                                    System.ValueType
True     True     String                                   System.Object
True     True     DateTime                                 System.ValueType
True     True     Hashtable                                System.Object
```

Nézzük meg, hogy ennek a `$foreach` változónak milyen tagjellemzői vannak:

```
 [50] PS C:\> foreach($elem in "a"){,$foreach | get-member}


   TypeName: System.Array+SZArrayEnumerator

Name         MemberType Definition
----         ---------- ----------
Clone        Method     System.Object Clone()
Equals       Method     System.Boolean Equals(Object obj)
GetHashCode Method     System.Int32 GetHashCode()
GetType      Method     System.Type GetType()
get Current Method     System.Object get Current()
MoveNext     Method     System.Boolean MoveNext()
Reset        Method     System.Void Reset()
ToString     Method     System.String ToString()
Current      Property   System.Object Current {get;}
```

Az igazán izgalmas számunkra a `MoveNext()` metódus, amellyel arra késztethetjük a `foreach` ciklust, hogy egy elemet kihagyjon:

```
[54] PS C:\> $tömb = 1,2,3,4,5,6,7
[55] PS C:\> foreach($elem in $tömb){$elem; $foreach.MoveNext()}
1
True
3
True
5
True
7
False
```

A fenti példában miután minden elemnél rögtön még egy elemet léptettünk, ezért csak minden második számot írtam ki. Ugyan a `MoveNext()` ad visszatérési `$true` vagy `$false` értéket attól függően, hogy van-e még elem vagy nincs, de ezt elnyomhatjuk a `[void]` típuskonverzióval:

```
[56] PS C:\> foreach($elem in $tömb){$elem; [void] $foreach.MoveNext()}
1
3
5
7
```

A `$foreach` változó másik fontos metódusa a `Reset()`, ezzel vissza lehet ugrani a ciklus első elemére:

```
[60] PS C:\> $második = $false
[61] PS C:\> foreach($elem in $tömb){$elem; if(-not ($második -or
$foreach.MoveNext())){$foreach.Reset(); $második = $true}}
1
3
5
7
1
```

```
2
3
4
5
6
7
```

A fenti példában egy ciklussal kétszer is végigjárom a `$tömb` tagjait, első menetben minden másodikat írom ki, a második menetben pedig az összes tagot. Itt is jól jött, hogy az `-or` feltétel második operandusa csak akkor kerül kiértékelésre, ha az első tag `$false`, ezzel értem el, hogy a második menetben már ne legyen végrehajtva a `MoveNext()`.

1.7.5 ForEach-Object cmdlet

Az előzőekben bemutatott `FOREACH` egy PowerShell kulcsszó ciklus létrehozására. Azonban létezik egy másik `FOREACH` is, ami a `Foreach-Object` cmdlet álneve:

```
[25] PS C:\> get-alias foreach

CommandType     Name                                Definition
-----------     ----                                ----------
Alias           foreach                             ForEach-Object
```

Honnan tudja a PowerShell hogy egy parancssorban most melyiket is akarom használni: a kulcsszót vagy az cmdletet? Nagyon egyszerűen: ha a `foreach` után egy zárójel „`(`” van, akkor kulcsszóval van dolga, ha „`{`” bajusz, akkor alias. Másrészt a kulcsszavak mindig a kifejezések elején kell álljanak. A kifejezések eleje vagy a parancssor eleje, vagy pontosvessző (`;`), vagy nyitó zárójel „(”. A csőjel (`|`) nem számít kifejezés elejének.

Mire lehet használni ezt a `foreach-object` cmdletet? Míg a `foreach` kulcsszó tömbje, amin végigmegy, rendelkezésre kell álljon teljes egészében készen, mire a vezérlés ráadódik, addig a `ForEach-Object` cmdlet csővezetékből érkező elemeket dolgoz fel, azaz ha már egy bemeneti elem elkészült, akkor azt már át is veszi és elvégzi a kijelölt műveletet:

```
[1] PS C:\> 1,2,3 | ForEach-Object {$  * 3}
3
6
9
```

A fenti példában nem a `$foreach` változón keresztül értem el az aktuális elemet, hanem a szokásos `$_` változón.

Megjegyzés:

Van egy érdekessége ennek a cmdletnek. Figyeljük meg a help-ben a szintaxisát:

```
[2] PS C:\> get-help foreach-object

NAME
    ForEach-Object

SYNOPSIS
    Performs an operation against each of a set of input objects.


SYNTAX
    ForEach-Object [-process] <ScriptBlock[]> [-inputObject <psobject>] [-b
    egin <scriptblock>] [-end <scriptblock>] [<CommonParameters>]
...
```

Hoppá! Nem is egy, hanem három szkriptblokkot lehet átadni neki paraméterként, kicsit hasonlóan, mint ahogy a FOR ciklusnál is három rész vezérli a ciklust. Itt a három rész szerepe: mit tegyen az első elem érkezése előtt, mit tegyen az éppen aktuális elemek érkezésekor, mit tegyen az utolsó elem érkezése után.
Nézzünk erre egy példát:

```
[13] PS C:\> "egy","kettő","három" | foreach-object {"----eleje----"} {$_}
{"----vége----"}
----eleje----
egy
kettő
három
----vége----
```

Ebben a példában nem is írtam ki a paraméterek neveit, azaz a pozíció alapján rendelte hozzá az egyes szkriptblokkokat a PowerShell. Ha csak egy szkriptblokkot adunk meg az a `Process` szekcióhoz sorolódik.

1.7.6 Where-Object cmdlet

Gyakran fordul az elő, hogy nem minden gyűjtemény-elemmel szeretnénk foglalkozni, hanem csak bizonyos tulajdonsággal rendelkezőkkel, és ezeket szeretnénk továbbküldeni a csővezetékünkben feldolgozásra, a szűrőfeltételünknek nem megfelelő elemeket egyszerűen el szeretnénk dobni. Ugyan ezt meg lehet tenni a `ForEach-Object` és az `IF` kombinációjával, de van egy direkt erre kitalált cmdletünk is, a `Where-Object`:

```
[1] PS C:\> $tömb = 1,2,"szöveg", 3
[2] PS C:\> $tömb | Where-Object {$  -is [int]} | ForEach-Object {$ *2}
2
4
6
```

Ugyan a `Where-Object` nem vezérlő utasítás, a hagyományos programozási értelemben, de mivel a csővezetékek kezelése annyira tipikus a PowerShellben, ezért minden csővezetéket „vezérlő” utasítást is nyugodtan sorolhatunk ebbe a kategóriába.

1.7.7 Címkék, törés (Break), folytatás (Continue)

Előfordulhat, hogy a ciklusunkat nem szeretnénk minden körülmények között végigfuttatni minden elemre. Ha teljesen ki akarunk lépni, akkor a `Break` kulcsszót használjuk, ha csak az adott elemen akarunk továbblépni, akkor a `Continue` kulcsszó használható. Nézzünk erre példákat:

```
[4] PS C:\> $tömb= 1,2,"szöveg",3
[5] PS C:\> for($i=0; $i -lt $tömb.length;$i++){if($tömb[$i] -is
[int]){$tömb[$i]*2}else{continue};" ez most a $i. elem duplája volt"}
2
 ez most a 0. elem duplája volt
4
 ez most a 1. elem duplája volt
6
 ez most a 3. elem duplája volt
```

A fenti példában van egy vegyes elemű `$tömb`-öm. Én az egész számokat tartalmazó elemekre szeretnék a szám dupláját kiírni és kikerülni a nem egész számokat. Látszik, hogy az `IF`-nek az `ELSE` ágán a `continue` után már nem hajtódik végre a szöveg kiírása.

Nézzünk egy másik példát, ahol nem átugrunk cikluselemet, hanem teljesen kiugrunk a ciklusból mihelyst oda nem illő elemet találunk:

```
[25] PS C:\> $tömb= 1,2,"szöveg",3
[34] PS C:\> foreach($elem in $tömb){if($elem -isnot [int]){"Hiba!!!!:
$elem"; break}; $elem}
1
2
Hiba!!!!: szöveg
```

Mi van akkor, ha nem egy, hanem két ciklusunk van egymásba ágyazva, és ekkor akarok kiugrani a ciklusokból? Honnan tudja a PowerShell, hogy melyik ciklusból akartam kiugorni, csak a belsőből, vagy mindkét rétegből? Ennek tisztázására használhatunk címkéket, amelyek megmondják a `break` utasításnak, hogy hova ugorjon ki:

```
[38] PS C:\> for($i=0; $i -lt 3;$i++)
>> {
>>     for($j=5; $j -lt 8;$j++)
>>         {
>>             "i:{0} j:{1}" -f $i, $j
>>             if($j -eq 6){break}
>>         }
>> }
>>
```

```
i:0 j:5
i:0 j:6
i:1 j:5
i:1 j:6
i:2 j:5
i:2 j:6
```

A fenti példában látszik, hogy amikor a `$j` eléri a 6-os értéket és lefut a `break`, akkor csak a belső ciklusból léptem ki, megjelenik az `$i=1` és `$i=2` érték is, azaz a külső ciklusból nem lépett ki. Hogyan tudok mindkét ciklusból egyszerre kilépni? Használjunk címkét:

```
[40] PS C:\> :külső for($i=0; $i -lt 3;$i++)
>> {
>>     for($j=5; $j -lt 8;$j++)
>>         {
>>             "i:{0} j:{1}" -f $i, $j
>>             if($j -eq 6){break külső}
>>         }
>> }
>>
i:0 j:5
i:0 j:6
```

Látszik, hogy a `$j=6` első előfordulásánál kilépett mindkét ciklusból.

Megjegyzés:

Még trükkösebb lehetőség, hogy a `break` attribútuma nem csak statikus szöveg lehet, hanem ez a címke egy változóból is jöhet. Azaz bizonyos helyzetekben, mondjuk, az egyik címkéhez ugrunk vissza, más esetekben egy másik címkéhez, és ehhez csak a változónk tartalmát kell változtatni.

1.7.8 SWITCH

A `SWITCH` kulcsszóval nagyon bonyolult `IF/ELSEIF/ELSE` feltételrendszerek tudunk nagyon átláthatóan, egyszerűen megvalósítani. Sőt! Még akár bonyolultabb `ForEach-Object` vagy `Where-Object` cmdleteket is tudunk helyettesíteni ezzel úgy, hogy a szkriptünk sokkal átláthatóbb és egyszerűbb lesz. Nézzünk erre egy példát, először egy `IF` szerkezet kiváltására:

```
[46] PS C:\> $a = 1
[47] PS C:\> if($a -gt 10){"nagyobb mint 10"}elseif($a -gt 5) {"kisebbegyenl
ő 10, nagyobb 5"} else{"kisebbegyenlő 5"}
kisebbegyenlő 5
```

Ennek `switch`-et alkalmazó megfelelője:

```
[48] PS C:\> switch($a)
>> { {$a -gt 10} {"nagyobb mint 10"}
>>     {$a -gt 5} {"kisebbegyenlő 10, nagyobb 5"}
>>     default {"kisebbegyenlő 5"}
>> }
>>
kisebbegyenlő 5
```

Vajon tényleg ekvivalens egymással? Nézzük meg, hogy mi van akkor, ha `$a=15`:

```
[49] PS C:\> $a = 15
[50] PS C:\> if($a -gt 10){"nagyobb mint 10"}elseif($a -gt 5){"kisebbegyenlő
10, nagyobb 5"}else{"kisebbegyenlő 5"}
nagyobb mint 10
```

`Switch`-csel:

```
[52] PS C:\> switch($a)
>> { {$a -gt 10} {"nagyobb mint 10"}
>>     {$a -gt 5} {"kisebbegyenlő 10, nagyobb 5"}
>>     default {"kisebbegyenlő 5"}
>> }
>>
nagyobb mint 10
kisebbegyenlő 10, nagyobb 5
```

Azt láthatjuk itt, hogy a `switch` különböző sorai nem igazi `ELSEIF` módon működnek, azaz nem csak akkor adódik rájuk a vezérlés, ha a megelőző sorok nem adtak eredményt, hanem minden esetben. Ha ezt nem szeretnénk, akkor használjuk a `BREAK` kulcsszót:

```
[53] PS C:\> switch($a)
>> { {$a -gt 10} {"nagyobb mint 10"; break}
>>     {$a -gt 5} {"kisebbegyenlő 10, nagyobb 5"; break}
>>     default {"kisebbegyenlő 5"}
>> }
>>
nagyobb mint 10
```

Ez már ugyanazt a kimenetet adta, mint az `IF`-es megoldás. Tulajdonképpen a default ág előtti break felesleges, mert oda a vezérlés csak akkor kerül, ha egyik megelőző feltétel sem teljesül, de ha kiírjuk, akkor sem okoz ez semmilyen problémát.

A fenti példák talán még nem érzékeltetik eléggé a `switch` előnyét, hiszen itt csak egy háromtagú `IF` láncolatot helyettesítettem. Többtagú kifejezés esetén jobban látszódna a `switch` átláthatósága, bár így is, a legbeágyazottabb `ELSE` ág helyett sokkal szebben néz ki egy `default` címke.

Még jobban kiviláglik a `switch` egyszerűsége, ha egyenlőségre vizsgálunk, hiszen ekkor nem kell külön az egyenlőség operátort sem használni:

```
[1] PS C:\> $a = 10
[2] PS C:\> switch($a){ 1 {"egy"} 2 {"kettő"} 3 {"három"} 10 {"tíz"} default
{"egyéb"}}
tíz
```

De itt nem is ér véget a `switch` előnye, mert – ellentétben az `IF`-fel – a `switch` még gyűjteményeket is képes kezelni, így nagyon jó alternatívája lehet a `ForEach-Object` és a `Where-Object` cmdleteknek is. Például egy „ékezetlenítő” szkript csak ennyiből áll:

```
[42] PS C:\> $a = "öt új űrírás az ütvefúrógépen"
[43] PS C:\> $ofs="";"$(switch([char[]] $a){'á' {'a'} 'é' {'e'} 'í' {'i'}
'ó' {'o'} 'ö' {'o'} 'ő' {'o'} 'ú' {'u'} 'ü' {'u'} 'ű' {'u'} default {$_}})"
ot uj uriras az utvefurogepen
```

Itt a `switch` megkapja az `$a` sztringből képzett karaktertömböt (`[char[]]`), majd a sok karaktercserét megvalósító `switch` tag után visszakonvertálom a karaktersorozatot adó kimenetet (az egész kifejezés `$()`-ben) sztringgé (””), de úgy, hogy ne legyen elválasztó szóköz karakter a betűk között (`$ofs=””`).

Megjegyzés

Bár nem használtam a fenti példában explicit csövet, azaz a „|” csőjelet, de mégis hivatkozható a `switch`-ben a $_ változó.

1.7.8.1 –wildcard

És még itt sem ér véget a `switch` lehetőség-tára. Gyakran foglakozunk szövegek vizsgálatával, így például ilyen jellegű kifejezések is gyakran szükségessé válnak:

```
[44] PS C:\> $a = "szöveg"
[45] PS C:\> switch ($a) {{$a -like "s*"} {"s-sel kezdődik"} {$a -like "z*"}
{"z-vel kezdődik"}}
s-sel kezdődik
```

Ha ezt próbálnánk egyszerűbben írni, és kihagyjuk a „`-like`” operátort, akkor nem kapunk jó eredményt:

```
[46] PS C:\> switch($a){s* {"s-sel kezdődik"} z* {"z-vel kezdődik"}}
[47] PS C:\>
```

Hiszen ekkor a `switch` egyenlőséget vizsgált. Ha `-like` feltétel van mindenütt, akkor a `switch -wildcard` kapcsolójával tudjuk az alaphelyzet szerinti egyenlőségvizsgálatot `-like` vizsgálattá alakítani:

```
[48] PS C:\> switch -wildcard ($a) {s* {"s-sel kezdődik"} z* {"z-vel
kezdődik"}}

s-sel kezdődik
```

1.7.8.2 –regex

Szintén gyakori eset a szövegminták vizsgálata, mint ahogy erre már láttunk példát a *1.6.6 Regex (-match, -replace)* fejezetben. A `switch` erre is fel van készítve:

```
[3] PS C:\> $vizsgálandó="telefon: (30) 311-6867"
[4] PS C:\> switch -regex ($vizsgálandó)
>> {'\(\d{1,2})\)(-|\s)' {"Körzetszám: $($matches[1])"; break}
>> default {"Nincs körzetszám"}}
>>
Körzetszám: 30
[5] PS C:\> $vizsgálandó = "Telefon: 123-4567"
[6] PS C:\> switch -regex ($vizsgálandó)
>> {'\(\d{1,2})\)(-|\s)' {"Körzetszám: $($matches[1])"; break}
>> default {"Nincs körzetszám"}}
>>
Nincs körzetszám
```

A fenti példában zárójelek közti 1 vagy 2 számjegyből álló mintát keresek a vizsgálandó szövegben, amely mintát vagy szóköz, vagy kötőjel követ. Ha van ilyen, akkor a regex belső csoportképzésével kiadódó zárójelek közti számot adom vissza körzetszám gyanánt, egyébként kiíratom, hogy nincs körzetszám.

1.7.8.3 $switch változó

Hasonlóan a `ForEach` kulcsszóhoz, a `switch`-nek is van belső változója, melynek `$switch` a neve, és amelyet felhasználhatunk ciklus-jellegű működés megvalósításra:

```
[8] PS C:\> $Hosszú = "Vezetéknév: Soós
>> Keresztnév: Tibor
>> e-mail: soostibor@citromail.hu
>> Telefon: 30-3116867
>> Város: Budapest"
>>
[9] PS C:\> $darabok = $Hosszú.Split()
[10] PS C:\> $darabok
Vezetéknév:
Soós
Keresztnév:
Tibor
e-mail:
soostibor@citromail.hu
Telefon:
30-3116867
Város:
Budapest
[12] PS C:\> switch($darabok) {
```

```
>> "Vezetéknév:" {[void] $switch.MoveNext(); $vez = $switch.Current}
>> "Keresztnév:" {[void] $switch.MoveNext(); $ker = $switch.Current}
>> "e-mail:" {[void] $switch.MoveNext(); $ema = $switch.Current}
>> "Telefon:" {[void] $switch.MoveNext(); $tel = $switch.Current}
>> "Város:" {[void] $switch.MoveNext(); $var = $switch.Current}
>> }
>>
[13] PS C:\> $ker
Tibor
[14] PS C:\> $vez
Soós
```

Tehát ez a változó akkor használható igazából, ha egy gyűjteményt adunk át a `switch`-nek. Ekkor a gyűjtemény egyes tagjait a `$switch` változó az ő `MoveNext()` metódusával ciklus-szerűen tudja kezelni. Maga a `switch` pedig meg tudja vizsgálni az egyes elemeket és a vizsgálat eredményének függvényében különböző kódrészletek futhatnak le.

A fenti példában van egy hosszú sztringem, amiből ki szeretném szedni a nem címke jellegű adatokat, és ezeket be akarom tölteni a megfelelő változókba. Elsőként feldarabolom a sztringet a `Split()` metódussal. Az így megszülető szavakból álló `$darabok` tömböt átadom a `switch`-nek. Ha a `switch` címkét talál (Vezetéknév:, Keresztnév:, stb.), akkor továbblép a következő darabra és azt betölti a megfelelő változóba. Így minden adat pont a helyére került.

1.8 Függvények

Láttuk, hogy a PowerShell 129 darab cmdlettel rendelkezik, ami elég soknak tűnik elsőre, de amint elkezdünk dolgozni ebben a környezetben rájöhetünk, hogy bizonyos műveletek elvégzéséhez újra és újra ugyanazt az utasítássort használjuk. Ilyenkor érdemes lehet ezeket a műveleteket elmenteni, elmentés előtt esetleg – a szélesebb körű felhasználhatóság érdekében – általánosítani. Az elmentésnek két lehetősége van: függvény és szkript készítése. Persze ez a kétfajta mentési típus nem különül el általában egymástól, hiszen függvénydefiníciót tehetünk szkriptbe, és szkriptet is meghívhatunk függvényből. A közös bennük tehát, hogy tartalmaznak egy olyan PowerShell utasítássort, amely már korábban definiált utasításokból – kulcsszavakból, cmdletekből, függvényekből, szkriptekből, stb. – áll.

A fő különbség a kettő között, hogy a függvényeknek minden esetben van saját nevük és futás időben jönnek létre, míg a szkriptek nem feltétlenül van saját nevük, viszont fájlokban tároljuk őket, ezeknek a fájloknak viszont biztos van nevük, és a fájljuk betöltésével hajtódnak végre.

Elsőként nézzük a függvényeket!

1.8.1 Az első függvényem

Az első függvényem nem sok mindent fog csinálni, kiírja üzembiztosan, hogy „első”:

```
[1] PS C:\> function függvény1 {"első"}
```

Ki is próbálom:

```
[2] PS C:\> függvény1
első
```

Amikor a függvényt meghívom, a PowerShell valójában az általam beírt függvénynév helyett a háttérben végrehajtja, amit én a függvénydefinícióban a két kapcsos zárójel {} közé írtam.

1.8.2 Paraméterek

Természetesen a `függvény1`-nek nem sok értelme van, de például készítsünk egy olyan függvényt, ami kiszámítja egy kör területét. Nem egy konkrét körére gondolok, hanem olyan függvényt szeretnék, amibe paraméterként átadhatom az aktuális kör sugarát, és ebből számítja a területet:

```
[3] PS C:\> function körterület ($sugár)
>> {
>>     $sugár*$sugár*[math]::Pi
>> }
```

```
>>
[4] PS C:\> körterület 12
452,38934211693
[5] PS C:\> körterület 1
3,14159265358979
```

Ez a formátum hasonlatos a hagyományos programnyelvek függvénydefinícióihoz, de a függvényem meghívásának módja kicsit más. Nem kell zárójelet használni, és ha több paraméterem lenne, azok közé nem kell vessző, mert ugye a PowerShellben a vessző a tömb elemeinek szétválasztására való!

1.8.2.1 Paraméterinicializálás

Nézzünk egy újabb függvényt, egy négyszög területét akarom kiszámoltatni:

```
[12] PS C:\> function négyszög ($x, $y)
>> { $x*$y }
>>
[13] PS C:\> négyszög 5 6
30
```

Mi van akkor, ha egy négyzetnek a területét akarom kiszámoltatni, aminek nincs külön x és y oldala, csak x:

```
[14] PS C:\> négyszög 7
0
```

Hát ez nem sikerült, hiszen a függvényem két paramétert várt, az `$y` nem kapott értéket, így a szorzat értéke 0 lett. Lehetne persze a függvényen belül egy IF feltétellel megvizsgálni, hogy vajon a `$y` kapott-e értéket, és ha nem, akkor x^2-et számolni. Ennél egyszerűbben is lehet ezt elérni a paraméterek inicializálásával:

```
[15] PS C:\> function négyszög ($x, $y = $x)
>> { $x*$y }
>>
[16] PS C:\> négyszög 7
49
```

Itt a [15]-ös sorban az `$y` változónak átadom az `$x` értékét. Ez az értékadás azonban csak akkor jut érvényre, ha a függvény meghívásakor én nem adok neki külön értéket, azaz továbbra is működik a téglalap területének kiszámítása is:

```
[17] PS C:\> négyszög 8 9
72
```

Megjegyzés

Mi lenne, ha a négyszög függvény meghívására a hagyományos, zárójeles formátumot használnám most:

```
[22] PS C:\> négyszög(2,3)
Cannot convert "System.Object[]" to "System.Int32".
At line:2 char:6
+ { $x*$ <<<< y }
```

Meg is kaptam a hibát, hiszen a függvényem nincs felkészülve egy kételemű tömb területének kiszámítására.

A paramétereknek nem csak másik változó adhat alapértéket, hanem tehetünk oda egy konstanst is:

```
[18] PS C:\> function üdvözlet ($név = "Tibi")
>> { "Szia $név!" }
>>
[19] PS C:\> üdvözlet Zsófi
Szia Zsófi!
[20] PS C:\> üdvözlet
Szia Tibi!
```

1.8.2.2 Típusos paraméterek

Hasonlóan, ahogy a változóknál is definiálhatunk típust, a függvények paramétereinél is jelezhetem, hogy milyen típusú adatot várok. Ez a függvény hibás működésének lehetőségét csökkentheti.

Nézzük a körterület-számoló függvényemet, mit tesz, ha szövegként adok be neki sugarat:

```
[33] PS C:\> körterület "2"
222222
```

Elég érdekesen alakul ez a geometria a PowerShellben! De eddigi ismereteink alapján rájöhetünk, hogy mi történt. Ugye a függvényünk belsejében ez található:

```
$sugár*$sugár*[math]::Pi
```

Az első `$sugár` a PowerShell szabályai szerint lehet „2” (szövegként). A második már nem, viszont működik az automatikus típuskonverzió, tehát csinál belőle 2-t (szám). Idáig kapunk „22”-t, azaz az első sztringemet („2”) kétszer egymás mellett. Majd jön a PI, ugyan az nem egész, de itt is jön a típuskonverzió, lesz belőle 3, azaz immár a „22”-t rakja egymás mellé háromszor, így lesz belőle „222222”.

Na, ezt nem szeretnénk, ezért tegyük típusossá a sugarat:

```
[42] PS C:\> function körterület ([double] $sugár)
>> {
>>     $sugár*$sugár*[math]::Pi
>> }
>>
[43] PS C:\> körterület "2"
12,5663706143592
```

Itt már helyes irányba tereltük a PowerShell típuskonverziós automatizmusát, rögtön a paraméterátadásnál konvertálja a szöveges 2-t számmá, innentől már nincs félreértés.

Megjegyzés

Az is helyes eredményt adott volna, ha felcseréltük volna a tényezők sorrendjét:

```
[math]::Pi *$sugár*$sugár
```

De elegánssabb és „hibatűrőbb” ha inkább típusjelzőkkel látjuk el a paramétereket.

1.8.2.3 Hibajelzés

Az előző körterület függvényemnél, ha a paraméter nem konvertálható `[double]` típussá, akkor hibát kapunk, anélkül, hogy mi ezt leprogramoztuk volna:

```
[19] PS C:\>körterület "nagy"
körterület : Cannot convert value "nagy" to type "System.Double". Error:
"Input string was not in a correct format."
At line:1 char:11
+ körterület  <<<< "nagy"
```

Viszont elképzelhető olyan paraméterérték is, ami bár nem okoz problémát közvetlenül se a típusos paraméternek, se a függvény belsejének, de mi mégis szeretnénk hibaként kezelni. A körterület példámnál tekintsük hibának, ha valaki negatív számot ad meg sugárnak. A fentihez hasonló formátumú (piros betűk) hibaüzenetet írattathatunk ki a `throw` kulcsszó segítségével:

```
[22] PS C:\>function körterület ([double] $sugár)
>> {
>>     if ($sugár -lt 0)
>>         { throw "Pozitív számot kérek!" }
>>      $sugár*$sugár*[math]::Pi
>> }
>>
[23] PS C:\>körterület -2
Pozitív számot kérek!
At line:4 char:16
+         { throw  <<<< "Pozitív számot kérek!" }
```

De nem csak ilyen esetben lehet szükség hibakezelésre, hanem akkor is, ha valaki nem ad meg paramétert. Egy paraméter kötelező vagy opcionális voltát nem tudom a

függvénydefinícióban jelezni, így ennek vizsgálatát nekem kell elvégeznem és a megfelelő válaszról gondoskodnom. De szerencsére ez sem túl bonyolult a `throw` segítségével:

```
[29] PS C:\>function körterület ([double] $sugár = $(throw "kötelező paramét
er"))
>> {
>>     if ($sugár -lt 0)
>>         { throw "Pozitív számot kérek!" }
>>      $sugár*$sugár*[math]::Pi
>> }
>>
[30] PS C:\>körterület
kötelező paraméter
At line:1 char:47
+ function körterület ([double] $sugár = $(throw  <<<< "kötelező paraméter"
))
[31] PS C:\>körterület 5
78,5398163397448
```

Látszik, hogy már az értékadásnál használhatjuk a hibajelzést, mint alapértéket. Ha nem ad a felhasználó paraméterként értéket, akkor végrehajtódik a hibakezelés, viszont ha ad értéket, akkor természetesen erre nem kerül sor.

1.8.2.4 Változó számú paraméter

Előfordulhat olyan is, hogy nem tudjuk előre, hogy hány paraméterünk lesz egy függvényben. Például szeretnénk kiszámolni tetszőleges darabszámú sztring hosszának az átlagát. Ilyenkor dilemmába kerülhetünk, hiszen hány paramétert soroljunk fel? Hogyan inicializáljuk ezeket?

Szerencsére nem kell ezen töprengenünk, mert a PowerShell készít automatikusan egy `$args` nevű változót, amely tartalmazza a függvénynek átadott valamennyi olyan paramétert, amelyet úgy adunk át, hogy nem névvel hivatkozunk rájuk:

```
[25] PS C:\> function átlaghossz
>> {
>>     $hossz = 0
>>     if($args)
>>         {
>>             foreach($arg in $args)
>>                 {$hossz += $arg.length}
>>             $hossz = $hossz/$args.count
>>         }
>>     $hossz
>> }
>>
[26] PS C:\> átlaghossz "Egy" "kettő" "három" "négy"
4,25
```

Az `átlaghossz` függvényemnek ezek alapján nincs explicit paramétere, azaz névvel nem is lehet hivatkozni a paramétereire, viszont implicit módon az `$args` tömbön keresztül természetesen megkapja mindazt a paramétert, amelyet átadunk neki. Ebben az

esetben azonban nehezebben tudjuk kézben tartani a paraméterek típusát, így ezzel csak a függvény belső, definíciós részében tudunk foglalkozni.

Mi van az `$args` változóval, ha vannak explicit paramétereink is? Nézzünk erre egy példát:

```
[46] PS C:\> function mondatgenerátor ($eleje, $vége)
>> {
>>     write-host $eleje, $args, $vége
>> }
>>
[47] PS C:\> mondatgenerátor "Egy" "kettő" "három" "négy"
Egy három négy kettő
```

Láthatjuk, hogy úgy működik a PowerShell, ahogy vártuk, a nevesített paraméterek nem kerülnek bele az `$args` tömbbe, így a fenti függvényemben egyetlen argumentum sem lett duplán kiírva.

1.8.2.5 Hivatkozás paraméterekre

Eddig az összes függvénypéldámban a függvények meghívásakor a paraméterátadás pozíció alapján történt. Ez egyszerűen azt jelenti, hogy az első paraméter átadódik a függvénydefiníció paraméterei között felsorolt első változónak, a második paraméter a második változónak és így tovább. Ha több paramétert adunk át, mint amennyit a függvény definíciójában felsoroltunk, akkor ezek az extra paraméterek az `$args` változóba kerülnek bele.

Mi van akkor, ha egy függvénynek opcionális paramétere is van, és nem akarok neki értéket adni? Ezt eggyel több szóköz leütésével jelezzem? Alakítsuk át ennek szemléltetésére az előző „mondatgenerátor” függvényemet:

```
[48] PS C:\> function mondatgenerátor ($eleje, $közepe, $vége)
>> {
>>     write-host "Eleje: $eleje közepe: $közepe vége: $vége"
>> }
>>
[49] PS C:\> mondatgenerátor egy kettő három
Eleje: egy közepe: kettő vége: három
[50] PS C:\> mondatgenerátor egy  három
Eleje: egy közepe: három vége:
```

Itt három explicit paramétert fogad a függvény. Ha tényleg három paraméterrel hívom meg, akkor minden paraméter a helye alapján kerül a megfelelő helyre. Ha a második paramétert ki szeretném hagyni, akkor ezt hiába akarom jelezni eggyel több szóközzel ([50]-es sor), ez természetesen a PowerShell parancsértelmezőjét nem hatja meg, így más módszerhez kell folyamodnom, ez pedig a név szerinti paraméterátadás:

```
[55] PS C:\> mondatgenerátor -eleje egy -vége három
Eleje: egy közepe:  vége: három
```

Így már tényleg minden a helyére került. A név szerinti paraméterátadásnál is számos gépelést segítő lehetőséget biztosít számunkra a PowerShell:

```
[56] PS C:\> mondatgenerátor -e egy  -v három
Eleje: egy közepe:  vége: három
[57] PS C:\> mondatgenerátor -e:egy  -v:három
Eleje: egy közepe:  vége: három
```

Az [56]-os sorban látható, hogy a nevekre olyan röviden elég hivatkozni, ami még egyértelművé teszi, hogy melyik paraméterre is gondolunk. Az [57]-es sorban meg az látható, hogy a PowerShell elfogadja a sok egyéb parancssori környezetben megszokott paraméterátadási szintaxist is, azaz hogy kettőspontot teszünk a paraméternév után.

A hely szerinti és pozíció szerinti paraméterhivatkozást vegyíthetjük is:

```
[58] PS C:\> mondatgenerátor -v: három egy -k: kettő
Eleje: egy közepe: kettő vége: három
```

Itt a szabály az, hogy a PowerShell először kiveszi a függvényhívásból a nevesített paramétereket, majd hely szerint feltölti a többi explicit paramétert, és ha még mindig marad paraméter, azt berakja az `$args` változóba.

1.8.2.6 Kapcsoló paraméter ([switch])

Sok függvénynél előfordul olyan paraméter, amelynek csak két állapota van: szerepeltetjük a paramétert vagy sem. Ez nagyon hasonlít a `[bool]` adattípushoz, de feleslegesen sokat kellene gépelni, ha tényleg `[bool]`-t használnánk:

```
függvény -bekapcsolva $true
```

Sokkal egyszerűbb lenne csak ennyit írni:

```
függvény -bekapcsolva
```

Erre találták ki a `[switch]` adattípust, amelyet elsődlegesen pont ilyen paraméterek esetében használunk.

```
[60] PS C:\> function lámpa ([switch] $bekapcsolva)
>> {
>>     if($bekapcsolva) {"Világos van!"}
>>     else {"Sötét van!"}
>> }
>>
[61] PS C:\> lámpa
Sötét van!
[62] PS C:\> lámpa -b
Világos van!
```

A fenti „lámpa” függvényemet, ha kapcsoló nélkül használom, akkor „sötétet” jelez (az IF ELSE ága), ha van kapcsoló, akkor „világos”. Azaz, ha szerepel a paraméterek között a kapcsoló, akkor változójának értéke `$true` lesz, ha nem szerepel a kapcsoló, akkor változójának értéke `$false` lesz.

Megjegyzés

Korábban volt arról szó, hogy a paraméterek név szerinti hivatkozása mellett akár lehet szóközzel, akár kettősponttal értéket adni. Ez a `[switch]` paramétertípusnál nem mindegy:

```
[63] PS C:\old> lámpa -b $false
Világos van!
[64] PS C:\old> lámpa -b:$false
Sötét van!
```

A [63]-as sorban szóközzel választottam el a paramétertől az értéket, ezt a függvényem figyelmen kívül hagyta és a kapcsolónak `$true` értéket adott. A [64]-es sorban kettőspontal adtam a paraméternek értéket, itt már az elvárt módon `$false` értéket vett fel a kapcsolóm.

1.8.2.7 Paraméter-definíció a függvénytörzsben (param)

Ha valaki olyan programozói háttérrel rendelkezik, ahol megszokta, hogy a függvénydefinícióknál van egy külön deklarációs rész, akkor használhatják a `param` kulcsszót a függvény paramétereinek deklarálásához:

```
[2] PS I:\>function get-random
>> {
>>     param ([double] $max = 1)
>>     $rnd = new-object random
>>     $rnd.NextDouble()*$max
>> }
>>
[3] PS I:\>get-random 1000
315,589330306085
```

Itt a `get-random` függvényem neve után nem szerepel semmilyen paraméter, hanem beköltöztettem a függvénytörzsbe, itt viszont kötelezően egy `param` kulcsszóval kell jelezni a paramétereket. Ez akkor is hasznos lehet, ha bonyolultabb kifejezésekkel adok kezdőértéket a paramétereimnek, mert ezzel a külön deklarációs résszel sokkal olvashatóbb lesz a függvényem.

Ennek paraméterezési lehetőségnek egyébként nem is itt, hanem majd a szrkipteknél lesz még inkább jelentősége.

1.8.2.8 Paraméterek, változók ellenőrzése (validálás)

Természetesen a paraméterek ellenőrzése akkor a legegyszerűbb, ha eleve csak a kívánalmainknak megfelelő értékeket lehetne átadni. A típus megfelelőségének vizsgálatához már láthattuk, egyszerűen csak egy típusjelölőt kell a paraméter neve előtt szerepeltetni. De vajon milyen lehetőségünk van, ha még további megszorításokat szeretnénk tenni?

Ennek egyik lehetősége, hogy a függvényünk törzsében saját programkóddal ellenőrizzük az érték helyességét. Például nézzünk a korábban már látott körterület kiszámító függvényt:

```
[22] PS C:\>function körterület ([double] $sugár)
>> {
>>     if ($sugár -lt 0)
>>         { throw "Pozitív számot kérek!" }
>>      $sugár*$sugár*[math]::Pi
>> }
>>
```

Itt én ellenőrzöm, hogy csak pozitív számot lehessen megadni sugárnak. A .NET keretrendszerben azonban vannak kifejezetten ilyen jellegű, paraméterellenőrzésre szolgáló osztályok is. Ilyen szabályokat hozzá is rendelhetünk változókhoz, erre szolgál a változók tulajdonságai között az `Attributes` tulajdonság:

```
[1] PS C:\> $a = 1
[2] PS C:\> Get-Variable a | fl


Name        : a
Description :
Value       : 1
Options     : None
Attributes  : {}
```

Alaphelyzetben ez a tulajdonság üres, vajon hogyan és mivel lehet feltölteni? Nézzük meg, hogy milyen metódusai vannak ennek az attribútumnak:

```
[8] PS C:\> ,(Get-Variable a).Attributes | gm


   TypeName: System.Management.Automation.PSVariableAttributeCollection

Name           MemberType            Definition
----           ----------            ----------
Add            Method                System.Void Add(Attribute item)
Clear          Method                System.Void Clear()
Contains       Method                System.Boolean Contains(Attribute item)
CopyTo         Method                System.Void CopyTo(Attribute[] array...
Equals         Method                System.Boolean Equals(Object obj)
GetEnumerator Method                 System.Collections.Generic.IEnumerat...
GetHashCode    Method                System.Int32 GetHashCode()
```

```
GetType       Method                 System.Type GetType()
get Count     Method                 System.Int32 get Count()
get_Item      Method                 System.Attribute get_Item(Int32 index)
IndexOf       Method                 System.Int32 IndexOf(Attribute item)
Insert        Method                 System.Void Insert(Int32 index, Attr...
Remove        Method                 System.Boolean Remove(Attribute item)
RemoveAt      Method                 System.Void RemoveAt(Int32 index)
set Item      Method                 System.Void set Item(Int32 index, At...
ToString      Method                 System.String ToString()
Item          ParameterizedProperty  System.Attribute Item(Int32 index) {...
Count         Property               System.Int32 Count {get;}
```

Tehát van neki `Add` metódusa, ezzel lehet ellenőrzési objektumokat hozzáadni ehhez az attribútumhoz. A PowerShell MSDN weboldalán[10] és a Reflector programmal a `System.Management.Automation` névtérbe beásva ezeket az ellenőrzési osztályokat lehet megtalálni:

Osztály neve	Felhasználási terület
System.Management.Automation.ValidateArgumentsAttribute	Szülő osztály
System.Management.Automation.ValidateCountAttribute	Attribútumok számának ellenőrzése
System.Management.Automation.ValidateEnumeratedArgumentsAttribute	Szülő osztály
System.Management.Automation.ValidateLengthAttribute	Hossz ellenőrzése (pl.: sztring, tömb)
System.Management.Automation.ValidatePatternAttribute	Regex minta ellenőrzése (sztring)
System.Management.Automation.ValidateRangeAttribute	Értéktartomány ellenőrzése (pl.: int, double)
System.Management.Automation.ValidateSetAttribute	Értéklista ellenőrzése (sztring)
System.Management.Automation.ValidateNotNullAttribute	Nem null érték ellenőrzése
System.Management.Automation.ValidateNotNullOrEmptyAttribute	Nem null és nem üresség ellenőrzése

Nézzünk ezek használatára néhány példát. Az elsőben szeretnék olyan változót használni, amely csak 1 és 5 darab karakter közötti hosszúságú szöveget fogad el:

```
[87] PS C:\old>$vl = New-Object System.Management.Automation.ValidateLengthA
ttribute 1,5
[88] PS C:\old>$szó = "rövid"
[89] PS C:\old>(get-variable szo).attributes.add($vl)
[90] PS C:\old>$szó="nagyonhosszú"
Cannot validate because of invalid value (nagyonhosszú) for variable szo.
At line:1 char:5
+ $szo= <<<< "nagyonhosszú"
```

[10] http://msdn.microsoft.com/
en-us/library/system.management.automation.validateargumentsattribute(VS.85).aspx

A [87]-es sorban definiálom a `ValidateLengthAttribute` osztály egy objektumát. Majd létrehozok egy `$szó` nevű változót „rövid” tartalommal. Ennek a változónak az `Attributes` tulajdonságához hozzáadom az előbb létrehozott ellenőrzési objektumot. A [90]-es sorban a változómnak egy 5 karakternél hosszabb értéket szeretnék adni, de erre hibát kaptam.

Nézzünk egy hasonló példát az üresség vizsgálatára:

```
[94] PS C:\old>$s2="valami"
[95] PS C:\old>$vnne = New-Object System.Management.Automation.ValidateNotNu
llOrEmptyAttribute
[96] PS C:\old>(get-variable s2).Attributes.add($vnne)
[97] PS C:\old>$s2=""
Cannot validate because of invalid value () for variable s2.
At line:1 char:4
+ $s2= <<<< ""
```

Ha egy ilyen ellenőrzési attribútummal látunk el egy változót, akkor az megmutatkozik a változó tulajdonságai között is:

```
[98] PS C:\old>Get-Variable s2 | fl *


Name        : s2
Description :
Value       : valami
Options     : None
Attributes  : {System.Management.Automation.ValidateNotNullOrEmptyAttribute
              }
```

Ezekből az ellenőrzési objektumokból egyszerre többet is lehet egy változóhoz rendelni.

Sajnos ezen ellenőrzési objektumok felhasználásának egyik fő nehézsége az, hogy ahhoz, hogy ilyen szabályt egy változóhoz rendelhessük, ahhoz addigra már a változónak olyan értékkel kell rendelkeznie, amelyre teljesül az ellenőrzési feltétel. Azaz, ha például egy *NotNull* típusú ellenőrzést szeretnénk hozzárendelni egy változóhoz, addigra már valamilyen értékkel kell rendelkeznie a változónak, különben nem engedi a szabályt hozzárendelni. Így a függvények paramétereinél történő felhasználásuk kicsit körülményes:

```
[9] PS C:\> function nemnull ($p)
>> {
>>     $pteszt = 1
>>     $vnn = New-Object System.Management.Automation.ValidateNotNullAttribu
te
>>     (Get-Variable pteszt).Attributes.Add($vnn)
>>
>>     $pteszt = $p
>> }
>>
[10] PS C:\> nemnull 100
[11] PS C:\> nemnull
Cannot validate because of invalid value () for variable pteszt.
```

```
At line:6 char:12
+     $pteszt  <<<< = $p
```

Azaz a függvény törzsében létrehozok egy tesztelési célokat szolgáló `$pteszt` változót, ami kielégíti azt a feltételt, hogy nem üres az értéke, és ez a változó hordozza az ellenőrzési objektumot is. A függvény paraméterének ellenőrzésére a `$p` változó értékét átadom ennek a tesztelési célra létrehozott változónak. Amikor a függvényemet paraméter nélkül hívtam meg, akkor hibát kaptam. Itt igazából csak annyit spóroltam, hogy nem nekem kell lekezelni és kiírni a hibát, ezt a PowerShell helyettem elvégezte.

1.8.3 Változók láthatósága (scope)

A PowerShellben a változóknak van szabályozható láthatóságuk (scope). Az alapszabály, hogy egy adott környezetben létrehozott változók az alkörnyezetekben is láthatók. Környezet maga a PowerShell konzol, egy függvény, egy script. Ahogy ezeket egymásból meghívjuk generálódnak az al-, al-, … környezetek. A PowerShell konzol az u.n. „global” környezet, az összes többi, innen hívott függvény vagy szkript ennek alkörnyezete.

Nézzük az egyszerű láthatóság alapesetét:

```
[1] PS I:\>$a = "valami"
[2] PS I:\>function fv{$a}
[3] PS I:\>fv
valami
```

[1]-ben létrehoztam egy `$a` változót, [2]-ben egy nagyon egyszerű függvényt, ami csak annyit csinál, hogy visszaadja `$a`-t, majd [3]-ban meghívtam ezt az `fv` nevű függvényt. Az eredmény látható: a függvény „látta” `$a`-t, ki tudta írni a tartalmát.

Nézzünk egy kicsit bonyolultabb esetet, mi van akkor, ha a függvényemnek is van egy `$a` változója:

```
[6] PS I:\>$a = "valami"
[7] PS I:\>function fv1{$a="másvalami";$a}
[8] PS I:\>fv1
másvalami
[9] PS I:\>$a
valami
```

[7]-ben az `fv1` függvényem definiál egy `$a`-t, majd ki is írja. [8]-ban meg is kaptam a „belső” `$a` eredményét, de ettől függetlenül [9]-ben a „global” környezetből nézve az `$a` továbbra is az, ami volt. Azaz a gyerek környezet nem hat vissza a szülőre.

Ha szükségünk van külön a szülő változójára is a függvényen belül, akkor ezt meg tudjuk tenni a `get-variable` cmdlet `-scope` paraméterének beállításával. A „`-scope 0`” jelenti a saját környezetet, „`-scope 1`” jelenti a szülőkörnyezetet, „`-scope 2`” jelentené a nagyszülőt és így tovább:

```
[11] PS I:\>function fv2{$a="másvalami";$a; get-variable a -scope 0}
[12] PS I:\>fv2
másvalami

Name                           Value
----                           -----
a                              másvalami
```

[11]-ben a saját `$a`-t írja ki a `get-variable`, a következő példában pedig a szülő `$a` változóját:

```
[14] PS I:\>function fv2{$a="másvalami";$a; get-variable a -scope 1}
[15] PS I:\>fv2
másvalami

Name                           Value
----                           -----
a                              valami
```

A gyerek környezet függvénye meg is tudja változtatni a szülő környezet változóját, ha a `set-variable` cmdlet ugyanilyen `-scope` paraméterét megadjuk:

```
[17] PS I:\>function fv3{$a="másvalami";$a; (get-variable a -scope 1).value;
set-variable a -scope 1 -value "változás"}
[18] PS I:\>fv3
másvalami
valami
[19] PS I:\>$a
változás
```

Itt a `fv3` függvényben vége felé megváltoztatom a szülő `$a` változóját „változás”-ra.
A 0 és a legfelső környezetre némiképp egyszerűbb szintaxissal is hivatkozhatunk:

```
[25] PS I:\>$c=1
[26] PS I:\>function kie{$c=2; "Én c-m:$local:c";"Te c-d:$global:c"}
[27] PS I:\>kie
Én c-m:2
Te c-d:1
[28] PS I:\>function ront{$c=2; $local:c=3;$global:c=4;"Én c-m:$local:c";"Te
c-d:$global:c"}
[29] PS I:\>ront
Én c-m:3
Te c-d:4
```

Egyébként vigyázni kell nagyon a függvényekkel! Az alábbi kis példában, ha a programozó abból a feltételezésből indul ki, hogy a `$b` az úgyis 0 kezdetben, így a `$b=$b+5` értéke 5 lesz, akkor nagyot fog csalódni:

```
[21] PS I:\>$b=10
[22] PS I:\>function elront{$b=$b+5; $b}
[23] PS I:\>elront
15
```

```
[24] PS I:\>$b
10
```

Mi történt itt? A láthatóság miatt az `elront` függvényben az értékadás jobb oldalán szereplő `$b` az a szülő `$b`-je, a bal oldali `$b` már a függvény belső `$b`-je. Tehát – bár a nevük ugyanaz – tárhely szempontjából különbözőek.

Tipp

A függvények belső változóit mindig értékadás után kezdjük el használni, nehogy tévedésből a szülő környezet által definiált ugyanolyan nevű változó értékével kezdjünk el dolgozni.

Ha nem szeretnénk, hogy egy változónkhoz valamelyik gyerek környezet hozzáférhessen, akkor nyilvánítsuk privátnak ezt a változót:

```
[32] PS I:\>$private:priv=1
[33] PS I:\>function latom{"Privát: $priv"}
[34] PS I:\>latom
Privát:
[35] PS I:\>$priv
1
```

Ha egy függvénynek mégis ennek értékével kell dolgoznia, akkor természetesen paraméterátadással ez megtehető minden további nélkül, lényeg az, hogy a privát változókat a gyerek környezet nem fogja tudni a szülő tudta nélkül megváltoztatni.

Azonban azért így sem lehetünk teljes biztonságban, hiszen a `get-variable` és `set-variable` cmdletekkel még a privát változókat is elérhetjük:

```
[36] PS C:\> $private:priv=1
[37] PS C:\> function latom{"Privát: $((get-variable priv -scope 1).value)"}
[38] PS C:\> latom
Privát: 1
[39] PS C:\> function ront{set-variable priv -scope 1 -value 99}
[40] PS C:\> ront
[41] PS C:\> $priv
99
```

A fenti példában láthatjuk, hogy annak ellenére, hogy privátnak definiáltam a `$priv` változómat, ennek ellenére a `latom` függvényem is el tudta érni a `get-variable` segítségével, sőt, a `ront` függvényem még meg is tudta változtatni az értékét.

1.8.4 Függvények láthatósága, „dotsourcing”

Hasonlóan a változókhoz, a függvényeknek is van láthatóságuk, „scope”-juk. Nézzünk erre is egy példát:

```
[12] PS I:\>function külső ($p)
>> {
>>     function belső ($b)
>>     {
>>         Write-Host "Belső paramétere: $b"
>>     }
>>     write-host "Külső paramétere: $p"
>>     belső 2
>>     $k = "külső változója"
>> }
>>
[13] PS I:\>külső 1
Külső paramétere: 1
Belső paramétere: 2
[14] PS I:\>belső 3
The term 'belső' is not recognized as a cmdlet, function, operable program,
 or script file. Verify the term and try again.
At line:1 char:6
+ belső  <<<< 3
[15] PS I:\>. külső 4
Külső paramétere: 4
Belső paramétere: 2
[16] PS I:\>belső 5
Belső paramétere: 5
```

A [12]-es sorban definiált `külső` függvényemben definiálok egy `belső` függvényt. Amúgy a külső csak annyit csinál, hogy kiírja a paraméterét, meghívja a belsőt és még definiál magának egy `$k` változót. A belső csak annyit csinál, hogy kiírja a paraméterét.

Amikor a külsőt meghívom a [13]-as sorban, az szépen le is fut: a külső és a belső paramétere is kiíródik. Amikor a [14]-es sorban közvetlenül a belsőt hívom meg, akkor a PowerShell csodálkozik, hiszen a global scope számára a belső függvény nem látható.

Viszont van lehetőség arra, hogy a külső függvényemet „felemelem” globális szintre és így hívom meg, ezt tettem meg a [15]-ös sorban. Ennek formátuma nagyon egyszerű: egy darab pont, egy szóköz (fontos!) és a függvény neve. Ezt hívjuk dotsourcing-nak, azaz a meghívás szintjére emelem a függvény (vagy szkript, ld. később) futtatását.

Ez természetesen nem csak a belső függvénydefiníciót emelte fel a globális scope-ba, hanem a külső saját változóját is:

```
[17] PS I:\>$k
külső változója
```

Ha nem akarunk dotsourcing-gal bíbelődni, akkor mi magunk is definiálhatjuk eleve globálisnak a függvényben definiált függvényünket:

```
[23] PS I:\>function függvénytár
>> {
>>     function global:első
>>         { "első" }
>>     function global:második
>>         { "második" }
>> }
```

```
>>
[24] PS I:\>függvénytár
[25] PS I:\>első
első
[26] PS I:\>második
második
```

Ha ezek után meghívom a „függvénytár” függvényemet, definiálódik benne az „első” és a „második” függvény is, de a globális scope-ban, így közvetlenül meghívhatók.

Megjegyzés:

Ez a „dotsourcing” nagyon erős, azaz ha így hívok meg egy függvényt, akkor a benne tárolt minden „titok” meghívható lesz közvetlenül. Még akkor is, ha `private`-nak definiálom a belső függvényemet:

```
[28] PS I:\>function titkos
>> {
>>     function private:egymegegy
>>         { "1+1=2" }
>> }
>>
[29] PS I:\>. titkos
[30] PS I:\>egymegegy
1+1=2
```

1.8.5 Referenciális hivatkozás paraméterekre ([ref])

Előfordulhat olyan igény, hogy a függvényemmel nem új értékeket akarok létrehozni, hanem jól bevált, meglevő változóm értékét akarom megváltoztattatni. Erre is lehetőséget ad a PowerShell, hiszen létezik referencia szerinti hivatkozás is:

```
[10] PS I:\>function bánt ([ref] $v)
>> {
>>     $v.value ++
>> }
>>
[11] PS I:\>$a = 1
[12] PS I:\>bánt ([ref] $a)
[13] PS I:\>$a
2
[14] PS I:\>bánt ([ref] $a)
[15] PS I:\>$a
3
```

A `bánt` függvényem közvetlenül bántja a paraméterként átadott változóm értékét. Ezt úgy tudja megtenni, hogy neki nem érték alapján, hanem referencia (azaz memóriacím) alapján adjuk át a változót, amin aztán ő tud dolgozni. Mivel ez a memóriacím egy referenciát tartalmazó változóba töltődik át (`$v`), ezért értelem szerűen nem ennek a

változónak az értékét kell manipulálni, hanem ezen referencia mögötti értéket kell módosítani (`$v.value`).

A függvény meghívásánál is oda kell figyelni, hiszen ha csak simán ennyit írnánk, az nem vezetne eredményre:

```
bánt $a
```

Ilyenkor a PowerShell nem végez automatikus típuskonverziót, azaz nem automatikusan az `$a` változó referenciáját adja át, segíteni kell neki a `[ref]` típusjelzővel. És mivel a szóközzel elválasztott két részt (`[ref]` és `$a`) alaphelyzetben két paraméterként értelmezné, ezért kell zárójelezni, és akkor már jó lesz, mint ahogy ez a [14]-es sorban látható volt:

```
[14] PS I:\>bánt ([ref] $a)
```

1.8.6 Kilépés a függvényből (return)

Elképzelhető, hogy egy függvényből nem a legvégén akarunk kilépni, mert időközben egy elágazás alapján már nem kell további műveletet végezni. Ennek egyik oka lehet valamilyen hiba, amit a *1.8.2.3 Hibajelzés* fejezetben látott `throw` kulcsszóval jelezni tudunk, és itt a függvény végrehajtása meg is szakad.

Ha nem hiba miatt, hanem „normális” módon akarunk egy függvényből kilépni, akkor erre használhatjuk a `return` kulcsszót.

```
[11] PS C:\>function típus ($i)
>> {
>>     switch ($i)
>>     {
>>         {$i -is [int32]} {"Egész" ; return}
>>         {$i -is [double]} {return "Akár tört"}
>>         {$i -is [string]} {return "Szöveg"}
>>     }
>>     "Nem igazán tudom, hogy mi az, hogy $i"
>> }
>>
[12] PS C:\>típus 32
Egész
[13] PS C:\>típus "bla"
Szöveg
[14] PS C:\>típus (get-date)
Nem igazán tudom, hogy mi az, hogy 04/15/2008 09:47:04
```

A példafüggvényemben típust vizsgálok a `switch` kulcsszó segítségével. Először vizsgálom, hogy az `$i` paraméter egész-e? Ha igen, akkor kiírom, hogy „Egész” és kilépek a függvényből. Használhattam volna a `break` kulcsszót is, de az csak a `switch`-ből lépett volna ki és végrehajtotta volna a függvény utolsó utasítását is.

A `return` másik szintaxisát használtam az „Akár tört” és a „Szöveg” ágon, azaz a függvény outputja lesz a `return` paramétere.

Látszik, hogy csak nagyon ritkán elkerülhetetlen a `return` alkalmazása, hiszen az én példámban is tehettük volna azt is, hogy a „Nem igazán tudom...” kiírást berakjuk a `switch default` ágába is, illetve a `return`-ök helyett alkalmazhattam volna `break`-et is. Mindenesetre bizonyos helyzetekben jól jöhet a `return`, tudjunk róla!

1.8.7 Pipe kezelése, filter

Készítsünk olyan függvényt, ami kiszámolja a neki átadott fájlok hosszainak összegét!

```
[19] PS C:\old>function fájlhossz ([System.IO.FileInfo] $f)
>> {
>>     $hossz = 0
>>     $hossz += $f.length
>>     $hossz
>> }
>>
[20] PS C:\old>fájlhossz C:\old\alice.txt
709
```

Ez tökéletes, de ehhez nem nagyon kellett volna függvény, hiszen a fájlok `Length` tulajdonsága pont ezt az értéket adja vissza. Én azt szeretném, hogy egy fájlgyűjteményt is átadhassak a függvénynek, így több fájlnak az együttes hosszát is megkaphassam. Ehhez alakítsuk át a függvényt:

```
[33] PS C:\old>function fájlhossz ([System.IO.FileInfo[]] $f)
>> {
>>     $hossz = 0
>>     foreach ($file in $f)
>>     {
>>         $hossz += $file.length
>>     }
>>     $hossz
>> }
>>
[34] PS C:\old>fájlhossz (dir)
395962
```

A függvény paraméterdefiníciós részében felkészülök fájltömb fogadására, majd a függvény törzsében egy `foreach` ciklussal végigszaladok az elemeken és összeadogatom a hosszokat. Ez már jó, csak nagyon nem PowerShell-szerű a függvény meghívása. Sokkal elegánsabb lenne, ha a `dir` kimenetét lehetne becsövezni a függvényembe. Nézzük meg, hogy alkalmas-e erre a függvényem átalakítás nélkül?

```
[35] PS C:\old>dir | fájlhossz
0
```

Nem igazán... Merthogy ilyen csövezős esetben a PowerShell nem ad át értéket a „normál” paramétereknek, hanem egy automatikusan generálódó `$input` változónak adja ezt át:

```
[36] PS C:\old>function fájlhossz
>> {
>>     $hossz = 0
>>     foreach ($file in $input)
>>     {
>>         $hossz += $file.length
>>     }
>>     $hossz
>> }
>>
[37] PS C:\old>dir | fájlhossz
395962
```

Így már majdnem tökéletes a függvényem, de lehet ezt még szebbé tenni! Mi ezzel a gond? Az, hogy ha egy nagyon mély, sok fájlt tartalmazó könyvtárstruktúrára alkalmazom, akkor azt fogjuk tapasztalni, hogy a PowerShell.exe memória-felhasználása jó alaposan felmegy, mire összeáll az `$input` változóban a teljes fájlobjektum lista, és csak utána tud lefutni a `foreach` ciklus. Sokkal optimálisabb lenne, ha már az első fájl átadásával elkezdődhetne a számolás, a memóriában így mindig csak egy fájllal kellene foglalkozni. Erre is van lehetőség, bár - érdekes módon – a help erről nem ír! Merthogy egy függvénynek valójában lehet három elkülönülő végrehajtási fázisú része:

```
function <név> ( <paraméter lista> )
{
        begin {
                <parancsok>
        }
        process {
                <parancsok>
        }
        end {
                <parancsok>
        }
}
```

Lehet tehát egy függvénynek egy „`begin`” része, ami egyszer fut le, a függvény meghívásakor. Lehet egy „`process`” része, ami minden egyes csőelem megérkezésekor lefut, és lehet egy „`end`” része, ami az utolsó csőelem érkezése után fut le.

Alakítsuk át úgy a fájlhossz függvényemet, hogy a `process` szekcióba kerüljön a feldolgozás:

```
[44] PS C:\old>function fájlhossz
>> {
>>     begin {$hossz = 0}
>>     process {$hossz += $ .length}
>>     end {$hossz}
>> }
```

```
>>
[45] PS C:\old>dir | fájlhossz
395962
```

Mik a főbb változások? Egyrészt nem kell nekünk ciklust szervezni, mert a csőelemek amúgy is egyesével érkeznek. Viszont a `process` szekcióban nem a `$input` változóval kell foglalkoznunk, hanem a `$_` változóval, az tartalmazza az aktuális csőelemet. Olyannyira, hogy ha így `begin`/`process`/`end` szekciókra bontjuk a függvényt, és a $_ változót használjuk, akkor nem is generálódik `$input`!

Ez olyan fontos, és annyira nincs benne a helpben, hogy kiemelem újra:

Fontos!

Ha a függvényemben külön `begin`/`process`/`end` szekciót használok, akkor nem képződik `$input` változó, de a `process` szekcióban a `$_` változón keresztül érhetem el a bejövő csőelemeket!

Ha egy függvénynek csak `process` szekciója lenne, akkor az ilyen függvényt egy külön kulcsszóval, a `filter` -rel is definiálhatjuk, és akkor egyszerűbb a szintaxis is. Például ha egy függvényem csak annyit csinálna, hogy a csőben beléérkező számokat megduplázza, akkor az így nézne ki:

```
[46] PS C:\ filter dupláz
>> {
>>     $_*2
>> }
>>
[47] PS C:\ 1,2,5,9 | dupláz
2
4
10
18
```

Látszik, hogy nincs szükség `process` kulcsszóra, nincs felesleges bajuszpár. Viszont nem lehetséges sem `begin`, sem `end` szekció definiálása, így ha ilyenre van szükségünk (valamilyen függvényváltozót kellene inicializálni, vagy az egész folyamat végén kellene még valamit kitenni az outputra), akkor azt nem ússzuk meg a `function` használata nélkül.

1.8.8 Szkriptblokkok

A paraméterek egyik nagyon érdekes fajtája a szkriptblokk. Enélkül csak nagyon nehézkesen, bonyolult programozással lehetne bizonyos feladatokat függvényekkel megoldani, szkripblokkokkal viszont nagyon egyszerűen. Nézzünk erre egy példát! Szeretnék egy általánosabb függvényt csinálni, az előző `dupláz` helyett, amellyel tetszőleges matematikai műveletet el tudok végezni a csővezetéken beérkező számokon. Ezt a műveletet

paraméterként szeretném átadni a függvényemnek. Elsőre elég bonyolultnak tűnhet ilyesmit csinálni, de a szkriptblokkokkal pofonegyszerű:

```
[49] PS C:\>filter művelet ([scriptblock] $s)
>> {
>>     &($s)
>> }
>>
[50] PS C:\>1,2,3 | művelet {$ *3}
3
6
9
[51] PS C:\>10,5,2 | művelet {$ /2}
5
2,5
1
```

A függvényem definíciója látható a [49]-es sorban. Egy „igazi” paramétere van, ami `[scriptblock]` típusú, a másik paramétere majd a csőből jön, és majd a `$_` változón keresztül tudjuk megfogni. Maga a függvény törzse néhány karakter: „`&($s)`”, azaz csak annyi történik, hogy a paraméterként átadott szkriptet meghívjuk az „`&`” operátorral. És ezzel el is készült a nagy tudású, univerzális matematikai függvényünk! Persze a használatához tudni kell, hogy olyan kifejezést kell neki átadni paraméterként, ami a `$_` változóval játszadozik, de amúgy tényleg egyszerű a dolog: az [50]-es sorban háromszoroz a függvényem, az [51]-es sorban pedig felez.

1.8.8.1 Anonim függvények

Valójában egy szkriptblokk nem csak paraméterként szerepelhet, hanem név nélküli függvényként is felfoghatjuk őket. Nézzük a legegyszerűbb csővezetékkel végezhető műveletet, adjuk vissza egyesével, változtatás nélkül a csőbe bemenő elemeket. Ugye megtanultuk, hogy a csőelemekhez a `process` szekcióban férünk hozzá a `$_` változón keresztül. Első próbálkozás:

```
[85] PS C:\old> 1, "kettő", 3.04 | process {$ }
Get-Process : A parameter cannot be found that matches parameter name '$_'.
At line:1 char:27
+ 1, "kettő", 3.04 | process  <<<< {$_}
```

Ez nem jó, a csőjel (`|`) után valami kifejezést vár, valami végrehajtható dolgot, márpedig a `process` kulcsszó, nem végrehajtható kifejezés, ezért a PowerShell itt ezt úgy értelmezi, hogy az a `Get-Process` álneve, annak pedig nincs `$_` paramétere. Segítsünk neki, csináljunk belőle szkriptet, pontosabban fogalmazva szkriptblokkot, amiben már van értelme a `process` kulcsszónak:

```
[86] PS C:\old> 1, "kettő", 3.04 | {process {$ }}
Expressions are only permitted as the first element of a pipeline.
At line:1 char:33
```

```
+ 1, "kettő", 3.04 | {process {$_}} <<<<
```

Ez sem lett sokkal jobb, merthogy – érdekes módon – a szkriptblokk sem végrehajtható! Az majdnem ugyanolyan adatszerűség, mint a változó, vagy akár egy szám. Segítsünk tovább neki, tegyünk elé egy végrehatás-operátort:

```
[87] PS C:\old> 1, "kettő", 3.04 | &{process {$_}}
1
kettő
3,04
```

Na, itt már jól lefutott. Kis átalakítással ugyanazt a funkcionalitást el tudjuk érni, mint az előző alfejezet `dupláz` filteré:

```
[88] PS C:\old> 1, "kettő", 3.04 | &{process {$ *2}}
2
kettőkettő
6,08
```

Azaz az ilyen szkriptblokkokat felfoghatjuk úgy is, mint anonim függvények. Ilyeneket akkor érdemes használni, ha nem akarom többször felhasználni a függvényt, hiszen akkor minek neki nevet adni?

1.8.9 Függvények törlése, módosítása

Ha egy függvényre nincs szükségem, akkor törölhetem is. Van ugye egy speciális `PSDrive`-om, a `function:`, ez tartalmazza az összes függvényt és ennek különböző elemeit ugyanúgy törölhetem, mint a fájlokat:

```
[33] PS I:\>Get-ChildItem function:

CommandType     Name                               Definition
-----------     ----                               ----------
...
Function        belső                              param($b) Write-Host "Bels...
Function        külső                              param($p) function belső (...
Function        függvénytár                        function global:első...
Function        első                               "első"
Function        második                            "második"
Function        titkos                             function private:egymegegy...
Function        egymegegy                          "1+1=2"
```

A lista végén vannak az általam definiált függvények, előtte „gyári” függvények vannak, amelyeket mindjárt átnézünk.

Ha meg szeretnék szabadulni például az `egymegegy` függvényemtől, akkor használhatom a `remove-item` cmdletet:

```
[34] PS I:\>Remove-Item function:egymegegy
```

Ha már itt tartunk, akkor vajon PSDrive-kezelő cmdletekkel is létrehozhatok-e függvényt? Természetesen igen, nézzük hogyan működik a `new-item`:

```
[35] PS C:\> new-item -path function: -name fv1 -value {"Fájlként new-item-m
el"}

CommandType     Name                            Definition
-----------     ----                            ----------
Function        fv1                             "Fájlként new-item-mel"


[36] PS C:\> fv1
Fájlként new-item-mel
```

Módosítani a függvényeimet vagy megismételt függvénydefinícióval tudom, vagy a szintén a PSDrive-kezelő `set-item` cmdlettel:

```
[37] PS C:\> Set-Item -path function:fv1 -Value {"Módosítás set-item-mel"}
[38] PS C:\> fv1
Módosítás set-item-mel
```

Hát ez sikerült! Akkor nézzük is meg, hogy amúgy mire képes a `set-item`, és az hogyan alkalmazható a függvényekre! Érdekes módon a helpben a szintaxisában nincsen `-options` paraméter:

```
[53] PS C:\old> (get-help set-item).syntax

Set-Item [-path] <string[]> [[-value] <Object>] [-force] [-include <string[
]>] [-exclude <string[]>] [-filter <string>] [-passThru] [-credential <PSCr
edential>] [-whatIf] [-confirm] [<CommonParameters>]
Set-Item [-literalPath] <string[]> [[-value] <Object>] [-force] [-include <
string[]>] [-exclude <string[]>] [-filter <string>] [-passThru] [-credentia
l <PSCredential>] [-whatIf] [-confirm] [<CommonParameters>]
```

Azonban a példákban már látható, hogy van:

```
[54] PS C:\old> (get-help set-item).examples

...

