

ELSŐ FEJEZET

Bevezetés

1.1. Új parancssor: a PowerShell

Még egy parancssor?

„Na, még egy parancssor. Még valami, amivel a Microsoft megnehezíti a rendszergazdák amúgy sem egyszerű életét.” – hallani ezt sok üzemeltetőtől. A könyv célja az, hogy eloszlassa ezt a tévhitet és megmutassa, hogy a PowerShell használatával a rendszerfelügyeleti feladatok jelentősen egyszerűsíthetők.

A Microsoft 2006 novemberének végén, az Exchange Server 2007 bejelentésével egy időben, ingyenesen letölthetővé tette új parancssorát, a PowerShellt. Az Exchange 2007 az első termék, mely a PowerShellre épül, és úgy tűnik, hogy a jövőben egyre több kiszolgáló-alkalmazás követi ezt a példát. Ez azt jelenti, hogy e programok valamennyi szolgáltatását teljességükben csak a PowerShellből érhetjük el. Ha tehát más nem is, ez motiválhat minket, hogy közelebbről megismerjük az új parancssort. Emellett, mint minden más parancssornak, ennek is az a fő rendeltetése, hogy a rendszerfelügyelettel kapcsolatos teendőinket automatizálhassuk.

A rendszerhéj

A rendszerhéjak használata létfontosságú az operációs rendszerekkel való kommunikációban, mert ezek segítségével végezhetünk el olyan alapvető feladatokat, mint a fájlok kezelése, parancsok futtatása vagy

az alkalmazások elindítása. Mindenkinék, aki használt már számítógépet, kapcsolatba kellett kerülnie a rendszerhékkel, akár úgy, hogy parancsokat gépelt be, akár úgy, hogy ikonokra kattintott. A számítógéppel végzett mindennapi munka során elengedhetetlen, hogy egy vagy több rendszerhéjjal „beszélgessünk”.

Kétféle rendszerhég van: a grafikus felületű (ilyen például a Windowsokban található Windows-intéző) és a szöveges, parancssori felületű (*CLI, Command Line Interface*). Ez utóbbi fajta tartozik többek között a Unixok és Linuxok alatt használt **bash**, a windowsos **Parancssor (Command Prompt)**, illetve a **PowerShell**¹ is.

Mindkét típusnak vannak előnyei és hátrányai. A grafikus felületek sokkal könnyebben kezelhetők, *intuitívabban*: egyszerűen csak a megfelelő ikonra kell kattintani, vagy egy parancsgombot megnyomni, és jól eséllyel az történik, amit elvárunk. Hátrányuk viszont, hogy nehezebben programozhatók és automatizálhatók.

A szöveges felületű rendszerhékjakra egyszerűen írhatunk több parancsból álló fájlokat, *szkripteket* (az angol *script* szó forgatókönyvet jelent), melyek automatikusan elvégzik a gyakran vagy sokszor ismétlődő műveleteket, viszont használatukhoz előbb tanulásra és jellemzően mélyebb ismeretekre van szükség.

Bizonyos feladatokhoz az egyik, másokhoz a másik felület a használhatóbb. Mindig célszerű megfelelően megválasztani az eszközt. Ha létre szeretnénk hozni egy felhasználót az Active Directory cím-tárban, egyszerűbb és gyorsabb, ha az **Active Directory - felhasználók és számítógépek (Active Directory Users and Computers)** nevű felügyeleti (MMC) konzolban tesszük ezt meg, mert így nem kell utánanézni annak, hogy ezt parancssorból milyen módon tehetjük meg. De más a helyzet, ha mondjuk, 500 felhasználói fiókot kell nagyon hamar kialakítanunk. Ilyenkor gyorsabb, ha valamiféle szkriptet írunk rá.

