

Ez a megközelítés nyilvánvalóan csak akkor működik, ha a típustól (ebben az esetben a Sportcar-ról) fordítási idejű információkkal rendelkezünk, és a típus egy példánya pillanatnyilag a memóriában található. Ezzel a megszortí-
tással érthetővé válik, hogy az olyan eszközök, mint az `Idasm.exe`, a típusin-
formációkhoz nem jutnak hozzá úgy, hogy a `system.object.GetType()` meto-
dust közvetlenül meghívják minden egyes típushoz, hiszen az `Idasm.exe`
fordítása nem az egyedi szereplvények használatával történt.

```
// Típusinformáció megszerzése egy Sportcar példány használatával.  
Sportcar sc = new Sportcar();  
Type t = sc.GetType();
```

A `Type` osztály egy példányát többféleképpen is megszerezhetjük. Mivel a `Type`
absztrakt osztály, az egyetlen dolog, amely nem működik, ha a `new` kulcsszóval
közvetlenül létrehozzuk a `Type` objektumot. A `system.object` viszont definiál-
ja a `GetType()` metódust, amely az aktuális objektum metaadatát megjelenítő
`Type` osztálypéldányt adja vissza:

Típusreferencia megszerzése a System.Object.GetType() segítségével

16.2. táblázat: A `System.Type` osztály néhány tagja

Típus	Valós jelentése
<code>GetConstructors()</code>	Ezek a metódusok teszik lehetővé, hogy hozzáférjünk a számunkra érdekes elemeket (interfész, metódus, tulaj-
<code>GetEvents()</code>	donság stb.) bemutató tömbhöz. Mindegyik metódus a
<code>GetTypeIds()</code>	kapcsolódó tömböt adja vissza (pl. a <code>GetTypeIds()</code> metó-
<code>GetInterfaces()</code>	dus a <code>TypeInfo</code> tömböt, a <code>GetMethods()</code> a <code>MethodInfo</code>
<code>GetMembers()</code>	tömböt). Ezeknek a metódusoknak létezik egyes számu
<code>GetMethods()</code>	formája (például <code>GetMethod()</code> , <code>GetProperty()</code> stb.) is,
<code>GetNestedTypes()</code>	amely azt teszi lehetővé, hogy név szerint keressünk ki
<code>GetPropertyInfos()</code>	egy elemet az összes elem lekérdezése helyett.
<code>FindMembers()</code>	Ez a metódus a keresési feltételek alapján <code>MemberInfo</code>
<code>GetType()</code>	típusok tömbjét adja vissza.
<code>InvokeMember()</code>	Ez a statikus metódus visszaadja a megadott típusnév- hez tartozó <code>Type</code> példányt.
	Ez a metódus biztosítja az adott elemhez történő késői kötését.

```
// Típusinformáció megszerzése egy olyan beágyazott felsorolt
// típushoz, amely az aktuális szerelevényben található.
type t = type.GetType("CarLibrary.JamesBondCar+spyoptions");
```

nénk hozzájutni. Ehhez a következő kód szükséges:

osztályba ágyazott felsorolt típus (spyoptions) típusinformációhoz szeret-
hat, amellyel a *beágyazott típust* jelöljük. Tétellezzük fel, hogy a JamesBondCar
A type.GetType() metódusnak átadott sztring pluszjel tokent (+) tartalmaz-
// Típusinformáció megszerzése egy külső szerelevény típusáról.
type t = type.GetType("CarLibrary.Sportscar, CarLibrary");

barátiságos nevével kell megadni (a neveket vessző választja el egymástól):

típus teljesen meghatározott nevével, valamint a típust tartalmazó szerelevény
privát szerelevény metaadathoz szeretnénk hozzáférni, a sztringparamétert a
pen végrehajtás alatt álló szerelevény definiálja a típust. Ha azonban külső
mágában foglalja a típust. Ekkor az az alapértelmezett felteves, hogy az ép-
A GetType() metódusnak átadott sztring nem említi azt a szerelevényt, amely
// Típusinformáció megszerzése a statikus type.GetType() metódus
// használataival (nem dob kivételt, ha nem talál sportscar típust,
// valamint figyelmen kívül hagyja a kis- és nagybetűket).
type t = type.GetType("CarLibrary.Sportscar", false, true);

tus érzékenységet állítja be. Ehhez nézzük meg a következő kódot:

vételt dobjon, ha nem találja a típust, a másik pedig a sztring kis- és nagybe-
két logikai paramétert, amelyek egyike azt szabályozza, hogy a rendszer ki-
A type.GetType() metódus túlterhelése lehetővé teszi, hogy meghatározzunk

Megjegyzés Amikor azt állítjuk, hogy a type.GetType() hívása során nincs szükségünk for-
ditási idejű ismeretekre, arra utalunk, hogy (erősen típusos változó helyett) a metódus bármi-
lyen sztringértéket kaphat. A típus sztringformátumú nevét természetesen ismernünk kell.

tem.string osztály példányát várja paraméterként.

tát vizsgáljuk, hiszen a type.GetType() metódus a mindenütt jelenlevő sys-
tási idejű információkkal rendelkezzünk arról a típusról, amelynek metaada-
jesen meghatározott nevét. Ezzel a módszerrel *nincs* szükség arra, hogy fordí-
tály statikus GetType() tagját hívjuk, és megadjuk a vizsgálni kívánt típus tel-
Egyszerűbben hozzáférhetünk a típusinformációkhoz, ha a system.Type osz-

Típusreferencia megszerzése a System.Type.GetType() segítségével

Típusreferencia készítése a `typeid` segítségével

A típusinformációk megszerzésének utolsó lehetséges módja a `C#` `typeid` operátornak az alkalmazása:

```
// Típus megszerzése a typeid használatával.  
Type t = typeid(sportscar);
```

A `typeid` metódushoz hasonlóan, ha a `typeid` operátorral szeretnénk megkapni a típusinformációkat, nem kell létrehozunk objektum példányt. Az alapkövetelmény azonban fordítási idejű ismeretekkel kell rendelkeznie a vizsgálni kívánt típusról, ugyanis a `typeid` operátornak a szöveges reprezentáció helyett a típus erősen típusított neve kell.

Egyedi metaadat-nézegető létrehozása

A reflexió alapvető folyamatainak (és a `System.Type` osztály lehetőségeinek) bemutatásához hozzuk létre a `MyTypeViewer` parancssori alkalmazást. A program megjeleníti az `mscorlib.dll`-ben (a .NET-alkalmazások automatikusan hozzáférnek ehhez a kulcsfontosságú keretrendszer-osztálykönyvtárhoz), illetve a `MyTypeViewer` alkalmazásban definiált típusok metódusait, tulajdonságait, mezőit és az itt található interfészek (továbbá egyéb érdekességeket) részleteit. Az alkalmazás létrehozása után mindenképpen importáljuk a `System.Reflection` névteret:

```
using System.Reflection;
```

A metódusok reflexiója

A program osztályt számos olyan statikus metódussal bővíti, amelyek közül mindegyik egyetlen `System.Type` paramétert vár, és a visszatérési értéke `void`. Először nézzük meg a `ListMethods()` metódust, amely a bejövő típus által definiált metódusok nevét listázza. A `Type.GetMethods()` a `System.Reflection.MethodInfo` típusok tömbjét adja vissza:

```

    }
    Console.WriteLine();
    Console.WriteLine("<->{0}", prop.Name);
    foreach(PropertyInfo pi in t.GetType().GetProperties())
    {
        Console.WriteLine("***** Properties *****");
        static void ListProps(Type t)
        // Megjeleníti a típus tulajdonságainak a nevét.

```

A típus tulajdonságait megjelenítő logika hasonló:

```

    }
    Console.WriteLine();
    Console.WriteLine("<->{0}", field.Name);
    foreach(FieldInfo fi in t.GetType().GetFields())
    {
        Console.WriteLine("***** Fields *****");
        static void ListFields(Type t)
        // Megjeleníti a típus mezőinek a nevét.