-------------------------- EXAMPLE 4 --------------------------

C:\PS>set-item -path function:prompt -options "AllScope,ReadOnly"


This command sets the AllScope and ReadOnly options for the "prompt" functi
on. This command uses the Options dynamic parameter of the Set-Item cmdlet.
 The Options parameter is available in Set-Item only when you use it with t
he Alias or Function provider.
```

A magyarázat is ott van legalul, hogy ez egy dinamikus paraméter, és csak az `Alias` és a `Function` PSDrive használatakor létezik. Ezzel lehet beállítani a függvényem látha-

tóságát (scope) és – hasonlóan a változókhoz – azt is, hogy a függvényem csak olvasható (azaz nem lehet felülírni), vagy konstans (még a `-force` kapcsolóval sem lehet felülírni).

Megjegyzés

Sajnos elég esetleges ezen dinamikus paraméterek fellelése a helpben, a cmdletek mellett. Talán eredményesebb, ha a providerek irányából keressük meg ezeket. Például nézzük, hogy az `Alias:` providernek milyen dinamikus paraméterei vannak:

```
[63] PS C:\> get-help -Category provider -name alias
...
DYNAMIC PARAMETERS
    -Options <System.Management.Automation.ScopedItemOptions>
        Determines the value of the Options property of an alias.

            None
                No options. "None" is the default.

            Constant
                The alias cannot be deleted and its properties cannot be ch
                anged. Constant is available only when you are creating an
                alias. You cannot change the option of an existing alias to
                 Constant.

            Private
                The alias is visible only in the current scope (not in chil
                d scopes).

            ReadOnly
                The properties of the alias cannot be changed, except by us
                ing the Force parameter. You can use Remove-Item to delete
                the alias.

            AllScope
                The alias is copied to any new scopes that are created.


        Cmdlets Supported: New-Item, Set-Item
```

Azaz több lehetőségünk is van arra, hogy milyen módon kezeljük a függvényeinket, álneveinket.

Megjegyzés

Kézenfekvőnek tűnhetne, hogy a függvényobjektum `definition` tulajdonságát írjuk át a függvény módosításához. Ez azonban nem megy, mert az *Read Only* attribútum, azaz nem írható át:

```
[41] PS C:\> (get-item function:fv1).definition = {"Módosítás a definition t
ulajdonságon keresztül"}
"Definition" is a ReadOnly property.
At line:1 char:25
+ (get-item function:fv1).d <<<< efinition = {"Módosítás a definition tulaj
```

```
donságon keresztül"}
```

A fenti példában látszik, hogy hibát a `definition` attribútum közvetlen módosításakor kaptam.

1.8.10 Gyári függvények

A függvényeken belül akkor most nézzük meg azokat, amelyek már az alap PowerShell környezetben is benne vannak. Ehhez nem kell a helpet hosszasan bújnunk, mivel – mint ahogy korábban már láthattuk – egy külön meghajtó van létrehozva a függvények elérésére és listázásához, a „`function:`” PSDrive:

```
PS C:\> dir function:

CommandType     Name                                Definition
-----------     ----                                ----------
Function        prompt                              'PS ' + $(Get-Location) + ...
Function        TabExpansion                        ...
Function        Clear-Host                          $spaceType = [System.Manag...
Function        more                                param([string[]]$paths);  ...
Function        help                                param([string]$Name,[strin...
Function        man                                 param([string]$Name,[strin...
Function        mkdir                               param([string[]]$paths); N...
Function        md                                  param([string[]]$paths); N...
Function        A:                                  Set-Location A:
Function        B:                                  Set-Location B:
Function        C:                                  Set-Location C:
...
Function        Z:                                  Set-Location Z:
```

Ezek közül a „*betű*:” formátumúak nagyon egyszerűek, ezek csak annyit csinálnak, hogy az aktuális meghajtónak beállítják az adott betűjellel jelzett meghajtót.

A `prompt` függvény automatikusan lefut, ha a konzolon új sort nyitunk. Ezt a függvényt is tetszőlegesen átszerkeszthetjük, például én is itt a könyvben, hogy jobban lehessen hivatkozni a begépelt kódsorokra, beraktam egy sorszámot is szögletes zárójelek közé, amit ráadásul mindig növelek eggyel.

Szintén egy függvénynek, a `TabExpansion`-nek köszönhetjük a TAB-ra történő parancs-kiegészítést, ami akkor fut le automatikusan, amikor megnyomjuk a TAB billentyűt. Ezt is testre szabhatjuk, ha nem találjuk elég okosnak.

A többi függvénynek is meg lehet nézni a definíciós részét, vagy akár újra lehet definiálni őket az előző fejezetben megismert módon.

Megjegyzés

A „gyári” függvények átdefiniálása nem tartós, azaz ha becsukjuk a PowerShell ablakot, majd újra megnyitjuk, akkor visszaáll az eredeti függvénydefiníció. Tartóssá változtatá-

sainkat úgy tehetjük, ha profil fájlban definiáljuk újra ezeket a függvényeket. Profilokról a *2.1.1 Profilok* fejezetben lesz szó.

1.9 Szkriptek

A függvények után nézzük a szkripteket. Amint korábban már említettük a PowerShell előre elkészített (fájlba mentett) parancssorozatait hívjuk szkriptnek, amelyeket közvetlenül is futtathatjuk, hasonlóan a függvényekhez. Ebben a fejezetben azokkal a szerkezetekkel fogunk megismerkedni, amelyek bár a függvényeknél is előfordultak, de használatuk elsősorban szkriptekben szokásos, bár természetesen minden begépelhető közvetlenül a parancssorba is.

1.9.1 Szkriptek engedélyezése és indítása

Készítsük is el mindjárt első szkriptünket! Olyan fájlt kell készítenünk, amelynek kiterjesztése `ps1`, ez a PowerShell szkriptek alapértelmezett kiterjesztése. Egyszerűbb esetben még a notepadra sincs szükség, létrehozhatjuk a fájl közvetlenül a PowerShellből is:

```
PS C:\> "`"Hurrá! Fut az első szkriptünk.`"" | Out-File elso.ps1
```

Hát ennyi. Látható, hogy nincs szükség túl sok cicomára, amit a parancssorba beírhatnánk, az minden további nélkül jó szkriptnek is[11]. Indítsuk is el gyorsan a szkriptet:

```
PS C:\> c:\elso.ps1
File C:\elso.ps1 cannot be loaded because the execution of scripts is disa
bled on this system. Please see "get-help about_signing" for more details.
At line:1 char:11
+ c:\elso.ps1 <<<<
```

Hoppá! A PowerShell olyan szkriptkörnyezet, amelyben nem engedélyezett a szkriptek futtatása?! Természetesen ez csak a „secure-by-default” jegyében beállított alapértelmezés, a szkriptekkel szemben tanúsított viselkedést a `Set-ExecutionPolicy` cmdlet segítségével szabályozhatjuk az alábbi négy állapot közül a megfelelő kiválasztásával:

```
PS C:\> Set-ExecutionPolicy -ExecutionPolicy Restricted
PS C:\> Set-ExecutionPolicy -ExecutionPolicy AllSigned
PS C:\> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
PS C:\> Set-ExecutionPolicy -ExecutionPolicy Unrestricted
```

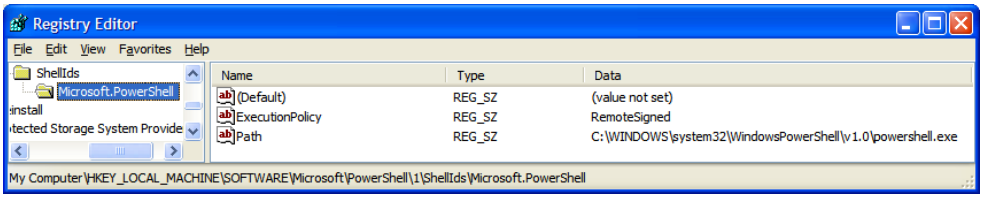
Az egyes paraméterek jelentése a következő:

- `Restricted` – A PowerShell semmiféle szkriptet nem futtathat (alapértelmezett állapot).

[11] A`" karakterek azért kellenek, hogy a nyitó és záró idézőjel is bekerüljön a fájlba. Az escape karakterek teljes listáját az about_escape_characters súgótéma tartalmazza. (A ` karaktert magyar billentyűzeten az AltGr+7 kombinációval tudjuk beírni.)

- `AllSigned` – csak digitális aláírással ellátott, megbízhatónak minősített szkriptek futtathatók.
- `RemoteSigned` – ha a szkript máshonnan származik (nem a lokális gépen készült, hálózati meghajtóról lett elindítva vagy az internetről lett letöltve), akkor aláírtnak kell lennie, de a helyben készült szkriptek korlátozás nélkül futtathatók.
- `Unrestricted` – minden szkript korlátozás nélkül futtatható.

A fenti biztonsági beállítást a registry-n keresztül is beállíthatjuk:



37. ábra Execution Policy helye a registry-ben

Vagy beállíthatjuk ezt az értéket központilag is Group Policy segítségével is. Ehhez letölthető a Microsoft downloads oldaláról az *„Administrative templates for Windows PowerShell”* adm fájl.

Állítsuk be a megfelelő biztonsági szintet a PowerShell cmdlet segítségével (legalább `RemoteSigned`) és próbáljuk meg ismét az indítást! (Természetesen csak rendszergazda jogosultságok birtokában tudjuk ezt megtenni.) Az előbb talán túl sokat is gépeltünk a fájl a c: gyökerében van, elég lesz csak ennyi:

```
 PS C:\> elso.ps1
The term 'elso.ps1' is not recognized as a cmdlet, function, operable prog
ram, or script file. Verify the term and try again.
At line:1 char:8
+ elso.ps1 <<<<
```

Hát ez nem elég! Régi jó UNIX / Linux szokás szerint a PowerShell nem próbálja az aktuális mappában megtalálni a megadott futtatható állományt, így teljes útvonalat (relatív vagy abszolút) kell megadnunk. Tehát például az alábbi módon indíthatjuk el végre a korábban létrehozott szkriptet:

```
PS C:\> ./elso.ps1
Hurrá! Fut az első szkriptünk.
```

Ha az aktuális mappában nem is, de a PATH környezeti változóban felsorolt mappákban természetesen próbálkozik a PowerShell, ilyen esetben nem szükséges útvonalat megadnunk.

Sajnos még mindig nem vagyunk teljesen készen, mivel ha szkriptünk útvonala szóköz karaktert is tartalmaz, újabb problémába ütközünk. Hozzunk létre egy újabb szkriptet a „`C:\Documents and Settings`” mappában!

```
PS C:\> "`"Hurrá! Fut a második szkriptünk.`"" | Out-File "c:\documents and
settings\masodik.ps1"
```

Rutinosabb versenyző persze nem is próbálkozik az idézőjelek nélküli indítással, ez azonban ebben az esetben kevés, mivel a parancs így nem parancs és nem is útvonal, csak egy közönséges karakterlánc:

```
PS C:\> "c:\documents and settings\masodik.ps1"
c:\documents and settings\masodik.ps1
```

Be kell vetnünk a korábban már használt „futtató” karaktert (`&`) az alábbi módon:

```
PS C:\> &"C:\Documents and Settings\masodik.ps1"
Hurrá! Fut a második szkriptünk.
```

Végül foglaljuk össze röviden mi is szükséges ahhoz, hogy PowerShell szkripteket futtassunk:

- Be kell állítanunk a megfelelő végrehajtási rendet (execution policy). Alapértelmezés szerint a PowerShell nem futtat le semmiféle szkriptet, akárhogy is adjuk meg az útvonalat.
- A szkript indításához adjuk meg annak teljes útvonalát, illetve ha a fájl az aktuális mappában van, használjuk a `.\` jelölést. Útvonal nélkül is elindíthatjuk a szkriptet, ha olyan mappában van, amely szerepel a Windows keresési útvonalán (PATH környezeti változó).
- Ha az útvonal szóközöket tartalmaz, tegyük idézőjelek közé és írjuk elé a futtató karaktert (`&`).

A korábbi szokásokkal (cmd, vbs, stb.) ellentétben (természetesen szintén biztonsági okokból) maga a szkriptfájl közvetlenül nem indítható el a shellen kívül, mivel a `ps1` kiterjesztésű fájlokat alapértelmezés szerint a notepad.exe nyitja meg[12]. Hogy lefuttathassuk szkriptünket, a powershell.exe-t kell elindítanunk (keresési útvonalon van), amelynek paraméterként adhatjuk át futtatandó szkriptfájl nevét. A cmd.exe-ből például az alábbi módon indíthatjuk el a fenti szkriptet (a `-noexit` kapcsoló hatására a powershell.exe nem áll le a szkript lefuttatása után, megfigyelhető, hogy a PowerShellre jellemző promptot kapunk vissza):

```
C:\>powershell -noexit c:\elso.ps1
Hurrá! Fut az első szkriptünk.
PS C:\>
```

Próbáljuk meg hasonló módon elindítani a `masodik.ps1` szkriptet is! Nyilván itt is kell a futtató karakter:

[12] Persze ez módosítható az összerendelés megváltoztatásával.

```
C:\>powershell -noexit &"c:\Documents and Settings\masodik.ps1"
Windows PowerShell
Copyright (C) 2006 Microsoft Corporation. All rights reserved.

PS C:\>
```

Ugyan látható a promptból, hogy a cmd shell átalakult PowerShellé, de sajnos a szkriptünk így sem futott le. Még egy trükkre van szükség, a cmd.exe elől el kell rejteni a PowerShellnek szóló `&` jelet egy újabb, külső idézőjelpárral:

```
C:\>powershell -noexit "& 'c:\Documents and Settings\masodik.ps1'"
Hurrá! Fut a második szkriptünk.
PS C:\>
```

1.9.2 Változók kiszippantása a szkriptekből (dot sourcing)

A függvényeknél már látott, de a scripteknél még gyakrabban alkalmazott dotsourcing lehetőséget elevenítsük fel ismét. Ehhez hasonló szolgáltatással korábban (cmd, VBScript, stb.) egyáltalán nem találkozhattunk. Készítsük el a következő szkriptet, akár a notepad segítségével:

```
$a = "egyik-"
$b = "másik"
$c = $a + $b
"`$c = $c"
```

Mentsük el a fájlt, mondjuk a `c:\scripts` mappába `source.ps1` néven! Ezután futtassuk le az elkészült szkriptet, ami kiírja a `$c` változó tartalmát, majd a szkript lefutása után nézzük meg, milyen érték van a `$c` változóban! A helyes tipp az, hogy semmilyen. Ez így természetes, a szkript és a konzol külön munkamenetnek tekintendő, a szkript a konzol „gyerekkörnyezete”, a konzol ezért nem éri el a gyereke adatait.

```
PS C:\> c:\scripts\source.ps1
$c = egyik-másik
PS C:\> $c
PS C:\>
```

Eddig tehát tulajdonképpen semmi meglepőt nem tapasztaltunk, inkább az lett volna furcsa, ha nem így van. Ha azonban egy speciális módon indítjuk el szkriptünket, akkor váratlan jelenség szemtanúi lehetünk. Indítsuk el most a szkriptet úgy, hogy a parancsot egy pont és egy szóköz karakterrel kezdjük.

```
PS C:\> . c:\scripts\source.ps1
$c = egyik-másik
PS C:\> $c
egyik-másik
```

Hoppá! A konzol munkamenet „tudja”, hogy mi volt a változó értéke a szkriptben, ami már réges-rég véget ért. Ez a jelenség a függvényeknél már megismert „dotsourcing” (*1.8.4 Függvények láthatósága, „dotsourcing”* fejezet), amelynek hatására a szkript valamennyi változója az őt meghívó környezet változójává alakul. Jelen esetben globálissá vált, vagyis minden munkamenet (így a konzol és más szkriptek is) elérhetik, felhasználhatják és megváltoztathatják a bennük tárolt értékeket még akkor is, amikor az őket létrehozó munkamenet (vagyis a szkript) már bezárult.

Talán ez már nyilvánvaló, hogy milyen hasznos ez a lehetőség, hiszen így a szkript által lekérdezett, létrehozott adatszerkezetek (például egy adatbázisból, vagy az Active Directoryból származó adathalmaz) a szkript lefuttatása után interaktív üzemmódban is listázhatók, feldolgozhatók. Vagyis egyszerre élvezhetjük a szkript- és az interaktív üzemmód előnyeit. Ha egy feladat egyik részét inkább szkriptként érdemes elkészíteni, de más részeknél egyszerűbb az interaktív üzemmód használata, akkor bátran használhatjuk ezt a lehetőséget, mindent olyan módon végezhetünk el, ahogy az a legkedvezőbb..

1.9.3 Paraméterek átvétele és a szkript által visszaadott érték

Szkriptjeinknek is lehetnek paraméterei, melyeket – hasonlóan a függvényekhez - az `$args` változóban érhetjük el. A `$args` egy tömb, amelybe tetszés szerinti típusú értékek kerülhetnek, a megjelenő típus a beírt értéktől függ, ezért kényesebb esetekben célszerű lehet a paraméterek számán kívül azok típusát is ellenőrizni a további ténykedés előtt. Az alábbi szkriptben csak a megfelelő számú paraméter meglétét ellenőrizzük:

```
if ($args.Length -ne 3)
{
        Write-Error "A szkript csak 3 paraméterrel indítható!"
        return "Hibás futás!"
}
Write-Host $args.GetType()
foreach($arg in $args)
{
        Write-Host $arg
        Write-Host $arg.GetType()

}
return "Jó futás!"
```

Ha a paramétereket tároló tömb nem három elemből áll, akkor hibaüzenetet írunk ki, és a `return` kulcsszó után megadjuk a szkript által visszaadott értéket (a `return` utáni

rész már nem fut le). Paraméter nélküli indítási kísérlet esetén a következő hibaüzenet jelenik meg:

```
PS C:\> ./parameter.ps1
C:\parameter.ps1 : A szkript csak 3 paraméterrel indítható!
At line:1 char:15
+ ./parameter.ps1 <<<<
Hibás futás!
```

Ha megvan mindhárom paraméter, akkor visszaírjuk azok típusát és értékét a képernyőre, a szkript pedig egy másik értéket fog visszaadni.

```
PS C:\> ./parameter.ps1 egy kettő három
System.Object[]
egy
System.String
kettő
System.String
három
System.String
Jó futás!
```

A visszaadott érték a fenti esetekben egyszerűen a képernyőre került, de ez nem igazán erre való. Az érték segítségével a hívó szkript kaphat információt az elindított szkript belsejében történtekről. Természetesen semmi akadálya nincsen több érték visszaadásának sem (bár csak egyetlen `return` futhat le), szkriptünk egy tömb elemeinek képében tetszőleges számú és típusú értéket is visszaadhat.

```
PS C:\> $vissza = ./parameter.ps1 egy kettő három
System.Object[]
egy
System.String
kettő
System.String
három
System.String
PS C:\> $vissza
Jó futás!
```

Természetesen nem csak az `$args` változó áll rendelkezésünkre, hanem használhatunk explicit paraméterdefiníciót is. Miután a szkriptnek nincs saját, „belső” neve, hanem csak a fájlnak, amibe rakjuk, ezért a paramétereit se tudjuk a nem létező neve mellett felsorolni, csak külön `param` kulcsszó megadásával:

```
param ($a, $b)
$a / $b
```

Elmentetem a fenti, nagyon bonyolult szkriptet „osztás.ps1” néven, és meg is hívhatom, hasonlóan egy függvényhez:

```
[6] PS C:\old>.\osztás.ps1 20 4
5
[7] PS C:\old>.\osztás.ps1 -b 3 -a 27
9
```

Akár a hely szerinti, akár a név szerinti paraméterátadás is működik. Természetesen a szkript által visszaadott eredményt felhasználhatjuk értékadás során is, megint csak ugyanúgy, mintha függvény lenne:

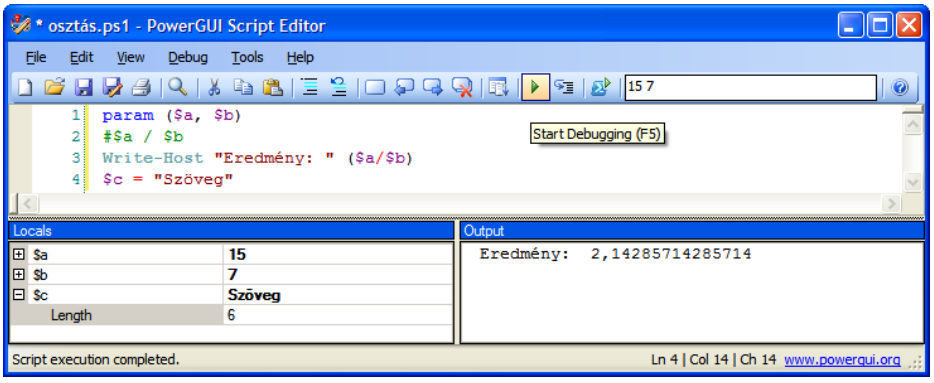
```
[8] PS C:\old>$eredmény = .\osztás 50 11
[9] PS C:\old>$eredmény
4,54545454545455
```

1.9.4 Szkriptek írása a gyakorlatban

Szkripteket a gyakorlatban ritkán írunk magával a PowerShell konzolablakon, még csak nem is a Notepad az ideális eszköz, hanem valamilyen szkriptszerkesztő. A *1.3 Segédprogramok* fejezetben már említett PowerGUI Script Editora talán a legegyszerűbb ilyen jellegű eszköz, ami ráadásul pont eleget tud ahhoz, hogy hatékonyan használhassuk. Ráadásul ingyenes!

1.9.4.1 PowerGUI Script Editor

A korábbi „osztás” szkriptpéldát kicsit még kiegészítettem, mindez így néz ki a PowerGUI Script Editorban:



38. ábra Szkriptem a PowerGUI Script Editorában

Sajnos a könyv nem színes, így nem látható a fenti ábra színeinek kavalkádja, de külön színnel jelzi a program a kulcsszavakat, cmdleteket, változókat, megjegyzéseket, operátorokat.

Az eszközsorban látható „Start Debugging (F5)” gombbal lehet futtatni ott helyben a szkriptet. Az eszközsorban, a beviteli mezőben lehet paramétereket is átadni (a képen ez

most „15 7”). A futás során generálódó output kikerül a jobb alsó ablakrészbe és a belső változók tartalma is azonnal megtekinthető a baloldali részben. A jobb szemléltethetőség kedvéért még egy felesleges `$c` változót is felvettem, hogy látható legyen az, ahogyan a „Locals” részben a változókban tárolt objektumok tulajdonságai is azonnal elérhetők és kiolvashatók, nem kell állandóan `get-member`-t futtatni.

1.9.4.2 Megjegyzések, kommentezés (#)

A fájlba mentett parancssorozatainkat érdemes kommentekkel ellátni, hogy ha pár hónap után nyitjuk meg őket, akkor is értsük, hogy mit csinál a szkript, a paraméterek, változók hogyan használhatók.

A komment jele a # jel.

```
[10] PS C:\old>$a = 5 #értékadás
[11] PS C:\old>$a
5
```

A fejlesztés során a megjegyzéseket, kommenteket arra is felhasználhatjuk, hogy különböző kódváltozatok tesztelésekor az egyik változatot „kikommentezzük”, míg a másikat teszteljük, aztán meg cserélünk, és megnézhetjük, hogy a másik változat hogyan működik. Vagy az is gyakori, hogy a szkriptünk kifejlesztése során jóval több átmeneti változó és állapot értékét íratjuk ki, míg a végleges verziónál ezt már nem szeretnénk. Ilyenkor szintén jól jön a „kikommentezési” lehetőség. Kitörölni nem akarjuk ezeket a kiíratási lehetőségeket, mert hátha jól jönnek majd a szkript továbbfejlesztésekor.

Ezt a gyors ki-be kommentezést jól támogatja a PowerGUI Script Editora, az eszközsorban található gombokkal egy kattintással tudjuk ezt ki-be kapcsolni, ráadásul több sor kijelölésével is működik.

Sajnos a # jel hatása csak az adott sorra érvényes a sor végéig. Ha többsoros kommenteket szeretnénk beszúrni, akkor vagy minden sort külön ellátunk ilyen jellel, vagy esetleg használhatjuk a „*here string*” formátumot, méghozzá semmivé konvertálva:

```
[void] "@ nagyon hosszú komment
igazából string
de nem hajtódik végre, mert a nihilbe küldöm
@"
```

1.9.5 Adatbekérés (Read-Host)

A szkriptek esetében gyakori, hogy egy kis interakciót várunk el tőlük, azaz ne kelljen mindig parancssorban felparaméterezni őket, hanem kérdezzék meg tőlünk, hogy mit szeretnének. Az első példában egy kis interaktív vezérlést mutatok be egy kis menüstruktúra segítségével:

```
do
{
        Write-Host
"============================================================="
        Write-Host "0 - Folyamatok listázása"
        Write-Host "1 - Szolgáltatások listázása"
        Write-Host "2 - Folyamat indítása"
        Write-Host "3 - Folyamat leállítása"
        Write-Host "9 - Kilépés"
        $v = Read-Host
        Write-Host
"============================================================="
        switch ($v)
        {
                "0" {Get-Process}
                "1" {Get-Service}
                "2" {&(Read-Host "Folyamat neve")}
                "3" {Stop-Process -name (&Read-Host "Folyamat neve")}
                "9" {break}
                default {Write-Host "Érvénytelen billentyű!";break  }
        }
}       while ($v -ne "9")
```

A szkript egy jó kis hátul tesztelő `do-while` ciklus, ami egészen addig fut, amíg a középtájon található `Read-Host` cmdletnek nem adunk át egy 9-est.

Ha 0, 1, 2, 3 értékek valamelyikét adjuk be neki, akkor egy `switch` kifejezés a megfelelő PowerShell cmdletet futtatja le. Ráadásul bizonyos parancsok még további paramétert igényelnek, így azokat egy újabb `Read-Host` kéri be.

A `Read-Host` az nem billentyű leütésére vár, hanem komplett sort kér be Enter leütéséig, és addig vár, amíg ezt meg nem kapja. A fenti példában látható, hogy nem muszáj a `Read-Host` kimenetét változóba tölteni, azt azon melegében is felhasználhatjuk.

Például a „2”-es választásával rögtön meghívjuk azt, amit begépeltünk:

```
&(Read-Host "Folyamat neve")
```

Ugye itt megint használjuk a „végrehajtatás” operátorát, az `&` jelet.

A `Read-Host`-nak van egy szöveg paramétere, ezt teszi fel kérdésként a sor beadására való várakozás előtt.

1.9.6 Szkriptek digitális aláírása

Láttuk a fejezet elején, hogy a szkriptek futtatását engedélyezni kell. Ha a legbiztonságosabb futtatási házirenddel dolgozunk, akkor csak olyan szkript futhat, amelyik digitálisan alá van írva, és az aláírást igazoló tanúsítványt az adott gép elismeri biztonságos tanúsítványnak. Na de honnan lesz nekünk ilyen tanúsítványunk?

Az első lehetőség, hogy van már kiépített PKI infrastruktúránk, amelyben van „code signing” típusú tanúsítványt kiosztani képes CA kiszolgáló. Innen igényelni kell ilyen tanúsítványt, és ezzel fogunk tudni dolgozni a később leírt módon.

A problematikusabb helyzet az, amikor nincs ilyen PKI infrastruktúránk. Például én otthon szeretnék aláírt szkripteket létrehozni, hogy tesztelhessem a működésüket. Ebben az esetben létrehozhatok u.n. önaláírt tanúsítványt, amelyet csak azok a gépek fognak elismerni hitelesnek, ahova manuálisan telepítem azokat.

Nézzünk ennek lépéseit! Honnan szedek én önaláírt tanúsítványt? Direkt ilyen tesztelési célokra van a .NET Framework Software Development Kit-ben (letölthető: http://msdn2.microsoft.com/en-us/netframework/aa731542.aspx) egy `makecert.exe` segédprogram, amely erre képes.

Ez egy parancssori eszköz, nézzük meg a meghívásának paramétereit:

```
C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0>"C:\Program Files\Micros
oft Visual Studio 8\SDK\v2.0\Bin\makecert.exe" -!
Usage: MakeCert [ basic|extended options] [outputCertificateFile]
Extended Options
 -sc  <file>          Subject's certificate file
 -sv  <pvkFile>       Subject's PVK file; To be created if not present
 -ic  <file>          Issuer's certificate file
 -ik  <keyName>       Issuer's key container name
 -iv  <pvkFile>       Issuer's PVK file
 -is  <store>         Issuer's certificate store name.
 -ir  <location>      Issuer's certificate store location
                         <CurrentUser|LocalMachine>.  Default to 'CurrentUser
'
 -in  <name>          Issuer's certificate common name.(eg: Fred Dews)
 -a   <algorithm>     The signature algorithm
                         <md5|sha1>.  Default to 'md5'
 -ip  <provider>      Issuer's CryptoAPI provider's name
 -iy  <type>          Issuer's CryptoAPI provider's type
 -sp  <provider>      Subject's CryptoAPI provider's name
 -sy  <type>          Subject's CryptoAPI provider's type
 -iky <keytype>       Issuer key type
                         <signature|exchange|<integer>>.
 -sky <keytype>       Subject key type
                         <signature|exchange|<integer>>.
 -l   <link>          Link to the policy information (such as a URL)
 -cy  <certType>      Certificate types
                         <end|authority>
 -b   <mm/dd/yyyy>    Start of the validity period; default to now.
 -m   <number>        The number of months for the cert validity period
 -e   <mm/dd/yyyy>    End of validity period; defaults to 2039
 -h   <number>        Max height of the tree below this cert
 -len <number>        Generated Key Length (Bits)
 -r                   Create a self signed certificate
 -nscp                Include netscape client auth extension
 -eku <oid[<,oid>]>   Comma separated enhanced key usage OIDs
 -?                   Return a list of basic options
 -!                   Return a list of extended options
```

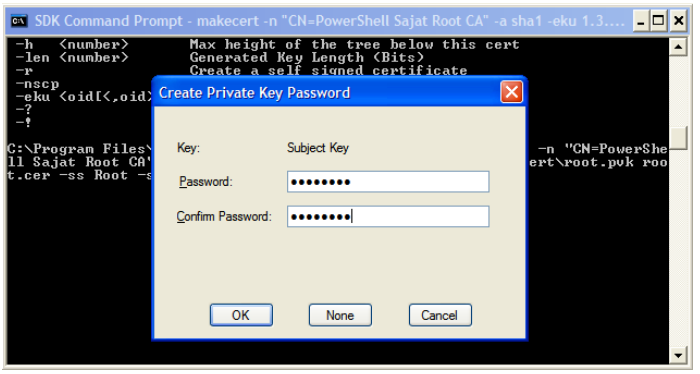
Nem egyszerű, de kezdjünk neki. Első lépésként a `makecert`-et rá kell bírni, hogy tanúsítványkiosztó hatóság szerepét vegye fel:

```
makecert -n "CN=PowerShell Sajat Root CA" -a sha1 -eku 1.3.6.1.5.5.7.3.3 -r
-sv c:\cert\root.pvk c:\cert\root.cer -ss Root -sr localMachine
```

Nézzük az egyes paraméterek jelentését:

Paraméter	Jelentés
-n "CN=PowerShell Sajat Root CA"	A tanúsítványkiosztóm neve
-a sha1	A digitális aláírás algoritmusa
-eku 1.3.6.1.5.5.7.3.3	A kiosztandó tanúsítványok fajtája, ez a szám jelenti a „Code signing” lehetőséget
-r	Önaláírt legyen a tanúsítványkiosztó tanúsítvány
-sv c:\cert\root.pvk c:\cert\root.cer	A tanúsítványkiosztó tanúsítványainak tárolási helye fájl szinten
-ss Root	A tanúsítványkiosztó tanúsítványainak tárolási helye a tanúsítványtárban
-sr localMachine	Méghozzá a helyi gépen

Amikor ezt a parancsot futtatom, akkor kétszer is megjelenik egy jelszóbekérő ablak, ami a privát kulcsok kezelésének védelmét szolgálja. Az első esetben adjuk meg a privát-kulcshoz való hozzáférés jelszavát:



39. ábra Makecert.exe futtatása tanúsítványkiosztó szerep céljára

A második esetben pedig már használja is a háttérben a `makecert` ezt a privát kulcsot, ezért kéri be az előbb megadott jelszót:

40. ábra Jelszóbeadás a privát kulcshoz való hozzáféréskor

Tehát idáig azt csináltuk, hogy létrehoztunk egy tanúsítványkiosztó önaláírt tanúsítványt, amellyel alá lesznek írva az ezután legenerálható, immár ténylegesen kódaláírásra szánt tanúsítványaink. Hozzunk is mindjárt létre egy ilyet:

```
makecert -pe -n "CN=Soos Tibi PS Guru" -ss PSCS -a sha1 -eku
1.3.6.1.5.5.7.3.3 -iv c:\cert\root.pvk -ic c:\cert\root.cer
```

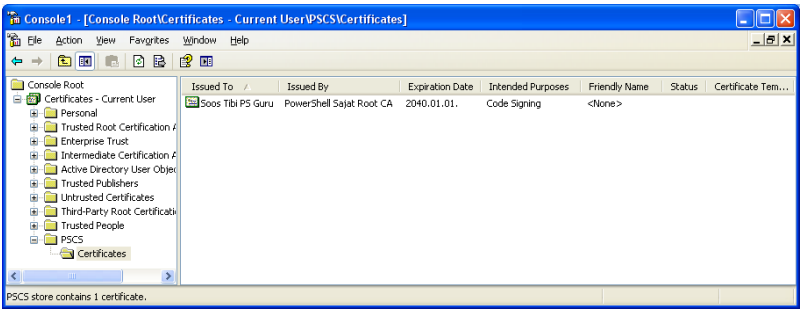
Nézzük ezeknek a jelentését:

Paraméter	Jelentés
-pe	Exportálható privát kulcsot hoz létre (exportálással biztonságba tudjuk helyezni)
-n "CN=Soos Tibi PS Guru"	A kódaláíró tanúsítványon szereplő név
-ss PSCS	A tanúsítvány tárolásának a helye, én itt egy új tárolót hoztam létre, ha személyes tárolóba szeretném berakni, akkor arra a „My” névvel lehet hivatkozni
-a sha1	Digitális aláírás algoritmusa
-eku 1.3.6.1.5.5.7.3.3	Tanúsítvány célja (code signing)
-iv c:\cert\root.pvk	A kiosztó hatóság privát kulcsa
-ic c:\cert\root.cer	Az én tanúsítványomat viszonthitelesítő tanúsítvány

Miután ezen parancs futtatása során is használjuk az első lépésben létrehozott privát kulcsot, így újra meg kell adni az ott megadott jelszót:

41. ábra A kódaláíró tanúsítvány generálása során megjelenő jelszóbekérő

Ha ezt is sikeresen lefuttattam, akkor megnézhetem, hogy a tanúsítványtáramban tényleg ott van-e a kódaláíró tanúsítvány:



42. ábra Kódaláíró tanúsítványom a tanúsítványtárban

Akkor most már van alkalmas kódaláíró tanúsítványom. Itt csatlakoznak be azok az olvasók, akiknek van „igazi” PKI infrastruktúrájuk, és egy normál tanúsítványkiosztótól szereztek kódaláíró tanúsítványt.

Most lehet aláírni a szkriptemet. Elsőként ragadjuk meg az aláírásra szánt tanúsítványunkat (itt nagyon nagy segítség a TAB-kiegészítés, azaz nem kell azt a hosszú számsort begépelni, ha a „könyvtárra” rálelünk, akkor elég a TAB-bal léptetni):

```
[9] PS C:\old> $signcert = (get-item cert:\CurrentUser\PSCS\8ED2798A04D5F794
DA6C8C3C167EB033CB8BE6C2)
[10] PS C:\old> $signcert


    Directory: Microsoft.PowerShell.Security\Certificate::CurrentUser\PSCS


Thumbprint                                Subject
----------                                -------
8ED2798A04D5F794DA6C8C3C167EB033CB8BE6C2  CN=Soos Tibi PS Guru
```

És akkor következzen a lényeg, írjuk alá a szkriptet az előbb betöltött tanúsítvánnyal:

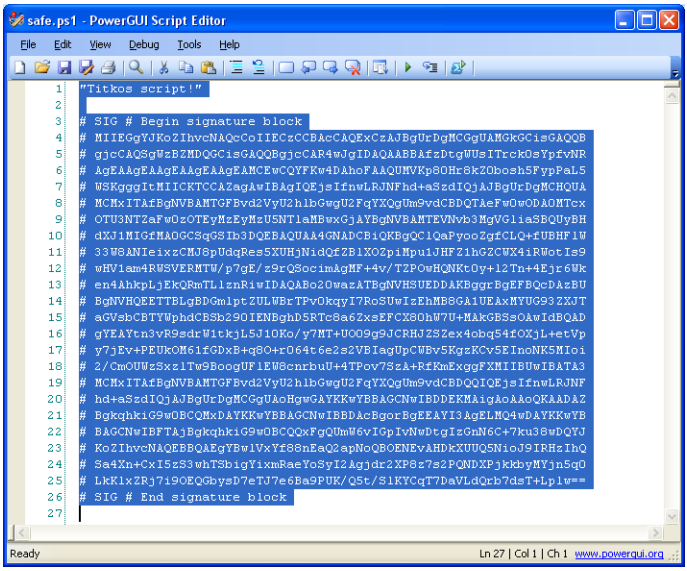
```
[11] PS C:\old> Set-AuthenticodeSignature safe.ps1 $signcert


    Directory: C:\old


SignerCertificate                         Status           Path
-----------------                         ------           ----
8ED2798A04D5F794DA6C8C3C167EB033CB8BE6C2  Valid            safe.ps1
```

Állítsuk át a végrehajtási házirendet úgy, hogy mindenképpen megkövetelje az aláírást, és futtatom a `safe.ps1` szkriptemet:

```
[12] PS C:\old> Set-ExecutionPolicy allsigned
[13] PS C:\old> .\safe.ps1

Do you want to run software from this untrusted publisher?
File C:\old\safe.ps1 is published by CN=Soos Tibi PS Guru and is not
trusted on your system. Only run scripts from trusted publishers.
[V] Never run  [D] Do not run  [R] Run once  [A] Always run  [?] Help
(default is "D"):a
Titkos script!
```

Miután ez egy önaláírt tanúsítványon alapuló tanúsítvány, így a rendszer rákérdez, hogy elfogadom-e hitelesnek? Én azt válaszoltam, hogy igen, sőt, minden alkalommal fogadja el ezt hitelesnek (`Always`) innentől kezdve.

Nézzük meg, hogy hogyan is néz ki egy ilyen aláírt szkript:



43. ábra Az elektronikusan aláírt szkript

Az eredeti szkriptem nagyon egyszerű volt, csak annyit csinált, hogy kiírta „Titkos script!”. Ez olvasható is az első sorban. Ami utána áll az az elektronikus aláírás.

Ha bármit módosítok ebben a szkriptben, akár egy karaktert is, és futtatni akarom, akkor a következő hibajelzést fogom kapni:

```
[14] PS C:\old> .\safe.ps1
File C:\old\safe.ps1 cannot be loaded. The contents of file C:\old\safe.ps1
 may have been tampered because the hash of the file does not match the has
h stored in the digital signature. The script will not execute on the syste
m. Please see "get-help about_signing" for more details..
At line:1 char:10
+ .\safe.ps1 <<<<
```

Ha sértetlen a szkriptem, akkor ellenőrizhetem, hogy ki is írta alá, azaz értelmeztethetem azt a sok zűrös karaktert a szkriptem végén:

```
[17] PS C:\old> Get-AuthenticodeSignature C:\old\safe.ps1 | fl


SignerCertificate      : [Subject]
                           CN=Soos Tibi PS Guru

                         [Issuer]
                           CN=PowerShell Sajat Root CA

                         [Serial Number]
                           123B087E7C0B44934585DF9A4B374842

                         [Not Before]
                           2008.04.17. 21:57:56

                         [Not After]
                           2040.01.01. 0:59:59

                         [Thumbprint]
                           8ED2798A04D5F794DA6C8C3C167EB033CB8BE6C2

TimeStamperCertificate :
Status                 : Valid
StatusMessage          : Signature verified.
Path                   : C:\old\safe.ps1
```

1.9.7 Végrehajtási preferencia

Beszéltünk az álnevekről, cmdletekről, függvényekről, szkriptekről. Láttuk azt is, hogy a PowerShell ugyanúgy végrehajtja az elérési úton található futtatható állományokat, mint ahogy a hagyományos parancssor.

Azt is láttuk, hogy elnevezések tekintetében elég szabad a PowerShell, nagyon kevés megszorítás van a neveket illetően, így például lehet egyforma neve egy általunk létrehozott függvénynek és aliasnak, mint egy meglevő cmdletnek. Például elnevezhetek egy függvény így is:

```
[14] PS C:\old> function script.ps1 {"Ez most függvény"}
[15] PS C:\old> script.ps1
Ez most függvény
```

Annak ellenére is, hogy ugyanilyen névvel van nekem egy szkriptem is, ráadásul éppen az aktuális könyvtárban:

```
[16] PS C:\old> Get-Content script.ps1
"Ez most a szkript"
```

Ennek ellenére a [15]-ös sorban a „script.ps1” beírására nem a szkriptem, hanem a függvényem futott le.

És ez még nem elég, ugyanilyen névvel létrehozhatok egy álnevet is:

```
[17] PS C:\old> New-Alias script.ps1 get-command
```

Ha most futtatom a „script.ps1”-et, akkor az alias fut le:

```
[18] PS C:\old> script.ps1

CommandType     Name                            Definition
-----------     ----                            ----------
Cmdlet          Add-Content                     Add-Content [-Path] <Strin...
Cmdlet          Add-History                     Add-History [[-InputObject...
Cmdlet          Add-Member                      Add-Member [-MemberType] <...
Cmdlet          Add-PSSnapin                    Add-PSSnapin [-Name] <Stri...
Cmdlet          Clear-Content                   Clear-Content [-Path] <Str...
...
```

Hogyan lehet mégis a szkriptet lefuttatni? Hát például így:

```
[19] PS C:\old> &(get-item script.ps1)
Ez most a szkript
```

Hogyan tudom a függvényt futtatni? Hasonló logikával, mint a szkriptet, csak más a tárolásának helye:

```
[20] PS C:\old> &(get-item function:script.ps1)
Ez most függvény
```

Nézzük összefoglalva a végrehajtási preferenciát:

1. Alias
2. Függvény
3. Cmdlet
4. Futtatható fájlok (exe, com, stb.)
5. Szkriptek
6. Kiterjesztés alapján a hozzá tartozó alkalmazás futtatása

Azaz nagyon vigyázni kell, hogy milyen néven hozunk létre álneveket és függvényeket, mert ezek a végrehajtási preferenciában megelőzik a cmdleteket is. Például egy gonosz szkript átdefiniálhatja a gyári cmdleteket ugyanolyan nevű álnevekkel vagy függvényekkel és a PowerShell teljesen másként kezd el viselkedni, mint ahogy várjuk.

1.10 Fontosabb cmdletek

A PowerShell főbb nyelvi elemeit áttekintettük az előző fejezetekben, most szemelgessünk a legfontosabb cmdletek között. Ezek a cmdletek annyira általánosan használhatók, hogy szinte nyelvi elemként lehet őket tekinteni, ezért mindenképpen itt, az elméleti részben kell őket tárgyalni.

1.10.1 Csővezeték feldolgozása (Foreach-Object) – újra

Ugyan már volt szó a `ForEach-Object` cmdletről a *1.7.5 ForEach-Object cmdlet* fejezetben, de most már ismerjük a függvényeket és filtereket, így tisztában vagyunk azzal, hogy hogyan történik a csővezetéken érkező objektumok feldolgozása paraméterként átadott szkriptblokkok segítségével.

Tulajdonképpen már mi magunk is meg tudnánk írni egy `ForEach-Object` cmdlethez hasonló működésű függvényt, de mivel ezt olyan gyakran használnánk, hogy érdemesebb volt ez gyorsabban lefutó, natív PowerShell parancsként implementálni.

Ennek ellenére elég tanulságos lenne egy ilyen függvényt írni. Ehhez először nézzük meg a `ForEach-Object` szintaxisát:

```
[1] PS C:\> (get-help foreach-object).syntax

ForEach-Object [-process] <ScriptBlock[]> [-inputObject <psobject>] [-begin
 <scriptblock>] [-end <scriptblock>] [<CommonParameters>]
```

Most a `CommonParameters` részt kihagyom, de a többit én is definiálom a függvényemben:

```
[2] PS C:\> function saját-foreachobject ([scriptblock] $process = $(throw "
Kötelező!"), [PSObject] $inputObject, [scriptblock] $begin, [scriptblock] $e
nd)
>> {
>>     begin
>>     {
>>       if($inputObject)
>>       {
>>          $inputObject | saját-foreachobject -p $process -b $begin -e $end
>>          return
>>       }
>>       if($begin) {&$begin}
>>     }
>>     process {&$process}
>>     end
>>     {
>>         if($end) {&$end}
>>     }
>> }
>>
```

Itt a `$process` paraméter kötelezően kitöltendő, hibát ad a függvényem, ha ez hiányzik:

```
[3] PS C:\> 222 | saját-foreachobject
Kötelező!
At line:1 char:63
+ function saját-foreachobject ([scriptblock] $process = $(throw  <<<< "Köt
elező!"), [PSObject] $inputObject, [scriptblock] $begin, [scriptblock] $end
)
```

Ha ez helyesen van kitöltve, akkor akár csővezetékkel is jól működik:

```
[4] PS C:\> 1,3,44, "q" | saját-foreachobject {$_*3}
3
9
132
qqq
```

De működik úgy is, ha nem „belecsövezzük” a feldolgozandó objektumot, hanem `-inputobject` paraméterként átadjuk:

```
[5] PS C:\> saját-foreachobject {$ *2} -i 654
1308
```

Ehhez a függvénydefinícióban egy kicsit trükköztem, ha ez utóbbi módszerrel akarnék objektumot átadni, akkor nem lenne `$_` változó, ezért a `begin` szekcióban meghívom rekurzív módon a függvényből saját magát, immár csövezős módszerrel. Azért, hogy a rekurzív hívásból visszatérve ne fusson le még egyszer a függvényem, ezért ott egy `return`-nel kilépek.

1.10.2 A csővezeték elágaztatása (Tee-Object)

A `Tee-Object` cmdlet a csővezeték megcsapolására szolgál. Tetszés szerinti pontra beillesztve az arra járó objektumkupacot egy változóba vagy egy megadott fájlba írja, de eközben változatlan formában továbbküldi a csövön is, a következő parancs pontosan úgy kapja meg az objektumokat, mintha a `Tee-Object` ott sem lett volna. Az alábbi parancs például a c: meghajtó mappalistáját kérdezi le, a `Tee-Object` ennek szöveges megfelelőjét beleírja a c:\dir.txt fájlba, a lekérdezett eredeti objektumokat pedig továbbadja a `Where-Object` cmdletnek:

```
PS C:\> Get-ChildItem | Tee-Object -FilePath c:\dir.txt | Where-Object
{$_.Name -eq "Windows"}

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----       2007.07.16.     22:42            WINDOWS
```

Leggyakrabban a fentihez hasonló esetekben használjuk ezt a cmdletet, azaz amikor a kimenetet kétfelé szeretnénk ágaztatni: egyrészt például naplózási céllal beirányítjuk a csővezeték tartalmát egy fájlba, másrészt valami egyéb tevékenységet is végzünk ezekkel az objektumokkal.

1.10.3 Csoportosítás (Group-Object)

A `Group-Object` cmdlet segítségével egy (vagy több) megadott tulajdonság értéke szerint csoportosíthatunk objektumokat. A csoportokba az azonos tulajdonságértékkel rendelkező objektumok kerülnek.

? **Feladat:** Készítsünk listát, amely a futó és a leállított szolgáltatások számát tartalmazza!

A szolgáltatásokat lekérdező `Get-Service` kimenetét a `Group-Object`-nek kell megkapnia, paraméterként pedig meg kell adnunk, hogy melyik tulajdonság értékei szerint akarjuk elvégezni a csoportosítást. Az alapértelmezett listában is szerepel a `Status` tulajdonság, a feladat szerint ezt kell majd megadnunk, de természetesen bármelyik, a listában szereplő egyéb tulajdonság szerint csoportosíthatnánk:

```
PS C:\> get-service | group-object -property status

Count Name                      Group
----- ----                      -----
   42 Stopped                   {Alerter, AppMgmt, aspnet_state, CiSvc...}
   59 Running                   {ALG, AudioSrv, BITS, Browser...}
```

? **Feladat:** Készítsünk listát, amely a PowerShell súgótémával rendelkező, különféle típusú elemeinek darabszámát tartalmazza!

A fenti feladatban tulajdonképpen semmi újdonság nincsen a megelőzőhöz képest, csak azért került ide, mert igen szép példát ad a PowerShell objektumkezelésének szinte tökéletesen egységes voltára. Mindegy, hogy rendszerszolgáltatásokról, vagy a PowerShell súgótémáinak listájáról van szó, a szükséges parancsok gyakorlatilag azonosak, a tökéletesen különböző típusú adatoktól függetlenül. A megoldás:

```
PS C:\> get-help * | group-object -property category

Count Name                      Group
----- ----                      -----
  102 Alias                     {ac, asnp, clc, cli...}
  129 Cmdlet                    {Get-Command, Get-Help, Get-History, In...
    7 Provider                  {Alias, Environment, FileSystem, Functi...
   56 HelpFile                  {about_alias, about_arithmetic_operator...
```

Ha valakinek ez sem elég, akkor még olyat is lehet csinálni a `Group-Object` segítségével, hogy egyszerre több szempont szerinti csoportot is létrehozhatunk:

```
[18] PS C:\> get-process | Group-Object -Property processname, company

Count Name                      Group
----- ----                      -----
    1 alg, Microsoft Corpora... {alg}
    1 ATKOSD,                   {ATKOSD}
    1 csrss                     {csrss}
    1 daemon, DT Soft Ltd       {daemon}
    1 explorer, Microsoft Co... {explorer}
    2 GoogleDesktop, Google     {GoogleDesktop, GoogleDesktop}
    1 gStart, GARMIN Corp.      {gStart}
    1 HControl,                 {HControl}
    1 hh, Microsoft Corporation {hh}
    1 Idle                      {Idle}
    1 inetinfo, Microsoft Co... {inetinfo}
    1 install, Microsoft Cor... {install}
    1 lsass, Microsoft Corpo... {lsass}
    1 msiexec, Microsoft Cor... {msiexec}
    1 msmsgs, Microsoft Corp... {msmsgs}
    1 NetSDK64setup, Microso... {NetSDK64setup}
    1 NMSAccessU                {NMSAccessU}
    1 nvsvc64, NVIDIA Corpor... {nvsvc64}
    1 powershell, Microsoft ... {powershell}
    1 Quest.PowerGUI.ScriptE... {Quest.PowerGUI.ScriptEditor}
    1 RTHDCPL, Realtek Semic... {RTHDCPL}
    2 rundll32, Microsoft Co... {rundll32, rundll32}
    1 scardsvr, Microsoft Co... {scardsvr}
    1 services, Microsoft Co... {services}
    1 sm56hlpr, Motorola Inc.   {sm56hlpr}
    1 smss, Microsoft Corpor... {smss}
    1 splwow64, Microsoft Co... {splwow64}
    1 spoolsv, Microsoft Cor... {spoolsv}
    1 StkCSrv, Syntek Americ... {StkCSrv}
    8 svchost, Microsoft Cor... {svchost, svchost, svchost, svchost...}
    1 System                    {System}
    1 vmh, Microsoft Corpora... {vmh}
    1 vssrvc, Microsoft Corp... {vssrvc}
    1 winlogon                  {winlogon}
    1 WINWORD, Microsoft Cor... {WINWORD}
    1 wmiprvse, Microsoft Co... {wmiprvse}
    1 wscntfy, Microsoft Cor... {wscntfy}
```

A fenti paranccsal a processzek neve és gyártójuk alapján együttesen csoportosítottam a futó folyamatokat. Ez nem igazi kétszintű csoportosítás, hanem képez egy „látszólagos” tulajdonságot, amely a processz nevéből és gyártójából áll, és ezen összetett tulajdonság alapján csoportosít.

Ha nincs szükségünk az egyes elemekre, csak a csoportokra, akkor használhatjuk a `-NoElement` kapcsolót:

```
[21] PS C:\> get-process | Group-Object -Property company -noelement

Count Name
----- ----
   30 Microsoft Corporation
```

```
    2
    5
    1 DT Soft Ltd
    2 Google
    1 GARMIN Corp.
    1 NVIDIA Corporation
    1 Quest Software
    1 Realtek Semiconductor ...
    1 Motorola Inc.
    1 Syntek America Inc.
```

A fenti példában csak a gyártókra voltam kíváncsi és a darabszámokra.

Ezekből a példákból még az is kiderülhetett számunkra, hogy a `Group-Object` használata előtt nem kell külön sorba rendezni a csővezetéken érkező objektumokat, hogy a csoportok jól kialakuljanak. Emlékezhetünk a `format-table` cmdlet `-groupby` paraméterére az *1.4.17 Egyszerű formázás* fejezetből, ott azt láthattuk, hogy a `format-table` számára előbb sorba kellett rendezni az objektumokat, hogy jó legyen a csoportosítás. Itt ilyesmire nincs szükség, hiszen a `Group-Object` halmazokat készít, nem csak megjeleníti a csoportokat.

1.10.4 Objektumok átalakítása (Select-Object)

A `Select-Object` cmdlet a kapott objektumok megcsonkítását képes elvégezni, a kimenetként kapott objektumokban már csak a paraméterlistában megadott tulajdonságok fognak szerepelni, a többi tulajdonságot (és a bemenetként kapott objektumreferenciát is) a cmdlet egyszerűen eldobja. Viszont az eredeti objektum tulajdonságait átalakíthatjuk, újabb tulajdonságokat számolhatunk ki, hozhatunk létre, amelyekkel esetleg könnyebben tudunk dolgozni.

? **Feladat:** Alakítsuk át a `Get-Process`-től érkező objektumokat úgy, hogy csak a folyamat nevét, gyártóját és leírását tartalmazzák!

A `Select-Object` cmdletnek egyszerűen azt kell megadnunk, hogy melyik tulajdonságokat szeretnénk megtartani:

```
PS C:\> get-process | select-object name, company, description

Name                      Company                   Description
----                      -------                   -----------
ctfmon                    Microsoft Corporation     CTF Loader
CTHELPER                  Creative Technology Ltd   CtHelper MFC Application
CTSVCCDA                  Creative Technology Ltd   Creative Service for ...
csrss
explorer                  Microsoft Corporation     Windows Intéző
...
```

Mi is történik ebben az esetben? A `Get-Process` Process objektumokat dobál a csőbe, ezeket kapja meg a `Select-Object`, amely a paraméterlista által meghatáro-

zott tulajdonságokat egy újonnan létrehozott egyedi objektum azonos nevű tulajdonságaiba másolja. Ezek az objektumok fognak aztán továbbvándorolni a csövön.

Természetesen nemcsak azt mondhatjuk meg, hogy mely tulajdonságokra van szükségünk, hanem azt is, hogy melyekre nincs: az `-excludeproperty` paraméter után felsorolt tulajdonságok nem fognak szerepelni a kimenő objektumokban (de az összes többi igen).

A kimenő objektumok szerkezetét egy `Get-Member` hívás mutatja meg:

```
PS C:\> get-process | select-object -property name, description, company |
get-member


   TypeName: System.Management.Automation.PSCustomObject

Name         MemberType   Definition
----         ----------   ----------
Equals       Method       System.Boolean Equals(Object obj)
GetHashCode  Method       System.Int32 GetHashCode()
GetType      Method       System.Type GetType()
ToString     Method       System.String ToString()
Company      NoteProperty System.Management.Automation.PSObject Company=...
Description  NoteProperty System.Management.Automation.PSObject Descript...
Name         NoteProperty System.String Name=alg
```

Látszik, hogy ez már nem az eredeti objektum, hanem egy `PSCustomObject`.

A fentieken kívül a `Select-Object` néhány más funkcióval is rendelkezik: nemcsak elemi objektumokat képes átalakítani, hanem gyűjteményeket is átalakít. Például képes a bemenetként kapott tömb elejéről vagy végéről meghatározott számú elemet leválasztani (`-first` és `-last` paraméter), illetve képes egy gyűjtemény elemei közül csak a különbözőeket (vagyis az egyforma elemek közül csak egyet) továbbadni a `-unique` kapcsoló segítségével:

```
PS C:\> PS C:\> 1,1,3,56,3,1,1,1,3 | Select-Object -unique
1
3
56
```

Ilyenkor a gyűjtemény, tömb egyes elemei természetesen megőrzik típusukat.

Nézzünk végül arra is példát, hogy a `Select-Object`-tel újabb, számolt tulajdonságokat is létrehozhatunk. Itt most a jobb értelmezhetőség kedvéért áttördeltem a kódot:

```
dir c:\old | Select-Object name,
@{      n = "Méret";
        e =     {switch ($m = $_.Length)
                        {       {$m -lt 1kb} {"Pici"; break}
                                {$m -lt 3kb} {"Közepes"; break}
                                {$m -lt 100kb} {"Nagy"; break}
                                default {"Óriási"}
                        }
                 }
 }
```

A fenti példában nem vagyok megelégedve a `dir`, azaz a `get-childitem` által a fájloknál megmutatott tulajdonságokkal, nekem kellene egy pici/közepes/nagy/óriási besorolása a fájloknak. Ehhez a `Select-Object`-nek a fájlok „igazi” `name` tulajdonsága mellett egy „képzett” tulajdonságot is megadok. Ennek formája:

```
@{ name = "Tulajdonságnév”; expression =  Tulajdonságérték}
```

A „name” és „expression” paramétereket lehet rövidíteni. Nálam az „expression” rész egy `switch` kifejezés, mellyel besorolom a fájlokat a méretük alapján.

És nézzük a kimenetet:

```
Name                                Méret
----                                -----
alice.txt                           Pici
coffee.txt                          Pici
lettercase.txt                      Pici
numbers.txt                         Pici
presidents.txt                      Pici
readme.txt                          Közepes
skaters.txt                         Nagy
symbols.txt                         Pici
vertical.txt                        Pici
votes.txt                           Nagy
wordlist.txt                        Óriási
```

Még egy gyakori felhasználási területe van a `select-object`-nek. Ennek felvezetésére nézzünk egy egyszerű példát:

```
[1] PS C:\>get-service a* | Group-Object -Property status

Count Name                      Group
----- ----                      -----
    3 Stopped                   {Alerter, AppMgmt, aspnet_state}
    2 Running                   {ALG, AudioSrv}
```

Az [1]-es sorban lekérdezem az „a” betűvel kezdődő szolgáltatásokat, majd csoportosítottam őket a status paraméterük alapján. A kimeneten az egyes csoportokat alkotó szolgáltatások tömbbe rendezve (kapcsos zárójelek között) láthatók.

Ez most a kevés szolgáltatásnál akár jó is lehet, de ha az összes szolgáltatásra futtattam volna, akkor nem nagyon fért volna ki az egyes csoportok listája. A kibontásban segíthet a `select-object` az `expandproperty` paraméterrel:

```
[2] PS C:\>get-service a* | Group-Object -Property status | Select-Object -E
xpandProperty group

Status   Name               DisplayName
------   ----               -----------
Stopped  Alerter            Alerter
Stopped  AppMgmt            Application Management
Stopped  aspnet_state       ASP.NET State Service
Running  ALG                Application Layer Gateway Service
Running  AudioSrv           Windows Audio
```

A [2]-es sor utolsó tagjában a `select-object` kifejti a `group` tulajdonságot, eredményeképpen visszakapjuk a szolgáltatások listáját, immár csoportosítás utáni sorrendben. Gyakorlatilag ugyanahhoz az eredményhez jutottunk, mint az *1.4.17 Egyszerű formázás* fejezet végén a `get | sort | format-table –groupby` kifejezéssorral. Mivel itt is gyűjteményen végeztem a `select-object`-tel a műveletet, az egyes elemek eredeti objektumtípusa megőrződött:

```
[3] PS C:\> get-service a* | Group-Object -Property status | Select-Object -
ExpandProperty group | Get-Member


   TypeName: System.ServiceProcess.ServiceController

Name                        MemberType    Definition
----                        ----------    ----------
Name                        AliasProperty Name = ServiceName
add_Disposed                Method        System.Void add_Disposed(EventHa...
Close                       Method        System.Void Close()
Continue                    Method        System.Void Continue()
...
```

Láthatjuk, hogy a művelet végén `ServiceController` típusú objektumokat kaptunk.

1.10.5 Rendezés (Sort-Object)

? **Feladat:** Listázzuk ki a tíz legtöbb memóriát foglaló folyamatot a memóriafoglalás sorrendjében!

A folyamatok listáját természetesen ismét a `Get-Process` adja. A folyamat által igénybe vett fizikai memória mennyiségét a „`Workingset`” tulajdonság adja meg, ami az alapértelmezett táblázatban is szerepel. A `Sort-Object` tetszőleges tulajdonság értékei szerint tudja sorba rendezni a kimenetet. A listázás alapértelmezés szerint a legki-

sebb értéktől indul, de ha megadjuk a `-desc` paramétert, akkor a legnagyobb értékek kerülnek a lista tetejére.