¹ *power*: erő, hatalom; az angol nyelvben előtagként szokás a nagy teljesítmény vagy a nagy hatékonyság szinonimájaként is használni; *shell*: rendszerhég

Olyan, mint régen

A PowerShelllel való ismerkedésünket kezdjük rögtön úgy, hogy elindítjuk a **Start** menü **Windows PowerShell 1.0** mappájából, majd begépeljük a már jól ismert parancssori parancsokat:

```
PS C:\Users\MartonSz> cd \  
PS C:\> dir  
  
Directory: Microsoft.PowerShell.Core\FileSystem::C\  
  
Mode                LastWriteTime         Length Name  
----                -  
d-r--           2007.11.06.   12:19             Program Files  
d-r--           2007.11.05.   16:55             Users  
d----           2007.11.05.   17:38             windows  
-a---           2006.09.18.   23:43             24 autoexec.bat  
-ar-s           2007.09.14.   21:43            8192 BOOTSECT.BAK  
-a---           2006.09.18.   23:43             10 config.sys  
  
PS C:\> md PS-konyv  
  
Directory: Microsoft.PowerShell.Core\FileSystem::C\  
  
Mode                LastWriteTime         Length Name  
----                -  
d----           2007.11.14.   12:37             PS-konyv  
  
PS C:\> dir  
  
Directory: Microsoft.PowerShell.Core\FileSystem::C\  
  
Mode                LastWriteTime         Length Name  
----                -  
d-r--           2007.11.06.   12:19             Program Files  
d----           2007.11.14.   12:37             PS-konyv  
d-r--           2007.11.05.   16:55             Users  
d----           2007.11.05.   17:38             windows  
-a---           2006.09.18.   23:43             24 autoexec.bat  
-ar-s           2007.09.14.   21:43            8192 BOOTSECT.BAK  
-a---           2006.09.18.   23:43             10 config.sys  
  
PS C:\>
```

Nagy meglepetés nem ér minket, az történik, amit vártunk: először átlépünk az aktuális meghajtó gyökérfájlrendszerébe, majd kilistázzuk annak tartalmát, és létrehozunk egy új mappát; bár a kimenet formátuma valamelyest különbözik a hagyományostól. Látunk furcsa, számunkra egyelőre értelmetlen kifejezéseket; ezeket később – ahogy egyre többet használjuk a PowerShellt – megtanuljuk.

A PowerShellt fejlesztői igyekeztek a Windows hagyományos parancssorához minél hasonlóbbá tenni, hogy könnyebb legyen megszokni és megtanulni. Próbálkozhatunk más feladatokkal is, minden alapvető funkció (**rd**, **copy**, **del**, **ren** stb.) hasonlóképp működik. Akár más programokat is futtathatunk belőle (**mmc.exe**, **cmd.exe** stb.).

Mivel tud többet a PowerShell más parancssoroknál?

Az eddigi példákból úgy tűnik, hogy a PowerShell ugyanúgy működik, mint a Windows-parancssor. A későbbiekben láthatjuk, hogy ez csak a felszínen van így, valójában semmi köze nincs a „hagyományos” parancssorhoz: teljesen más logikára épül.

Valamennyi manapság használt parancssor tudja kezelni az úgynevezett *csövezést* (*piping*). Ez a következőt jelenti: amit az egyik parancs a képernyőre írna, a képernyő helyett egy másik program bemenetére irányítjuk. A második program az első által kiírt adatokat dolgozza fel bemenetként, majd a feldolgozás eredményét jeleníti meg a képernyőn. Így olyan összetettebb feladatokat is megoldhatunk, amelyeket nem tudnánk, ha az adott eszközöket külön kellene alkalmaznunk. Ezt a műveletet azért nevezik csövezésnek, mert a különböző programokat a *pipe* (`|`) karakterrel kapcsoljuk össze, az angol *pipe* szó pedig csövet jelent.