```

zók nevét listázzuk:

A `ListFields()` metódus megvalósítása hasonló a fentiekhez. Az egyetlen figyelemet érdemlő különbség a `Type.GetFields()` metódus hívása, valamint az eredményként kapott `FieldInfo` tömb. Az egyszerűség kedvéért csak a me-

Mezők és tulajdonságok reflexiója

A példában a `MethodInfo.Name` tulajdonság segítségével kiírjuk a metódus nevét. A `MethodInfo` további tagokkal rendelkezik, s ezekkel meghatározhatjuk azt, hogy a metódus statikus, virtuális, generikus vagy absztrakt legyen. A `MethodInfo` típus lehetővé teszi azt is, hogy hozzáférjünk a metódus visszatérési értékéhez és paraméterkészletéhez.

```

    }
    Console.WriteLine();
    Console.WriteLine("<->{0}", m.Name);
    foreach(MethodInfo mi in t.GetType().GetMethods())
    {
        Console.WriteLine("***** Methods *****");
        static void ListMethods(Type t)
        // Megjeleníti a típus metódusainak nevét.

```

A `Program osztály main()` metódusa a típus teljeseen meghatározott nevét kéri a felhasználótól. Mivel ezt a sztringadat a birtokunkban van, átadjuk a `Type.getType()` metódusnak, a kinyert `System.Type` osztályt pedig elküldjük az összes segédmetódusnak. A folyamatot addig ismétli a rendszer, amíg a felhasználó a `q` beírásával leállítja az alkalmazást:

A Main() metódus megvalósítása

```

    }
    Console.WriteLine();
    Console.WriteLine("Is type a class type? {0}", t.IsClass);
    t.IsGenericTypeDefinition();
    Console.WriteLine("Is type generic? {0}",
        Console.WriteLine("Is type sealed? {0}", t.IsSealed);
    Console.WriteLine("Is type abstract? {0}", t.IsAbstract);
    Console.WriteLine("Base class is: {0}", t.BaseType);
    Console.WriteLine("***** Various statistics *****");
}
static void ListVariationStats(Type t)
// Pusztán ráadásként.

```

Végül, de nem utolsósorban, rendelkezésünkre áll egy olyan segédmetódus, amely a bejövő típusról különböző statisztikákat jelenít meg (ezek azt jelzik, hogy a típus generikus-e, lezár-e, mi a típus alaposztálya, és így tovább):

Egyéb érdekességek megjelenítése

```

    }
    Console.WriteLine("<->{0}", t.Name);
    foreach (Type i in t.Faces)
        Type[] ifaces = t.GetInterfaces();
    Console.WriteLine("***** Interfaces *****");
}
static void ListInterfaces(Type t)
// A megvalósított interfészek megjelenítése.

```

A következő példában elkészítjük a `ListInterfaces()` metódust, amely a bejövő típus által támogatott interfészek nevét listázza. Az egyetlen igazán érdekes pont az, hogy a `GetInterfaces()` a `System.Type` osztályok tömbjét adja vissza. Ez azért nem meglepő, mert az interfészek valójában típusok:

A típus által megvalósított interfészek reflexiója

- System.Void
- System.Threading.Thread
- System.Collections.ArrayList
- System.Int32

tározása érdekében a kis- és a nagybetűre):

A következőkben a MyTypeVtewer.exe alkalmazást teszteljük. Futtassuk az al-kalmazást, majd írjuk be a következőket teljesen megadott neveket (a type.gettype() metódus hívását úgy paramétereztük, hogy a sztringnevek megha-

```
static void Main(string[] args)
{
    Console.WriteLine("***** Welcome to MyTypeVtewer *****");
    string typeName = "";
    bool userIsDone = false;
    do
    {
        Console.WriteLine("\nEnter a type name to evaluate");
        Console.WriteLine("or enter q to quit: ");
        // A típus nevének megszerzése.
        typeName = Console.ReadLine();
        // A felhasználó ki akar lépni?
        if (typeName.ToUpper() == "q")
        {
            userIsDone = true;
            break;
        }
        // Próbáljuk meg megjelölni a típust.
        try
        {
            Type t = Type.GetType(typeName);
            Console.WriteLine("");
            ListVariousStats(t);
            ListFields(t);
            ListProps(t);
            ListMethods(t);
            ListInterfaces(t);
        }
        catch
        {
            Console.WriteLine("Sorry, can't find type");
        }
        while (userIsDone);
    }
}
```

System.Collections.Generic.Dictionary`2

numerikus érték 2:

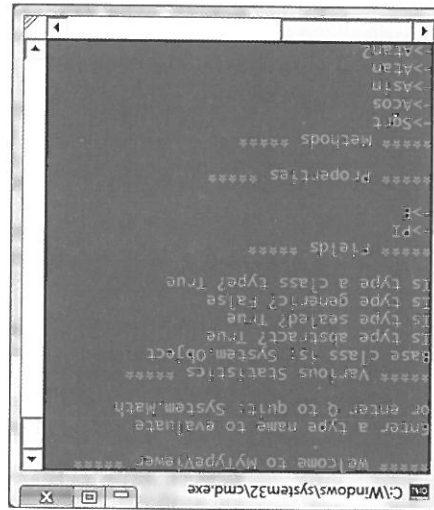
A numerikus érték az 1 lesz, hiszen a List<T> egyetlen típusparaméterrel rendelkezik. Ha azonban a Dictionary<Tkey, TValue> osztállyal dolgozunk, a

System.Collections.Generic.List`1

Amikor generikus típusok metaadat-leírásának megszerzéséhez hívjuk a Type.GetMethod() metódust, speciális szintaxist kell alkalmaznunk a „visszafele aposztrofál” (’), amelyet egy numerikus érték követ. A numerikus érték jelzi, hogy a típus hány típusparamétert támogat. Ha például a List<T> metaadat leírását szeretnénk kiírni, a következő sztringet kell beírunk az alkalmazásunkba:

Generikus típusok reflexiója

16.2. ábra: A System.Math reflexiója



A 16.2. ábrán a System.Math osztály részleges kimenete látható.

- System.IO.BinaryWriter
- System.Math
- System.Console
- MyTypeViewer.Program

Metódusparaméterek és visszatérési értékek reflexiója

Módosítsuk a `ListMethods()` segédfüggvényt, hogy az ne csak a meghatározott metódus nevét listázza, de a visszatérési értéket és a bejövő paramétereket is. Ezt a metódus típusa `returnType` tulajdonságával és `getParameters()` metódusával oldhatjuk meg. Az alábbi forráskódban begyűjtöttük az összes paraméter típusát és nevét tartalmazza:

```
static void ListMethods(Type t)
{
    Console.WriteLine("***** Methods *****");
    MethodInfo[] mi = t.GetMethods();
    foreach (MethodInfo m in mi)
    {
        // A visszatérési érték megszerzése.
        string retVal = m.ReturnType.FullName;
        string paramInfo = "(";
        // A paraméterek megszerzése.
        foreach (ParameterInfo pi in m.GetParameters())
        {
            paramInfo += string.Format("{0} {1} ", pi.ParameterType,
                pi.Name);
        }
        paramInfo += " )";
        // Most jelenítsük meg az egyszerű metódus-aláírást.
        Console.WriteLine("<-> {0} {1} {2}", retVal, m.Name, paramInfo);
    }
    Console.WriteLine();
}
```

Há a módosított alkalmazást futtatjuk, azt tapasztaljuk, hogy egy adott típus metódusai sokkal több információt hordoznak. A 16.3. ábrán a `system.globalization.GregorianCalendar` típus metódusainak metaadatai láthatók. A `ListMethods()` fenti megvalósítása azért hasznos, mert a `system.Reflection` objektummodell segítségével közvetlenül megvizsgálhatjuk a paramétereket és a metódusok visszatérési értékét. Az `xxxInfo` típusok mindegyike (`MethodInfo`, `PropertyInfo`, `EventInfo` stb.) felülbírálja a `ToString()` metódust a kért elem aláírásának a megjelenítéséhez. A `ListMethods()` metódust tehát megvalósíthatjuk a következőképpen:

Dinamikusan betöltődő szerezvények

16. fejezet: Típusreflexió, késői kötés és attribútumalapú programozás

A CLR a szerezvény manifestumához fordul, amikor külső hivatkozott szerezvényt keres (lásd a 15. fejezetben). Sokszor előfordulhat, hogy programozottan, menet közben kell betöltenünk szerezvényeket, még akkor is, ha az említett szerezvényről a manifestum említést sem tesz. Szakmai szempontból a külső szerezvények igény szerinti betöltésének folyamatát *dinamikus betöltésnek* nevezzük.