```
PS C:\> get-process | sort-object -property workingset -desc
```

A lista tehát már rendezett, de nekünk csak az elején szereplő tíz sorra lenne szükségünk. A korábban már használt `Select-Object` cmdlet egyik alfunkciójának segítségével ez is könnyen megoldható, egyszerűen paraméterként kell megadnunk, hogy hány sorra van szükségünk. A megoldás tehát:

```
PS C:\> get-process | sort-object -property workingset -desc | select-object
-first 10

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
   2779      45   197548      45732   439   407,91    460 iexplore
    420       9    41488      34552   159     6,13    612 powershell
    463      18    34580      23060   343   114,03   3952 WINWORD
    693      20    22396      20312   123    39,34    960 explorer
    317      19    45132      20048   125   121,17    624 Mcshield
   1729      82    25332      16368   215    96,22   1436 svchost
    557      16    46872       7720   180    42,11   2896 OUTLOOK
    779      20    67944       7200   198    19,30   2496 iexplore
    340      11    17876       6716   122   143,34   1540 svchostl
    456      13    48196       6452   139     4,84   3604 iexplore
```

Nem mindegy, hogy milyen kultúrkör, nyelv ábécéje szerint rendezzük az objektumokat. Alaphelyzetben az adott gép területi beállításait veszi figyelembe a PowerShell, de ezt a `-culture` paraméterrel megváltoztathatjuk:

```
[24] PS C:\> "zug", "zsuga", "csak", "cuki" | Sort-Object -Culture "hu-hu"
cuki
csak
zug
zsuga
[25] PS C:\> "zug", "zsuga", "csak", "cuki" | Sort-Object -Culture "en-us"
csak
cuki
zsuga
zug
```

A [24]-es sorban magyar nyelv szabályai szerint rendeztem sorba a szavakat, míg a [25]-ös sorban ugyanezeket a szavakat az angol ábécé szerint. Látjuk, hogy jelentős eltérés van a két sorrend között, hiszen a magyar nyelvben a kettős betűk külön sorolódnak be a az ábécébe.

A különböző nyelvekhez tartozó nyelvi kódok jelölését a http://msdn2.microsoft.com/en-us/library/ms970637.aspx oldalon meg lehet nézni.

1.10.6 Még egyszer format-table

Az előzőekben láthattuk, hogy a `select-object` segítségével kiszámoltathatunk új tulajdonságértékeket is. Azonban ilyesmire a `format-table` is alkalmas:

```
[9] PS C:\old>Get-ChildItem | Format-Table name,@{Expression={if($_.psiscont
ainer){"Könyvtár"}else{"Fájl"}};Label="Típus";width=10}

Name                                                                 Típus
----                                                                 -----
alfolder                                                             Könyvtár
alice.txt                                                            Fájl
coffee.txt                                                           Fájl
dir.xml                                                              Fájl
```

Azaz itt is megadható egy hashtábla kifejezés, amit a `format-table` kiszámol, illetve beépít a táblázatba:

Mező	Jelentése
Expression	A kiszámolandó kifejezés
Label	A táblázat oszlopának címkéje
Width	Az oszlop szélessége

Ez annyiban különbözik a `select-object`-es átalakítástól, hogy itt az eredeti objektumhoz nem nyúlunk, az minden eredeti tulajdonságát és a típusát megőrzi, csak a megjelenítést változtatjuk meg.

1.10.7 Gyűjtemények összehasonlítása (Compare-Object)

A `Compare-Object` cmdlet segítségével két tetszőleges gyűjteményt hasonlíthatunk össze, kimenetül a gyűjtemények közötti különbséget leíró objektumokat kapunk. Az alábbi példában egy változóba mentjük a gépen futó folyamatok listáját. Ezután leállítunk, illetve elindítunk néhány folyamatot, majd ezt az új állapotot egy másik változóba írjuk. A `Compare-Object`-nek odaadjuk a két változót, ő pedig kilistázza különbségeket:

```
PS C:\> $a = Get-Process
PS C:\> $b = Get-Process
PS C:\> Compare-Object $a $b

InputObject                          SideIndicator
-----------                          -------------
System.Diagnostics.Process (cmd)     =>
System.Diagnostics.Process (iexplore) =>
System.Diagnostics.Process (notepad)  =>
System.Diagnostics.Process (OUTLOOK)  <=
```

Mire használható ez? Szeretnénk például megtudni, hogy az internetről gyűjtött csodaprogram telepítője pontosan mit garázdálkodik a gépünkön? Semmi gond, készítsünk pillanatfelvételt az érzékeny területekről (futó folyamatok, fájlrendszer, registry) a telepítés előtt, majd hasonlítsuk össze a telepítőprogram lefutása utáni állapottal. Lesz nagy meglepetés! Nem kell elaprózni, bátran lekérhetjük például a teljes c: meghajtó állapotát, a gép majd beleizzad kicsit az összehasonlításba, de így mindenre fény derül:

```
PS C:\> $a = Get-ChildItem c: -recurse
PS C:\> $b = Get-ChildItem c: -recurse
PS C:\> Compare-Object $a $b
```

Vigyázzunk azonban a `compare-object`-tel! Ha túl sok a különbség a két gyűjtemény között, akkor nem biztos, hogy minden különbséget felfedez. Nézzünk erre egy példát:

```
[29] PS C:\> Compare-Object 1,2,3,4,5,6,7,8,9,10 11,12,13,14,15,16,1

                           InputObject SideIndicator
                           ----------- -------------
                                    11 =>
                                     1 <=
                                    12 =>
                                     2 <=
                                     3 <=
                                     4 <=
                                     5 <=
                                    13 =>
                                    14 =>
                                    15 =>
                                    16 =>
                                     1 =>
                                     6 <=
                                     7 <=
                                     8 <=
                                     9 <=
                                    10 <=
```

A fenti példában az egyik gyűjteményem 1-től 10-ig a számok, a másik gyűjteményem 11-től 16-ig, plusz az 1-es. Azaz az 1 az nem különbség a két gyűjtemény között, mégis az eredményben, ami ugye az eltéréseket adja meg, az 1-es is szerepel, ráadásul kétszer is. Ennek az az oka, hogy a `compare-object` nem minden elemhez néz meg minden elemet, hanem alaphelyzetben csak öt elem távolságra. Azaz az első halmaz 1-esét összehasonlítja a második tömbbeli 11-el, 12-vel stb., 16-tal, de az 1-gyel már nem. Rábírhatjuk a `compare-object`-et, hogy messzebbre tekintsen a `-syncwindow` paraméterrel:

```
[31] PS C:\> Compare-Object 1,2,3,4,5,6,7,8,9,10 11,12,13,14,15,16,1 -SyncWi
ndow 6

                           InputObject SideIndicator
                           ----------- -------------
                                     2 <=
```

```
                                    3 <=
                                    4 <=
                                   11 =>
                                   12 =>
                                   13 =>
                                   14 =>
                                   15 =>
                                   16 =>
                                    5 <=
                                    6 <=
                                    7 <=
                                    8 <=
                                    9 <=
                                   10 <=
```

Ebben a példában az alaphelyzet szerinti 5 távolságot megtoldottam még eggyel, így már rálelt az összehasonlítás a két 1-esre.

1.10.8 Különböző objektumok (Get-Unique)

Sokszor előfordulhat, hogy egyedi objektumokat keresünk, azaz a duplikációkra nem vagyunk kíváncsiak. Például a futó processzek esetében ugyanazokat nem akarom többször látni. Alaphelyzetben ezt a listát kapom:

```
[45] PS C:\>Get-Process | sort

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
...
      0       0        0         28     0               0 Idle
    758      20    28864      24160   162    29,53   2504 iexplore
    807      24    54592      28636   207    93,50   1296 iexplore
     97       3      740       2864    23     0,09   2788 igfxpers
     80       3     1216       3572    34     0,16   2532 igfxtray
    473      10     4284       1444    43     7,66   1016 lsass
...
```

Nézzük csak az egyedieket a `get-unique` segítségével:

```
[47] PS C:\>Get-Process | get-unique | sort

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
...
      0       0        0         28     0               0 Idle
    807      24    54592      28636   207    93,50   1296 iexplore
     97       3      740       2864    23     0,09   2788 igfxpers
     80       3     1216       3572    34     0,16   2532 igfxtray
    481      11     4316       1880    43     7,66   1016 lsass
...
```

Látszik, hogy az `iexplorer` csak egyszer szerepel a második listámban.

Vajon mit lehet tenni akkor, ha pont a duplikáltakra vagyok kíváncsi? Sajnos nincs „get-duplicate” cmdlet, de ilyen jellegű működést mi magunk is elő tudunk idézni a get-unique és az előbb látott compare-object ötvözetével:

```
[55] PS C:\>$elemek = "a","b","c","d","e","a","b","a"
[56] PS C:\>Compare-Object ($elemek) ($elemek| sort | Get-Unique)

InputObject                           SideIndicator
-----------                           -------------
a                                     <=
b                                     <=
a                                     <=
```

Nézzük meg ugyanezt a processzekre:

```
[57] PS C:\>Compare-Object (Get-Process ) (Get-Process| Get-Unique)

InputObject                           SideIndicator
-----------                           -------------
System.Diagnostics.Process (System)   =>
System.Diagnostics.Process (iexplore) <=
System.Diagnostics.Process (UdaterUI) =>
System.Diagnostics.Process (svchost)  <=
System.Diagnostics.Process (usnsvc)   =>
System.Diagnostics.Process (svchost)  <=
System.Diagnostics.Process (VsTskMgr) =>
System.Diagnostics.Process (svchost)  <=
System.Diagnostics.Process (vVX3000)  =>
System.Diagnostics.Process (svchost)  <=
System.Diagnostics.Process (svchost)  <=
System.Diagnostics.Process (System)   <=
System.Diagnostics.Process (UdaterUI) <=
System.Diagnostics.Process (usnsvc)   <=
System.Diagnostics.Process (VsTskMgr) <=
System.Diagnostics.Process (vVX3000)  <=
```

Hoppá! Ez nem túl barátságos. Az még érthető, hogy az iexplore az benne van a listában, hiszen az két példányban fut a gépemen. De vajon a System processz miért nincs benne a „unique” listában? Ez valójában a compare-object előbb már látott sara, mivel – ahogy láttuk – nem minden elemet minden elemmel hasonlít össze, hanem az egymás mellé tett listákban csak +/- 5 elem környezetben vizsgálja az egyezőséget (hogy ne legyen túl lassú az egész cmdlet). De persze alaposabbá tehetjük a keresését megint csak a -syncWindow paraméter megnövelésével (miután a gépemen 100-nál kevesebb processz futott, ezért a 100-as érték bőven elég):

```
[59] PS C:\>Compare-Object (Get-Process ) (Get-Process| Get-Unique) -syncwin
dow 100

InputObject                           SideIndicator
-----------                           -------------
System.Diagnostics.Process (iexplore) <=
System.Diagnostics.Process (svchost)  <=
```

```
System.Diagnostics.Process (svchost)  <=
System.Diagnostics.Process (svchost)  <=
System.Diagnostics.Process (svchost)  <=
System.Diagnostics.Process (svchost)  <=
```

Így már jól működik a „`get-duplicate`” kifejezésem!

1.10.9 Számlálás (Measure-Object)

Tetszőleges objektumcsoport elemeivel kapcsolatos összegzést átlagolást, stb. végezhetünk el a `Measure-Object` cmdlet segítségével. A cmdlet két különböző üzemmódban használható, és ennek megfelelően két különböző típusú objektumot adhat kimenetül. Szöveges üzemmódot a `-line`, `-word`, `-character` paraméterek valamelyikével kérhetünk, ekkor a bemenetként érkező objektumok szöveges reprezentációjából származó értékek kerülnek a kimenetbe, vagyis a cmdlet megszámolja a szövegben található sorokat, szavakat és karaktereket.

? **Feladat:** Számoljuk meg a korábban készített dir.txt sorait, szavait és karaktereit!

A megoldás egyszerűen a következő (az `-ignorewhitespace` paraméter használata esetén a szóközök és tabulátorok nem számítanak bele a karakterek számába):

```
PS C:\> get-content c:\dir.txt | measure-object -ignorewhitespace -line -
word -char | format-table -autosize

Lines Words Characters Property
----- ----- ---------- --------
   13    55        403
```

Ha a `-property` paraméter után megadjuk a bemenő objektumok valamelyik tulajdonságának nevét, akkor a kimeneten a megadott tulajdonság összege, átlaga, maximális és minimális értéke fog megjelenni.

? **Feladat:** Összegezzük a rendszerfolyamatok fizikai memóriafoglalását!

A megoldás egyszerűen a következő:

```
PS C:\> get-process | measure-object -property workingset -sum -average -m
ax -min


Count    : 42
Average  : 4400664,38095238
Sum      : 184827904
Maximum  : 80818176
Minimum  : 16384
```

```
Property : WorkingSet
```

A `count` sorban lévő szám ebben az esetben azt jelenti, hogy 42 darab „workingset” tulajdonságot átlagolt, illetve összegzett a cmdlet, vagyis ennyi Process objektumot kapott bemenetként.

Ha csak a fenti statisztika egyik számadatával szeretnénk dolgozni, akkor a szokásos tulajdonsághivatkozással ezt is megtehetjük. Például szeretném a fenti memóriafoglalást megabájtokban kijelezni:

```
[7] PS I:\>(get-process | measure-object -property workingset -sum).sum / 1m
b
968,5390625
```

1.10.10 Nyomtatás (Out-Printer)

Az `Out-Printer` cmdlet a bemenetként kapott adatokat az alapértelmezett, vagy a paraméterként megadott nyomtatóra írja.

```
PS C:\> Get-Process | Out-Printer \\server\HPLJ5si
```

1.10.11 Kiírás fájlba (Out-File, Export-)

Alapértelmezés szerint a PowerShell parancsok kimenete (vagyis a kimenetként kapott objektumok szöveges reprezentációja) a konzolablakban jelenik meg. A szöveges kimenet fájlba irányítását az `Out-File` és a `Set-Content` cmdlet segítségével végezhetjük el, amelyek paramétereként a létrehozandó fájl nevét kell megadnunk. Az `Export-Csv` és az `Export-CliXML` cmdletek nevükhöz méltóan csv, illetve xml formátumban írják fájlba a bemenetül kapott objektumok adatait.

? **Feladat:** Készítsünk szövegfájlt, csv és xml fájlt a c: meghajtó mappalistájából!

A megoldás mindhárom esetben nagyon egyszerű, viszont az eredmény a formátumon túl, a további felhasználhatóság szempontjából is lényegesen különbözik egymástól.

```
PS C:\> Get-ChildItem | Out-File c:\dir.txt
```

Ebben az esetben a fájlba csak az kerül, amit az eredeti parancs a képernyőre írt volna. Ha a fájlt visszaolvassuk (`Get-Content`), az eredeti kimenet sorainak megfelelő karakterláncokat kapunk.

A kimeneti fájlban alaphelyzetben 80 karakter szélességűre tördelt sorokat kapunk, ha ennél szélesebb sorokat szeretnénk, akkor használjuk a `-width` paramétert.

```
[27] PS C:\>Get-ChildItem c:\old | ft -property * -auto | Out-File c:\old\di
r2.txt -Width 800
```

A fenti példában a fájllistámat táblázatos formában akarom kirakni egy szöveges állományba, de az összes tulajdonsággal. Ez jó széles táblázatot eredményez, így az `out-file`-nál egy brutálisan széles sorméretet adok meg lehetséges értéknek. De hogy feleslegesen ne terpeszkedjen szét a táblázatom, ezért használtam a `format-table` cmdletnél az `-auto` kapcsolót, mert ez olyan sűrűn teszi az oszlopokat, amilyen sűrűn adatvesztés nélkül lehet. Így tehát kaptam egy optimális szélességű szöveget, amely ténylegesen csak 399 karakter széles lett, azaz nem kellett kihasználni a 800 karakteres maximumot.

Nézzük meg, hogy még milyen formátumokba tudjuk kitenni fájlba az outputot. A következő a csv formátum, ami táblázatos forma:

```
PS C:\> Get-ChildItem | Export-Csv c:\dir.csv
```

Ha belenézünk az eredményül kapott csv fájlba (notepad, vagy excel is jó), akkor örömmel láthatjuk, hogy ebben az esetben az objektumok minden tulajdonsága bekerült a kimenetbe, sőt a fejlécben még a típus megnevezése is megtalálható. Van egy `Import-Csv` cmdletünk is, ami a fájlt visszaolvasva újra létrehozza az eredeti objektumot, jobban mondva egy ahhoz hasonló objektumot. Próbáljuk is ki:

```
PS C:\> Import-Csv c:\dir.csv


PSPath            : Microsoft.PowerShell.Core\FileSystem::C:\Config.Msi
PSParentPath      : Microsoft.PowerShell.Core\FileSystem::C:\
PSChildName       : Config.Msi
PSDrive           : C
PSProvider        : Microsoft.PowerShell.Core\FileSystem
PSIsContainer     : True
Mode              : d----
```

Az eredmény érdekesen néz ki, viszont sajnos nem hasonlít túlságosan az eredeti mappalistára. Egy `Get-Member` hívás segítségével megtudhatjuk, hogy mi is történt az adatainkkal:

```
PS C:\> Import-Csv c:\dir.csv | get-member


   TypeName: CSV:System.IO.DirectoryInfo

Name              MemberType   Definition
----              ----------   ----------
Equals            Method       System.Boolean Equals(Object obj)
GetHashCode       Method       System.Int32 GetHashCode()
GetType           Method       System.Type GetType()
ToString          Method       System.String ToString()
Attributes        NoteProperty System.String Attributes=Directory
```

```
CreationTime      NoteProperty System.String CreationTime=2006.11.15. 8...
CreationTimeUtc   NoteProperty System.String CreationTimeUtc=2006.11.15...
Exists            NoteProperty System.String Exists=True
Extension         NoteProperty System.String Extension=
FullName          NoteProperty System.String FullName=C:\752e9949d08195...
LastAccessTime    NoteProperty System.String LastAccessTime=2007.07.06....
...
```

Látható, hogy nem az igazi `DirectoryInfo` típust kaptuk vissza (ráadásul a `FileInfo` típus teljesen eltűnt), hanem csak valamiféle pótlékot, az eredeti objektum leegyszerűsített változatát, amelynek nincsenek metódusai (csak amelyek az `Object` osztályból öröklődnek, ezekkel minden objektum rendelkezik), tulajdonságai pedig elvesztették eredeti típusukat. Minden tulajdonságérték megvan ugyan, de csak karakterláncként, az eredeti típusinformáció tárolására a csv formátum nem képes.

Ha kevesebb információvesztéssel szeretnénk tárolni és visszaolvasni objektumainkat, akkor az xml formátumot kell választanunk.

```
PS C:\> Get-ChildItem | Export-CliXML c:\dir.xml
```

Az xml fájl mérete közel tízszerese a csv-nek, de ebből pontosan az eredeti adathalmaz tulajdonságai olvashatók vissza, mindenféle veszteség, vagy torzulás nélkül. Ilyen módon tehát tetszőleges .NET objektum, gyűjtemény, akármi teljes tartalmát lemezre menthetjük, és később, a fájlt egyetlen paranccsal visszaolvasva helyreállíthatjuk az eredeti objektumot[13]. Nem rossz!

```
PS C:\> Import-CliXML c:\dir.xml


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
            2007.06.10.     12:41            Config.Msi
            2007.06.13.     21:30            Documents and Settings
            2003.10.25.     18:59            Inetpub
            2007.06.04.     20:36            Program Files
```

Azért egy kis csalás itt is van. Ha megnézzük `get-member`-rel, hogy mit is kaptunk, akkor kiderül, hogy itt sem ugyanolyan objektumtípust kaptunk vissza, például a metódusokat ez is elveszítette:

```
[13] PS C:\old>import-clixml C:\dir.xml | gm


   TypeName: Deserialized.System.IO.DirectoryInfo
```

[13] A Mode oszlop azért hiányzik, mert az eredeti .NET objektumnak sincs ilyen tulajdonsága, ezt csak a PowerShell hazudja oda.

```
Name              MemberType   Definition
----              ----------   ----------
PSChildName       NoteProperty System.String PSChildName=alfolder
PSDrive           NoteProperty System.Management.Automation.PSObject PSD...
PSIsContainer     NoteProperty System.Boolean PSIsContainer=True
PSParentPath      NoteProperty System.String PSParentPath=Microsoft.Powe...
PSPath            NoteProperty System.String PSPath=Microsoft.PowerShell...
PSProvider        NoteProperty System.Management.Automation.PSObject PSP...
Attributes        Property     System.String {get;set;}
CreationTime      Property     System.DateTime {get;set;}
CreationTimeUtc   Property     System.DateTime {get;set;}
Exists            Property     System.Boolean {get;set;}
Extension         Property     System.String {get;set;}
FullName          Property     System.String {get;set;}
LastAccessTime    Property     System.DateTime {get;set;}
LastAccessTimeUtc Property     System.DateTime {get;set;}
LastWriteTime     Property     System.DateTime {get;set;}
LastWriteTimeUtc  Property     System.DateTime {get;set;}
Name              Property     System.String {get;set;}
Parent            Property     System.String {get;set;}
Root              Property     System.String {get;set;}


   TypeName: Deserialized.System.IO.FileInfo

Name              MemberType   Definition
----              ----------   ----------
PSChildName       NoteProperty System.String PSChildName=alice.txt
PSDrive           NoteProperty System.Management.Automation.PSObject PSD...
PSIsContainer     NoteProperty System.Boolean PSIsContainer=False
PSParentPath      NoteProperty System.String PSParentPath=Microsoft.Powe...
PSPath            NoteProperty System.String PSPath=Microsoft.PowerShell...
PSProvider        NoteProperty System.Management.Automation.PSObject PSP...
Attributes        Property     System.String {get;set;}
CreationTime      Property     System.DateTime {get;set;}
CreationTimeUtc   Property     System.DateTime {get;set;}
Directory         Property     System.String {get;set;}
DirectoryName     Property     System.String {get;set;}
Exists            Property     System.Boolean {get;set;}
Extension         Property     System.String {get;set;}
FullName          Property     System.String {get;set;}
IsReadOnly        Property     System.Boolean {get;set;}
LastAccessTime    Property     System.DateTime {get;set;}
LastAccessTimeUtc Property     System.DateTime {get;set;}
LastWriteTime     Property     System.DateTime {get;set;}
LastWriteTimeUtc  Property     System.DateTime {get;set;}
Length            Property     System.Int64 {get;set;}
Name              Property     System.String {get;set;}
```

Azaz az `Export-CliXML` és `Import-CliXML` párossal az objektumot tulajdonságai megőrződnek, ugyanúgy felhasználhatjuk ezeket műveletek végzésére, mint az eredeti objektumok tulajdonságait, viszont a metódusok elvesztek, ha azokra is szükségünk lenne, akkor sajnos azon melegében, még exportálás előtt meg kell hívnunk ezeket.

1.10.12 Átalakítás szöveggé (Out-String)

Láthattuk, hogy jó dolog az objektumorientált megközelítés, az esetek döntő többségében jelentősen egyszerűbbé vált tőle az élet a hagyományos shellekkel összehasonlítva. Bizonyos esetekben mégis szükség lehet arra, hogy az objektumokból álló kimenetet szövegként dolgozzuk fel, és például egy szöveges keresés alapján válogassuk ki belőle azokat a sorokat, amelyekre szükségünk van. Nem szerencsés például az objektumos kimenet, ha az, amit keresünk, több tulajdonság értékében is előfordulhat, és nekünk azok a sorok kellenek, amelyekben akárhol is, de előfordul (vagy éppen hiányzik) a keresett érték.

? **Feladat:** Listázzuk ki egy `Get-Process` alapértelmezett kimenetéből azokat a sorokat, amelyekben (bármelyik oszlopban!) előfordul a „44” karakterlánc!

Az objektumok szöveges változatának előállítására az `Out-String` cmdlet képes. Bemenetére tetszőleges objektumokat küldhetünk (például formázott táblázatot, listát, stb. is), a kimenetén egy hosszú karakterlánc fog megjelenni, amely mindazt tartalmazza, amit a bemenet a képernyőre írt volna. Próbáljuk ki a következőt:

```
PS C:\> get-process | out-string


Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     66       3      864       2808    30     0,08    416 ctfmon
    173       5     3564       5340    33     0,19   1100 CTHELPER
     29       1      380        756    15     0,02   1436 CTSVCCDA
    473       6     2144       2420    28    11,45    612 csrss
    689      18    25440      18048   107    50,84   3804 explorer
```

Látszólag semmi különbség nincs az önálló `Get-Process`-hez képest, de ha a kimenetre kérünk egy `Get-Member`-t, akkor látható, hogy az már nem `System.Diagnostics.Process` objektumokból áll, hanem egyszerű karakterlánccá alakult.

Már csak a szöveges keresés van hátra, amelyet a `Select-String` cmdlet segítségével fogunk elvégezni (ezzel részletesen a *2.3.3 Szövegfájlok feldolgozása (Get-Content, Select-String)* fejezetben lesz szó):

```
PS C:\> get-process | out-string | select-string -pattern "44"
```

Valami még nem egészen kerek, mivel ismét csak a teljes listát kaptuk vissza. Mi lehet a baj? Az `Out-String` cmdlet alapértelmezés szerint **egyetlen** karakterláncot ad vissza, ami a teljes listát tartalmazza a sortörésektől függetlenül. A teljes listában persze ott van a keresett minta, így az egyetlen karakterlánc közül azt az egyet kiírtuk a képernyőre.

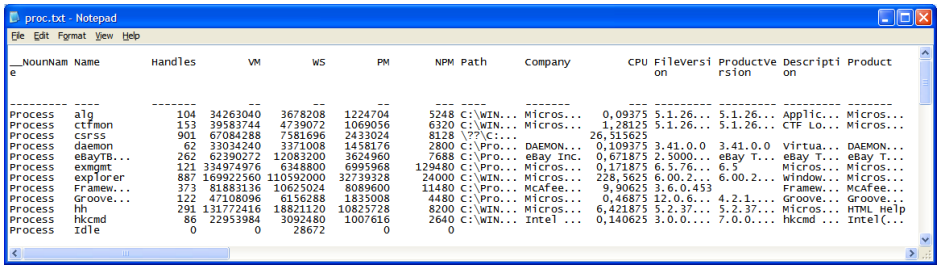
Soronként tördelt kimenetet (karakterláncokból álló tömböt) a `-stream` paraméter használatával kérhetünk, a helyes megoldás tehát:

```
PS C:\> get-process | out-string -stream | select-string -pattern "44"

     37       2     2016          40    30     0,09   3144 cmd
     63       3     1068         372    30     0,09    844 daemon
    779      20    67944        7200   198    19,30   2496 iexplore
     21       1      168         144     4     0,08    920 smss
    162       7     4960        2652    60     0,44   1968 spoolsv
    142       4     2396         372    44     0,50    640 VsTskMgr
```

Az `out-string`-nek van még egy praktikus paramétere, ez pedig a `-width`. Ez akkor jöhet jól, ha nagyon széles táblázatot akarunk megjeleníteni, és az alaphelyzet szerinti 80 karakteres szélesség túl kevésnek tűnik. Mi van akkor, ha a `get-process`-szel a folyamatok összes tulajdonságát meg akarom jeleníteni. Ez 80 karakterben gyakorlatilag reménytelen, nézzük meg, hogy 600 karakterbe hogyan fér el:

```
[19] PS C:\>get-process | ft -Property * | out-string -Width 600 -Stream > c
:\old\proc.txt
```

Ezek után nézzük notepad-del a fájlt:



44. ábra 600 karakter széles szöveg Notepadben

A sortörést kikapcsoltam, és látszik, hogy viszonylag jól elférünk már. Az alsó gördítősáv méretéből látható, hogy elég jócskán lehet vízszintesen görgetni ezt a táblázatot. Ezzel gyakorlatilag ugyanazt az eredményt értem el, mint az `out-file` cmdlet használatával, megfelelő `-width` paraméterrel.

1.10.13 Kimenet törlése (Out-Null)

Bizonyos feldolgozások során (elsősorban automatikusan futtatott szkriptekben) zavaró lehet a konzolon megjelenő kimenet. Ebben az esetben tehet jó szolgálatot az `Out-Null` cmdlet, amelyet a sor végére biggyesztve mindenféle esetleg megjelenő üzenettől szövegtől (kivéve a hibaüzeneteket) megszabadulhatunk.

1.11 Összefoglaló: PowerShell programozási stílus

Ez elméleti rész lezárásaként összefoglalnám azokat a főbb jellemzőket, amelyek a PowerShellben történő programozás jellegzetességeinek érzek:

- Csövezés minden mennyiségben!
  Egy-egy művelet eredményét legtöbb esetben felesleges változókba rakni, érdemes azonnal továbbküldeni további feldolgozásra a csővezetéken keresztül egy újabb parancs számára. Ezzel nem csak egyszerűbb, tömörebb lesz a programunk, hanem egy csomó átmeneti adattárolás memória-felhasználását spóroljuk meg.

- Gyűjtemény vagy objektum?
  Mint ahogy a láttuk például a `get-member` cmdlet működésénél, a PowerShell néha túl „okosan” próbál gyűjteményeket kezelni, azaz nem mint objektumokat használja, hanem kifejti az egyes elemeit. Míg ha egy elemet adunk neki, akkor meg azt az adott objektumként kezeli. Ez gyakran félrevezető, főleg amikor egy objektum tulajdonságait próbáljuk feltérképezni, és azt hisszük, hogy már egy indexelhető tömbnél tartunk, és akkor derül ki számunkra, hogy nem, még mélyebbre kell ásnunk, mert amit látunk az még mindig egy egyetlen elemet tartalmazó tömbobjektum.

- Hagyományos „programolós” ciklus helyett bármi más!
  Akinek valamilyen klasszikus programnyelvben van gyakorlata, az PowerShellben is hajlamos eleinte minden többelemű adat feldolgozásakor `FOR` ciklust írni. De ha nincs szükségünk az elemek ilyen jellegű indexelésére, nyugodtan használjunk `FOREACH` ciklust. Ha csővezetéken érkeznek az adatok, akkor meg használjunk `ForEach-Object` cmdletet. Sőt, mint ahogy az „Exchange 12 Rocks” példámban szerepelt (*1.5.7 Típuskonverzió* fejezet), a típuskonverzió is kiválthat sok esetben ciklust, illetve még a `SWITCH` kifejezés is használható cikluskén t.

- Írjunk függvényeket, szkripteket, de ha nem akarjuk újra felhasználni ezeket, akkor készítsünk anonim függvényt, azaz olyan szkriptblokkot, mint amit a *1.8.8.1 Anonim függvények* fejezetben mutattam. Ez azért jó, mert itt rögtön használhatjuk a kifejezésünkben a `$_` változót, azaz kezelhetjük a csőelemeket.

- Minden objektum!
  Soha ne feledkezzünk meg arról, hogy a PowerShellben minden kimenet objektum. Ugyan a konzolon szövegeket, karaktereket látunk, de ezek az esetek túlnyomó többségében nem egyszerű szövegek, hanem mindig összetett objektumok, melyeknek csak néhány tulajdonságát látjuk a képernyőn szövegként. Ráadásul ezek a tulajdonságok is általában szintén objektumok. Ne legyünk restek ezeknek a mélyére ásni. Ebben segít bennünket a `get-member` cmdlet, a különböző szkript szerkesztők

(mint például a PowerGUI Script Editora) és a Reflector segédprogram, valamint az MSDN weboldal. Nézzünk utána a tulajdonságoknak, metódusoknak, konstruktoroknak, statikus metódusoknak, nehogy leprogramozzunk valami olyasmit, ami már készen megtalálható az objektum jellemzői között vagy a .NET keretrendszer valamelyik osztályában.

- Szabjuk testre a PowerShellt, de ez ne menjen a kompatibilitás kárára
  Láthattuk, hogy a PowerShell jelenleg még csak 1.0-ás verziójú és van néhány hiányossága, de nagyon egyszerűen lehet bővíteni a környezetet. Újabb függvényekkel új funkciókat valósíthatunk meg. Új álnevekkel kevesebbet kell gépelnünk. A típusok kiegészítésével újabb tulajdonságokat és metódusokat készíthetünk objektumainkhoz. Vigyázzunk azonban, hogy ha valamilyen bővítést készítünk, akkor az vajon hogyan működik egy másik gépen? Ott is megvannak-e azok a kiegészítések, amik lehetővé teszik a bővítményeink futtatását? Ezeket se felejtsük el magunkkal vinni, készítsünk függvénytárakat a gyakran használt függvényeinkből. Ezzel folytatom majd a könyv második részét.
  Viszont a beépített cmdleteket, álneveket lehetőleg ne definiáljuk újra, mert ez a programjaink, parancssoraink értelmezhetőségének kárára megy.

2. Gyakorlat

Az elméleti részben áttekintettük a PowerShell telepítését, nyelvi elemeit. A könyv ezen részében gyakorlatiasabb példákkal folytatjuk. Természetesen itt sem fogunk több oldalas példaprogramokat írni, hiszen ez a könyv nem fejlesztőknek, hanem gyakorló rendszergazdáknak szól, akik gyorsan, kevés programozással szeretnének eredményre jutni.

2.1 PowerShell környezet

A gyakorlati rész első fejezete a PowerShell környezet komfortosabbá tételéről szól, milyen lehetőségek állnak rendelkezésre, hogy a PowerShell ablak ne ugyanolyan tudással induljon, mint korábban, hanem építse be az általunk korábban elkészített függvényeket, szkripteket, illetve töltse be azokat a külső bővítményeket, amelyekkel szintén kiterjeszthetjük a képességeit.

2.1.1 Szkriptkönyvtárak, futtatási információk ($myinvocation)

Munkánk során valószínű jó néhány hasznos függvényt készítünk, amelyeket rendszeresen használni szeretnénk. Ezekhez úgy férünk hozzá legegyszerűbben, ha ezeket a függvényeket szkriptfájlokban elmentjük egy könyvtárba, majd a sok kis szkriptfájlunkat egy „beemelő”, „include” jellegű központi szkripttel lefuttatjuk.

Ennek modellezésére készítettem egy „scripts” könyvtárat, amelyben három szkriptem három függvényt definiál. Ezen kívül van egy `include.ps1` szkriptem, ami csak annyit csinál, hogy a saját könyvtárában levő másik három szkriptet meghívja „dotsourcing” jelleggel, azaz úgy, hogy a szkriptek által definiált függvények bárhonnan elérhetők, meghívhatók legyenek.

```
[17] PS C:\powershell2\egyik> Get-ChildItem C:\powershell2\scripts


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\powershell2\scripts


Mode           LastWriteTime       Length Name
----           -------------       ------ ----
-a---   2008.04.19.    12:31           42 fv1.ps1
-a---   2008.04.19.    12:31           45 fv2.ps1
-a---   2008.04.19.    12:31           45 fv3.ps1
-a---   2008.04.19.    12:32           40 include.ps1
```

```
[18] PS C:\powershell2\egyik> get-content C:\powershell2\scripts\fv1.ps1
function fv1
{
        "Első függvény"
}
[19] PS C:\powershell2\egyik> get-content C:\powershell2\scripts\include.ps1

. .\fv1.ps1
. .\fv2.ps1
. .\fv3.ps1
```

Ez egyes függvények nagyon egyszerűek, csak annyit írnak ki, hogy hányadik függvényről van szó. Ez így külön-külön nagyon szépnek és logikusnak tűnik, próbáljuk meg futtatni az `include.ps1` szkriptünket az „egyik” nevű könyvtárból:

```
[22] PS C:\powershell2\egyik> C:\powershell2\scripts\include.ps1
The term '.\fv1.ps1' is not recognized as a cmdlet, function, operable prog
ram, or script file. Verify the term and try again.
At C:\powershell2\scripts\include.ps1:1 char:2
+ .  <<<< .\fv1.ps1
The term '.\fv2.ps1' is not recognized as a cmdlet, function, operable prog
ram, or script file. Verify the term and try again.
At C:\powershell2\scripts\include.ps1:2 char:2
+ .  <<<< .\fv2.ps1
The term '.\fv3.ps1' is not recognized as a cmdlet, function, operable prog
ram, or script file. Verify the term and try again.
At C:\powershell2\scripts\include.ps1:3 char:2
+ .  <<<< .\fv3.ps1
```

Valami nem jó! Nyomozzunk utána, cseréljük le az `include.ps1` belsejét egy olyan vizsgálatra, amely megmutatja, hogy mit érez a szkript aktuális könyvtárnak. Amíg nem találom meg a hibát, a függvényeket definiáló szkriptek hívását kikommenteztem:

```
#. .\fv1.ps1
#. .\fv2.ps1
#. .\fv3.ps1
Get-Location
```

Futtatva ezt kapjuk:

```
[23] PS C:\powershell2\egyik> C:\powershell2\scripts\include.ps1

Path
----
C:\powershell2\egyik
```

Kiderült a hiba oka, annak ellenére, hogy az `include.ps1` a `scripts` könyvtárban fut, számára is az aktuális könyvtár az „`egyik`”. Hogyan lehetne azt megoldani, hogy az `include.ps1` számára a saját könyvtára legyen az aktuális?

Szerencsére van egy `$MyInvocation` nevű automatikus változó, amely a szkriptek számára a futtatásukkal kapcsolatos információkat árulja el. Nézzük is ezt meg, módosítottak az `include.ps1` szkriptemet:

```
#. .\fv1.ps1
#. .\fv2.ps1
#. .\fv3.ps1
$MyInvocation
```

Ezt futtatva kapjuk a következőket:

```
[30] PS C:\powershell2\egyik> C:\powershell2\scripts\include.ps1


MyCommand        : include.ps1
ScriptLineNumber : 1
OffsetInLine     : -2147483648
ScriptName       :
Line             : C:\powershell2\scripts\include.ps1
PositionMessage  :
                   At line:1 char:34
                   + C:\powershell2\scripts\include.ps1 <<<<
InvocationName   : C:\powershell2\scripts\include.ps1
PipelineLength   : 1
PipelinePosition : 1
```

Ha átlépünk a szülőkönyvtárba, és onnan hívjuk meg a szkriptet a következőképpen alakul a `$MyInvocation` értéke:

```
[31] PS C:\powershell2\egyik> cd ..
C:\powershell2
[32] PS C:\powershell2> .\scripts\include.ps1


MyCommand        : include.ps1
ScriptLineNumber : 1
OffsetInLine     : -2147483648
ScriptName       :
Line             : .\scripts\include.ps1
PositionMessage  :
                   At line:1 char:21
                   + .\scripts\include.ps1 <<<<
InvocationName   : .\scripts\include.ps1
PipelineLength   : 1
PipelinePosition : 1
```

Ebből az látszik, hogy se a `Line`, se az `InvocationName` tulajdonság nem a szkript tényleges elérési útját tartalmazza, hanem azt, ahonnan meghívtuk. Így ezekkel a tulajdonságokkal nem tudunk dolgozni, mert ezek sem adnak iránymutatást arra vonatkozólag, hogy hol van ténylegesen az `include.ps1`, és vele együtt a függvényeimet tartalmazó szkriptek.

Nézzük, hogy vajon a `MyCommand`-nak vannak-e tulajdonságai:

```
#. .\fv1.ps1
#. .\fv2.ps1
#. .\fv3.ps1
$MyInvocation.MyCommand
```

A futtatás eredménye:

```
[34] PS C:\powershell2> .\scripts\include.ps1 | fl


Path        : C:\powershell2\scripts\include.ps1
Definition  : C:\powershell2\scripts\include.ps1
Name        : include.ps1
CommandType : ExternalScript
```

Itt már van egy sokat sejtető `Path` tulajdonság, ami egy teljes elérési út, így remény van rá, hogy ebből kiindulva már jól el fogjuk tudni érni a függvényeket tartalmazó szkripteket.

Létezik egy `split-path` cmdlet, amellyel le tudjuk választani egy elérési útról a könyvtár-hierarchia részt, ennek segítségével már el tudjuk készíteni az univerzális, bárhonnan működő `include.ps1` szkriptünket:

```
Push-Location
Set-Location (split-path $MyInvocation.MyCommand.Path)
. .\fv1.ps1
. .\fv2.ps1
. .\fv3.ps1
Pop-Location
```

És a futtatásának eredménye:

```
[41] PS C:\powershell2> .\scripts\include.ps1
[42] PS C:\powershell2> fv1
The term 'fv1' is not recognized as a cmdlet, function, operable program, o
r script file. Verify the term and try again.
At line:1 char:3
+ fv1 <<<<
```

Na, most mi a hiba? Hát az, hogy bár jól lefutott az `include.ps1`, de a betöltött függvényszkriptek csak az ő szintjére lettek „dotsource”-olva. Ahhoz, hogy a globális scope-ból is elérhessük ezeket a függvényeket, magát az `include.ps1`-et is dotsource-szal kell meghívni ([43]-as sorban a plusz pont és szóköz a prompt után):

```
[43] PS C:\powershell2> . .\scripts\include.ps1
[44] PS C:\powershell2> fv1
Első függvény
[45] PS C:\powershell2> fv2
Második függvény
```

Így már tökéletesen működik a szkriptkönyvtárunk.

2.1.1.1 A $MyInvocation felhasználása parancssor-elemzésre

Nézzük kicsit alaposabban meg ezt a `$myinvocation` változót. Ha interaktívan szeretnénk megnézni, akkor ezt kapjuk:

```
[1] PS C:\> "kakukk"; $myinvocation
kakukk


MyCommand        : "kakukk"; $myinvocation
ScriptLineNumber : 0
OffsetInLine     : 0
ScriptName       :
Line             :
PositionMessage  :
InvocationName   :
PipelineLength   : 2
PipelinePosition : 1
```

Látszik, hogy a `MyCommand` property tartalmazza, hogy mi is az éppen futtatott parancs. Mivel ezt ritkán használjuk interaktívan, nézzük meg, hogy egy szkriptből futtatva mit ad. Maga a szkript nagyon egyszerű:

```
$myinvocation | fl
```

És a kimenet:

```
[7] PS C:\> C:\powershell2\scripts\get-mycommand.ps1


MyCommand        : get-mycommand.ps1
ScriptLineNumber : 1
OffsetInLine     : -2147483648
ScriptName       :
Line             : C:\powershell2\scripts\get-mycommand.ps1
PositionMessage  :
                   At line:1 char:40
                   + C:\powershell2\scripts\get-mycommand.ps1 <<<<
InvocationName   : C:\powershell2\scripts\get-mycommand.ps1
PipelineLength   : 1
PipelinePosition : 1
```

Nézzük meg, hogy egy függvényben hogyan alakul ez a változó:

```
[8] PS C:\> function get-myinvocation {$myinvocation | fl *}
[9] PS C:\> "kakukk" | get-myinvocation


MyCommand        : get-myinvocation
ScriptLineNumber : 1
OffsetInLine     : -2147483648
```

```
ScriptName       :
Line             : "kakukk" | get-myinvocation
PositionMessage  :
                   At line:1 char:27
                   + "kakukk" | get-myinvocation <<<<
InvocationName   : get-myinvocation
PipelineLength   : 1
PipelinePosition : 1
```

A fenti példákból látszik, hogy elsősorban a `MyCommand` és a `Line` tulajdonság hasznos. Ha csak a szűken vett futtatási környezetre vagyunk kíváncsi, akkor a `MyCommand` kell nekünk, ha a teljes parancssor, akkor `Line`.

Ezt felhasználva készítsünk egy olyan függvényt, ami megismétli valahányszor az előtte formailag csővezetékként megadott kifejezést ilyen formában:

```
kifejezés | repeat-commandline 5
```

Az ezt megvalósító függvény:

```
[1] PS C:\> function repeat-commandline ([int] $x = 2)
>> {
>>     $s = $myinvocation.line
>>     $last = $s.LastIndexOf("|")
>>     if ($last -lt 0) {throw "Nincs mit ismételni!"}
>>     $cropped = $s.substring(0,$last)
>>     for ($r = 0; $r -lt $x; $r++)
>>     {
>>         invoke-expression $cropped
>>     }
>> }
>>
[2] PS C:\> "többször" | repeat-commandline 5
többször
többször
többször
többször
többször
```

A függvény lényegi része a `$myinvocation` automatikus változó `Line` tulajdonságának felhasználása. Nekünk itt a teljes parancssor kell, így a `Line` tulajdonságot használom. Megkeresem az utolsó csőjelet, (ha nincs ilyen benne, akkor hibát jelzek) és csonkolom odáig a kifejezést, majd egy ciklussal végrehajtom ezt annyiszor, amennyi a függvénynek átadott paraméter.

2.1.2 Környezeti változók (env:)

A DOS/Windows környezeti változókat is elérjük PowerShellből az `env`: PSDrive-on keresztül:

```
[3] PS C:\> cd env:

[4] PS Env:\> dir

Name                           Value
----                           -----
COMPUTERNAME                   ASUS
HOMEPATH                       \Documents and Settings\Administrator
USERNAME                       Administrator
PROCESSOR_ARCHITECTURE         AMD64
Path                           C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS...
CommonProgramFiles(x86)        C:\Program Files (x86)\Common Files
ProgramFiles(x86)              C:\Program Files (x86)
PROCESSOR_LEVEL                6
LOGONSERVER                    \\ASUS
HOMEDRIVE                      C:
USERPROFILE                    C:\Documents and Settings\Administrator
SystemRoot                     C:\WINDOWS
TEMP                           C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp
SESSIONNAME                    Console
ALLUSERSPROFILE                C:\Documents and Settings\All Users
FP_NO_HOST_CHECK               NO
APPDATA                        C:\Documents and Settings\Administrator\A...
ProgramFiles                   C:\Program Files
PATHEXT                        .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.W...
CLIENTNAME                     Console
OS                             Windows_NT
CommonProgramFiles             C:\Program Files\Common Files
PROCESSOR_IDENTIFIER           EM64T Family 6 Model 15 Stepping 10, Genu...
ComSpec                        C:\WINDOWS\system32\cmd.exe
SystemDrive                    C:
PROCESSOR_REVISION             0f0a
NUMBER_OF_PROCESSORS           2
TMP                            C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp
USERDOMAIN                     ASUS
windir                         C:\WINDOWS
```

Ezeket a változókat felhasználhatjuk szkriptjeinkben. Például keressük az összes PATH-ban található txt kiterjesztésű fájlt:

```
[41] PS C:\> $env:path.split(";") | ForEach-Object {$  + "\*"} | Get-ChildIt
em -Include *.txt


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS\system32


Mode           LastWriteTime       Length Name
----           -------------       ------ ----
-a---   2006.03.29.    14:00        41661 eula.txt
-a---   2008.01.18.    14:45            0 h323log.txt


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS
```

```
Mode           LastWriteTime       Length Name
----           -------------       ------ ----
-a---   2008.01.28.    12:38         1672 OEWABLog.txt
-a---   2008.01.18.    13:57       749897 setuplog.txt
...
```

Vagy például ki szeretném törölni az összes tmp kiterjesztésű átmeneti állományt:

```
[47] PS C:\> dir ($($env:temp)+"\*") -Include *.tmp | Remove-Item
```

Itt előfordulhat, hogy ha vannak zárolt állományaink, akkor törlésük nem sikerül, hibát kapunk, de a többi állományt törli a kifejezés.

2.1.3 Lépjünk kapcsolatba a konzolablakkal ($host)

Jó lenne a PowerShell ablakot is minél komfortosabbá tenni. Ennek néhány lehetősége (QuickEdit mode, ablakszélesség, stb.) már a *1.2.5 Gyorsbillentyűk, beállítások* fejezetben szerepelt. Az ott leírtaknál kicsit több is rendelkezésünkre áll, ehhez tudni kell, hogy létezik egy `$host` automatikus változó, ami a PowerShell ablak sok jellemzőjét tartalmazza.

```
[1] PS I:\>$host


Name             : ConsoleHost
Version          : 1.0.0.0
InstanceId       : 197f1a6f-6cdc-4e47-a81f-0fa93e8276e8
UI               : System.Management.Automation.Internal.Host.InternalHostU
                   serInterface
CurrentCulture   : hu-HU
CurrentUICulture : en-US
PrivateData      : Microsoft.PowerShell.ConsoleHost+ConsoleColorProxy
```

Közvetlenül a `$host` még nem igazán a mi barátunk, de nézzük ennek tagjellemzőit:

```
[3] PS I:\>$host.ui | gm


   TypeName: System.Management.Automation.Internal.Host.InternalHostUserInt
erface

Name                     MemberType Definition
----                     ---------- ----------
Equals                   Method     System.Boolean Equals(Object obj)
GetHashCode              Method     System.Int32 GetHashCode()
GetType                  Method     System.Type GetType()
get_RawUI                Method     System.Management.Automation.Host.PSHo...
Prompt                   Method     System.Collections.Generic.Dictionary`...
PromptForChoice          Method     System.Int32 PromptForChoice(String ca...
PromptForCredential      Method     System.Management.Automation.PSCredent...
ReadLine                 Method     System.String ReadLine()
```

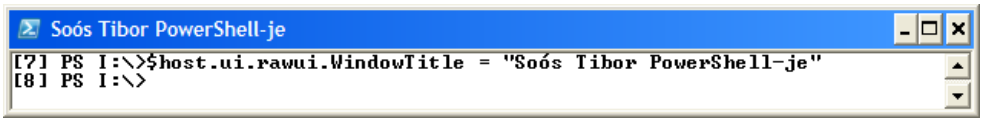
```
ReadLineAsSecureString Method     System.Security.SecureString ReadLineA...
ToString               Method     System.String ToString()
Write                  Method     System.Void Write(String value), Syste...
WriteDebugLine         Method     System.Void WriteDebugLine(String mess...
WriteErrorLine         Method     System.Void WriteErrorLine(String value)
WriteLine              Method     System.Void WriteLine(), System.Void W...
WriteProgress          Method     System.Void WriteProgress(Int64 source...
WriteVerboseLine       Method     System.Void WriteVerboseLine(String me...
WriteWarningLine       Method     System.Void WriteWarningLine(String me...
RawUI                  Property   System.Management.Automation.Host.PSHo...
```

Még itt sem biztos, hogy felcsillan a szemünk, de nézzük a sokat sejtető `RawUI`-t:

```
[4] PS I:\>$host.ui.rawui | gm


   TypeName: System.Management.Automation.Internal.Host.InternalHostRawUser
Interface

Name                        MemberType Definition
----                        ---------- ----------
Equals                      Method     System.Boolean Equals(Object obj)
FlushInputBuffer            Method     System.Void FlushInputBuffer()
GetBufferContents           Method     System.Management.Automation.Host.B...
GetHashCode                 Method     System.Int32 GetHashCode()
GetType                     Method     System.Type GetType()
get_BackgroundColor         Method     System.ConsoleColor get_BackgroundC...
get_BufferSize              Method     System.Management.Automation.Host.S...
get_CursorPosition          Method     System.Management.Automation.Host.C...
get_CursorSize              Method     System.Int32 get_CursorSize()
get_ForegroundColor         Method     System.ConsoleColor get_ForegroundC...
get_KeyAvailable            Method     System.Boolean get_KeyAvailable()
get_MaxPhysicalWindowSize   Method     System.Management.Automation.Host.S...
get_MaxWindowSize           Method     System.Management.Automation.Host.S...
get_WindowPosition          Method     System.Management.Automation.Host.C...
get_WindowSize              Method     System.Management.Automation.Host.S...
get_WindowTitle             Method     System.String get_WindowTitle()
LengthInBufferCells         Method     System.Int32 LengthInBufferCells(St...
NewBufferCellArray          Method     System.Management.Automation.Host.B...
ReadKey                     Method     System.Management.Automation.Host.K...
ScrollBufferContents        Method     System.Void ScrollBufferContents(Re...
SetBufferContents           Method     System.Void SetBufferContents(Coord...
set_BackgroundColor         Method     System.Void set_BackgroundColor(Con...
set_BufferSize              Method     System.Void set_BufferSize(Size value)
set_CursorPosition          Method     System.Void set_CursorPosition(Coor...
set_CursorSize              Method     System.Void set_CursorSize(Int32 va...
set_ForegroundColor         Method     System.Void set_ForegroundColor(Con...
set_WindowPosition          Method     System.Void set_WindowPosition(Coor...
set_WindowSize              Method     System.Void set_WindowSize(Size value)
set_WindowTitle             Method     System.Void set_WindowTitle(String ...
ToString                    Method     System.String ToString()
BackgroundColor             Property   System.ConsoleColor BackgroundColor...
BufferSize                  Property   System.Management.Automation.Host.S...
CursorPosition              Property   System.Management.Automation.Host.C...
CursorSize                  Property   System.Int32 CursorSize {get;set;}
ForegroundColor             Property   System.ConsoleColor ForegroundColor...
```

```
KeyAvailable               Property   System.Boolean KeyAvailable {get;}
MaxPhysicalWindowSize      Property   System.Management.Automation.Host.S...
MaxWindowSize              Property   System.Management.Automation.Host.S...
WindowPosition             Property   System.Management.Automation.Host.C...
WindowSize                 Property   System.Management.Automation.Host.S...
WindowTitle                Property   System.String WindowTitle {get;set;}
```

Itt már minden van, ami hasznos lehet. Gyakorlatilag az ablak legtöbb tulajdonsága ezen objektumon keresztül lekérdezhető és beállítható. Például cseréljük le az ablak fejlécének szövegét:



Billentyűleütésre váró programok készíthetők a `ReadKey()` metódus segítségével:

```
[13] PS I:\>$host.ui.rawui.ReadKey()
a
    VirtualKeyCode          Character    ControlKeyState            KeyDown
    --------------          ---------    ---------------            -------
                65                  a          NumLockOn               True
```

Kimeneteként láthatjuk a karakterkódot és a leütött karaktert magát is. Ha nem akarjuk látni a leütött karaktert, akkor használhatjuk a `ReadKey()` különböző opciói közül a `NoEcho`-t, amelyet vagy az `IncludeKeyDown`, vagy az `IncludeKeyDown` opcióval együtt kell használni:

```
[14] PS C:\>$host.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown")

    VirtualKeyCode          Character    ControlKeyState            KeyDown
    --------------          ---------    ---------------            -------
                74                  j          NumLockOn               True
```

Az `IncludeKeyUp` használatával csak a billentyű felengedésekor ad kimenetet a kifejezés. Ez akkor hasznos, ha a Shift vagy egyéb kiegészítő billentyűt is akarjuk használni, hiszen ha a lenyomásra élesedne, akkor már a Shifthez való hozzáéréskor lefutna a metódus és nem lenne idő az „igazi” billentyű lenyomására. Például egy Shift+Ctrl+Alt+w megnyomása esetén a metódus futásának eredménye így néz ki:

```
[15] PS C:\>$host.UI.RawUI.ReadKey("NoEcho,IncludeKeyUp") | fl *


VirtualKeyCode  : 87
Character       :
ControlKeyState : LeftAltPressed, LeftCtrlPressed, ShiftPressed, NumLockOn
KeyDown         : False
```

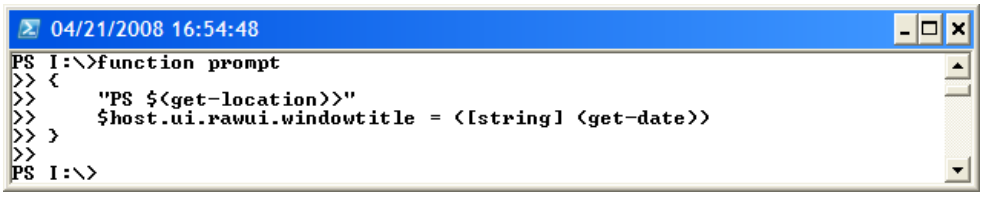
Ezzel, és még egy opció, az „AllowCtrlC” megadásával akár a Ctrl+C is megfigyelhető:

```
[16] PS C:\>$host.UI.RawUI.ReadKey("NoEcho,AllowCtrlC,IncludeKeyUp")

    VirtualKeyCode          Character    ControlKeyState            KeyDown
    --------------          ---------    ---------------            -------
                67                  ♥    LeftCtrlPressed              False
```

2.1.4 Prompt beállítása

A PowerShellben a prompt testre szabható. Mint ahogy a *1.8.10 Gyári függvények* fejezetben láttuk, a promptot egy automatikusan, minden beviteli sor megnyílásakor meghívódó `prompt` nevű függvény generálja. Ez a függvény átdefiniálható, testre szabható. Én ebben a könyvben a normál „`[PS] C:\>`” jellegű prompt elé biggyesztettem egy folytonosan növekvő sorszámot, hogy jobban lehessen hivatkozni a bemásolt parancssorokra. De természetesen a `prompt` függvénnyel nem csak olyan tevékenységeket végeztethetünk, ami kizárólag a megjelenő promptot szabja testre, hanem például a PowerShell ablakunk fejlécét is folyamatosan aktualizálhatjuk az aktuális időre.

Nézzünk erre egy példát:

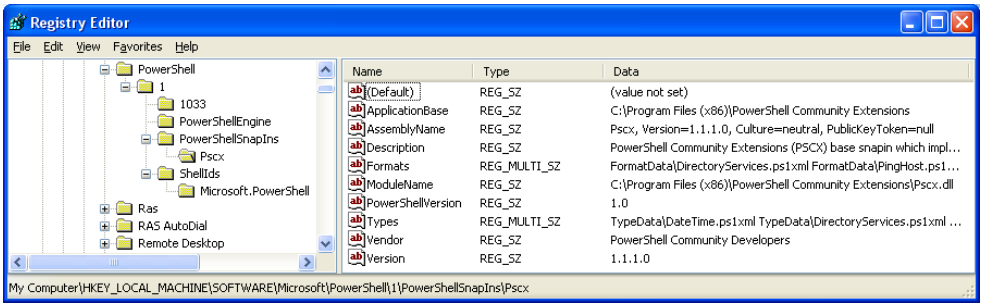


45. ábra A prompt testre szabása és annak eredménye

Látjuk, hogy amint a függvény definíciója megtörtént, már életbe is lép az új prompt. Azonban vigyázzunk, ha becsukjuk az ablakot és újra megnyitjuk, akkor visszakapjuk az eredeti promptot, a mi testre szabásunkat elfelejti. Ezen majd a nemsokára bemutatásra kerülő profilok segítenek majd.

2.1.5 Snapin-ek

A PowerShell moduláris felépítésű, az egyes cmdletek és providerek különböző moduloknak, u.n. snapin-eknek köszönhetők. A snapinek általában dll fájlok, és a registry-ben kell őket regisztrálni a PowerShell számára. Egy ilyen snapin regisztrációja valahogy így néz ki (csak akkor jön ez a registry ág létre, ha legalább egy, nem „gyári” snapinünk van):



46. ábra Snapin regisztrációja

Ezt természetesen nem nekünk, manuálisan kell bejegyezni, hanem a snapin gyártója általában biztosít valamilyen telepítőt, ami ezt megteszi. Ha van ilyen szépen előkészített snapinünk, akkor – ha a telepítője valamilyen egyéb módon nem integrálja be a PowerShell konzolba – a `Add-PSSnapin` cmdlettel be tudjuk építeni a PowerShell konzolba.

Snapinekből már eleve több van a PowerShell 1.0-ban, és ezeket a „gyári” snapineket nem kell külön regisztrálni. A snapinek lekérdezhetők a `get-pssnapin` cmdlettel:

```
[1] PS I:\>Get-PSSnapin


Name        : Microsoft.PowerShell.Core
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains Windows PowerShell m
              anagement cmdlets used to manage components of Windows PowerS
              hell.

Name        : Microsoft.PowerShell.Host
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains cmdlets used by the
              Windows PowerShell host.

Name        : Microsoft.PowerShell.Management
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains management cmdlets u
              sed to manage Windows components.

Name        : Microsoft.PowerShell.Security
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains cmdlets to manage Wi
              ndows PowerShell security.