Bár ezt a funkciót – korlátozásokkal – használhattuk a korábbi Windows-parancssorban is, a Linux-világban sokkal általánosabban

1. fejezet: Bevezetés

alkalmazzák. Ha a **bash** parancshéjat használjuk, és begépeljük az **ls** parancsot, megkapjuk az aktuális könyvtár tartalmát:

```
[martonsz@extra ~/ps]$ ls
dump.txt  FILES  index.html  installer.sh  vid1resp.txt
```

Ha ebből csak azokra a fájlokra vagyunk kíváncsiak, amelyek nevében szerepel a **.txt** kiterjesztés, az **ls** parancs kimenetét bele kell csőveznünk a **grep** parancsba, melynek argumentumként megadjuk a **.txt** karaktersorozatát:²

```
[martonsz@extra ~/ps]$ ls | grep .txt
dump.txt
vid1resp.txt
```

Viszont nem mindegy, hogy mi áll a kifejezés bal oldalán. Ha az **ls** parancsnak megadjuk a **-l** kapcsolót is, teljesen más lesz az eredmény:

```
[martonsz@extra ~/ps]$ ls -l | grep .txt
-rw-r--r-- 1 root  root  1630195 2006-10-19 11:04 dump.txt
-rw-r--r-- 1 root  root    1286 2007-07-30 00:33 vid1resp.txt
```

A fenti példákban az történt, hogy az **ls** parancs megkérdezte az operációs rendszert, hogy az aktuális könyvtárban milyen fájlok találhatóak, majd ezt a listát – a kiegészítő adatokkal együtt – egy karakterláncban visszaadta. Ezt a karakterláncot a **grep** program soronként feldolgozta, majd ezek közül kiírta azokat, amelyekre illeszkedett a megadott minta („**.txt**”); a többit „lenyelte”. Ez nagyon jól működik egészen addig, amíg az így csővezett adatokkal nem próbálunk meg valami egyéb műveletet végrehajtani, például összeszámolni a **.txt** kiterjesztésű fájlok méretét, vagy kiszámolni, hogy két fájl létrehozásának dátuma között mennyi idő telt el. Erre már kisebbfajta prog-

² Mindezt persze megtehettük volna az **ls *.txt** parancssal is: ezt a megoldást a példa kedvéért használtuk.

ramot kell írunk, amely először kikeresi a karakterláncból, hogy mely karakterek jelzik az adott fájl méretét vagy létrehozásának dátumát és idejét. Ezután az adatokat olyan „emészthető” formára hozza, amellyel már lehet matematikai műveleteket végezni.

Mennyivel egyszerűbb lenne a dolgunk, ha karakterlánc helyett adatszerkezetet kapnánk eredményül: közvetlenül, az operációs rendszer által is használt tulajdonságokat, méghozzá a megfelelő formátumban; a fájl méretét például egész számként (*integer*).

A PowerShell egyszerűbb, mint a Visual Basic-szkriptek

Írhatunk Visual Basic-szkriptet is, hogy elkérje a Windowstól a könyvtárlistát, és abból kiválogassa azokat a fájlokat, melyek neve a .txt karaktersorozatra végződik, majd összeadja ezek méretét:

```
On Error Resume Next
Dim ext, fso, size
ext = ".txt"

Set fso = CreateObject("Scripting.FileSystemObject")
size = 0
For each folderIdx In fso.GetFolder(".").files
    If Right(folderIdx.Name,4)=ext Then size = size + _
        fso.GetFile(folderIdx.Name).Size
Next

wscript.Stdout.WriteLine("A " & ext & " fájlok mérete " & _
    size & " bájt.")
```

Ennek az a hátránya, hogy alapos Visual Basic-ismereteket igényel, továbbá nem árt, ha az embernek van fejlesztői tapasztalata is – és ne feledkezzünk meg a temérdek szabadidőről sem. A feladat megoldásához 13 sor kódot kellett írni! Képzeld el, mekkora lenne egy olyan szkript, amely végigjárja az adott kötet összes almappáját, ki-

gyűjti belőlük azokat a fájlokat, melyek két hétnél régebbiek, és áthelyezi őket egy **archiv** nevű mappába – jobb nem is gondolni rá. Látható, hogy Visual Basicben rengeteg feladat elvégezhető, de bonyolultsága miatt nem a rendszergazdák legkedvesebb eszköze.

A PowerShell ötvözi a parancssorok és a Visual Basic-szkriptek előnyeit, sőt, annál sokkal többet is tud. Csövezéssel összeköthetjük az egyes parancsokat, de közben mégsem szöveggel, hanem objektumokkal dolgozunk; a dátum például dátum marad, nyugodtan hozzáadhatunk egy évet vagy kivonhatunk belőle három hónapot: az eredmény egy másik dátum lesz.