A `System.Reflection` névtér definiálja az `Assembly` osztályt. A típus segítségével dinamikusan tölthetünk be egy szerezvényt, valamint ismerhetjük meg annak tulajdonságait. Az `Assembly` típussal dinamikusan betölthetünk privát és megosztott szerezvényeket, valamint tetszőleges helyen tárolt szerezvényt is. Az `Assembly` osztály lenyegében metódusokat (nevezetesen a `load()` és a `loadfrom()` metódusokat) biztosít, amelyekkel programozottan beszerezhetjük az ügyféloldali konfigurációs fájlban található információkat. A dinamikus betöltés bementetéséhez készítjük el az újabb paramcsort al-kalmazásunkat, az `ExternalAssemblyReflector`. A feladatunk az, hogy olyan `main()` metódust készítsünk, amely a dinamikusan betölteni kívánt szerezvény barátságos nevét kérdezi meg. Az `Assembly` referenciát a `displayTypes()` segédmetódusnak adjuk át, amely a benne foglalt osztályok, interfészek, struktúrák, felsorolások és metódusreferenciák nevét listázza. A forráskód meglehetősen egyszerű:

```
using System;
using System.Reflection;
using System.IO; // A FileNotFoundException definíciójához.

namespace ExternalAssemblyReflector
{
    class Program
    {
        static void DisplayTypesInAsm(Assembly asm)
        {
            Console.WriteLine("\n***** Types in Assembly *****");
            Console.WriteLine("<{0}"", asm.FullName);
            Console.WriteLine(">{0}"", asm.GetType());
            foreach (Type t in types)
            {
                Console.WriteLine("{0}"", t);
            }
        }
    }
}
```



```

static void Main(string[] args)
{
    Console.WriteLine("***** External Assembly Viewer *****");
    string asmName = "";
    bool userIsDone = false;
    Assembly asm = null;

    do
    {
        Console.WriteLine("\nEnter an assembly to evaluate");
        Console.WriteLine("or enter q to quit: ");

        // A típus nevének megszerzése.
        asmName = Console.ReadLine();

        // A felhasználó ki akar lépni?
        if (asmName.ToUpper() == "q")
        {
            userIsDone = true;
            break;
        }

        // Próbáljuk meg betölteni a szerezvényt.
        try
        {
            asm = Assembly.Load(asmName);
            DisplayTypeInAsm(asm);
        }
        catch
        {
            Console.WriteLine("Sorry, can't find assembly.");
        }
        while (!userIsDone);
    }
}

```

A statikus Assembly.Load() módszernek kizárólag a memóriába betölteni kívánt szerelvény barátságos nevét adjuk át. Ha tehát a carLibrary.d11 szerelvényen reflexiót szeretnénk alkalmazni, a carLibrary.d11 bináris fájl a program futtatásához az externalAssemblyReflector alkalmazás \bin\Debug könyvtárba kell másolnunk. A másolás után a 16.4. ábrán látható kimenet-hez hasonló eredményt kapunk.

```
// A CarLibrary 1.0.982.23972 verziójának betöltése
// az alapértelmezett kultúra használatával.
Assembly a = Assembly.Load(
    @"CarLibrary, Version=1.0.982.23972, PublicKeyToken=null,
    Culture=null");
```

A megjelenített név összehasonlításakor a PublicKeyToken = null konvenció jelzi, hogy erős nével nem rendelkező szerelvényhez történő kötés és egyeztetés szükséges. Továbbá a Culture = "" érték mutatja, hogy a rendszernek a cél-számtípus alapértelmezett kultúraazonosítójával kell egyeztetnie:

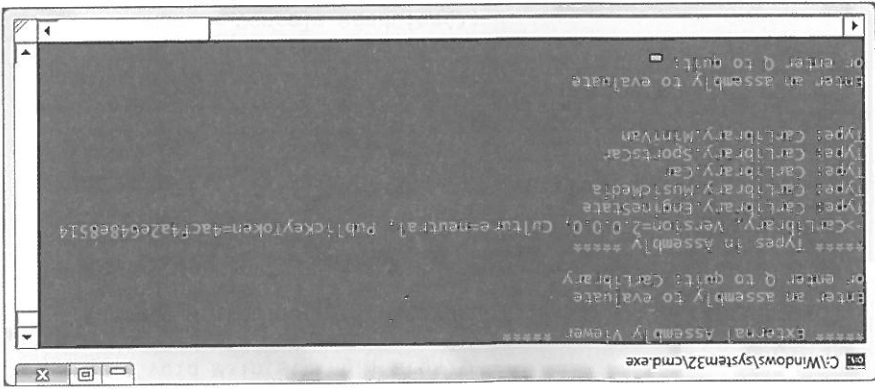
```
Name(,version = major.minor.build.revision)
(,culture = culture token) (,PublicKeyToken = public key token)
```

Az Assembly.Load() többszörösen túlterhelt módszer. Az egyik változat lehetővé teszi kultúraazonosító (lokalizált szerelvények esetén), valamint verziószám és nyilvános kulcs értékeknek (megosztott szerelvények számára) a meghatározását. A szerelvényt azonosító elemek gyűjteménye a `/// megjelenítési név`. A megjelenített név formátuma a név/érték páros vesszővel elválasztott sztringje. A sztring a szerelvény barátságos nevével kezdődik, amelyet (tetszőleges sorrendben) opcionális szerelvényminősítők követnek. A következő sablon a következő (az opcionális minősítők zárójelben jelennek meg):

Megosztott szerelvények

Forráskód Az ExternalAssemblyReflector projekt megtalálható a 16. fejezet alkönyvtárában.

16.4. ábra: A külső CarLibrary szerelvény reflexiója



16. fejezet: Típusreflexió, késői kötés és attribútumalapú programozás

A `System.Reflection` névter által biztosított `AssemblyName` típus lehetővé teszi, hogy az előbb összeállított sztringinformációt egy jól kezelhető objektumváltó-zóval ábrázoljuk. Ezt az osztályt rendszerint a `System.Version` osztállyal együtt alkalmazzuk, amely a szerelvény verziószámának OO-burkolója. A megjelített név létrehozása után ezt átadhatjuk a `Assembly.Load()` módszernek:

```
// Használjuk az AssemblyName típust a megjelített név
// definiálásához.
AssemblyName asmName;
asmName = new AssemblyName();
asmName.Name = "CarLibrary";
Version v = new Version("1.0.982.23972");
asmName.Version = v;
Assembly a = Assembly.Load(asmName);
```

Ha a globális szerelvénytárból (GAC) szeretnénk betölteni egy megosztott szerelvényt, az `Assembly.Load()` paraméternek meg kell határoznia a `PublicKeyToken` értékét. Tételizzük fel például, hogy a .NET-alapoztálykönyvtárak `System.Windows.Forms.dll` szerelvényének 2.0.0.0 verzióját töltjük be. Mivel a szerelvény rengeteg típust foglal magában, a következő alkalmazás, egyszerű LINQ-lekérdezés segítségével, csak a nyilvános felsorolások nevét listázza:

```
using System;
using System.Reflection;
using System.IO;
using System.Linq;

namespace SharedAssemblyReflector
{
    public class SharedAssemblyReflector
    {
        private static void DisplayInfo(Assembly a)
        {
            Console.WriteLine("***** Info about Assembly *****");
            Console.WriteLine("Loaded from GAC? {0}",
                a.GlobalAssemblyCache);
            Console.WriteLine("Assembly Name: {0}", a.GetName().Name);
            Console.WriteLine("Assembly Version: {0}", a.GetName().Version);
            Console.WriteLine("Assembly Culture: {0}",
                a.GetName().CultureInfo.DisplayName);
            Console.WriteLine("\nHere are the public enums:");
            // Használjunk LINQ-lekérdezést a nyilvános felsorolások
            // listázásához.
            Type[] types = a.GetTypes();
            var publicEnums = from pe in types where pe.IsEnum &&
                pe.IsPublic select pe;
```

A *kései kötés* olyan technika, amelynek segítségével futásidőben létrehozhatjuk egy adott típus egy példányát, és meghívhatjuk a típus tagjait anélkül, hogy az alkalmazásba kódolt fordítási idejű ismeretekkel rendelkezzenek a fejlesztésről. Ha az alkalmazás, amelyet készítünk, késői kötetet használ egy külső szerelvény típusán, nem kell a szerelvényre hivatkoznunk; így a hívó felületre nem listázza a szerelvényt.

