ELSŐ FEJEZET

A .NET filozófiája

Ma már a programozók számára elengedhetetlen, hogy a néhány évente vagy akár még gyakrabban továbbképzéseken vegyenek részt, hiszen mindig naprakésznek kell lenniük a legújabb technológiákból. A szoftverfejlesztés gyöngyszemeinek kikiáltott nyelveket (C++, Visual Basic 6.0, Java), keretrendszereket (OWL, MFC, ATL, STL), architektúrákat (COM, COBRA, EJB) és API-kat (mint a .NET Windows-űrlapjai és a GDI+ könyvtárai) végül mindig háttérbe szorítja valami jobb, de legalábbis valami új. Bár a belső adatbázisunk frissítésekor nyilván csalódottságot érzünk, ám mindez elkerülhetetlen. Éppen ezért könyvünknek az a legfontosabb célja, hogy betekintsünk a Microsoft szoftverfejlesztéssel kapcsolatos aktuális kínálatának részleteibe: a .NET platformba és a C# programozási nyelvbe.

Ebben a fejezetben elsődleges feladatunk az, hogy tisztázzuk a könyv további részeinek megértéséhez szükséges alapfogalmakat. Köztük számos .NET-hez kapcsolódó téma: például a szerelvények, a köztes nyelvi kód (CIL) és a JIT-fordítás tárgyalását is nyomon követhetjük. A C# programozási nyelv legfontosabb sajátosságainak ismertetése mellett a .NET-keretrendszer különböző megjelenési formái – mint a nyelvek közös futtatórendszere (CLR), a közös típusrendszer (CTS) és a közös nyelvi specifikáció (CLS) – között létrejövő kapcsolatra is fény derül.

Ezeken kívül az olykor BCL-nek, esetleg FCL-nek (keretrendszer-osztálykönyvtár) rövidített .NET-alaposztálykönyvtárak funkcionalitását is megvizsgáljuk. Végezetül áttekintjük a .NET platform nyelvagnosztikus és platformfüggetlen természetét is. (Tudniillik a .NET nem korlátozódik a Windows operációs rendszerre.) Ezeket a témákat a továbbiakban részletesen is kifejtjük.

Az előzmények áttekintése

A .NET univerzum sajátosságainak számbavétele előtt hasznos lehet néhány olyan tényezőt összegezni, amely a Microsoft jelenlegi platformjának megalkotását előidézte. A tisztánlátáshoz kezdjük a fejezetet egy rövid, „fájdalommentes" történelemórával, hogy felelevenítsük az előzményeket, és megértsük azok korlátait. (Elvégre az első lépés a megoldáshoz vezető úton, ha beismerjük, hogy problémával állunk szemben.) Miután tettünk egy gyors körutazást a múltban, figyelmünket a C# és a .NET-platform számtalan előnyének a megismerésére fordíthatjuk.

Egy C/Win32 API programozó élete

Azt szokták mondani, hogy a Windows operációs rendszerek családjába tartozó szoftverek fejlesztése a C programozási nyelv és a Windows alkalmazásprogramozói felületének (API) együttes használatából áll. Igaz, hogy számos alkalmazást sikeresen létrehoztak ezzel a hagyományos módszerrel, néhányan mégis vitatják, hogy az alkalmazások nyers API-val történő felépítése összetett feladat lenne.

Az első nyilvánvaló probléma az, hogy a C nagyon tömör nyelv. A C# fejlesztői kénytelenek kézi memóriakezeléssel, csúnya pointeraritmetikával és ugyancsak csúnya szintaktikai konstrukciókkal megküzdeni. Ráadásul, mivel a C strukturált nyelv, hiányzik belőle az objektumorientált megoldások előnye (valaki el tudja mondani, mi az a *spagettikód*?). Amikor a Win32 API segítségével definiált globális függvények és adattípusok ezreit hozzáadjuk egy eleve túl részletes nyelvhez, nem csoda, hogy olyan sok hibás alkalmazást kapunk.

Egy C++/MFC-programozó élete

A csupasz C/API fejlesztésének egyik nagy horderejű újítása a C++ programozási nyelv használata volt. A C++ sok szempontból a C-re helyezett objektumorientált *rétegnek* tekinthető. Így annak ellenére, hogy a C++ programozóinak hasznára vannak az objektumorientált programozás közismert alappillérei (egységbezárás, származtatás és polimorfizmus), továbbra is ki vannak szolgáltatva a C nyelv nehézségeinek (pl. kézi memóriakezelés, csúnya pointeraritmetika és csúnya szintaktikai konstrukciók).

Összetettsége ellenére, manapság is sok C++-keretrendszer létezik. Például a Microsoft Foundation Classes (MFC) olyan C++-osztály-készletet biztosítanak a fejlesztők számára, amely megkönnyíti a Win32-alkalmazások felépítését. Az MFC legfőbb szerepe, hogy számos osztály, makró és kódgeneráló eszköz (varázsló) mögött a natív Win32 API egy részét beburkolja. Az MFC-keretrendszer hasznos támogatását figyelmen kívül hagyjuk (ugyanúgy, mint sok egyéb C++-alapú eszközrendszert), be kell látnunk, hogy a C++-programozás nehézkes, és több hibalehetőséget rejt magában, tekintve, hogy a C-ből ered.

Egy Visual Basic 6.0 programozó élete

Az egyszerűségre törekvés jegyében sok programozó a C(++)-alapú keretrendszerekről olyan barátságosabb és kifinomultabb nyelvekre tért át, mint például a Visual Basic 6.0 (VB6). A VB6 annak köszönheti népszerűségét, hogy egyszerűen lehet vele összetett felhasználói felületeket, kódkönyvtárakat (pl. COM-kiszolgálók) és adatelérési logikát készíteni. A VB6 sokkal jobban elrejti szem elől a csupasz Win32 API bonyolultságát, mint az MFC, számos beépített forráskód-varázsló, valódi adattípusok, osztályok és VB-specifikus függvények segítségével.

A VB6 legnagyobb hibája (amelyet a .NET-platform megjelenése orvosolt), hogy nem teljesen objektumorientált nyelv, hanem inkább „objektumtudatos". A VB6 például nem engedi a programozónak, hogy „ez egy…" („is-a") kapcsolatokat hozzanak létre a típusok között (azaz nem engedi a klasszikus származtatást), és nincs belső támogatása paraméterezett osztályok készítéséhez. Ezenkívül a VB6 nem képes többszálú alkalmazások létrehozására sem, kivéve, ha alacsony szintű Win32 API hívásokat szeretnénk (ám ez legjobb esetben összetett, legrosszabb esetben pedig veszélyes).

Egy Java/J2EE-programozó élete

A Java megérkezett. A Java objektumorientált programozási nyelv, amelynek szintaktikai gyökerei a C++-ból erednek. Amint azzal sokan tisztában vannak, a Java előnyei messze túlszárnyalják platformfüggetlenségi támogatását. A Java (mint nyelv) a C++ számos kellemetlen szintaktikai jellemvonását kiküszöbölte. A Java (mint platform) a programozók számára számtalan előre megadott „csomagot" biztosít, amelyek különféle típusdefiníciókat tartal-

maznak. Ezekkel a típusokkal a Java-programozók képesek 100%-ban tiszta Java-alkalmazásokat létrehozni, amelyek teljes adatbázis-összekapcsolhatósággal, üzenetközvetítési támogatással, webes ügyféloldallal és gazdag asztali felhasználói felülettel rendelkeznek.

Habár a Java nagyon elegáns nyelv, potenciális problémája, hogy használatakor jellemzően elölről hátrafelé kell haladni a fejlesztés folyamán. A Java tulajdonképpen kismértékben nyújt nyelvi integritást, minthogy ez ellentmond elsődleges feladatának – a Java egy egyszerű, bármilyen célra felhasználható programozási nyelv. A valóságban azonban forráskódok milliónyi sora létezik, s ezeket előnyös lenne Java-kódból felhasználni. Sajnos a probléma megoldását a Java nem könnyíti meg. Amíg ugyanis korlátozott képességet mutat a nem Java-s API-k eléréséhez, addig csak kismértékben támogatja a valódi nyelvközi integrációt.

Egy COM-fejlesztő élete

A Component Object Model (COM – komponensobjektum-modell) volt a Microsoft előző alkalmazásfejlesztési keretrendszere. A COM architektúrája lényegében azt fejezi ki, hogy „[h]a az osztályainkat a COM szabályai szerint építjük fel, *újrafelhasználható bináris kódblokkot* kapunk".

Egy bináris COM-kiszolgálóban az a nagyszerű, hogy nyelvfüggetlen módon hozzá lehet férni. Így a C++-programozók olyan COM-osztályokat hozhatnak létre, amelyeket a VB6 felhasználhat. A Delphi-programozók használhatnak olyan COM-osztályokat, amelyeket C segítségével hoztak létre, és így tovább. A COM nyelvfüggetlensége azonban némileg korlátozott. Egy meglévő COM-osztályból például nem lehet újabbat származtatni (mivel a COM nem támogatja a klasszikus származtatást). Ehelyett, a sokkal nehézkesebb „van neki egy…" („has-a") kapcsolat segítségével kell újra felhasználni a COM-osztály típusait.

A COM másik előnye az, hogy transzparens tárhellyel rendelkezik. Az olyan szerkezetek használatával, mint az alkalmazásazonosító (`AppIDs`), a csonk, a proxyk és a COM futási környezet, a programozóknak nem kell socketekkel, távoli eljáráshívásokkal és egyéb alacsony szintű részletekkel dolgozniuk. Nézzük meg például a következő VB6–COM-ügyfélkódot:

```
' A MyCOMClass típus bármely
' COM-alapú nyelven megírható, és a hálózaton bárhol
' elhelyezhető (helyi számítógépen is).
Dim obj as MyCOMClass
Set obj = New MyCOMClass     ' AppID használatának a helye.
obj.DoSomeWork
```

Habár a COM-ot nagyon sikeres objektummodellnek is tarthatjuk, valójában elképesztően bonyolult (legalábbis addig, amíg nem fordítunk több hónapot az átkötések vizsgálatára – elsősorban C++-programozóként). A COM-binárisok fejlesztésének egyszerűsítésére számos COM-alapú keretrendszer létezik. Ilyen például az ATL (Active Template Library), amely olyan C++-osztályok, -sablonok és -makrók készletével szolgál, amelyek megkönnyítik a COM-típusok létrehozását.

Sok egyéb nyelv ugyancsak elrejti szem elől a COM-infrastruktúra nagy részét. A nyelvi támogatás azonban önmagában nem elég a COM komplexitásának elrejtéséhez. Még ha egy olyan viszonylag egyszerű COM-alapú nyelvet választunk, mint a VB6, akkor is meg kell küzdenünk a sérülékeny regisztrációs bejegyzésekkel és számos telepítéssel kapcsolatos problémával (gyűjtőnéven, némileg komikusan, *DLL-pokol*).

Egy Windows DNA programozó élete

Tovább bonyolítja a helyzetet az internet. Az elmúlt néhány évben a Microsoft jó néhány további internetalapú szolgáltatást épített be az operációsrendszer-családjába és termékeibe. Sajnos a COM-alapú Windows DNA segítségével történő webes alkalmazások létrehozása ugyancsak meglehetősen bonyolult.

Ez a bonyolultság részben annak a ténynek köszönhető, hogy a Windows DNA számos technológia és nyelv használatát igényli (ASP, HTML, XML, JScript, VBScript és COM[+], valamint olyan adatelérési API-t, mint az ADO). Problémát jelent, hogy szintaktikai szempontból a technológiák nagy része nem kapcsolódik egymáshoz. A JScript szintaxisa például sokban hasonlít a C-re, míg a VBScript a VB6 részhalmaza. A COM+-futtatókörnyezetre tervezett COM-kiszolgálók teljesen eltérő megjelenésűek, mint azok az ASP-oldalak, amelyek hívják őket. Mindez a technológiák teljesen zavaros keverékét eredményezi.

Ezenkívül, és ez talán sokkal fontosabb, minden nyelvnek és/vagy technológiának megvan a saját típusrendszere (és ez lehet, hogy semmiben sem hasonlít egy másikéra). Túl azon, hogy minden egyes API rendelkezik saját előre gyártott kódgyűjteménnyel, az alapvető adattípusokat nem lehet mindig ugyanúgy kezelni. A `CComBSTR` az ATL-ben nem teljesen ugyanaz, mint egy `String` a VB6-ban; egyiknek sincs semmi köze a C `char*` adattípusához.