Nem túlzás azt állítani, hogy a PowerShell objektumorientált parancssor – ezért tisztában kell lennünk az objektumorientált programozás alapjaival, hogy hatékonyan kihasználhassuk.

A következőkben röviden ismertetjük az objektumorientált programozás alapfogalmait; ez azonban kevés lesz ahhoz, hogy az Olvasó részletesen is elmélyedhessen benne. Ezért az objektumorientált programozásról számos könyvet felsoroltunk e könyv irodalomjegyzékében.

1.2. Bevezetés az objektumorientált programozásba

A számítástechnika hőskorában a programok néhány száz sornyi, viszonylag alacsony szintű utasításból álltak. Ezek könnyen áttekinthetők és egy ember számára is jól kezelhetők voltak.

A magas szintű, strukturált programnyelvek megjelenése hatékonyabb eszközzel látta el a fejlesztőket. Ezekben a megoldandó feladatot át kell alakítani a számítógép által értelmezhető adatokra és az azt feldolgozó algoritmusra. Ennek megvalósítására számos programnyelv készült, és ahogy nőtt a számítógéppel megoldott felada-

tok száma, úgy váltak a programozási nyelvek – és a programok – is egyre bonyolultabbakká.

Amikor az 1980-as évekre a programok mérete és összetettsége megnőtt, egyre inkább kezdtek előjönni a strukturált programtervezés korlátai. Ez idő tájt gyakori volt a határidők csúszása és az eredménytelen fejlesztés. A problémákat igyekeztek valamilyen szabványosítás, új módszertan vagy újrahasznosítható programkód segítségével megoldani.

Az 60-as és a 80-as évek között a strukturált programozási módszerek mellett fokozatosan megjelent egy újfajta elgondolás, az objektumorientált programozás.

Az objektumorientált programozás alapja az objektumszemlélet. Ennek az a lényege, hogy a programozást az ember számára természetes úton közelítjük meg. Számunkra a világ objektumokból áll: asztalok, székek, számítógépek, telefonok, focimeccsek, bankszámlák stb. Szintén emberi tulajdonság, hogy megpróbáljuk ezeket az objektumokat osztályozni, valamiféle sorba rendezni, bizonyos tulajdonságaikat kiemelve, másokat háttérbe szorítva. Például a kutya és a macska emlősök, a mosógép és a tűzhely háztartási gép, az úszás és a tenisz sport, az Opel és a Renault autók, a busz és az autó járművek.

A különböző objektumokat számtalan módon osztályozhatjuk. Ez attól függ, hogy miképpen viszonyulunk az adott objektumhoz, mennyire ismerjük, milyen szolgáltatásait használjuk. Másképp osztályozza az emberi agyat egy pszichiáter, és megint másképp egy agysebész vagy a kannibál törzs egy tagja: mindegyikük számára más a fontos.

Az objektumorientált programozás először is abból áll, hogy az egyes objektumokból hierarchiát építünk. Ezt bizonyos fokig bármely programnyelvben megtehetjük, mint ahogy elméletileg öngyűjtővel is felforralthatunk egy kád vizet – de sokkal egyszerűbb objektumorientáltan programozni egy olyan nyelven, amelyet kifejezetten erre terveztek. Az objektumorientált programozási nyelveknek, pél-

dául a C++-nak vagy a C#-nak, három szolgáltatásuk van, amely fontos az emberek számára is természetes objektumazonosításban és -osztályozásban: az egységbezárás (*encapsulation*), az öröklődés (*inheritance*) és a többalakúság (*polymorphism*).

Osztályok és példányok

Mint említettük, az emberi gondolkodásban nagyon fontos a különböző dolgok osztályozása, kategóriákba sorolása. Enélkül nem tudnánk a dolgokat elnevezni, és valószínűleg a beszéd és a gondolkodás is lehetetlen volna.