A késői kötés

Megismerkedtünk tehát a `system.reflection` névtér alaptypusaival a metaadatok futásidőjű vizsgálatához. Annak ellenére, hogy rendkívül hasznos lehetőség, munkánk során valószínűleg nem kell egyedi objektumbőngészőket készítenünk. A reflexió szolgálatások képezik azonban több általános programozási tevékenység, például a *késői kötés* alapját.

Forráskód A `SharedAssemblyReflector` projekt megtalálható a 16. fejezet alkönyvtárában.

```
foreach (var pe in publicenums)
{
    console.WriteLine(pe);
}

static void Main(string[] args)
{
    console.WriteLine("***** The Shared Asm Reflector App *****\n");

    // Töltsük be a System.Windows.Forms.dll szerelvényt
    // a GAC-ból.
    string displayName = null;
    displayName = "System.Windows.Forms, " +
        "version=2.0.0.0, " +
        "PublicKeyToken=b77a5c561934e089, " +
        @"Culture=""""";
    Assembly asm = Assembly.Load(displayName);
    displayNameInfo(asm);
    console.WriteLine("Done!");
    console.ReadLine();
}
```

Mi a késői kötés használata? Igaz, hogy ha alkalmaznánk a típus korai kötését (pl. beállítunk szerelevényreferenciát, és a C# new kulcsszavának segítségével lefoglaljuk a típust), akkor érdemes valóban használni. Egyrészt a késői kötés lehetővé teszi, hogy a hibákat a futásidő helyett már a fordítás során kiszűrjük. Másrészt kritikus szereppel rendelkezik a kiterjeszthető alkalmazások készítésében. A fejezet későbbi részében alkalmunk nyílik kiterjeszthető program létrehozására; addig is azonban vizsgáljuk meg az aktivátor típus szerepét.

A System.Activator osztály

A System.Activator osztály a .NET késői kötési folyamatainak a kulcsa. A system.object osztálytól örökölt metódusokon túl az Activator a tagok olyan ki-sebb gyűjteményét definiálja, amelyek többsége a .NET-remoteing API-hoz kapcsolódik. Ezek közül az Activator.CreateInstance() metódus segítségével-vel, késői kötéssel létrehozhatjuk a típus egy példányát. A metódus többszörösen túlterhelt, így rendkívül rugalmas funkcionali-tást biztosít. A CreateInstance() tag leggyeszerűbb változata érvényes type objektummal dolgozik, amely a menetközben lefoglalni kívánt entitást írja le. Hozzuk létre a LateBindingApp parancssori alkalmazást, és a következőkép-pen módosítsuk a main() metódust (ne felejtsük el a CarLibrary.dll másola-tát a projekt \bin\Debug könyvtárába helyezni):

```
// Típus dinamikus létrehozása.
public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Late Binding *****");
        // Próbáljuk meg betölteni a CarLibrary helyi másolatát.
        Assembly a = null;
        try
        {
            a = Assembly.Load("CarLibrary");
        }
        catch (FileNotFoundException e)
        {
            Console.WriteLine(e.Message);
        }
        return;
    }
}

// A Minivan típus metaadatainak kiolvasása.
Type minivan = a.GetType("CarLibrary.Minivan");
```

```
// A minivan létrehozása menet közben.
object obj = Activator.CreateInstance(minivan);
Console.WriteLine("Created a {0} using late binding!", obj);
}
}
```

Az `Activator.CreateInstance()` metódus az erősen típusos `minivan` helyett a `system.object` típust adja vissza. Ezért, ha az `obj` változóra a pont operátort alkalmazzuk, nem látjuk a `minivan` típus egyetlen tagját sem. A problémát explicit kasszt segítségével orvosolhatjuk; ennek a programnak azonban főgalma sincs arról, hogy a `minivan` egyáltalán micsoda (ha megoldhatnánk a problémát, mi szükségünk lenne a késői kötésre?).

A késői kötés lényege, hogy a fordítási időben a rendszer számára ismeretlen objektumok példányait tudjuk létrehozni. Ennek fényében felmerül a kérdés: Hogyan hívhatjuk meg a `system.object` változóban tárolt `minivan` objektum mögöttes metódusait? Természetesen reflexió segítségével.

Paraméter nélküli metódusok hívása

Tételezzük fel, hogy a `minivan` típus `turboboost()` metódusát szeretnénk meghívni. Ez a metódus a motor állapotát „tönkrement” értékre állítja, és tájékoztatást üzenetdobozt jelenít meg. Az első lépés, hogy a `type.getmethod()` segítségével kinyerjük a `turboboost()` metódushoz tartozó metódusinfót. Az eredményként kapott metódusinfó példányon keresztül meghívhatjuk a `minivan` példány `turboboost` tagját. A `methodinfo.invoke()` metódus számára az összes olyan paramétert meg kell határoznunk, amelyeket a metódusinfó által képviselt metódusnak átadunk. A paramétereket a `system.object` típusok tömbje képviseli (ugyanis a metódus paraméterei között több különböző entitás szerepelhet). Mivel a `turboboost()` nem igényel paramétereket, null értéket adhatunk át (ez jelzi, hogy a metódus nem rendelkezik paraméterekkel). Módosítsuk `main()` metódusunkat a következőképpen:

```
static void main(string[] args)
{
    ...
    // A minivan típus metaadatainak kiolvasása.
    type minivan = a.GetType("CarLibrary.minivan");
    // A minivan létrehozása menet közben.
    object obj = Activator.CreateInstance(minivan);
    Console.WriteLine("Created a {0} using late binding!", obj);
}
```

```
// Információk megszerzése a TurboBoost módszerrel.
MethodInfo mi = minivan.GetType().GetMethod("TurboBoost");

// A módszer meghívása ('null' akkor, ha nincs paraméter).
mi.Invoke(obj, null);
Console.ReadLine();
}
```

Ekkor a 16.5. ábrán látható üzenetdoboz tartalma a következő lesz:



16.5. ábra: Módozustítás késői kötéssel

Paraméterekkel rendelkező metódusok hívása

Nézünk meg, hogyan lehet dinamikusan hívni paraméterekkel rendelkező metódust. Tétélezzük fel, hogy módosítottuk az előzőekben elkészített minivan típust, és a `TellChitToBequiet()` metódussal bővítettük:

```
// A csapat lecsendesítése...
public void TellChitToBequiet(string kidName, int shameIntensity)
{
    for(int i = 0 ; i < shameIntensity; i++)
        MessageBox.Show(string.Format("Be quiet {0} !!", kidName));
}
```

A `TellChitToBequiet()` metódusnak két paramétere van: `string` tárolja a gyermek nevét, és egész szám képviseli az érzelem pillanatnyi mértékét. Ha késői kötést alkalmazunk, a bemenő paraméterek `system.object` tömbben adhatók át. Az új metódus hívásához a következőképpen módosítsuk a `main()` metódust:

```
static void main(string[] args)
{
    ...
    // A minivan típus metaadatainak kiolvasása.
    Type minivan = a.GetType("CarLibrary.Minivan");
}
```

```
// A MiniVan létrehozása menet közben.
object obj = Activator.CreateInstance(miniVan);
console.WriteLine("Created a {0} using late binding!", obj);

// késői kötés paraméterekkel rendelkező metódushoz.
object[] paramArray = new object[2];
paramArray[0] = "Fred"; // Gyermek neve.
paramArray[1] = 4; // Erzelem mértéke.
mi = miniVan.GetMethod("Te11Ch11dToBeQu1et");
mi.Invoke(obj, paramArray);
console.ReadLine();
}
```

Ha futtatjuk a programot, négy üzenetdoboz jelenik meg az ifjú Fred megszégyenítésére. A reflexió, a dinamikus betöltés és a késői kötés közötti kapcsolatot már megvillagítottuk, ám még kétséges lehet, hogy az alkalmazásunk készítése során pontosan *mi*kor vehetjük hasznát ezeknek a módszereknek. A választ a fejezet összességében adjuk meg. A következőkben a .NET-attribútumok szerepét vizsgáljuk.