A .NET-megoldás

Az előző rövid történelemóra tanulsága az volt, hogy a Windows-programozók élete bizony nem egyszerű. A .NET-keretrendszer azonban radikálisan egyszerűsíti és megkönnyíti a programozó életét. A .NET a „Változtassunk meg mindent!" megoldást kínálja („elnézést kérek, de ne az üzenet közvetítőjét okoljuk az üzenetért"). Ahogy a könyv további részeiben látni fogjuk, a .NET-keretrendszer a rendszerfelépítés teljesen új modellje a Windows operációs rendszercsaládban éppúgy, mint számos nem Microsoft-alapú operációs rendszerben (mint például a Mac OS X és a különféle Unix-/Linux-termékek). Előzetesen íme a .NET néhány alapvető jellemzőjének összefoglalása:

- *Együttműködési képesség a meglévő forráskóddal.* Ez (természetesen) nagyon hasznos. A meglévő COM-binárisok keverednek (azaz együttműködnek) az újabb .NET bináris fájlokkal, és viszont. A PInvoke lehetővé teszi a C-alapú könyvtárak (beleértve az operációs rendszer API) hívását a .NET-kódból.
- *Teljes nyelvi integráció.* A .NET támogatja a nyelvközi származtatást, a nyelvközi kivételkezelést és a nyelvközi hibakeresést.
- *Közös futtatórendszer az összes .NET-alapú nyelv számára.* Ez a rendszer definiál egy olyan típuskészlet, amelyet minden .NET-alapú nyelv megért.
- *Átfogó alaposztálykönyvtár.* Ez a könyvtár teszi lehetővé a natív API-hívások komplexitásának elkerülését, és olyan konzisztens objektummodellt kínál, amelyet az összes .NET-alapú nyelv használ.

- *Nincs több COM-átkötés.* Az `IClassFactorynak`, az `IUnknownnak`, az `IDispatchnek`, az IDL-kód és a nem túl szimpatikus `variant`-kompatibilis adattípusoknak (`BSTR`, `SAFEARRAY` stb.) nincs helyük egyetlen .NET–binárisban sem.
- *Egy valóban leegyszerűsített telepítési modell.* .NET alatt nem szükséges bejegyezni a bináris egységeket a rendszerleíró adatbázisba. Ráadásul a .NET lehetővé teszi, hogy egyetlen gépen ugyanazon `.dll` több verziója legyenek jelen.

Ahogy a fenti felsorolásból látszik, a .NET-platformnak semmi köze a COM-hoz (azonfelül, hogy mindkét keretrendszer a Microsofttól származik). A .NET- és a COM-típusok valójában csak az együttműködési réteg segítségével tudnak kommunikálni.

Megjegyzés A .NET együttműködési réteg leírása a 2 kötet A függelékében található.

Bevezetés a .NET platform építőelemeibe (CLR, CTS és CLS)

A .NET néhány előnyét már megismerve nézzük át azt a három alapvető, egymással összefüggő egységet, amelyek mindezt lehetővé teszik: a CLR-t, a CTS-t és a CLS-t. Egy programozó szemszögéből a .NET felfogható futtatókörnyezetként és átfogó alaposztálykönyvtárként. A futtató réteg más néven *közös nyelvi futtatórendszer*, vagyis a *CLR*. A CLR elsődleges feladata, hogy megkeresse, betöltse és kezelje helyettünk a .NET-típusokat. Ezenkívül olyan alacsony szintű részletekre is figyel, mint a memóriakezelés, alkalmazástartományok létrehozása, végrehajtási szálak és objektumkontextus-határok, valamint különböző biztonsági ellenőrzések végrehajtása.

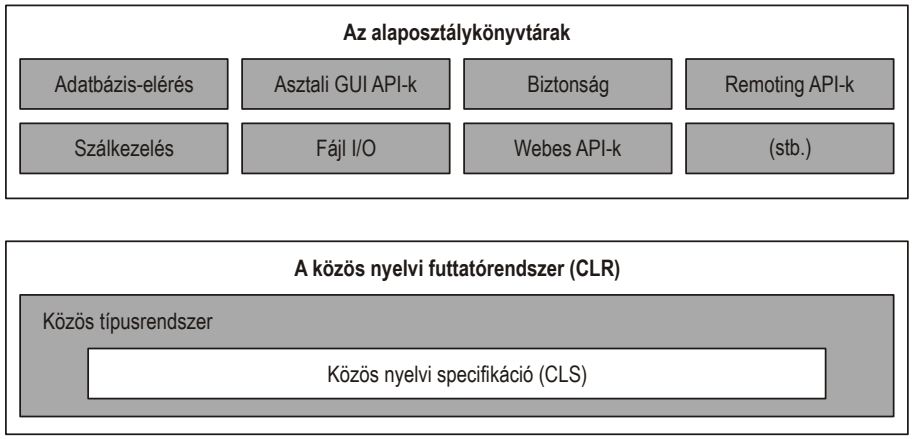
A .NET-platform másik építőeleme a *közös típusrendszer*, azaz a *CTS*. A CTS-specifikáció teljes körűen leírja az összes lehetséges adattípust és programozási szerkezetet, amelyet a futtatókörnyezet támogat, meghatározza, hogy ezek az egységek hogyan kommunikálhatnak egymással, és részletezi azok .NET-metaadat-formátumban való megjelenésének módját (a metaadatokkal kapcsolatos további információkat lásd a fejezet későbbi részében; a teljes körű leírást pedig lásd a 16. fejezetben).

Egy adott .NET-alapú nyelv nem szükségszerűen támogat minden egyes CTS-ben definiált funkciót. A *közös nyelvi specifikáció* (CLS) egy olyan kapcsolódó specifikáció, amely definiálja azoknak a közös típusoknak és programozási szerkezeteknek a részhalmazát, amelyeket minden .NET programozási nyelv elfogad. Ezért, ha olyan .NET-típusokat építünk fel, amelyek csak CLS-kompatibilis funkciókkal rendelkeznek, nyugodtak lehetünk afelől, hogy az összes .NET-alapú nyelv használni tudja azokat. Ellenben, ha olyan adattípust vagy programozási szerkezetet használunk, amely a CLS határain kívül esik, nem biztos, hogy az összes .NET programozási nyelv képes lesz kommunikálni a .NET-kódkönyvtárunkkal.

Az alaposztálykönyvtárak szerepe

A CLR- és CTS/CLS-specifikációkon kívül, a .NET-platform az összes .NET programozási nyelv számára biztosít egy alaposztálykönyvtárat. Ez az alaposztálykönyvtár nemcsak beágyaz különböző primitíveket, mint a végrehajtási szálak, fájl input/output (I/O), grafikus renderelés és különféle külső hardvereszközökkel történő kommunikáció, de számos, leginkább az életszerű alkalmazások igényeihez mért szolgáltatást is támogat.

Az alaposztálykönyvtárak például olyan típusokat definiálnak, amelyek megkönnyítik az adatbázis-hozzáférést, az XML-dokumentumok manipulációját, a programozott adatvédelmet és a webes (csakúgy, mint a hagyományos asztali és konzolos) front endek létrehozását. A CLR, CTS, CLS és az alaposztálykönyvtár közötti kapcsolatot, az 1.1. ábra szemlélteti.



1.1. ábra: *A CLR, CTS, CLS és az alaposztálykönyvtár kapcsolata*

A C# előnyei

Mivel a .NET radikálisan eltér a korábbi technológiáktól, a Microsoft kifejezetten ehhez az új platformhoz hozott létre egy új programozási nyelvet, a C#-t (kiejtve: [si: sa:p]). A C# alapszintaxisa *nagyon* hasonlít a Java szintaxisára. A C# azonban nem a Java egyik módosulata. Mind a C#, mind a Java a C programozási nyelv családjába (C, Objective C, C++ stb.) tartozik, ezért hasonló szintaxissal rendelkeznek. Ahogy a Java sok tekintetben a C++ letisztult verziója, a C#-ot ugyanígy a Java letisztult verziójának tekinthetjük.

Ettől függetlenül az igazság az, hogy a C# szintaktikai felépítése nagyrészt a Visual Basic 6.0 és a C++ alapján készült. Ahogy például a VB6, a C# is támogatja a tulajdonságok létrehozását (ellentétben a tradicionális `getter` és `setter` metódusokkal) és a metódusok deklarálásának képességét változó számú argumentummal (argumentumtömbök révén). A C++-hoz hasonlóan, a C# lehetővé teszi az operátorok túlterhelését, valamint struktúrák, felsorolt típusok és visszahívható függvények létrehozását (metódusreferenciák révén).

Megjegyzés Ahogy a 13. fejezetben látni fogjuk, a C# 2008 számos olyan konstrukciót átvett, amelyek eredetileg különböző funkcionális nyelvekből (pl. LISP vagy Haskell) származnak. Ráadásul, a LINQ megjelenésével (további információkért lásd a 14. és a II. kötetben található 24. fejezetet) a C# több olyan programozási konstrukciót támogat, amelyek meglehetősen egyedivé teszik a programnyelvek palettáján. Mindamellett probléma a C#-pal, hogy valóban hatással voltak rá a C-alapú nyelvek.

Mivel a C# több nyelv keresztezéséből született, szintaktikailag legalább annyira, ha nem még inkább letisztultnak tekinthető, mint a Java, olyan egyszerű, mint a VB6, és annyira hatékony és rugalmas, mint a C++ (a vele járó apró csúnyaságok nélkül). Íme, egy lista a C# alapvető tulajdonságairól, amelyek a nyelv valamennyi verzióját jellemzik:

- Nincs szükség mutatókra. A C#-programoknak jellemzően nincs szükségük közvetlen mutatómanipulálásra (de, ha mindenképpen szükséges, le lehet erre a szintre menni).
- Automatikus memóriakezelés a szemétgyűjtésen keresztül. Így a C# nem támogatja a `delete` kulcsszót.
- Formális szintaktikai konstrukcióként megjelenik az osztály, az interfész, a struktúra, a felsorolás és a metódusreferenciák.

- A C++-hoz hasonló operátor-túlterhelés az egyedi típusok esetében, csak egyszerűbben (pl. a meggyőződés arról, hogy a „return `*this` to allow chaining” nem a mi problémánk).
- Az attribútumalapú programozás támogatása. Az ilyen típusú fejlesztés lehetővé teszi, hogy a típusokat és a tagjaikat magyarázatokkal lássuk el viselkedésük további leírásához.

A .NET 2.0 verziójával (kb. 2005-től) a C# programozási nyelv számos újdonsággal frissült, amelyek közül az alábbiak a legfontosabbak:

- Generikus típusok és tagok felépítésének a képessége. A generikus típusok használatával rendkívül hatékony és típusbiztos forráskódokat építhetünk fel, amelyek a generikus elemmel való kommunikáció során meghatározott helyőrzőket definiálnak.
- Névtelen metódusok támogatása, amelyek lehetővé teszik, hogy bárhol, ahol metódusreferenciára van szükség, `inline` függvényeket alkalmazzunk.
- A metódusreferencia/esemény modell számtalan egyszerűsítése, köztük kovariancia, kontravariancia és metóduscsoport-átalakítás.
- Egyetlen típus több kódfájlon keresztüli definiálásának képessége (vagy ha szükséges, memórián belüli ábrázolása) a `partial` kulcsszó segítségével.

A .NET 3.5 egyértelműen még több funkcionalitással látta el a C# programozási nyelvet (pontosabban a C# 2008 verziót), köztük az alábbiakkal:

- A különféle adatformátumokkal való kommunikációhoz szükséges erősen típusos lekérdezések (LINQ) támogatása.
- Névtelen típusok támogatása, amellyel sokkal inkább a típusok „alakját”, mint viselkedését modellezhetjük.
- Létező típusok funkcionalitásának kiterjesztése, bővítő metódusok segítségével.
- A lambda operátor (`=>`) bevezetése, amely még inkább leegyszerűsíti a .NET-metódusreferencia-típussal végzett munkát.
- Új objektuminicializáló szintaxis, amely lehetővé teszi a tulajdonságértékek beállítását az objektum létrehozásakor.

A C# nyelvvel kapcsolatban talán az a legfontosabb, hogy csak olyan forráskódot tud létrehozni, amely .NET-futtatókörnyezetben működik (C# segítségével nem tudunk natív COM-kiszolgálót vagy nem felügyelt Win32 API alkalmazást felépíteni). A .NET-futtatókörnyezethez készült forráskód leírására hivatalosan a *felügyelt programkód* kifejezést használjuk. A felügyelt programkódot tartalmazó bináris egységet *szerelvénynek* nevezzük. (A szerelvényekkel kapcsolatos további információkért lásd a „.NET-szerelvények áttekintése" című részt alább.) Azt a forráskódot ellenben, amelyiket a .NET-futtatókörnyezet közvetlenül nem tudja hosztolni, *nem felügyelt programkódnak* nevezzük.

További .NET-alapú programozási nyelvek

.NET-alkalmazásokat nem csak a C# nyelv segítségével építhetünk fel. Amikor 2000-ben, a Microsoft fejlesztői konferenciáján először bemutatták a nagyközönségnek a .NET-platformot, több gyártó is bejelentette, hogy felépítik a saját fordítóprogramjuk .NET-alapú verzióját.