Name        : Microsoft.PowerShell.Utility
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains utility Cmdlets used
               to manipulate data.
```

Snap-inekkel kapcsolatos egyéb cmdletek:

```
[3] PS I:\>Get-Command -noun pssnapin

CommandType     Name                               Definition
-----------     ----                               ----------
Cmdlet          Add-PSSnapin                       Add-PSSnapin [-Name] <Stri...
Cmdlet          Get-PSSnapin                       Get-PSSnapin [[-Name] <Str...
Cmdlet          Remove-PSSnapin                    Remove-PSSnapin [-Name] <S...
```

Azaz hozzá lehet adni (`Add-PSSnapin`) ezeket, le lehet kérdezni (`Get-PSSnapin`), és el lehet távolítani (`Remove-PSSnapin`).

Még azt is megnézhetjük, hogy az egyes cmdletek melyik snapinben tanyáznak:

```
[17] PS I:\>get-command | Select-Object name, pssnapin | Group-Object pssnap
in | ft -wrap

Count Name                      Group
----- ----                      -----
   47 Microsoft.PowerShell.Mana {Add-Content, Clear-Content, Clear-Item, Cl
      gement                    ear-ItemProperty...}
   12 Microsoft.PowerShell.Core {Add-History, Add-PSSnapin, Export-Console,
                                 ForEach-Object...}
   58 Microsoft.PowerShell.Util {Add-Member, Clear-Variable, Compare-Object
      ity                       , ConvertTo-Html...}
   10 Microsoft.PowerShell.Secu {ConvertFrom-SecureString, ConvertTo-Secure
      rity                      String, Get-Acl, Get-AuthenticodeSignature.
                                ..}
    2 Microsoft.PowerShell.Host {Start-Transcript, Stop-Transcript}
```

Mikor lehet erre szükség? Igazából az alap snap-inekkel én nem nagyon játszadoznék, de majd a *3. Bővítmények* fejezetben látunk olyan bővítményeket, amelyeket esetleg ki lehet kapcsolni, vagy éppen az alap PowerShell ablakhoz hozzá lehet adni.

Snap-inekkel kapcsolatos változtatások sem időt állók, azaz ha becsukjuk a PowerShell ablakot és újra megnyitjuk, akkor megint az alaphelyzet szerinti snap-inek köszönnek vissza.

Sajnos a `get-pssnapin` nem listázza ki azokat a beépülő modulokat, amelyek még nincsenek hozzáadva a rendszerhez, így nem is tudjuk, hogy mikkel bővíthetnénk azt. Készítsünk egy ilyen „hozzáadható” snap-in listát:

```
[54] PS C:\> function get-customsnapins
>> {
>>     Get-ChildItem hklm:\SOFTWARE\Microsoft\PowerShell\1\PowerShellSnapIns
 |
>>         Select-Object @{n="PSSnapIns" ; e={$ .pschildname}}
>> }
>>
[55] PS C:\> get-customsnapins

PSSnapIns
---------
PowerGadgets
Pscx
```

A fenti `get-customsnapins` függvény alaposabb megértéséhez kell majd a *2.5 Registry kezelése* fejezet elolvasása is, de azért nagyjából érthető így is a működése. Azaz kilistázom egy `Get-ChildItem` cmdlettel a snapineket tartalmazó registry-ág tartalmát, majd átalakítom az így kijövő objektumgyűjteményt, hogy csak a számomra fontos adat, a `PSChildName` kerüljön a kimenetre, méghozzá `PSSnapIns` néven.

2.1.6 Konzolfájl

Mind a szkriptkönyvtáraknál, mind a promptnál, mind pedig a snapineknél megjegyeztem, hogy a saját bővítményeinket, testre szabásainkat a PowerShell ablak elfelejti, ha becsukjuk. Ezért vannak olyan megoldások, amelyek a bővítményeink definiálását, beemelését végző szkriptjeinket automatikusan minden PowerShell ablak nyitáskor lefuttatják.

Az egyik ilyen lehetőséggel magát a konzolt tudjuk testre szabni a snapinek tekintetében függetlenül attól, hogy ki indítja el a PowerShell környezetet. Ezt konzolfájl segítségével tudjuk elérni, ilyen fájlt az `Export-Console` cmdlet segítségével lehet legegyszerűbben létrehozni. Nézzük, hogy most milyen snapinek vannak a gépemen a Start menüből elindított PowerShell ikon után megjelenő konzolban:

```
[3] PS C:\> Get-PSSnapin | ft -Property Name

Name
----
Microsoft.PowerShell.Core
Microsoft.PowerShell.Host
Microsoft.PowerShell.Management
Microsoft.PowerShell.Security
Microsoft.PowerShell.Utility
Pscx
```

Látszik, hogy én már telepítettem egy bővítményt, amiről majd a *3.2 PCX – PowerShell Community Extensions* fejezetben lesz szó.

Nézzük ezek után, mit eredményez az `export-console` cmdlet:

```
[3] PS C:\> Export-Console C:\powershell2\scripts\console
[4] PS C:\> Get-Content C:\powershell2\scripts\console.psc1
<?xml version="1.0" encoding="utf-8"?>
<PSConsoleFile ConsoleSchemaVersion="1.0">
  <PSVersion>1.0</PSVersion>
  <PSSnapIns>
    <PSSnapIn Name="Pscx" />
  </PSSnapIns>
</PSConsoleFile>
```

Ez a cmdlet egy psc1 kiterjesztésű XML fájlt generál, aminek a mélyén ott található azon snapineknek a listája, amelyek nem „gyáriak”. Ha egy ilyen psc1 fájlra duplán kattintunk, vagy a powershell.exe paramétereként szerepeltetjük, akkor az itt felsorolt

snapineket automatikusan betölti a konzol. Nézzük, hogy hogyan nézne ki ezzel a powershell.exe felparaméterezése:

```
powershell.exe -PSConsoleFile C:\powershell2\scripts\console.psc1
```

2.1.7 Profilok

Láttuk, hogy konzolfájlokkal a snapineket tudjuk automatikusan testre szabni. Ennél sokkal több lehetőséget biztosítanak a profilok.

A profilok egyszerű fájlban tárolt szkriptek, amelyek a PowerShell indításakor automatikusan lefutnak. Kicsit hasonlatosak a Windows StartUp mappáihoz. A PowerShellben összesen négy különböző hely van, ahova létrehozhatjuk a profilunkat:

Elérési út	Magyarázat
%windir%\system32\WindowsPowerShell\v1.0\ profile.ps1	Az adott gép összes felhasználójának, mindenfajta PowerShelles ügyködésére hatással van.
%windir%\system32\WindowsPowerShell\v1.0\ Microsoft.PowerShell_profile.ps1	Az adott gép összes felhasználójának, de csak az alaphelyzet szerinti Microsoft PowerShell konzol ügyködésére van hatással.
%UserProfile%\My Documents\ WindowsPowerShell\profile.ps1	Az adott gép adott felhasználójának, mindenfajta PowerShelles ügyködésére hatással van.
%UserProfile%\\My Documents\ WindowsPowerShell\ Microsoft.PowerShell_profile.ps1	Az adott gép adott felhasználójának csak az alaphelyzet szerinti Microsoft PowerShell ügyködésére van hatással.

Mit jelent az, hogy "Microsoft PowerShell ügyködés"? Azt, hogy nem feltétlenül csak a hagyományos "DOS" ablakos kinézetet veheti fel a PowerShell, egyéb gyártók más, akár grafikus felületeket is csinálhatnak. Na, ilyen esetekben nekik lehet külön profil fájljuk is a Microsoftoshoz képest.

Ilyen profilban meghívhatok „include” jellegű, függvénykönyvtárak betöltését végző szkripteket, testre szabhatjuk a promptot, globális változókat definiálhatok, de akár még snapineket is betölthetek. Én a sorszámozott promptomhoz egy globális `$promptno` változót és a prompt testre szabott függvénydefinícióját tettem bele egy ilyen profilba.

2.1.8 Örökítsük meg munkánkat (start-transcript)

A PowerShell magnóként is tud működni, azaz az összes parancssort, amit mi begépelünk, és az összes kimenetet, amit kapunk a konzolra képes elmenteni automatikusan egy fájlba. Ezt az üzemmódot a `start-transcript` cmdlettel tudjuk elindítani, és a `stop-transcript`-tel megállítani:

```
[6] PS C:\> Start-Transcript
Transcript started, output file is C:\Documents and Settings\Administrator\
My Documents\PowerShell transcript.20080421212835.txt
[7] PS C:\> $a = "Valaki figyel!"
[8] PS C:\> $a.Split()
Valaki
figyel!
[9] PS C:\> Stop-Transcript
Transcript stopped, output file is C:\Documents and Settings\Administrator\
My Documents\PowerShell_transcript.20080421212835.txt
```

Ha nem adjuk meg, hogy milyen fájlba rögzítsen, akkor az aktuális felhasználó dokumentumkönyvtárába ment, dátummal, idővel ellátott nevű szöveges fájlba. Nézzük, hogy hogyan néz ki egy ilyen fájl:

```
[11] PS C:\> Get-Content 'C:\Documents and Settings\Administrator\My Documen
ts\PowerShell transcript.20080421212835.txt'
**********************
Windows PowerShell Transcript Start
Start time: 20080421212835
Username  : ASUS\Administrator
Machine   : ASUS (Microsoft Windows NT 5.2.3790 Service Pack 2)
**********************
Transcript started, output file is C:\Documents and Settings\Administrator\
My Documents\PowerShell_transcript.20080421212835.txt
[7] PS C:\> $a = "Valaki figyel!"
[8] PS C:\> $a.Split()
Valaki
figyel!
[9] PS C:\> Stop-Transcript
**********************
Windows PowerShell Transcript End
End time: 20080421212912
**********************
```

Látszik, hogy mindent, még a promptokat is szóról-szóra rögzítette fejléc és lábléc információk között. Ennek a fájlnak egyfajta feldolgozására látunk példát a *2.3.4 Sortörés kezelése szövegfájlokban* fejezetben.

2.1.9 Stopperoljuk a futási időt és várakozzunk (measure-command, start-sleep)

Bizonyos esetekben fontos lehet a szkriptünk futásának hatékonysága, melynek egyik ismérve, hogy milyen gyorsan fut le a szkript. Szerencsére nem kell stopperórával a kezünkben figyelni a konzolt, mert a PowerShell rendelkezik beépített stopperrel, melyet a `measure-command` cmdlettel tudunk üzembe helyezni.

Például stopperoljuk le, hogy mennyi ideig tart végiglistázni a c:\ meghajtó összes alkönyvtárát és fájlját:

```
[5] PS C:\> Measure-Command {get-childitem c:\ -recurse}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 33
Milliseconds      : 306
Ticks             : 333067130
TotalDays         : 0,000385494363425926
TotalHours        : 0,00925186472222222
TotalMinutes      : 0,555111883333333
TotalSeconds      : 33,306713
TotalMilliseconds : 33306,713
```

Láthatjuk, hogy a `measure-command` paramétere egy szkriptblokk, kimenete egy `timespan` típusú objektum, amelyben mindenféle mértékegységben láthatjuk az eltelt időt.

Ha egy szkriptünknél pont az lenne a feladat, hogy ne legyen túl gyors, mert például percenként kellene, hogy valamilyen paramétert kiolvassunk, akkor a `start-sleep` cmdlettel tudunk várakozni:

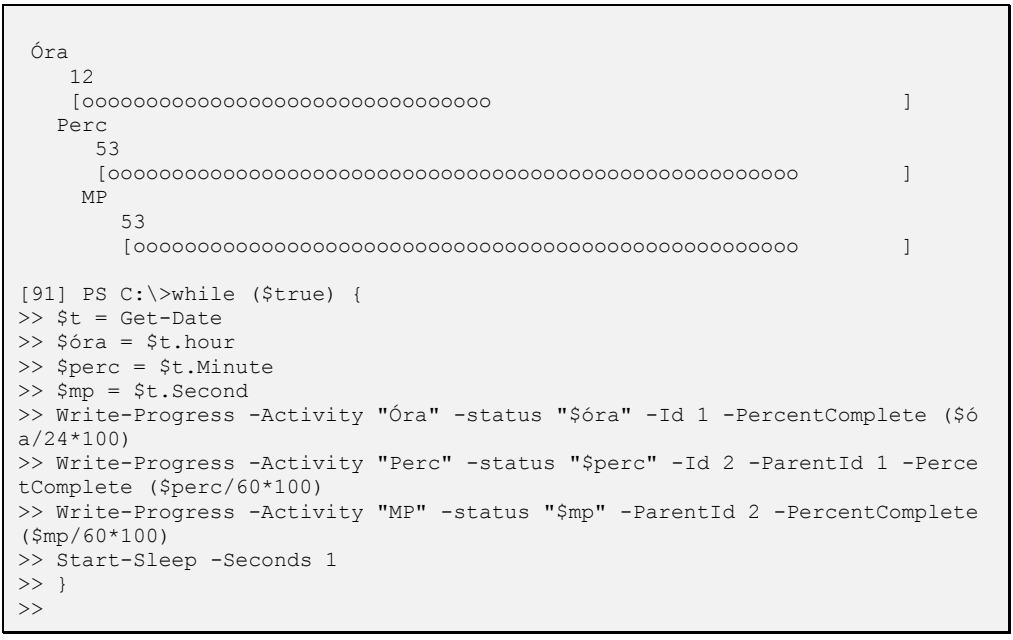
```
[12] PS C:\> get-date; start-sleep 10; get-date

2008. április 24. 22:40:59
2008. április 24. 22:41:09
```

A fenti példában 10 másodpercet várakoztam, ha rövidebb várakozási időre van szükség, akkor `–milliseconds` paraméterrel lehet ezt megadni.

A `measure-command` és a `start-sleep` kombinálásával lehet olyan ciklust írni, ami például pontosan minden egész percben fut le. Azaz mérhetjük, hogy mennyi ideig tart a futása egy ciklusnak, majd kiegészítésként annyit várunk, hogy a következő perc elején induljon újra a ciklusunk.

2.1.10 Előrehaladás jelzése (write-progress)

Ha már időmérés és várakozás, akkor érdemes megemlíteni, hogy szép „progress bar”-t, azaz előrehaladás-jelzőt is megjeleníthetünk PowerShellben a `write-progress` cmdlet segítségével. Ez főleg akkor hasznos, ha időigényesebb valamelyik parancssorunk, szkriptünk és nyomon szeretnénk követni, hogy ténylegesen hol is tart a futás. Példaként nézzünk egy „analóg” órát:




```
[91] PS C:\>while ($true) {
>> $t = Get-Date
>> $óra = $t.hour
>> $perc = $t.Minute
>> $mp = $t.Second
>> Write-Progress -Activity "Óra" -status "$óra" -Id 1 -PercentComplete ($ó
a/24*100)
>> Write-Progress -Activity "Perc" -status "$perc" -Id 2 -ParentId 1 -Perce
tComplete ($perc/60*100)
>> Write-Progress -Activity "MP" -status "$mp" -ParentId 2 -PercentComplete
($mp/60*100)
>> Start-Sleep -Seconds 1
>> }
>>
```

Sajnos nyomtatásban nem annyira „szép” a kép, nem látszik az ablak tetején a türkizes háttérszínen, de a kis karikákból álló előrehaladás-jelzőt azért látható. Maga a szkript, ami ezt produkálja a karikás sorok alatt található. Egy `while` ciklusban folyamatosan kiolvasom az aktuális rendszeridőt, szétszedem külön óra, perc, másodpercre. Ezután jön a lényegi rész. Az egyes „csíkoknak” adhatok címkét (`Activity` paraméter), státusz címkét (`Status`), azonosító számot (`Id` paraméter), az előrehaladás értékét (0 és 100 közötti szám, `PercentComplete` paraméter). Látszik, hogy a kijelzők egymásba is ágyazhatók, ha definiálom a `ParentID` paraméterrel a „szülő” csíkot. Ekkor a „gyerek” a szülő alatt fog megjelenni, kicsit beljebb.

2.2 Hibakezelés

Az elméleti részben nem foglalkoztam hibákkal, hiszen elméletileg minden tökéletes ☺, na de a gyakorlat! Nézzük meg, hogy a PowerShell milyen lehetőségeket nyújt a hibák detektálására, elhárítására, illetve milyen lehetőségek vannak arra, ha a felmerülő hibákat én a szkriptemben akarom lekezelni.

2.2.1 Megszakító és nem megszakító hibák

Elsőként nézzünk pár fogalmat. A PowerShellben két hibafajta van, a „terminating error”, azaz a futást mindenképpen megszakító, és a „nonterminating error”, azaz a futást nem feltétlenül megszakító hiba.

Megszakító hibák például a szintaktikai hibák, amikor elgépelünk valamit. Vagy például amikor nullával szeretnénk osztani. Mi magunk is generálhatunk ilyen hibákat a korábban már látott `throw` kulcsszóval, amikor például egy függvényünknek nem ad át a felhasználó minden fontos paramétert.

Előfordulhatnak olyan hibák, amelyek előállásakor nem kívánjuk, hogy a szkript futása megszakadjon, de azért szeretnénk értesülni ezekről. Ez főleg olyan cmdleteknél és függvényeknél jöhet jól, amelyek csőelemeket dolgoznak fel, és egy-két csőelem esetében megengedjük, hogy ne fusson le a szkript, de azért a többi elemre nyugodtan próbálkozzon.

Nézzünk példát a kétfajta hibára, elsőként a megszakító hibára, rögtön két példát is:

```
[7] PS C:\old> "Eleje"; 15/0; "Vége"
Attempted to divide by zero.
At line:1 char:13
+ "Eleje"; 15/0 <<<< ; "Vége"
```

A [7]-es sorban 0-val osztok. Ez olyannyira „terminating”, hogy már a parancssor ellenőrzése során kiszúrja ezt a hibát, és már az „Eleje” sem fut le.

```
[8] PS C:\old> "Eleje"; throw "kakukk"; "Vége"
Eleje
kakukk
At line:1 char:15
+ "Eleje"; throw  <<<< "kakukk"; "Vége"
```

A [8]-as sorban én magam okoztam egy megszakító hibát a `throw` kulcsszó segítségével.

Most nézzünk nem megszakító hibára példát:

```
[9] PS C:\old> "Eleje"; Remove-Item c:\old\nincs.txt; "Vége"
Eleje
Remove-Item : Cannot find path 'C:\old\nincs.txt' because it does not exist
.
At line:1 char:21
```

```
+ "Eleje"; Remove-Item  <<<< c:\old\nincs.txt; "Vége"
Vége
```

A fenti példában nem megszakító hiba történt, amikor törölni akartam egy nem létező fájlt, ugyan hibát kaptam, de kiírásra került a „Vége” is.

Mi magunk is befolyásolhatjuk a hibák lefolyását. Akár parancsonként vagy akár az adott konzolra vonatkozóan is. Először nézzük, hogy a cmdletek esetében mit tehetünk. A parancsokhoz tartozó súgókban legtöbbször valami hasonlót látunk:

```
[18] PS C:\old> (get-help remove-item).syntax

Remove-Item [-path] <string[]> [-recurse] [-force] [-include <string[]>] [-
exclude <string[]>] [-filter <string>] [-credential <PSCredential>] [-whatI
f] [-confirm] [<CommonParameters>]
Remove-Item [-literalPath] <string[]> [-recurse] [-force] [-include <string
[]>] [-exclude <string[]>] [-filter <string>] [-credential <PSCredential>]
[-whatIf] [-confirm] [<CommonParameters>]
```

Nézzük meg, a „`CommonParameters`” mi lehet?

Paraméter	Magyarázat
Verbose	Bőbeszédes kimenetet ad a művelet lefolyásáról.
Debug	Hibakereső információkat ad, és interaktív módon lekezelhetők a hibák.
ErrorAction	Az előzőhöz hasonló, de nem csak interaktívan, hanem fixen beállítható hibakezelési mód: `Continue` [default] - folytat, `Stop` - megáll, `SilentlyContinue` – figyelmeztetés nélkül továbbmegy, `Inquire` - rákérdez.
ErrorVariable	Saját hibaváltozónk neve ($ jel nélkül!). A `$error` változó mellett ide is betöltődik a hibát leíró objektum.
OutVariable	Kimenetet ide tölti be.
OutBuffer	Az objektum-puffer mérete, ennyi elemet „magában” tart, mielőtt továbbítja az outputot a következő csőszakasznak.

A `-debug` paraméter pont a mostani témánkba vág, nézzük, milyen opciókat kínál:

```
[23] PS C:\old> "Eleje"; get-content blabla.txt -debug; "Vége"
Eleje

Confirm
Cannot find path 'C:\old\blabla.txt' because it does not exist.
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):h
Get-Content : Command execution stopped because the user selected the Halt
option.
At line:1 char:21
```

```
+ "Eleje"; get-content  <<<< blabla.txt -debug; "Vége"
```

Láthatjuk, hogy ezzel a paraméterrel a nem megszakító hibák lefolyásáról dönthetünk interaktív módon. Ha a „Halt” opciót választom, akkor kiírta a megállás okát, és a „Vége” nem futott le. Nézzünk, hogy hogyan viselkedik más válasz esetén:

```
[24] PS C:\old> "Eleje"; get-content blabla.txt -debug; "Vége"
Eleje

Confirm
Cannot find path 'C:\old\blabla.txt' because it does not exist.
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):a
Get-Content : Cannot find path 'C:\old\blabla.txt' because it does not exis
t.
At line:1 char:21
+ "Eleje"; get-content  <<<< blabla.txt -debug; "Vége"
Vége
```

Ha „Yes” vagy „Yes to All” opciót választom, akkor a cmdlet úgy fut le, ahogy eddig láttuk a nem megszakító hibáknál, azaz kiírta a hibát, de tovább futott. Nézzük a további választási lehetőségünket:

```
[25] PS C:\old> "Eleje"; get-content blabla.txt -debug; "Vége"
Eleje

Confirm
Cannot find path 'C:\old\blabla.txt' because it does not exist.
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):s
```

Ha a „Suspend” opciót választjuk, akkor a szkript futása megszakad, és nem is ír ki semmilyen további hibajelzést. Azaz hasonló az eredmény a Halt opció választásakor tapasztaltakhoz, csak itt nem íratunk ki részletesebb hibaleírást.

Nézzük a másik témánkba vágó „CommonParameter”-t, az –ErrorAction-t:

```
[44] PS C:\old> "Eleje"; get-content blabla.txt -ErrorAction Stop; "Vége"
Eleje
Get-Content : Command execution stopped because the shell variable "ErrorAc
tionPreference" is set to Stop: Cannot find path 'C:\old\blabla.txt' becaus
e it does not exist.
At line:1 char:21
+ "Eleje"; get-content  <<<< blabla.txt -ErrorAction Stop; "Vége"
```

Ha a Stop opciót választjuk, akkor a szkript a nem megszakító hibáknál is megszakítódik.

```
[45] PS C:\old> "Eleje"; get-content blabla.txt -ErrorAction SilentlyContinu
e; "Vége"
Eleje
```

```
Vége
```

A `SilentlyContinue` opcióval folytatja a szkriptet és nem is ad semmilyen hibajelzést.

```
[46] PS C:\old> "Eleje"; get-content blabla.txt -ErrorAction Inquire; "Vége"

Eleje

Confirm
Cannot find path 'C:\old\blabla.txt' because it does not exist.
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
(default is "Y"):a
Get-Content : Cannot find path 'C:\old\blabla.txt' because it does not exis
t.
At line:1 char:21
+ "Eleje"; get-content  <<<< blabla.txt -ErrorAction Inquire; "Vége"
Vége
```

Ha az `Inquire` opciót választjuk, akkor - hasonlóan a `-debug` paraméter használatakor látottakhoz - rákérdez a folytatás mikéntjére.

2.2.2 Hibajelzés kiolvasása ($error)

Az előzőekben a hibajelzéseket a képernyőn olvastuk el, de ezeket a PowerShell automatikusan egy `$error` tömbbe gyűjti. A [0]-ás indexű elem mindig a legfrissebb hibajelzést tartalmazza, és így tolja mindig el eggyel a hibákat. Ez azért jó, mert ha ki is kapcsoljuk a hibajelzést (`SilentlyContinue`), az `$error` változó ennek ellenére őrizni fogja a fellépő hibát:

```
[54] PS C:\old> "54Eleje"; get-content blabla.txt -ErrorAction SilentlyConti
nue; "54Vége"
54Eleje
54Vége
[55] PS C:\old> $error[0]
Get-Content : Cannot find path 'C:\old\blabla.txt' because it does not exis
t.
At line:1 char:23
+ "54Eleje"; get-content  <<<< blabla.txt -ErrorAction SilentlyContinue; "5
4Vége"
```

Ráadásul – most már nem is csodálkozunk – a `$error` elemei is objektumok! Nézzük meg a tagjellemzőit:

```
[59] PS C:\old> $error[0] | gm


   TypeName: System.Management.Automation.ErrorRecord

Name                        MemberType Definition
```

```
----                        ---------- ----------
Equals                      Method     System.Boolean Equals(Object obj)
GetHashCode                 Method     System.Int32 GetHashCode()
GetObjectData               Method     System.Void GetObjectData(Serializa...
GetType                     Method     System.Type GetType()
get CategoryInfo            Method     System.Management.Automation.ErrorC...
get ErrorDetails            Method     System.Management.Automation.ErrorD...
get Exception               Method     System.Exception get Exception()
get_FullyQualifiedErrorId   Method     System.String get_FullyQualifiedErr...
get_InvocationInfo          Method     System.Management.Automation.Invoca...
get TargetObject            Method     System.Object get TargetObject()
set ErrorDetails            Method     System.Void set ErrorDetails(ErrorD...
ToString                    Method     System.String ToString()
CategoryInfo                Property   System.Management.Automation.ErrorC...
ErrorDetails                Property   System.Management.Automation.ErrorD...
Exception                   Property   System.Exception Exception {get;}
FullyQualifiedErrorId       Property   System.String FullyQualifiedErrorId...
InvocationInfo              Property   System.Management.Automation.Invoca...
TargetObject                Property   System.Object TargetObject {get;}
```

Itt igazából a tulajdonságok az érdekesek számunkra:

```
[62] PS C:\old> $error[0].categoryinfo


Category   : ObjectNotFound
Activity   : Get-Content
Reason     : ItemNotFoundException
TargetName : C:\old\blabla.txt
TargetType : String



[63] PS C:\old> $error[0].errordetails
[64] PS C:\old> $error[0].fullyqualifiederrorid
PathNotFound,Microsoft.PowerShell.Commands.GetContentCommand
[65] PS C:\old> $error[0].invocationinfo


MyCommand        : Get-Content
ScriptLineNumber : 1
OffsetInLine     : -2147483648
ScriptName       :
Line             : "54Eleje"; get-content blabla.txt -ErrorAction SilentlyC
                   ontinue; "54Vége"
PositionMessage  :
                   At line:1 char:23
                   + "54Eleje"; get-content  <<<< blabla.txt -ErrorAction S
                   ilentlyContinue; "54Vége"
InvocationName   : get-content
PipelineLength   : 1
PipelinePosition : 1



[66] PS C:\old> $error[0].targetobject
```

```
C:\old\blabla.txt
```

A `CategoryInfo` tulajdonság önmagában nagyon sok mindent elárul a hibáról, amit az `InvocationInfo` még további részleteket láthatunk. Ezen információk birtokában tényleg nem jelenthet gondot a hibák felderítése és elhárítása.

2.2.3 Hibakezelés globális paraméterei

Az előzőekben látott hibakezelési módszerek úgy működtek, hogy egy adott cmdletnél állítottam be a „`CommonParameter`-ek” segítségével, hogy az hogyan reagáljon egy hibára. Azonban ez elég nehézkes egy hosszabb szkript esetében. Szerencsére ezt a fajta működést szerencsére globálisan is beállíthatjuk. Nézzük meg, hogy milyen „error”-ral kapcsolatos változóink vannak:

```
[69] PS C:\old> Get-ChildItem variable:\*error*

Name                           Value
----                           -----
Error                          {PathNotFound,Microsoft.PowerShell.Comman...
ReportErrorShowSource          1
ReportErrorShowStackTrace      0
ReportErrorShowExceptionClass  0
ErrorActionPreference          Continue
MaximumErrorCount              256
ReportErrorShowInnerException  0
ErrorView                      NormalView
```

Nézzük ezek közül azoknak a magyarázatát, amelyek a mindennapi használatkor érdekesek:

Változó	Magyarázat
$Error	A korábban már látott hibajelzések tömbje.
$ErrorActionPreference	Globális hibakezelési mód: `Continue` [default] - folytat, `Stop` - megáll, `SilentlyContinue` – figyelmeztetés nélkül továbbmegy, `Inquire` - rákérdez.
$MaximumErrorCount	Az `$error` tömb maximális mérete. Az ennél régebbi (nagyobb sorszámú) hibajelzések kihullnak a tömbből.
$ErrorView	A hibajelzések nézete: `Normal` vagy `CategoryView`

2.2.4 Hibakezelés saját függvényeinkben (trap)

Az eddigi sok okosságot a hibakezeléssel kapcsolatban mind a cmdletek fejlesztőinek köszönhetjük. Azaz hogy egy cmdlet rendelkezik az `-ErrorAction` paraméterrel, és ennek különböző értékeire hogyan reagál, az mind leprogramozandó. Ha mi készítünk egy függvényt vagy szkriptet, és abban mi is figyelembe akarjuk venni a globális `$ErrorActionPreference` változó értékét, vagy mi is implementálni akarjuk az `-ErrorAction` paramétert, akkor ezt a függvényünk, szkriptünk belsejében nekünk le kell programozni.

Ehhez a PowerShell nyelvi szinten a `trap` kulcsszót biztosítja. Ennek szintaxisa így néz ki:

```
trap [ExceptionType] {code; keyword}
```

A `trap`-pel tehát át lehet venni a vezérlést a hibák felmerülésekor. Nézzünk erre pár példát:

```
trap
{
        "Hibajelenség: $_"
}

Remove-Item c:\nemlétezőfile.txt
```

A fenti példa egy szkript, nézzük mit ad ez kimenetként, ha lefuttatom:

```
[2] PS C:\powershell2\scripts> .\trap1.ps1
Remove-Item : Cannot find path 'C:\nemlétezőfile.txt' because it does not e
xist.
At C:\powershell2\scripts\trap1.ps1:6 char:12
+ Remove-Item  <<<< c:\nemlétezőfile.txt
```

Nem sok mindent. Mivelhogy a `trap` csak megszakító hibákra éled fel, azaz a fenti nem megszakító hibát át kell alakítani megszakító hibává. Nézzük a módosított szkriptet:

```
trap
{
        "Hibajelenség: $_"
}

Remove-Item c:\nemlétezőfile.txt -ErrorAction Stop
```

Itt már mindenképpen megállíttatom a cmdlet futását bármilyen hibajelenségre. És ennek kimenete:

```
[4] PS C:\powershell2\scripts> .\trap2.ps1
Hibajelenség: Command execution stopped because the shell variable "ErrorAc
tionPreference" is set to Stop: Cannot find path 'C:\nemlétezőfile.txt' bec
ause it does not exist.
```

```
Remove-Item : Command execution stopped because the shell variable "ErrorAc
tionPreference" is set to Stop: Cannot find path 'C:\nemlétezőfile.txt' bec
ause it does not exist.
At C:\powershell2\scripts\trap2.ps1:6 char:12
+ Remove-Item  <<<< c:\nemlétezőfile.txt -ErrorAction Stop
```

Itt már megjelenik az én `trap` kódom. A `trap` megkapja a hibát jelző objektumot a `$_` változóba, amit ki is írtam. Viszont nem rendelkeztem arról, hogy a saját hibakezelő rutinom lefutása után mi történjen, így utána visszakapja a vezérlést a PowerShell, ami szintén kiírja a hibajelzést. Ez felesleges, így módosítsuk a hibakezelést:

```
trap
{
        "Hibajelenség: $ "
        continue
}

Remove-Item c:\nemlétezőfile.txt -ErrorAction Stop
```

Ebben a változatban a `trap` szkriptblokkjába elhelyeztem egy `continue` kulcsszót, ami azt jelzi a PowerShellnek, hogy már minden rendben, neki már nem kell foglalkozni a hibával. Nézzük ekkor a kimenetet:

```
[5] PS C:\powershell2\scripts> .\trap3.ps1
Hibajelenség: Command execution stopped because the shell variable "ErrorAc
tionPreference" is set to Stop: Cannot find path 'C:\nemlétezőfile.txt' bec
ause it does not exist.
```

Fejlesszük tovább a kódot, jó lenne kicsit hasznosabb információkat is kiírni a hibáról:

```
trap
{
        "Hibajelenség: $_"
        "-------- Részletek --------------"
        $_.Exception.GetType().FullName
        $_.InvocationInfo
        "-------- Részletek vége ---------"
        continue
}

Remove-Item c:\nemlétezőfile.txt -ErrorAction Stop
```

Nézzük ennek a kimenetét:

```
[37] PS C:\powershell2\scripts> .\trap4.ps1
Hibajelenség: Command execution stopped because the shell variable "ErrorAc
tionPreference" is set to Stop: Cannot find path 'C:\nemlétezőfile.txt' bec
ause it does not exist.
-------- Részletek --------------
System.Management.Automation.ActionPreferenceStopException
```

```
MyCommand        : Remove-Item
ScriptLineNumber : 11
OffsetInLine     : 12
ScriptName       : C:\powershell2\scripts\trap4.ps1
Line             : Remove-Item c:\nemlétezőfile.txt -ErrorAction Stop
PositionMessage  :
                   At C:\powershell2\scripts\trap4.ps1:11 char:12
                   + Remove-Item  <<<< c:\nemlétezőfile.txt -ErrorAction St
                   op
InvocationName   : Remove-Item
PipelineLength   : 1
PipelinePosition : 1

-------- Részletek vége ---------
```

A „Részletek” első eleme a hibajelenség típusa, azaz az `ExceptionType`. Jelen esetben ez egy:

`System.Management.Automation.ActionPreferenceStopException` típusú hiba. Ha csak bizonyos típusú hibákra szeretném, hogy a `trap`-em reagáljon, akkor ezt a típuselnevezést kell a `trap` paramétereként beírni (lásd nemsokára).

Ezután kiírattam a hibaobjektum `InvocationInfo` paraméterét, ami természetesen önmaga is egy objektum, így annak is számos tulajdonsága van. Ezen tulajdonságok között láthatjuk, hogy milyen parancs végrehajtása során lépett fel a hiba, milyen szkriptben, mi volt a teljes parancssor, stb. Ezek az információk hasznos segítséget nyújtanak a hiba felderítésében.

Nézzük, mit kapunk, ha egy másik hibajelenséget kap a `trap`-em, mondjuk egy nullával való osztást:

```
trap
{
        "Hibajelenség: $ "
        "-------- Részletek ---------------"
        $_.Exception.GetType().FullName
        $_.InvocationInfo
        "-------- Részletek vége ---------"
        continue
}

$nulla = 0
8/$nulla
```

Kicsit trükközni kellett, hiszen korábban láttuk, hogy ha közvetlenül ezt írnám. „8/0”, akkor azt már maga a parancsértelmező kiszúrná, és u.n. „nontrappable”, azaz nem elkapható hibaként kezelné. Ezért elrejtem egy változóba a nullát, és így osztok vele. Nézzük a kimenetet:

```
[38] PS C:\powershell2\scripts> .\trap5.ps1
Hibajelenség: Attempted to divide by zero.
-------- Részletek --------------
System.DivideByZeroException
```

```
MyCommand        :
ScriptLineNumber : 12
OffsetInLine     : 3
ScriptName       : C:\powershell2\scripts\trap5.ps1
Line             : 8/$nulla
PositionMessage  :
                   At C:\powershell2\scripts\trap5.ps1:12 char:3
                   + 8/$ <<<< nulla
InvocationName   : /
PipelineLength   : 0
PipelinePosition : 0

-------- Részletek vége ---------
```

Látjuk a részletek első sorában, hogy ez egy másfajta hiba, egy `System.DivideByZeroException` típusú. Ezek után, ha én csak az ilyen fajta hibákat akarom lekezelni, akkor a `trap` definícióját kell módosítani:

```
trap [System.DivideByZeroException]
{
       "Hibajelenség: $ "
       "-------- Részletek --------------"
       $_.Exception.GetType().FullName
       $_.InvocationInfo
       "-------- Részletek vége ---------"
       continue
}

$nulla = 0
8/$nulla

Remove-Item nemlétezőfile.txt
```

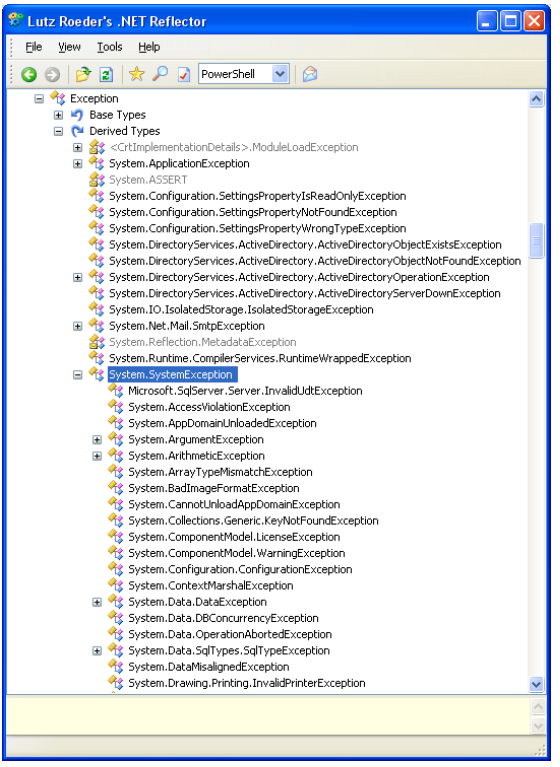
Ebben a szkriptben a `trap` mellett látható annak a hibatípusnak a neve, amire szeretnénk, ha a `trap`-ünk reagálna. Az összes többi hibafajtát a PowerShell fogja lekezelni. A szkript végén kétfajta hibát okozó műveletet hajtok végre. Nézzük, hogyan reagál ezekre a `trap`:

```
[39] PS C:\powershell2\scripts> .\trap6.ps1
Hibajelenség: Attempted to divide by zero.
-------- Részletek --------------
System.DivideByZeroException


MyCommand        :
ScriptLineNumber : 12
OffsetInLine     : 3
ScriptName       : C:\powershell2\scripts\trap6.ps1
Line             : 8/$nulla
PositionMessage  :
                   At C:\powershell2\scripts\trap6.ps1:12 char:3
```

```
                  + 8/$ <<<< nulla
InvocationName   : /
PipelineLength   : 0
PipelinePosition : 0

-------- Részletek vége ---------
Remove-Item : Cannot find path 'C:\powershell2\scripts\nemlétezőfile.txt' b
ecause it does not exist.
At C:\powershell2\scripts\trap6.ps1:14 char:12
+ Remove-Item  <<<< nemlétezőfile.txt
```

Láthatjuk, hogy csak egyszer fut le az én hibakezelő kódom, a nullával való osztásra. A nem létező fájl törlésekor a PowerShell eredeti hibakezelője futott le.

Az ilyen hibatípusokat legegyszerűbben „megtapasztalás” által lehet felderíteni vagy egy általános trapkezelő rutinnal, vagy az `$error` tömb elemeinek vizsgálatával. Ha valaki előre fel akar készülni a lehetséges hibákra, akkor a .NET Framework `System.Exception` leszármaztatott objektumait kell nézni, például a már említett Reflector programmal:



47. ábra Exception típusok a .NET Frameworkben a Reflector programmal szemlélve

Nagyon sok ilyen hibatípus van, így valószínű a megtapasztalás által könnyebben eredményre jutunk.

2.2.4.1 Többszintű csapda

Az előbb látott `trap`-eket (csapdákat) a szkriptünk bármelyik szintjén használhatjuk: a globális scope-ban, szkriptek vagy függvények al-scope-jaiban is. Ha egy mélyebb szinten `trap` által lekezelt hibaesemény lép fel, akkor az – beállítható módon – a magasabb szinteken is jelezhet hibát.

Nézzük azt az esetet, amikor van egy szkriptem, benne egy „fő” trap, egy függvény és abban is egy trap, a végén meg a függvény segítségével nullával osztok:

```
trap
{
        Write-Error "Külső trap"
}

function belső ($osztó)
{
        trap
        {
                Write-Error "Belső trap"
        }

        20/$osztó
}

belső 0
```

Nézzük, hogy mi történik, ha futtatom ezt:

```
[1] PS C:\powershell2\scripts> .\nestedtrap.ps1
belső : Belső trap
At C:\powershell2\scripts\nestedtrap.ps1:16 char:6
+ belső  <<<< 0
Attempted to divide by zero.
At C:\powershell2\scripts\nestedtrap.ps1:13 char:5
+     20/$ <<<< osztó
```

Látszik, hogy a belső trap éledt fel, és mivel nem mondtuk, hogy folytathatja, ezért ki is száll a futtatásból a program, megkapjuk még a PowerShell saját hibajelzését is.

Módosítsuk a belső trap-et, mondjuk neki, hogy folytathatja:

```
trap
{
        Write-Error "Külső trap"
}

function belső ($osztó)
{
        trap
```

```
        {
                Write-Error "Belső trap"
                continue
        }

        20/$osztó
}

belső 0
```

Nézzük ennek is a kimenetét:

```
[5] PS C:\powershell2\scripts> .\nestedtrap.ps1
belső : Belső trap
At C:\powershell2\scripts\nestedtrap.ps1:17 char:6
+ belső  <<<< 0
```

Itt ugye az történt, hogy a belső trap egy folytatással (`continue`) zárult le, így a külső környezet már erről a hibáról nem is értesült, azt hiszi, minden rendben.

Zárjuk le akkor a belső trap-et egy `break`-kel:

```
trap
{
        Write-Error "Külső trap"
}

function belső ($osztó)
{
        trap
        {
                Write-Error "Belső trap"
                break
        }

        20/$osztó
}

belső 0
```

Ennek kimenete:

```
[6] PS C:\powershell2\scripts> .\nestedtrap.ps1
belső : Belső trap
At C:\powershell2\scripts\nestedtrap.ps1:17 char:6
+ belső  <<<< 0
C:\powershell2\scripts\nestedtrap.ps1 : Külső trap
At line:1 char:16
+ .\nestedtrap.ps1 <<<<
Attempted to divide by zero.
At C:\powershell2\scripts\nestedtrap.ps1:14 char:5
+     20/$ <<<< osztó
```

Itt már mindkét szint trapje megjelenik, hiszen a belső trap rendhagyó módon, egy megtöréssel (`break`) ért véget, ami kívülről nézve is hibát jelez. Ezért a külső trap is felélesedik és az is jelzi a hibát.

Mindezek figyelembevételével el lehet dönteni, hogy hol érdemes, és a működés szempontjából hol kell trapet elhelyezni.

2.2.4.2 Dobom és elkapom

Gyakori hibakezelési feladat, hogy a függvényünk, szkriptünk használata során nem ad meg valaki valamilyen kötelező paramétert. Ennek kezelésére már láttuk a `throw` kulcsszót a *1.8 Függvények* fejezet *1.8.2.3 Hibajelzés* alfejezetében1.8.2.3, de ennek hatására megjelenő hibajelzés nem annyira szép, mert nem csak az általunk megadott szöveg kerül kiírásra, hanem egyéb szövegek is. Adódik az ötlet, hogy akkor kapjuk el egy `trap`-pel az általunk `throw`-val dobott hibajelzést, és szabjuk testre a `trap` kódjában a hibajelzést.

Készítettem egy újabb szkriptet az előzőek alapján:

```
trap
{
        "Hibajelenség: $_"
        "-------- Részletek --------------"
        $ .Exception.GetType().FullName
        $ .InvocationInfo
        "-------- Részletek vége ---------"
        continue
}

function dupla ($v = $(throw "Adj meg paramétert!"))
{
        $v*2
}

dupla
```

A `trap` része nem nagyon változott, és ott van mellette a korábban már látott duplázó függvényem, hibajelzéssel a paraméterhiány esetére. A szkript legvégén meghívom magát a dupla függvényt mindenféle paraméter nélkül. Mindennek ez lesz az eredménye:

```
[49] PS C:\powershell2\scripts> .\trap7.ps1
Hibajelenség: Adj meg paramétert!
-------- Részletek --------------
System.Management.Automation.RuntimeException


MyCommand        :
ScriptLineNumber : 11
OffsetInLine     : 29
ScriptName       : C:\powershell2\scripts\trap7.ps1
Line             : function dupla ($v = $(throw "Adj meg paramétert!"))
PositionMessage  :
```

```
                    At C:\powershell2\scripts\trap7.ps1:11 char:29
                    + function dupla ($v = $(throw  <<<< "Adj meg paramétert
                    !"))
InvocationName    : throw
PipelineLength    : 0
PipelinePosition : 0

-------- Részletek vége ---------
```

Tényleg működik a `throw` elkapása, csak az én általam megadott hibajelzés jelent meg, viszont elég általánosnak tűnik a hiba típusa:

`System.Management.Automation.RuntimeException.`

Ez nem túl specifikus, általános futási hiba. Viszont lehetünk specifikusabbak, adjunk a `throw`-val egy értelmesebb hibaobjektumot a `trap` számára, azaz ne csak egy egyszerű szöveget adjunk neki, hanem egy megfelelő `Exception` objektumot!

```
trap
{
        "Hibajelenség: $ "
        "-------- Részletek --------------"
        $_.Exception.GetType().FullName
        $_.InvocationInfo
        "-------- Részletek vége ---------"
        continue
}

function dupla ($v = $(throw `
        (New-Object System.ArgumentException `
                -arg "Adjál meg valamit, amit duplázni lehet!")))
{
        $v*2
}

dupla
```

En úgy gondoltam, hogy az ilyen jellegű hiba leírására a `System.ArgumentException` típus lesz a legjobb, így ennek egy objektumát hozom létre a `throw` paramétereként. Ráadásul ennek az objektumnak a konstruktora argumentumot is képes fogadni, az általam megadott hibaleírást szöveggel. Így már nagyon elegáns és hiba specifikus lesz a kimenete is:

```
[50] PS C:\powershell2\scripts> .\trap8.ps1
Hibajelenség: Adjál meg valamit, amit duplázni lehet!
-------- Részletek --------------
System.ArgumentException


MyCommand        :
ScriptLineNumber : 11
OffsetInLine     : 29
ScriptName       : C:\powershell2\scripts\trap8.ps1
```

```
Line              : function dupla ($v = $(throw `
PositionMessage   :
                    At C:\powershell2\scripts\trap8.ps1:11 char:29
                    + function dupla ($v = $(throw  <<<< `
InvocationName    : throw
PipelineLength    : 0
PipelinePosition : 0

-------- Részletek vége ---------
```

Így már specifikusabb `trap`-et is lehet készíteni, ami csak erre a hibatípusra éled fel.

2.2.5 Nem megszakító hibák kezelése függvényeinkben

A `trap` tárgyalásának elején már láthattuk, hogy hibakezelést csak megszakító hibákra tudunk készíteni, legfeljebb a `trap` végére tett `continue` kulcsszóval átalakítjuk a hibát nem megszakítóra. De vajon hogyan tudunk nem megszakító hibákat kezelni? Erre nincs külön kulcsszó, a többi PowerShelles nyelvi eszközzel kell ezt megoldani.

Az egyik ilyen helyzet, hogy a nem megszakító hibát nem akarom csak azért megszakítóvá alakítani, hogy aztán egy `continue`-val elnyomhassam a hibajelzést, hanem eleve nem akarok látni semmilyen hibainformációt a konzolon. Ehhez azt kell tudni, hogy külön outputja van a hibajelzéseknek, így lehet olyat is csinálni, hogy csak ezeket a hibajelzéseket küldjük át egy fájlba, vagy akár a semmibe. Nézzünk erre példát:

```
[43] PS C:\> Remove-Item nemlétezőfájl.txt 2> $null
```

Ez tehát a „`2>`” operátor, ami a 2-es számú outputot, a hibajelzések outputját irányítja a semmibe. (Amúgy „`1>`” nem létezik, csak a kettessel kezdődő változat.)

Ha pont ellentétes lenne a feladat, azaz valami egyedi, saját hibajelzést szeretnénk adni a „gyári” jelzés helyett, arra külön `write-...` kezdetű cmdletek állnak rendelkezésünkre. Ezekkel szintén nem a „normál” outputra, hanem erre a hibajelzések számára elkülönített outputra küldhetjük a jelzést.

```
[67] PS C:\powershell2\scripts> filter bulk-remove
>> {
>>     if (-not (Test-Path $_))
>>         { Write-Warning "Nincs ilyen file: $_!" }
>>     else
>>         { Remove-Item $  }
>> }
>>
```

A fenti `filter` kifejezéssel definiálok egy olyan fájltörlő függvényt, amely először ellenőrzi, hagy van-e ott ténylegesen fájl, amit megadott a felhasználó, és ha nincs, akkor nem csúnya hibajelzést ad, hanem csak egy sárga színű figyelmeztetést a `write-warning` cmdlet segítségével.

```
[68] PS C:\powershell2\scripts> "nincs.txt", "file.txt" | bulk-remove
WARNING: Nincs ilyen file: nincs.txt!
```

Ez azért is jó, mert ez nem kerül be a `$error` tömbbe, így nem „terheli” azt. Természetesen lehetne piros feliratú `write-error`-t is használni, ekkor olyan hibajelzést kapunk, ami bekerül az `$error` tömbbe.

Nem csak ilyen következményei vannak a `write-error` és `write-warning` használatának. Ha átállítjuk a következő automatikus változókat, akkor szabályozhatjuk függvényünk, szkriptünk futását is:

```
[79] PS C:\powershell2\scripts> $WarningPreference
Continue
[80] PS C:\powershell2\scripts> $ErrorActionPreference
Continue
```

Természetesen a `write-error` még profibbá tehető az `Exception` objektumok használatával:

```
filter bulk-remove
{
        if (-not (Test-Path $_))
                { Write-Error -Exception `
                        (New-Object System.IO.FileNotFoundException `
                                -arg "Nincs ilyen file!",$ ) }
        else
                {
                        Remove-Item $_
                }
}
```

És ennek kimenete hiba esetén:

```
[82] PS C:\powershell2\scripts> "semmi.txt" | bulk-remove
bulk-remove : Nincs ilyen file!
At line:1 char:25
+ "semmi.txt" | bulk-remove <<<<
[83] PS C:\powershell2\scripts> $error[0].exception.gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     FileNotFoundException                    System.IO.IOE...
```

A [83]-as sorban láthatjuk, hogy ez a hiba már „igazi” `FileNotFoundException` lett.

2.2.6 Hibakeresés

Akármennyire sok lehetőségünk van a hibák jelzésére (`write-warning`, `write-error` és `trap`), kezelésére mégsem minden esetben elég ez. Főleg akkor, ha a

szkriptünk a maga módján jól működik, csak éppen mi rossz programot írtunk. Ilyenkor van szükség a „dibággolásra”, azaz a hibakeresésre. Erre is lehetőséget biztosít a PowerShell, bár valószínű egy bizonyos bonyolultság felett már inkább használunk erre a célra valamilyen grafikus szkriptszerkesztőt, mint például a PowerGUI Script Editora vagy a PowerShell Plus.

2.2.6.1 Státusjelzés (write-verbose, write-debug)

Az egyik „bogártalanítási” módszer, hogy jól teletűzdeljük a programunkat kiíratási parancsokkal, amelyek segítségével a változók aktuális állapotát írhatjuk ki, illetve jelezhetjük, hogy éppen milyen ágon fut a programon. Erre használhatjuk akár a `write-host` cmdletet, csak az a baj ezzel, hogy a végleges, kijavított, telesztelt programból elég nehéz kiszedegetni ezeket. Ezért találtak ki másfajta kiíratási lehetőségeket is: a `write-verbose` és a `write-debug` cmdleteket.

Nézzünk erre példát. Van egy faktoriálist számoló szkriptem, amit jól teletűzdelek `write-verbose` helyzetjelentésekkel:

```
Write-Verbose "Eleje"
$a = 10; $result = 1
Write-Verbose "Közepe $a"
for($i=1; $i -le $a; $i++)
        {
                $result = $result * $i
                Write-Verbose "`$i = $i, `$result = $result"
        }
Write-Verbose "Vége"
$result
```

Nézzük, mi történik, ha futtatom:

```
[7] PS C:\powershell2\scripts> .\verbose.ps1
3628800
```

Nem sok mindent látunk ezen, mintha ott sem lennének a `write-verbose` parancsok. Merthogy a `write-verbose` hatását egy globális változóval, a `$VerbosePreference`-szel lehet ki- és bekapcsolni:

```
[9] PS C:\powershell2\scripts> $verbosepreference = "continue"
[10] PS C:\powershell2\scripts> .\verbose.ps1
VERBOSE: Eleje
VERBOSE: Közepe 10
VERBOSE: $i = 1, $result = 1
VERBOSE: $i = 2, $result = 2
VERBOSE: $i = 3, $result = 6
VERBOSE: $i = 4, $result = 24
VERBOSE: $i = 5, $result = 120
VERBOSE: $i = 6, $result = 720
VERBOSE: $i = 7, $result = 5040
VERBOSE: $i = 8, $result = 40320
VERBOSE: $i = 9, $result = 362880
```

```
VERBOSE: $i = 10, $result = 3628800
VERBOSE: Vége
3628800
```

Sajnos a könyvben nem látszik, de a `VERBOSE:` kimenetek szép sárgák, hasonlóan a „Warning” jelzésekhez. Ennek további értékei lehetnek – hasonlóan az `$ErrorActionPreference` változóhoz – `SilentlyContinue` (alapérték), `Inquire`, `Stop`.

Ugyanígy működik a `write-debug` cmdlet is, csak az ő hatását a `$DebugPreference` változó szabályozza.

Azaz van két, egymástól függetlenül használható kiíratő cmdletünk, amelyekkel a szkriptjeink futásának státusát írathatjuk ki a fejlesztés során, majd ha készen vagyunk és meggyőződtünk, hogy minden jól működik, akkor anélkül, hogy a szkriptből ki kellene szedegetni ezeket a státusjelzéseket, egy globális változó beállításával ezek hatását ki tudjuk kapcsolni.

2.2.6.2 Lépésenkénti végrehajtás és szigorú változókezelés (set-psdebug)

A másik hibaűző módszer, ha lépésenként hajtatjuk végre a szkriptünket és így jobban mg tudjuk figyelni, hogy merre jár. Azaz nem kell minden sor után egy `write-debug` sort beiktatni. Erre is lehetőséget biztosít a PowerShell a `Set-PSDebug` cmdlettel. Nézzük meg a szintaxisát:

```
[22] PS C:\powershell2\scripts> (get-help set-psdebug).syntax

Set-PSDebug [-trace <int>] [-step] [-strict] [<CommonParameters>]
Set-PSDebug [-off] [<CommonParameters>]
```

Az első változatban kapcsolhatjuk be a hibakeresési üzemmódot. A `-trace` paraméter lehetséges értékei:

- 0: kikapcsoljuk a nyomkövetést
- 1: csak a végrehajtás során az aktuális sor számát jelzi
- 2: a sorok száma mellett a változók értékadását is jelzi

Az előző szkriptet fogom használni, kicsit átalakítva:

```
$a = 3; $result = 1
for($i=1; $i -le $a; $i++)
        {
                $result = $result * $i
        }
$result
```

És akkor nézzük a `-Trace 1` opcióval a futását:

```
[29] PS C:\powershell2\scripts> Set-PSDebug -Trace 1
[30] PS C:\powershell2\scripts> .\psdebug.ps1
DEBUG:    1+ .\psdebug.ps1
DEBUG:    1+ $a = 3; $result = 1
DEBUG:    1+ $a = 3; $result = 1
DEBUG:    2+ for($i=1; $i -le $a; $i++)
DEBUG:    4+   $result = $result * $i
DEBUG:    4+   $result = $result * $i
DEBUG:    4+   $result = $result * $i
DEBUG:    6+ $result
6
```

Láthatjuk, hogy szépen kiírja az éppen végrehajtott sorok számát, meg az ottani parancssor tartalmát is. Nézzük a `-Trace 2` opció hatását:

```
[32] PS C:\powershell2\scripts> Set-PSDebug -Trace 2
[33] PS C:\powershell2\scripts> .\psdebug.ps1
DEBUG:    1+ .\psdebug.ps1
DEBUG:     ! CALL script 'psdebug.ps1'
DEBUG:    1+ $a = 3; $result = 1
DEBUG:     ! SET $a = '3'.
DEBUG:    1+ $a = 3; $result = 1
DEBUG:     ! SET $result = '1'.
DEBUG:    2+ for($i=1; $i -le $a; $i++)
DEBUG:     ! SET $i = '1'.
DEBUG:    4+   $result = $result * $i
DEBUG:     ! SET $result = '1'.
DEBUG:     ! SET $i = '2'.
DEBUG:    4+   $result = $result * $i
DEBUG:     ! SET $result = '2'.
DEBUG:     ! SET $i = '3'.
DEBUG:    4+   $result = $result * $i
DEBUG:     ! SET $result = '6'.
DEBUG:     ! SET $i = '4'.
DEBUG:    6+ $result
6
```

Láthatjuk, hogy az előzőek mellett még a szkripthívást is látjuk, és minden egyes változó értékadását is az új értékekkel együtt. Ha még a `-step` paramétert is használjuk, akkor minden sor végrehajtására külön rákérdez:

```
[37] PS C:\powershell2\scripts> Set-PSDebug -Trace 1 -step
[38] PS C:\powershell2\scripts> .\psdebug.ps1

Continue with this operation?
   1+ .\psdebug.ps1
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):y
DEBUG:    1+ .\psdebug.ps1

Continue with this operation?
   1+ $a = 3; $result = 1
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):y
```

```
DEBUG:    1+ $a = 3; $result = 1

Continue with this operation?
   1+ $a = 3; $result = 1
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):s
```

Ekkor választhatunk, hogy tovább lépünk egy sorral (`Yes`), folytatjuk a futtatást végig (`Yes to All`), megszakítjuk a futtatást (`No` és `No to All`). A `Suspend` „némán” állítja meg a futtatást, nem lesz hibajelzés a rendkívüli programmegszakításról.

A `-strict` kapcsoló használata után csak olyan változókra hivatkozhatunk, amelynek korábban adtunk kezdőértéket:

```
[41] PS C:\powershell2\scripts> Set-PSDebug -strict
[42] PS C:\powershell2\scripts> $c=1
[43] PS C:\powershell2\scripts> $c
1
[44] PS C:\powershell2\scripts> $d
The variable $d cannot be retrieved because it has not been set yet.
At line:1 char:2
+ $d <<<<
```

A fenti példában a `$c` értékadás után lekérdezhető volt, szemben a `$d`-vel, amire hibajelzést kaptunk, hiszen nem adtam neki kezdőértéket.

Ha már nincs szükségünk a `Set-PSDebug`-gal beállított hibafelderítő üzemmódokra, akkor az `-Off` kapcsolóval kapcsolhatjuk ki ezekez:

```
[45] PS C:\powershell2\scripts> Set-PSDebug -off
[46] PS C:\powershell2\scripts> $d
[47] PS C:\powershell2\scripts>
```

Itt látható, hogy már nem okozott hibajelzést az értékadás nélküli `$d`-re való hivatkozás.

2.2.6.3 Ássunk még mélyebbre (Trace-Command)

Ha még ez sem lenne elég, akkor még mélyebbre áshatunk a `Trace-Command` cmdlet segítségével. Igazából ez nem biztos, hogy a rendszergazdák mindennapos eszköze lesz, inkább azon fejlesztők tudnak profitálni használatából, akik cmdleteket fejlesztenek.

Nézzük a szintaxisát:

```
[5] PS I:\>(get-help trace-command).syntax

Trace-Command [-name] <string[]> [-expression] <scriptblock> [[-option] {<N
one> | <Constructor> | <Dispose> | <Finalizer> | <Method> | <Property> | <D
elegates> | <Events> | <Exception> | <Lock> | <Error> | <Errors> | <Warning
> | <Verbose> | <WriteLine> | <Data> | <Scope> | <ExecutionFlow> | <Assert>
 | <All>}] [-filePath <string>] [-debugger] [-pSHost] [-listenerOption {<No
ne> | <LogicalOperationStack> | <DateTime> | <Timestamp> | <ProcessId> | <T
```

```
hreadId> | <Callstack>}] [-inputObject <psobject>] [-force] [<CommonParamet
ers>]
Trace-Command [-name] <string[]> [-command] <string> [[-option] {<None> | <
Constructor> | <Dispose> | <Finalizer> | <Method> | <Property> | <Delegates
> | <Events> | <Exception> | <Lock> | <Error> | <Errors> | <Warning> | <Ver
bose> | <WriteLine> | <Data> | <Scope> | <ExecutionFlow> | <Assert> | <All>
}] [-filePath <string>] [-debugger] [-pSHost] [-listenerOption {<None> | <L
ogicalOperationStack> | <DateTime> | <Timestamp> | <ProcessId> | <ThreadId>
 | <Callstack>}] [-inputObject <psobject>] [-argumentList <Object[]>] [-for
ce] [<CommonParameters>]
```

Huh, nem túl egyszerű! Az látható, hogy alapvetően kétfajta dolgot lehet vele vizsgálni: parancsokat és kifejezéseket. Alapvetően mi (rendszergazdák) két esetben használhatjuk: az első eset, amikor a parancsok, kifejezések meghívásának körülményeit szeretnénk tisztázni, a másik, amikor a paraméterek átadásának körülményeit. Nézzünk példát az elsőre:

```
[9] PS I:\>Trace-Command commanddiscovery {dir i:\} -pshost
DEBUG: CommandDiscovery Information: 0 : Looking up command: dir
DEBUG: CommandDiscovery Information: 0 : Alias found: dir  Get-ChildItem
DEBUG: CommandDiscovery Information: 0 : Attempting to resolve function or
filter: Get-ChildItem
DEBUG: CommandDiscovery Information: 0 : Cmdlet found: Get-ChildItem
Microsoft.PowerShell.Commands.GetChildItemCommand,
Microsoft.PowerShell.Commands.Management, Version=1.0.0.0, Culture=neutral,
 PublicKeyToken=31bf3856ad364e35


    Directory: Microsoft.PowerShell.Core\FileSystem::I:\


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----     2006. 04. 24.     10:00            2272
d----     2007. 01. 18.     17:40            asi
d----     2005. 07. 07.      3:12            books
```

Itt a `trace-command` cmdletet a „`commanddiscovery`” névvel hívtam meg. Mögötte ott van az a kifejezés szkriptblokk formában, amit elemeztetni szeretnék, majd megadom, hogy az eredményt a konzolra írja ki.

Az elemzés felderíti, hogy a PowerShell a „`dir`” álnevet hogyan párosítja a `Get-ChildItem` cmdlettel, megnézi, hogy nem tréfáltuk-e meg, és nem készítettünk-e ilyen névvel valamilyen függvényt vagy szűrőt (filter). Ha nem, akkor kiírja, hogy melyik PSSnapIn-ben van ez a cmdlet definiálva és utána már csak a futásának eredményét látjuk. A `trace-command` ezzel az üzemmódjával tehát akkor jön jól nekünk, ha mindenféle aliast, függvényt, szkriptet és egyebeket definiálunk, gyanítjuk, hogy esetleg többször is ugyanolyan névvel, és amikor futtatunk valamit, nem értjük, hogy mi is fut le valójában.

Nézzük a paraméterátadás felderítését:

```
[7] PS I:\>Trace-Command parameterbinding {get-process notepad} -pshost
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args
[Get-Process]
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args
[Get-Process]
DEBUG: ParameterBinding Information: 0 :     BIND arg [notepad] to
parameter [Name]
DEBUG: ParameterBinding Information: 0 :         Binding collection
parameter Name: argument type [String], parameter type [System.String[]],
collection type Array, element type [System.String], no coerceElementType
DEBUG: ParameterBinding Information: 0 :         Creating array with
element type [System.String] and 1 elements
DEBUG: ParameterBinding Information: 0 :         Argument type String is
not IList, treating this as scalar
DEBUG: ParameterBinding Information: 0 :         Adding scalar element of
type String to array position 0
DEBUG: ParameterBinding Information: 0 :         Executing VALIDATION
metadata: [System.Management.Automation.ValidateNotNullOrEmptyAttribute]
DEBUG: ParameterBinding Information: 0 :         BIND arg [System.String[]]
 to param [Name] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on
cmdlet [Get-Process]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : CALLING ProcessRecord
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     38       2     1280       3508    30     0,89   5372 notepad
```

Itt a `trace-command` név paramétere „`parameterbinding`”, mögötte megint a vizsgálandó kifejezés, majd az, hogy a konzolra küldje az eredményt. Látjuk, hogy a „notepad” karaktersorozatot a „`Name`” paraméterhez fűzte. Ez nekünk természetesnek tűnik, de azért ez nem ilyen triviális. Nézzük, vajon mit is vár a `get-process` „`Name`” paraméter gyanánt:

```
[10] PS I:\>(get-help get-process).syntax

Get-Process [[-name] <string[]>] [<CommonParameters>]
```

Láthatjuk a súgóban, hogy egy sztringtömböt vár a `get-process`. Az elemzés további része annak lépéseit mutatja, hogy hogyan csinál a PowerShell a sima sztringből sztringtömböt, és hogy ezzel a paraméterrel sikeresen vette a cmdlet mindhárom feldolgozási fázisát is, a begin, a process és az end részt is.

A másik eset, hogy nem értjük, hogy egy cmdletnek vagy szkriptnek átadott paraméter miért nem jó, miért nem úgy értelmezi a kifejezés, mint ahogy mi szeretnénk. Ez utóbbi demonstrálására nézzünk egy példát, amelyben annak nézünk utána, hogy ha egy gyűjteményt adunk át a `get-member`-nek, akkor az vajon miért a gyűjtemény elemeinek tagjellemzőit adja vissza, miért nem a gyűjteményét:

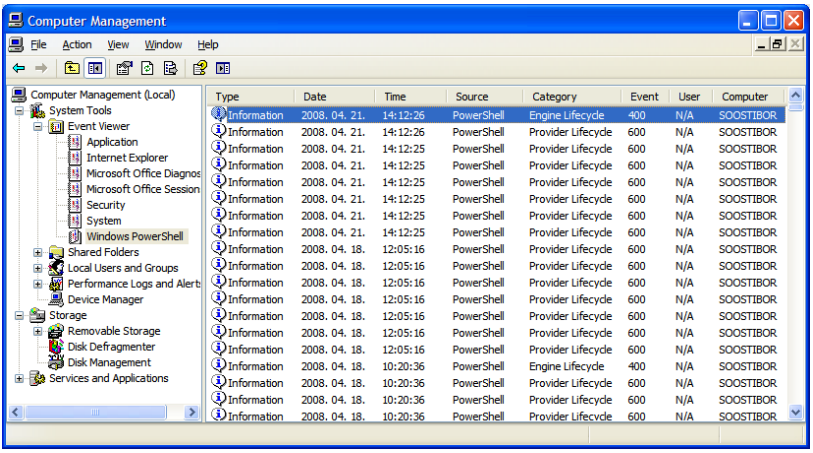
```
[15] PS I:\>Trace-Command parameterbinding {1, "kettő" | gm} -pshost
```

```
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args
[Get-Member]
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args
[Get-Member]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on
cmdlet [Get-Member]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to
parameters: [Get-Member]
DEBUG: ParameterBinding Information: 0 :     PIPELINE object TYPE =
[System.Int32]
DEBUG: ParameterBinding Information: 0 :     RESTORING pipeline parameter's
 original values
DEBUG: ParameterBinding Information: 0 :     Parameter [InputObject]
PIPELINE INPUT ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg [1] to parameter
[InputObject]
DEBUG: ParameterBinding Information: 0 :         BIND arg [1] to param
[InputObject] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on
cmdlet [Get-Member]
DEBUG: ParameterBinding Information: 0 : CALLING ProcessRecord
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to
parameters: [Get-Member]
DEBUG: ParameterBinding Information: 0 :     PIPELINE object TYPE =
[System.String]
DEBUG: ParameterBinding Information: 0 :     RESTORING pipeline parameter's
 original values
DEBUG: ParameterBinding Information: 0 :     Parameter [InputObject]
PIPELINE INPUT ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 :     BIND arg [kettő] to parameter
[InputObject]
DEBUG: ParameterBinding Information: 0 :         BIND arg [kettő] to param
[InputObject] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on
cmdlet [Get-Member]
DEBUG: ParameterBinding Information: 0 : CALLING ProcessRecord
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing
...
```

Láthatjuk, hogy a `get-member` hogyan dolgozza fel a csővezetéken érkező különböző objektumokat, hogyan emelődnek át az argumentumok `InputObject`-té. Elsőként a parancsértelmező megnézi, hogy vannak-e név szerinti paraméterek a parancssorban, majd ellenőrzi a pozíció alapján történő paraméterátadást. Ezután ellenőrzi, hogy a kötelező paraméterek meg vannak-e adva, a `get-member`-nek nincs ilyenje. Mindezek után veszi észre, hogy csővezetékben adom át a paramétereket, így annak kezelése történik meg, először az „`1`” lesz behelyettesítve „`InputObject`” paraméterként, majd a „`kettő`”.

2.2.7 A PowerShell eseménynaplója

A PowerShell számára külön eseménynapló nyílik telepítése után. Ide nem az egyes parancsok, szkriptek futásának eredménye kerül be, hanem a konzol és a providerek betöltésének státusinformációi:



48. ábra A PowerShell eseménynaplója

Azaz itt akkor látunk figyelmeztető vagy hibabejegyzéseket, ha valami nagyon súlyos hiba lép fel a PowerShell környezetben.

2.3 Fájlkezelés

Már többször foglalkoztunk fájlokkal, használtuk a `get-childitem`, `get-item`, stb. cmdleteket, egy kicsit ássunk ebben a témában mélyebbre.

2.3.1 Fájl és könyvtár létrehozása (new-item), ellenőrzése (test-path)

Új fájlt létrehozni a `new-item` cmdlettel lehet:

```
[8] PS C:\scripts> new-item -Path . -Name szöveg.txt -type file -Value "Ez e
gy szöveg"


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\scripts


Mode           LastWriteTime       Length Name
----           -------------       ------ ----
-a---   2008.04.22.    21:31           14 szöveg.txt
```

Természeten nem csak a fájlt, hanem könyvtárt is készíthetünk, talán ez gyakoribb:

```
[9] PS C:\scripts> New-Item -Path . -Name "Alkönyvtár" -type Directory


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\scripts


Mode           LastWriteTime       Length Name
----           -------------       ------ ----
d----   2008.04.22.    21:33        <DIR> Alkönyvtár
```

Könyvtár létrehozása előtt érdemes megnézni, hogy létezik-e esetleg már a könyvtár. Erre a célra a `Test-Path` cmdlet áll rendelkezésünkre:

```
[12] PS C:\> Test-Path C:\scripts
True
```

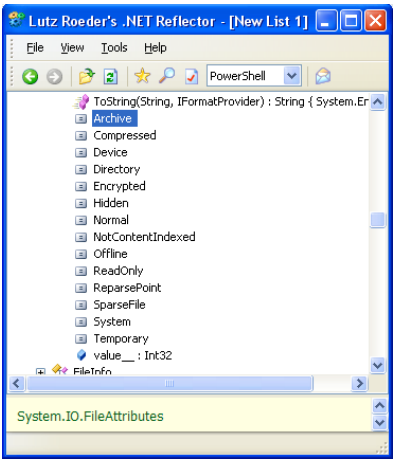
Feladat lehet még a fájlok attribútumainak beállítása. Ezeket egyszerűen állíthatjuk be a fájlobjektumok tulajdonságainak módosításával:

```
[13] PS C:\> (Get-Item C:\scripts\alice.txt).attributes = "Archive, hidden"
```

De vajon milyen lehetőségek közül válogathatok az attribútumoknál? Ebben is a Reflector segít, de előtte meg kell nézni, hogy valójában milyen típusú adat a fájlok attribútuma:

```
[14] PS C:\> (Get-Item C:\scripts\alice.txt).attributes.gettype().fullname
System.IO.FileAttributes
```

Nem meglepő módon ez `System.IO.FileAttributes` típusú, rákeresve erre a Reflectorban ezt láthatjuk:



49. ábra A fájlok attribútumai a Reflectorban

A PowerShellben a típuskonverzió annyira okos, hogy a [13]-as sorban sztringként megadott fájl-attribútumokat (vesszővel elválasztott két attribútum egy sztringben) is képes volt `System.IO.FileAttributes` típusúvá konvertálni.

2.3.2 Rejtett fájlok

Nézzük meg a C: gyökerének elemeit (fájlok, könyvtárak), amelyek „p” betűvel kezdődnek:

```
[12] PS C:\> Get-ChildItem p*


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\


Mode            LastWriteTime       Length Name
----            -------------       ------ ----
d----    2008.03.29.    20:47         <DIR> powershell2
d-r--    2008.04.16.    23:03         <DIR> Program Files
d-r--    2008.04.17.    22:48         <DIR> Program Files (x86)
```

De hol van például a pagefile.sys? Alaphelyzetben a `Get-ChildItem` a rejtett fájlokról nem vesz tudomást. Van neki egy `-force` kapcsolója, amellyel a rejtett fájlokat is láttathatjuk:

```
[30] PS C:\> Get-ChildItem p* -force


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\


Mode           LastWriteTime       Length Name
----           -------------       ------ ----
d----   2008.03.29.    20:47        <DIR> powershell2
d-r--   2008.04.16.    23:03        <DIR> Program Files
d-r--   2008.04.17.    22:48        <DIR> Program Files (x86)
-a-hs   2008.04.18.    20:45   2145386496 pagefile.sys
```

Ugyanezt a kapcsolót kell használni a `get-item` cmdlet esetében is.

2.3.3 Szövegfájlok feldolgozása (Get-Content, Select-String)

Szöveges fájlok olvasására a `Get-Content` cmdlet szolgál, amint azt már korábban is láthattuk. Paraméterként egy, vagy több fájl nevét kell megadnunk, a kimenetbe pedig a fájlok tartalma kerül soronként egy-egy karakterlánc képében.

? **Feladat:** Keressük meg a Windows mappában azokat a naplófájlokat, amelyekben szerepel az „error” kifejezés. A listában szerepeljen a fájl neve, a megtalált sor szövege, és hogy az hányadik sor az adott fájlon belül!

A feladatot két különböző módon is meg fogjuk oldani, bár az első próbálkozás közel sem ad majd tökéletes eredményt. Ha megvizsgáljuk a csőben áramló adatok természetét, akkor nyilvánvalóvá válik, hogy ezzel a módszerrel nem is lehetséges a tökéletes megoldás. Kezdjünk hozzá! Az első megoldásban tulajdonképpen semmi különleges nincsen, a `Get-Content` sorban megnyitja valamennyi logfájlt, az eredményt odaadjuk a `Select-Stringnek`, aki kiválogatja a megfelelő sorokat. Kell még egy kis formázgatás és készen is vagyunk.