Forráskód A LateBindingApp projekt megtalálható a 16. fejezet alkönyvtárában.

Programozás attribútumokkal

A .NET-fordító egyik szerepe az, hogy elkészítse a definiált és hivatkozott típusok metaadat-leírását. A szerelvényben tárolt standard metaadatokon kívül a .NET platform lehetőséget biztosít a fejlesztőknek arra, hogy *attribútumok* segítségével további metaadatokat ágyazzanak a szerelvénybe. Röviden: az attribútumok forráskódjegyzetek, amelyeket egy adott típusra (osztály, interfész, struktúra stb.), tagra (tulajdonság, metódus stb.), szerelvényre vagy modorra alkalmazhatunk.

Az attribútumok által jegyzetekkel ellátott forráskód ötlete nem újdonság. A COM IDL több olyan előre definiált attribútumot biztosított, amelyek segítségével a fejlesztők leírhatták az adott COM-kiszolgálón található típusokat. A COM-attribútumok azonban alig jelentettek többet kulcsszavak gyűjteményénél. Ha a COM-fejlesztőnek egyedi attribútumot kellett létrehoznia, ezt ugyan nyugodtan megtehetette, de a rendszer a forráskódban az attribútumra csak egy 128 bites számmal (GUID) hivatkozott, és ez nagyon kényelmetlen megoldás.

A COM IDL attribútumoktól eltérően (amelyek pusztán kulcsszavakat jele-
lentettek) a .NET-attribútumok osztálytípusok, amelyek a `System.Attribute`
összestályból származnak. A .NET-névterek felfedezése során jó néhány olyan
előre definiált attribútummal találkozhatunk, amelyeket az alkalmazásaink-
ban hasznosítani tudunk. Továbbá módunkban áll egyedi attribútumokat is
készíteni, hogy tovább pontosítsuk a típusaink viselkedését az `Attribute` osz-
tályból származó új típusok létrehozásával.

Amikor a forráskódunkban attribútumokat alkalmazunk, a beágyazott
metaadat egészen addig hasznavehetően, amíg egy másik kódreszt ex-
plicit módon alkalmazott reflexióval ki nem nyeri azt. Ennek hiányában a sze-
relvénybe ágyazott metaadatról senki sem szerez tudomást, így annak sem-
milyen hatása nincs.

Attribútumok feldolgozása

A .NET Framework 3.5 SDK több olyan segédprogramot tartalmaz, amelyek a
különböző .NET-attribútumok alapján dolgoznak. A fordítási ciklus során ma-
ga a C#-fordító (`csc.exe`) is észleli a különféle attribútumok jelenlétét. Ha pél-
dául a C#-fordító a `[CLSCompliant]` attribútummal találkozik, automatikusan
ellenőrzi azt az elemet, amelyen az attribútumot alkalmazzuk, és megbizonyo-
sodik róla, hogy az kizárólag CLS-kompatibilis szerkezetekkel dolgozik. Egy
másik esetben, ha a C#-fordító `[Obsolete]` attribútummal ellátott elemet talál, a
Visual Studio 2008 hibajelzőjelekében fordítási figyelmeztetést jelent meg.

A fejlesztői eszközökön kívül a .NET-alapoztálykönyvtár számos metó-
dusa is jellegzetes attribútumokat figyel. Ha például fájlban szeretnénk rögzí-
teni az objektum állapotát, akkor egyetlen tennivalónk az, hogy az osztá-
lyunkat ellátjuk a `[Serializable]` attribútummal. Ha a `BinaryFormatter` osz-
tály `Serialize()` metódusa ezzel az attribútummal találkozik, az objektum
mentése automatikusan egy tömör binárisformátumban történik.

A .NET CLR attribútuma ugyancsak bizonyos attribútumok jelenlétét ku-
taja. A legismertebb .NET-attribútum talán a `WebMethod`. Ha szeretnénk
HTTP-kérésekkel hozzáférhetővé tenni a metódust és a függvény visszatérési
értéket automatikusan XML-formátumban kódolni, alkalmazzuk a `WebMethod`
attribútumot, és a CLR minden egyéb részletről gondoskodik. A webszojgálta-
s fejlesztésén túl az attribútumok a .NET biztonsági rendszer, a Windows
Communication Foundation és a COM/.NET együttműködési képesség szá-
mára is egyaránt szükségesek.

Bátran készíthetünk olyan alkalmazásokat, amelyekben saját egyedi attribútumainkon, illetve a .NET-alapoztálykönyvtárak tetszőleges attribútumainak alkalmazunk reflexiót. Attribútumok készítésével gyakorlatilag szerelevé-nyek számára érthető kulcsszavakat hozhatunk létre.

Attribútumok alkalmazása a C# nyelvben

A .NET-alapoztálykönyvtár különböző névterekben több attribútumot biztosít. A 16.3. táblázatban néhány (de *nem* az összes) előre definiált attribútumot mutatunk.

Attribútum	Valós jelentése
[CLSCompliant]	Kényszeríti a jeggyzettel ellátott elemet, hogy megfeleljen a CLS szabályainak. A CLS-kompatibilis típusokat zökkenőmentesen használhatjuk bármely .NET programozási nyelvben.
[DllImport]	Lehetővé teszi a .NET-kód számára, hogy bármilyen nem felügyelt C- vagy C++-alapú kódkönyvtárat meghívjon – beleértve a mögöttes operációs rendszer API-ját is. A [DllImport] nem használatos COM-alapú programokkal folytatott kommunikációban.
[Obsolete]	Elavult típust vagy tagot jelöl. Ha a programozó ilyen elemet próbál használni, a fordító figyelmeztetést küld, amelyben a probléma leírása található.
[Serializable]	„Sorosítható” osztályt vagy szerkezetet jelöl. Ez annyit jelent, hogy az elem aktuális állapot adataiolyanban rögzíthető.
[NonSerialized]	Meghatározza, hogy az osztály vagy szerkezet adott mezőjét a sorosítási folyamat ne tárolja.
[WebMethod]	Jelzi, hogy a metódust hívhatják HTTP-kérések, és utasítja a CLR-t, hogy a függvény visszatérési értékét XML-formátumba sorosítsa.

16.3. táblázat: Néhány előre definiált attribútum

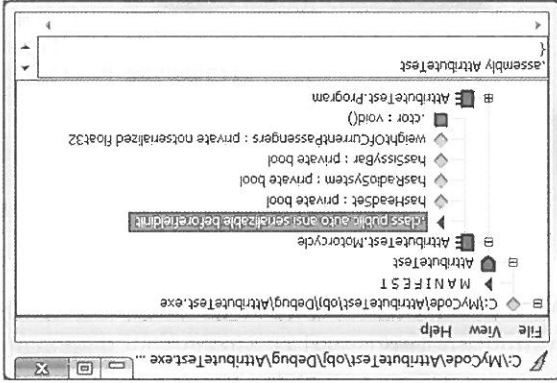
Az attribútumok alkalmazásának bemutatásához tételizzük fel, hogy szeretnénk elkészíteni a bináris formátumban tárolható `Motorcycle` osztályt. Ennek érdekében alkalmazzuk az osztálydefinição a `[Serializable]` attribútumot. Ha olyan mezővel dolgozunk, amelyet nem szeretnénk tárolni, alkalmazzuk a `[NonSerialized]` attribútumot:

```
// Ez az osztály elmenthető.
[Serializable]
public class Motorcycle
{
    // Ez a mező azonban nem marad meg.
    [NonSerialized]
    float weightOfCurrentPassengers;

    // Ezek a mezők még sorosíthatók.
    bool hasRadiatorSystem;
    bool hasHeadset;
    bool hasSissyBar;
}
```

Megjegyzés Az attribútum kizárólag a „közvetlenül mellette álló” elemre érvényes. A `Motorcycle` osztály egyetlen nem sorosított mezője a `weightOfCurrentPassengers` mező. A többi mező sorosítható, hiszen a teljes osztályt megjelöltük a `[Serializable]` attribútummal.

Ha attribútumot szeretnénk alkalmazni, az attribútum nevét szögletes zárójelbe kell foglalnunk. Az osztály fordítását követően az `ildasm.exe` segítségével megnézhetjük az újabb metaadatokat. A fenti attribútumokat a `serializable` és a `notserializable` tokenek rögzítik (lásd 16.6. ábrát).