Csak Magyarországon több millió autó van: ezek közül egy a saját autónk (ha van). A magyar nyelvben ezt így fejezzük ki: ha azt mondjuk: *az autó*, akkor a sajátunkra gondolunk vagy éppen arra, amellyel megérkeztünk. Ha viszont csak azt mondjuk: *autó*, akkor az összesről beszélünk – pontosabban egyikről sem, hanem arról az elvont kategóriáról, amelybe – közös tulajdonságai alapján – az összes autó besorolható.

A fentieket az objektumorientált programozásban úgy fogalmazzuk meg, hogy az autó mint elvont kategória az osztály (*class*), a saját autónk pedig az osztály egyik példánya (*instance*). Az osztály azt határozza meg hogy az autó típusú objektumok milyen tulajdonságokkal *rendelkezhetnek*, a példány pedig azt, hogy a konkrét autó a lehetségesek közül konkrétan mely tulajdonságokkal *rendelkezik*.

Autóvásárláskor fontos szempont az autó színe vagy az, hogy mennyit fogyaszt. Az objektumszemlélet szerint ez azt jelenti, hogy az autó osztály meghatározza, hogy az autónak van színe és fogyasztása. Amikor már nézegetünk egy konkrét járművet, akkor az autó osztály egy példányával foglalkozunk, amelyről tudjuk, hogy a színe kék és a fogyasztása 5,5 l száz kilométerenként.

Az autó osztályból tehát úgy lesz az autó példánya, hogy az osztály által meghatározott elvont tulajdonságokat (szín; fogyasztás) *kitöltöttük* konkrét értékekkel (kék; 5,5 l). Az objektumorientált programozásban azt a műveletet, amelynek során konkrét értéket adunk az osztályban megadott elvont tulajdonságoknak, és így létrehozunk az objektum példányát, *példányosításnak* (*instantiation*) nevezzük. Mondhatjuk azt is, hogy az osztály az objektum *típusa*.

Egységbezárás

A szoftverfejlesztés egyik legnagyobb problémája, hogy a fejlesztett rendszerek egyre nagyobbak és bonyolultabbak. Az egységbezárás lehetőséget nyújt arra, hogy a programot kisebb, önálló összetevőkre bontsuk. Például egy banki rendszer fejlesztésekor valószínűleg szükség lesz egy, a számlákat kifejező objektumra. Miután megterveztük a **Számla** osztályt, a továbbiakban nem kell foglalkoznunk ennek a megvalósításával. Ugyanúgy alkalmazhatjuk ezt is, mint az egész számok tárolására használt **integer** típust. Az osztálynak köszönhetően a programozónak elég a **Számla** típusú objektumok azon tulajdonságaival foglalkoznia, amelyek az adott esetben fontosak, és nem kell törődnie az objektum megvalósításával. Például fontos lehet a számlatulajdonos neve és a számla egyenlege, de lényegtelen, hogy a név 80 elemű karaktertömbben vagy karakterláncban van-e tárolva.

Öröklődés

Az embereknek szokásuk, hogy az objektumokat hierarchiába rendezik. Ez programozási szempontból is hasznos, és minden objektumorientált nyelv támogatja az öröklődés révén. Az öröklődésnek két nagyon fontos előnye van: az objektumokat hierarchiába szer-

1. fejezet: Bevezetés

vezhetjük, illetve a hierarchiában lejjebb található objektumok örökölhetnek tulajdonságokat a feljebb lévő objektumoktól (a szülő-objektumoktól). Vegyünk két osztályt, a **FolyóSzám**la és a **HitelSzám**la osztályt. Mindkettő a **Szám**la osztályból ered. A **Szám**la osztály általánosabb, mint a **FolyóSzám**la vagy a **HitelSzám**la. Az öröklődésnek köszönhetően megtalálhatók bennük a **Szám**la osztály jellemzői: a számlatulajdonos neve és a számla egyenlege.

Az új osztályok örökölhetik a meglévők tulajdonságait, de akár módosíthatják is őket. A hitelszámláról például többet is költhetünk a hitelkeretünknel (a számla egyenlegénél), de ilyenkor hitelkamatot számítanak fel, míg a folyószámla esetén egyikre sincs lehetőség.