```
PS C:\> Get-Content $env:windir\*.log | select-string -pattern "error" | f
ormat-list filename,line,linenumber


Filename   : InputStream
Line       : COM+[7:32:26]: Warning: error 0x800704cf in IsWin2001Primary
             DomainController
LineNumber : 260
...
```

Sajnos azonban ezzel a módszerrel útközben olyan információt is eldobáltunk, amire feltétlenül szükségünk lenne a helyes eredmény előállításához. Vizsgáljuk meg, milyen kimenetet produkál a fenti esetben a `Get-Content`! A fájlok tartalma jelenik meg a

kimeneten, soronként egy-egy karakterlánc képében, de mindenféle strukturáltság nélkül. A `Select-String` már semmiféle információt nem kap arról, hogy melyik karakterlánc melyik fájlhoz tartozott eredetileg, és még kevésbé tudhatja azt, hogy hányadik sor volt az a fájlban.

Mi került akkor a `Select-String` kimenetébe? A fájlnév helyén mindenhol az `InputStream` kifejezés található, a `LineNumber` pedig azt mutatja, hogy az adott karakterlánc hányadik volt a teljes bemenetben, vagyis az egymás mögé illesztett napló-fájlokban. Hát ez nem az igazi!

A `Select-String` azonban bemenetként nem csak karakterláncokat, hanem közvetlenül szöveges fájlokat is fogadhat, a következő megoldásban ezt a tulajdonságot fogjuk felhasználni. Ebben az esetben a `Get-ChildItem` cmdlettől nem a fájlok tartalmát, hanem csak egy `FileInfo` objektumokból álló gyűjteményt kap a `Select-String`, a fájlok tartalmát már ő maga fogja kiolvasni.

```
PS C:\> Get-ChildItem $env:windir\*.log | select-string -pattern "error" -
list | format-list filename,line,linenumber


Filename   : comsetup.log
Line       : COM+[7:32:26]: Warning: error 0x800704cf in IsWin2001Primary
             DomainController
LineNumber : 220
...
```

Ebben az esetben minden szükséges információ rendelkezésre áll, és helyesen kerül be a `Select-String` kimenetébe, így helyesen jelenhet meg a táblázatban is. A `-list` paraméter arra utasítja a cmdletet, hogy csak az első találatig olvasson minden egyes fájlt, így a kapott lista már nem lesz olyan hosszú, mint korábban.

? **Feladat:** Készítsünk statisztikát az about_signing.help.txt fájl tartalmáról!

Kezdjük a legegyszerűbb, már ismert módszerrel, használjuk a `Measure-Object` cmdletet!

```
PS C:\> Get-Content $pshome\about signing.help.txt | Measure-Object -line -
word -character

             Lines             Words         Characters Property
             -----             -----         ---------- --------
               219              1571              11266
```

Eddig rendben is van, de mit kell tennünk, ha arra is kíváncsiak vagyunk, hogy egy adott szó hányszor szerepel a fájlban, vagy például arra, hogy melyik szó fordul elő benne a legtöbbször. A `Get-Content` soronként tördelt kimenetet ad, először is tördeljük ezt tovább szavakká:

```
PS C:\> [string[]]$szavak = Get-Content $pshome\about_signing.help.txt |
foreach-object {$_.Split(" `t",[stringsplitoptions]::RemoveEmptyEntries)}
```

Azt hiszem, a fenti parancs azért igényelhet némi magyarázatot. Először is készítünk egy tömböt, ami karakterlánc változókat tud majd fogadni, ebbe kell majd beledobálni a szöveg szavait. A `Get-Content` szállítja a szöveget soronként, a `Foreach-Object` pedig minden egyes sort szavakká tördel a `String` osztály `Split()` függvényének használatával. A `Split()` minden sort a szavaiból álló karakterlánc-tömb képében ad vissza, ezeket adjuk hozzá egyesével a `$szavak` tömbhöz. A `Split()` első paramétereként azokat a karaktereket kell megadnunk, amelyek mentén tördelni szeretnénk a szöveget, a szavakat egymástól a megadott szóköz és tabulátor karakterek választhatják el. A második paraméterrel arra utasítjuk a `Split()` függvényt, hogy a tördelés közben talált üres karakterláncokat és hasonló haszontalanságokat eldobja (ezt is például a Reflector program használatával lehet felderíteni). Nézzük mi lett ebből!

```
PS C:\> $szavak.length
1571
```

Remek! A szavak száma pontosan megegyezik azzal, amit a `Measure-Object` adott vissza, valószínűleg minden rendben van. A statisztika most már nem gond, a `Group-Object` csoportosít, a `Sort-Object` pedig az előfordulások száma szerint sorba rendez:

```
PS C:\> $szavak | group-object | sort-object count -descending

Count Name                      Group
----- ----                      -----
  110 the                       {the, the, the, the...}
   46 a                         {a, a, a, a...}
   44 You                       {You, you, you, you...}
   42 to                        {to, to, To, to...}
   31 and                       {and, and, and, and...}
   27 certificate               {certificate, certificate, certificate,...
   25 PowerShell                {PowerShell, PowerShell, PowerShell, Po...
...
```

Nem probléma az sem, ha egy adott szó előfordulásainak számára vagyunk kíváncsiak, a sorba rendezés helyett egyszerűen a csoportosított listából ki kell választanunk a megfelelő sort. A „scripts” szó előfordulásainak számát például a következő parancs írja ki:

```
PS C:\> $szavak | group-object | where-object {$_.Name -eq "scripts"}

Count Name                      Group
----- ----                      -----
   23 scripts                   {scripts, scripts, scripts, Scripts...}
```

Az egyszerűség☺ kedvéért egyetlen sorba is belesűríthetjük a feladat teljes megoldását, ebben az esetben nincs szükség a változóra csak a következő parancsot kell begépelnünk:

```
PS C:\> (Get-Content $pshome\about signing.help.txt |
foreach-object {$_.Split(" `t",[stringsplitoptions]::RemoveEmptyEntries)})
| group-object | sort-object count -descending
```

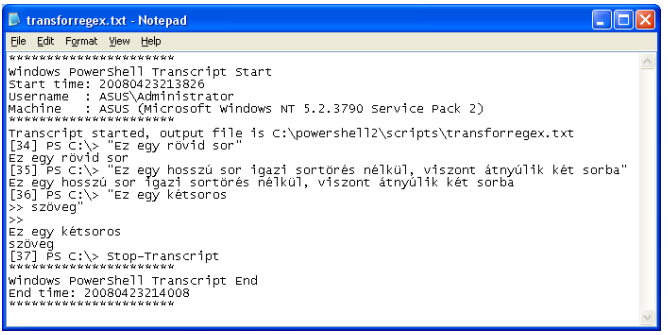
2.3.4 Sortörés kezelése szövegfájlokban

Gyakran előfordulnak olyan szöveges fájlok, amelyekben az információblokkok nem korlátozódnak egy-egy sorra, hanem átnyúlnak sorvégeken. Ez speciális odafigyelést és feldolgozási módot igényel, hiszen – ahogy láttuk – például a `get-content` is soronként dolgozza fel a szöveges állományokat.

Példaként nézzünk egy olyan szöveges állományt, amelyet maga a PowerShell hoz létre a `start-transcript` cmdlet segítségével:

```
[33] PS C:\> Start-Transcript C:\powershell2\scripts\transforregex.txt
Transcript started, output file is C:\powershell2\scripts\transforregex.txt
[34] PS C:\> "Ez egy rövid sor"
Ez egy rövid sor
[35] PS C:\> "Ez egy hosszú sor igazi sortörés nélkül, viszont átnyúlik két
sorba"
Ez egy hosszú sor igazi sortörés nélkül, viszont átnyúlik két sorba
[36] PS C:\> "Ez egy kétsoros
>> szöveg"
>>
Ez egy kétsoros
szöveg
[37] PS C:\> Stop-Transcript
Transcript stopped, output file is C:\powershell2\scripts\transforregex.txt
```

A fenti példában indított transcript tartalmaz mindenféle hosszúságú sort, mesterségesen nyitott új sort alprompttal. Az elkészült fájl a következőképpen néz ki notepaddal nézve:



50. ábra Transcript fájl

Láthatjuk, hogy csak ott tett a fájlba sortörést, ahol tényleg Entert ütöttünk. Olvassuk be ezt a fájlt:

```
[41] PS C:\> $text = get-content C:\powershell2\scripts\transforregex.txt
```

Az így kapott `$text` változó egy sztringtömböt eredményez, amelynek egyes elemei a fájl sorai:

```
[42] PS C:\> $text[8]
Ez egy rövid sor
```

Hogy lehetne ebből egy olyan sztringet készíteni, amely tartalmazza az összes sort? Szerencsére a .NET Framework itt is a segítségünkre siet:

```
[43] PS C:\> $text = [string]::join("`r`n",$text)
```

A `[string]` osztálynak tehát van egy `join` statikus metódusa, amellyel ilyen összefűzéseket lehet végezni. Paraméterként át kell adni egy olyan karaktert (vagy karaktersorozatot), amelyet az összefűzések helyére beilleszt, meg az összefűzendő szöveget. Hogy az eredeti információtartalmat megőrizzük, én egy „kocsivissza-újsor” kombinációval fűztem össze a szövegem darabjait, merthogy a Windowsban ez a „hivatalos” sortörés.

Másik lehetőség az, hogy eleve a beolvasást más módszerrel végezzük, szintén a .NET keretrendszer segítségével, a `File` objektumtípus `ReadAllText` statikus metódusának segítségével:

```
[44] PS C:\> $text2 = [System.IO.File]::ReadAllText("c:\powershell2\scripts\
transforregex.txt")
```

Akármelyik módszert is választjuk, ugyanolyan eredményt kapunk, így immár egyszerűbben tudunk akár soron átnyúló regex kifejezéssel keresni.

2.3.5 Fájl hozzáférési listája (get-acl, set-acl)

A fájlrendszer és a registry objektumainál kiolvasható a hozzáférési lista a `get-acl` cmdlet segítségével:

```
[87] PS C:\> get-acl C:\powershell2\scripts\1.txt | fl


Path   : Microsoft.PowerShell.Core\FileSystem::C:\powershell2\scripts\1.txt
Owner  : ASUS\Administrator
Group  : ASUS\None
Access : BUILTIN\Administrators Allow  FullControl
         NT AUTHORITY\SYSTEM Allow  FullControl
         ASUS\Administrator Allow  FullControl
         BUILTIN\Users Allow  ReadAndExecute, Synchronize
```

```
Audit  :
Sddl   : O:LAG:S-1-5-21-2919093906-1695458891-47906081-513D:(A;ID;FA;;;BA)(
         A;ID;FA;;;SY)(A;ID;FA;;;LA)(A;ID;0x1200a9;;;BU)
```

Az így visszakapott objektum egy `FileSecurity` típusú objektum, melynek a fent látható tulajdonságai közül az `Sddl` elég rémisztőnek néz ki, de szerencsére nem muszáj azzal foglalkozni, emberi fogyasztásra jobban alkalmas hozzáférési szabályok segítségével is lehet beállítani a hozzáférést a fájlokhoz, könyvtárakhoz. A hozzáférési lehetőségeket a következő táblázat tartalmazza:

Hozzáférési jogok (FileSystemRights)	
ListDirectory	WriteAttributes
ReadData	Write
WriteData	Delete
CreateFiles	ReadPermissions
CreateDirectories	Read
AppendData	ReadAndExecute
ReadExtendedAttributes	Modify
WriteExtendedAttributes	ChangePermissions
Traverse	TakeOwnership
ExecuteFile	Synchronize
DeleteSubdirectoriesAndFilesReadAttributes	FullControl

Ezek közül lehet összerakni a kívánt hozzáférési lehetőségeket a következő módon:

```
[17] PS C:\> $Acl = Get-Acl C:\scripts
[18] PS C:\> $entry = New-Object System.Security.AccessControl.FileSystemAcc
essRule("Szkriptelők","Read","Allow")
[19] PS C:\> $entry


FileSystemRights  : Read, Synchronize
AccessControlType : Allow
IdentityReference : Szkriptelők
IsInherited       : False
InheritanceFlags  : None
PropagationFlags  : None

[20] PS C:\> $acl.AddAccessRule($entry)
[21] PS C:\> set-acl C:\scripts $acl
[22] PS C:\> (Get-Acl C:\scripts).Access

FileSystemRights  : Read, Synchronize
AccessControlType : Allow
IdentityReference : ASUS\Szkriptelők
IsInherited       : False
InheritanceFlags  : None
PropagationFlags  : None
...
```

A fenti példában a c:\scripts könyvtárhoz szeretném adni a Szkriptelők csoportot olvasási jogosultsággal. Ehhez a [17]-es sorban kiolvasom a meglevő hozzáférési listát, a [18]-as sorban definiálom az új hozzáférési bejegyzést. Ezt ellenőrzésképpen kiíratom a [19]-es sorban. A [20]-as sorban hozzáadom ezt a bejegyzést a hozzáférési listához. Mivel ezt egyelőre csak a memóriában tárolt `$acl` objektum tartalmazza, ezért ezt ki is kell írni a könyvtárobjektumra, amit a [21]-es sorban teszek meg a `set-acl` cmdlettel. Végezetül, a [22]-es sorban ellenőrzésképpen kiolvasom az új hozzáférési listát, amelyben ott szerepel az imént hozzáadott bejegyzés.

Megjegyzés:

A registry elemeinek hozzáférési jogosultságait hasonló módon, de a `RegistryAccessRule` osztály objektumainak segítségével állíthatjuk be.

2.3.5.1 Fájlok tulajdonosai

Az előzőekben láttuk, hogy a `get-acl` kimentében a fájl vagy könyvtár tulajdonosa is kiolvasható. Nézzünk ennek felhasználására egy kis szkriptet, mely segítségével a tulajdonosok szerint szortírozom szét a fájlokat:

```
Set-Location C:\fájlok

Get-ChildItem |
Where-Object {-not $_.PSIsContainer} |
        ForEach-Object {
                $d = (Get-Acl $_).Owner.Split("\")[1]
                if(-not (Test-Path ((get-location).path + '\' + $d)))
                {
                        new-item -path (get-location).path -name $d `
                                -type directory | Out-Null
                }
                Move-Item -path $_.pspath `
                        -destination ((get-location).path + '\' + $d +'\')
}
```

Az elején beállítom az aktuális könyvtárat, majd kilistáztatom az összes fájlját és alkönyvtárát. Mivel nekem csak a fájlok kellenek, ezért a `where-object`-tel kiszűröm a `PSIsContainer` típusú objektumokat.

Az így megmaradt objektumokon egy `foreach-object` ciklussal végigszaladok. Képzem egy `$d` változóba a fájl tulajdonosának a nevét. Itt egy kis trükközésre van szükség, hiszen a felhasználó neve *tartomány\felhasználónév* vagy *gépnév\felhasználónév* formátumú. Nekem csak a felhasználónév kell, így `split()`-tel kettétöröm és veszem a 2. elemet ([1]-es indexű), ami a felhasználói név.

Ezután megvizsgálom a `Test-Path` cmdlettel, hogy van-e már a névnek megfelelő alkönyvtár. Ha nincs, akkor a `new-item` cmdlettel létrehozom. Mire a `move-item`

cmdlethez érünk, addigra már biztos van a felhasználónévnek megfelelő alkönyvtár, így át tudom mozgatni oda a fájlt.

2.3.6 Ideiglenes fájlok létrehozása

Gyakran előfordul nagyobb adatfeldolgozással járó feladatoknál, hogy ki kell írni az átmeneti adatokat ideiglenes fájlokba, hiszen a rendelkezésre álló memória nem biztos, hogy elegendő.

A .NET egyik osztálya, a `System.IO.Path` ebben is segítségünkre van. A `GetTempFileName()` statikus metódusa a `temp` környezeti változó által meghatározott könyvtárba létrehoz egy üres fájlt, mely neve „tmp” karaktersorozattal kezdődik, majd jön egy sorszám, és a kiterjesztése „tmp”:

```
[95] PS C:\> [System.IO.Path]::GetTempFileName()
C:\Documents and Settings\Administrator\Local Settings\Temp\tmp16.tmp
```

Érdekes, hogy ezzel a kifejezéssel nem csak megkapjuk a lehetséges következő ideiglenes fájl nevét (mint ahogy a metódus neve sugallná), hanem létre is hozza az ideiglenes fájlt:

```
[104] PS C:\Documents and Settings\Administrator\Local Settings\Temp> dir tm
p*.tmp


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Documents and Setti
    ngs\Administrator\Local Settings\Temp


Mode            LastWriteTime       Length Name
----            -------------       ------ ----
-a---   2008.04.20.    11:46            0 tmp15.tmp
-a---   2008.04.24.    23:36            0 tmp16.tmp
```

Az így létrehozott üres fájlba átirányíthatjuk a PowerShell kifejezéseink kimenetét, és felhasználhatjuk majd azt későbbi feldolgozásra. Ha már nincs szükségünk már az ideiglenes fájlunkra, akkor ne feledkezzünk meg törlésükről.

2.3.7 XML fájlok kezelése

Korábban láttuk, hogy a PowerShell képes XML formátumban exportálni objektumait. Nézzük, hogy hogyan lehet egyéb forrásból származó XML állományokkal dolgozni. Példámban egy nagyon egyszerű Access adatbázist exportáltam XML formátumba. Vigyázni kell, hogy megfelelő kódolású legyen az XML fájl, mert az ékezetes betűket UTF-8 kódolással nem fogja tudni értelmezni a PowerShell.

Az XML fájl tartalmát a következő kifejezéssel tudjuk változóba tölteni:

```
[9] PS C:\> $xml = [xml] (get-content c:\powershell2\munka\emberek.xml)
```

Ha megnézzük ennek az `$xml` változónak a tartalmát, akkor már nem a fájlban tárolt karaktersorozatot kapjuk vissza, hanem az XML adatformátumnak megfelelő adattartalmakat, és ebből a *1.10.11 Kiírás fájlba (Out-File, Export-)* fejezetben látottaknak megfelelően lehet az adatokat kiolvasni:

```
[22] PS C:\> $xml

xml                                root
---                                ----
                                   root
[23] PS C:\> $xml.root

xsd                od              schema             dataroot
---                --              ------             --------
http://www.w3.o... urn:schemas-mic... schema           dataroot


[24] PS C:\> $xml.root.dataroot

xsi                generated          Emberek            Városok
---                ---------          -------            -------
http://www.w3.o... 2008-08-10T08:3... {1, 2, 3}         {1, 2}


[25] PS C:\> $xml.root.dataroot.emberek

ID                 Név                Mellék             VárosID
--                 ---                ------             -------
1                  Soós Tibor         1234               1
2                  Fájdalom Csilla    1230               1
3                  Beléd Márton       1299               2


[26] PS C:\> $xml.root.dataroot.városok

ID                                 Név
--                                 ---
1                                  Budapest
2                                  Debrecen
```

Vajon hogyan lehetne a „nyers” XML állománnyá visszaalakítani az `$xml` változónkat? Ebben az `[XML]` adattípus `Save` metódusa segíthet. Alapvetően ez a mentés fájlba irányulna, egy elérési utat vár paraméterként. Ha a képernyőre akarjuk kiíratni, akkor át kell verni, a mentés helyének a konzolt kell megadni a következő módon:

```
[26] PS C:\> $xml.save([console]::out)
<?xml version="1.0" encoding="ibm852"?>
<root xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:od="urn:schemas-mic
rosoft-com:officedata">
  <xsd:schema>
    <xsd:element name="dataroot">
```

```
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="Emberek" minOccurs="0" maxOccurs="unbounded" />
...
```

2.3.8 Megosztások és webmappák elérése

A megosztások elérése nagyon egyszerű a PowerShellből: egyszerűen a UNC névvel kell hivatkozni a megosztott könyvtárakra és az ott található fájlokra:

```
[2] PS C:\> Get-ChildItem \\asus\powershell


    Directory: Microsoft.PowerShell.Core\FileSystem::\\asus\powershell


Mode           LastWriteTime       Length Name
----           -------------       ------ ----
d----   2008.05.04.    12:55        <DIR> cd
d----   2008.07.15.    20:01        <DIR> Copy of scripts
d----   2008.02.28.    21:14        <DIR> demo
```

Sőt! A TAB-kiegészítés is működik! Valamint az ebben a fejezetben leírt összes cmdlet is működik az UNC nevekkel is, például:

```
[4] PS C:\> get-acl \\asus\powershell\munka\tömbök.txt


    Directory: Microsoft.PowerShell.Core\FileSystem::\\asus\powershell\munk
    a


Path                      Owner                   Access
----                      -----                   ------
tömbök.txt                ASUS\Administrator      BUILTIN\Administrator...
```

Megosztások létrehozása már nem ilyen egyszerű, de ehhez is van megoldás, melyet a *2.6.8 WMI objektumok metódusainak meghívása* fejezetben mutatok egy WMI kifejezés segítségével.

Mindezen kívül a weben megosztott mappák is elérhetők ezekkel a cmdletekkel:

```
[5] PS C:\> get-childitem \\live.sysinternals.com\Tools


    Directory: Microsoft.PowerShell.Core\FileSystem::\\live.sysinternals.co
    m\Tools


Mode           LastWriteTime       Length Name
----           -------------       ------ ----
d---- 2008. 06. 02.     1:16        <DIR> WindowsInternals
```

```
-a--- 2008. 05. 30.    17:55          668 About This Site.txt
-a--- 2008. 07. 15.    17:39       229928 accesschk.exe
-a--- 2006. 11. 01.    14:06       174968 AccessEnum.exe
-a--- 2006. 11. 01.    22:05       121712 accvio.EXE
-a--- 2007. 07. 12.     7:26        50379 AdExplorer.chm
-a--- 2007. 11. 26.    13:21       422952 ADExplorer.exe
-a--- 2007. 11. 07.    10:13       401616 ADInsight.chm
-a--- 2007. 11. 20.    13:25      1049640 ADInsight.exe
-a--- 2006. 11. 01.    14:05       150328 adrestore.exe
-a--- 2006. 11. 01.    14:06       154424 Autologon.exe
-a--- 2008. 08. 20.    15:18        48986 autoruns.chm
...
```

2.4 Az Eseménynapló feldolgozása (Get-Eventlog)

Az Eseménynapló bejegyzéseinek kiolvasására a `Get-Eventlog` cmdletet használhatjuk, ami `EventLogEntry` (vagy `EventLog`) típusú objektumokat ad vissza. Először is tudjuk meg, hogy milyen naplók vannak a gépünkön:

```
[3] PS C:\> Get-EventLog -list

  Max(K) Retain OverflowAction        Entries Name
  ------ ------ --------------        ------- ----
   8 192      0 OverwriteAsNeeded       2 595 Application
     512      7 OverwriteOlder              0 Internet Explorer
  16 384      0 OverwriteAsNeeded           8 Microsoft Office Diagnostics
  16 384      0 OverwriteAsNeeded         858 Microsoft Office Sessions
   8 192      0 OverwriteAsNeeded       4 486 Security
   8 192      0 OverwriteAsNeeded      10 965 System
  16 384      0 OverwriteAsNeeded         617 Virtual Server
  15 360      0 OverwriteAsNeeded       5 504 Windows PowerShell
```

A fenti parancs `EventLog` objektumokat ad vissza, ezektől lekérdezhetők az adott napló különféle tulajdonságai, maximális mérete, bejegyzéseinek száma, stb.

```
[4] PS C:\> (get-eventlog -list)[0] | Format-List *


Entries               : {MACHINENAME, MACHINENAME, MACHINENAME, ASUS...}
LogDisplayName        : Application
Log                   : Application
MachineName           : .
MaximumKilobytes      : 8192
OverflowAction        : OverwriteAsNeeded
MinimumRetentionDays : 0
EnableRaisingEvents   : False
SynchronizingObject   :
Source                :
Site                  :
Container             :
[5] PS C:\> (get-eventlog -list)[0].entries.count
2595
```

Ha egy konkrét napló bejegyzéseire vagyunk kíváncsiak, akkor a napló nevét kell megadnunk paraméterként, a `-newest` paraméter után álló szám pedig a listába kerülő bejegyzések számát korlátozza. Az következő parancs a rendszernapló legutóbbi tíz bejegyzését fogja kiolvasni:

```
PS C:\> Get-EventLog system -newest 10

Index Time          Type Source                EventID Message
----- ----          ---- ------                ------- -------
13237 aug. 02 08:14 Info Service Control M...     7036 A(z) Távelérési ...
13236 aug. 02 08:14 Info Service Control M...     7035 A(z) Távelérési ...
...
```

Az előző két parancs kombinálásával valamennyi napló legfrissebb bejegyzéseit is könnyen lekérdezhetjük egyetlen paranccsal. A bal oldali `Get-Eventlog` szállítja a naplók objektumait, az innen származó adatokat fogjuk odaadni paraméterként a `Foreach-Object` belsejében elinduló `Get-Eventlog`-nak, így szép sorban valamennyi napló bejegyzései elő fognak kerülni. (Közben még kiírjuk a napló nevét is, hogy tudjuk hol járunk.)

```
PS C:\> Get-Eventlog -list | Foreach-Object {Write-Host $ .LogDisplayName;
Get-Eventlog $ .Log -newest 10}
Alkalmazás

Index Time          Type Source                EventID Message
----- ----          ---- ------                ------- -------
18387 aug. 02 15:22 Info MSSQL$SQLEXPRESS       17896 The time stamp c...
18386 aug. 02 14:18 Info MSSQL$SQLEXPRESS       17896 The time stamp c...
...
Rendszer
13237 aug. 02 08:14 Info Service Control M...     7036 A(z) Távelérési ...
13236 aug. 02 08:14 Info Service Control M...     7035 A(z) Távelérési ...
...
```

Természetesen a megszokott eszközök segítségével tetszés szerint szűrhetjük és formázhatjuk is a naplókból nyert objektumokat. Kiválogathatjuk például csak a hibákat vagy a figyelmeztetéseket, és válogathatunk a kiírandó jellemzők között.

? **Feladat:** Listázzuk ki a rendszernaplóból az utolsó héten keletkezett „Error” típusú bejegyzéseket időrend szerint!

A `Where-Object` szűrőjén tehát azoknak a bejegyzéseknek kellene átjutniuk, amelyekben az `EntryType` tulajdonság értéke „Error”, a `TimeGenerated` érték pedig későbbi az egy héttel ezelőtti dátumnál. A lenti parancs `Where-Object` részében éppen ezt a logikai kifejezést láthatjuk. Ezután már csak a sorba rendezés valamint egy kis alakítgatás van hátra, és el is készült a táblázat.

```
PS C:\> Get-Eventlog system | Where-Object {$_.EntryType -eq "Error" -and
$_.timegenerated -ge (Get-Date).AddDays(-7)} | Sort-Object -descending
timegenerated | Format-Table timegenerated, message -autosize

TimeGenerated        Message
-------------        -------
2007.07.11. 16:50:23 A szolgáltatás (Conexant's BtPCI WDM Video Captur...
2007.07.07. 10:39:29 A(z) SQL Server Reporting Services (SQLEXPRESS) s...
```

```
2007.07.07. 9:41:20  A(z) SQL Server Reporting Services (SQLEXPRESS) s...
2007.07.07. 9:35:17  A(z) SQL Server Reporting Services (SQLEXPRESS) s...
```

Rakjuk össze ismét az előző két utasítást! Listázzuk ki valamennyi naplóból az előző heti „Error” típusú bejegyzéseket! Az alábbi hosszú, de még mindig egysoros parancs a megoldás:

```
PS C:\> Get-Eventlog -list | Foreach-Object {Write-Host $_.LogDisplayName;
Get-Eventlog $ .Log | Where-Object {$ .EntryType -eq "Error" -and
$ .timegenerated -ge (Get-Date).AddDays(-7)} | Sort-Object -descending
timegenerated | Format-Table timegenerated, message -autosize}

Alkalmazás

TimeGenerated        Message
-------------        -------
2007.08.01. 15:17:46 Nem válaszoló alkalmazás: wmplayer.exe, verzió: 11...
2007.07.31. 14:05:58 Accepted Safe Mode action : Microsoft Outlook.
2007.07.31. 14:05:39 Rejected Safe Mode action : Microsoft Outlook.

Internet Explorer
Biztonság
Rendszer

TimeGenerated        Message
-------------        -------
2007.07.06. 9:03:50  Az I/O alrendszertől érkezett illesztőprogram-csom...
2007.07.06. 9:03:48  Az I/O alrendszertől érkezett illesztőprogram-csom...
2007.07.06. 9:03:48  Az I/O alrendszertől érkezett illesztőprogram-csom...

Virtual Server
Windows PowerShell
...
```

Távoli gépeken lévő Eseménynaplók eléréséhez és naplóbejegyzések készítéséhez nincs cmdlet-támogatás, ezeket a feladatokat például a megfelelő WMI komponensek felhasználásával lehet elvégeznünk (lásd *2.6.9 Fontosabb WMI osztályok* fejezet), vagy közvetlenül a megfelelő .NET osztályok megszólításával:

```
[6] PS C:\> $eventlogs = [System.Diagnostics.EventLog]::GetEventLogs("asus")
[7] PS C:\> $eventlogs

  Max(K) Retain OverflowAction        Entries Name
  ------ ------ --------------        ------- ----
   8 192      0 OverwriteAsNeeded       2 595 Application
     512      7 OverwriteOlder              0 Internet Explorer
  16 384      0 OverwriteAsNeeded           8 Microsoft Office Diagnostics
  16 384      0 OverwriteAsNeeded         858 Microsoft Office Sessions
   8 192      0 OverwriteAsNeeded       4 486 Security
   8 192      0 OverwriteAsNeeded      10 965 System
  16 384      0 OverwriteAsNeeded         617 Virtual Server
  15 360      0 OverwriteAsNeeded       5 504 Windows PowerShell
```

A [6]-os sorban látható, hogy a `GetEventLogs` statikus metódusnak paraméterként átadható a gépnév, így akár távoli gép eseménynaplói is lekérdezhetők ezzel a módszerrel.

2.5 Registry kezelése

Amikor a PowerShell egyszerűségét demonstrálják különböző szakmai rendezvényeken, akkor gyakran a registry kezelését is bemutatják, hiszen jól érzékeltethető az, hogy a fájlok kezelésével kapcsolatos cmdletek legtöbbje a registryre is használható, mivel a fájlok és a registry is u.n. PSDrive-ként érhető el. A személyes tapasztalatom az, hogy azért viszonylag ritkán kell a registryt szkriptből machinálni, de azért foglalkozzunk ezzel is.

2.5.1 Olvasás a registryből

Elsőként nézzük meg, hogy hogyan lehet olvasni a registryből. Például listázzuk ki a registry alapján a számítógépre telepített alkalmazások nevét!

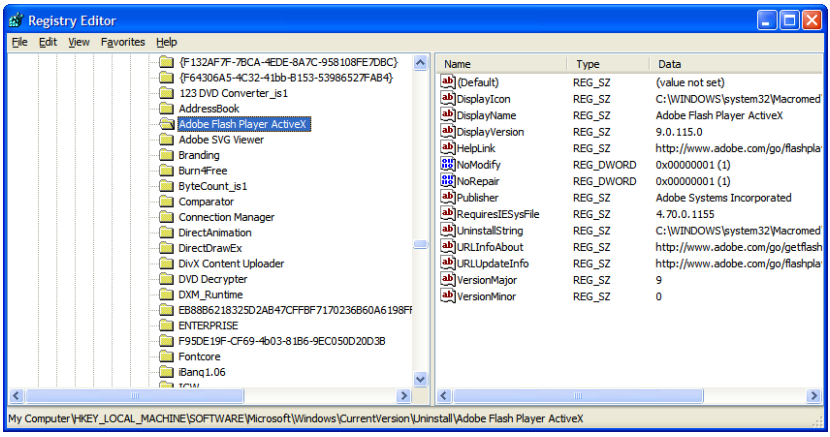
Miután a registryt is egy PSDrive-on keresztül érhetjük el, érdemes lehet akár új, a minket érdeklő információkat tartalmazó ágra új „shortcut” PSDrive-ot definiálni, hogy később már rövidebb elérési úttal is hivatkozhatók legyenek az egyes elemek:

```
[1] PS I:\>New-PSDrive -Name Uninstall -PSProvider Registry -Root HKLM:\SOFT
WARE\Microsoft\Windows\CurrentVersion\Uninstall

Name        Provider      Root                                    CurrentLocatio
                                                                          n
----        --------      ----                                    --------------
Uninstall   Registry      HKEY_LOCAL_MACHINE\SOFTWARE\Micr...
```

A mostani feladatunkban például az `Uninstall` registry kulcsot egy új, „Uninstall” nevű meghajtón keresztül lehet elérni. Erre már nagyon egyszerűen végezhetünk szokásos fájlműveleteket, például kilistázhatjuk az itt található kulcsokat a `Get-ChildItem` cmdlet segítségével:

```
[2] PS I:\>Get-ChildItem uninstall:


   Hive: Microsoft.PowerShell.Core\Registry::HKEY LOCAL MACHINE\SOFTWARE\Mi
crosoft\Windows\CurrentVersion\Uninstall

SKC  VC Name                           Property
---  -- ----                           --------
  0  16 123 DVD Converter is1          {Inno Setup: Setup Version, Inno ...
  0   1 AddressBook                    {(default)}
  0  13 Adobe Flash Player ActiveX     {DisplayName, DisplayVersion, Pub...
  0  12 Adobe SVG Viewer               {DisplayName, DisplayVersion, Dis...
  0   2 Branding                       {QuietUninstallString, RequiresIE...
  0   2 Burn4Free                      {DisplayName, UninstallString}
...
```

Nézzük meg ugyanezt a regedittel:



51. ábra Uninstall registry-ág

Látható, hogy amit mi a Regeditben látunk „fájlként”, azaz az Adobe Flash Playernél mint `DisplayName`, az a PowerShell-ben nem „fájl”, hanem az már a fájlnak a tulajdonsága (property). Ez egy kicsit zavarónak tűnhet, hiszen a szemünk előtt az Adobe Flash Player mint mappa lebeg, holott a PowerShellben az maga a fájlszintű objektum a hierarchiában. De akkor hogyan jutunk hozzá az egyes tulajdonságokhoz egyszerűen? Erre is van cmdlet, a `Get-ItemProperty`:

```
[3] PS I:\>cd Uninstall:
HKEY LOCAL MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\
[4] PS Uninstall:\>Get-ItemProperty "Adobe Flash Player ActiveX"


PSPath            : Microsoft.PowerShell.Core\Registry::HKEY LOCAL MACHINE\
                    SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\Ado
                    be Flash Player ActiveX
PSParentPath      : Microsoft.PowerShell.Core\Registry::HKEY LOCAL MACHINE\
                    SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall
PSChildName       : Adobe Flash Player ActiveX
PSDrive           : Uninstall
PSProvider        : Microsoft.PowerShell.Core\Registry
DisplayName       : Adobe Flash Player ActiveX
DisplayVersion    : 9.0.115.0
Publisher         : Adobe Systems Incorporated
URLInfoAbout      : http://www.adobe.com/go/getflashplayer
VersionMajor      : 9
VersionMinor      : 0
HelpLink          : http://www.adobe.com/go/flashplayer support/
URLUpdateInfo     : http://www.adobe.com/go/flashplayer/
DisplayIcon       : C:\WINDOWS\system32\Macromed\Flash\uninstall_activeX.ex
                    e
```

```
UninstallString   : C:\WINDOWS\system32\Macromed\Flash\uninstall activeX.ex
                    e
RequiresIESysFile : 4.70.0.1155
NoModify          : 1
NoRepair          : 1
```

A [4]-es sorban lekérdezem az *Adobe Flash Player* elem tulajdonságait. Egy kicsit többet is visszaad nekem a PowerShell, mint ami ténylegesen a registryben található: `PSPath`, `PSPartentPath`, stb. Ezek természetesen nincsenek benne a registryben, ezeket a PowerShell típusadaptációs rétege teszi hozzá. Sajnos ezeket a plusz adatokat akkor is hozzábiggyeszti, ha csak mondjuk a `DisplayName` paramétert akarjuk kiolvasni:

```
[14] PS Uninstall:\>Get-ItemProperty "Adobe Flash Player ActiveX" -name Disp
layName


PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTW
               ARE\Microsoft\Windows\CurrentVersion\Uninstall\Adobe Flash P
               layer ActiveX
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTW
               ARE\Microsoft\Windows\CurrentVersion\Uninstall
PSChildName  : Adobe Flash Player ActiveX
PSDrive      : Uninstall
PSProvider   : Microsoft.PowerShell.Core\Registry
DisplayName  : Adobe Flash Player ActiveX
```

Szóval trükközni kell, ha tényleg csak az adott kulcs értékét akarom megtekinteni:

```
[24] PS Uninstall:\>(Get-ItemProperty "Adobe Flash Player ActiveX").DisplayN
ame
Adobe Flash Player ActiveX
```

Azaz külön hivatkozni kell az adott kulcs nevére, nem igazán jó a `-name` paraméter használata. Ez függvényért kiált, de ennek megvalósítását az olvasóra bízom.

Még két lehetőségre hívnám fel a figyelmet a registry kulcsok olvasásával kapcsolatban. Az egyik `GetValueNames()` metódus, amellyel az adott kulcs tényleges értékeinek neveit lehet kiolvasni. A nevek birtokában aztán például valamelyik ciklussal lehet műveleteket végezni az értékekkel. Erre példaként nézzünk egy szkriptet, amely összeszámolja, hogy gépünkre hány font van telepítve, és abból mennyi a *TrueType* típusú:

```
[15] PS C:\> $ttf=0; (get-item "hklm:\\Software\Microsoft\Windows NT\Current
Version\Fonts").GetValueNames() | Where-Object {$_.contains("TrueType")} | F
orEach-Object {$ttf++}; $ttf
277
```

Ugyanezt a funkciót valósítottam meg következő szkriptben is, csak itt nem a `GetValueNames()` metódust alkalmaztam, hanem a PowerShell `Select-Object` cmdletjét, az `ExtendProperty` paraméterrel, mellyel egy adott objektum egy adott, tömböt tartalmazó tulajdonságának értékeit lehet kifejteni:

```
[22] PS C:\> $ttf=0; Get-Item "HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVe
rsion\Fonts" | Select-Object -ExpandProperty Property | where-object {$ .con
tains("TrueType")} | ForEach-Object {$ttf++}; $ttf
277
```

2.5.2 Registry elemek létrehozása, módosítása

Ezek után nem meglepő, hogy a registry elemek létrehozása is hasonló módon történik, mint a fájlok létrehozása. Megint fontos tudatosítani, hogy mi a „fájlszintű” objektum a registryben, és mi a tulajdonság.

Nézzük egy új tulajdonság létrehozását, hozzunk létre az Outlook törölt elemek visszaállíthatóságát megkönnyítő kulcsot:

```
[2] PS I:\>Set-Location HKLM:\SOFTWARE\Microsoft\Exchange\Client\Options
[3] PS HKLM:\SOFTWARE\Microsoft\Exchange\Client\Options>New-ItemProperty . -
Name DumpsterAlwaysOn -Value 1 -type DWORD


PSPath           : Microsoft.PowerShell.Core\Registry::HKEY LOCAL MACHINE\S
                   OFTWARE\Microsoft\Exchange\Client\Options
PSParentPath     : Microsoft.PowerShell.Core\Registry::HKEY LOCAL MACHINE\S
                   OFTWARE\Microsoft\Exchange\Client
PSChildName      : Options
PSDrive          : HKLM
PSProvider       : Microsoft.PowerShell.Core\Registry
DumpsterAlwaysOn : 1
```

A [2]-es sorban az aktuális helynek beállítom az a registry „mappát”, ahol új értéket akarok felvenni, majd a `new-itemproperty` cmdlettel hozom létre az új értéket. Ennek paraméterei a `path` (nincs kiírva, értéke egy darab pont, azaz az aktuális elérési út), a kulcs neve és típusa. Típusként az alábbi táblázat lehetőségeit használhatjuk fel a registryben:

Property típus	Leírás
Binary	bináris adat
DWord	UInt32 egész
ExpandString	Környezeti változókat kifejtő szöveg
MultiString	Többsoros szöveg
String	Szöveg
QWord	8 bájtos bináris adat

Meglevő kulcsok módosítására a `set-itemproperty` cmdlet áll a rendelkezésünkre:

```
[9] PS HKLM:\SOFTWARE\Microsoft\Exchange\Client\Options>set-ItemProperty . -
Name DumpsterAlwaysOn -Value 0
```

Ha esetleg új kulcsot kellene létrehoznunk, arra a `new-item` cmdletet használhatjuk:

```
[14] PS HKLM:\SOFTWARE>New-Item SoosTibor


   Hive: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE

SKC  VC Name                           Property
---  -- ----                           --------
  0   0 SoosTibor                      {}
```

Ezután ehhez a `new-itemproperty` cmdlettel lehet felvenni értékeket.

2.5.3 Registry elemek hozzáférési listájának kiolvasása

A registry kulcsok is rendelkeznek hozzáférési listával, itt is tetten érhető a fájlokkal való analógia, azaz a registry esetében is a `get-acl` cmdlettel lehet a hozzáférési listát kiolvasni, illetve a `set-acl` cmdlettel módosítani. Miután ez tényleg a fájlokkal teljesen azonos módon történik, így csak a kiolvasásra mutatok egy példát:

```
[84] PS HKCU:\Software> get-acl -path hklm:\system\currentcontrolset\ | fl


Path   : Microsoft.PowerShell.Core\Registry::HKEY LOCAL MACHINE\system\curr
         entcontrolset
Owner  : BUILTIN\Administrators
Group  : NT AUTHORITY\SYSTEM
Access : BUILTIN\Users Allow  ReadKey
         BUILTIN\Users Allow  -2147483648
         BUILTIN\Power Users Allow  ReadKey
         BUILTIN\Power Users Allow  -2147483648
         BUILTIN\Administrators Allow  FullControl
         BUILTIN\Administrators Allow  268435456
         NT AUTHORITY\SYSTEM Allow  FullControl
         NT AUTHORITY\SYSTEM Allow  268435456
         CREATOR OWNER Allow  268435456
Audit  :
Sddl   : O:BAG:SYD:AI(A;ID;KR;;;BU)(A;CIIOID;GR;;;BU)(A;ID;KR;;;PU)(A;CIIOI
         D;GR;;;PU)(A;ID;KA;;;BA)(A;CIIOID;GA;;;BA)(A;ID;KA;;;SY)(A;CIIOID;
         GA;;;SY)(A;CIIOID;GA;;;CO)
```

Megjegyzés:

Itt jöhetünk rá, hogy vajon miért nem az értékek a „fájlszintű” objektumok a registry PSDrive-ban. Azért, mert ACL-t állítani csak a kulcsokon lehet, a tulajdonságokon (property) nem. Így a fájlrendszerrel való analógia a „kulcs ≈ mappa, fájl” megfeleltetéssel a legoptimálisabb.

2.6 WMI

A WMI technológia biztosítja a hálózatfelügyeleti szoftverek (így például a Microsoft Systems Management Server) számára szükséges infrastruktúrát. Kifejlesztésének célja elsősorban az volt, hogy egységes keretet (és felületet) biztosítson a már létező felügyeleti technológiáknak (SNMP, DMI, stb.).

A WMI első verziója a Windows NT SP4 CD-n jelent meg (Option Pack), de az SMS 2.0 verziója már teljes egészében erre épül. Jelentős különbségek vannak azonban a Windows 2000-ben, az XP-ben és a Windows Server 2003-ban található változatok között. A legfontosabb különbség az, hogy a későbbi WMI változatok egyre több írható tulajdonságot tartalmaznak, vagyis lehetőséget adnak nem csak a rendszerjellemzők lekérdezésére, hanem beállítására is.

A WMI tartozéka a WMI Tools nevű csomag, de ez nem része a Windows telepítésnek, külön kell letölteni és telepíteni a Microsoft „downloads" weboldaláról. A csomag tartalmazza többek között a CIM Studio és az Object Browser nevű alkalmazásokat; ezekkel a későbbiekben még fogunk találkozni.

WMI-re alapuló rendszerfelügyeleti megoldásokat természetesen nem csak a Microsoft, hanem számos más szoftvercég palettáján is találhatunk.

2.6.1 A WMI áttekintése

Mielőtt belemerülnénk a WMI technológia részleteibe, érdemes tisztázni, hogy mire is jó ezek ismerete, hiszen a szoftvercégek jó pénzért megírják nekünk a kiváló felügyeleti alkalmazásokat, mi kattintgathatunk a remek felületen. A WMI technológia pedig legyen csak a szoftvercégek fejlesztőinek problémája, ők azért kapják a fizetésüket, hogy ilyesmit megtanuljanak és használjanak.

Nagyjából három olyan ok van, ami miatt mégsem kerülhetjük el a WMI megismerését:

Egyedi igények: – Bár a rendszerfelügyeleti szoftverek meglehetősen sokféle feladat megoldására képesek, speciális, egyedi funkciók mégis hiányozhatnak belőlük. Ilyenkor két dolgot tehetünk: kivárjuk azt a néhány évet, amíg a következő verzió megjelenik (talán abban benne lesz), vagy előkapjuk a Notepadot, és néhány sorban megírjuk magunk a hiányzó funkciót, WMI-t használó PowerShell script segítségével.

Költségek: A szoftvercégek kiváló termékei sajnos pénzbe kerülnek, méghozzá rendszer-felügyeleti szoftver esetében (nagyon) sok pénzbe. Egy Microsoft Systems Center Configuration Manager vagy IBM Tivoli megvásárlása néhány tucat gép esetében szinte reménytelen (és teljesen fölösleges is), de még pár száz gép esetén sem biztos, hogy kifizetődő. A néhány valóban szükséges funkció (például hardver és szoftverleltár, számítógépek monitorozása, riasztások, stb.) WMI segítségével egészen könnyen megvalósítható.

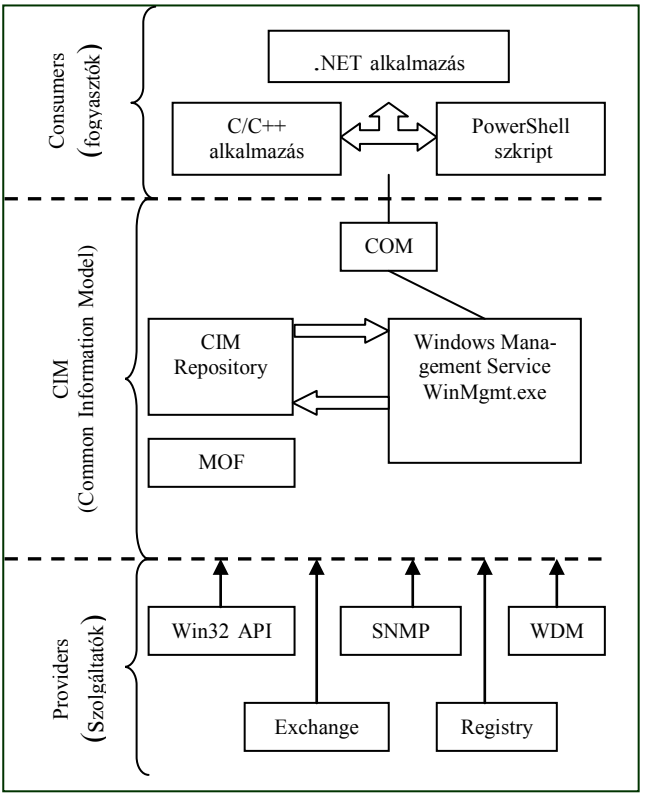
WMI-szűrők: A Windows Server 2003 újdonsága, hogy a csoportházirend (Group Policy) hatókörét WMI-szűrők segítségével módosíthatjuk. Mit is jelent ez? Tegyük fel, hogy

csoportházirend segítségével szoftvert terítünk a hálózaton (300 különféle számítógép). A telepítendő szoftvernek viszont megvan az az eléggé el nem ítélhető tulajdonsága, hogy csak olyan gépen működik megfelelően, amelyben legalább 128 MB RAM van. Ha ezek a gépek nem egy külön OU-ban vannak (miért is lennének?), akkor vagy körbejárjuk a 300 gépet a manuális telepítéshez, vagy felkészülünk rá, hogy az automatikus telepítés valamilyen hibaüzenettel megszakad. A harmadik megoldás a WMI-szűrő használata. Ekkor az adott számítógép a szűrő hatására nyilatkozik a benne lévő memória mennyiségéről, és a telepítés csak megfelelő eredmény esetén indul el.

2.6.2 A WMI felépítése

A WMI a CIM (Common Information Model) segítségével jeleníti meg az operációs rendszer felügyelt objektumait. Felügyelt objektum lehet bármelyik szoftver, hardvereszköz, logikai vagy fizikai komponens, amelynek állapota a rendszergazdát érdekelheti.

A WMI három elkülöníthető rétegből áll, a következőkben ezekről lesz szó.



52. ábra A WMI rétegei

Consumers (fogyasztók) – Fogyasztóknak nevezzük azokat az alkalmazásokat, amelyek felhasználják a WMI által biztosított adatokat. Fogyasztók lehetnek például szkriptek, Active X vezérlők, .NET alapú programok, vagy vállalati rendszerfelügyeleti eszközök (MOM, SMS, stb.). Valamennyi fogyasztó a Windows Management Service (WinMgmt.exe) által megvalósított COM csatolófelületen keresztül fér hozzá az adatokhoz. Az alkalmazásoknak természetesen nem kell tudniuk, hogy az egyes rendszerkomponensekre vonatkozó adatok valójában honnan és milyen módon származnak; nekik csak a COM csatolófelület által nyújtott lehetőségeket kell felhasználniuk.

CIM – A CIM rétegben találhatók a WMI központi komponensei. A CIM Repository tartalmazza azokat az osztálydefiníciókat, amelyekre a rendszer felügyelt objektumainak megjelenítéséhez szükség van, a Windows Management Service pedig a CIM Repository alapján továbbítja a providerektől kapott adatokat a fogyasztó alkalmazások felé. A MOF (Management Object Format) fájlok a CIM Repository bővítését teszik lehetővé. Ilyen fájlokat a WMI részeként kapott MOF compiler (mofcomp.exe) segítségével készíthetünk. Maga a CIM Repository is több ilyen módon lefordított .mof fájlból tevődik össze.

Providers (szolgáltatók) – a szolgáltatók feladata a felügyelt objektumokkal való közvetlen kommunikáció, azok saját API-jának felhasználásával. A különféle rendszerkomponens-csoportok adatainak lekérdezését önálló szolgáltatók végzik. A WMI csomag maga is számos szolgáltatót tartalmaz (a későbbi Windows verziók egyre többet), de természetesen sok alkalmazás hozza a saját szolgáltatóját, amelynek segítségével az alkalmazás adatai elérhetővé válnak WMI-n keresztül

Az adatok továbbítása a következő módon történik: az alkalmazás a COM felület használatával bármely felügyelt objektum tetszőleges adatára rákérdezhet. A WinMgmt.exe a CIM Repository adatainak felhasználásával meghatározza, hogy az adott információt melyik providertől, és milyen módon kell elkérnie. Ezután megszólítja a providert, az pedig lekérdezi az objektum megfelelő adatát. Ezt azután a WinMgmt.exe továbbítja a fogyasztó alkalmazás felé.

Vagyis mondhatjuk azt, hogy a CIM Repository tulajdonképpen egy egységes nyilvántartás, amelynek segítségével a különböző providerek kezelése azonos módon történhet. Minden adatot, amelyet a WMI-n keresztül el szeretnénk érni, tartalmaznia kell a CIM Repository objektummodelljének, és regisztrálnunk kell a használni kívánt providereket is.

2.6.3 A WMI objektummodellje

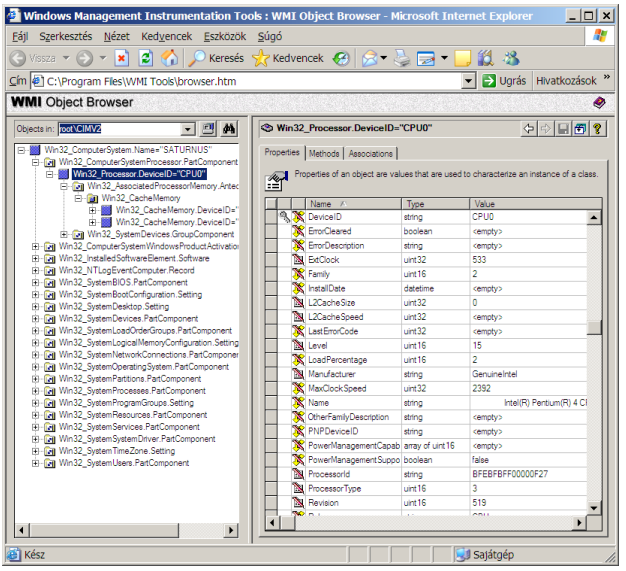
A WMI működésének megértéséhez mindenképpen szükséges az objektummodell, és az ezzel kapcsolatos fogalmak ismerete; a következőkben erről lesz szó.

Osztályok (Classes) – A WMI objektummodellje osztályokon alapul. Az osztály a rendszer valamely felügyelt objektumának típusleírása, amely tartalmazza az adott objektum tulajdonságait és az általa támogatott metódusokat. A Win32_NetworkAdapter osztály például a számítógépben lévő hálózati adapterek típusleírását tartalmazza. Az osztályok között természetesen öröklődés is létezik. A szokásos felügyeleti szkriptek (programok) általában az öröklési lánc legvégén lévő (levél) objektumokkal foglalkoznak, de természe-

tesen lehetőség van arra is, hogy feljebb menjünk a hierarchiában, és egy adott kezdőpont alatt lévő osztályok hasznos adatait nyerjük ki, anélkül, hogy pontosan tudnánk azok nevét. Absztrakt osztálynak nevezzük azokat az osztályokat, amelyekből nem lehet példányt létrehozni; kizárólag örökítési célokat szolgálnak.

Tulajdonságok (Properties) – Az osztályokhoz tulajdonságok tartoznak, amelyek az osztály által meghatározott objektumok leírására szolgálnak. Természetesen minden osztályban olyan tulajdonságok vannak definiálva, amelyek az adott típusú objektum leírásához szükségesek. Számos osztályban találkozhatunk kulcs (key) tulajdonsággal, ezek (az adatbázisokhoz hasonlóan) az adott osztály példányainak egyedi azonosítására használhatók. A Win32_NetworkAdapter osztály kulcstulajdonsága például a „DeviceID”.

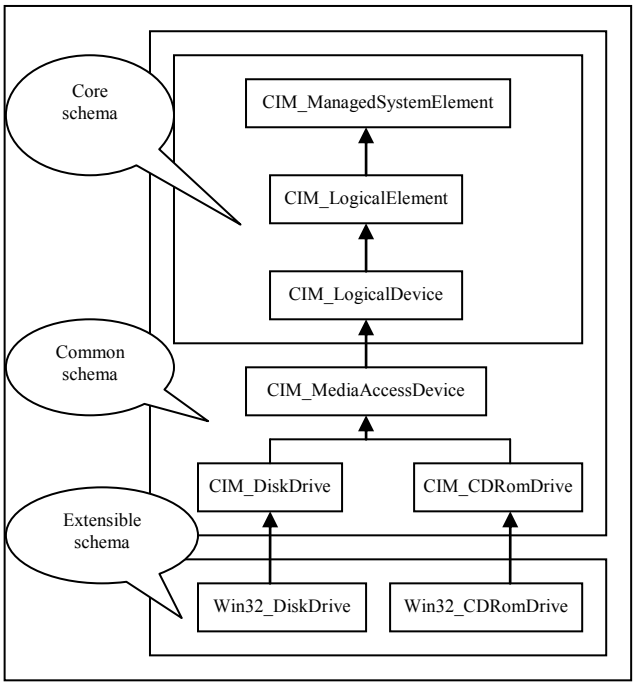
Példányok (Instances) – Míg a Win32_NetworkAdapter osztály bármilyen hálózati adapter leírására képes, az osztály példánya egy bizonyos, fizikailag is létező adapter reprezentációja. Az egyes példányokat az adott osztály kulcs tulajdonságának ismeretében szólíthatjuk meg. Az objektum példányokat a WMI Object Browser segítségével jeleníthetjük meg (az eszköz a WMI Tools csomag része).



53. ábra WMI Object Browser

2.6.4 Sémák

A Common Information Model sémákból épül fel, amelyek egymással kapcsolatban álló osztályokat tartalmaznak. Jelenleg a CIM három sémát tartalmaz, az alábbi ábrának megfelelően:



54. ábra A CIM sémák

A „core” sémához tartozó absztrakt osztályok kevés konkrétumot tartalmaznak, céljuk az, hogy újabb osztályokat örökíthessünk belőlük. Ezen a szinten tulajdonképpen még az sem biztos, hogy az örökített osztályoknak bármi köze is lesz a számítógépes rendszerekhez; modellezhetnek akár épületeket, vagy berendezési tárgyakat is. A „common” séma viszont már határozottan számítógéprendszerek modellezésére szolgál.

Az említett két sémát a WBEM szabvány definiálja, így azok gyártó- és platformfüggetlenek.

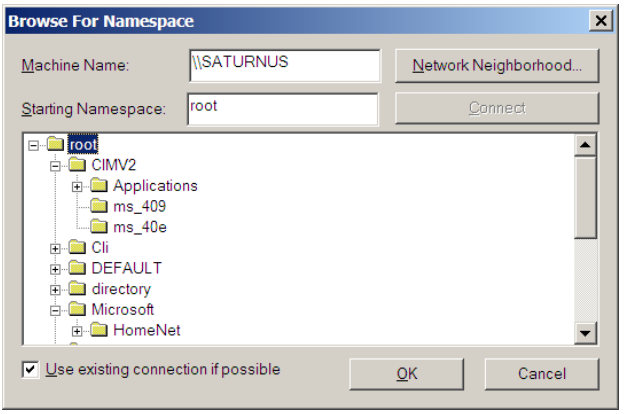
Az „extensible” sémák viszont már az egyes szoftvergyártók hatáskörébe tartoznak; ide kerülhetnek az adott platformra vonatkozó speciális osztályok. A WMI a Win32_ sémát használja a felügyelt objektumok modellezésére; az itt megtalálható osztályok a „common” séma osztályainak bővített (örökített) változatai, így megjeleníthetik a Windows operációs rendszerekre jellemző speciális tulajdonságokat is.

2.6.5 Névterek

A WMI osztályai különböző névterekhez tartoznak annak megfelelően, hogy melyik rendszerterületet jelenítik meg. A névterek szervezése hierarchikus, a fa gyökere a „root” névtér. Az egyes osztályok útvonalának megadásakor először is meg kell határoznunk azt

a számítógépet, amelyik a CIM Repositoryt tartalmazza, majd sorban meg kell adnunk a hozzá vezető névtér-hierarchia elemeit:

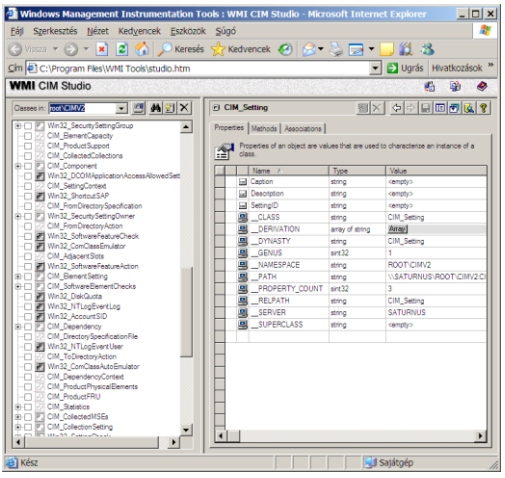
```
\\Gep\Root\Cimv2:Win32_LogicalDisk.DeviceID='C:'
```



55. ábra A névterek szervezése hierarchikus

Amint az ábrán is látható, minden számítógépen számos különböző névteret találhatunk, de a számítógép hardverelemeivel és az operációs rendszerrel kapcsolatos objektumokat leíró osztályok szinte kivétel nélkül a Cimv2 névtérben találhatók, így természetesen a rendszerfelügyeleti szkriptekben ez lesz a legtöbbet használt névtér.

A WMI CIM Studio (WMI Tools) segítségével a sémát és a névtereket jeleníthetjük meg. Felhasználhatjuk osztályok, illetve ezek tulajdonságainak és metódusainak megkeresésére (erre gyakran lesz szükség), és az osztályok közötti kapcsolatok feltérképezésére is.



56. ábra WMI CIM Studio

2.6.6 A legfontosabb providerek

A WMI providerek két forrásból származhatnak; vannak alkalmazás specifikus (ezeket egyes alkalmazások telepítik), és vannak beépített (ezeket a Windows részeként kapjuk) providerek. A következőkben áttekintjük az operációs rendszer részeként érkező legfontosabb providereket:

Win32 – a Win32 provider a számítógép hardver elemeihez és az operációs rendszer legfontosabb komponenseihez tartozó osztályok kiszolgálását végzi. A provider az adatok összegyűjtéséhez a Win32 API-t és különféle registry értékeket használ. A root/cimv2 névtér osztályainak döntő többségét ez a provider szolgálja ki.

SNMP – A provider a meglévő SNMP infrastruktúra és a WMI lehetőségeinek együttes használatát teszi lehetővé. A Windows Server 2003-ból ez a provider már hiányzik.

Performance Counter – A WMI legújabb verziójában a Windows Management Service a Performance Monitor által is használt adatfájlok alapján felépíti a teljesítményobjektumokat reprezentáló osztályokat a CIM Repositoryban. Az alkalmazások a többi WMI osztályhoz hasonlóan kérdezhetik le a teljesítményadatokat.

Registry – A registry providert felhasználó osztályok lehetővé teszik, hogy az alkalmazások írhassák és olvashassák a registryben szereplő értékeket. A RegistryEvent provider (regevent.mof) segítségével pedig alkalmazásunk értesítést kaphat a kiválasztott registry értékek módosulásáról.

Windows Driver Model – A Windows Driver Model lehetővé teszi, hogy az eszközvezérlő programok adatokat szolgáltassanak az általuk vezérelt eszközzel kapcsolatban .A provider ezekhez az adatokhoz biztosít hozzáférést a root/wmi névtérben létrehozott osztályok segítségével.

Directory Services – a DS provider az Active Directory osztályait és objektumait teszi elérhetővé a WMI-t használó alkalmazások számára. A provider az AD sémát képezi le a WMI sémába. A DS provider ADSI segítségével csatlakozik az Active Directoryhoz.

Event Log – a provider a Windows Eseménynapló szolgáltatáshoz biztosít hozzáférést, és lehetővé teszi azt is, hogy programunk értesítést kapjon az új naplóbejegyzések keletkezéséről.

Windows Installer – a provider a root/cimv2 névtérben létrehozott osztályok segítségével biztosítja a hozzáférést a Windows Installer szolgáltatás által telepített csomagokkal kapcsolatos adatokhoz. A provider lehetővé teszi az MSI csomagok telepítését, eltávolítását és beállítását is.

Security – A provider lehetővé teszi a Windows rendszer biztonsági beállításainak kiolvasását és módosítását. Beállíthatjuk a fájlok és mappák tulajdonosát, naplózását, és a hozzáférési jogokat is.

2.6.7 WMI objektumok elérése PowerShell-ből

A PowerShell 1.0-ban egyetlen egy cmdletbe sűrítették az alkotók a WMI-vel kapcsolatos funkciókat, ez a `Get-WMIObject`. Nézzük meg a lehetséges szintaxisokat:

```
[5] PS I:\>(get-help Get-WmiObject).syntax

Get-WmiObject [-class] <string> [[-property] <string[]>] [-namespace <strin
g>] [-computerName <string[]>] [-filter <string>] [-credential <PSCredentia
l>] [<CommonParameters>]
Get-WmiObject [-namespace <string>] [-computerName <string[]>] [-credential
 <PSCredential>] [-list] [<CommonParameters>]
Get-WmiObject -query <string> [-namespace <string>] [-computerName <string[
]>] [-credential <PSCredential>] [<CommonParameters>]
```

Elsőként próbáljuk meg kilistázni a különböző WMI osztályokat. Ehhez a három lehetséges szintaxis közül a második használata szükséges, ahol meghatározhatunk egy névteret a listánk szűkítése céljából:

```
[7] PS I:\>get-wmiobject -namespace "root/CIMV2" -list


__SecurityRelatedClass                  __NTLMUser9X
__PARAMETERS                            __SystemSecurity
__NotifyStatus                          __ExtendedStatus
Win32 PrivilegesStatus                  Win32 TSNetworkAdapterSettingError
Win32 TSRemoteControlSettingError       Win32 TSEnvironmentSettingError
Win32_TSSessionDirectoryError           Win32_TSLogonSettingError
Win32_TerminalError                     Win32_JobObjectStatus
...
Win32_Product                           Win32_ComputerSystemProduct
Win32 ImplementedCategory               CIM SoftwareElementActions
Win32_SoftwareElementAction             CIM_ToDirectorySpecification
CIM_ReplacementSet                      CIM_Configuration
CIM ActsAsSpare                         CIM FRUIncludesProduct
CIM_PhysicalElementLocation
```

Ha nem használjuk a `-list` kapcsolót, akkor az első szintaxis használatára gondol és bekéri az abban használatos `-class` paramétert.

Ha már tudjuk, hogy milyen osztály objektumait keressük, akkor használhatjuk az első szintaxist. Például keressük az adott gépre telepített különböző szoftvereket:

```
[11] PS I:\>get-wmiobject Win32_Product


IdentifyingNumber : {121634B0-2F4B-11D3-ADA3-00C04F52DD52}
Name              : Windows Installer Clean Up
Vendor            : Microsoft Corporation
Version           : 3.00.00.0000
Caption           : Windows Installer Clean Up

IdentifyingNumber : {EA1E2517-9281-4684-93B2-460530295ED3}
Name              : PowerShellScriptOMatic v.1.0
Vendor            : MrEdSoftware
Version           : 1.0.0
Caption           : PowerShellScriptOMatic v.1.0
...
```

Gyakran nem az adott osztály összes objektumára van szükségünk, hanem valamilyen szempont szerint kiválasztott példányokra. Erre a célra a WMI-ben az SQL nyelvhez nagyon hasonló WQL (WMI Query Language) nyelvet lehet használni. Ilyen lekérdezéseket a `Get-WMIObject` harmadik szintaxisa szerint lehet a parancssorunkba beágyazni:

```
[14] PS I:\>Get-WmiObject -Query 'select * from Win32 Product where Vendor="
Microsoft Corporation"'


IdentifyingNumber : {121634B0-2F4B-11D3-ADA3-00C04F52DD52}
Name              : Windows Installer Clean Up
Vendor            : Microsoft Corporation
Version           : 3.00.00.0000
Caption           : Windows Installer Clean Up

IdentifyingNumber : {90120000-0010-0409-0000-0000000FF1CE}
Name              : Microsoft Software Update for Web Folders  (English) 12
Vendor            : Microsoft Corporation
Version           : 12.0.6215.1000
Caption           : Microsoft Software Update for Web Folders  (English) 12
...
```

Megjegyzés

Vigyázni kell az idézőjelezésre, hiszen a WQL kifejezést idézőjelezni kell, amin belül szintén idézőjel szerepel általában a `where` feltételnél. Ezt legegyszerűbben a kétfajta idézőjel használatával oldhatjuk meg, mint ahogy a fenti példában tettem, és akkor nem kell az *escape* (` Alt Gr+7) karaktert használni.

Ha nagyon egyszerű a kiválasztási kritériumunk, akkor használhatjuk az első szintaxis `-filter` opcióját is:

```
[20] PS I:\>Get-WmiObject Win32 Product -filter 'Vendor="Microsoft Corporati
on"'
```

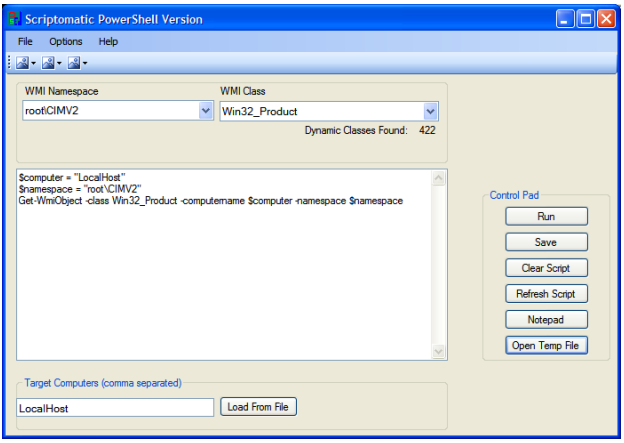
A `-filter` paraméterrel gyakorlatilag a lekérdezés `where` részét adhatjuk meg.

A WMI-ben az még a nagyszerű, hogy távoli gépekre is kiadhatók lekérdezések. Magának a PowerShellnek ilyen jellegű „távfuttatása” csak a 2.0 verzióban fog megjelenni, azaz jelenleg a `get-service` cmdlet csak saját gépre adható ki, de az alábbi paranccsal hasonló információkhoz juthatunk a WMI segítségével távoli gépre vonatkozólag is:

```
[21] PS I:\>get-wmiobject win32_service -credential IQJB\soostibor -computer
 J-CRM
```

A `-computer` és `-credential` paraméterek meghatározásával adhatjuk meg, hogy melyik gépre és kinek a nevében akarunk csatlakozni. Természetesen a parancs sikeres futtatásához biztosítani kell a Windows tűzfalon a megfelelő portok megnyitását, hogy a kommunikáció sikeres legyen.

A WMI osztályok böngészésében, és az osztályok példányainak kilistázásához szükséges PowerShell parancs összeállításában segít a PowerShellScriptOMatic ingyenesen letölthető apró kis eszköz:



57. ábra PowerShellScriptOMatic

2.6.8 WMI objektumok metódusainak meghívása

A WMI objektumok is rendelkeznek metódusokkal. Ezek felderítésére használhatjuk a PowerShell `get-member` cmdletjét. Például nézzük a `Win32_Share` WMI osztály elemeinél milyen metódusok állnak rendelkezésünkre:

```
[23] PS I:\>Get-WmiObject Win32_Share

Name                    Path                    Description
----                    ----                    -----------
IPC$                                            Remote IPC
_download               C:\_download
DVD                     E:\
DVD2                    D:\
ADMIN$                  C:\WINDOWS              Remote Admin
C$                      C:\                     Default share


[24] PS I:\>Get-WmiObject Win32_Share | Get-Member


   TypeName: System.Management.ManagementObject#root\cimv2\Win32 Share

Name                MemberType   Definition
----                ----------   ----------
GetAccessMask       Method       System.Management.ManagementBaseObject ...
SetShareInfo        Method       System.Management.ManagementBaseObject ...
AccessMask          Property     System.UInt32 AccessMask {get;set;}
AllowMaximum        Property     System.Boolean AllowMaximum {get;set;}
```

```
Caption              Property     System.String Caption {get;set;}
Description          Property     System.String Description {get;set;}
InstallDate          Property     System.String InstallDate {get;set;}
MaximumAllowed       Property     System.UInt32 MaximumAllowed {get;set;}
Name                 Property     System.String Name {get;set;}
Path                 Property     System.String Path {get;set;}
Status               Property     System.String Status {get;set;}
Type                 Property     System.UInt32 Type {get;set;}
__CLASS              Property     System.String __CLASS {get;set;}
__DERIVATION         Property     System.String[] __DERIVATION {get;set;}
  DYNASTY            Property     System.String   DYNASTY {get;set;}
  GENUS              Property     System.Int32   GENUS {get;set;}
  NAMESPACE          Property     System.String   NAMESPACE {get;set;}
__PATH               Property     System.String __PATH {get;set;}
__PROPERTY_COUNT     Property     System.Int32 __PROPERTY_COUNT {get;set;}
__RELPATH            Property     System.String __RELPATH {get;set;}
  SERVER             Property     System.String   SERVER {get;set;}
  SUPERCLASS         Property     System.String   SUPERCLASS {get;set;}
PSStatus             PropertySet  PSStatus {Status, Type, Name}
ConvertFromDateTime ScriptMethod System.Object ConvertFromDateTime();
ConvertToDateTime   ScriptMethod System.Object ConvertToDateTime();
Delete               ScriptMethod System.Object Delete();
GetType              ScriptMethod System.Object GetType();
Put                  ScriptMethod System.Object Put();
```

Például rendelkezésünkre áll a `Delete()` metódus, amellyel az adott megosztást tudjuk megszűntetni:

```
[25] PS I:\>$shares = Get-WmiObject Win32 Share
[26] PS I:\>$shares

Name                      Path                      Description
----                      ----                      -----------
IPC$                                                Remote IPC
 download                 C:\ download
DVD                       E:\
DVD2                      D:\
ADMIN$                    C:\WINDOWS                Remote Admin
C$                        C:\                       Default share


[27] PS I:\>$shares[3].Delete()
[28] PS I:\>Get-WmiObject Win32_Share

Name                      Path                      Description
----                      ----                      -----------
IPC$                                                Remote IPC
 download                 C:\ download
DVD                       E:\
ADMIN$                    C:\WINDOWS                Remote Admin
C$                        C:\                       Default share
```

A fenti példában a [27]-es sorban töröltem a DVD2 nevű megosztásomat.

De vajon hogyan lehet új megosztást létrehozni? Nagyon egyszerű, egy új `Win32_Share` objektumot kell definiálni, és annak van `Create()` metódusa:

```
[31] PS I:\>$newshare = [wmiclass] "Win32 Share"
[32] PS I:\>$newshare | Get-Member


   TypeName: System.Management.ManagementClass#ROOT\cimv2\Win32_Share

Name                    MemberType    Definition
----                    ----------    ----------
Name                    AliasProperty Name = __Class
Create                  Method        System.Management.ManagementBaseObj...
  CLASS                 Property      System.String   CLASS {get;set;}
...
```

Ennek paraméterezését a következő módon tudjuk megnézni, a tényleges paraméterezés a `Value` mellett látható:

```
[33] PS I:\>$newshare.create


MemberType          : Method
OverloadDefinitions : {System.Management.ManagementBaseObject Create(System
                      .String Path, System.String Name, System.UInt32 Type,
                       System.UInt32 MaximumAllowed, System.String Descript
                      ion, System.String Password, System.Management.Manage
                      mentObject#Win32 SecurityDescriptor Access)}
TypeNameOfValue     : System.Management.Automation.PSMethod
Value               : System.Management.ManagementBaseObject Create(System.
                      String Path, System.String Name, System.UInt32 Type,
                      System.UInt32 MaximumAllowed, System.String Descripti
                      on, System.String Password, System.Management.Managem
                      entObject#Win32_SecurityDescriptor Access)
Name                : Create
IsInstance          : True
```

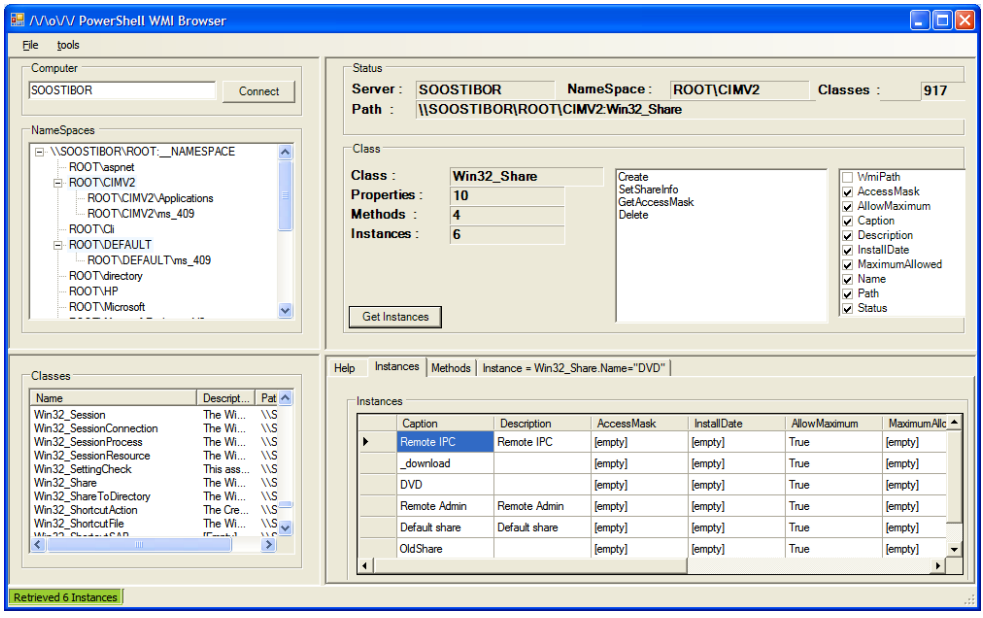
Szerencsére nem minden paramétert kötelező kitölteni, így egy egyszerű megosztás a következő kifejezéssel is létrehozható:

```
[34] PS I:\>$newshare.create("c:\old","OldShare",0)


  GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
  RELPATH        :
  PROPERTY COUNT : 1
  DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
```

Láthatjuk a 0-s visszatérési értékből, hogy a művelet sikeres volt, így kilistázhatjuk az aktuális megosztásokat, melynek végén ott az új megosztásunk:

```
[35] PS I:\>Get-WmiObject Win32_Share

Name                            Path                            Description
----                            ----                            -----------
IPC$                                                            Remote IPC
 download                       C:\ download
DVD                             E:\
ADMIN$                          C:\WINDOWS                      Remote Admin
C$                              C:\                             Default share
OldShare                        c:\old
```

A metódusok böngészésére is alkalmas másik eszköz a *PowerShell WMI Browser*, vagy más néven *WMIExplorer.ps1*:



58. ábra PowerShell WMI Browser

Ez az eszköz azért is érdekes, mert színtisztán PowerShellben van megírva! Érdemes belenézni a szkript forráskódjába, látható benne, hogy hogyan szólítja meg a .NET keretrendszer grafikus osztályait, melyek segítségével felépíti a fenti képen látható ablakot a sok vezérlőelemmel együtt. Persze a PowerShellt nem grafikus alkalmazások írására találták ki, így ilyeneket nem túl egyszerű létrehozni. A PowerShell WMI Browser alkotója, a „The PowerShell Guy” sem Notepad előtt ülve pötyögte be a programsorokat, hanem egy C#-ban megírt programot a forráskód alapján egy szkript segítségével alakított át színtisztán PowerShell szkriptté.

2.6.9 Fontosabb WMI osztályok

A következőkben néhány egyszerűbb példát válogattam össze a WMI legkülönbözőbb területeiről.

Alapvető számítógép-információk kijelzése:

```
[45] PS I:\>get-wmiobject -class "Win32_ComputerSystem" -namespace "root\CIM
V2"
```

BIOS információk:

```
[46] PS I:\>get-wmiobject -class "Win32_BIOS" -namespace "root\CIMV2"
```

Alaplap információk:

```
[47] PS I:\>get-wmiobject -class "Win32_BaseBoard" -namespace "root\CIMV2"
```

Számítógép házának információi (pl. sorozatszám):

```
[48] PS I:\>get-wmiobject -class "Win32 SystemEnclosure" -namespace "root\CI
MV2"
```

Processzor információk:

```
[53] PS I:\>get-wmiobject -class "Win32_Processor" -namespace "root\CIMV2"
```

Részletes memóriainformációk (blokkok szintjén is):

```
[55] PS I:\>get-wmiobject -class "Win32_PhysicalMemory" -namespace "root\CIM
V2"
```

Plug'n'Play eszközök:

```
[59] PS I:\>get-wmiobject -class "Win32_PnPEntity" -namespace "root\CIMV2"
```

Videokártya információk:

```
[61] PS I:\>get-wmiobject -class "Win32_DisplayConfiguration" -namespace "ro
ot\CIMV2"
```

Eventlog állományok:

```
[64] PS I:\>get-wmiobject -class "Win32 NTEventlogFile" -namespace "root\CIM
V2"
```

Eventlog állományok távoli gépről (lásd *2.4 Az Eseménynapló feldolgozása (Get-Eventlog)* fejezet):

```
[64] PS I:\>get-wmiobject -class "Win32_NTEventlogFile" -namespace "root\CIM
V2" -credential IQJB\soostibor -computer J-CRM
```

Hálózati adapterek konfigurációja:

```
[66] PS I:\>get-wmiobject -class "Win32_NetworkAdapterConfiguration" -namesp
ace "root\CIMV2"
```

Login információk:

```
[68] PS I:\>get-wmiobject -class "Win32_NetworkLoginProfile" -namespace "roo
t\CIMV2"
```

A Windows verzóinformációi:

```
[69] PS I:\>get-wmiobject -class "Win32 OperatingSystem" -namespace "root\CI
MV2"
```

Page-file információk:

```
[70] PS I:\>get-wmiobject -class "Win32_PageFile" -namespace "root\CIMV2"
```

Gép helyi ideje:

```
[73] PS I:\>get-wmiobject -class "Win32_LocalTime" -namespace "root\CIMV2"
```

Időzóna:

```
[74] PS I:\>get-wmiobject -class "Win32_TimeZone" -namespace "root\CIMV2"
```

Printer-információk:

```
[76] PS I:\>get-wmiobject -class "Win32_Printer" -namespace "root\CIMV2"
```

Telepített javítócsomagok:

```
[77] PS I:\>get-wmiobject -class "Win32_QuickFixEngineering" -namespace "roo
t\CIMV2"
```

Partíciók:

```
[78] PS I:\>get-wmiobject -class "Win32 DiskPartition" -namespace "root\CIMV
2"
```

Logikai meghajtók:

```
[79] PS I:\>get-wmiobject -class "Win32_LogicalDisk" -namespace "root\CIMV2"
```

Meghajtók:

```
[80] PS I:\>get-wmiobject -class "Win32_DiskDrive" -namespace "root\CIMV2"
```

2.7 Rendszerfolyamatok és szolgáltatások

Korábban már találkozhattunk a `Get-Process` és a `Get-Service` cmdlettel, amelyek segítségével a rendszerfolyamatok és a szolgáltatások listáját kérhetjük le. A folyamatokkal és szolgáltatásokkal kapcsolatban azonban nem csak listázást, hanem bármilyen más feladatot is elvégezhetünk a PowerShell cmdletjeivel, illetve ha minden kötél szakad, közvetlenül a megfelelő .NET komponensek segítségével.

? | Feladat: Indítsuk el, majd állítsuk le a notepad.exe-t!

Először is nézzük meg, hogy milyen cmdleteket használhatunk a folyamatok kezelésére:

```
PS C:\> get-help process

Name                      Category                  Synopsis
----                      --------                  --------
Get-Process               Cmdlet                    Gets the processes th...
Stop-Process              Cmdlet                    Stops one or more run...
```

Mindössze két cmdlet került a listába, a már ismert `Get-Process` és a `Stop-Process`. Nincsen `Start-Process`!? Mindegy, szerencsére már minden eszközünk megvan a helyes megoldáshoz, csak annyit kell tudnunk, hogy a .NET `System.Diagnostics.Process` osztályában van egy statikus `Start()` metódus. Tehát:

```
PS C:\> [System.Diagnostics.Process]::Start("notepad")

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
      0       0      140         92     1     0,02   2980 notepad
```

A megjelenő táblázat egy kicsit furcsa lehet, még visszatérünk rá, hogy mi is ez. Persze lehetne egy kissé egyszerűbben is, hiszen bármilyen végrehajtható fájlt közvetlenül is elindíthatunk a PowerShellből:

```
PS C:\> notepad
```

Ez a megoldás valóban egyszerű, azonban korántsem egyenértékű az előzővel. A notepad persze így is elindul, majd csak a leállításnál lesznek kisebb problémák. Nézzük meg jobban a korábban használt `Start()` metódust:

```
PS C:\> [System.Diagnostics.Process] | get-member -static | format-table
-wrap


   TypeName: System.Diagnostics.Process
```

```
Name                MemberType Definition
----                ---------- ----------
...
Start               Method     static System.Diagnostics.Process Start(Strin
                               g fileName, String userName, SecureString pas
                               sword, String domain), static System.Diagnost
                               ics.Process Start(String fileName, String arg
                               uments, String userName, SecureString passwor
                               d, String domain), static System.Diagnostics.
                               Process Start(String fileName), static System
                               .Diagnostics.Process Start(String fileName, S
                               tring arguments), static System.Diagnostics.P
                               rocess Start(ProcessStartInfo startInfo)
```

Láthatjuk, hogy ez függvény; méghozzá egy `Process` objektumot ad vissza (ennek szöveges reprezentációja jelent meg a konzolon a korábbi parancs hatására!), amelynek segítségével később is kapcsolatban maradhatunk az elindított folyamattal. Lekérdezhetjük például, hogy mennyi processzoridőt, vagy memóriát használ, mikor indult el, stb. És, természetesen, le is állíthatjuk, méghozzá nem csak úgy találomra, hanem pontosan azt a notepadot, amelyet korábban elindítottunk.

Adjuk ki a következő utasítást:

```
PS C:\> stop-process -name "notepad"
```

Mi fog történni? Leállítjuk a notepad nevű folyamatot, de kissé nagyvonalú módszerrel nem csak azt az egyet, amit elindítottunk, hanem az összest. Persze nem csak név, hanem *Process ID* (PID) is megadható a pontos célzáshoz, csak sajnos az egyszerűbb indítás esetén semmit nem tudunk a folyamatról, tehát a PID-jét sem. A feladat teljesen korrekt megoldása tehát a következő:

Indítás (a visszaadott `Process` objektum referenciája belekerül a `$notepad` változóba):

```
PS C:\> $notepad = [System.Diagnostics.Process]::Start("notepad")
```

Leállítás:

```
PS C:\> stop-process –inputobject $notepad
```

? | **Feladat:** Állítsuk le gépünkön az összes leállítható szolgáltatást!

Először is kérnünk kell egy listát a szolgáltatásokról, és ki kell válogatnunk közülük azokat, amelyek leállíthatók, vagyis a `CanStop` tulajdonságuk „igaz” értéket ad vissza:

```
PS C:\> get-service | where-object {$_.CanStop}

Status   Name               DisplayName
------   ----               -----------
Running  ALG                Alkalmazási réteg átjárószolgáltatása
Running  AudioSrv           Windows audió
```

```
Running  BITS                Háttérben futó intelligens átviteli...
Running  Browser             Számítógép-tallózó
...
```

A kiírt táblázatban nem szerepel a `CanStop` érték, így ha nem vagyunk biztosak a dolgunkban, érdemes lehet egy olyan listát kérni, amelyben saját szemünkkel is meggyőződhetünk a helyes eredményről:

```
PS C:\> get-service | where-object {$ .CanStop} | format-table name, canstop
-autosize

Name              CanStop
----              -------
ALG                  True
AudioSrv             True
BITS                 True
Browser              True
...
```

Ezután már csak végig kell lépkednünk a gyűjtemény valamennyi elemén, és mindegyikre meghívni a `Stop()` metódust:

```
PS C:\> get-service | where-object {$_.CanStop} | foreach-object {$_.Stop()}
```

Vigyázat! A szolgáltatások leállítása után valószínűleg újra kell indítanunk a számítógépet.

Amint a bevezetőben már említettük a PowerShell jelenlegi verziója közvetlenül nem támogatja távoli gépek elérését. Elérhetőek azonban ezek a gépek is, ha külső segítséget veszünk igénybe: WMI használatával lekérdezhetők és felügyelhetők a távoli gépeken futó folyamatok és szolgáltatások is. Példaként nézzünk egy szkriptet, amellyel akár távoli gépeken is meghívhatunk WMI metódusokat, jelenleg a szolgáltatások *StopService* metódusát hívja. Ez a szkript a *2.6.8 WMI objektumok metódusainak meghívása* fejezet végén már látott WMI Explorer segítségével készült eredetileg, egy kicsit javítottam és optimalizáltam rajta:

```
# Win32_Service. StartService-Method Template Script"
# Created by PowerShell WmiExplorer
# /\/\o\/\/ 2006
# www.ThePowerShellGuy.com
#

$Computer = "."
$Class = "Win32 Service"
$Method = "StopService"

# Win32_Service. Key Properties :
$Name = [string] "Browser"

$filter="Name = '$Name'"
```

```
$MC = get-WMIObject $class -computer $Computer -Namespace "ROOT\CIMV2" -
filter $filter

"Calling $Class:$Name : $Method "

$R = $mc.PSBase.InvokeMethod($Method,$Null)
"Result :"
$R | Format-list
```

A `$Computer` változóba a cél gép nevét kell beírni, a `$Name` változóba a leállítani kívánt szolgáltatás nevét.

2.7.1.1 Szolgáltatások Startup tulajdonsága

Ha már `get-service`, akkor nézzük meg, hogy az előbb látott hasznos `CanStop` tulajdonság mellett, vajon van-e `StartupType`, vagy valami hasonló tulajdonsága a szolgáltatás-objektumoknak? Nézzük meg a tulajdonság jellegű tagjellemzőket:

```
[13] PS C:\> get-service alerter | get-member -MemberType properties


   TypeName: System.ServiceProcess.ServiceController

Name                MemberType     Definition
----                ----------     ----------
Name                AliasProperty  Name = ServiceName
CanPauseAndContinue Property       System.Boolean CanPauseAndContinue {get;}
CanShutdown         Property       System.Boolean CanShutdown {get;}
CanStop             Property       System.Boolean CanStop {get;}
Container           Property       System.ComponentModel.IContainer Conta...
DependentServices   Property       System.ServiceProcess.ServiceControlle...
DisplayName         Property       System.String DisplayName {get;set;}
MachineName         Property       System.String MachineName {get;set;}
ServiceHandle       Property       System.Runtime.InteropServices.SafeHan...
ServiceName         Property       System.String ServiceName {get;set;}
ServicesDependedOn  Property       System.ServiceProcess.ServiceControlle...
ServiceType         Property       System.ServiceProcess.ServiceType Serv...
Site                Property       System.ComponentModel.ISite Site {get;...
Status              Property       System.ServiceProcess.ServiceControlle...
```

Engem igazából az indulási állapot érdekelne (Automatic, Manual, Disabled). Nézzük meg, hogy pl. a sokat sejtető `ServiceType` vajon ezt rejti-e?

```
[14] PS C:\> get-service alerter | Format-Table -Property Name, Status, Serv
iceType -auto

Name     Status       ServiceType
----     ------       -----------
Alerter Stopped Win32ShareProcess
```

Sajnos nem. Hát ez bosszantó! Annál is inkább, mert a `set-service` cmdlet vidáman tartalmaz erre a tulajdonságra vonatkozó beállítási lehetőséget:

```
[15] PS C:\> get-help Set-Service

NAME
    Set-Service

SYNOPSIS
    Changes the display name, description, or starting mode of a service.


SYNTAX
    Set-Service [-name] <string> [-displayName <string>] [-description <str
    ing>] [-startupType {<Automatic> | <Manual> | <Disabled>}] [-whatIf] [-
    confirm] [<CommonParameters>]
...
```

Na de WMI-ből a `StartupType` adathoz is hozzáférhetünk, ugyan ott `StartMode`-nak hívják:

```
PS I:\> $service=[WMI] "Win32 Service.Name=""Alerter"""
PS I:\> $service.StartMode
Disabled
```

Hogyan lehetne azt elérni, hogy ne kelljen két menetben kiszedni egy szolgáltatás adatait (egy `get-service` PowerShell cmdlet és egy WMI lekérdezés), hanem valahogy ezeket egyben látni? Szerencsére a PowerShell annyira rugalmas, hogy még ezt is meg lehet tenni az *1.5.10 Objektumok testre szabása, kiegészítése* fejezetben látott módon, *types.ps1xml* fájl létrehozásával.

Létrehoztam egy `soost.service.ps1xml` fájlt ugyanazon elérési úton, mint ahol az eredeti `types.ps1xml` is van.

```
<Types>
   <Type>
      <Name>System.ServiceProcess.ServiceController</Name>
      <Members>
        <ScriptProperty>
          <Name>StartupType</Name>
          <GetScriptBlock>
              ([Wmi] "Win32 Service.Name=""$($this.Name)""").StartMode
          </GetScriptBlock>
        </ScriptProperty>
      </Members>
   </Type>
</Types>
```

A szerkezet majdnem magáért beszél. Egyrészt a "`GetScriptBlock`" szorul magyarázatra. Általában egy tulajdonságot kiolvashatunk, esetleg értéket is adhatunk neki. Én itt ebben a fájlban csak kiolvasási viselkedését definiáltam, az értékadásit nem. (Az egyébként a `SetScriptBlock` lenne.)

Másrészt itt nem `$_` változóval hivatkozunk magunkra, hanem a `$this` változóval.

Most már csak be kell etetni a rendszerbe az én típusmódosításomat és már nézhetjük is az eredményt:

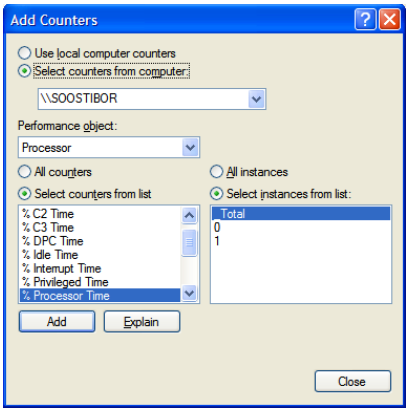
```
PS C:\> Update-TypeData soost.service.ps1xml
PS C:\> (Get-Service alerter).StartupType
Disabled
```

2.8 Teljesítmény-monitorozás

Önmagában a PowerShell teljesítményt sem tud monitorozni, jelenleg nincsenek specifikusan ilyen jellegű cmdletek, viszont a .NET keretrendszer erre a célra is nyújt osztályokat számunkra. A legpraktikusabb osztály a `System.Diagnostics.PerformanceCounter`. Ennek egy általunk felépített objektuma fogja a teljesítményszámlálókat generálni számunkra a következő módon:

```
[32] PS C:\>$perfobj="Processor"
[33] PS C:\>$perfcounter="% Processor Time"
[34] PS C:\>$perfinstance="_Total"
[35] PS C:\>$perftarget="soostibor"
[36] PS C:\>$perfdata = New-Object System.Diagnostics.PerformanceCounter( $p
erfobj, $perfcounter, $perfinstance, $perftarget)
[37] PS C:\>$perfdata.NextValue()
0
[38] PS C:\>$perfdata.NextValue()
1,367338
```

Gyakorlatilag a fenti példában a grafikus *Performace Monitor* alkalmazás alábbi dialógusablakának adatait rakom be különböző változókba:



59. ábra Performance Monitor adatai

Majd a [36]-os sorban hozom létre a teljesítményszámlálót. A mintavétel a `NextValue()` metódus meghívására történik, első esetben ez még nem ad igazi eredményt, csak indítja a mérést, a további hívásokkal már az aktuális értékeket kapjuk meg.

A `New-Object` meghívásánál nem kötelező az összes argumentumot megadni, elég a teljesítmény-objektum és -számláló definiálása, és a többpéldányos számlálóknál a példány (instance) megadása. Ha a célszámítógépet nem definiáljuk, akkor az az aktuális gép megfelelő paraméterét fogja mérni.

2.9 Felhasználó-menedzsment, Active Directory

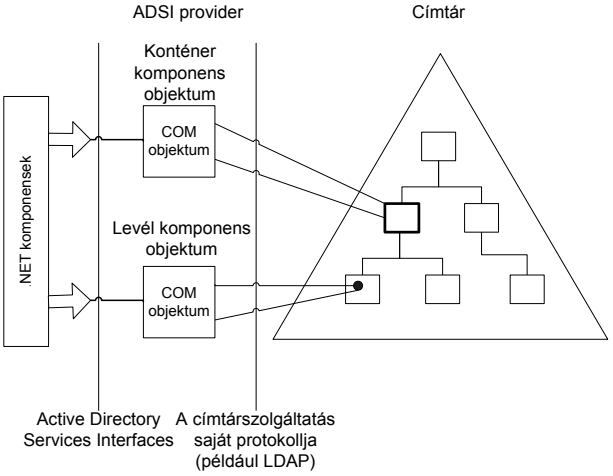
2.9.1 Mi is az ADSI?

Az ADSI segítségével a különféle címtárakat egységes formában kezelhetjük, mivel az általa biztosított csatolófelületek felhasználásával alkalmazásunk a különféle szerkezetű címtárakat is azonos módon érheti el. Az ADSI hasonló szerepet tölt be a címtárak elérésében, mint az ODBC a különféle adatbázis-kiszolgálók esetében. Segítségével a címtárakkal kapcsolatos gyakori feladatok – felhasználók kezelése, különféle erőforrások keresése – az elosztott, több címtármegoldást használó környezetekben is könnyen elvégezhetőek.

Az ADSI objektumai valójában COM objektumok, amelyek a kezelni kívánt címtár objektumait jelenítik meg, és COM csatolófelületeken keresztül érhetőek el. Az objektumok két csoportba sorolhatók; a konténer típusú objektumok más konténereket, és levél típusú objektumokat tartalmazhatnak.

2.9.2 ADSI providerek

Az ADSI providerek valósítják meg a konkrét címtártípus eléréséhez szükséges funkciókat. Amint az alábbi ábrán is látható, az ADSI-t használó ügyfeleknek (így szkriptjeinknek, vagy a .NET Framework komponenseinek is) csak az ADSI provider által biztosított csatolófelület megfelelő használatával kell törődniük, a háttérben lévő címtár egyedi adottságainak megfelelő műveletek kezdeményezése már a provider feladata.



60. ábra Címtár elérése ADSI provideren keresztül

Az ADSI által biztosított legfontosabb providerek a következők:

WinNT:// – a provider segítségével a Windows NT 4.0 tartományokhoz férhetünk hozzá, és ez teszi lehetővé az önálló (Windows NT, 2000, XP) számítógépeken levő helyi címtár-adatbázisokhoz (SAM) való hozzáférést is. Meg kell jegyeznünk, hogy a WinNT provider kompatibilitási okokból használható Active Directory esetében is, de ez számos korlátozással jár (a keresés nem támogatott, szervezeti egységek megadása nem lehetséges, stb.) ezért használata nem célszerű.

LDAP:// – ez a provider az LDAP protokoll segítségével elérhető címtárak (például az Active Directory) kezelésére szolgál.

NDS:// – segítségével a Novell Directory Services szolgáltatáshoz férhetünk hozzá.

2.9.3 Az ADSI gyorsítótár

Minden ADSI objektum ügyféloldali attribútum-gyorsítótárral rendelkezik, amelyben ideiglenesen tárolódnak az objektumhoz tartozó tulajdonságok nevei és értékei. A gyorsítótár jelentősen növeli az attribútumokkal végzett különféle műveletek teljesítményét, mivel nélküle programunknak minden egyes attribútum értékének beállítása után a címtárhoz kéne fordulnia.

2.9.4 Active Directory információk lekérdezése

Az elméleti áttekintés után nézzük élesben a címtárak elérését PowerShell segítségével, mellyel az Active Directory-val kapcsolatos felügyeleti tevékenységek is hatékonyan automatizálhatók. De még mielőtt ebbe beleásnánk magunkat, nézzünk pár előkészítő tevékenységet. Miután a PowerShell alatt a .NET keretrendszer található, így fontos ismerni a címtárak elérésének problémáit diagnosztizálni képes fontosabb osztályokat.

Miután az Active Directory hibák 110%-a (☺) valamilyen DNS hibára vezethető vissza, így elsőként ellenőrizzük a névfeloldást. Az ezzel kapcsolatos .NET osztály a `System.Net.Dns`, melynek `GetHostEntry` statikus metódusával tudjuk ellenőrizni például a tartományvezérlőnk nevének feloldását:

```
PS C:\Users\Administrator> [System.Net.Dns]::GetHostEntry("adds.iqjb.w08")

HostName                Aliases                 AddressList
--------                -------                 -----------
adds.iqjb.w08           {}                      {192.168.1.2}
```

Ha ez helyes eredményt ad, akkor folytathatjuk az AD felderítését, ellenőrzését az erdő legfontosabb objektumaival. Erre a célra a `System.DirectoryServices.ActiveDirectory.Forest` osztály alkalmas, annak is a `GetCurrentForest` statikus metódusa:

```
PS C:\Users\Administrator> [System.DirectoryServices.ActiveDirectory.Forest]
::getcurrentforest()
```

```
Name                  : iqjb.w08
Sites                 : {Default-First-Site-Name}
Domains               : {iqjb.w08}
GlobalCatalogs        : {adds.iqjb.w08}
ApplicationPartitions : {DC=ForestDnsZones,DC=iqjb,DC=w08, DC=DomainDnsZone
                        s,DC=iqjb,DC=w08}
ForestMode            : Windows2008Forest
RootDomain            : iqjb.w08
Schema                : CN=Schema,CN=Configuration,DC=iqjb,DC=w08
SchemaRoleOwner       : adds.iqjb.w08
NamingRoleOwner       : adds.iqjb.w08
```

Láthatjuk, hogy ez a metódus a legfontosabb adatokat megadja az erdőről: a *root* tartomány és a többi tartomány nevét, a globális katalógusok listáját, az erdő működési szintjét és az erdőszintű egyedi szerepeket hordozó tartományvezérlők neveit.

Hasonló módon megvizsgálhatjuk az aktuális tartomány adatait is a `System.DirectoryServices.ActiveDirectory.Domain` osztály `getcurrentdomain` statikus metódusa segítségével:

```
PS C:\Users\Administrator> [System.DirectoryServices.ActiveDirectory.Domain]
::getcurrentdomain()


Forest                  : iqjb.w08
DomainControllers       : {adds.iqjb.w08}
Children                : {}
DomainMode              : Windows2008Domain
Parent                  :
PdcRoleOwner            : adds.iqjb.w08
RidRoleOwner            : adds.iqjb.w08
InfrastructureRoleOwner : adds.iqjb.w08
Name                    : iqjb.w08
```

Itt a legfontosabb plusz információ a tartományi szintű egyedi szerepeket hordozó tartományvezérlők nevei.

Nem mellékes, hogy milyen AD site (telephely) beállításokkal dolgozunk, hiszen ez befolyásolja az ügyfél gépek tartományvezérlő választását és a címtár replikációt. A telephely információk kiolvasására a `System.DirectoryServices.ActiveDirectory.ActiveDirectorySite` osztály `GetComputerSite` statikus metódusa használható:

```
PS C:\> [System.DirectoryServices.ActiveDirectory.ActiveDirectorySite]::GetC
omputerSite()


Name                          : Default-First-Site-Name
Domains                       : {iqjb.w08}
Subnets                       : {192.168.112.0/24}
Servers                       : {w2k8.iqjb.w08}
AdjacentSites                 : {}
```

```
SiteLinks                       : {DEFAULTIPSITELINK}
InterSiteTopologyGenerator      : w2k8.iqjb.w08
Options                         : None
Location                        : Budapest
BridgeheadServers               : {}
PreferredSmtpBridgeheadServers  : {}
PreferredRpcBridgeheadServers   : {}
IntraSiteReplicationSchedule    : System.DirectoryServices.ActiveDirectory.A
                                  ctiveDirectorySchedule
```

Ezzel a néhány kifejezéssel tehát elég jól át lehet tekinteni, hogy milyen az AD infrastruktúránk, fájlba történő átirányítással akár az AD rendszerünk dokumentálásához is segítséget kapunk.

2.9.5 Csatlakozás az Active Directory-hoz

Az előzőekben a .NET keretrendszer osztályainak statikus metódusaival dolgoztam, melyek segítségével általános információkat lehetett kinyerni az Active Directory környezetről. Ha konkrét, adott tartományra vagy címtárelemre vonatkozó információkhoz akarunk hozzájutni, akkor csatlakozni kell az adott címtár objektumhoz. Az előzőekben leírtuk, hogy a címtár kezelésében fontos szerepe van a gyorsítótárnak, azaz egy ilyen csatlakoztatás előkészíti a memóriában az adott címtár objektum egy reprezentációját. Ha ezek után változtatunk az objektum valamely tulajdonságán, akkor ez csak a memóriában hajtódik végre, külön metódussal kell ezt a változást a címtárba visszaírni, mint ahogy ezt látni fogjuk.

Elsőként azonban nézzük meg a legegyszerűbb csatlakoztatást:

```
PS C:\> $domain = [ADSI] ""
```

Az `[ADSI]` típusjelölővel hivatkozunk a címtáras elérésre, és ha egy üres sztiringet adunk meg a „konstruktor” paramétereként, akkor az aktuális tartományi objektumhoz csatlakozunk. Ez a típusjelölő a `System.DirectoryServices.DirectoryEntry` .NET osztály rövidített neve. Ezt akár ki is írhatjuk, és ekkor további paramétereket is megadhatunk, ha szükséges:

```
PS C:\> $domain = new-object DirectoryServices.DirectoryEntry("","iqjb\Admin
istrator", "Password1")
```

Olvassuk ki, hogy mi került a `$domain` változónkba:

```
PS C:\> $domain

distinguishedName
-----------------
{DC=iqjb,DC=w08}
```

Ez még nem túl sok, nézzük meg a tagjellemzőit:

```
PS C:\> $domain | Get-Member


   TypeName: System.DirectoryServices.DirectoryEntry

Name                                MemberType Definition
----                                ---------- ----------
auditingPolicy                      Property   System.DirectoryServices.Pro...
creationTime                        Property   System.DirectoryServices.Pro...
dc                                  Property   System.DirectoryServices.Pro...
distinguishedName                   Property   System.DirectoryServices.Pro...
dSCorePropagationData               Property   System.DirectoryServices.Pro...
forceLogoff                         Property   System.DirectoryServices.Pro...
fSMORoleOwner                       Property   System.DirectoryServices.Pro...
gPLink                              Property   System.DirectoryServices.Pro...
instanceType                        Property   System.DirectoryServices.Pro...
isCriticalSystemObject              Property   System.DirectoryServices.Pro...
lockoutDuration                     Property   System.DirectoryServices.Pro...
lockOutObservationWindow            Property   System.DirectoryServices.Pro...
...
uSNChanged                          Property   System.DirectoryServices.Pro...
uSNCreated                          Property   System.DirectoryServices.Pro...
wellKnownObjects                    Property   System.DirectoryServices.Pro...
whenChanged                         Property   System.DirectoryServices.Pro...
whenCreated                         Property   System.DirectoryServices.Pro...
```

Kicsit megvágtam, de ebből is látszik, hogy jó néhány tulajdonsága van egy ilyen objektumnak. Nézzük, hogy hogyan tudunk egy nevesített objektumhoz, mondjuk egy felhasználói fiókhoz csatlakozni:

```
PS C:\> $user = [ADSI] "LDAP://cn=János Vegetári,ou=Demó,dc=iqjb,dc=w08"
PS C:\> $user

distinguishedName
-----------------
{CN=János Vegetári,OU=Demó,DC=iqjb,DC=w08}
```

Fontos!

Az `[ADSI]` utáni sztringben csupa nagybetűs az LDAP, és normál perjelek vannak utána. Ha nem csupa nagybetűs az LDAP, vagy fordított perjelet használunk, akkor nem rögtön kapunk hibajelzést, hanem akkor, amikor először használjuk az objektumot.

A későbbiekben majd azt is megnézzük, hogy egy ilyen felhasználói fióknak milyen tulajdonságai vannak és hogyan lehet azokat módosítani.

2.9.6 AD objektumok létrehozása

AD objektumokat létrehozni a VBScriptben megszokott (már aki programozott VBScriptben) ADSI szintaxishoz nagyon hasonló módon lehet. Először egy AD konténerre

kell csatlakozni, ahova létre szeretnénk hozni az új objektumot. Ez a csatlakozás az előző fejezetben látott módon megy, ezzel ugye „átemeljük” a PowerShellbe az adott konténert, mint objektumot.

Az így „átemelt” AD objektum `Create` metódusával lehet létrehozni az új AD elemet. A `Create` paramétereként meg kell adni a létrehozandó objektum típusát és az u.n. „*relative distinguished name*” nevét, azaz az adott konténeren belüli megkülönböztető nevét.

Az alábbi példában magán a tartományvezérlőn (`localhost`), közvetlenül a tartomány objektum alá hozok létre egy szervezeti egységet (`organizational unit`):

```
PS C:\> $konténer = [adsi] "LDAP://localhost:389/dc=iqjb,dc=w08"
PS C:\> $adObj = $konténer.Create("OrganizationalUnit", "OU=Emberek")
PS C:\> $adObj.Put("Description", "Normál felhasználók")
PS C:\> $adObj.SetInfo()
```

Ebben ugye az az újdonság, hogy az LDAP kifejezésbe beillesztettem a tartományvezérlő nevét és a portszámot, ahol a címtárszolgáltatás elérhető. A szemléltetés kedvéért még a „`Description`” attribútumát is kitöltöttem. Vigyázat, amit idáig tettem azt mind a memóriában tárolódó gyorsítótárban végeztem el, ahhoz, hogy mindez ténylegesen bekerüljön a címtár adatbázisba meg kell hívni a `SetInfo` metódust!

2.9.7 AD objektumok tulajdonságainak kiolvasása, módosítása

Ha PowerShell, akkor objektumok. Az előzőekhez hasonlóan csatlakozzunk a most létrehozott szervezeti egység objektumhoz, és nézzük meg a tagjellemzőit a `get-member` cmdlet segítségével:

```
PS C:\> $adou = [ADSI] "LDAP://OU=Emberek,DC=iqjb,DC=w08"
PS C:\> $adou

distinguishedName
-----------------
{OU=Emberek,DC=iqjb,DC=w08}


PS C:\> $adou | get-member


   TypeName: System.DirectoryServices.DirectoryEntry

Name                 MemberType Definition
----                 ---------- ----------
description          Property   System.DirectoryServices.PropertyValueC...
distinguishedName    Property   System.DirectoryServices.PropertyValueC...
dSCorePropagationData Property   System.DirectoryServices.PropertyValueC...
instanceType         Property   System.DirectoryServices.PropertyValueC...
name                 Property   System.DirectoryServices.PropertyValueC...
```

```
nTSecurityDescriptor  Property   System.DirectoryServices.PropertyValueC...
objectCategory        Property   System.DirectoryServices.PropertyValueC...
objectClass           Property   System.DirectoryServices.PropertyValueC...
objectGUID            Property   System.DirectoryServices.PropertyValueC...
ou                    Property   System.DirectoryServices.PropertyValueC...
uSNChanged            Property   System.DirectoryServices.PropertyValueC...
uSNCreated            Property   System.DirectoryServices.PropertyValueC...
whenChanged           Property   System.DirectoryServices.PropertyValueC...
whenCreated           Property   System.DirectoryServices.PropertyValueC...
```

Hát elég furcsa, amit kaptunk. Látjuk a szervezeti egységünk tulajdonságait, de hol vannak a metódusok? Hol a `Create`? Sajnos a PowerShell 1.0-ba még nincsen 100%-ban adaptálva a `System.DirectoryServices` osztály. Ennek több oka van. Az egyik, hogy valójában itt nem színtiszta .NET osztályról van szó, hanem COM objektum is meghúzódik a felszín alatt, és annak metódusait nem olyan egyszerű átemelni. Gondoljunk csak arra, hogy egy ilyen `DirectoryEntry` típusú objektum lehet felhasználói fiók, számítógép fiók, telephely, csoport, stb., ezeknek mind más és más metódusuk van, ezeknek az adaptálása a PowerShell környezetbe nem olyan egyszerű. Ebből származik a második ok, ami miatt ez nincs adaptálva, az pedig az, hogy a fejlesztők az 1.0 megjelenését nem akarták ezzel késleltetni, várhatóan a 2.0 verzió már precízebb AD támogatást fog nyújtani.

Szerencsére van egy kis menekvési ösvényünk, azaz kikapcsolhatjuk a PowerShell adaptációs rétegét, és megnézhetjük a „színtiszta" .NET objektumot is, ha a `psbase` nézeten keresztül nézzük az objektumunkat:

```
PS C:\> $adou.psbase | get-member


   TypeName: System.Management.Automation.PSMemberSet

Name                      MemberType Definition
----                      ---------- ----------
...
MoveTo                    Method     System.Void MoveTo(DirectoryEntry n...
RefreshCache              Method     System.Void RefreshCache(), System....
remove_Disposed           Method     System.Void remove_Disposed(EventHa...
Rename                    Method     System.Void Rename(String newName)
...
ToString                  Method     System.String ToString()
AuthenticationType        Property   System.DirectoryServices.Authentica...
Children                  Property   System.DirectoryServices.DirectoryE...
Container                 Property   System.ComponentModel.IContainer Co...
Guid                      Property   System.Guid Guid {get;}
Name                      Property   System.String Name {get;}
NativeGuid                Property   System.String NativeGuid {get;}
NativeObject              Property   System.Object NativeObject {get;}
ObjectSecurity            Property   System.DirectoryServices.ActiveDire...
Options                   Property   System.DirectoryServices.DirectoryE...
Parent                    Property   System.DirectoryServices.DirectoryE...
Password                  Property   System.String Password {set;}
Path                      Property   System.String Path {get;set;}
```

```
Properties                Property   System.DirectoryServices.PropertyCo...
SchemaClassName           Property   System.String SchemaClassName {get;}
SchemaEntry               Property   System.DirectoryServices.DirectoryE...
Site                      Property   System.ComponentModel.ISite Site {g...
UsePropertyCache          Property   System.Boolean UsePropertyCache {ge...
Username                  Property   System.String Username {get;set;}
```

A fenti, kicsit megvágott, de még így is hosszú listából látszik, hogy az objektumot valójában lehet például mozgatni, átnevezni, és néhány újabb tulajdonság is feltárul a szemünk előtt. De például még mindig nem látjuk a `SetInfo` és a `Create` metódust, mert ezek az ADSI COM interfészből jönnek, és a .NET nem kérdezi ezeket le, így nem is mutatja meg, viszont meghívni, használni lehet őket.

Vagy nézzük a következőket:

```
PS C:\> $d = [ADSI] ""
PS C:\> $d

distinguishedName
-----------------
{DC=iqjb,DC=w08}
```

A fenti módon például nagyon egyszerűen lehet az aktuális tartományunkhoz csatlakozni. Próbáljuk meg ennek megnézni a „rejtett” `children` tulajdonságát:

```
PS C:\> $adou = [ADSI] "LDAP://OU=Demó,DC=iqjb,DC=w08"
PS C:\> $adou.psbase.Children

distinguishedName
-----------------
{CN=Csilla Fájdalom,OU=Demó,DC=iqjb,DC=w08}
{CN=Csoport,OU=Demó,DC=iqjb,DC=w08}
{CN=group1,OU=Demó,DC=iqjb,DC=w08}
{CN=János Vegetári,OU=Demó,DC=iqjb,DC=w08}
{CN=Márton Beléd,OU=Demó,DC=iqjb,DC=w08}
```

Hiszen ez megadta az adott konténer objektumban található al-objektumokat!

Megjegyzés

A lokális gép esetében a `PSBase` egészen extrém információkat rejt el:

```
[40] PS C:\> $computer = [ADSI]"WinNT://."
[41] PS C:\> $v = $computer.psbase.children | ForEach-Object {$_}
[42] PS C:\> $v[0].name; $v[0].psbase.schemaclassname
Administrator
User
[43] PS C:\> $v[30].name; $v[30].psbase.schemaclassname
BITS
Service
```

A [40]-es sorban hozzákapcsolódom a helyi géphez egy `$computer` változón keresztül. Majd betöltöm egy `$v` változóba ennek a `psbase` által feltárt `children` tulajdonságának elemeit egy ciklussal. Ennek 0. elemének `name` tulajdonságát nézve kiderül, hogy ez egy helyi felhasználó, az Administrator. Ennek típusát is ki lehet olvasni egy újabb `psbase` feltárás után a `schemaclassname` tulajdonsággal.
A meglepetés akkor jön, ha egy jóval későbbi elemet nézünk meg, mondjuk a 90.-et. Nálam ez meg egy szolgáltatás volt, a BITS (a kedves olvasónál lehet, hogy ez valami más). Azaz a lokális gép esetében a `children` a gépnek nagyon sok jellemzőjét megadja. Az eligazodást segíti a `Find` metódus:

```
[47] PS C:\> $service = $computer.psbase.Children.Find("Alerter")
[48] PS C:\> $service.serviceaccountname
NT AUTHORITY\LocalService
```

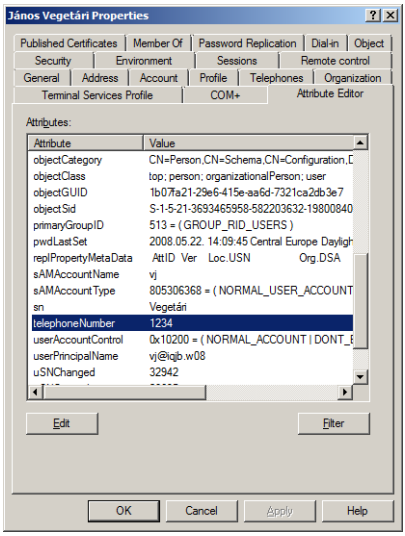
Miután itt vegyes elemekkel dolgozunk, itt különösen jól jön a `get-member` cmdlet.

Mindebből az következik, hogy nem érdemes még kidobni korábbi ADSI ismereteinket, illetve ismernünk kell az AD objektumok tulajdonságainak neveit, hogy ezeket a tulajdonságokat módosíthassuk. Nézzünk ez utóbbira példát egy felhasználói fiókkal kapcsolatban. Van egy már létező felhasználóm, annak szeretném kiolvasni és beállítani a telefonszám tulajdonságát. Ehhez kell nekünk az, hogy tudjuk, hogy az AD-ben a telefonszám tulajdonságnak mi is a belső elnevezése. Ennek felderítésére több módszer van, nézzünk egy PowerShell-est:

```
PS C:\> $user = [ADSI] "LDAP://cn=János Vegetári,OU=Demó,DC=iqjb,DC=w08"
PS C:\> $user.psbase.properties

PropertyName       Value                      Capacity              Count
------------       -----                      --------              -----
objectClass        {top, person, o...                4                  4
cn                 János Vegetári                    4                  1
sn                 Vegetári                          4                  1
telephoneNumber    2008                              4                  1
givenName          János                             4                  1
distinguishedName  CN=János Vegetá...                4                  1
...
```

A fenti listában látjuk, hogy a telefonszám attribútum neve - meglepő módon – `telephoneNumber`.

Vagy a Windows Server 2008 esetében az *Active Directory Users and Computers* eszköz szerencsére már tartalmaz egy *Attribute Editor* fület, a korábbi Windows változatoknál az *ADSIEdit* eszközzel érhetjük el ugyanezt:



61. ábra A Windows Server 2008 ADUC új Attribute Editor-a

Nézzük meg, hogyan lehet ezt a telefonszámot kiolvasni, majd átírni. Az első megoldás a „PowerShell-es”:

```
PS C:\> $user.telephoneNumber
1234
PS C:\> $user.telephoneNumber=2008
PS C:\> $user.setinfo()
PS C:\> $user.telephoneNumber
2008
```

Ebben az a furcsa, hogy a `Get-Member`-rel nem lehetett kiolvasni, hogy a `$user`-nek van `telephoneNumber` tulajdonsága, mégis lehet használni.

A második megoldás a hagyományos, ADSI-stílus:

```
PS C:\> $u.Get("telephoneNumber")
1234
PS C:\> $u.Put("telephoneNumber",9876)
PS C:\> $u.SetInfo()
PS C:\> $u.Get("telephoneNumber")
9876
```

A `Get` metódussal tudjuk az adott tulajdonságot kiolvasni, a `Put`-tal átírni. Egyik esetben sem szabad megfeledkezni a `SetInfo`-ról, ami az objektum memóriabeli reprezentációját írja be ténylegesen az AD-ba.

A `Get` és a `Put` is „rejtett” metódus, a paraméterezésük a példában látható. Az SD attribútumok LDAP nevére kell hivatkozni mindkettőnél.

Megjegyzés

Sajnos nem minden attribútum kezelhető a PowerShell-es módszerrel. Ilyen például a *Company* attribútum:

```
PS C:\> $u.company
PS C:\> $u.get("company")
Cég
```

Az első esetben nem kaptam semmilyen választ az attribútum kiolvasására, de `get`-tel mégis működött.

Mi van akkor, ha kiolvastuk egy felhasználó adatait egy változóba, majd ezután valaki egy másik gépről vagy egy másik alkalmazással módosítja a felhasználónk valamely attribútumát. Ilyenkor a `getinfo` metódussal lehet frissíttetni az adatokat a memóriában:

```
PS C:\> $u.get("company")
Egyik
PS C:\> $u.getinfo()
PS C:\> $u.get("company")
Másik
```

A fenti példában az első kiolvasás után az ADUC eszközzel az első `get` után átírtam a felhasználó *Company* attribútumát, és a `getinfo`-val ezt frissítettem a memóriában, így az új érték már a PowerShell-ből is látszik.

2.9.7.1 Munka többértékű (multivalued) attribútumokkal

Az Active Directory egyik jellegzetes attribútuma az u.n. „multivalued property”. Ez olyan tulajdonság, ahova az értékek listáját, tömbjét tehetjük. Legtipikusabb ilyen attribútum az Exchange Server bevezetése után a felhasználók e-mail címeit tartalmazó `ProxyAddresses`, vagy a csoportok `Member` attribútuma, de erről majd külön értekezek. Marad mondjuk az `other`... kezdetű különböző telefonszámok tárolására szolgáló attribútumok, mint például az `otherMobile` vagy az `otherTelephone`.

Ezeket ki lehet olvasni az eddig megismert módszerekkel is, de nézzük, hogy milyen problémákkal szembesülhetünk. Ha a `get` metódust használom, és csak egy értéket tárol a „multivalued propery”, akkor nem egy elemű tömböt kapok, hanem sima skaláris értéket:

```
PS C:\> $user.get("otherMobile")
othermobil1234
PS C:\> $user.get("otherMobile").gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object
```

Ezzel szemben, ha több értéket tárolunk, akkor már tömböt kapunk:

```
PS C:\> $user.getinfo()
PS C:\> $user.get("otherMobile")
othermobil2345
othermobil1234
PS C:\> $user.get("otherMobile").gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

Ez nem biztos, hogy jó nekünk, mert így a szkriptünket kétfajta esetre kell felkészítenünk: külön arra az esetre, ha csak egy értéket tárolunk és külön arra az esetre is, ha többet. Ez bonyolítja a programjainkat.