16.6. ábra: `AttributeTest` az `ildasm.exe` segédprogramban

Egyetlen elemhez több attribútumot társíthatunk. Tétélezzük fel, hogy rendelkezünk egy sorosíthatóként megjelölt, örökölt C#-osztálytípussal (horse-buggy), amely az aktuális fejlesztés szempontjából már elavultnak számít. Ahelyett, hogy az alapködből töröljünk az osztály definícióját (és esetleg tönkretennénk a programot), megjelölhetjük az osztályt az [obsolete] attribútummal. Ha egyetlen elemen több attribútumot szeretnénk alkalmazni, állítsuk össze az attribútumok vesszővel elválasztott listáját:

```
[Serializable, obsolete("use another vehicle")]
public class HorseAndBuggy
{
    // ...
}
```

Másik megoldás az, ha úgy alkalmazzunk több attribútumot egyetlen elemen, hogy az egyes attribútumokat egymásra halmozzuk (a végeredmény ugyanaz):

```
[Serializable
[obsolete("use another vehicle")]
public class HorseAndBuggy
{
    // ...
}
```

Attribútumok konstruktorparamétereinek meghatározása

Figyeljük meg, hogy az [obsolete] olyan paramétereket is fogad, amelyek leginkább a konstruktorok bemenő paramétereire hasonlítanak. Ha a Visual Studio 2008 Code Definition ablakában megnézzük az [obsolete] attribútum formális definícióját, láthatjuk, hogy az osztály valóban rendelkezik egy olyan konstruktorral, amely a system.string típust kapja:

```
public sealed class ObsoleteAttribute : System.Attribute
{
    public bool IsError { get; }
    public string Message { get; }
    public ObsoleteAttribute(string message, bool error);
    public ObsoleteAttribute();
}
```

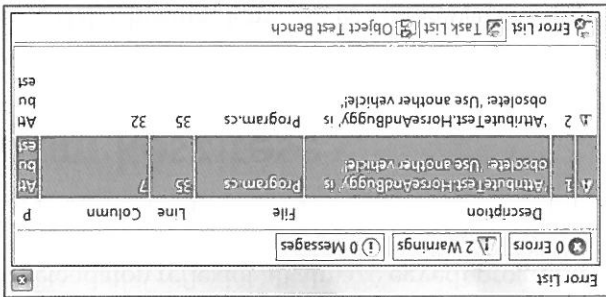
Ha konstruktorparaméterekkel látjuk el az attribútumot, a rendszer egészen addig *nem* foglalja le az attribútumot a memóriában, amíg a paramétereken más típus vagy külső eszköz nem alkalmaz reflexiót. Az attribútumszinten definiált sztringadatokat a szereplvény metaadat formájában tárolja.

Az Obsolete attribútum működés közben

Ha elavultként megjelöltük a HorseAndBuggy típust, és szeretnénk lefoglalni a típus egy példányát:

```
static void main(string[] args)
{
    HorseAndBuggy muLe = new HorseAndBuggy();
}
```

láthatnánk, hogy „az obsolete attribútum paramétereként megadott” sztring-adat a Visual Studio 2008 hibalistáblakában jelenik meg (lásd a 16.7. ábrát).



16.7. ábra: Attribútumok működés közben

Jelen esetben az a szoftver, amely reflexiót alkalmaz, maga a C#-fordító.

C#-attribútum rövidített formája

Az [obsolete] attribútum tényleges neve obsoleteAttribute, nem pedig obsolete. Névadási konvencióként minden .NET-attribútum (a saját készítésű egyedi attribútumokat is beleértve) utótagja az Attribute token. Az attribútumok alkalmazásának egyszerűsítésére a C# nyelvben nem szükséges beírunk az Attribute utótagot. Ennek megfelelően a HorseAndBuggy típus követ-kező verziója megegyezik az előzővel (csak kicsit többet kell gépelnünk):

```
[SerializedAttribute]
[ObsoleteAttribute("Use another vehicle")]
public class HorseAndBuggy
{
    // ...
}
```

Ez a lehetőség a C# révén áll a rendelkezésünkre. A .NET-kompatibilis nyelvek közül csak néhány támogatja ezt az attribútumgyorsazonosító-szintaxist. A .NET-attribútumokkal kapcsolatos fontos tények:

- Az attribútumok a `System.Attribute` származnak.
- Az attribútumok beágyazott metadatot generálnak.
- Az attribútumok addig használhatóak, amíg egy másik program nem alkalmaz reflexiót rajtuk.
- Az attribútumokat a C# nyelvben szögletes zárójelek között használjuk.

A következőkben vizsgáljuk meg, hogyan készíthetjük el saját egyedi attribútumainkat és a beágyazott metadaton reflexiót alkalmazó egyedi programot.

Egyedi attribútum készítése

Az egyedi attribútumok készítésének első lépése az, hogy a `System.Attribute` osztályból új származtatott osztályt hozunk létre. Az eddigi gépjárműpéldánál maradva tegyük fel, hogy létrehoztuk az `AttributedCarLibrary` új C#-osztálykönyvtárat. A szerelvény több járművet definiál (amelyek közül néhányat már találkoztunk), és ezek mindegyikét a `VehicleDescriptionAttribute` egyedi attribútum írja le:

```
// Egy egyedi attribútum.
public sealed class VehicleDescriptionAttribute : System.Attribute
{
    private string msgData;

    public VehicleDescriptionAttribute(string description)
    { msgData = description; }

    public VehicleDescriptionAttribute() {}
}
```

```
public string Description
{
    get { return msgData; }
    set { msgData = value; }
}
```

A `VehicleDescriptionAttribute` belül egy privát sztringet tárol, amelyet a `ti-`pus konstruktorán keresztül állíthatunk be, és a típus tulajdonságán keresztül módosíthatunk. Azon túl, hogy az osztály a `System.Attribute` osztályból származik, az osztály definíciója semmilyen különlegességet nem tartalmaz.

Megjegyzés Biztonsági okokból a .NET gyakorlati ajánlata az egyedi attribútumokat lezárt attribútumként létrehozni. A Visual Studio egy `Attribute` nevű kódsnippetet biztosít, amely egy `SystemAttribute`-ből származó kész osztályt generál a forráskódunkba. A kódrészletek alkalmazásának magyarázatát lásd a 2. fejezetben.

Egyedi attribútumok alkalmazása

Mivel a `VehicleDescriptionAttribute` a `System.Attribute` osztályból származik, tesztés szerinti jegyzetekkel láthatjuk el a járműveinket. Tesztelésre adjuk a következő osztálydefiníciókat az új osztálykönyvtárunkhoz:

```
// Leírás hozzáadása "nevesített tulajdonság" használatával.
[Serialize]
[VehicleDescription(description = "My rocking Harley")]
public class Motorcycle
{
}

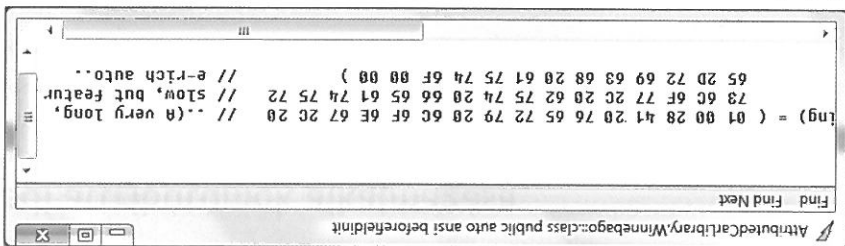
[SerializeAttribute]
[ObsoleteAttribute("Use another vehicle")]
[VehicleDescription("The old gray mare,
she ain't what she used to be...")]
public class HorseAndBuggy
{
}

[VehicleDescription("A very long, slow, but feature-rich auto")]
public class Winnebago
{
}
```

Nevesített tulajdonság szintaxisa

A `Motorcycle` `description` tulajdonságát az új attribútum-központú szintaxis segítségével adtuk meg: ezt nevezzük *nevesített tulajdonságnak*. Az első `[vehicle.description]` attribútum konstruktorában a név/érték páros segítségével beállítjuk a mögöttes `system.string` osztályt. Ha külső beavatkozással reflexiót alkalmazunk az attribútumon, a rendszer a `description` tulajdonságnak adja az értéket (a nevesített tulajdonság szintaxisa kizárólag akkor megengedett, ha az attribútum írható .NET-tulajdonsággal rendelkezik).