Azóta nyelvek tucatjai estek át a .NET-felvilágosodáson. Azon az öt nyelven kívül, amelyek a Microsoft .NET Framework 3.5 SDK verzióval kerültek forgalomba (C#, Visual Basic .NET, J#, C++/CLI [korábban „Managed Extensions for C++"] és JScript .NET), léteznek .NET-fordítóprogramok Smalltalkra, COBOL-ra és Pascalra is. Jóllehet könyvünk (szinte) kizárólag csak a C#-pal foglalkozik, érdemes ellátogatni a következő webhelyre (a Microsoft fenntartja az URL változtatásának jogát):

```
http://www.dotnetlanguages.net.
```

Ha a főoldal tetején lévő Resources hivatkozásra kattintunk, számtalan .NET programozási nyelvet és azokhoz kapcsolódó hivatkozásokat tartalmazó listát kapunk, ezekre kattintva pedig különféle fordítókat tölthetünk le (lásd 1.2. ábra).

Ha elsősorban a C# szintaxisa alapján szeretnénk .NET-programokat létrehozni, tanácsos felkeresni ezt a webhelyet, itt biztosan találunk ugyanis a célnak megfelelő. NET-nyelvet (esetleg a LISP .NET-et?).

1.2. ábra: A *www.DotNetLanguages.net az egyik olyan webhely, amely ismert .NET programozási nyelveket dokumentál*

Élet egy többnyelvű világban

Amikor a fejlesztők ráébredtek a .NET nyelvagnosztikai természetére, számos kérdés merült fel. Talán az egyik leggyakoribb kérdés ez lehetett: „Ha az összes .NET-nyelv felügyelt programkódot fordít, miért van szükség több fordítóra?”. Ezt a kérdést többféleképpen meg lehet válaszolni. Először is, mi programozók *nagyon* válogatósak vagyunk, ha bármilyen programozási nyelv kiválasztásáról van szó. Néhányan a pontosvesszős és kapcsos zárójeles nyelveket szeretik, a lehető legkevesebb nyelvi kulcsszóval. Mások azokat a nyelveket szeretik, amelyek az ember számára olvashatóbb szintaktikai tokeneket tartalmaznak (mint a Visual Basic). Megint mások pedig a nagyszámítógépes tapasztalataikat szeretnék érvényesíteni, amíg áttérnek a .NET-platformra (a COBOL .NET-en keresztül).

Legyünk őszinték. Ha a Microsoft létrehozott volna egy „hivatalos” .NET-nyelvet, amely a BASIC nyelvcsaládból származik, vajon az összes programozó elégedett volna? Vagy, ha az egyetlen „hivatalos” .NET-nyelv a Fortran szintaxisán alapulna, nyilván sokan lennének, akik a .NET-et végképp semmibe vennék. Mivel a .NET-futtatókörnyezetben nem fontos, hogy milyen nyelv segítségével építették fel a felügyelt kódot, a .NET-programozók hűek maradhatnak ahhoz a szintaxishoz, amelyet korábban választottak, és a lefordított szerelvényeket probléma nélkül megoszthatják a csapattagokkal, a részlegekkel és a külső szervezetekkel (tekintet nélkül arra, hogy a másik fél milyen .NET-nyelvet használ).

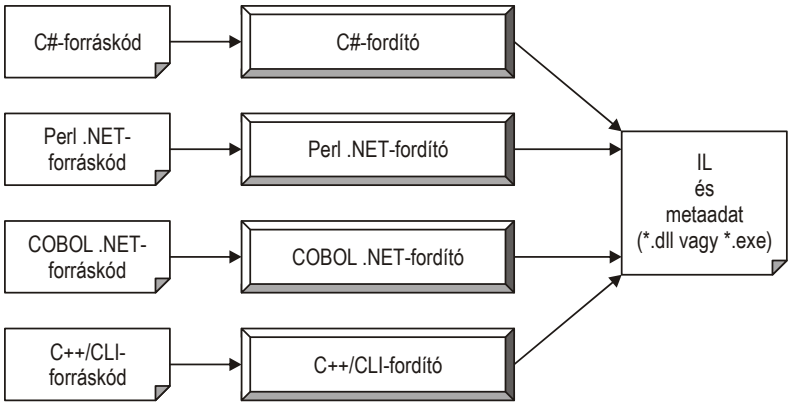
A különféle .NET-nyelveknek egyetlen, egységes szoftvermegoldásba történt integrálásának másik előnye, hogy minden egyes programozási nyelvnek megvannak a maga erősségei és gyengeségei. Egyes programozási nyelvek például remekül támogatják a fejlett matematikai feldolgozást. Mások pedig elsősorban a pénzügyi számításokat, a logikai számításokat, a nagyszámítógépekkel való kommunikációt stb. Ha az egyik programozási nyelv erősségeit vesszük, majd azokat egyesítjük a .NET-platform előnyeivel, voltaképpen mindenki jól jár.

A valóságban persze elég nagy az esély arra, hogy a szoftverek felépítését a kiválasztott .NET-nyelv segítségével fogjuk végezni. Ha azonban egy .NET-nyelv szintaxisát megismerjük, nagyon könnyű lesz egy másikét is megtanulni. Ez szintén meglehetősen hasznosnak tűnik, különösen a szoftvertanácsadóknak. Ha a választott nyelvünk például a C#, de egy olyan ügyfélnél dolgozunk, ahol csak Visual Basic .NET-et használhatunk, a .NET-keretrendszer funkcionalitását így is kihasználhatjuk, és az alapkód általános szerkezetét nagyobb nehézségek nélkül megérthetjük.

A .NET-szerelvények áttekintése

Függetlenül attól, melyik .NET-nyelven programozunk, fogadjuk el, hogy noha a .NET bináris fájlok kiterjesztései megegyeznek a COM-kiszolgálók és a nem felügyelt Win32 bináris fájlok kiterjesztéseivel (`*.dll` vagy `*.exe`), nincs bennük semmiféle belső hasonlóság. A `*.dll` .NET bináris fájlok például nem exportálnak metódusokat a COM-futtatókörnyezettel való kommunikációhoz (mivel a .NET *nem* COM). Továbbá a .NET bináris fájlok nem használnak COM-típuskönyvtárakat, és nincsenek bejegyezve a rendszerleíró adatbázisba.

De talán a legfontosabb, hogy a .NET bináris fájlok platformfüggő utasítások helyett platformfüggetlen *köztes nyelvet (IL)* és típusmetaadatokat tartalmaznak. Az 1.3. ábra az eddigieket mutatja be.



1.3. ábra: *Az összes .NET-alapú fordító IL-utasításokat és metaadatokat ad ki*

Megjegyzés Az „IL" rövidítést illetően meg kell említeni, hogy a .NET kifejlesztése során az IL helyett hivatalosan a Microsoft köztes nyelv (MSIL) kifejezést használták. A .NET végleges kiadásakor azonban már közös köztes nyelvként (CIL) említették. Ebből következően a .NET-irodalomban az IL, az MSIL és a CIL tehát pontosan ugyanarra utal. Az aktuális terminológiát követve a továbiakban a „CIL" rövidítést fogjuk alkalmazni.

Amikor .NET-alapú fordító segítségével hozunk létre egy `*.dll` vagy egy `*.exe` kiterjesztésű fájlt, az eredményül kapott modul *szerelvényt* alkot. (A .NET-szerelvényeket részletesen a 15. fejezetben tárgyaljuk.) A .NET futtató környezet részletes tárgyalása előtt meg kell ismerkednünk az új fájlformátum néhány alapvető tulajdonságával.

A szerelvények tehát köztes nyelvi kódokat tartalmaznak, amelyek elvben a Java-bájtkódhoz hasonlítanak, hiszen addig nem változnak platformfüggő gépi utasítássá, amíg ez nem feltétlenül szükséges. A „feltétlenül szükséges" pont jellemzően az, amikor a .NET-futtatókörnyezet a CIL-utasításblokk (például egy metódusimplementáció) használatára hivatkozik.

A CIL-utasításokon kívül a szerelvények *metaadatokat* is tartalmaznak, amelyek a binárisban található típusok karakterisztikáját részletezik. Ha van például egy `SportsCar` nevű osztályunk, a típusmetaadat olyan részleteket ír le, mint a `SportsCar` ősosztálya, amelynek interfészeit (ha léteznek) a `SportsCar` implementálja, valamint az összes olyan tagot, amelyeket a `SportsCar` típusa támogat.

A .NET-metaadatok hatalmas előrelépést jelentenek a COM-metaadatokhoz képest. Talán már közismert, hogy a COM-binárisok jellemzően hozzárendelt típuskönyvtárakat használnak (ez valamivel több, mint az interfészleíró nyelv [IDL] forráskódjának bináris verziója). A COM-típusinformációkkal az a probléma, hogy nem biztos, hogy megvannak, illetve, hogy az IDL-kód nem tudja dokumentálni a kívülről hivatkozott kiszolgálókat, amelyek az aktuális COM-kiszolgáló megfelelő működéséhez szükségesek. Ezzel szemben a .NET-metaadatok mindig jelen vannak, és az adott .NET-alapú fordító automatikusan generálja őket.

Végezetül, a CIL és a típusmetaadatokon kívül maguk a szerelvények is használnak metaadatot, amelyet *manifesztumnak* nevezünk. A manifesztum a szerelvény aktuális változatáról szóló és kulturális információkat tartalmaz (a sztring- és a képerőforrások lokalizációjához), illetve egy listát, amely a megfelelő végrehajtáshoz szükséges összes külsőleg hivatkozott szerelvényt felsorolja. A következő néhány fejezetben a szerelvények típusainak, metaadatainak és önleíró adatainak vizsgálatához szükséges különféle eszközöket ismerhetjük meg.

Egyfájlos és többfájlos szerelvények

Az esetek nagy többségében egyszerű egy-egy kapcsolat van a .NET-szerelvény és a bináris fájl (`*.dll` vagy `*.exe`) között. Ebből következően, ha egy .NET `*.dll` létrehozásán dolgozunk, úgy vehetjük, hogy a bináris és a szerelvény egy és ugyanaz. Hasonlóképpen, ha egy végrehajtható asztali alkalmazást készítünk, az `*.exe` fájlt egyszerűen tekinthetjük magának a szerelvénynek. (Később azonban, a 15. fejezetben majd kiderül, hogy ez így nem pontos.) Szaknyelven, ha a szerelvény egy `*.dll` vagy egy `*.exe` modulból áll, azt *egyfájlos szerelvénynek* nevezzük. Az egyfájlos szerelvények az összes szükséges CIL-t, metaadatot és hozzárendelt manifesztumot önálló, egyedi és pontosan definiált csomagban tartalmazzák.

A *többfájlos szerelvények* viszont számos .NET bináris fájlból állnak, amelyeket *moduloknak* nevezünk. A többfájlos szerelvény esetében az egyik modul *(elsődleges modul)* tartalmazza a szerelvény menifesztumát (esetleg CIL-utasításokat és különféle típusok metaadatait). A többi modul egy modulszintű manifesztummalsal, CIL-utasításokkal és típusmetaadatokkal rendelkezik. Nyilvánvaló, hogy az elsődleges modul dokumentálja a szükséges másodlagos modulok csoportját a szerelvény manifesztumában.

Miért hozzunk létre többfájlos szerelvényeket? Ha egy szerelvényt különálló modulokra osztunk fel, jóval rugalmasabb telepítési beállításokhoz jutunk. Ha például a felhasználó egy olyan távoli szerelvényre hivatkozik, amelyet szeretne letölteni a számítógépére, a futtatókörnyezet csak a szükséges modulokat fogja letölteni. Ezért felépíthetjük úgy is a szerelvényünket, hogy azokat a típusokat, amelyekre ritkábban van szükség (mint például a `HardDriveRef` nevű osztály), külön modul tárolja.

Ellenben, ha az összes típust egyfájlos szerelvénybe helyezzük, a végfelhasználó jó néhány felesleges adatot fog letölteni (és ez nyilvánvalóan időpocsékolás). Egy szerelvény tehát valóban egy vagy több kapcsolódó modul *logikai csoportosítását* tartalmazza, amelyeket önálló egységként telepítünk és módosítunk.

A közös köztes nyelv (CIL) szerepe

A következőkben vizsgáljuk meg kicsit közelebbről a köztes nyelvi kódot, a típusmetaadatot és a szerelvény-manifesztumot. A CIL olyan nyelv, amely bármilyen platformfüggő gépiutasítás-készlet felett áll. A következő C#-kód például egy hétköznapi számológépet modellez. Ne foglalkozzunk most a pontos szintaxissal, de figyeljünk a `Calc` osztály `Add()` metódusának formátumára:

```
// Calc.cs
using System;

namespace CalculatorExample
{
  // Az osztály tartalmazza az alkalmazás belépési pontját.
  class Program
  {
    static void Main()
    {
      Calc c = new Calc();
      int ans = c.Add(10, 84);
      Console.WriteLine("10 + 84 is {0}.", ans);

      // Várakozás, amíg a felhasználó megnyomja az Enter
      // billentyűt.
      Console.ReadLine();
    }
  }
```

```
  // C#-számológép.
  class Calc
  {
    public int Add(int x, int y)
    { return x + y; }
  }
}
```

Ha ezt a kódfájlt a C#-fordítóval (`csc.exe`) lefordítjuk, olyan egyfájlos `*.exe` szerelvényt kapunk, amely tartalmaz manifesztumot, CIL-utasításokat és metaadatokat, és ezek minden szempontból leírják a `Calc` és a `Program` osztályt.