Ha konzisztensen, mindig tömbként akarjuk kezelni az ilyen „multivalued property”-ket, akkor vagy használjuk a PowerShell-es stílust:

```
PS C:\> $user.otherMobile
othermobil1234
PS C:\> $user.otherMobile[0]
othermobil1234
PS C:\> $user.otherMobile.gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    PropertyValueCollection                  System.Collec...
```

Vagy használjuk a `GetEx` metódust:

```
PS C:\> $user.getex("otherMobile")
othermobil1234
PS C:\> $user.getex("otherMobile").gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

Az ilyen multivalued property-k írása sem egyértelmű, hiszen több lehetőségem is van:

- a meglevő értékekhez akarok egy újabbat hozzáfűzni,
- a meglevő értékek helyett akarok egy újat betölteni.

Ezeket a lehetőségeket én magam is le tudom programozni a szkriptemben. Ha az első változatra van szükségem, akkor előbb kiolvasom az attribútum aktuális tartalmát egy változóba, hozzárakom az új értéket és így rakom vissza a `put`-tal, vagy egyszerű értékadással. Ha pedig a második változatra van szükségem, akkor egyszerűen felülírom az attribútumot az új értékkel.

Sokkal elegánsabb, ha ezt már maga az objektum tudná egy „okosabb” metódussal. Ilyen létezik, ez pedig a `PutEx`:

```
PS C:\> $user.getex("otherMobile")
othermobil1234
PS C:\> $user.putex(3,"otherMobile",@("othermobilPutEx2"));$user.setinfo()
PS C:\> $user.getex("otherMobile")
othermobilPutEx2
othermobil1234
PS C:\> $user.putex(2,"otherMobile",@("othermobilPutEx3"));$user.setinfo()
PS C:\> $user.getex("otherMobile")
othermobilPutEx3
```

A fenti példában a kiinduló állapotban egy mobilszámunk van. Ezután hozzáfűzök egy újabbat a `putex` használatával, a hozzáfűzést az első paraméterként szereplő 3-as jelzi. Fontos, hogy a hozzáfűzendő értéket tömbként kell kezelni, ezért van ott a kukac-zárójelpár! Ezután egy újabb `putex`-et hívok meg, immár 2-es paraméterrel, ez a felülírás művelete, ennek hatására már csak ez a legújabb mobilszám lesz az attribútumban.

Használhatom még az 1-es paramétert is, ez ekvivalens az attribútum értékeinek törlésével, vagy használhatom a 4-es paramétert, ez a paraméterként megadott elemet töröli az értékek közül:

```
PS C:\> $user.putex(3,"otherMobile",@("Append"));$user.setinfo()
PS C:\> $user.getex("otherMobile")
Append
othermobilPutEx3
PS C:\> $user.putex(4,"otherMobile",@("Append"));$user.setinfo()
PS C:\> $user.getex("otherMobile")
othermobilPutEx3
```

A fenti példában elsőként hozzáfűzök egy értéket, majd ugyanezt eltávolítom.

2.9.7.2 Speciális tulajdonságok kezelése

Van néhány attribútum, amelyek az eddig megismert módszerek egyikével sem kezelhetők:

```
PS C:\> $user.AccountDisabled
PS C:\> $user.get("AccountDisabled")
Exception calling "get" with "1" argument(s): "The directory property canno
t be found in the cache.
"
At line:1 char:10
+ $user.get( <<<< "AccountDisabled")
```

A PowerShelles szintaxis meg se nyikkan, a `get` meg még hibát is jelez. Ilyen esetekben használhatjuk a `psbase` nézeten keresztül az `InvokeGet` és `InvokeSet` metódusokat:

```
PS C:\> $user.psbase.invokeget("AccountDisabled")
False
PS C:\> $user.psbase.invokeset("AccountDisabled","TRUE")
PS C:\> $user.SetInfo()
```

```
PS C:\> $user.psbase.invokeget("AccountDisabled")
True
```

2.9.8 Jelszó megváltoztatása

Speciális attribútum a jelszó, hiszen tudjuk, hogy valójában nem (feltétlenül) tárolja a címtár a jelszavakat, hanem csak a belőlük képzett hasht. Így a jelszó kezelésekor nem egyszerűen egy attribútumot kell beállítani, hanem ezt a hasht kell képezni. Szerencsére erre a célra rendelkezésünkre áll két metódus, a `SetPassword`, illetve a `ChangePassword`:

```
PS C:\> $user.SetPassword("UjPass2")
PS C:\> $user.ChangePassword("UjPass2","MégújabbPass3")
```

A `SetPassword` felel meg a *Reset Password* műveletnek. Ezt ugye csak megfelelő rendszergazda jogosultságok birtokában tudjuk megtenni. A `ChangePassword` a meglevő jelszó birtokában módosítja a jelszót, ehhez már nem kell külön rendszergazda jogosultság.

2.9.9 Csoportok kezelése

Az Active Directory-ban csoportokat leginkább a rendszer üzemeltetésének megkönnyítésére veszünk fel. Segítségükkel osztunk ki hozzáférési jogokat, felhasználói jogokat, de még a csoportos házirendek érvényre jutását is szabályozhatjuk csoportokkal. Miután ilyen széleskörű a felhasználásuk, így fontos lehet a csoportok kezelésének automatizálása. Erre is kiválóan alkalmas a PowerShell, nézzük meg a leggyakoribb műveleteket.

Csoportot létrehozni a már látott módszerrel lehet:

```
PS C:\> $target = [ADSI] "LDAP://ou=Demó,DC=iqjb,DC=w08"
PS C:\> $group = $target.create("group","CN=Csoport")
PS C:\> $group.setinfo()
```

Ez alaphelyzetben globális biztonsági csoportot hoz létre. A későbbi, összetett példában majd bemutatom, hogy hogyan lehet másfajta csoportokat létrehozni.

Ezután kétféleképpen lehet tagokat adni a csoportokhoz. Az első módszer a hagyományos „ADSI”-s módszer, ahol a csoport `Add` metódusát hívom meg, paramétereként a berakni kívánt felhasználó LDAP-os szintaxisú elérési útját kell megadni. Vagy, ha már megragadtam a felhasználói fiókot, akkor vissza kell alakítani az LDAP-os elérési úttá, mint ahogy ebben a példában tettem:

```
PS C:\> $user = [ADSI] "LDAP://CN=János Vegetári,OU=Demó,DC=iqjb,DC=w08"
PS C:\> $group.add("LDAP://$($user.distinguishedname)")
PS C:\> $group.setinfo()
```

Hasonlóan lehet tagot eltávolítani, csak az `Add` helyett a `Remove` metódust kell meghívni.

A második módszer kicsit PowerShell-szerűbb, itt nem kell ide-oda alakítgatni, elég a felhasználó `distinguishedname` tulajdonságát használni:

```
PS C:\> $user = [ADSI] "LDAP://CN=Csilla Fájdalom,OU=Demó,DC=iqjb,DC=w08"
PS C:\> $group.member += $user.distinguishedname
PS C:\> $group.setinfo()
```

Természetesen a két megoldás egyenértékű, csak stílusbeli különbség van közöttük. A második módszer hátránya talán, hogy egyszerűen nem lehet csoporttagot eltávolítani, külön képezni kellene a nemkívánatos tag nélküli tömböt, és azt betölteni a csoport `member` tulajdonságába.

2.9.10 Keresés az AD-ben

A következő gyakori feladat a keresés az AD-ben. Az első módszer a `Find` metódus használata, ami szintén a `PSBase` nézetben érhető el:

```
PS C:\> $ou = [ADSI] "LDAP://ou=Demó,dc=iqjb,dc=w08"
PS C:\> $ou.psbase.children.find("cn=Csilla Fájdalom")

distinguishedName
-----------------
{CN=Csilla Fájdalom,OU=Demó,DC=iqjb,DC=w08}
```

Ez a keresés azonban csak az adott konténerobjektum gyerekei között keres, így nem biztos, hogy igazán jó segítség ez nekünk

Az igazán profi keresőhöz a .NET keretrendszer egyik osztályát, a `System.DirectoryServices.DirectorySearcher`-t hívjuk segítségül, ennek egy objektuma lesz a keresőnk, és ennek különböző tulajdonságait beállítva adjuk meg a keresésünk mindenféle feltételét. Elsőként nézzünk egy nagyon egyszerű feladatot, egy konkrét felhasználói fiókra keressünk rá:

```
[6] PS I:\>$objRoot = [ADSI] "LDAP://OU=IQJB,DC=dom"
[7] PS I:\>$objSearcher = New-Object System.DirectoryServices.DirectorySearc
her
[8] PS I:\>$objSearcher.SearchRoot = $objRoot
[9] PS I:\>$objSearcher.Filter = "(&(objectCategory=user)(displayName=Soós
Tibor))"
[10] PS I:\>$objSearcher.SearchScope = "Subtree"
[11] PS I:\>$colResults = $objSearcher.FindAll()
[12] PS I:\>$colresults

Path                                    Properties
----                                    ----------
LDAP://CN=Soós Tibor,OU=Normal,OU=... {homemdb, distinguishedname, count...
```

Elsőként definiálom, hogy az AD adatbázis-elemek fastruktúrájában, hol is keresek majd (`$objRoot`). Majd elkészítem a keresőt (`$objSearcher`), aminek `SearchRoot` tulajdonságaként az előbb létrehozott keresési helyet adom meg. Majd definiálom az LDAP formátumú szűrőt, amely ebben az esetben a „Soós Tibor” nevű felhasználókat jelenti, és ezt betöltöm a kereső `Filter` tulajdonságaként. LDAP szűrőben a következő összehasonlító operátorokat használhatok:

Operátor	Jelentés
=	Egyenlő
~=	Közel egyenlő
<=	Kisebb egyenlő
>=	Nagyobb egyenlő
&	És
|	Vagy
!	Nem

Végül meghatározom a keresés mélységét, ami itt `Subtree`, azaz mélységi, mert nem pont közvetlenül a kiindulópontként megadott helyen van a keresett objektum. Nincs más hátra, ezek alapján ki kell listázni a feltételeknek megfelelő objektumokat a `FindAll` metódussal.

A `$colResult` változóban tárolt eredmény nem közvetlenül `DirectoryEntry` típusú elemek tömbje! Hanem tulajdonképpen egy hashtábla-szerűség, ahol a `Path` oszlop tartalmazza a megtalált objektum LDAP formátumú elérési útját, a `Properties` meg a kiolvasható tulajdonságait. Azaz ahhoz, hogy kiolvassuk például az én nevemet és beosztásomat egy kicsit trükközni kell:

```
[25] PS I:\>"$($colResults[0].properties.displayname) az én nevem, beoszt
ásom $($colResults[0].properties.title)"
Soós Tibor az én nevem, beosztásom műszaki igazgató
```

Megjegyzés

A PowerShell tényked̄ésem során ez a második eset, amikor kis-nagybetű érzékenységet tapasztaltam! (Az első az LDAP:: kifejezésnél volt, de ez félig-meddig betudható az ADSI örökségnek.) A második ez: ha `$colResults[0].properties.displayName`-et írok (nagy „N” az utolsó tagban), akkor nem kapok semmit. Ez azért is furcsa, mert eredetileg a címtárban nagy az „N”.

A következő példában kikeresem a sémából az összes olyan tulajdonság sémaelemet, amely a globális katalógusba replikálódik:

```
$strFilter =
"(&(objectCategory=attributeSchema)(isMemberOfPartialAttributeSet=TRUE))"
```

```
$objRoot = [ADSI] "LDAP://CN=Schema,CN=Configuration,DC=iqjb,DC=w08"
$objSearcher = New-Object System.DirectoryServices.DirectorySearcher
$objSearcher.SearchRoot = $objRoot
$objSearcher.PageSize = 1000
$objSearcher.Filter = $strFilter
$objSearcher.SearchScope = "Subtree"

$colProplist = "name"
foreach ($i in $colPropList){$objSearcher.PropertiesToLoad.Add($i)}

$colResults = $objSearcher.FindAll()

foreach ($objResult in $colResults)
    {
        $objItem = $objResult.Properties
        "$($objItem.item('name'))"
    }

$objSearcher.Dispose()
$colResults.Dispose()
```

Itt annyival gazdagítottam a keresőm tulajdonságait, hogy meghatároztam a maximális találatok számát (`PageSize`) és a találati listába betöltött elemek tulajdonságainak listáját (`PropertiesToLoad`), ami itt most csak az elem neve. A szkriptemben az első `foreach` ciklust tulajdonképpen feleslegesen használtam, hiszen csak egy tulajdonságot (name) töltöttem a kereső „`PropertiesToLoad`” képességébe, de én ezt a szkriptet sokszor olyankor is fel akarom használni kevés változtatással, amikor több tulajdonságot is meg akarok kapni a keresés eredményében.

A kiíratásnál most a hashtáblás stílussal hivatkoztam a „`name`” tulajdonságra, és a folyamat végén, memóriatakarékossági okokból, eldobom a kereső és a találati lista objektumokat.

A következő példa függvényével felhasználói fiókok tetszőleges attribútumát lehet tömegesen lecserélni valami másra. A kód megfejtését az előzőek ismeretében az olvasóra bízom. Csak egy kis segítséget adok: a középrészen található `If` vizsgálat `Else` ágában azt érem el, hogy ha a kicserélendő attribútum érték üres, azaz azt szeretnénk, hogy a ki nem töltött attribútumokat töltsük ki, akkor ez az LDAP filterben a `!$Attr=*` kifejezést kell szerepeltetni, ennek az a jelentése, hogy „*az $Attr változó által jelzett attribútum nem egyenlő akármi, azaz van értéke*”.

```
function ModifyUserAttrib
{
param (
$domain =
        [System.DirectoryServices.ActiveDirectory.Domain]::getcurrentdomain(
).Name,
$Attr = $(throw "Melyik attribútumot?"),
$sValue = $null,
$cValue = $(throw "Mire változtassam?")
)
```

```
$root= [ADSI] "LDAP://$domain"
$Searcher = New-Object DirectoryServices.DirectorySearcher
$Searcher.SearchRoot = $root
if($sValue)
{
        $buildFilter = "(&(objectClass=user)($Attr=$sValue))"
}
else
{
        $buildFilter = "(&(objectClass=user)(!$Attr=*))"
}
$Searcher.Filter = $buildFilter
$users = $searcher.findall()
Foreach ($i in $users)
{
        $dn=$i.path
        $user = [ADSI] $dn
        write-host $dn
        $user.Put($Attr,$cValue)
        $user.SetInfo()
}
}
```

2.9.10.1 Keresés idő típusú adatokra

A címtárban nem csak szöveges adatok vannak, hanem például dátum típusúak is. Ezekre nem triviális a keresés. Például keresem az utóbbi 2 napban módosított AD felhasználói objektumokat:

```
PS C:\> $tól=get-date ((get-date).AddDays(-2)) -Format yyyyMMddHHMMss.0Z
PS C:\> $tól
20080530110514.0Z
PS C:\> $searcher = New-Object directoryservices.directorysearcher
PS C:\> $searcher.searchroot = [ADSI] ""
PS C:\> $searcher.filter = "(&(objectCategory=person)(objectClass=User)(when
Changed>=$tól))"
PS C:\> $result = $searcher.findall()
PS C:\> $result

Path                                Properties
----                                ----------
LDAP://CN=János Vegetári,OU=Demó,D... {samaccounttype, lastlogon, dscore...
```

Az egészben a lényeg a `$tól` változó generálása. Látjuk, hogy egy speciális formátumra kell hozni a dátumot: ÉÉÉÉHHNNÓÓPPMM.0Z. Ilyen formázást szerencsére a `get-date format` paraméterével könnyen elvégezhetünk.

Sajnos nem mindig ilyen egyszerű a helyzetünk, hiszen néhány dátumot jelző attribútum nem ilyen formátumban tárolja az időt, hanem u.n. „tick”-ekben, ketyegésekben. Ez 1600. január 1. 0:00 időponttól eltelt 100 nanoszekundumokat jelenti, mindez *long-integer* formátumban. Ilyen attribútum például a *lastLogon*, *lastLogonTimestamp*,

lastLogoff, *pwdLastSet*. Ha ilyen attribútumok valamely értékére akarunk rákeresni, akkor elő kell tudnunk állítani ezt az értéket. Szerencsére a .NET keretrendszer ebben is segít. Elsőként nézzük, hogy dátumból hogyan tudunk ilyen *long-integer*-t előállítani:

```
[5] PS C:\> $most = get-date
[6] PS C:\> $most

2008. június 1. 20:49:41
[7] PS C:\> $most.ticks
633479501812656250
```

Látjuk, hogy elég a `ticks` tulajdonságát meghívni a dátumnak. Nézzük visszafele, az-az *long-integer*-ből hogyan kapunk dátumot? Ez sem nagyon bonyolult:

```
[8] PS C:\> $MyLongDate = 633479501812656250
[9] PS C:\> $mydate = New-Object datetime $MyLongDate
[10] PS C:\> $mydate

2008. június 1. 20:49:41
```

Egyszerűen kell kreálni egy dátum típusú objektumot, az objektum konstruktorának kell átadni ezt a számot és ebből a .NET létrehozza a dátumot.

2.9.10.2 Keresés bitekre

A másik nem triviális eset, hogy bizonyos attribútumokban tárolt szám egyes bitjei jelentenek valamit. Például a felhasználói fiók *userAccountControl* attribútumának különböző bitjei a következőket jelentik:

Jellemző	userAccountControl bit	2^x
ACCOUNT DISABLED	2	1
PASSWORD NOT REQUIRED	32	5
PASSWORD NEVER EXPIRES	65536	16
SMARTCARD REQUIRED	262144	18
ACCOUNT TRUSTED FOR DELEGATION	524288	19
ACCOUNT CANNOT BE DELEGATED	1048576	20

Itt a feladat az, hogy olyan vizsgálatot végezzünk a szűrőben, amellyel csak egy adott bit értékére keresünk rá. Szerencsére van erre a célra két u.n. vezérlő (control) az AD-ben, ami ezt a célt szolgálja:

- 1.2.840.113556.1.4.803 – bit szintű AND szabály
- 1.2.840.113556.1.4.804 – bit szintű OR szabály

Hogyan kell ezeket használni? Nézzük például a hatástalanított felhasználói fiókokat:

```
(&(objectCategory=person)(objectClass=user)(userAccountControl:1.2.840.11355
6.1.4.803:=2))
```

Ebben a példában az utolsó feltétel azt jelenti, hogy a `userAccountControl` 2-es bitje helyén 1 van. Azaz „összeéselem” 2-vel az attribútum értékét, a többi bit nem érdekel engem, így az adott feltétel akkor értékelődik ki igazzá, ha ott 1-es szerepel.

Nézzünk egy olyan példát, hogy keresem a Global és Domain Local biztonsági csoportokat:

```
(&(groupType:1.2.840.113556.1.4.804:=6)(groupType:1.2.840.113556.1.4.803:=21
47483648))
```

Itt az első feltételelem akkor értékelődik igazzá, ha vagy a 2-es, vagy a 4-es bit helyén szerepel 1-es, ez a két bit jelzi a Globális és Domain Local csoporttípust.

A második feltétel az már egy AND szabály, az azt vizsgálja, hogy a csoport biztonsági típusú.

2.9.11 Objektumok törlése

Az objektumok törlésének menete nagyon hasonlatos a létrehozásukhoz. Egy felhasználó törlése például így néz ki, ha már megragadtuk a felhasználói fiókot a `$user` változóba:

```
PS C:\> $user.psbase.parent.delete("user","cn=$($user.cn)")
```

A `$user.psbase.parent` megadja a szülő konténert, ezen belül most a `Create` helyett a `Delete` metódust kell meghívni. Miután már van `$user`, ezért legegyszerűbb, ha rögtön ebből olvassuk ki a *relative distinguished* nevét.

Ha nincs megragadva a felhasználó, akkor rá is kereshetünk, és utána törölhetjük:

```
PS C:\> $t = "tt"
PS C:\> $searcher = New-Object directoryservices.directorysearcher
PS C:\> $searcher.searchroot = [ADSI] ""
PS C:\> $searcher.filter = "(&(objectclass=user)(sAMAccountName=$t))"
PS C:\> $tuser = $searcher.findone()
PS C:\> $user=$tuser.getdirectoryentry()
PS C:\> $user.psbase.parent.delete("user","cn=$($user.cn)")
```

Itt most a *pre-Windows 2000* név, azaz a *samaccountname* („tt”) alapján kerestem rá a felhasználóra. Megint oda kell figyelni, hogy a keresés eredménye még nem közvetlenül `DirectoryEntry` típusú objektum, ezért kell az utolsó előtti sorban a találatból igazi felhasználói fiókot konvertálni. A törlés maga már a megismert módon megy.

2.9.12 AD objektumok hozzáférési listájának kezelése

Miután jelenleg az AD adatbázisa PSDrive-ként nem elérhető a PowerShell 1.0-ban, így kicsit körülményesebb a hozzáférési lista kezelése. Ráadásul az ADSI objektumoknál láttuk, hogy a PowerShell adaptációs rétege sok tulajdonságot és metódust elrejt, például

épp a hozzáférési listával kapcsolatos adatokat, így itt is szükségünk van a `PSBase` nézet használatára.

Elsőként nézzük, hogy egy konkrét AD objektum, jelen esetben egy felhasználó hozzáférési listáját hogyan lehet kiolvasni:

```
[PS] C:\>$u= [ADSI] "LDAP://cn=János Vegetári,OU=Demó,DC=adatum,DC=com"
[PS] C:\>$acl = $u.psbase.objectsecurity
[PS] C:\>$acl.GetAccessRules($true,$true,[System.Security.Principal.Security
Identifier])


ActiveDirectoryRights : GenericRead
InheritanceType       : None
ObjectType            : 00000000-0000-0000-0000-000000000000
InheritedObjectType   : 00000000-0000-0000-0000-000000000000
ObjectFlags           : None
AccessControlType     : Allow
IdentityReference     : S-1-5-10
IsInherited           : False
InheritanceFlags      : None
PropagationFlags      : None
...
```

A `$acl` változó tartalmazná a hozzáférési listát, de valójában ezt nem tudjuk közvetlenül kiolvasni egy propertyből, hanem egy metódust kell meghívni (`GetAccessRules`), hogy emberi fogyasztásra alkalmas információkhoz jussunk. Ezen metódusnak paramétereket kell átadni a következők szerint:

```
PS C:\> $acl.GetAccessRules.Value


MemberType          : Method
OverloadDefinitions : {System.Security.AccessControl.AuthorizationRuleColle
                      ction GetAccessRules(Boolean includeExplicit, Boolean
                       includeInherited, Type targetType)}
TypeNameOfValue     : System.Management.Automation.PSMethod
Value               : System.Security.AccessControl.AuthorizationRuleCollec
                      tion GetAccessRules(Boolean includeExplicit, Boolean
                      includeInherited, Type targetType)
Name                : GetAccessRules
IsInstance          : True
```

Az első bool paraméterrel lehet szabályozni, hogy kíváncsi vagyok-e az explicit hozzáférési bejegyzésekre, a második bool paraméter szabályozza, hogy kíváncsi vagyok-e az örökölt hozzáférési bejegyzésekre. A harmadik paraméter a zűrösebb, azzal lehet szabályozni, hogy a metódus kimenete milyen formátumú legyen. Én a fenti példában SID-ekre kértem, hogy fordítsa le a bejegyzéseket. Ez nem biztos, hogy jól értelmezhető, így használhatunk egy másik paraméterezési formát is, amellyel a felhasználói fiókok neveit kapjuk vissza a hozzáférési bejegyzésekben:

```
[PS] C:\>$acl.GetAccessRules($true,$true,[System.Security.Principal.NTAccoun
t])


ActiveDirectoryRights : GenericRead
InheritanceType       : None
ObjectType            : 00000000-0000-0000-0000-000000000000
InheritedObjectType   : 00000000-0000-0000-0000-000000000000
ObjectFlags           : None
AccessControlType     : Allow
IdentityReference     : NT AUTHORITY\SELF
IsInherited           : False
InheritanceFlags      : None
PropagationFlags      : None
...
```

A tulajdonosra is szükségünk lehet, ahhoz a `GetOwner` metódussal juthatunk hozzá, szintén használhatnánk akár a SID-es formátumot is, de talán itt is gyakoribb az olvasmányosabb nevek használata:

```
[PS] C:\>$acl.GetOwner([System.Security.Principal.NTAccount])

Value
-----
ADATUM\Judy
```

Azért álljon itt emlékeztetőül a SID-es formátum listázása is:

```
[PS] C:\>$acl.GetOwner([System.Security.Principal.SecurityIdentifier])

             BinaryLength AccountDomainSid         Value
             ------------ ----------------         -----
                       28 S-1-5-21-150787130-28... S-1-5-21-150787130-28...
```

A hozzáférési listák módosításához szintén számos metódust tudunk segítségül hívni:

```
ModifyAccessRule                Method         System.Boolean ModifyAcce...
ModifyAuditRule                 Method         System.Boolean ModifyAudi...
PurgeAccessRules                Method         System.Void PurgeAccessRu...
PurgeAuditRules                 Method         System.Void PurgeAuditRul...
RemoveAccess                    Method         System.Void RemoveAccess(...
RemoveAccessRule                Method         System.Boolean RemoveAcce...
RemoveAccessRuleSpecific        Method         System.Void RemoveAccessR...
RemoveAudit                     Method         System.Void RemoveAudit(I...
RemoveAuditRule                 Method         System.Boolean RemoveAudi...
RemoveAuditRuleSpecific         Method         System.Void RemoveAuditRu...
ResetAccessRule                 Method         System.Void ResetAccessRu...
SetAccessRule                   Method         System.Void SetAccessRule...
SetAccessRuleProtection         Method         System.Void SetAccessRule...
SetAuditRule                    Method         System.Void SetAuditRule(...
SetAuditRuleProtection          Method         System.Void SetAuditRuleP...
SetGroup                        Method         System.Void SetGroup(Iden...
SetOwner                        Method         System.Void SetOwner(Iden...
```

A könyv keretein túlmutat, hogy mindegyiket külön bemutassam, de ezen információk alapján azt hiszem, el lehet indulni.

2.9.13 Összetett feladat ADSI műveletekkel

Végezetül nézzünk egy összetettebb feladatot a Windows Server 2008 tartományi környezetben felmerülő problémára. Egy új lehetőség itt az u.n. „*Fine grained password policy*”, azaz a tartományi szintnél alacsonyabb szinten beállítható jelszó házirendek létrehozásának lehetősége. Ezzel csak egy probléma van: nem szervezeti egységekre, hanem biztonsági csoportokra tudjuk ezeket kiosztani. Ha konzisztensek szeretnénk maradni a házirendek tekintetében, akkor érdemes „árnyékcsoportokat” létrehozni, azaz olyan csoportokat, amelyek egy adott szervezeti egység összes felhasználóját tartalmazzák. Erre készítettem ezt a függvényt, amely automatikusan létrehozza „*shadow*” névelőtaggal és az OU nevével mint utótaggal a csoportot az adott OU-ban, ha még nincs ilyen. Belerakja az összes olyan felhasználót ebbe a csoportba, akik még nem tagjai. Paraméterként egy OU *distinguished name* adatát kell megadni. Ha egy felhasználót kitörlünk az adott OU-ból, akkor azt az AD automatikusan kiszedi a shadowgroup-ból, viszont ha csak átmozgatjuk a felhasználót egy másik tárolóba, akkor annak a csoport tagjaiból való eltávolításáról nekünk kell gondoskodni. Ezt hajtja végre a szkript vége.

Rendszeresen futtatva szinkronban tarthatjuk az árnyékcsoportok tagságát az OU felhasználói objektumaival.

```
function update-shadowgroup ([string] $ou = "OU=Demó 2,DC=iqjb,DC=w08")
{
        $adou = [ADSI] "LDAP://$ou"

        $query = new-object system.directoryservices.directorysearcher
        $query.SearchScope = "OneLevel"
        $query.SearchRoot = $adou
        $query.filter = "(&(objectCategory=group)(name=shadow-
$($adou.name)))"
        $sg = $query.FindOne()
        if(-not $sg)
        {
                $ADS_GROUP_TYPE_GLOBAL_GROUP = 0x00000002
                $ADS_GROUP_TYPE_DOMAIN_LOCAL_GROUP = 0x00000004
                $ADS_GROUP_TYPE_LOCAL_GROUP = 0x00000004
                $ADS_GROUP_TYPE_UNIVERSAL_GROUP = 0x00000008
                $ADS_GROUP_TYPE_SECURITY_ENABLED = 0x80000000

                $groupType = $ADS_GROUP_TYPE_SECURITY_ENABLED -bor
$ADS_GROUP_TYPE_GLOBAL_GROUP
                $sg = $adou.Create("Group", "CN=shadow-$($adou.name)")
                Write-Host "Creating group: shadow-$($adou.name)..."
                $sg.Put("groupType", $groupType)
                $sg.SetInfo()
        }
        else
        {
```

```
                $sg = [ADSI] $sg.Path
        }
        $query.SearchScope = "OneLevel"
        $query.SearchRoot = $adou
        $query.filter = "(objectCategory=user)"
        $users = $query.FindAll()

        foreach($user in $users)
        {
                if($user.properties.memberof -notcontains
$sg.distinguishedname)
                        {
                                $sg.member +=
$user.properties.distinguishedname
                                write-host "Inserting
$($user.properties.name)..."
                        }
        }
        $sg.setinfo()

        $members = $sg.member
        foreach($member in $members)
        {
                if(-not $member.Contains(",OU=$($adou.name),"))
                {
                        write-host "Removing $member..."
                        $sg.Remove("LDAP://$member")
                        $sg.setinfo()
                }
        }
}
```

2.10 .NET Framework hasznos osztályai

Az eddigiekben is láthattunk már jó néhány olyan problémát, amelyben a .NET keretrendszer osztályai nyújtottak segítséget, mert a PowerShell saját cmdletjei, típusai nem voltak elegendőek. Ebben a fejezetben néhány további nagy tudású .NET osztályból szemezgetek, ezzel kívánok mindenkit arra bíztatni, hogy nyugodtan böngészgessen a .NET osztályok között a Reflector program vagy az MSDN weboldal segítségével, hátha olyanra bukkan, amely kész megoldást nyújt valamely összetettebb feladatra.

2.10.1 Levélküldés

Szkriptekben gyakori feladat a levélküldés, hiszen egy ütemezett szkript eredményéről legkényelmesebben e-mailben kaphatunk visszajelzést. A PowerShell nem rendelkezik e-mailküldésre képes cmdlettel, de szerencsére a .NET keretrendszer igen. Erre a célra a `System.Net.Mail` névtér különböző osztályait használhatjuk fel.

Elsőként a levelet kell összeraknunk egy `MailMessage` típus segítségével. Az alábbi szkript ezt a feladatot végzi el:

```
$mail = New-Object System.Net.Mail.MailMessage
$mail.From = New-Object System.Net.Mail.MailAddress("soost@iqjb.hu")
$mail.To.Add("soostibor@citromail.hu")
$mail.Subject = "Haliho"
$mail.Body = "Ez itt a levél."
```

A `$mail` változó tartalmaz egy levélobjektumot, melynek feladója soost@iqjb.hu, címzettje soostibor@citromail.hu, tárgya „Haliho”, szövege „Ez itt a levél.”

Ezután már csak el kell küldeni ezt. Ehhez egy erre alkalmas SMTP kiszolgálót kell találni. Ezt az SMTP kiszolgálót megszólítani képes `SmtpClient` .NET típus segítségével el is tudjuk küldeni a levelet:

```
$smtp = New-Object System.Net.Mail.SmtpClient -argumentList "mail.iqjb.corp"
$smtp.Credentials = New-Object System.Net.NetworkCredential `
-argumentList "soost@iqjb.hu","password"
$smtp.Send($mail)
```

Ez a típus rendelkezik minden olyan funkcióval, ami az SMTP kiszolgálókkal való kommunikációban szükséges lehet. Például kezeli a hitelesítési információkat és természetesen átadja a levélobjektumot kézbesítésre.

2.10.2 Böngészés

A következő gyakori feladat a weben található információk letöltése. Erre a célra a .NET keretrendszerben a `System.Net.WebClient` osztály áll rendelkezésre. Itt nem egy emberi fogyasztásra szánt webböngészőt kell elképzelni, hanem egy olyan alapszol-

gáltatást, ami csatlakozik egy adott URL segítségével a webkiszolgálóhoz és kiolvassa az adott weboldalt:

```
[2] PS C:\> $client = New-Object System.Net.WebClient
[3] PS C:\> $client.DownloadString("http://www.geocaching.hu")
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">


<html>
<head>
        <meta http-equiv="Content-Type" content="text/html; charset=iso-8859
-2">
        <title>geocaching.hu</title>
        <meta name="description" content="A magyar geocaching központja, geo
ládák
koodinátái, leírások, fényképek, letöltés, fórum.">
        <meta name="keywords" content="kincskeresés, geocaching, geochaching
, geoc
ashing, geokesing, gps, kincs, kincsvadászat, keresés, láda, geoláda, koord
ináta, koordináták, letöltés, kincsesláda, cache">
<link rel="stylesheet" href="style.css" type="text/css">
...
```

A [3]-as sorban a `DownloadString` metódussal indítom a böngészést. Az eredmény – mivel nem rendelkeztem róla – a konzolra került ki. Látható, hogy a teljes HTML tartalmat megkaptam. Természetesen szkriptjeimben ezt nem a képernyőre fogom kifolyatni, hanem egy változóba töltök be és valamilyen elemzés, átalakítás (például a HTML címkéktől való megszabadítás) után adom csak vissza az engem ténylegesen érdeklő információkat.

A `DonwloadString` metódus olvasható karaktereket vár azon a weboldalon, amelyre ráirányítjuk. Ha azonban egy bináris állományt, például egy ZIP fájlt akarunk letölteni, akkor ez nem lesz nekünk jó. Ilyen letöltésekre egy másik metódust, a `DownloadFile`-t használhatjuk:

```
[5] PS I:\>$client = New-Object System.Net.WebClient
[6] PS I:\>$url= "http://www.xs4all.nl/~hneel/software/bytecount.zip"
[7] PS I:\>$filename = "c:\bytecount.zip"
[8] PS I:\>$client.DownloadFile($url, $filename)
[9] PS I:\>Get-ChildItem c:\b*


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---     2008. 04. 22.      9:53     419660 bytecount.zip
```

A fenti példában látható, hogy ennek a metódusnak az URL-en kívül egy fájl elérési útja is paramétere, ahova majd letölti a bináris állományt.

2.10.3 Felhasználói információk

A felhasználói információkat is egyszerűen összegyűjthetjük az alábbi `WindowsIdentity` osztály `GetCurrent` metódusával:

```
[1] PS I:\>[System.Security.Principal.WindowsIdentity]::GetCurrent()


AuthenticationType : Kerberos
ImpersonationLevel : None
IsAuthenticated    : True
IsGuest            : False
IsSystem           : False
IsAnonymous        : False
Name               : IQJB\soostibor
Owner              : S-1-5-21-861567501-1202660629-1801674531-6051
User               : S-1-5-21-861567501-1202660629-1801674531-6051
Groups             : {S-1-5-21-861567501-1202660629-1801674531, , , ...}
Token              : 904
```

A kimenetben láthatóak a legfontosabb adatok: a felhasználói név, a csoporttagságra utaló SID-ek, az autentikáció módja, stb.

Miután SID-ekkel nem annyira könnyű dolgozni, ezért egy másik osztállyal, a `WindowsPrincipal`-lal további szolgáltatásokhoz jutunk. Nagyon egyszerűen le lehet kérdezni például, hogy vajon az éppen aktuális felhasználó rendszergazda jogosultságokkal bír-e, azaz tagja-e az `Administrators` csoportnak:

```
[6] PS C:\> $u = [System.Security.Principal.WindowsIdentity]::GetCurrent()
[7] PS C:\> $principal = New-Object Security.principal.windowsprincipal($u)
[8] PS C:\> $principal.IsInRole("Administrators")
True
```

Ezekkel nagyon hatékonyan lehet megvizsgálni az éppen aktuális felhasználó különböző jellemzőit akár tartományi, akár helyi gépes környezetben.

2.10.4 DNS adatok lekérdezése

A DNS adatokról volt már szó a *2.9.4 Active Directory információk lekérdezése* fejezetben, de ott az Active Directory szempontjait vettem előtérbe. Ha egy általános DNS névfeloldást szeretnénk végrehajtani, arra a `System.Net.DNS` osztály használható, annak is a `GetHostByName` metódusa:

```
PS C:\Users\Administrator> [System.Net.Dns]::GetHostByName("www.microsoft.co
m")

HostName                  Aliases                   AddressList
--------                  -------                   -----------
lb1.www.ms.akadns.net     {www.microsoft.com, t... {207.46.192.254, 207....
```

```
PS C:\Users\Administrator> [System.Net.Dns]::GetHostByName("www.microsoft.co
m").aliases
www.microsoft.com
toggle.www.ms.akadns.net
g.www.ms.akadns.net
PS C:\Users\Administrator> [System.Net.Dns]::GetHostByName("www.microsoft.co
m").addresslist


IPAddressToString : 207.46.192.254
Address           : 4274007759
AddressFamily     : InterNetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False

IPAddressToString : 207.46.193.254
Address           : 4274073295
AddressFamily     : InterNetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False

IPAddressToString : 207.46.19.254
Address           : 4262670031
AddressFamily     : InterNetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False

IPAddressToString : 207.46.19.190
Address           : 3188928207
AddressFamily     : InterNetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False
```

Látható, hogy ez egyszerű IP cím lekérdezésen kívül számos egyéb információhoz hozzájuthatunk.

2.11 SQL adatelérés

Az adatbázisok adatai is elérhetők a PowerShell segítségével, itt is .NET osztályokat hívhatunk segítségül. Nézzünk egy nagyon egyszerű adatbázist, annak is egy tábláját:

Személyek			
ID	Név	e-mail	Telefon
1	Soós Tibor	soostibor@citromail.hu	123-45678
2	Vegetári János	vj@sehol.hu	654-6543
3	Fájdalom Csilla	fcs@valahol.hu	161-6161

Ezt én most éppen *Microsoft Access*ben hoztam létre, de lehetne ez az adatbázis akár SQL Serveren, vagy bármilyen olyan adatbázis-kezelőben, amelynek van ODBC csatolója.

Az alábbi sorokkal egy SQL kifejezéssel adatokat tudunk átemelni PowerShell változóba:

```
[1] PS C:\> $connectionString = "Provider=microsoft.jet.oledb.4.0; " +
>>                     "Data Source=C:\scripts\emberek.mdb; "
>>
[2] PS C:\> $sqlCommand = "select * from személyek"
[3] PS C:\> $connection = New-Object System.Data.OleDb.OleDbConnection $conn
ectionString
[4] PS C:\> $command = New-Object System.Data.OleDb.OleDbCommand $sqlCommand
,$connection
[5] PS C:\> $connection.Open()
[6] PS C:\> $adapter = New-Object System.Data.OleDb.OleDbDataAdapter $comman
d
[7] PS C:\> $dataset = New-Object System.Data.DataSet
[8] PS C:\> [void] $adapter.Fill($dataSet)
[9] PS C:\> $connection.Close()
[10] PS C:\> $records = $dataSet.Tables | Select-Object -Expand Rows
[11] PS C:\> $records

                ID Név               e-mail             Telefon
                -- ---               ------             -------
                 1 Soós Tibor         soostibor@citro... 123-45678
                 2 Vegetári János     vj@sehol.hu        654-6543
                 3 Fájdalom Csilla    fcs@valahol.hu     161-6161
```

Az [1]-es sorban összeállítok egy u.n. *„connection string”*-et, ami most éppen Access adatbázishoz való csatlakozást rejt, de kis módosítással bármilyen ODBC adatforráshoz testre szabható. A [2]-es sorban összerakom az SQL kifejezést, amelynek eredményét szeretném PowerShellből kezelni. A [3]-astól a [6]-os sorig felépítem a kapcsolatot az adatforráshoz, és végrehajtom az SQL kifejezést, majd a [7]-es sorban töltöm be a `$dataset` változóba az adatforrásból kapott adatokat. Ezután már a [9]-es sorban bonthatom is a kapcsolatot az adatbázissal. A `$dataset` még elég összetett típus, közvetlen adatkinyerésre nehézkesen használható, így a [10]-es sorban sorokra bontom ezt az ob-

jektumot a `Select-Object` cmdlet segítségével. Látható, hogy a `$records` már a fejezet elején látható táblázathoz nagyon hasonló formátumot adott vissza.

Vajon honnan tudtam a [10]-es sorban, hogy van a `$dataset.Tables` tagjellemzőnek (jobban mondva nem is annak, hanem a `Tables` tömb egyes elemeinek) `Rows` tulajdonsága, amit a `Select-Object`-tel ki tudok fejteni? Ezt nem nagyon lehet tudni magából a PowerShell segítségével, mert - mint ahogy az ADSI objektumoknál is láttuk - itt is COM objektumok húzódnak meg a háttérben. Nézzük, mit látunk:

```
[14] PS C:\> $dataset.tables[0] | Get-Member -MemberType properties


   TypeName: System.Data.DataRow

Name    MemberType Definition
----    ---------- ----------
e-mail  Property   System.String e-mail {get;set;}
ID      Property   System.Int32 ID {get;set;}
Név     Property   System.String Név {get;set;}
Telefon Property   System.String Telefon {get;set;}
```

A `Tables` tömbben csak egy elem van, így néztem annak tulajdonság típusú tagjellemzőit, de ott nem szerepel `Rows`. Ezzel szemben mégis van:

```
[15] PS C:\> $dataset.tables[0].rows

                ID Név                e-mail             Telefon
                -- ---                ------             -------
                 1 Soós Tibor          soostibor@citro... 123-45678
                 2 Vegetári János      vj@sehol.hu        654-6543
                 3 Fájdalom Csilla     fcs@valahol.hu     161-6161
```

Ezek után hogyan tudjuk ezeket az adatokat kezelni? A `$records` tömb egyes elemei maguk az adatrekordok. Például a második sorra így lehet hivatkozni:

```
[16] PS C:\> $records[1]

                ID Név                e-mail             Telefon
                -- ---                ------             -------
                 2 Vegetári János      vj@sehol.hu        654-6543
```

Ezen adatrekord egyik mezőjére már pedig a szokásos tulajdonság-hivatkozással:

```
[17] PS C:\> $records[1].Név
Vegetári János
```

Természetesen SQL adatokat módosítani is lehet, ehhez a fenti példát úgy kell megváltoztatni, hogy az SQL utasításban valamilyen adatmanipulációs parancsot kell kiadni.

Megjegyzés

A 64-bites Windows XP-n a fenti SQL-es PowerShell kifejezések hibát adnak a „normál” PowerShell ablakban futtatva. Ennek oka az, hogy a szükséges ODBC driver 64-bites változata nincsen benne az operációs rendszerben, csak a 32-bites változat, amit csak 32 bites alkalmazásból szólíthatunk meg. Ha ilyen hibát tapasztalunk, akkor futtassuk a fenti kódot 32-bites PowerShell ablakban, ami szintén elérhető a 64-bites gépen is, az már működni fog minden baj nélkül.

2.12 COM objektumok kezelése

A *Component Object Model (COM)* a Microsoft által 1993-ban kifejlesztett interfész szabvány szoftverkomponensek közti kommunikációra. Segítségével minden ilyen programnyelven, amely támogatja ezt a szabványt, lehet készíteni olyan szoftverkomponenseket, amelyek képesek egymással kommunikálni és dinamikusan egymás objektumait létrehozni, kezelni. Sőt! Akár szkriptnyelvekből (VBScript, PowerShell) is meg lehet szólítani ezeket a komponenseket, mint ahogy az alábbiakban látható lesz.

A COM-ba számos más „altechnológia" tartozik: OLE, OLE Automation, ActiveX, COM+ és DCOM.

2.12.1 A Windows shell kezelése

Maga a Windows grafikus keretprogramja, shellje is COM objektum, azaz megszólítható, a publikus metódusai és tulajdonságai meghívhatók, lekérdezhetőek. Ehhez először meg kell hívni az shellt, ezt a PowerShellben nagyon egyszerűen, a `new-object -com` commandlettel tehetjük meg:

```
[2] PS C:\> $sh = new-object -com Shell.Application
```

Miután ezt az objektumot még többször akarom használni, ezért egy `$sh` változóba tettem. Nézzük meg ennek tagjellemzőit:

```
[3] PS C:\> $sh | gm


   TypeName: System.__ComObject#{efd84b2d-4bcf-4298-be25-eb542a59fbda}

Name                 MemberType Definition
----                 ---------- ----------
AddToRecent          Method     void AddToRecent (Variant, string)
BrowseForFolder      Method     Folder BrowseForFolder (int, string, int...
CanStartStopService  Method     Variant CanStartStopService (string)
CascadeWindows       Method     void CascadeWindows ()
ControlPanelItem     Method     void ControlPanelItem (string)
EjectPC              Method     void EjectPC ()
Explore              Method     void Explore (Variant)
ExplorerPolicy       Method     Variant ExplorerPolicy (string)
FileRun              Method     void FileRun ()
FindComputer         Method     void FindComputer ()
FindFiles            Method     void FindFiles ()
FindPrinter          Method     void FindPrinter (string, string, string)
GetSetting           Method     bool GetSetting (int)
GetSystemInformation Method     Variant GetSystemInformation (string)
Help                 Method     void Help ()
IsRestricted         Method     int IsRestricted (string, string)
IsServiceRunning     Method     Variant IsServiceRunning (string)
MinimizeAll          Method     void MinimizeAll ()
NameSpace            Method     Folder NameSpace (Variant)
```

```
Open                Method     void Open (Variant)
RefreshMenu         Method     void RefreshMenu ()
ServiceStart        Method     Variant ServiceStart (string, Variant)
ServiceStop         Method     Variant ServiceStop (string, Variant)
SetTime             Method     void SetTime ()
ShellExecute        Method     void ShellExecute (string, Variant, Vari...
ShowBrowserBar      Method     Variant ShowBrowserBar (string, Variant)
ShutdownWindows     Method     void ShutdownWindows ()
Suspend             Method     void Suspend ()
TileHorizontally    Method     void TileHorizontally ()
TileVertically      Method     void TileVertically ()
ToggleDesktop       Method     void ToggleDesktop ()
TrayProperties      Method     void TrayProperties ()
UndoMinimizeALL     Method     void UndoMinimizeALL ()
Windows             Method     IDispatch Windows ()
WindowsSecurity     Method     void WindowsSecurity ()
Application         Property   IDispatch Application () {get}
Parent              Property   IDispatch Parent () {get}
```

Ezzel láthatóvá váltak a `Shell.Application` tagjellemzői, azaz a Windows szkriptből is könnyen elérhető szolgáltatásai, mint például a súgó megnyitása vagy akár egy mappa megnyitása a Windows Intézőben:

```
[4] PS C:\> $sh.help()
[5] PS C:\> $sh.explore("C:\scripts")
```

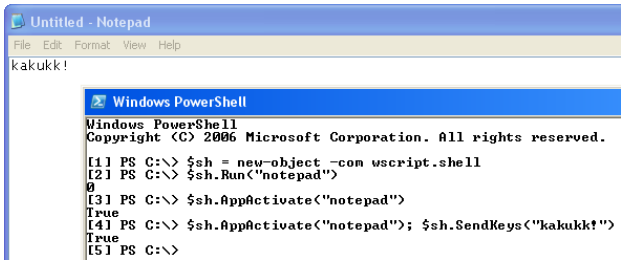
Természetesen az így megnyitott Súgóval vagy az Intézővel olyan sok mindent nem tudunk PowerShellel továbbiakban kezdeni, ez inkább csak a COM objektumok kezelésének a legegyszerűbb és látványos demonstrációja akart lenni. Sokkal praktikusabb lett volna például a `ShutdownWindows` metódus meghívása, de ezt a tisztelt olvasóra bízom ☺.

2.12.2 WScript osztály használata

Az előzőekben láttuk, hogy a Windows Shellt hogyan lehet szkriptből megszólítani. Van egy másik COM objektum, a `WScript`, amelynek `Shell` alosztálya is hasonló célokat, de még mélyebb szinten valósít meg. Például szeretnénk egy olyan alkalmazással kommunikálni, amely nem COM alkalmazás, például a Notepaddal. Jobb híján billentyűzetleütéseket tudunk neki küldeni a `WScript.Shell` osztály `SendKeys` metódusával:

```
[1] PS C:\> $sh = new-object -com wscript.shell
[2] PS C:\> $sh.Run("notepad")
0
[3] PS C:\> $sh.AppActivate("notepad")
True
[4] PS C:\> $sh.AppActivate("notepad"); $sh.SendKeys("kakukk!")
True
```

Az [1]-es sorban elindítom a notepad-et a `Run` metódussal. Aztán előtérbe helyezem az ablakát, hogy ő kapja a billentyűleütéseket, majd kiküldöm a billentyűzetütéseket. Az alábbi képernyőfotón látszik, hogy a kiadott karakterek tényleg beíródtak a notepad szerkesztőfelületére.



62. ábra Nem COM alkalmazás vezérlése WScrip.Shell objektummal

Nézzük ennek az objektumnak a tagjellemzőit:

```
[5] PS C:\> $sh | gm


   TypeName: System.  ComObject#{41904400-be18-11d3-a28b-00104bd35090}

Name                       MemberType            Definition
----                       ----------            ----------
AppActivate                Method                bool AppActivate (Variant...
CreateShortcut             Method                IDispatch CreateShortcut ...
Exec                       Method                IWshExec Exec (string)
ExpandEnvironmentStrings   Method                string ExpandEnvironmentS...
LogEvent                   Method                bool LogEvent (Variant, s...
Popup                      Method                int Popup (string, Varian...
RegDelete                  Method                void RegDelete (string)
RegRead                    Method                Variant RegRead (string)
RegWrite                   Method                void RegWrite (string, Va...
Run                        Method                int Run (string, Variant,...
SendKeys                   Method                void SendKeys (string, Va...
Environment                ParameterizedProperty IWshEnvironment Environme...
CurrentDirectory           Property              string CurrentDirectory (...
SpecialFolders             Property              IWshCollection SpecialFol...
```

Látható, hogy a billentyűleütések mellet többek között registry-szerkesztő, shortcut-létrehozó és dialógusablak-kirakó metódusai is vannak.

A WScriptnek nem csak Shell alosztálya van, hanem Network is. Ezzel számtalan hasznos hálózattal kapcsolatos műveletet tudunk végrehajtani:

```
[11] PS I:\>$ws | gm


   TypeName: System.__ComObject#{24be5a31-edfe-11d2-b933-00104b365c9f}

Name                           MemberType Definition
```

```
----                          ---------- ----------
AddPrinterConnection          Method     void AddPrinterConnection (string...
AddWindowsPrinterConnection Method     void AddWindowsPrinterConnection ...
EnumNetworkDrives             Method     IWshCollection EnumNetworkDrives ()
EnumPrinterConnections        Method     IWshCollection EnumPrinterConnect...
MapNetworkDrive               Method     void MapNetworkDrive (string, str...
RemoveNetworkDrive            Method     void RemoveNetworkDrive (string, ...
RemovePrinterConnection       Method     void RemovePrinterConnection (str...
SetDefaultPrinter             Method     void SetDefaultPrinter (string)
ComputerName                  Property   string ComputerName () {get}
Organization                  Property   string Organization () {get}
Site                          Property   string Site () {get}
UserDomain                    Property   string UserDomain () {get}
UserName                      Property   string UserName () {get}
UserProfile                   Property   string UserProfile () {get}
```

Nézzünk például egy hálózati meghajtók felsorolását:

```
[12] PS C:\> $ws = New-Object -com WScript.Network
[13] PS C:\> $ws.EnumNetworkDrives()
I:
\\K-FILE1\soostibor$
```

2.12.3 Alkalmazások kezelése

A COM által nyújtott interfész lehetővé teszi, hogy nem csak a Windows különböző funkcióit tudjuk elérni, hanem a legtöbb alkalmazás is elérhetővé tesz olyan objektumokat, amelyek hasznos szolgáltatásait felhasználhatjuk szkriptjeinkben. A legegyszerűbb erre vonatkozó példa az Internet Explorer megszólítása, amelyben nem csinálok semmit, csak látható módon elindítom az alkalmazást:

```
[17] PS C:\> $ie = New-Object -ComObject InternetExplorer.Application
[18] PS C:\> $ie.visible=$true
```

Nézzünk egy kicsit bonyolultabb példát, olvastassunk be egy Word dokumentumot, de csak az emberi fogyasztásra szánt értelmes szöveget! Ez nem olyan egyszerű feladat, hiszen ha egyszerű fájlművelettel próbálkoznánk, akkor a Word dokumentum mindenféle formázó információját is beolvassuk, és abból elég nehéz kinyerni a tényleges szöveget. Szerencsére a Word is rendelkezik COM felülettel, így az alábbi néhány soros szkripttel könnyen felolvastathatjuk az „igazi” szöveget a dokumentumból:

```
[25] PS I:\>$wordApp = New-Object -COM Word.Application
[26] PS I:\>$file = (Get-Item C:\_docs\tematikák.docx).FullName
[27] PS I:\>$doc = $wordApp.Documents.Open($file)
[28] PS I:\>$text = $doc.Content.Text
```

```
[29] PS I:\>$text
Microsoft PowerShell for Administrators Who Should AttendAnyone Who Scrip
ts For Windows - this course will help you build scripting skills in PowerS
hell when you are coming from a background in scripting on Windows operatin
...
[30] PS I:\>$wordApp.Quit()
```

A szkript elején megszólítom a `Word.Application` COM objektumot és betöltöm a `$wordApp` változóba, majd a megnyitandó dokumentum elérési útját berakom egy változóba, majd a `$wordApp` segítségével megnyitom ezt a fájlt. Mivel nem rendelkeztem arról, hogy a Word látható legyen, mindez csak a háttérben történik. A fájl megnyitásával egy új objektumhoz jutok, magához a dokumentumhoz, majd ennek veszem a nyers szöveges részét a `$doc.Content.Text` kifejezés segítségével, amit a [29]-es sorban ki is írattam a konzolra.

Ilyen jellegű alkalmazások kezelésekor illik azokat a végén bezárni, hogy ne foglalja feleslegesen a memóriát. Ezt a [30]-as sorban tettem meg.

Természetesen az előző két példa csak ízelítő próbált lenni a COM objektumok kezeléséből, nem kívántam egy komplett COM szakkönyvet írni, hiszen ennek már széles a szakirodalma. A lényeg az, hogy bármilyen COM objektum nagyon egyszerűen megszólítható PowerShellből, így egy meglevő, például VBScriptben írt példa nagyon egyszerűen átemelhető PowerShellbe.

3. Bővítmények

A PowerShell bár jelenleg az 1.0-ás verziónál tart, és ezzel együtt jár az, hogy van néhány hiányossága, de mégis könnyű ezeket orvosolni, hiszen nagyon jól bővíthető. Akár a korábban látott, programozást nem igénylő módon, mint például az *1.5.10 Objektumok testre szabása, kiegészítése* fejezetben látott types.ps1xml segítségével, akár komolyabb programozást igénylő módon. Ez utóbbi részleteibe ebben a könyvben nem megyünk bele, egyrészt mert én nem vagyok fejlesztő, ezért nem is értek hozzá, másrészt ez a könyv kifejezetten rendszergazdáknak íródott. Szerencsére jó néhány funkció megvalósítására különböző szoftvercégek és felhasználói közösségek elkészítették ezeket a programozást igénylő bővítményeket, a *snap-in*-eket. Ezek közül itt példaként bemutatok néhányat, amelyek vagy ingyenesen hozzáférhetők, vagy valamely Microsoft kiszolgáló szoftver (Exchange Server 2007) részeként elérhetők.

3.1 Quest – ActiveRoles Management Shell for Active Directory

A *2.9 Felhasználó-menedzsment, Active Directory* fejezetben láttuk, hogy a címtár kezelése még nem 100%-os az alap PowerShellben. Jórészt COM objektumok húzódnak meg a felszín alatt, és ezek adaptálása nem történt még meg tökéletesen, nem igazán „powershell-szerű” a kezelésük. Ennek orvoslására készített a Quest Software egy bővítményt, ami ingyenesen letölthető a weboldalukról[14].

Nézzük, milyen új cmdleteket kínál ez a bővítmény a telepítés és a PSSnap-in hozzáadása után:

```
PS C:\Users\Administrator> Add-PSSnapin Quest.ActiveRoles.ADManagement
PS C:\Users\Administrator> Get-Command -PSSnapin Quest.ActiveRoles.ADManagem
ent

CommandType     Name                           Definition
-----------     ----                           ----------
Cmdlet          Add-QADGroupMember             Add-QADGroupMember [-Ident...
Cmdlet          Add-QADPasswordSettingsObj...  Add-QADPasswordSettingsObj...
Cmdlet          Add-QADPermission              Add-QADPermission [-Identi...
Cmdlet          Connect-QADService             Connect-QADService [-Proxy...
Cmdlet          Convert-QADAttributeValue      Convert-QADAttributeValue ...
Cmdlet          Deprovision-QADUser            Deprovision-QADUser [-Iden...
Cmdlet          Disable-QADUser                Disable-QADUser [[-Identit...
Cmdlet          Disconnect-QADService          Disconnect-QADService [-Pr...
Cmdlet          Enable-QADUser                 Enable-QADUser [[-Identity...
Cmdlet          Get-QADComputer                Get-QADComputer [[-Identit...
Cmdlet          Get-QADGroup                   Get-QADGroup [[-Identity] ...
```

[14] http://www.quest.com/powershell/activeroles-server.aspx

```
Cmdlet          Get-QADGroupMember            Get-QADGroupMember [-Ident...
Cmdlet          Get-QADObject                 Get-QADObject [[-Identity]...
Cmdlet          Get-QADObjectSecurity         Get-QADObjectSecurity [-Id...
Cmdlet          Get-QADPasswordSettingsObject Get-QADPasswordSettingsObj...
Cmdlet          Get-QADPermission             Get-QADPermission [-Identi...
Cmdlet          Get-QADPSSnapinSettings       Get-QADPSSnapinSettings [-...
Cmdlet          Get-QADRootDSE                Get-QADRootDSE [-Proxy] [-...
Cmdlet          Get-QADUser                   Get-QADUser [[-Identity] <...
Cmdlet          Get-QARSAccessTemplate        Get-QARSAccessTemplate [[-...
Cmdlet          Get-QARSAccessTemplateLink    Get-QARSAccessTemplateLink...
Cmdlet          Move-QADObject                Move-QADObject [-Identity]...
Cmdlet          New-QADGroup                  New-QADGroup [-Name] <Stri...
Cmdlet          New-QADObject                 New-QADObject [-Name] <Str...
Cmdlet          New-QADPasswordSettingsObject New-QADPasswordSettingsObj...
Cmdlet          New-QADUser                   New-QADUser [-Name] <Strin...
Cmdlet          New-QARSAccessTemplateLink    New-QARSAccessTemplateLink...
Cmdlet          Remove-QADGroupMember         Remove-QADGroupMember [-Id...
Cmdlet          Remove-QADObject              Remove-QADObject [-Identit...
Cmdlet          Remove-QADPasswordSettings... Remove-QADPasswordSettings...
Cmdlet          Remove-QADPermission          Remove-QADPermission [-Inp...
Cmdlet          Remove-QARSAccessTemplateLink Remove-QARSAccessTemplateL...
Cmdlet          Rename-QADObject              Rename-QADObject [-Identit...
Cmdlet          Set-QADGroup                  Set-QADGroup [-Identity] <...
Cmdlet          Set-QADObject                 Set-QADObject [[-Identity]...
Cmdlet          Set-QADObjectSecurity         Set-QADObjectSecurity [-Id...
Cmdlet          Set-QADPSSnapinSettings       Set-QADPSSnapinSettings [-...
Cmdlet          Set-QADUser                   Set-QADUser [-Identity] <I...
Cmdlet          Set-QARSAccessTemplateLink    Set-QARSAccessTemplateLink...
Cmdlet          Unlock-QADUser                Unlock-QADUser [[-Identity...
```

A cmdletek főnévi részében a QAD utal a Quest Active Directory kifejezésre, ezzel kívánta a gyártó megkülönböztetni a cmdletjeit a nagy valószínűséggel hasonló neveken majd egyszer a Microsofttól érkező Active Directoryval kapcsolatos cmdletektől.

Nézzünk például egy új felhasználó létrehozását:

```
PS C:\Users\Administrator> New-QADUser -Name "QAD User" -ParentContainer dem
ó -UserPassword "P@ssw0rd"

Name                            Type            DN
----                            ----            --
QAD User                        user            CN=QAD User,OU=Demó,DC=iq...
```

Nagyon tömör kifejezés. Felhívom a figyelmet a `-ParentContainer` paraméterre, ahol nem kellett teljes *distinguished name* adatot megadni, hanem csak elég volt a szervezeti egység neve. Bár a help félrevezető, *distinguished name*-ről beszél:

```
...
PARAMETERS
    -ParentContainer <IdentityParameter>
        Specify the distinguished name (DN) of the container in which you w
        ant this cmdlet to create a new user account.
...
```

De itt valójában bármilyen nevét megadhatjuk a szülő konténernek, ami egyedileg azonosítja azt a helyet, ahova létre szeretnénk hozni a felhasználót. Az én teszt környezetemben csak egy Demó OU-van, így elég volt ennyit írni.

Nézzük, hogyan tudjuk ezt az új felhasználót egy csoportba rakni:

```
PS C:\Users\Administrator> Add-QADGroupMember "Csoport" -Member "QAD User"

Name                            Type            DN
----                            ----            --
QAD User                        user            CN=QAD User,OU=Demó,DC=iq...
```

Ez sem túl bonyolult. Az AD objektumokon való jogosultságkezelés már nem ilyen egyszerű, de az ADSI lehetőségekhez képest azért nagyságrendekkel kényelmesebb a munka. Nézzünk például jogosultságolvasást a Demó OU-n:

```
PS C:\Users\Administrator> Get-QADPermission "Demó" -Inherited -SchemaDefaul
t
Permissions for: iqjb.w08/Demó

WARNING: 2 columns do not fit into the display and were removed.

Ctrl   Account                                  Rights
----   -------                                  ------
Deny   Everyone                                 Special
       NT AUTHORITY\ENTERPRISE DOMAIN CONTRO... Special
       NT AUTHORITY\Authenticated Users         Special
       NT AUTHORITY\SYSTEM                      Full control
       iqjb.w08\Domain Admins                   Full control
       iqjb.w08\Account Operators               Create/Delete user
       iqjb.w08\Account Operators               Create/Delete group
       iqjb.w08\Account Operators               Create/Delete computer
       iqjb.w08\Account Operators               Create/Delete inetOrgPerson
       iqjb.w08\Print Operators                 Create/Delete printQueue
       iqjb.w08\Pre-Windows 2000 Compatible ... Read Account Restrictions
       iqjb.w08\Pre-Windows 2000 Compatible ... Read Account Restrictions
       iqjb.w08\Pre-Windows 2000 Compatible ... Read Logon Information
       iqjb.w08\Pre-Windows 2000 Compatible ... Read Logon Information
       iqjb.w08\Pre-Windows 2000 Compatible ... Read Group Membership
       iqjb.w08\Pre-Windows 2000 Compatible ... Read Group Membership
       iqjb.w08\Pre-Windows 2000 Compatible ... Read General Information
       iqjb.w08\Pre-Windows 2000 Compatible ... Read General Information
       iqjb.w08\Pre-Windows 2000 Compatible ... Read Remote Access Infor...
       iqjb.w08\Pre-Windows 2000 Compatible ... Read Remote Access Infor...
       NT AUTHORITY\ENTERPRISE DOMAIN CONTRO... Read tokenGroups
       NT AUTHORITY\ENTERPRISE DOMAIN CONTRO... Read tokenGroups
       NT AUTHORITY\ENTERPRISE DOMAIN CONTRO... Read tokenGroups
       iqjb.w08\Pre-Windows 2000 Compatible ... Special
       iqjb.w08\Pre-Windows 2000 Compatible ... Special
       iqjb.w08\Pre-Windows 2000 Compatible ... Special
       NT AUTHORITY\SELF                        Special
       iqjb.w08\Enterprise Admins               Full control
       iqjb.w08\Pre-Windows 2000 Compatible ... List Contents
       iqjb.w08\Administrators                  Special
```

Mielőtt nagyon belemerülnénk, érdemes megnézni, hogy milyen alapbeállításokkal működik ez a snap-in, hiszen ezek szabályozzák például egy keresés maximális találatainak számát:

```
PS C:\Users\Administrator> get-help Get-QADPSSnapinSettings

NAME
    Get-QADPSSnapinSettings

SYNOPSIS
    View default settings that apply to all cmdlets of this PowerShell snap
    -in.