Többalakúság

Az objektumorientált programozás harmadik jellemzője a többalakúság. Ez lényegében azt jelenti, hogy több különböző objektumnak lehet olyan közös tulajdonsága, amelyet különbözőképp valósítanak meg. Vegyünk különböző járműveket: mindet valahol be kell indítani, vagyis programozási szemszögből, mindnek van „indítás” funkciója. Hogy ezt valójában milyen módon kell megvalósítani, az a járműtől függ. Egy autónál egyszerűen el kell fordítani a gyújtáskapcsolót, míg egy gőzmozdony esetében ez sokkal bonyolultabb.

Osztályok, objektumok és metódusok

Az objektumok az adott dolgot leíró adatokból (tulajdonságokból), és az azon végezhető műveletekből (metódusokból) állnak. Amikor viszont objektumorientált programot írunk, osztályokat kell meghatároznunk, nem objektumokat.

Az osztály a felhasználó vagy a rendszer által meghatározott típus, amely tulajdonságokból és az azokon végrehajtható metódu-

sokból áll. Felfoghatjuk egyfajta sablonként, amelyet arra használhatunk, hogy objektumpéldányokat hozzunk létre belőle. Amikor programozunk, az objektumokat előbb példányosítani kell, és csak azután használhatjuk őket. A használat ez esetben azt jelenti, hogy kitöltjük a tulajdonságait – értéket adunk a tulajdonságokat képviselő úgynevezett `tagváltóknak` –, és futtatjuk az osztályban meghatározott metódusokat.

Nem tudjuk közvetlenül létrehozni az **Állat** objektumpéldányt. Előbb meg kell határoznunk az **Állat** osztályt, az összes lehetséges tulajdonságával és metódusával együtt. Az **Állat** osztály tehát nem egy konkrét állatot jelképez, hanem bármelyik állatot jelentheti. Ha használni szeretnénk az **Állat** osztályt, példányosítanunk kell azt, hogy értékeket adhassunk neki és műveleteket végezhesünk rajta – vagyis a sablon alapján létre kell hoznunk egy konkrét állatot.

1.3. Ismerkedés a .NET keretrendszerrel

Mi a .NET?

A .NET a Microsoft előre gyártott objektumorientált környezete – tulajdonképpen objektumgyűjteménye –, amely nagy segítséget nyújt a programozóknak, mert kész megoldásokat ad az alkalmazások fejlesztése során felmerülő gyakori (és kevésbé gyakori) problémákra.

A programozási feladatok egyre összetettebbek, a programok egyre bonyolultabbak, és az operációs rendszernek is egyre több szolgáltatása van. A programozóknak, ha a hagyományos eszközöket használják – a C vagy a C++ programnyelvet és az operációs rendszer alapfunkcióit – rengeteget kell dolgozniuk az úgynevezett

infrastruktúrakódon. Az infrastruktúrakód a programok azon része, amely a hálózati kommunikáció, az adat- és adatbázis-kezelés és a grafikus felhasználói felület alapelemeit valósítja meg, de nem része, csupán alapvető, általános építőeleme a konkrét alkalmazáslogikának. Ha adatokat akarunk átküldeni a hálózaton, és mondjuk csak a C++ nyelvet és a Win32-rendszert használjuk, a konkrét adatküldéshez akár több száz programsort is le kell írunk.

A .NET ezt az infrastruktúrakódot kínálja készen, jelentősen lerövidítve a programok kifejlesztéséhez szükséges időt és csökkentve a programfejlesztés költségét. A .NET-rendszerben egy XML-fájl beolvasásához, elemzéséhez és objektumként való eléréséhez egyetlen programsort kell írni. Ha ezt az objektumot át kell küldenünk a hálózaton, talán már három programsort is le kell írunk.

A keretrendszer a Windows Server 2003 óta szerves része a Windowsoknak, illetve külön összetevőként korábbi operációs rendszerekre letölthető, sőt bizonyos korábbi változatai akár Windows 98 alatt is futnak.