Megjegyzés A C#-fordító segítségével történő kódfordítást, illetve a grafikus IDE-k, például a Visual Studio, a Visual C# Express és a SharpDevelop, használatát a 2. fejezetben vizsgáljuk meg részletesen.

Ha ezt a szerelvényt például az `ildasm.exe` segítségével szeretnénk megnyitni (lásd alább), láthatjuk, hogy az `Add()` metódus köztes nyelven, a következőképpen jelenik meg:

```
.method public hidebysig instance int32 Add(int32 x,
int32 y) cil managed
{
  // A kód mérete 9 (0x9)
  .maxstack 2
  .locals init (int32 V_0)
  IL_0000:  nop
  IL_0001:  ldarg.1
  IL_0002:  ldarg.2
  IL_0003:  add
  IL_0004:  stloc.0
  IL_0005:  br.s IL_0007
  IL_0007:  ldloc.0
  IL_0008:  ret
} // a Calc::Add metódus vége
```

(Az eredményül kapott metódust a 19. fejezetben találjuk a CIL programozási nyelv alapjainak tárgyalásáról szóló részben.) A hangsúly most azon van, hogy a C#-fordító nem platformfüggő gépi utasításokat ad ki, hanem CIL-re fordít.

Mindez, ahogy láttuk, az összes .NET-alapú fordítóra igaz. Ennek illusztrálására, tételezzük fel, hogy ugyanezt az alkalmazást C# helyett Visual Basic .NET segítségével is létrehoztuk:

```
' Calc.vb
Imports System

Namespace CalculatorExample
  ' A VB "Module" is a class that contains only
  ' static members.
  Module Program
    Sub Main()
      Dim c As New Calc
      Dim ans As Integer = c.Add(10, 84)
      Console.WriteLine("10 + 84 is {0}.", ans)
      Console.ReadLine()
    End Sub
  End Module

  Class Calc
    Public Function Add(ByVal x As Integer, ByVal y As Integer)
                    As Integer
      Return x + y
    End Function
  End Class
End Namespace
```

Ha megnézzük az `Add()` metódus köztes nyelvét, ott hasonló utasításokat találhatunk (a VB .NET-fordító [`vbc.exe`] által optimalizálva):

```
.method public instance int32 Add(int32 x, int32 y) cil managed
{
  // A kód mérete 8 (0x8)
  .maxstack 2
  .locals init (int32 V_0)
  IL_0000: ldarg.1
  IL_0001: ldarg.2
  IL_0002: add.ovf
  IL_0003: stloc.0
  IL_0004: br.s IL_0006
  IL_0006: ldloc.0
  IL_0007: ret
} // a Calc::Add metódus vége
```

Forráskód A Calc.cs és a Calc.vb kódfájlokat az 1. fejezet alkönyvtára tartalmazza.

A köztes nyelv előnyei

Adódik a kérdés: miért volt jó, hogy köztes nyelvre fordítottuk a forráskódot, és nem közvetlenül egy specifikus utasításkészletbe. Ennek egyik oka a nyelvi integráció. Ahogy már láttuk, minden egyes .NET-alapú fordító közel azonos CIL-utasításokat hoz létre. Így az összes nyelv egy jól meghatározott bináris térben kommunikálhat egymással.

Ráadásul, mivel a CIL platformfüggetlen, és maga a .NET-keretrendszer is az, így a Java-fejlesztők számára már jól ismert előnyöket biztosítja (vagyis egy szimpla kódbázis fut több operációs rendszeren). A C# nyelvre létezik nemzetközi szabvány , és a .NET-platform, valamint implementációinak a nagy része már több nem Windows operációs rendszeren is rendelkezésre áll (további információkért lásd a fejezet későbbi részét). Ellentétben a Javával, a .NET lehetővé teszi, hogy a kívánt nyelven hozzunk létre alkalmazásokat.

A köztes nyelv fordítása platformfüggő utasítássá

Mivel a szerelvények inkább CIL-utasításokat, mintsem platformfüggő utasításokat tartalmaznak, a köztes nyelvi kódot használat előtt, menet közben kell lefordítani. Azt az egységet, amely a CIL-kódot CPU-utasításokká alakítja, *azonnal (JIT) fordítónak* nevezzük, néha tréfásan *Jitter*nek. A .NET-futtatókörnyezet minden futó processzor számára fenntart egy JIT-fordítót, s ezek mindegyikét az alapplatformra optimalizálták.

Ha például egy kézieszközbe (pl. Pocket PC-be) szánt alkalmazást készítünk, a megfelelő Jitter kitűnően fut a kevés memóriával rendelkező környezetben. Viszont, ha egy kiszolgálóra (ahol memóriában ritkán van hiány) telepítjük a szerelvényt, a Jittert a sok memóriával rendelkező környezethez optimalizáljuk. Elegendő tehát, ha a fejlesztők megírják azt a kódot, amely JIT-fordítással hatékonyan lefordítható, és a gépeken különböző architektúrákkal is végrehajtható.

Ráadásul, mivel a Jitterek a CIL-utasításokat megfelelő gépi kódokká fordítják, az eredményeket a memóriában a céloperációs rendszernek megfelelő formátumban gyorsítótárazzák. Így, ha a `PrintDocument()` metódust hívjuk meg, a CIL-utasításokat a rendszer az első híváskor platformfüggő utasításokká fordítja, és a későbbi felhasználás érdekében a memóriában tárolja. Ebből következően, ha legközelebb meghívjuk a `PrintDocument()` metódust, a CIL-utasítást nem kell újra lefordítani.

Megjegyzés Azt is megtehetjük, hogy amikor a .NET Framework 3.5 SDK-ban lévő ngen.exe parancssori eszköz segítségével telepítjük az alkalmazást, végrehajtunk egy JIT-előfordítást a szerelvényen. Ezzel tökéletesíthetjük a grafikailag intenzív alkalmazások indulási idejét.

A .NET-típusmetaadat szerepe

A .NET-szerelvények a CIL-utasításokon kívül olyan teljes és szabatos metaadatot tartalmaznak, amely a binárisban definiált összes típust (osztály, struktúra, felsorolt típus stb.), valamint azok tagjait (tulajdonságok, metódusok, események stb.) is leírja. Szerencsére a legfrissebb és legnagyobb típusmetaadat előállítása mindig a fordító (és nem a programozó) feladata. Mivel a .NET-metaadat rendkívül aprólékos, a szerelvények teljesen önleíró egységek.

A .NET-típusmetaadat formátumának szemléltetésére, nézzük meg a korábban megvizsgált, C# `Calc` osztály `Add()` metódusa számára generált metaadatot (az `Add()` metódus VB .NET verziója számára generált metaadat is hasonló):

```
TypeDef #2 (02000003)
-------------------------------------------------------
  TypDefName: CalculatorExample.Calc (02000003)
  Flags     : [NotPublic] [AutoLayout] [Class]
  [AnsiClass] [BeforeFieldInit] (00100001)
  Extends   : 01000001 [TypeRef] System.Object
  Method #1 (06000003)
-------------------------------------------------------
  MethodName: Add (06000003)
  Flags     : [Public] [HideBySig] [ReuseSlot] (00000086)
  RVA       : 0x00002090
  ImplFlags : [IL] [Managed] (00000000)
  CallCnvntn: [DEFAULT]
  hasThis
  ReturnType: I4
    2 Arguments
    Argument #1: I4
    Argument #2: I4
    2 Parameters
     (1) ParamToken : (08000001) Name : x flags: [none] (00000000)
     (2) ParamToken : (08000002) Name : y flags: [none] (00000000)
```

A metaadatot a .NET-futtatókörnyezet számos célra felhasználja ugyanúgy, ahogy a különféle fejlesztőeszközök. Az IntelliSense szolgáltatás például – ezt az olyan eszközök, mint a Visual Studio 2008 tartalmazzák – lehetővé teszi, hogy a szerelvények metaadata a tervezés folyamán olvasható legyen. A metaadatot különféle objektumböngésző segédprogramok, hibakereső eszközök és maga a C#-fordító is használja. Kétségtelen, hogy a metaadat a gerince számos .NET-technológiának, köztük a WCF-nek, az XML-webszolgáltatásoknak vagy a .NET remotingnak, a reflexiónak, a késői kötésnek és az objektumsorosításnak. (A 16. fejezet tartalmazza a .NET-metaadat szerepének formalizálását.)

A szerelvény manifesztumának szerepe

Végül, de nem utolsósorban, ahogy már szó volt róla, a .NET-szerelvények olyan metaadatot tartalmaznak, amely magát a szerelvényt írja le (szakszóval *manifesztum , önleírás)*. Egyéb részletek mellett, a manifesztum rögzíti az aktuális szerelvény megfelelő működéséhez szükséges összes külső szerelvényt, a szerelvény verziószámát, a szerzői jogra vonatkozó adatokat stb. Ugyanúgy, mint a típusmetaadatot, a szerelvény manifesztumát is a fordító generálja. Íme a fejezetben korábban bemutatott `Calc.csc` kódfájl fordításakor generált manifesztum néhány releváns részlete (tételezzük fel, hogy utasítottuk a fordítót, hogy a szerelvényt `Calc.exe`-nek nevezze el):

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 2:0:0:0
}
.assembly Calc
{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module Calc.exe
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
```

Röviden, a manifesztum dokumentálja a `Calc.exe` számára szükséges külső szerelvények listáját (az `.assembly extern` direktíva révén), valamint a szerelvény különböző jellemzőit (verziószám, modulnév stb.). (Az önleíró adatok hasznosságáról a 15. fejezetben lesz szó részletesebben.)

A közös típusrendszer (CTS)

Egy adott szerelvény bármennyi különböző típust tartalmazhat. A .NET világában a *típus* általános kifejezés, amely a halmaz (osztály, interfész, struktúra, felsorolt típus, metódusreferencia) egy tagjára utal. Amikor egy .NET-alapú nyelv segítségével hozunk létre megoldásokat, valószínűleg sok ilyen típussal találkozunk. Például akkor, amikor a szerelvényünk olyan osztályt definiál, amely néhány interfészt implementál.

Előfordulhat, hogy az egyik interfészmetódus bemeneti paramétere egy felsorolt típus lesz, a hívónak pedig egy struktúrát ad vissza.

Ahogy láttuk, a közös típusrendszer (CTS) olyan formális specifikáció, amely azt dokumentálja, hogyan kell a típusokat úgy definiálni, hogy azokat a közös nyelvi futtatórendszer (CLR) befogadja. Jellemzően kizárólag azokat a fejlesztőket érdekli a közös típusrendszer belső működése, akik a .NET-platformot megcélzó eszközöket és/vagy fordítóprogramokat készítenek. Minden .NET-programozó számára fontos azonban, hogy megtanulja, hogyan kell a közös típusrendszer által definiált öt típussal bármilyen választott nyelven dolgozni. Az alábbiakban következzen egy rövid áttekintés.

Az egységes típusrendszer osztálytípusai

Minden .NET-alapú nyelv támogatja legalább az *osztálytípus* fogalmát, amely az objektumorientált programozás (OOP) alappillére. Egy osztálynak bármennyi tagja (pl. tulajdonságok, metódusok és események) és adatpontja (mezője) lehet. A C#-ban a `class` kulcsszóval deklaráljuk az osztályokat:

```
// C# osztálytípus.
class Calc
{
  public int Add(int x, int y)
  { return x + y; }
}
```

A C# segítségével készült CTS-osztályok felépítését az 5. fejezetben vizsgáljuk meg; az 1.1. táblázatban felsoroljuk az osztálytípusok néhány jellemzőjét.

Osztályjellemző	Valós jelentése
Lezárt-e az osztály, vagy sem?	A lezárt osztályok nem lehetnek más osztályok alaposztályai.
Megvalósít az osztály bármilyen interfészt?	Az *interfész* olyan absztrakt tagok gyűjteménye, amelyek kapcsolatot biztosítanak az objektum és az objektumfelhasználó között. Az egységes típusrendszer lehetővé teszi, hogy egy osztály vagy szerkezet bármennyi interfészt megvalósítson.
Az osztály absztrakt vagy valós?	Az *absztrakt osztályokat* nem lehet közvetlenül létrehozni, de egységes viselkedést biztosítanak a származtatott típusoknak. A *valós* osztályokat közvetlenül létre lehet hozni.

Osztályjellemző	Valós jelentése
Milyen az osztály „láthatósága”?	Minden osztályt egy láthatósági attribútummal kell konfigurálni. Alapvetően ez a jellemző határozza meg, hogy használhatják-e külső szerelvények is az osztályt, vagy csak a definiáló szerelvény.

1.1. táblázat: *Az egységes típusrendszer osztályjellemzői*

Az egységes típusrendszer interfésztípusai

Az *interfész* nem más, mint absztrakt tagok definícióinak megnevezett gyűjteménye, amelyet egy adott osztály vagy struktúra támogathat (megvalósíthat). Eltérően a COM-tól, a .NET-interfészek *nem* egy közös ősinterfészből származnak (mint az `IUnknown`). A C#-ban az interfésztípusokat az `interface` kulcsszó segítségével definiálhatjuk:

```
// A C# interfész típust rendszerint publikus
// típusként deklaráljuk, így más szerelvények típusai
// is megvalósíthatják a viselkedését.
public interface IDraw
{
  void Draw();
}
```

Az interfészeknek önmagukban nincs sok hasznuk. Viszont, amikor egy osztály vagy struktúra megvalósít egy adott interfészt a maga egyedi módján, egy interfész-hivatkozás révén polimorf módon hozzáférhetünk a funkcionalitáshoz. (Az interfészalapú programozást a 9. fejezetben tárgyaljuk.)

Az egységes típusrendszer struktúratípusai

A struktúra fogalmát ugyancsak formalizálták az egységes típusrendszerben. Ha a C-t ismerjük, akkor egyértelmű, hogy ezek a felhasználói típusok (UDT-k) a .NET világában is tovább élnek (bár egy kicsit másképp viselkednek). Egyszerűen fogalmazva, a *struktúrát* egy értékalapú szemantikával rendelkező pehelysúlyú osztálytípusnak is tekinthetjük. (A struktúrákkal kapcsolatos további részletekért, lásd a 4. fejezetet.) A struktúrák jellemzően geometriai és matematikai adatok modellezésére alkalmasak, és a C#-ban a `struct` kulcsszó segítségével lehet létrehozni őket:

```
// C# struktúratípus.
struct Point
{
  // A struktúrák mezőket tartalmazhatnak.
  public int xPos, yPos;

  // A struktúrák paraméterezett konstruktorokat tartalmazhatnak.
  public Point(int x, int y)
  { xPos = x; yPos = y;}

  // A struktúrák metódusokat definiálhatnak.
  public void Display()
  {
    Console.WriteLine("({0}, {1}", xPos, yPos);
  }
}
```

Az egységes típusrendszer felsorolt típusai

A *felsorolt típus* olyan praktikus programozási szerkezet, amely lehetővé teszi a név/érték párok csoportosítását. Például tételezzük fel, hogy olyan videojáték-alkalmazást hozunk létre, amely lehetővé teszi a játékosok számára, hogy három karakterkategóriából válasszanak (Wizard – varázsló, Fighter – harcos vagy Thief – tolvaj). A numerikus értékek nyomon követése helyett, amelynek célja az egyes lehetőségek egyesével történő bemutatása lenne, egyedi felsorolt típusokatkat is létrehozhatunk, az `enum` kulcsszó segítségével:

```
// C# felsorolt típus.
enum CharacterType
{
  18 CHAPTER 1 n THE PHILOSOPHY OF .NET
  8849CH01.qxd 10/1/07 10:30 AM Page 18
  Wizard = 100,
  Fighter = 200,
  Thief = 300
}
```

Az alapértelmezés szerint minden egyes elemet 32 bites egész tárol; módosíthatjuk azonban a tárolót, ha szükséges (pl. amikor egy Pocket PC-hez hasonló, kevés memóriás eszközre programozunk). Az egységes típusrendszer emellett azt igényli, hogy a felsorolttípusok egy közös, `System.Enum` ősosztályból származzanak. Ahogy a 4. fejezetből később kiderül: ez az alaposztály egy sor olyan érdekes tagot definiál, amelyek révén a mögöttes név/érték párokat programozással lehet kibontani, manipulálni és átalakítani.

Az egységes típusrendszer metódusreferencia-típusai

A *metódusreferencia* a típusbiztos C-stílusú függvénymutató .NET-megfelelője. A legfőbb különbség az, hogy a .NET-metódusreferencia a `System.MulticastDelegate`-ből származó *osztály*, nem pedig egy nyers memóriacím egyszerű mutatója. A C#-ban a metódusreferenciákat a `delegate` kulcsszóval deklaráljuk:

```
// Ez a C# metódusreferencia típus bármely metódusra mutathat,
// amely egész számot ad vissza, és két egész szám a bemenete.
delegate int BinaryOp(int x, int y);
```

A metódusreferenciák egyrészt akkor nagyon hasznosak, ha valamelyik egyed számára lehetőséget szeretnénk teremteni arra, hogy egy másiknak hívást továbbítson, másrészt a .NET-eseményarchitektúra alapjainak megteremtésekor. A metódusreferenciák a többes küldéshez valódi támogatással rendelkeznek (pl. egy kérés több címzettnek való továbbításához), valamint az aszinkron függvényhíváshoz (részletesen lásd a 11. és 18. fejezetben).

Az egységes típusrendszer típustagjai

Az egységes típusrendszerben formalizált összes típus közül a legtöbbnek bármennyi *tagja* is lehet. Szaknyelven szólva, a *típustagot* a halmaz korlátozza (konstruktor, véglegesítő, statikus konstruktor, beágyazott típus, operátor, metódus, tulajdonság, indexelő, mező, írásvédett mező, konstans, esemény).

Az egységes típusrendszer különféle „díszítőelemeket” definiál, amelyeket hozzárendelhetünk egy adott taghoz. Minden tag rendelkezik például egy adott láthatósági jellemzővel (pl. nyilvános, privát, védett stb.). Néhány tag absztrakt tagként deklarálható, amely a származtatott típusokra polimorf viselkedést kényszerít, valamint virtuális tagként, amely alapértelmezett (de felüldefiniálható) megvalósítást definiál. Sőt a legtöbb tagot statikusként (osztályszinten kötött) vagy példányként (objektumszinten kötött) is konfigurálhatjuk. A típustagok létrehozását a következő néhány fejezet során vizsgáljuk meg.

Megjegyzés A C# nyelv támogatja a generikus típusok és tagok létrehozását is (lásd a 10. fejezetben).

Valódi CTS-adattípusok

Az utolsó szempontként említhető az egységes típusrendszer vizsgálatában, hogy alapvető adattípusok jól meghatározott halmazát hozza létre. Habár minden nyelv rendelkezik egyedi kulcsszóval a valódi CTS-adattípusok deklarálására, az összes kulcsszó az `mscorlib.dll` nevű szerelvényben definiált típust adja meg. Nézzük meg az 1.2. táblázat alapján, hogy milyen kulcsszavakkal fejezzük ki a CTS-adattípusokat a különböző .NET-nyelvekben.

CTS-adattípus	VB .NET-kulcsszó	C#-kulcsszó	C++/CLI-kulcsszó
`System.Byte`	`Byte`	`byte`	`unsigned char`
`System.SByte`	`SByte`	`sbyte`	`signed char`
`System.Int16`	`Short`	`short`	`short`
`System.Int32`	`Integer`	`int`	`int` vagy `long`
`System.Int64`	`Long`	`long`	`__int64`
`System.UInt16`	`UShort`	`ushort`	`unsigned short`
`System.UInt32`	`UInteger`	`uint`	`unsigned int` vagy `unsigned long`
`System.UInt64`	`ULong`	`ulong`	`unsigned __int64`
`System.Single`	`Single`	`float`	`Float`
`System.Double`	`Double`	`double`	`Double`
`System.Object`	`Object`	`object`	`Object^`
`System.Char`	`Char`	`char`	`wchar_t`
`System.String`	`String`	`String`	`String^`
`System.Decimal`	`Decimal`	`decimal`	`Decimal`
`System.Boolean`	`Boolean`	`bool`	`Bool`

1.2. táblázat: *A valódi CTS-adattípusok*

Mivel egy felügyelt nyelv egyedi kulcsszavai csak a `System` névtér valódi típusainak rövidített jelölései lehetnek, többé nem kell a numerikus adatok túlcsordulásától, illetve alácsordulásától tartanunk a különböző nyelveken, ahogy a sztringek és a logikai értékek belső megjelenésétől sem.

Nézzük meg a következő kódrészleteket, amelyek nyelvi kulcsszavak és formális CTS-típusok segítségével definiálnak 32 bites numerikus változókat C# és VB .NET nyelveken.

```
// Egész számok definíciója C# nyelvben.
int i = 0;
System.Int32 j = 0;

' Egész számok definíciója "ints" VB .NET-ben.
Dim i As Integer = 0
Dim j As System.Int32 = 0
```

A közös nyelvi specifikáció (CLS)

A különböző nyelvek tehát ugyanazokat a programozási szerkezeteket egyedi, nyelvspecifikus kifejezésekkel fejezik ki. A C# nyelvben például a sztringek összefűzését a plusz operátorral (+) jelöljük, míg VB .NET-ben az és jelet (&) használjuk. Ha két különböző nyelv ugyanazt a programozási kifejezést használja is (pl. egy olyan függvényt, amelynek nincs visszatérési értéke), nagy az esély, hogy a szintaxis mégis elég eltérő lesz:

```
// C# metódus, amely nem ad vissza semmit.
public void MyMethod()
{
  // Érdekes kód...
}
' VB metódus, amely nem ad vissza semmit.
Public Sub MyMethod()
  ' Érdekes kód...
End Sub
```

Ezek az apró szintaktikai eltérések azonban nem fontosak a .NET-futtatókörnyezet szempontjából, mivel a megfelelő fordítók (ebben az esetben a `csc.exe` vagy a `vb.exe`) hasonló CIL-utasításokat adnak ki. A nyelvek viszont a működésük általános szintje alapján is eltérőek lehetnek.

Egy .NET-nyelv például vagy rendelkezik kulcsszóval az előjel nélküli adatok megjelenítésére, vagy nem, illetve vagy támogatja a mutatótípusokat, vagy nem. Ilyen lehetőségek mellett jó lenne, ha volna egy olyan referenciaadat, amellyel az összes .NET-alapú nyelv összhangban lenne.

A közös nyelvi specifikáció olyan szabálykészlet, amely részletesen leírja, hogy egy adott .NET-alapú fordítónak milyen minimális és teljes szolgáltatásokat kell támogatnia a közös nyelvi futtatórendszerben (CLR) hosztolt forráskódok létrehozásához, illetve ahhoz, hogy minden .NET-platformot célzó nyelv egységes módon férhessen hozzá. A közös nyelvi specifikáció több szempontból az egységes típusrendszer által definiált, teljes funkcionalitás *részhalmazának* tekinthető.

A közös nyelvi specifikáció végül is olyan szabálykészlet, amelyhez a fordító megalkotóinak igazodniuk kell, ha azt szeretnék, hogy a termékük zavartalanul működjön a .NET-univerzumban. Minden szabályhoz olyan egyszerű nevet rendeltek hozzá (pl. 6. CLS-szabály), amely leírja, hogy a szabály milyen hatással van azokra, akik a fordítót létrehozzák, és azokra, akik (valamilyen módon) kapcsolatba kerülnek vele. A közös nyelvi specifikáció alapja a 1. szabály:

- *1. szabály:* A CLS-szabályok a típusnak csak azokra a részeire vonatkoznak, amelyek a definiáló szerelvényen kívülről is hozzáférhetők.

Ebből a szabályból (helyesen) arra következtethetünk, hogy a többi CLS-szabály nem vonatkozik a .NET-típusok belső működésének létrehozásakor alkalmazott logikára. A típusoknak egyetlen szempontból kell összhangban lenniük a közös nyelvi specifikációval: a tagok definíciója (azaz elnevezési rend, paraméterek és visszatérési típusok) alapján. A tagok megvalósítási logikája bármennyi nem CLS-technikát használhat, hiszen a külvilág nem veszi észre a különbséget.

Ennek illusztrálására nézzük az alábbiakat. A következő `Add()` metódus nem CLS-kompatibilis, ugyanis a paraméterek és a visszatérési értékek előjel nélküli adatokat használnak (és ez nem CLS-követelmény):

```
class Calc
{
  // Előjel nélküli adatok hozzáférése nem CLS-kompatibilis!
  public ulong Add(ulong x, ulong y)
  { return x + y;}
}
```

Ha viszont belsőleg, egyszerűen előjel nélküli adatokat használtunk volna:

```
class Calc
{
  public int Add(int x, int y)
  {
    // Mivel az ulong változót csak belsőleg használjuk,
    // a kód továbbra is CLS-kompatibilis.
    ulong temp = 0;
    ...
    return x + y;
  }
},
```

akkor még mindig megfelelnénk a CLS-szabályoknak, és nyugodtak lehetnénk afelől, hogy bármelyik .NET-nyelv képes meghívni az `Add()` metódust.

A közös nyelvi specifikáció persze számos egyéb szabályt is definiál az 1. szabályon kívül. Leírja például, hogy egy adott nyelvnek hogyan kell a sztringeket és a felsorolt típusokat (belül tárolt típus) belsőleg reprezentálni, hogyan definiáljunk statikus tagokat, stb. Szerencsére nem kell ezeket a szabályokat megjegyeznünk ahhoz, hogy profi .NET-fejlesztők legyünk. Összegezve, a közös nyelvi specifikáció és az egységes típusrendszer mélyebb ismerete csak az eszköz-, illetve fordítóalkotók számára szükséges.

A CLS-megfeleltetés biztosítása

Ahogy azt a későbbiekben látni fogjuk, a C# számos, nem CLS-kompatibilis programszerkezetet definiál. Jó hír azonban, hogy egy .NET-attribútum segítségével utasíthatjuk a C#-fordítót, hogy ellenőrizze a kódunk CLS-megfeleltetését:

```
// Utasítja a C#-fordítót a kód CLS-megfeleltetésének ellenőrzésére.
[assembly: System.CLSCompliant(true)]
```

A `[CLSCompliant]` attribútum utasítja a C#-fordítót, hogy ellenőrizze, a kód minden egyes sora megfelel-e a CLS-szabályoknak. Ha szabályellenes dolgot talál, a fordító hibát jelez, és megmutatja a hibás kódot. (Az attribútumalapú programozás részleteit a 16. fejezetben tárgyaljuk.)

A közös nyelvi futtatórendszer (CLR)

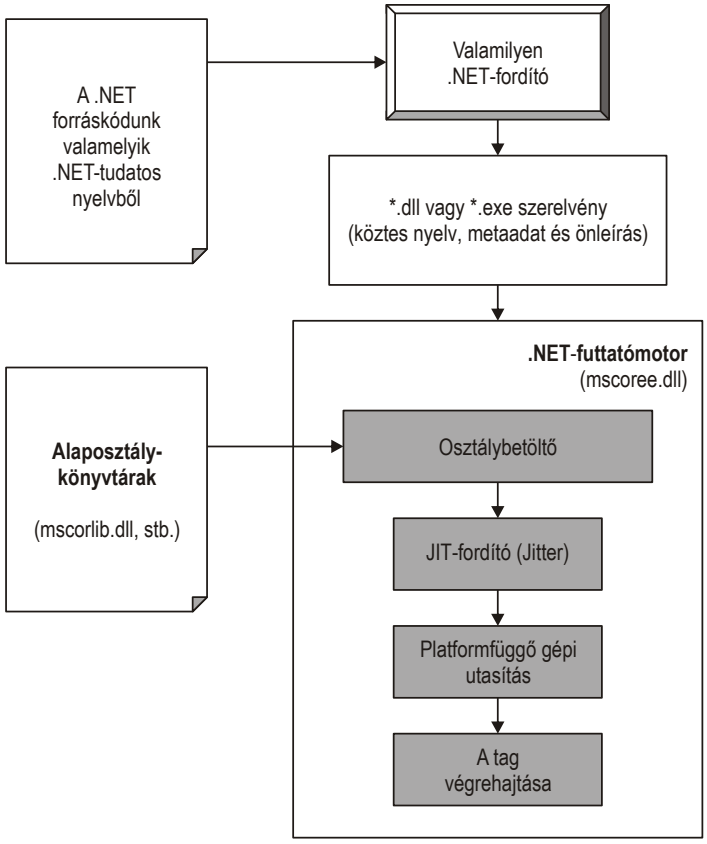
Az egységes típusrendszeren (CTS) és a közös nyelvi specifikáción (CLS) kívül a harmadik hárombetűs rövidítés, amelyről még beszélnünk kell, a közös nyelvi futtatórendszer (CLR). Programozási szempontból a *futtatórendszer* kifejezés olyan külső szolgáltatások gyűjteményére utal, amelyek a kód egy adott lefordított egységének a végrehajtásához szükségesek. Például, amikor a fejlesztők a Microsoft Foundation Classes (MFC) segítségével hoznak létre új alkalmazást, tisztában vannak azzal, hogy a programjuk igényli az MFC-alapkönyvtárat (azaz az `mfc42.dll`-t). A többi népszerű nyelvnek is megvan a megfelelő futtatórendszere. A VB6-programozóknak szintén egy vagy két futtatókörnyezeti modulhoz kell igazodniuk (pl. `msvbvm60.dll`). A Java-fejlesztők a Java virtuális géphez (JVM) kötődnek; stb.

A .NET-platform egy további futtatási rendszert kínál. A legnagyobb különbség a .NET-futtatórendszer és a többi, imént említett futtatórendszer között az, hogy a .NET-futtatórendszer rendelkezik egy jól meghatározott futtatási réteggel, amelyet az *összes* .NET-alapú nyelv és -platform használ.

A közös nyelvi futtatórendszer lényege fizikailag az `mscoree.dll` könyvtárban (más néven Common Object Runtime Execution Engine) jelenik meg. Amikor egy szerelvényt használni szeretnénk, és hivatkozunk rá, az `mscoree.dll` automatikusan betöltődik, és ez tölti be a memóriába a szerelvényt. A futtatórendszer számos feladatért felelős. Elsősorban megadja egy szerelvény helyét, és a binárisban lévő metaadatot olvasva megkeresi az igényelt típust. A CLR ezt követően a memóriában helyezi el a típust, a hozzárendelt CIL-utasítást platformfüggő gépi utasításokká fordítja, végrehajtja a szükséges biztonsági ellenőrzéseket, majd végrehajtja a szóban forgó kódot.

Az egyedi szerelvények betöltése és az egyedi típusok létrehozása mellett, a CLR, ha szükséges, a .NET-alaposztálykönyvtárban tárolt típusokkal is kommunikál. Habár a teljes alaposztálykönyvtár több különálló szerelvényre oszlik, a legfontosabb szerelvény (`mscorlib.dll`) sok alaptípust tartalmaz, amelyek között sok közös programozási feladat található az összes .NET-nyelv által használt alapvető adattípussal egyetemben. Amikor .NET-megoldásokat hozunk létre, automatikusan hozzáférünk ehhez a szerelvényhez.

Az 1.4. ábra a forráskód (amely az alaposztálykönyvtár-típusokat használja), egy adott .NET-fordító és a .NET-futtatómotor közötti munkafolyamot mutatja be.



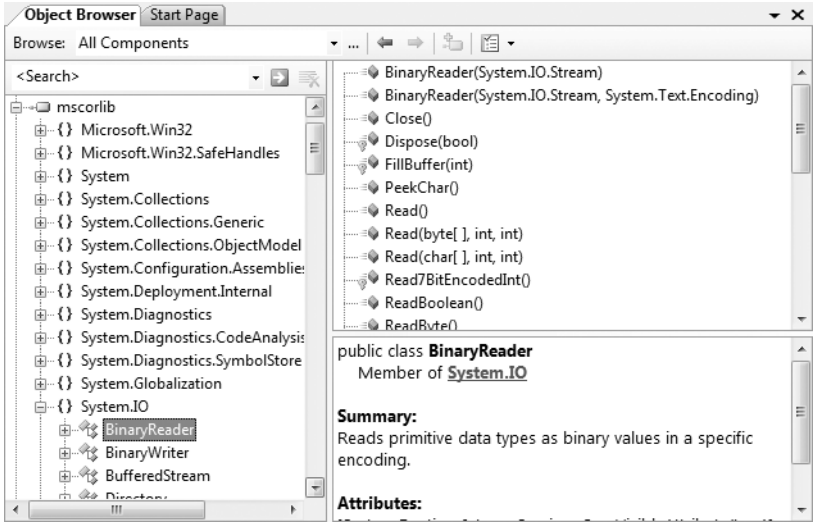
1.4. ábra: Az *mscoree.dll működés közben*

A szerelvény, a névtér és a típus megkülönböztetése

Mindannyian elfogadjuk a forráskódkönyvtárak fontosságát. Az MFC-hez, J2EE-hez és az ATL-hez hasonló könyvtáraknak az a célja, hogy a meglévő forráskódok olyan jól meghatározott készletét biztosítsa a fejlesztők számára, amelyet az alkalmazásaikban felhasználhatnak. A C# nyelv azonban nem rendelkezik nyelvfüggő kódkönyvtárral. A C#-fejlesztők ehelyett a nyelvsemleges .NET-könyvtárakat használják. Annak érdekében, hogy a típusok rendezetten helyezkedjenek el az alaposztálykönyvtárakban, a .NET-platform messzemenően alkalmazza a *névtér* elvét.

Egyszerűen fogalmazva, a névtér az egy szerelvényben lévő, szemantikailag kapcsolódó típusok csoportja. A `System.IO` névtér például I/O-típusokat tartalmaz, a `System.Data` névtér pedig alapvető adatbázistípusokat definiál; stb. Hangsúlyoznunk kell, hogy egyetlen szerelvény (pl. az `mscorlib.dll`) bármennyi névteret tartalmazhat, s ezek mindegyike bármennyi típussal rendelkezhet.

Ennek illusztrálására mutatja be az 1.5. ábra a Visual Studio 2008 Object Browserének egyik képernyőfotóját. Ezzel az eszközzel megvizsgálhatjuk azokat a szerelvényeket, amelyekre az aktuális projektünk hivatkozik, az egy adott szerelvényben lévő névtereket, az egy névtérben lévő típusokat, valamint egy bizonyos típus tagjait. Az `mscorlib.dll` tehát több különböző névteret tartalmaz, mindegyiket a saját, szemantikailag kapcsolódó típusaival.



1.5. ábra: *Egy szerelvény bármennyi névteret tartalmazhat*

A legfontosabb különbség e között a felfogás és egy MFC-hez hasonló, nyelvfüggő könyvtár között, hogy bármelyik nyelv, amelyik a .NET-futtatórendszert veszi célba, *ugyanazokat* a névtereket és *ugyanazokat* a típusokat használja. A következő három program ugyanazt a „Hello World” alkalmazást mutatja be C#, VB .NET és C++/CLI nyelveken:

```
// Hello world C# nyelven
using System;

public class MyApp
{
  static void Main()
  {
    Console.WriteLine("Hi from C#");
  }
}

' Hello world VB nyelven
Imports System

Public Module MyApp
  Sub Main()
    Console.WriteLine("Hi from VB")
  End Sub
End Module

// Hello world C++/CLI nyelven
#include "stdafx.h"
using namespace System;

int main(array<System::String ^> ^args)
{
  Console::WriteLine(L"Hi from C++/CLI");
  return 0;
}
```

Minden egyes nyelv használja a `System` névtérben definiált `Console` osztályt. Apróbb szintaktikai eltéréseken túl, ez a három alkalmazás nagyon sokban hasonlít egymásra, mind fizikailag, mind logikailag.

.NET-fejlesztőként az elsődleges célunk természetesen a számos .NET-névtérben definiált, temérdek típus megismerése. A legalapvetőbb névtér, amelylyel találkozunk, a `System`. Ez a névtér biztosítja azoknak a típusoknak az alapvető készletét, amelyekre .NET-fejlesztőként időről időre szükségünk van. Valójában egyetlen működőképes C#-alkalmazást sem hozhatunk létre anélkül, hogy legalább ne hivatkoznánk a `System` névtérre, hiszen az alapvető adattípusokat (`System.Int32`, `System.String` stb.) ez a névtér definiálja. Az 1.3. táblázatban a kapcsolódó működések alapján csoportosított, néhány (de nem az összes) .NET-névtér látható.

.NET-névtér	Valós jelentése
`System`	A `System` névtérben számos hasznos típust találunk, amelyek valós adatokkal, matematikai számításokkal, véletlenszám-generálással, környezeti változókkal és szemétgyűjtéssel foglalkoznak, továbbá közösen használt kivételeket és attribútumokat.
`System.Collections` `System.Collections.Generic`	Ezek a névterek számos tárolótípust definiálnak, valamint alaptípusokat és interfészeket, amelyek révén testre szabott gyűjteményeket építhetünk fel.

`System.Data`
`System.Data.Odbc`
`System.Data.OracleClient`
`System.Data.OleDb`
`System.Data.SqlClient`
	Ezeket a névtereket a relációs adatbázisokkal folytatott kommunikációra használjuk az ADO .NET alkalmazásakor.

`System.IO`
`System.IO.Compression`
`System.IO.Ports`
	Ezek a névterek olyan típusokat definiálnak, amelyeket fájl I/O-ra, adattömörítésre és portkezelésre használunk.

`System.Reflection` `System.Reflection.Emit`	Ezek a névterek olyan típusokat definiálnak, amelyek támogatják a futási időben történő típusmeghatározást, valamint a típusok dinamikus létrehozását.
`System.Runtime.InteropServices`	Ez a névtér biztosítja a .NET-típusok számára azt a képességet, hogy a nem felügyelt kódokkal (pl. C-alapú Dll-ek és Com-kiszolgálók) kommunikáljanak és viszont.
`System.Drawing` `System.Windows.Forms`	Ezek a névterek olyan típusokat definiálnak, amelyek révén a .NET eredeti felhasználói felületének eszközkészletét (Windows-űrlapok) használó asztali alkalmazásokat hozhatunk létre.

`System.Windows`
`System.Windows.Controls`
`System.Windows.Shapes`
	A `System.Windows` névtér, számos egyéb névtér gyökere, amely a Windows Presentation Foundation (WPF) felhasználói felületi eszközkészletet reprezentálja.

`System.Linq`
`System.Xml.Linq`
`System.Data.Linq`
	Ezek a névterek a LINQ API-ban végzett programozás során alkalmazott típusokat definiálják.

.NET-névtér	Valós jelentése
`System.Web`	Ez az egyik olyan névtér a sok közül, amelyik lehetővé teszi az ASP .NET webalkalmazások létrehozását.
`System.ServiceModel`	Ez az egyik olyan névtér a sok közül, amelyekkel a WCF API segítségével elosztott alkalmazásokat hozhatunk létre.
`System.Workflow.Runtime` `System.Workflow.Activities`	Ez a két névtér olyan típusokat definiál, amelyekkel a WF API segítségével munkafolyamat-engedélyezett alkalmazásokat hozhatunk létre.
`System.Threading`	Ez a névtér számos olyan típust definiál, amelyeket többszálú alkalmazások létrehozására használunk.
`System.Security`	A Security a .NET-univerzum integrált nézőpontja. A biztonságos névterekben számos olyan típust találunk, amelyek engedélyekkel, kriptográfiával stb. foglalkoznak.
`System.Xml`	Az XML-központú névterek számos olyan típust tartalmaznak, amelyeket az XML-adatokkal való kommunikációra használunk.

1.3. táblázat: *A .NET-névterek áttekintése*

A Microsoft-névterek szerepe

Az 1.3. táblázatot végigolvasva bizonyára feltűnt, hogy jó néhány beágyazott névtérnek (`System.IO`, `System.Data` stb.) a System a gyökérnévtere. Ahogy azonban kiderült, a .NET-alaposztálykönyvtár definiálja a `System` névtéren kívüli legfontosabb gyökérnévterek nagy részét, amelyek közül a leghasznosabb a Microsoft.

Dióhéjban, a Microsoft névtérbe ágyazott névterek (pl. `Microsoft.CSharp`, `Microsoft.Ink`, `Microsoft.ManagementConsole` és `Microsoft.Win32`) olyan típusokat tartalmaznak, amelyek egyedi, Windows operációs rendszerbeli szolgáltatásokkal való kommunikációra használatosak. Így ezeket a típusokat más .NET-engedélyezett operációs rendszerrel (pl. a Mac OS X-szel) nem használhatjuk. Könyvünk a Microsoft-gyökerű névterek részleteit azonban nem tárgyalja.

Megjegyzés A .NET Framework 3.5 SDK dokumentációjának használatát a 2. fejezet írja le, részletekkel szolgálva a névterekről, a típusokról és az alaposztálykönyvtárakban található tagokról.

Programozott hozzáférés a névterekhez

A névtér tehát nem más, mint a kapcsolódó típusok logikai megértésének és elrendezésének a megfelelő módja. A `System` névtérre gondolva, azt feltételezhetjük, hogy a `System.Console` egy olyan `Console` nevű osztályt jelent, amely a `System` nevű névtérben található. A .NET-futtatórendszer szerint azonban ez nem így van. A futtatómodul csak egy `System.Console` nevű egységet lát.

A C#-ban a `using` kulcsszó leegyszerűsíti a névterekben definiált típusokra történő hivatkozás folyamatát. A működése a következő. Tegyük fel, hogy egy hagyományos asztali alkalmazást szeretnénk létrehozni. A főablak egy kiszolgálóoldali adatbázisból származó adatokra épülő oszlopdiagramot jelenít meg, valamint a cégünk logóját. Amíg kevéssé ismerjük még a névterekben lévő típusokat, íme néhány, amelyekre hivatkozhatunk a programunkban:

```
// Íme, az alkalmazás elkészítéséhez szükséges összes névtér.
using System;              // Általános alaposztálykönyvtár típusok.
using System.Drawing;        // Grafikus renderelési típusok.
using System.Windows.Forms;  // Windows Forms GUI céleszköz típusok.
using System.Data;           // Általános adatközpontú típusok.
using System.Data.SqlClient; // MS SQL Server adathozzáférés
                             // típusok.
```

Ha egyszer már meghatároztunk néhány névteret (és hivatkoztunk az őket definiáló szerelvényekre), szabadon létrehozhatjuk a bennük található típusok példányait. Ha például a `Bitmap` osztály (amelyet a `System.Drawing` névtér definiál) egy példányát szeretnénk létrehozni, írjuk a következőket:

```
// A fájl által használt névterek listázása explicit módon.
using System;
using System.Drawing;

class Program
{
  public void DisplayLogo()
  {
    // 20 * 20 pixeles bittérkép létrehozása.
    Bitmap companyLogo = new Bitmap(20, 20);
    ...
  }
}
```

Mivel a kódfájlunk a `System.Drawing`-ra hivatkozik, a fordító ennek a névtérnek a tagjaként értelmezi a `Bitmap` osztályt. Ha nem határoztuk meg a `System.Drawing` névteret, fordítási hibához jutunk. Teljesen meghatározott névvel azonban szabadon deklarálhatunk változókat is:

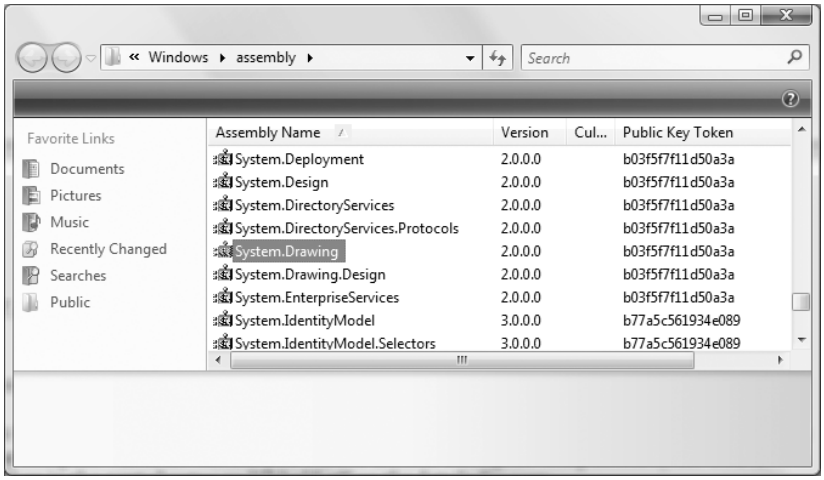
```
// A System.Drawing névteret nem listázzuk!
using System;