Ezzel szemben a `HorseAndBuggy` és a `WineBago` típusok nem alkalmazzák a nevesített tulajdonság szintaxisát, egyszerűen egyedi konstruktorral adják át a stringadatokat. Az `AttributedCarLibrary` szerelvény fordítását követően az `Idasm.exe` segítségével megtekinthetjük a típusban elhelyezett újabb metaadat-leírásokat. A 16.8. ábrán például a `WineBago` típus beágyazott leírását láthatjuk.



16.8. ábra: Beágyazott járműleíró adatok

Az attribútumok használatának korlátozása

Az alapértelmezés szerint az egyedi attribútumokat a forráskód bármely elemére (metódusokra, osztályokra, tulajdonságokra stb.) alkalmazhatjuk. Ha tehát van értelme, akkor a `vehicle.description` attribútummal minősíthetjük a metódusokat, a tulajdonságokat vagy a mezőket (és egyéb elemeket):

```
[vehicle.description("A very long, slow, but feature-rich auto")]
public class WineBago
{
    [vehicle.description("My rocking CD player")]
    public void PlayMusic(bool on)
    {
        ...
    }
}
```


Néhányszor pontosan erre a viselkedésre van szükség. Mások azonban előfordulhat, hogy olyan egyedi attribútumot kell készítenünk, amelyet csak bizonyos kódelemeken alkalmazhatunk. Ha szeretnénk korlátozni az egyedi attribútum hatókörét, az egyedi attribútum definícióján az [AttributeUsage] attribútumot kell alkalmaznunk. Az [AttributeUsage] lehetővé teszi, hogy az AttributeTargets felsorolásból (or operátor segítségével) az értékek tetszőleges kombinációját határozzuk meg:

```
// Ez a felsorolás definiálja egy tulajdonság lehetséges céljait.
public enum AttributeTargets
{
    All, Assembly, Class, Constructor,
    Delegate, Enum, Event, Field,
    Interface, Method, Module, Parameter,
    Property, ReturnValue, Struct
}
```

Az [AttributeUsage] lehetővé teszi, hogy beállítsunk egy nevesített tulajdonságot (AttributeUsage). Ezzel a tulajdonsággal meghatározza, hogy az attribútum többször is alkalmazható-e ugyanazon az elemen. Az [AttributeUsage] segítségével beállíthatjuk, hogy az inherited nevesített tulajdonság révén a származtatott osztályok öröklőhessék-e az attribútumot. Ha biztosítani szeretnénk, hogy a [VehicleDescription] attribútumot az osztályon vagy szerkezeten csak egyszer lehessen alkalmazni (és az érték ne öröklődjön tovább a származtatott típusokra), a VehicleDescriptionAttribute definícióját a következőképpen módosíthatjuk:

```
// Ez alkalmazza az AttributeUsage attribútummal
// látjuk el egyedi attribútumunkat.
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct,
    AllowMultiple = false, Inherited = false)]
public sealed class VehicleDescriptionAttribute : System.Attribute
{
    ...
}
```

Ekkor, ha a fejlesztő megkísérli osztály vagy egyéb struktúra helyett más elemen alkalmazni a [VehicleDescription] attribútumot, az eredmény fordítási idejű hiba lesz.

Tipp Szokjuk meg, hogy a létrehozott egyedi attribútumokat a használataukra vonatkozó jelölésekkel látjuk el, ugyanis nem minden .NET-nyelv részesíti előnyben a nem minősített attribútumok használatát.

Alkalmazásszintű (és modulszintű) attribútumok

Meghatározott modul minden típusán (többfajlos szerelvény esetén) vagy meghatározott szerelvény összes modulján alkalmazhatunk attribútumokat a [module:], illetve az [assembly:] címkek segítségével. Tételizzuk fel például, hogy biztosítani szeretnénk a szerelvényünkben definiált összes nyilvános típusCLS-kompatibilitását. Ennek érdekében a következő sort kell az egyik C#-forráskódfájlunkhoz adni (a szerelvény szintű attribútumokat a névtérdefiniáció hatókörén kívül kell elhelyeznünk):

```
//CLS-megfeleltetés kényszerítése a szerelvény összes nyilvános
// típusa számára.
[assembly: System.CLSCompliantAttribute(true)]
```

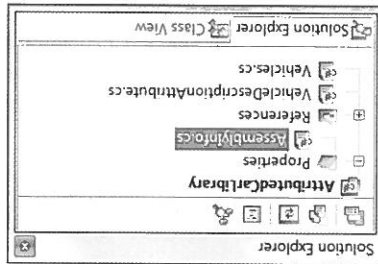
Ha a CLS-specifikációnak nem megfelelő kódreszt helyezünk el az alkalmazásban (például előjel nélküli publikus adatot):

```
// Az ulong típusok nem felelnek meg a CLS-nek.
public class Winnebago
{
    public ulong notCompliant;
```

akkor a fordításakor figyelmeztetés jelenik meg.

A Visual Studio 2008 AssemblyInfo.cs fájlja

Az alapértelmezés szerint a Visual Studio 2008 projektek rendelkeznek egy AssemblyInfo.cs fájjal, amelyet a Solution Explorer Properties ikonjának segítségével nézhetünk meg (lásd a 16.9. ábrát).



16.9. ábra: Az AssemblyInfo.cs fájl

A fájlbán kényelmesen elhelyezhető azokat az attribútumokat, amelyeket szerelvény szinten szeretnénk alkalmazni. A 15. fejezetben a „NET-szerelvényeket vizsgáljuk. Az önleírás szerelvény szintű metaadatokat foglal magában, amelyek nagy része a 16.4. táblázatban összefoglaló szerelvény szintű attribútumokból származik.

Attribútum		Valós jelentése
AssemblyCompanyAttribute	A vállalatra vonatkozó alapvető információkat tárol.	
AssemblyCopyrightAttribute	A termék vagy szerelvény szerzői joggal kapcsolatos információt tárolja.	
AssemblyCultureAttribute	A szerelvény által támogatott kultúrákról vagy nyelvekről nyújt információt.	
AssemblyDescriptionAttribute	Azon termékek vagy modulok barátságos leírását foglalja magában, amelyekből a szerelvény felépül.	
AssemblyKeyFileAttribute	Meghatározza a fájl nevét, amely a szerelvény aláírásához szükséges kulcspárt tárolja (például a szerelvény erős nevének elkészítéséhez).	
AssemblyOperationalSystemAttribute	Információt biztosít az operációs rendszerről, amelyet a szerelvény támogat.	
AssemblyProcessorAttribute	Információt biztosít a processzorról, amelyet a szerelvény támogat.	
AssemblyProductAttribute	Termék információkat biztosít.	
AssemblyTrademarkAttribute	Védjegy-információkat biztosít.	
AssemblyVersionAttribute	Meghatározza a szerelvény verzióinfor-	Meghatározza a szerelvény verzióinfor-
		mációt <major.minor.build.revision> for-
		mátumban.

16.4. táblázat: Néhány szerelvény szintű attribútum

Forráskód Az AttributedCarLibrary projekt megtalálható a 16. fejezet alkönyvtárában.

Attribútumok korai kötés használataival

16. fejezet: Típusreflexió, késői kötés és attribútumalapú programozás

Az attribútum addig felesleges, amíg valamilyen szoftverösszetevő reflexiót nem alkalmaz annak értékein. Ha a szóban forgó szoftver felfedezi az adott attribútumot, tesztése szerint bármilyen műveletet végrehajthat. A többi alkalmazáshoz hasonlóan korai kötés vagy késői kötés segítségével ez a „szoftverösszetevő” is felfedezheti az egyedi attribútum jelenlétét. Ha a korai kötés előnyeit szeretnénk kihasználni, gondoskodnunk kell arról, hogy az ügyfélalkalmazás rendelkezzen a kérdéses attribútum (ebben az esetben a vehicleDescriptionAttribute) fordítási idejű definíciójával. Mivel az attributedCarLibrary szerelevény az egyedi attribútumot nyilvános osztályként definiálta, a korai kötés a lehető legjobb megoldás.