SYNTAX
    Get-QADPSSnapinSettings [-DefaultExcludedProperties] [-DefaultPropertie
    sExcludedFromNonBaseSearch] [-Integer8AttributesThatContainDateTimes] [
    -Integer8AttributesThatContainNegativeTimeSpans] [-DefaultPageSize] [-D
    efaultSizeLimit] [-DefaultSearchScope] [-DefaultWildcardMode] [-Default
    OutputPropertiesForUserObject] [-DefaultOutputPropertiesForGroupObject]
     [-DefaultOutputPropertiesForComputerObject] [-DefaultOutputPropertiesF
    orAdObject] [-DefaultOutputPropertiesForPasswordSettingsObject] [<Commo
    nParameters>]
```

Nézzünk pár ilyen paramétert:

```
PS C:\Users\Administrator> Get-QADPSSnapinSettings -DefaultSizeLimit
1000
PS C:\Users\Administrator> Get-QADPSSnapinSettings -DefaultSearchScope
Subtree
```

Természetesen ezeket átállíthatjuk a `set-QADPSSnapinSettings` cmdlettel.

Még egy fontos cmdletet nézzünk meg! Ehhez először figyeljük meg, hogy milyen formában kapjuk meg az adatokat ezen cmdletek futtatása után! Például mit ad egy felhasználó `LastLogon` attribútuma:

```
PS C:\Users\Administrator> Get-QADUser "Administrator" | ForEach-Object {$ .
LastLogon}

Value                                DisplayName
-----                                -----------
2008.09.02. 11:57:23                 2008. szeptember 2.
```

Szemre egész szép `datetime` formátumot kaptam, de ezt ellenőrizzük azért le:

```
PS C:\Users\Administrator> (Get-QADUser "Administrator" | ForEach-Object {$
.LastLogon}).gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    DisplayNameWrapper`1                     System.ValueType
```

Hát ez mégsem az! Hiszen az AD nem .NET objektumokat tárol, ezért szükséges lehet ezeket az attribútumokat .NET-es típussá alakítani. Nézzük ezt meg a fenti példát folytatva! Hámozzuk ki ebből a csomagolásból az AD objektumot, nézzük az eredeti címtárbejegyzést! Itt valami olyasmit csinálok, mint a PowerShell `psbase` nézete:

```
PS C:\Users\Administrator> Get-QADUser "Administrator" | ForEach-Object {$ .
DirectoryEntry}

distinguishedName
-----------------
{CN=Administrator,CN=Users,DC=iqjb,DC=w08}

PS C:\Users\Administrator> Get-QADUser "Administrator" | ForEach-Object {$_.
DirectoryEntry.LastLogon}
```

Na, de hol a `LastLogon` attribútum értéke? Az utolsó parancsnál nem nyomdahiba miatt nincs eredmény, hanem nincs a kifejezésnek semmi visszatérési értéke. Ennek láthatóvá tételére van a `Convert-QADAttributeValue` cmdlet:

```
PS C:\Users\Administrator> Get-QADUser "Administrator" | ForEach-Object {$ .
DirectoryEntry.LastLogon} | Convert-QADAttributeValue -OutputTypeName DateTi
me

2008. szeptember 2. 11:57:23
```

Ez már igazi `DateTime` értéket ad.

Összefoglalásul elmondhatjuk, hogy nagyon kényelmessé teszik az Active Directoryval kapcsolatos munkánkat ezek a cmdletek. A *snap-in*-hez részletes dokumentációt is kapunk, így ezek alapján mindenki maga is felderítheti a Quest cmdleteit.

3.2 PCX – PowerShell Community Extensions

Lelkes PowerShell fejlesztők futtatnak egy projektet, amelyben ügyes kis bővítményeket készítenek a PowerShellhez. Az így létrehozott „termék” telepítője letölthető a projekt weboldaláról[15]. Telepítés után számos változáson esik keresztül a PowerShell környezetünk, ezeknek jó része az alábbi sok új cmdlet, de kapunk új függvényeket, promptokat és providereket is.

```
[4] PS I:\>get-command -PSSnapin pscx

CommandType     Name                           Definition
-----------     ----                           ----------
Cmdlet          ConvertFrom-Base64             ConvertFrom-Base64 [-Base6...
Cmdlet          ConvertTo-Base64               ConvertTo-Base64 [-Path] <...
Cmdlet          ConvertTo-MacOs9LineEnding     ConvertTo-MacOs9LineEnding...
Cmdlet          ConvertTo-UnixLineEnding       ConvertTo-UnixLineEnding [...
Cmdlet          ConvertTo-WindowsLineEnding    ConvertTo-WindowsLineEndin...
Cmdlet          Convert-Xml                    Convert-Xml [-Path] <Strin...
Cmdlet          Disconnect-TerminalSession     Disconnect-TerminalSession...
Cmdlet          Export-Bitmap                  Export-Bitmap [-Bitmap] <B...
Cmdlet          Format-Byte                    Format-Byte [-Value] <Int6...
Cmdlet          Format-Hex                     Format-Hex [-Path] <String...
Cmdlet          Format-Xml                     Format-Xml [-Path] <String...
Cmdlet          Get-ADObject                   Get-ADObject [-Domain <Str...
Cmdlet          Get-Clipboard                  Get-Clipboard [-Text] [-Ve...
Cmdlet          Get-DhcpServer                 Get-DhcpServer [-Server <S...
Cmdlet          Get-DomainController           Get-DomainController [-Ser...
Cmdlet          Get-ExportedType               Get-ExportedType [-Assembl...
Cmdlet          Get-FileVersionInfo            Get-FileVersionInfo [-Path...
Cmdlet          Get-ForegroundWindow           Get-ForegroundWindow [-Ver...
Cmdlet          Get-Hash                       Get-Hash [-Path] <String[]>...
Cmdlet          Get-MountPoint                 Get-MountPoint [[-Volume] ...
Cmdlet          Get-PEHeader                   Get-PEHeader [-Path] <Stri...
Cmdlet          Get-Privilege                  Get-Privilege [[-Identity]...
Cmdlet          Get-PSSnapinHelp               Get-PSSnapinHelp [-Path] <...
Cmdlet          Get-Random                     Get-Random [-Verbose] [-De...
Cmdlet          Get-ReparsePoint               Get-ReparsePoint [-Path] <...
Cmdlet          Get-ShortPath                  Get-ShortPath [-Path] <Str...
Cmdlet          Get-TabExpansion               Get-TabExpansion [-Line] <...
Cmdlet          Get-TerminalSession            Get-TerminalSession [[-Com...
Cmdlet          Import-Bitmap                  Import-Bitmap [-Path] <Str...
Cmdlet          Join-String                    Join-String [-Strings] <St...
Cmdlet          New-Hardlink                   New-Hardlink [-Path] <Stri...
Cmdlet          New-Junction                   New-Junction [-Path] <Stri...
Cmdlet          New-Shortcut                   New-Shortcut [-Path] <Stri...
Cmdlet          New-Symlink                    New-Symlink [-Path] <Strin...
Cmdlet          Out-Clipboard                  Out-Clipboard [-InputObjec...
Cmdlet          Ping-Host                      Ping-Host [-HostName] <PSO...
Cmdlet          Remove-MountPoint              Remove-MountPoint [[-Name]...
Cmdlet          Remove-ReparsePoint            Remove-ReparsePoint [-Path...
Cmdlet          Resize-Bitmap                  Resize-Bitmap [-Bitmap] <B...
Cmdlet          Resolve-Assembly               Resolve-Assembly [-Name] <...
```

[15] http://www.codeplex.com/PowerShellCX

```
Cmdlet          Resolve-Host                    Resolve-Host [-HostName] <...
Cmdlet          Select-Xml                      Select-Xml [-Path] <String...
Cmdlet          Send-SmtpMail                   Send-SmtpMail [-InputObjec...
Cmdlet          Set-Clipboard                   Set-Clipboard [-Text <Stri...
Cmdlet          Set-FileTime                    Set-FileTime [-Path] <Stri...
Cmdlet          Set-ForegroundWindow            Set-ForegroundWindow [[-Ha...
Cmdlet          Set-Privilege                   Set-Privilege [-Privileges...
Cmdlet          Set-VolumeLabel                 Set-VolumeLabel [[-Path] <...
Cmdlet          Split-String                    Split-String [[-Separator]...
Cmdlet          Start-Process                   Start-Process [[-Path] <St...
Cmdlet          Start-TabExpansion              Start-TabExpansion [-Verbo...
Cmdlet          Stop-TerminalSession            Stop-TerminalSession [[-Co...
Cmdlet          Test-Assembly                   Test-Assembly [-Path] <Str...
Cmdlet          Test-Xml                        Test-Xml [-Path] <String[]...
Cmdlet          Write-BZip2                     Write-BZip2 [-Path] <Strin...
Cmdlet          Write-Clipboard                 Write-Clipboard [-Object] ...
Cmdlet          Write-GZip                      Write-GZip [-Path] <String...
Cmdlet          Write-Tar                       Write-Tar [-Path] <String[...
Cmdlet          Write-Zip                       Write-Zip [-Path] <String[...
```

Párat kiemelnék a fontosabbak közül:

cmdlet	funkció
Get-ADObject	Active Directory objektumok kiolvasása
Get-Privilege, Set-Privilege	Felhasználói jogok kilistázása, beállítása
Out-Clipboard	Kimenet átirányítása a vágólapra
Resolve-Host	Névfeloldás DNS alapján
Start-Process	Folyamat elindítása
Split-String	Sztring feldarabolása akár regex kifejezés alapján
Test-XML	XML állomány szintaktikus ellenőrzése

Több olyan cmdletet is felfedezhetünk, amelyek funkcióját itt a könyvben korábban már .NET osztályok közvetlen használatával már kiváltottuk, de mennyivel elegánsabb igazi cmdletet használni.

Az igazán nagy durranás a PCX új providerei:

```
[6] PS I:\>Get-PSProvider

Name                 Capabilities                Drives
----                 ------------                ------
Alias                ShouldProcess               {Alias}
Environment          ShouldProcess               {Env}
FileSystem           Filter, ShouldProcess       {C, A, D, E...}
Function             ShouldProcess               {Function}
Registry             ShouldProcess               {HKLM, HKCU}
Variable             ShouldProcess               {Variable}
Certificate          ShouldProcess               {cert}
FeedStore            ShouldProcess               {Feed}
AssemblyCache        ShouldProcess               {Gac}
DirectoryServices    ShouldProcess, Credentials  {IQJB}
```

Láthatjuk, hogy van egy `FeedStore` nevezetű új provider, amely táplál egy „`feed`” nevezetű PSDrive-ot, amellyel RSS forrásokhoz tudunk csatlakozni:

```
[12] PS I:\>get-childitem feed:

Type     Name                                     ItemCount UnreadItemCount
----     ----                                     --------- ---------------
folder   Microsoft Feeds                            100             100
feed     Microsoft Technet                           92              92
feed     PowerShell Pro!                             21              10
```

Vagy észrevehetjük, hogy maga az Active Directory adatbázis is látható `PSDrive`-ként! Navigáljunk egy keveset ebben a meghajtóban a „fájlkezelő” cmdletekkel:

```
5# cd adatum:
Adatum.com\
6# dir

         LastWriteTime Type                 Name
         ------------- ----                 ----
  2007.01.01.    19:47 builtinDomain        Builtin
  2007.01.01.    19:47 container            Computers
  2007.01.03.    19:05 organizationalUnit   Contractors
  2007.01.01.    19:47 organizationalUnit   Domain Controllers
  2007.01.03.    19:04 organizationalUnit   Finance
  2007.01.01.    19:47 container            ForeignSecurityPrincipals
  2007.01.01.    19:47 infrastructureUpdate Infrastructure
  2007.01.03.    19:04 organizationalUnit   ITAdmins
  2007.01.03.    19:04 organizationalUnit   Legal
  2007.01.01.    19:47 lostAndFound         LostAndFound
  2007.01.03.    19:04 organizationalUnit   Marketing
  2007.01.03.    17:54 organizationalUnit   Microsoft Exchange Security ...
  2008.03.24.     0:02 msExchSystemObjec... Microsoft Exchange System Ob...
  2007.01.01.    19:47 msDS-QuotaContainer  NTDS Quotas
  2007.01.01.    19:47 container            Program Data
  2007.01.03.    19:05 organizationalUnit   Resources
  2007.01.03.    19:04 organizationalUnit   Sales
  2007.01.01.    19:47 container            System
  2007.01.01.    19:47 container            Users


7# cd finance
Adatum.com\finance
8# dir

         LastWriteTime Type                 Name
         ------------- ----                 ----
  2007.01.03.    19:50 group                Finance
  2008.03.31.    14:16 user                 Katie Jordan
  2008.03.31.    14:16 user                 Linda Mitchell
  2007.01.03.    19:49 user                 Qin Hong
```

Nézzünk egy konkrét elemet, mondjuk „Katie Jordan”-t, és az ő tulajdonságait:

```
9# $u = get-item 'Katie Jordan'

22# $u | fl *


PSPath            : Pscx\DirectoryServices::ADATUM:\Finance\Katie Jordan
PSParentPath      : Pscx\DirectoryServices::ADATUM:\Finance
PSChildName       : Katie Jordan
PSDrive           : ADATUM
PSProvider        : Pscx\DirectoryServices
PSIsContainer     : False
Type              : user
Name              : Katie Jordan
Description       :
LastWriteTime     : 2008.03.31. 14:16:59
FullName          : ADATUM:\Finance\Katie Jordan
CanonicalName     : Adatum.com/Finance/Katie Jordan
DistinguishedName : CN=Katie Jordan,OU=Finance,DC=Adatum,DC=com
Types             : {top, person, organizationalPerson, user}
Entry             : System.DirectoryServices.DirectoryEntry
IsContainer       : False
```

Láthatjuk, hogy ez még nem az „igazi” felhasználói objektum, megint csak egy „csomagolást” látunk, a belsejét az `Entry` tulajdonságon keresztül érjük el:

```
23# $u2 = $u.Entry
24# $u2 | fl *


objectClass                     : {top, person, organizationalPerson, user}
cn                              : {Katie Jordan}
sn                              : {Jordan}
givenName                       : {Katie}
distinguishedName               : {CN=Katie Jordan,OU=Finance,DC=Adatum,DC=
                                  com}
instanceType                    : {4}
whenCreated                     : {2007.01.04. 3:14:59}
whenChanged                     : {2008.03.31. 21:16:59}
displayName                     : {Katie Jordan}
uSNCreated                      : {System.__ComObject}
memberOf                        : {CN=Finance,OU=Finance,DC=Adatum,DC=com}
uSNChanged                      : {System.  ComObject}
department                      : {Finance}
company                         : {Contoso, Ltd}
...
```

A felhasználói attribútumok módosítása is nagyon egyszerű, de itt se felejtsük el a memóriában tárolódó adatokat az AD-ba beírni a `setinfo()` metódussal!

```
27# $u2.department = "Sóhivatal"
28# $u2.setinfo()
```

Új felhasználó létrehozása sem nagy ördöngösség:

```
41# new-item -Path "ADATUM:\Finance" -Name "Új júzer" -type user

         LastWriteTime Type                 Name
         ------------- ----                 ----
  2008.05.08.    14:07 user                 Új júzer
```

A fenti AD-val kapcsolatos példák promtjában is látható némi trükk, a kettős kereszt jelzi, hogy rendszergazda jogkörrel ténykedem a gépemen.

Mindenképpen érdemes rendszeresen nézegetni a PCX honlapját, hiszen gyakran jelennek meg frissítések ehhez a bővítménycsomaghoz, ráadásul a forráskódot is le lehet tölteni és meg lehet szemlélni.

3.3 Exchange Server 2007

Sok informatikus első találkozása a PowerShellel az Exchange Server 2007 kapcsán történt meg, nálam is ez a helyzet. Hiszen míg a Windows XP, Server 2003, Vista, sőt még a Windows Server 2008 esetében is opcionális ez a komponens, nem kötelező alkalmazni, addig az Exchange Server 2007 üzemeltetése során nem lehet megkerülni a használatát.

Ebben a fejezetben nem kívánok az Exchange Server 2007 részleteibe belefolyni, így azok is bátran elolvashatják, akik azt nem ismerik. A célom inkább a PowerShell bővítési lehetőségeinek bemutatása, illetve azon eltérések, újdonságok kiemelése az „alap" PowerShellhez képest, amelyek – véleményem szerint – előrevetítik a PowerShell továbbfejlesztésének irányait is.

A PowerShell moduláris, bővíthető parancssori környezet, azaz a meglevő parancskészletet újabbakkal lehet kiegészíteni beépülő modulok (snap-in) telepítésével. Több ilyet már az előző fejezetben is bemutattam, de az Exchange Server 2007 üzemeltetői programcsomagja (*Exchange Management Tools*) is telepít ilyen beépülő modult.

3.3.1 Te jó ég! Csak nem egy újabb parancssori környezet?

Ha az Exchange Server 2007 programcsoportból indítjuk el az *Exchange Management Shell*-t, az elnevezés alapján megijedhetünk, hogy ez egy újabb parancssori környezet, de szerencsére nem.

63. ábra Az Exchange Management Shell helye a Start menüben

Ha rákattintunk az ikonra, akkor egy PowerShell ablakot kapunk, de ez nem teljesen ugyanolyan, mint az „igazi” PowerShell ablak. Ez alaphelyzetben kicsit „csúnya”, fekete a háttere, nincs bekapcsolva a *Quick Edit* üzemmód, kicsi az ablak puffermérete. De természetesen ezeket a hiányosságokat nyugodtan orvosolhatjuk.

A másik különbség az „igazi” PowerShell ablakhoz képest, hogy az ablak fejlécében az ablakot futtató gép neve és az Active Directory valamely ágának megnevezése látható, valamint alaphelyzetben be van töltve az Exchange snap-in, amit le is kérdezhetünk:

```
[PS] C:\>Get-PSSnapin


Name        : Microsoft.PowerShell.Core
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains Windows PowerShell
manag
              ement cmdlets used to manage components of Windows PowerShell.

Name        : Microsoft.PowerShell.Host
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains cmdlets used by the
Wind
              ows PowerShell host.

Name        : Microsoft.PowerShell.Management
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains management cmdlets
used
              to manage Windows components.

Name        : Microsoft.PowerShell.Security
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains cmdlets to manage
Window
              s PowerShell security.

Name        : Microsoft.PowerShell.Utility
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains utility Cmdlets used
to
              manipulate data.

Name        : Microsoft.Exchange.Management.PowerShell.Admin
PSVersion   : 1.0
Description : Admin Tasks for the Exchange Server
```

Ez látható a fenti lista utolsó helyén. Minden más tekintetben ez egy „normális” PowerShell ablak, tehát mindaz alkalmazható, használható benne, amit eddig megszoktunk.

Minek köszönhetők ezek a változások az alap PowerShell konzolhoz képest? Ha megnézzük a fenti parancsikon mögötti parancssort, akkor ezt láthatjuk:

```
C:\WINDOWS\system32\windowspowershell\v1.0\powershell.exe -PSConsoleFile
"E:\Program Files\Microsoft\Exchange Server\bin\exshell.psc1" -noexit -
command ". 'E:\Program Files\Microsoft\Exchange Server\bin\Exchange.ps1'"
```

Nem rövid, a lényege egy konzolfájl és egy szkript közvetlen meghívása a PowerShell környezet indítása során. Nézzük meg ezt a két állományt! Elsőként a konzolfájlt:

```
<?xml version="1.0" encoding="utf-8"?>
<PSConsoleFile ConsoleSchemaVersion="1.0">
  <PSVersion>1.0</PSVersion>
  <PSSnapIns>
    <PSSnapIn Name="Microsoft.Exchange.Management.PowerShell.Admin" />
  </PSSnapIns>
</PSConsoleFile>
```

Itt kerül betöltésre az Exchange Server rendszerkezeléssel kapcsolatos cmdleteket tartalmazó snap-in-je. Nézzük a szkriptet (helyhiány miatt néhány részt kipontoztam):

```
# Copyright (c) Microsoft Corporation. All rights reserved.

## ALIASES
#####################################################################

set-alias list       format-list
set-alias table      format-table

## Confirmation is enabled by default, uncommenting next line will disable
it
# $ConfirmPreference = "None"

## EXCHANGE VARIABLEs
#####################################################

$global:exbin = (get-itemproperty
HKLM:\SOFTWARE\Microsoft\Exchange\Setup).MsiInstallPath + "bin\"
$global:exinstall = (get-itemproperty
HKLM:\SOFTWARE\Microsoft\Exchange\Setup).MsiInstallPath
$global:exscripts = (get-itemproperty
HKLM:\SOFTWARE\Microsoft\Exchange\Setup).MsiInstallPath + "scripts\"
$global:AdminSessionADSettings =
[Microsoft.Exchange.Data.Directory.AdminSessionADSettings]::Instance
## Reset the Default Domain
$global:AdminSessionADSettings.ViewEntireForest = $false

$FormatEnumerationLimit = 16

## PROMPT
#####################################################################

## PowerShell can support very rich prompts, this simple one prints the
current
## working directory and updates the console window title to show the
machine
## name and directory.
```

```
function prompt
{
        $cwd = (get-location).Path
        $scope = "View Entire Forest"
        if (!$AdminSessionADSettings.ViewEntireForest)
        {
                $scope = $AdminSessionADSettings.DefaultScope
        }
        $host.UI.RawUI.WindowTitle = "Machine: " + $(hostname) + " | Scope:
" + $scope
        $host.UI.Write("Yellow", $host.UI.RawUI.BackGroundColor, "[PS]")
        " $cwd>"
}

## FUNCTIONs
#####################################################################
## returns all defined functions
function functions
{
...
}

## generates a tip of the day
function get-tip
{
...
}

## only returns exchange commands
function get-excommand
{
        if ($args[0] -eq $null)
        {
                get-command -pssnapin Microsoft.Exchange*
        }
        else
        {
                get-command $args[0] | where { $_.psSnapin -ilike
'Microsoft.Exchange*' }
        }
}

## only returns PowerShell commands
function get-pscommand
{
...
}

## prints the Exchange Banner in pretty colors
function get-exbanner
{
...
}

## shows quickref guide
function quickref
{
```

```
...
}

function get-exblog
{
       invoke-expression 'cmd /c start
http://go.microsoft.com/fwlink/?LinkId=35786'
}

## now actually call the functions

get-exbanner
get-tip
...
```

Ebben a szkriptben láthatjuk néhány változó definiálását, a prompt és az ablak fejlécének beállítását, néhány függvény definiálását. Érdekes módon az Exchange snap-in által definiált cmdletek kilistázását végző `get-excommand` az nem cmdlet, hanem függvény! Ebből következően például nem működik esetében a tab kiegészítés, és a `get-help` cmdlet sem.

3.3.2 Az Exchange cmdletek feltérképezése

Nem egyszerű ezen Exchange snap-in által biztosított új cmdletek feltérképezése. Nézzük meg, mennyi új cmdletünk van:

```
[PS] C:\>(get-command -PSSnapin "Microsoft.Exchange.Management.PowerShell
.Admin").Count
394
```

Nem kevés, 394 darab új cmdlet, az alap 129-cel szemben! Hogyan lehet ezeket áttekinteni? Szerencsére kapunk segítséget egyrészt a `get-command`, másrészt a `get-help` cmdlettől is. A `get-command` esetében használhatjuk a `verb` és a `noun` paramétereket, amelyekkel be tudjuk határolni a cmdleteket. Például a levelesládákkal kapcsolatos cmdletek listázása:

```
[PS] C:\>get-command -noun mailbox

CommandType     Name                           Definition
-----------     ----                           ----------
Cmdlet          Connect-Mailbox                Connect-Mailbox [-Identity...
Cmdlet          Disable-Mailbox                Disable-Mailbox [-Identity...
Cmdlet          Enable-Mailbox                 Enable-Mailbox [-Identity]...
Cmdlet          Export-Mailbox                 Export-Mailbox [-Identity]...
Cmdlet          Get-Mailbox                    Get-Mailbox [[-Identity] <...
Cmdlet          Import-Mailbox                 Import-Mailbox [[-Identity...
Cmdlet          Move-Mailbox                   Move-Mailbox [-Identity] <...
Cmdlet          New-Mailbox                    New-Mailbox [-Name] <Strin...
Cmdlet          Remove-Mailbox                 Remove-Mailbox [-Identity]...
Cmdlet          Restore-Mailbox                Restore-Mailbox [-Identity...
```

```
Cmdlet          Set-Mailbox                   Set-Mailbox [-Identity] <M...
```

Vagy az összes exportálást végző cmdletek:

```
[PS] C:\>get-command -verb export

CommandType     Name                          Definition
-----------     ----                          ----------
Cmdlet          Export-ActiveSyncLog          Export-ActiveSyncLog -File...
Cmdlet          Export-Alias                  Export-Alias [-Path] <Stri...
Cmdlet          Export-AutoDiscoverConfig     Export-AutoDiscoverConfig ...
Cmdlet          Export-Bitmap                 Export-Bitmap [-Bitmap] <B...
Cmdlet          Export-Clixml                 Export-Clixml [-Path] <Str...
Cmdlet          Export-Console                Export-Console [[-Path] <S...
Cmdlet          Export-Csv                    Export-Csv [-Path] <String...
Cmdlet          Export-ExchangeCertificate    Export-ExchangeCertificate...
Cmdlet          Export-Mailbox                Export-Mailbox [-Identity]...
Cmdlet          Export-Message                Export-Message [-Identity]...
Cmdlet          Export-TransportRuleCollec... Export-TransportRuleCollec...
```

Ha még ez sem lenne kellő segítség a keresett cmdlet megtalálásához, akkor a `get-help` esetében használhatjuk a `role`, `component` és `functionality` paramétereket. Nézzük, mik lehet a `role` paraméter lehetséges értékei:

Érték		Jelentés
client	***ca***	Client Access Server
edge	***et***	Edge Transport server
hub	***ht***	Hub Transport server
mail	***mb***	Mailbox server
unified	***um***	Unified Messaging server
org	***oa***	Exchange Organization
rpct	***ra***	Exchange Recipient
srv	***sa***	Exchange Server
win	***wa***	Windows
read	***ro***	„Read only” jellegű cmdletek

A `component` paraméter lehetséges értékei:

Érték	Jelentés
addressing	Címlisták és e-mail cím házirendek
agent	Transport Agent, Content/Sender Filtering, IP Allow/Block, és Transport Rules
antispam	IP Allow/Block, Content/Sender Filtering, és Anti Spam
autoDiscover	Autodiscover
calendaring	Rendelkezésre állási információk és erőforrás fiókok naptárkezelése
certificate	Exchange SSL tanusítványok

classification	Message Classification és Transport Rule
client	Outlook Anywhere és CAS Mailbox
cluster	Clustered Mailbox Server
compliance	Transport Rule és Journal Rule
delegate	Exchange Administrator rendszergazda jogosultságok
diagnostic	Message Queue és tesztelés
domain	Remote és Accepted Domain
extensibility	Transport Agent és Web Services Virtual Directory
freebusy	Availability Service konfiguráció
gal	Address List és Offline Address Book
group	Group/Distribution Group/Dynamic Distribution Group
highavailability	Clustered Mailbox Server és Local/Cluster Continuous Replication
imap	IMAP és CAS Mailbox
mailbox	Mailbox, UM Mailbox és postaláda jogosultságok
mailflow	Transport Agent, Message Queue, Accepted/Remote Domain, Receive/Send Connector, Edge Subscription, és Routing Group Connector
managedfolder	Messaging Records Management
mobility	ActiveSync
oab	Offline Address Book
outlook	Outlook Provider és Outlook Anywhere
owa	OWA és Exchange Web Services Virtual Directory
permission	Active Directory Permission és Mailbox Permission
pop	POP3
publicfolder	Public Folder
queuing	Message és Message Queue
recipient	Mail-enabled object (mailbox, contact, user)
routing	Accepted Domain, AD Site Link és Routing Group Connector
rule	Message Classification és Transport/Journal Rule
search	Exchange Search
server	Exchange Server
statistics	Mailbox/Public Folder/Logon Statistic
storage	Storage Group és Mailbox/Public Folder adatbázis
um	Unified Messaging
virtualdirectory	Client Access Server Virtual Directory

A `Functionality` paraméter lehetséges értékei:

Érték	Jelentés
Global	Exchange szervezet szintje
Server	Kiszolgáló szintje
User	Címzettek szintje

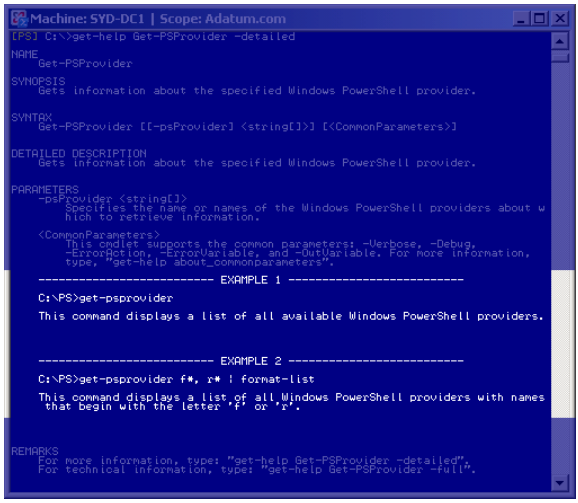
Például a mailbox szerepű kiszolgálókon használható, szervezetszintű jogosultságok beállításával kapcsolatos cmdletek listáját az alábbi módon kérhetjük le:

```
[PS] C:\>get-help -role *mail* -component *permission* -functionality global


Name                      Category                  Synopsis
----                      --------                  --------
Get-ADPermission          Cmdlet                    Use the Get-ADPermiss...
Add-ADPermission          Cmdlet                    Use the Add-ADPermiss...
Remove-ADPermission       Cmdlet                    Use the Remove-ADPerm...
```

Ez már elég jól beszűkítette az eredetileg 394 darabos listát.

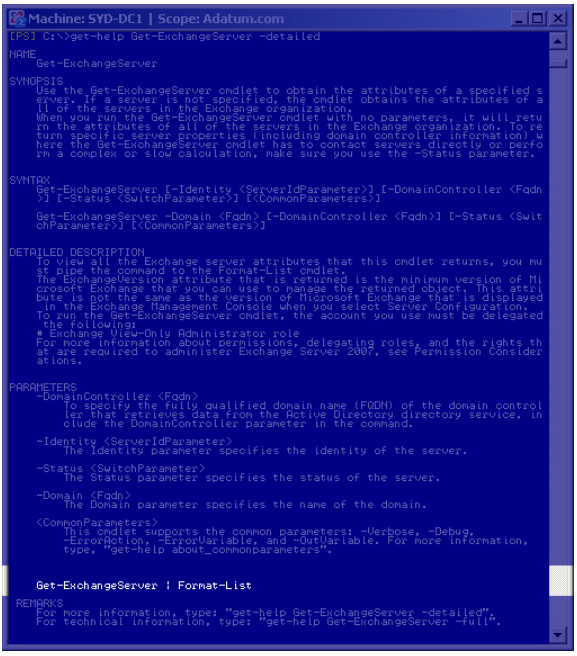
3.3.3 Help szerkezete

Némiképp meglátszik a súgó szerkezetén, hogy más „műhelyben” készült az alap PowerShell, mint az Exchange Server 2007 beépülő modul. Nézzük, hogy hogyan néz ki egy alap súgó:



64. ábra Eredeti PowerShell help

Láthatjuk, hogy szépen elkülönülő példák vannak feltüntetve. Ezzel szemben egy Exchange cmdletben a példák nem ennyire részletesek, és formailag sem vehetők annyira észre:



65. ábra Exchange cmdlet helpje

Valószínű ennek oka az, hogy az Exchange cmdletek jóval összetettebbek, gyakran egymásra épülnek, így a súgó keretei között nehéz lenne értelmes példákat bemutatni.

3.3.4 Felhasználó- és csoportkezelés

Az Exchange rendszergazda tevékenységei közé a felhasználók és csoportok kezelése is hozzátartozik, így az Exchange PowerShell snap-in tartalmaz ezzel kapcsolatos cmdleteket is. Az alap PowerShellben az `[ADSI]` típusjelölővel lehet hivatkozni AD objektumokra, mint ahogy láttuk ezt a *2.9 Felhasználó-menedzsment, Active Directory* fejezetben, az Exchange környezetben még egyszerűbb a helyzet. Ugyan nem teljes értékű az AD felhasználó-menedzsment parancskészlete, hanem kifejezetten csak az Exchange Server igényeire szabott, de azért így is sokat profitálhatunk belőle. Nézzük például a `get-user` cmdletet:

```
 [PS] C:\>get-user brian

Name                                    RecipientType
----                                    -------------
Brian Cox                               UserMailbox
```

Nézzük meg kicsit részletesebben Brian-t:

```
[PS] C:\>get-user brian | fl


IsSecurityPrincipal      : True
SamAccountName           : Brian
Sid                      : S-1-5-21-150787130-2833054149-3883060369-1132
SidHistory               : {}
UserPrincipalName        : Brian@Adatum.com
ResetPasswordOnNextLogon : False
AssistantName            :
City                     :
Company                  :
CountryOrRegion          :
Department               : Legal
DirectReports            : {}
DisplayName              : Brian Cox
Fax                      :
FirstName                : Brian
HomePhone                :
Initials                 :
LastName                 : Cox
Manager                  :
MobilePhone              :
Notes                    :
Office                   :
OtherFax                 : {}
OtherHomePhone           : {}
OtherTelephone           : {}
Pager                    :
Phone                    :
PhoneticDisplayName      :
PostalCode               :
PostOfficeBox            : {}
RecipientType            : UserMailbox
RecipientTypeDetails     : UserMailbox
SimpleDisplayName        :
StateOrProvince          :
StreetAddress            :
Title                    :
UMDialPlan               :
UMDtmfMap                : {}
AllowUMCallsFromNonUsers : SearchEnabled
WebPage                  :
TelephoneAssistant       :
WindowsEmailAddress      : Brian@adatum.com
IsValid                  : True
OriginatingServer        : SYD-DC1.Adatum.com
ExchangeVersion          : 0.1 (8.0.535.0)
Name                     : Brian Cox
DistinguishedName        : CN=Brian Cox,OU=Legal,DC=Adatum,DC=com
Identity                 : Adatum.com/Legal/Brian Cox
Guid                     : debe1f52-f0ea-4da0-b049-8f22c2d45db2
ObjectCategory           : Adatum.com/Configuration/Schema/Person
ObjectClass              : {top, person, organizationalPerson, user}
WhenChanged              : 2007.01.03. 19:50:42
WhenCreated              : 2007.01.03. 19:17:26
```

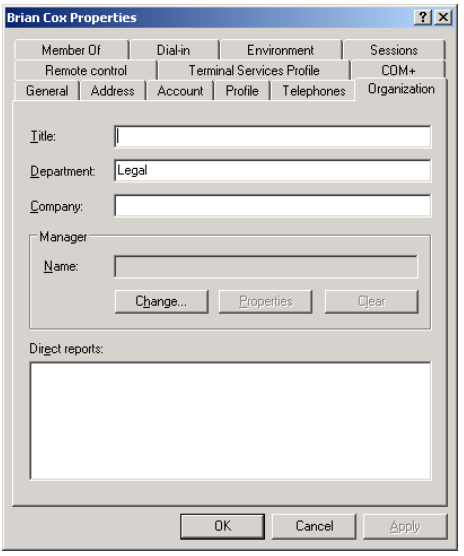
Látható, hogy nagyon sok tulajdonság lekérdezhető a felhasználói fiókokkal kapcsolatban. Az Exchange-el kapcsolatos attribútumokat nem ezzel, hanem a `get-mailbox` cmdlettel lehet lekérdezni.

3.3.5 Get és Set

A fenti táblázatban Brian adatai között a `Title` (beosztás) paraméter üres. Próbáljuk ezt feltölteni:

```
[PS] C:\>$u = Get-User brian
[PS] C:\>$u.title = 'Portás'
[PS] C:\>$u.title
Portás
```

Látszólag sikerült beállítani Brian beosztását, de ha az *Active Directory Users and Computer* eszközzel megnézzük Briant, akkor nem látjuk ezt a paramétert:



66. ábra Beállítottam a beosztást, mégsem látszik

Mi történt akkor? Az AD-vel kapcsolatos `Get-...` kezdetű cmdletek nem direktben dolgoznak a címtárban, hanem a címtár-információkat betöltik a memóriába. Így az értékadás utasításom is a memóriában hajtódott végre és nem ténylegesen a felhasználói fiókon. Ezért van az AD-val kapcsolatos `Get-...` cmdleteknek egy `Set-...` párjuk is, mert azok már ténylegesen beírják az értékeket a címtár adatbázisába. Nézzük meg, hogy hogyan lehet ténylegesen címtárparamétereket megváltoztatni a `Set-User` cmdlettel:

```
[PS] C:\>Set-User brian -Title "Portás"
[PS] C:\>Get-User brian | fl Name, Title


Name  : Brian Cox
Title : Portás
```

Vigyázni kell, hogy nem teljesen szimmetrikusak a `Get-...` és a `Set-...` cmdlet-párok. Jól mutatja ezt a csoportok kezelését végző `Get-Group` és a `Set-Group` páros:

```
[PS] C:\>get-group sales | fl


DisplayName          :
GroupType            : Universal, SecurityEnabled
ManagedBy            :
SamAccountName       : Sales
Sid                  : S-1-5-21-150787130-2833054149-3883060369-1121
SidHistory           : {}
SimpleDisplayName    :
RecipientType        : Group
RecipientTypeDetails : UniversalSecurityGroup
WindowsEmailAddress  :
Notes                :
Members              : {Jeff Hay, Don Hall, Judy Lew}
PhoneticDisplayName  :
IsValid              : True
OriginatingServer    : SYD-DC1.Adatum.com
ExchangeVersion      : 0.0 (6.5.6500.0)
Name                 : Sales
DistinguishedName    : CN=Sales,OU=Sales,DC=Adatum,DC=com
Identity             : Adatum.com/Sales/Sales
Guid                 : 5ec43b9a-8789-4437-8abb-b25970bcdd88
ObjectCategory       : Adatum.com/Configuration/Schema/Group
ObjectClass          : {top, group}
WhenChanged          : 2007.01.03. 19:12:41
WhenCreated          : 2007.01.03. 19:09:04
```

A fenti példából látszik, hogy a csoport tagjai (`Members`) lekérdezhető a `get-group`-pal. Nézzük meg a `Set-Group` szintaxisát:

```
Set-Group
       -Identity <GroupIdParameter>
       [-Confirm [<SwitchParameter>]]
       [-DisplayName <String>]
       [-DomainController <Fqdn>]
       [-IgnoreDefaultScope <SwitchParameter>]
       [-ManagedBy <GeneralRecipientIdParameter>]
       [-Name <String>]
       [-Notes <String>]
       [-PhoneticDisplayName <String>]
```

```
        [-SimpleDisplayName <String>]
        [-WhatIf [<SwitchParameter>]]
        [-WindowsEmailAddress <SmtpAddress>]
        [<CommonParameters>]
```

De hol van a `Members` tulajdonság beállításához szükséges paraméter? Hát ez hiányzik. Mindez azt bizonyítja, hogy az Exchange cmdletek nem próbálják meg magukra vállalni az Active Directory kezelését, arra vagy az `[ADSI]` típushivatkozást, vagy külső gyártó snap-in-jét használhatjuk.

3.3.6 Új paraméterfajta: Identity

Az előző példákban elég nagyvonalúan hivatkoztam Brianre, semmi *distinguished name*, semmi *canonical name* nem szerepelt a hivatkozásban, holott például az `[ADSI]` használatánál teljes, pontos distinguished name kitöltésére volt szükség. Mi teszi lehetővé ezt az egyszerű, kényelmes használatot? Nézzük meg a `get-user` súgóját:

```
[PS] C:\>get-help Get-User -full

NAME
    Get-User

SYNOPSIS
    Use the Get-User cmdlet to retrieve all users in the forest that match
    the specified conditions.

SYNTAX
    get-User [-Identity <UserIdParameter>] [-Credential <PSCredential>] [-D
    omainController <Fqdn>] [-IgnoreDefaultScope <SwitchParameter>] [-Organ
    izationalUnit <OrganizationalUnitIdParameter>] [-ReadFromDomainControll
    er <SwitchParameter>] [-RecipientTypeDetails <RecipientTypeDetails[]>]
    [-ResultSize <Unlimited>] [-SortBy <String>] [<CommonParameters>]

    get-User [-Credential <PSCredential>] [-DomainController <Fqdn>] [-Filt
    er <String>] [-IgnoreDefaultScope <SwitchParameter>] [-OrganizationalUn
    it <OrganizationalUnitIdParameter>] [-ReadFromDomainController <SwitchP
    arameter>] [-RecipientTypeDetails <RecipientTypeDetails[]>] [-ResultSiz
    e <Unlimited>] [-SortBy <String>] [<CommonParameters>]

    get-User [-Anr <String>] [-Credential <PSCredential>] [-DomainControlle
    r <Fqdn>] [-IgnoreDefaultScope <SwitchParameter>] [-OrganizationalUnit
    <OrganizationalUnitIdParameter>] [-ReadFromDomainController <SwitchPara
    meter>] [-RecipientTypeDetails <RecipientTypeDetails[]>] [-ResultSize <
    Unlimited>] [-SortBy <String>] [<CommonParameters>]
...

PARAMETERS
...
    -Identity <UserIdParameter>
        The Identity parameter takes one of the following values:
        * GUID
        * Distinguished name (DN)
```

```
        * Domain\Account
        * User principal name (UPN)
        * Legacy Exchange DN
        * Simple Mail Transfer Protocol (SMTP) address
        * Alias

        Required?                    false
        Position?                    1
        Default value
        Accept pipeline input?       True
        Accept wildcard characters?  true
...
```

Látszik, hogy a többfajta szintaxisa alapján az első paramétert vagy `ANR` (*Ambigous Name Resolution*) névfeloldásra, vagy ehhez nagyon hasonló `Identity` típusú névfeloldásra használja. Az `Identity` paraméter magyarázatát a fenti helprészletben láthatjuk is. Azt, hogy most melyiket használta nem tudjuk, hiszen mindkettő jó eredményre vezet:

```
[PS] C:\>Get-User -anr brian

Name                                  RecipientType
----                                  -------------
Brian Cox                             UserMailbox


[PS] C:\>Get-User -identity brian

Name                                  RecipientType
----                                  -------------
Brian Cox                             UserMailbox
```

Mi ebből a tanulság? Hogy az Exchange cmdletek kialakításánál törekedtek arra a fejlesztők, hogy a lehető legegyszerűbben lehessen hivatkozni az AD objektumokra. Elég hamar elmenne a kedvünk a szkriptek írásától, ha minden esetben csak a teljes *distinguished name* kiírásával lehetne hivatkozni az AD objektumokra, így azonban az `Identity` (és a felhasználókkal kapcsolatban az `ANR`) paraméterrel elég csak olyan mértékig utalni a keresett objektumokra, amikor már egyértelmű a cmdlet számára, hogy kire vagy mire gondoltunk.

Például a `get-MailboxDatabase` cmdletnél az `Identity` paraméter a következőket jelentheti:

```
[PS] C:\>get-help get-mailboxdatabase -Parameter Identity

-Identity <DatabaseIdParameter>
    The Identity parameter specifies a mailbox database. You can use the fo
    llowing values:
    * GUID
    * Distinguished name (DN)
    * Server\storage group\database name
    * Server\database name
    * Storage groupname\database name
```

```
    If you do not specify the server name, the cmdlet will search for datab
    ases on the local server. If you have multiple databases with the same
    name, the cmdlet will retrieve all databases with the same name in the
    specified scope.

    Required?                    false
    Position?                    1
    Default value
    Accept pipeline input?       True
    Accept wildcard characters?  true
```

Azaz, a megjegyzés alapján, akár elég csak az adatbázis nevét megadni, ha az egyértelmű, és nem kell az adatbázis *Distinguished* nevét használni, ami elég elrettentő lenne:

```
[PS] C:\>(Get-MailboxDatabase "syd-dc1\mailbox database").DistinguishedName
CN=Mailbox Database,CN=First Storage Group,CN=InformationStore,CN=SYD-DC1,C
N=Servers,CN=Exchange Administrative Group (FYDIBOHF23SPDLT),CN=Administrat
ive Groups,CN=Adatum Organization,CN=Microsoft Exchange,CN=Services,CN=Conf
iguration,DC=Adatum,DC=com
```

Amit a fenti példában kiemeltem három és fél sor terjedelemben, az egy mailbox adatbázis *distinguished* neve. Nem hiszem, hogy bárki szívesen gépelne ilyen hosszú nevű paramétereket.

3.3.7 Filter paraméter

Másik hatékonyságnövelő paraméter számos Exchange cmdletnél a `Filter` paraméter. Nézzük meg, hogy eddigi ismereteink alapján hogyan keresnénk meg az összes olyan felhasználónkat, akiknek ki van töltve a beosztás tulajdonságuk:

```
[PS] C:\>get-user | Where-Object {$ .Title -ne ""} | Format-Table -Property
DisplayName, Title

DisplayName                          Title
-----------                          -----
Brian Cox                            Portás
Kim Akers                            Naplopó
```

Hogyan hajtódik ez végre? A `get-user` cmdlettel az összes felhasználói objektumot kigyűjtjük, és maga a PowerShell környezet válogatja ki közülük azokat, amelyeknél a `Title` paraméter nem üres sztring. Azaz az egész feldolgozás terhe áttevődik a kliensoldalra. Ez egy több tízezres felhasználószámnál komoly feldolgozási munkát igényel. Ezért kidolgoztak egy kiszolgáló-oldali feldolgozási lehetőséget is az ilyen jellegű AD cmdleteknél, ezt pedig a `Filter` paraméter biztosítja. Itt úgynevezett *OPATH* filtereket, vagy más szóval „*prescanned filter*”-t (elő feldolgozott szűrő) használhatunk, amely szűrőfeltételt a szerver-oldal hajt végre.

```
[PS] C:\>Get-User -filter {Title -ne $null} | Format-Table -Property Display
Name, Title

DisplayName                           Title
-----------                           -----
Brian Cox                             Portás
Kim Akers                             Naplopó
```

Ezzel jóval hatékonyabb és gyorsabb a feldolgozás. Azonban tudni kell, hogy mely attribútumokra lehet így hivatkozni, és milyen szintaxissal. Az alábbi táblázatban a legfontosabb szűrhető attribútumok listája található, természetesen nem mindegyik felhasználói attribútum, hanem vannak köztük `group`, `mailbox`, `contact`, `public folder` objektumok tulajdonságainak elnevezései is:

AcceptMessagesOnlyFrom	ExternalEmailAddress	Phone
AcceptMessagesOnlyFromDLMembers	ExternalOofOptions	PhoneticDisplayName
ActiveSyncAllowedDeviceIDs	Fax	PoliciesExcluded
ActiveSyncDebugLogging	FirstName	PostalCode
ActiveSyncEnabled	ForwardingAddress	PostOfficeBox
ActiveSyncMailboxPolicy	GrantSendOnBehalfTo	PrimarySmtpAddress
AddressListMembership	GroupType	ProhibitSendQuota
Alias	Guid	ProhibitSendReceiveQuota
AllowUMCallsFromNonUsers	HiddenFromAddressListsEnabled	PublicFolderContacts
AssistantName	HomePhone	PublicFolderType
CallAnsweringAudioCodec	IncludedRecipients	RecipientFilter
City	Initials	RecipientLimits
Company	IsLinked	RecipientType
CountryOrRegion	IsMailboxEnabled	RecipientTypeDetails
CustomAttribute1	IsResource	RejectMessagesFrom
CustomAttribute10	IsShared	RejectMessagesFromDLMembers
CustomAttribute11	IssueWarningQuota	ResourceCapacity
CustomAttribute12	LanguagesRaw	ResourceCustom
CustomAttribute13	LastName	RetainDeletedItemsFor
CustomAttribute14	LinkedMasterAccount	RulesQuota
CustomAttribute15	ManagedBy	SamAccountName
CustomAttribute2	ManagedFolderMailboxPolicy	SendOofMessageToOriginatorEnabled
CustomAttribute3	Manager	ServerName
CustomAttribute4	MaxBlockedSenders	SimpleDisplayName
CustomAttribute5	MaxReceiveSize	StateOrProvince
CustomAttribute6	MaxSafeSenders	StreetAddress
CustomAttribute7	MaxSendSize	TelephoneAssistant
CustomAttribute8	MemberOfGroup	Title

CustomAttribute9	MobilePhone	UMDtmfMap
Database	Name	UMEnabled
Department	Notes	UMMailboxPolicy
DisplayName	Office	UMRecipientDialPlanId
DistinguishedName	OfflineAddressBook	UseDatabaseQuotaDefaults
EmailAddresses	OperatorNumber	UserPrincipalName
EmailAddressPolicyEnabled	OtherFax	WhenChanged
ExchangeGuid	OtherHomePhone	WhenCreated
ExchangeVersion	OtherTelephone	WindowsEmailAddress
ExpansionServer	Pager	

A `Filter` paraméterben is használhatjuk a PowerShellben megszokott összehasonlító operátorokat és az ott alkalmazott szintaxist.

3.4 Adatok vizualizálása, PowerGadgets

A PowerGadgets[16] eszköz segítségével a PowerShell konzolos kimenetét lehet egyszerűen vizualizálni, mindenféle színes, árnyékolt háromdimenziós, kör-, oszlop- és fánkdiagramban lehet kirajzoltatni az eredményeket. Azonban ehhez ellenőrizni kell, hogy a megjelenítendő adatok megfelelőek-e, azaz tényleg számokat akarunk-e grafikonokon megjeleníteni. Erre a problémára jól világít rá a `get-mailboxstatistics` Exchange cmdlet, amellyel ki lehet olvasni az egyes levelesládák méreteit.

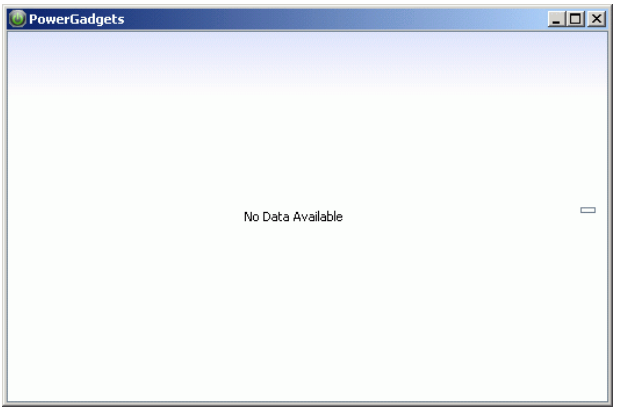
Nézzük, hogy mit is ad ez a cmdlet eredményül, egyelőre a konzolon:

```
[PS] C:\>Get-MailboxStatistics | where-object {$_.DisplayName -notlike "*Sys
tem*" } | format-table -Property displayname, totalitemsize

DisplayName                            TotalItemSize
-----------                            -------------
Főnök                                  17853B
Brian Cox                              8539B
Beth Gilchrist                         6926B
Beosztott                              33982B
Administrator                          61348387B
Qin Hong                               2819B
Ron Gabel                              2819B
Katarina Larsson                       14838B
Titkárnő                               14549B
tárgyaló                               40276B
Linda Mitchell                         22350817B
Kim Akers                              30263144B
Katie Jordan                           8755060B
Gregory Weber                          5287489B
```

A kifejezésem azért nem egy puszta `get-mailboxstatistics`, mert az tartalmazna mindenféle „belső” levelesládát, ezért ezeket a `where-object` cmdlettel kiszűröm, plusz a kifejezésem végén a `format-table`-lel csak az engem érdeklő oszlopokat jelenítem meg. Első ránézésre jónak tűnik az eredmény, csövezzük bele a PowerGadgets `out-chart` cmdletjébe:

```
[PS] C:\>Get-MailboxStatistics | where-object {$ .DisplayName -notlike "*Sys
tem*" } | ft -Property displayname, totalitemsize | out-chart
```

[16] Letölthető demo verzió: http://www.softwarefx.com/sfxSqlProducts/powergadgets/



67. ábra A get-mailboxstatistics alapban üres diagramot ad

Hát ennek a kimenete nem túl szép! Mi lehet az ok? Az a "B" betű a `get-mailboxstatistics` kimenetének második oszlopában minden szám végén. Túl okos akar lenni az Exchange cmdlet, jelzi, hogy ezt az értéket bájtban kell értelmezni, de ez épp elég ahhoz, hogy a számokra váró `out-chart`-ot szomorúvá tegye.

Próbáljunk ebből a „túl okos" értékből normális számot csinálni, ehhez vizsgáljuk meg, hogy mi is a kimenete a `get-mailboxstatistics`-nak, egyelőre egy konkrét levelesláda esetében a `TotalItemSize` tulajdonságát:

```
[PS] C:\>(Get-MailboxStatistics titkárnő).TotalItemSize | Get-Member -Member
Type properties


   TypeName: Microsoft.Exchange.Data.Unlimited`1[[Microsoft.Exchange.Data.B
yteQuantifiedSize, Microsoft.Exchange.Data, Version=8.0.0.0, Culture=neutra
l, PublicKeyToken=31bf3856ad364e35]]

Name        MemberType Definition
----        ---------- ----------
IsUnlimited Property   System.Boolean IsUnlimited {get;}
Value       Property   Microsoft.Exchange.Data.ByteQuantifiedSize Value ...
```

Ebből még nem sok minden derült ki, de nem gond, akkor vegyük a `TotalItemSize.Value` értéket, az hátha jobb:

```
[PS] C:\>(Get-MailboxStatistics titkárnő).TotalItemSize.Value

```

Hoppá! Ez meg nem adott semmit! Hát az hogy lehet? Hiszen az eredeti `get-mailboxstatistics` az adott értéket. Nézzük meg ennek tagjellemzőit:

```
 [PS] C:\>(Get-MailboxStatistics titkárnő).TotalItemSize.Value | Get-Member


   TypeName: Microsoft.Exchange.Data.ByteQuantifiedSize

Name           MemberType Definition
----           ---------- ----------
CompareTo      Method     System.Int32 CompareTo(ByteQuantifiedSize other)
Equals         Method     System.Boolean Equals(Object obj), System.Boole...
GetHashCode    Method     System.Int32 GetHashCode()
GetType        Method     System.Type GetType()
RoundUpToUnit Method     System.UInt64 RoundUpToUnit(Quantifier quantifier)
ToBytes        Method     System.UInt64 ToBytes()
ToGB           Method     System.UInt64 ToGB()
ToKB           Method     System.UInt64 ToKB()
ToMB           Method     System.UInt64 ToMB()
ToString       Method     System.String ToString(), System.String ToStrin...
ToTB           Method     System.UInt64 ToTB()
```

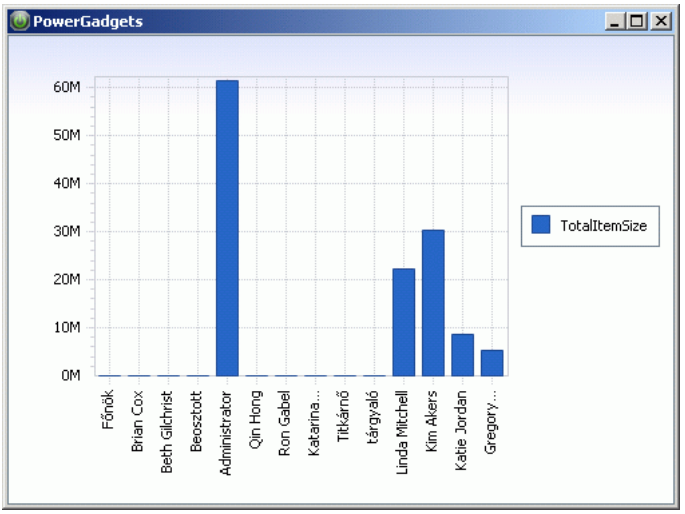
Látszik, hogy a `Value` tagjellemzőnek nincs is "property" típusú tulajdonsága, csak metódusa. Ezért nem kaptunk semmi kimenetet az előbb! Ahhoz, hogy értéket kapjunk meg kell hívni például a `ToBytes()` metódust, és akkor már igazi számot kapunk:

```
[PS] C:\>(Get-MailboxStatistics titkárnő).TotalItemSize.Value.ToBytes()
14549
```

Ez már jó. Ennek ismeretében alakítsuk át az eredeti kifejezésünket! Még egy trükkjét kihasználom itt a PowerGadgets-nek, azt, hogy 5 másodperces frissítéssel is ki lehet rajzoltatni a megadott kifejezés eredményét, így folyamatosan újrarajzolja a grafikont az éppen aktuális helyzetnek megfelelően:

```
[PS] C:\>Get-MailboxStatistics | where-object {$_.DisplayName -notlike "*Sys
tem*" } | select-object displayname, @{n="TotalItemSize"; e={$_.totalitemsiz
e.Value.ToBytes()}} | out-chart -Refresh 0:0:5
```

Itt egy kicsit trükközni kellett, hiszen itt most nem csak meglevő tulajdonságértékeket kell kiírni, hanem egy metódussal új tulajdonságot kell számoltatni. Erre nem alkalmas a `format-table`, hanem a `select-object` cmdletet kell segítségül hívni, amely egy hashtábla kifejezéssel képes kiszámolni az új tulajdonságértéket, mint ahogy láttuk ezt a *1.10.4 Objektumok átalakítása (Select-Object)* fejezetben.



68. ábra A jó diagram új tulajdonságérték kiszámolása után

Ez már teljesen szép és jó eredményt adott. Sőt! Látjuk, hogy a PowerGadgets is okos, hiszen ő is átszámolta a számokat, immár megabájtba, tehát felesleges volt az Exchange cmdlet okoskodása.

3.5 Kitekintés a PowerShell 2.0-ra

A könyv írásának időpontjában a PowerShell 2.0 még nagyon kezdeti fejlesztési fázisban van. Elérhető egy u.n. Community Technology Preview változat, amellyel a Microsoft szondázza a felhasználói visszajelzéseket az egyes tervezett új funkciókkal kapcsolatban. Ezek az új funkciók jelenleg a következők:

Remoting

A PowerShell parancsok távoli gépen történő futtatását teszi lehetővé. Ehhez olyan új fogalmakra van szükség, mint *RunSpace*, azaz futtatási környezet, amelyet létre lehet hozni távoli gépen, ott el lehet indítani PowerShell szkripteket, a kapcsolatot lehet bontani, majd később ugyanehhez a futtatási környezethez csatlakozva láthatjuk a végeredményt.

Új és továbbfejlesztett WMI cmdletek

Az „univerzális” `Get-WMIObject` cmdlet tovább okosodik, DCOM autentikációval, *impersonation* szintekkel, ami lehetővé teszi IIS 6.0 kiszolgálók kezelését is. További WMI cmdletek is megjelennek:

- `Invoke-WMIMethod`
- `Remove-WMIObject`
- `Set-WMIInstance`

Hibakeresés

PowerShell 2.0 várhatóan tartalmazni fog egy cmdlet-alapú debuggert, amellyel megszakítási pontokat lehet definiálni, ki- és belépni lehet szkriptekbe. Mindezt magából a konzolablakból, azaz nincs feltétlenül szükség a PowerGUI szkriptszerkesztőjére, vagy más grafikus szkriptszerkesztőre.

Background Jobs

PowerShell 1.0-ban, ha elindítunk egy parancsot, akkor egészen addig nem kapjuk vissza a promptot, amíg az végre nem hajtódik. A PowerShell 2.0-ban várhatóan aszinkron módon is lehet parancsokat végrehajtatni a háttérben. Ezzel a lehetőséggel párhuzamosan lehet sok PowerShell szkriptet végrehajtatni.

Graphical PowerShell

Várhatóan a mostani külső gyártól által adott szkripteditorokhoz hasonló grafikus alkalmazást is fog tartalmazni a PowerShell 2.0: színnel jelzett szintaxis, azonnali változóki-értékelés, stb.

Új és megváltozott cmdletek

A meglevő cmdletek közül számos fejlődni fog, újabb paraméterekkel lesznek használhatók. Új cmdlet lesz az `Out-GridView`, mellyel a gyakori táblázatos outputot lehet

igazi táblázatként: grafikus rácsban megtekinteni, az oszlopok átméretezésével, mozgatásával, sorrendváltoztatással.

ScriptCmdlets

Új cmdleteket eddig sem volt probléma készíteni, csak a C# vagy a Visual Basic .NET nyelvek és a VisualStudio ismeretére volt szükség. A PowerShell 2.0-ban akár szkripttel is lehet új cmdleteket készíteni, így nem csak a fejlesztők privilégiuma lesz ez az alkotó munka.

4. Hasznos linkek

Az alábbi linklista kb. 7 hónap gyűjtőmunkájának eredménye. Ez a könyv nem jöhetett volna létre, ha ezek az információk nem állnak rendelkezésemre. Természetesen nem az itteni információkat emeltem át egy az egyben, hanem ezek a weboldalak inspiráltak arra engem, hogy bizonyos témákat járjak jobban körül, illetve ötleteket adtak arra vonatkozólag, hogy milyen témákat érintsek.

Sok ezen linkek közül nem statikus oldal, hanem dinamikus tartalom, blog, folyóirat jellegű oldal, így érdemes ezeket rendszeresen látogatni.

PowerShell QuickStart:

http://channel9.msdn.com/wiki/default.aspx/Channel9.WindowsPowerShellQuickStart

PowerShell Wiki:

http://channel9.msdn.com/wiki/default.aspx/Channel9.WindowsPowerShellWiki

Scripting Ezine:

http://www.computerperformance.co.uk/ezine/

Tip of the Week (Script Center):

http://www.microsoft.com/technet/scriptcenter/resources/pstips/archive.mspx

Gyorsbillentyűk:

http://www.microsoft.com/technet/scriptcenter/topics/winpsh/manual/hotkeys.mspx

.NET formázási kifejezések:

http://msdn2.microsoft.com/en-us/library/fbxft59x(vs.71).aspx

Egyedi objektum, add-member:

http://thepowershellguy.com/blogs/posh/rss.aspx?Tags=PsObject&AndTags=1

http://blog.sapien.com/index.php/2008/02/20/how-can-i-write-a-powershell-function-that-outputs-a-table/

http://technet.microsoft.com/en-us/library/bb978596.aspx

Egyedi típus:

http://msdn2.microsoft.com/en-us/library/cc136149(VS.85).aspx

http://technet.microsoft.com/en-us/library/bb978568.aspx

http://blogs.msdn.com/powershell/archive/2006/06/24/645981.aspx

http://www.hanselman.com/blog/MakingJunctionsReparsePointsVisibleInPowerShell.aspx

Regex:

http://msdn2.microsoft.com/en-us/library/hs600312.aspx

http://www.regular-expressions.info/

http://regexlib.com/

http://www.weitz.de/regex-coach/

http://www.sellsbrothers.com/tools/#regexd

http://www.ultrapico.com/ExpressoDownload.htm

Keresés az Active Directory-ban:

http://www.microsoft.com/technet/scriptcenter/resources/qanda/nov06/hey1109.mspx

http://blogs.technet.com/benp/archive/2007/03/26/searching-the-active-directory-with-powershell.aspx

http://www.computerperformance.co.uk/powershell/powershell_active_directory.htm

http://www.microsoft.com/technet/scriptcenter/topics/winpsh/searchad.mspx

WMI:

http://www.microsoft.com/technet/scriptcenter/topics/msh/mshandwmi.mspx

http://www.codeplex.com/PsObject/Wiki/View.aspx?title=WMI&referringTitle=PSH%20Community%20Guide

PowerShell blog:

http://poshoholic.com/

http://blogs.microsoft.co.il/blogs/ScriptFanatic/

ADSI:

http://blogs.technet.com/industry_insiders/pages/windows-server-2008-protection-from-accidental-deletion.aspx

http://msdn.microsoft.com/en-us/library/ms256752(VS.80).aspx

http://bsonposh.com/archives/tag/adsi

http://www.leadfollowmove.com/archives/powershell/managing-group-membership-in-active-directory-with-powershell-part-1
http://www.leadfollowmove.com/archives/powershell/managing-group-membership-in-active-directory-with-powershell-part-2

http://technet.microsoft.com/en-us/magazine/cc162323.aspx

http://powershelllive.com/blogs/lunch/archive/2007/04/04/day-6-adsi-connecting-to-domains-computers-and-binding-to-objects.aspx

PowerShell on Windows Server 2008 Server Core:

http://dmitrysotnikov.wordpress.com/2008/05/15/powershell-on-server-core/

Effective PowerShell:

http://keithhill.spaces.live.com/?_c11_BlogPart_BlogPart=blogview&_c=BlogPart&partqs=cat%3dEffective%2bPowerShell

ACL-ek kezelése:

http://www.microsoft.com/technet/scriptcenter/resources/pstips/may08/pstip0516.msp
x

Performance Monitor:

http://www.leeholmes.com/blog/AccessingPerformanceCountersInPowerShell.aspx

LDAP filter:

http://www.tek-tips.com/faqs.cfm?fid=5667

DateTime típus a .NET keretrendszerben:

http://www.meshplex.org/wiki/C_Sharp/Dates_and_Times

COM Automation:

http://blogs.msdn.com/jmanning/archive/2007/01/25/using-powershell-for-outlook-
automation.aspx
http://www.computerperformance.co.uk/powershell/powershell_com.htm
http://www.microsoft.com/technet/scriptcenter/resources/pstips/nov07/pstip1130.msp
x

5. Tárgymutató

F

G

H

I,Í

K

L

M

Sz

T

Ty

V

W