class Program
{
  public void DisplayLogo()
  {
    // Teljesen minősített név használata.
    System.Drawing.Bitmap companyLogo =
      new System.Drawing.Bitmap(20, 20);
    ...
  }
}
```

Ha a típusdefiniálás teljesen minősített névvel történik, akkor tökéletesebb olvashatóságot biztosít, a C# `using` kulcsszava ezzel szemben lecsökkenti a billentyűleütések számát. A könyvben nem fogjuk a teljesen minősített neveket használni (kivéve, ha kétértelműséget kell feloldani), hanem a C# `using` kulcsszavának egyszerűsített szemléletét követjük.

Ne feledjük azonban, hogy a `using` kulcsszó a típus teljesen minősített nevének a rövidített jelölése, és mindkét szemlélet pontosan ugyanazt a mögöttes CIL-t eredményezi (ugyanis a CIL-kód mindig a teljesen minősített neveket használja), illetőleg nincs hatással a teljesítményre vagy a szerelvény méretére.

Hivatkozás külső szerelvényekre

Amellett, hogy a C# `using` kulcsszavával megadjuk a névteret, a hivatkozott típus aktuális CIL-definícióját tartalmazó szerelvény nevét is meg kell adnunk a C#-fordítónak. Amint láttuk, sok alapvető .NET-névtér az `mscorlib.dll`-ben található. A `System.Drawing.Bitmap` típust azonban a `System.Drawing.dll` nevű különálló szerelvényben találjuk. A .NET-keretrendszer szerelvényeinek túlnyomó többsége a globális szerelvénytárban (GAC) található. Ez a könyvtár egy Windows-gépen alapértelmezés szerint a C:\Windows\Assembly elérési útvonalon érhető el, ahogy az 1.6. ábra mutatja.



1.6. ábra: *A globális szerelvénytár (GAC) alaposztálykönyvtárai*

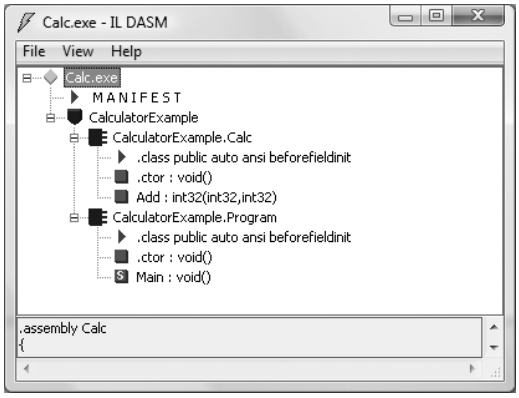
Attól függően, hogy milyen fejlesztőeszköz segítségével hozzuk létre a .NET-alkalmazásokat, többféleképpen is közölhetjük a fordítóval, hogy mely szerelvényeket szeretnénk beemelni a fordítási ciklusba. Ennek a menetével a következő fejezetben ismerkedünk meg.

Szerelvények feltárása az ildasm.exe segítségével

Nincs szükség arra, hogy a .NET-platform összes névterét ismerjük és kezeljük, hiszen egy névteret az tesz egyedivé, hogy szemantikailag kapcsolódó típusokat tartalmaz. Ezért, ha nincs szükségünk felhasználói felületre egy egyszerű konzolalkalmazáson kívül, el is felejthetjük a `System.Windows.Forms`, `System.Windows` és a `System.Web` névtereket. Ha egy rajzoló alkalmazást hozunk létre, ahhoz az adatbázis-névtereknek valószínűleg nincs sok közük. Ahogy az előre gyártott kódkészletekkel általában lenni szokott, használat közben ismerjük meg őket.

A .NET Framework 3.5 SDK-val megjelent Intermediate Language Disassembler (`ildasm.exe`) lehetővé teszi bármilyen .NET-szerelvény betöltését és tartalmának vizsgálatát, köztük a hozzárendelt manifesztumot, a IL kódot és a típusmetaadatot. Az `ildasm.exe` betöltéséhez nyissuk meg a Visual Studio parancssort (Start > All Programs > Microsoft Visual Studio 2008 > Visual Studio Tools), írjuk be, **ildasm**, majd nyomjuk meg az Enter billentyűt.

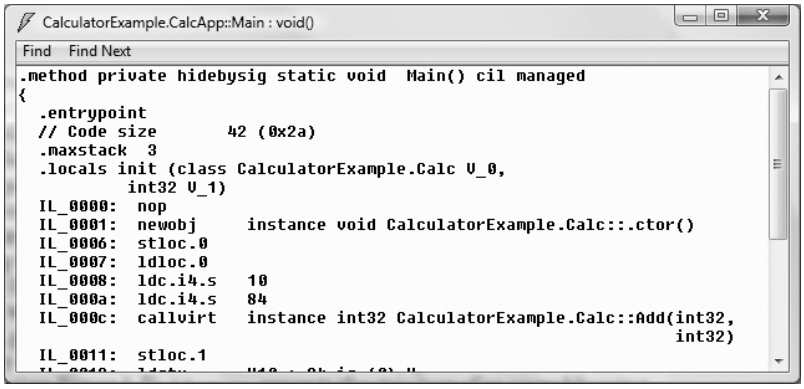
Ha már elindítottuk ezt az eszközt, csak nyissuk meg a File > Open menüpontot, és navigáljunk a vizsgálni kívánt szerelvényhez. Ennek szemléltetésére, nézzük meg a `Calc.exe` szerelvényt, amelyet a fejezet korábbi részében bemutatott `Calc.csc` fájl alapján generáltunk (lásd 1.7. ábra). Az `ildasm.exe` egy hasonló, fanézetű szerelvény szerkezetét mutatja.



1.7. ábra: *Az ildasm.exe lehetővé teszi a .NET-szerelvény köztes nyelvi kódjának, manifesztumának és metaadatának megtekintését*

A köztes nyelvi kód megtekintése

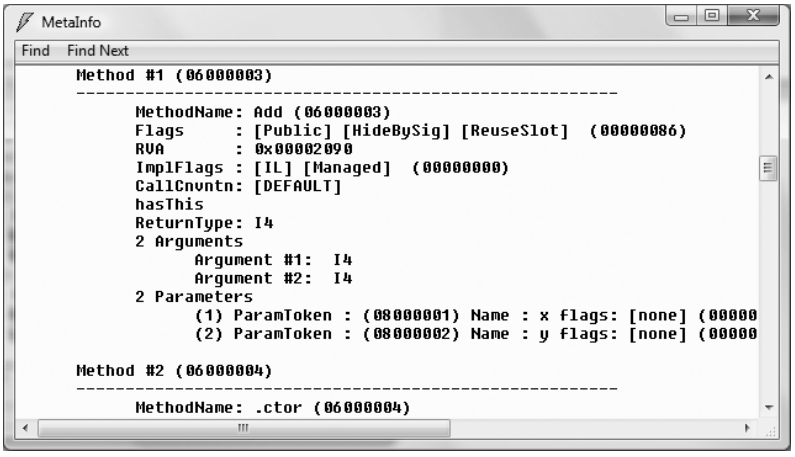
Amellett, hogy megmutatja az adott szerelvényben található névtereket, típusokat és tagokat, az `ildasm.exe` segítségével egy adott tag CIL-utasításait is megtekinthetjük. Ha például kétszer kattintunk a `Program` osztály `Main()` metódusára, egy új ablakban megjelenik a mögöttes köztes nyelvi kód (lásd 1.8. ábra).



1.8. ábra: *A mögöttes köztes nyelvi kód megtekintése*

A típusmetaadat megtekintése

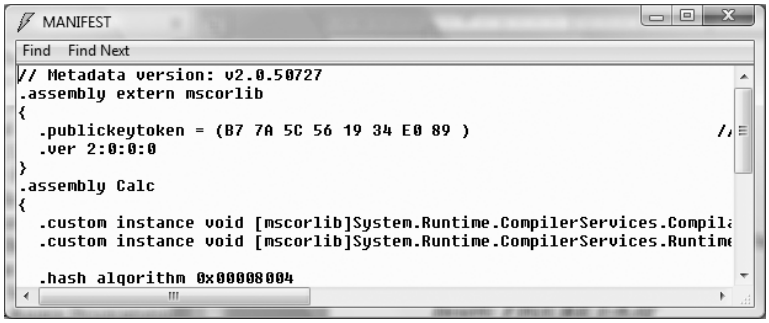
Az aktuálisan betöltött szerelvény típusmetaadatának megtekintéséhez, nyomjuk meg a Ctrl + M billentyűkombinációt. Az 1.9. ábrán a `Calc.Add()` metódus metaadatát láthatjuk.



1.9. ábra: *Típusmetaadat megtekintése az ildasm.exe segítségével*

Szerelvény metaadatának (manifesztumának) megtekintése

Végezetül, ha szeretnénk megtekinteni egy szerelvény manifesztumának tartalmát, kattintsunk kétszer a MANIFEST ikonra (lásd 1.10. ábra).



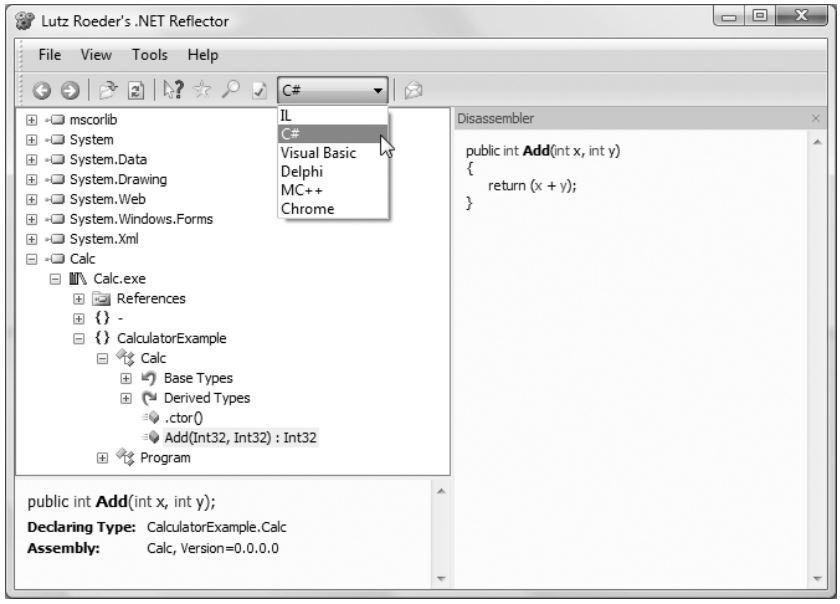
1.10. ábra: *Önleíró adatok megtekintése az ildasm.exe segítségével*

Természetesen az `ildasm.exe` többre is képes, mint amit itt bemutattunk, ezeket a megfelelő részeknél további funkciókkal is szemléltetjük.

Szerelvények feltárása a Lutz Roeder's Reflector segítségével

Az `ildasm.exe` segítségével végzett kutatás a .NET-binárisban nagyon gyakori feladat, az egyetlen bökkenője, hogy csak a mögöttes köztes nyelvi kódot tekinthetjük meg, ahelyett, hogy egy szerelvény megvalósításába nézhetnénk bele az általunk kiválasztott nyelv segítségével. Szerencsére sok .NET-objektumböngésző, köztük a népszerű Reflector is letölthető.

Ezt az ingyenes eszközt a http://www.aisto.com/roeder/dotnet címről lehet letölteni. Kicsomagolás után futtathatjuk is az eszközt, és megadhatunk neki bármilyen szerelvényt, amelyet szeretnénk megnyitni a File > Open menüpont alatt. Az 1.11. ábra a `Calc.exe` alkalmazást mutatja be újra.



1.11. ábra: *A Reflector nagyon népszerű objektumböngésző eszköz*

A `reflector.exe` egy Disassembler ablakot (a szóköz billentyű lenyomására nyílik meg) is támogat, valamint egy legördülő listát, amelyben a választott nyelv mögöttes alapkódbázisát tekinthetjük meg (benne természetesen a köztes nyelvi kódot).

Az eszköz további funkcióit könyvünk nem részletezi. A különféle elméletek szemléltetésére mind az `ildasm.exe`, mind a `reflector.exe` a segítségünkre lesz a továbbiakban.

A .NET-futtatókörnyezet telepítése

Bizonyára nem meglepő, hogy a .NET-szerelvények csak olyan gépeken futtathatók, amelyekre telepítve van a .NET-keretrendszer. Annak, aki .NET-szoftvereket épít, ez nem jelenthet problémát, hiszen az ingyenesen elérhető .NET-keretrendszer 3.5 SDK verziójának (ugyanúgy, mint a kereskedelemben kapható .NET fejlesztői környezetek, mint például a Visual Studio 2008) telepítésével megfelelően konfiguráljuk a gépünket.

Ha azonban olyan gépre telepítünk szerelvényt, amelyen nincs .NET, a szerelvény nem fog tudni lefutni. Ezért a Microsoft kiadott egy `dotnetfx3setup.exe` nevű telepítőcsomagot, amely ingyenesen letölthető és telepíthető a .NET-szoftverekkel együtt. Ez a telepítőprogram a Microsoft .NET-letöltő területéről (http://msdn.microsoft.com/netframework) elérhető. Ha telepítettük a `dotNetFx35setup.exe`-t, a célszámítógép már tartalmazza a .NET-alaposztálykönyvtárakat, a .NET-futtatórendszert (`mscoree.dll`) és egyéb .NET-infrastruktúrákat (mint pl. a GAC).

Megjegyzés A Vista operációs rendszerbe előre konfigurálták a szükséges .NET-futtatókörnyezet infrastruktúráját. Ha azonban Windows XP-re vagy Windows Server 2003-ra telepítjük az alkalmazást, meg kell bizonyosodnunk arról, hogy telepítve van-e a célszámítógépre a .NET futási környezet.

A .NET platformfüggetlen természete

A fejezet lezárásaként, röviden szólnunk kell a .NET platformfüggetlen természetéről. A .NET-szerelvények nem Microsoft operációs rendszereken is fejleszthetők és futtathatók (Mac OS X, számos Linux–disztribúció, Solaris, hogy csak egy párat említsünk). Ahhoz, hogy ezt megértsük, meg kell ismernünk a .NET-univerzum még egy hárombetűs rövidítését: a CLI-t (közös nyelvi infrastruktúra).

Amikor a Microsoft kiadta a C# programozási nyelvet és a .NET-platformot, megalkottak egy formális dokumentumkészletet is, amely a C# és a köztes nyelv, a .NET-szerelvényformátum, az alapvető .NET-névterek és a feltételezett .NET-futtatómotor (ismertebb nevén, virtuális végrehajtó rendszer vagy VES) működésének szintaxisát és szemantikáját írja le.

Ezeket a dokumentumokat az ECMA-hoz is benyújtották (ahol jóváhagyták őket) hivatalos nemzetközi szabványként (http://www.ecma-international.org). Lényeges előírások:

- *ECMA-334:* The C# Language Specification
- *ECMA-335:* The Common Language Infrastructure (CLI)

Ezek a dokumentumok azért fontosak, mert lehetővé teszik a .NET-platform disztribúcióinak létrehozását külső gyártók számára az összes operációs rendszerre és/vagy processzorra. A két előírás közül talán az ECMA–335 a tartalmasabb, így azt több részre bontották, ahogy azt az 1.4. táblázatban láthatjuk.

Az ECMA-335 részei	Valós jelentése
I. rész: Architektúra	A CLI általános architektúráját írja le, benne a CTS és a CLS szabályait, valamint a .NET-futtatómotor működését.
II. rész: Metaadat	A .NET-metaadat részleteit írja le.
III. rész: CIL	A CIL-kód szintaxisát és szemantikáját írja le.
IV. rész: Könyvtárak	Magas szintű áttekintést ad a .NET-disztribúció által kötelezően támogatandó minimális és teljes osztálykönyvtárakról.
V. rész: Mellékletek	Olyan részletek gyűjteménye, mint az osztálykönyvtár-tervezési útmutató és a CIL-fordító megvalósítása.

1.4. táblázat: *A CLI részei*

A IV. rész (Könyvtárak) csak *minimális* névtérkészletet definiál, ezek egy CLI-disztribúció alapvető szolgáltatásai (gyűjtemények, konzol I/O, fájl I/O, szálkezelés, reflexió, hálózati hozzáférés, alapvető biztonsági intézkedések, XML-kezelés stb.). A CLI *nem* definiál olyan névtereket, amelyek megkönnyítik a webes fejlesztést (ASP .NET), az adatbázis-hozzáférést (ADO .NET) vagy a grafikus felhasználói felületen (GUI) az alkalmazásfejlesztést (Windows Forms/Windows Presentation Foundation).

Jó hír azonban, hogy a .NET-disztribúciók fő irányvonala az ASP .NET, az ADO .NET és a Windows Forms Microsoft-kompatibilis ekvivalenseivel bővíti ki a CLI-könyvtárakat, hogy teljes körű, termékszintű fejlesztői platformokat kínáljon.

A CLI-nek a mai napig két fő megvalósítása létezik (a Microsoft Windows-specifikus kínálatán kívül). Habár könyvünk a Microsoft .NET-disztribúcióival létrehozott .NET-alkalmazásokra összpontosít, az 1.5. táblázatban a Mono és a Portable .NET-projektekről adunk tájékoztatást.

Disztribúció	Valós jelentése
`http://www.mono-project.com`	A Mono projekt egy nyílt forráskódú CLI-disztribúció, amely különböző Linux-disztribúciókhoz (pl. SuSE, Fedora stb.), valamint Win32-höz és Mac OS X-hez készült.
`http: //www.dotgnu.org`	A Portable. NET egy másik nyílt forráskódú CLI-disztribúció, amely számos operációs rendszeren fut. A Portable. NET a lehető legtöbb operációs rendszert célozza meg (Win32, AIX, BeOS, Mac OS X, Solaris, az összes fontosabb Linux-disztribúció stb.).

1.5. táblázat: *Nyílt forráskódú .NET-disztribúciók*

Mind a Mono, mind a Portable. NET tartalmaz egy ECMA-kompatibilis C#-fordítót, .NET-futtatómotort, forráskódpéldákat, dokumentációkat, valamint számos olyan fejlesztőeszközt, amelyek működésük szempontjából egyenértékűek a Microsoft .NET keretrendszer 3.5 SDK verziójában találhatókkal. Továbbá, a Mono-ban és a Portable. NET-ben is megtalálható egy VB .NET-, egyJava- és egy C-fordítók.

Megjegyzés A Monót használó platformközi .NET-alkalmazások lefedése a II. kötetben helyet kapó B függelékben található.

Összefoglalás

A fejezet célja az volt, hogy a könyv többi részének megértéséhez szükséges fogalmakat magyarázza. A .NET előtti technológiák korlátainak és bonyolultságának bemutatásával kezdtük, majd annak az áttekintésével folytattuk, hogyan próbálja meg a .NET és a C# leegyszerűsíteni a jelenlegi helyzetet.

A .NET lényege alapvetően egy futtatómotor (`mscoree.dll`) és egy alaposztálykönyvtár (`mscorlib.dll`). A közös nyelvi futtatórendszer (CLR) képes hosztolni bármilyen .NET-binárist (szerelvényt), amely a felügyelt forráskód szabályainak megfelel. Ahogy azt láthattuk, a szerelvények CIL-utasításokat tartalmaznak (a típusmetaadattal és a szerelvény manifesztumával együtt), amelyek egy JIT-fordító segítségével platformfüggő utasításokká alakulnak. Ezután megvizsgáltuk a közös nyelvi specifikáció (CLS) és az egységes típusrendszer (CTS) szerepét.

Majd az `ildasm.exe` és a `reflector.exe` objektumböngésző segédprogramok bemutatása következett, valamint annak ismertetése, hogyan konfiguráljunk egy számítógépet a `dotnetfx3setup.exe` segítségével, hogy .NET-alkalmazásokat hosztoljon. A C# és a .NET-platform platformfüggetlen természetének rövid leírásával zárul ez a rész (ezt a leírást a B függelékben részletezzük).