A .NET legfontosabb szolgáltatásai

Programfuttató környezet: A .NET keretrendszer programfuttató környezet, amelynek köszönhetően sokkal könnyebb stabil, megbízható programot írni. Minden általunk írt program ebben a környezetben fut. A környezet automatikus memóriakezelést biztosít, és egyszerűsíti a rendszerszolgáltatásokhoz való hozzáférést.

HTML-fejlesztési modell (ASP.NET): A statikus HTML oldalak unalmasak és kevésbé hatékonyak: mindig minden felhasználónak ugyanazt a tartalmat adják vissza. Ahhoz, hogy valami változás is történjen az oldalon, vagy a webhely tartalmát adatbázisból lehessen szolgáltatni, kiszolgálóoldali programot kell írunk. Az ASP.NET környezet az Internet Information Services (IIS) webkiszolgálón fut;

segítségével könnyebben készíthetünk eseményvezérelt, dinamikus HTML-lapokat, még hozzá bármely .NET-alapú programozási nyelven (felügyelt C++, C#, Visual Basic.NET).

Windows-alkalmazások fejlesztése: Az asztali alkalmazásoknak könnyen kezelhető felhasználói felülettel kell rendelkezniük. A szépen megtervezett felület javítja a felhasználói élményt is: sokkal szívesebben használjuk Microsoft Outlookot, mint a Hotmail webes felületét. A Microsoft .NET segítségével egyszerűbbé válik a Windows-alkalmazások programozása is.

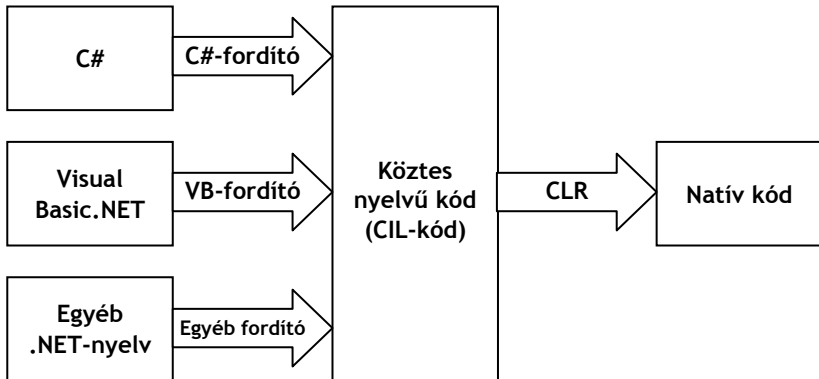
Többszálú programozás: 1993-ban, a 32-bites Windows NT megjelenésével vált valóban többfeladatossá a Windows. Ennek ellenére egész mostanáig meglehetősen nehéz volt többszálú programot írni, mivel ezt az operációs rendszer csak korlátozottan támogatja. A .NET-keretrendszer a korábbiaknál magasabb szintű támogatást nyújt az operációs rendszer többszálú programfuttatási képességeinek kihasználására.

A .NET-alapú programozás

Amikor egy .NET-alapú nyelven programot írunk, majd azt lefordítjuk, a keletkező bináris (végrehajtható) fájl nem értelmezhető közvetlenül az operációs rendszer számára. A .NET-alkalmazások úgynevezett *köztes nyelvre* fordulnak le: ez a Common Intermediate Language (CIL) nevet kapta. Ez már független az alkalmazott programozási nyelvtől, tehát akár a Visual Basic.NET-ben, akár a C#-ban, akár bármelyik másik .NET-alapú nyelven írjuk a programot, a kapott kód ugyanaz lesz. Az ilyen programkódot nevezzük *felügyelt kódnak* (*managed code*), mivel csak a .NET-keretrendszer programfuttató környezetének felügyelete alatt futtathatók. A programfuttató környezet neve *Common Language Runtime* (CLR). Ez a környezet nyújtja az olyan magasabb szintű szolgáltatásokat, mint

1. fejezet: Bevezetés

amilyen az automatikus memóriakezelés, benne a szemétyűjtés (*garbage collection*), a többszálú programfuttatás támogatása és a kivételkezelés.



1.1. ábra: A .NET-programok futtatásának sémája

Amikor elindítunk egy .NET-programot, a köztes nyelvre (CIL) lefordított programot a CLR futtatókörnyezet igény szerinti fordítója (*just-in-time - JIT - compiler*) natív, a számítógép processzora által értelmezhető kódra fordítja. Ez közvetlenül a program futtatása előtt történik, ezért nagyobb .NET-alkalmazások indítása kissé tovább tarthat, mint hasonló funkciójú natív kódban írt társaiké.

A .NET-objektumok

Most áttekintjük a .NET azon objektumait, adattípusait és a rájuk jellemző metódusokat (műveleteket), amelyekre feltétlenül szükség van a PowerShell használatához.

Mint minden programozási nyelvben, a .NET-alapú nyelvekben is használhatunk változókat. A .NET-alapú programozási nyelvek erősen típusosak, tehát az, hogy melyik változónak milyen értéket

adhatunk, a változó deklarálásakor meghatározott típusától függ. Ez erősen érezhető, amikor valamilyen „rendes” programozási nyelven (pl. C#) dolgozunk, a PowerShellben ezzel kevésbé kell foglalkoznunk. Ha például egy változó típusa egész szám (*integer*), az csak egész számokat tárolhat, egy karakterlánc (*string*) csak karaktereket, a dátum típusú változó pedig csak dátumot.

Ezért minden objektumnak – az egész típusnak, a lebegőpontos számoknak, a dátumoknak, sőt még a karakterláncoknak is – van **ToString** metódusa, mely azt karakterláncá alakítja. Itt jegyezzük meg, hogy a .NET-ben minden, még a legegyszerűbb adattípus is objektumként van kezelve, vagyis például egy egész számokat tároló változónak is vannak tulajdonságai és metódusai.

A **ToString** metódusnak leginkább összehasonlításakor vesszük hasznát: például ha a **ma** nevű változó típusa **DateTime** és értéke az aktuális dátum és idő, ezt összehasonlíthatjuk a „**2000.12.31. 12:31:00**” karakterláncsal. A **ma==„2000.12.31. 12:31:00”** kifejezés nem ad eredményt, mivel két teljesen különböző típusú objektumot próbálunk összevetni. Erre a helyzetre leginkább a régi favicc illik: „Mi a különbség a krokodil között? Hosszabb, mint zöld.” Ha eredményhez akarunk jutni, meg kell hívni a **ma** változó **ToString** metódusát, amely a dátumból karakterláncot hoz létre az operációs rendszer aktuális területi beállításainak megfelelően:

```
ma.ToString()==„2000.12.31. 12:31:00”
```

Azt is megnézhetjük, hogy a **ma** változó **Year**, **Month**, **Day** stb. tulajdonsága megegyezik-e a kívánt értékkel (pl. **ma.Year==2008**).³

Több dátumból létrehozhatunk úgynevezett **gyűjtemény** (*collection*) is. A gyűjtemény olyan objektum, amely több más, azonos

³ Az utóbbival jobban járunk, mert a dátumok esetén a **ToString()** eredménye teljesen más lehet különböző nyelvi – területi – beállítások esetén, és sok esetben nem tudjuk megjósolni, hogy a programunkat futtató Windows milyen nyelvre van beállítva.

1. fejezet: Bevezetés

típusú objektumot tartalmaz. Ha a gyűjtemény neve **datumok**, akkor annak egyes elemeire (melyek külön-külön önálló **DateTime** típusú adatok) a **datumok[0]**, **datumok[1]**, **datumok[2]** stb. formában hivatkozhatunk. Vegyük észre, hogy a gyűjtemény első elemét a 0 indexszel kell jelölni. Ilyenkor az első dátum hónap részét a **datumok[0].Month**, a huszonharmadik dátum év részét pedig a **datumok[22].Year** tulajdonságra való hivatkozással érjük el. Természetesen gyűjteményt nem csak dátumokból, hanem bármilyen típusból létrehozhatunk, így akár egész számokból, fájlokból vagy e-mail címekből is.

Azt, hogy az adott gyűjteményben hány objektum található, a **Count** tulajdonság adja meg, azaz a fenti példánál maradván: **datumok.Count**.