Az egyedi attribútumokon alkalmazott reflexió bemutatására hozzuk létre a vehicleDescriptionAttributeReader C# parancssori alkalmazást, majd készítsük el az attributedCarLibrary szerelevény referenciáját. Végül frissítsük a kezdeti *.cs fájlnkat a következő kóddal:

```
// Reflexió az egyedi attribútumokon korai kötés használataival.
using System;
using AttributedCarLibrary;

public class Program
{
    static void Main(string[] args)
    {
        // Szerezzük meg a Winnebagot képviselő típust.
        Type t = typeof(Winnebago);

        // Szerezzük meg a Winnebago összes attribútumát.
        object[] customatts = t.GetCustomAttributes(false);

        // Írjuk ki a leírást.
        Console.WriteLine("***** Value of VehicleDescriptionAttribute *****\n");
        foreach (VehicleDescriptionAttribute v in customatts)
        {
            Console.WriteLine("-> {0}\n", v.Description);
            Console.ReadLine();
        }
    }
}
```

Ahogy a neve is sugallja, a `Type.GetCustomAttributes()` objektumtombot ad vissza, amely a `Type` által tartalmazott tagon alkalmazott összes attribútumot megjeleníti (a logikai paraméter szabályozza, hogy a keresést a rendszer bővítse-e felfelé a származtatási láncban). Ha megszerztük az attribútumok listáját, iteráljunk végig a `VehicleDescriptionAttribute` osztályokon, és listázzuk a `Description` tulajdonság segítségével megszerzett értéket.

Forráskód A `VehicleDescriptionAttributeReader` projekt megtalálható a 16. fejezet alkönyvtárában.

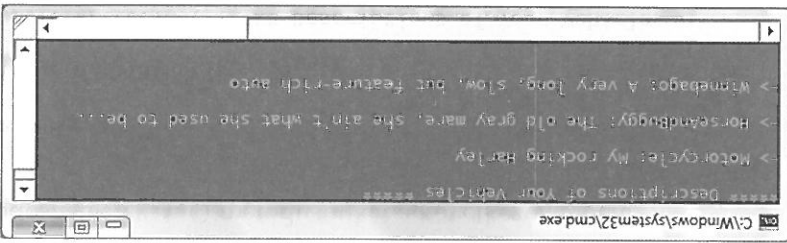
Az előző példa a korai kötés segítségével listázta a `WineBago` típusú jármű leíró adatait. Ezt az tette lehetővé, hogy a `VehicleDescriptionAttribute` osztálytípust az `AttributeReader` szerezvényben nyilvános tagként definiáltuk. A dinamikus betöltés és a késői kötés segítségével reflexiót alkalmazhatunk az attribútumokon.

Hozzuk létre a `VehicleDescriptionAttributeReader` projektet, és másoljuk az `AttributeReader.d11` fájlt a projekt `bin\Debug` könyvtárába. Módosítsuk a `Main()` metódusunkat a következőképpen:

```
using System.Reflection;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Descriptions of your Vehicles *****\n");
        // Töltsük be az AttributeReader helyi másolatát.
        Assembly asm = Assembly.Load("AttributeReader");
        // Szerezzük meg a VehicleDescriptionAttribute
        // típusinformációját.
        Type vehicleDesc =
            asm.GetType("AttributeReader.
            VehicleDescriptionAttribute");
        // Szerezzük meg a Description tulajdonság típusinformációját.
        PropertyInfo propDesc = vehicleDesc.GetProperty("Description");
```

Forráskód A `VehicleDescriptionAttribute` binding projekt megtalálható a 16. fejezet alkönyvtárában.

16.10. ábra: Attribútumok késői kötés használata



Ez a `main()` metódus már (többé-kevésbé) magától értetődő és világos. Az egyetlen érdekesség a `PropertyInfo.GetValue()` metódus alkalmazása, amelynek segítségével a `Description` tulajdonság olvasó tagját futtathatjuk. A 16.10. ábrán láthatjuk a kimenetet.

```
//Szerezzük meg a szereplvény összes típusát.
Type[] types = asm.GetTypes();

// Iteráljunk végig minden típuson, és szerezzük meg az összes
// VehicleDescriptionAttribute attribútumot.
foreach (Type t in types)
{
    object[] objs = t.GetCustomAttributes(VehicleDesc, false);

    // Iteráljunk végig az összes VehicleDescriptionAttribute
    // attribútumon, és a késői kötés használatával írassuk ki
    // a leírást.
    foreach (object o in objs)
    {
        Console.WriteLine("-> {0}: {1}\n",
            t.Name, propDesc.GetValue(o, null));
    }
}

Console.ReadLine();
}
```

16. fejezet: Típusreflexió, késői kötés és attribútumalapú programozás

Reflexió, késői kötés és egyedi attribútumok kifejtése

Reflexió, késői kötés és egyedi attribútumok kifejtése

Annak ellenére, hogy több példát láttuk a címben szereplő jelenségekre működés közben, továbbra is lehetnek kétségeink a reflexió, a dinamikus betöltés, a késői kötés és az egyedi attribútumok alkalmazásával kapcsolatban. Annak érdekében, hogy elétszerű szituációhoz tudjuk ezeket kapcsolatni, egy összetettebb példára van szükség. Tegyük fel, hogy egy olyan programozói csapatban dolgozunk, amely az alábbi követelménynek megfelelően alkalmazást készít.

- A termék külső gyártótól származó eszközök segítségével legyen kiterjeszthető.

Pontosan mit kell a *kiterjeszthetőség* értünk? Gondoljunk a Visual Studio 2008 integrált fejlesztői környezetre. Az alkalmazás fejlesztése során különböző „kampókat” helyeztek el a programban, hogy más szoftvergyártók egyedi modulokat kapcsolhassanak az integrált fejlesztői környezethez. A Visual Studio 2008 fejlesztői csapata nyilvánvalóan nem tudott olyan külső .NET-sze-relvényreferenciákat beállítani, amelyek még el sem készültek (tehát a korai kötés értelmét veszti), akkor viszont hogyan is biztosíthatná az alkalmazás a szükséges kapcsolódási pontokat? A probléma megoldásának egyik lehetséges módja a következő:

- Először is a kiterjeszthető alkalmazásnak valamilyen bemeneti felületet kell biztosítania, amely lehetővé teszi a felhasználó számára a beépített kívánt modul meghatározását (ez lehet párbeszédpanel vagy parancssori kapcsoló). Ezt *dinamikus betöltéssel* oldhatjuk meg.
- Másodsorban a kiterjeszthető alkalmazásnak meg kell tudnia határozni, hogy a modul a megfelelő funkcionalitást (pl. a szükséges inter-fészek gyűjteményt) támogatja, és így csatlakoztatható a környezethez. Ezt *reflexióval* oldhatjuk meg.

- Végül a kiterjeszthető alkalmazásnak meg kell szereznie a szükséges infrastruktúra (például interfésztypusok gyűjteményének) referenciáját, és a mögöttes funkcionalitás elindításához meg kell hívnia a tago-kat. Ezt *késői kötéssel* oldhatjuk meg.

Ha a kiterjeszthető alkalmazás meghatározott interfészek lekérdezésére képes szűlt, futásidőben képes meghatározni azt, hogy a típus aktíválható vagy sem. Ha az ellenőrzés sikeres, a kérdéses típus további interfészeket támogat, amelyek a funkcionálisukhoz sokoldalú szerkezetet biztosítanak. A Visual Studio 2008 fejlesztői pontosan ezt a megközelítést követték.

Kiterjeszthető alkalmazás készítése

A következőkben megvizsgálunk egy teljes példát, amely bemutatja egy kiterjeszthető Windows Forms-alkalmazás készítésének a folyamatát. Az alkalmazást külső szerezvények funkcionálisával gyarápíthatjuk. A példában nem foglalkozunk a Windows Forms-alkalmazások programozásának folyamataival. A Windows Forms alkalmazói felületét a II. kötet 27. fejezetében fogjuk áttekinteni. Aki tehát nem rendelkezik kellő jártassággal ezen a területen, ott talál majd tanulmányozható példát. A kiterjeszthető alkalmazásunk a következők szerezvényekkel dolgozik:

- `CommonSnapableTypes.dll`: Ez a szerezvény olyan típusdefiniciókat tartalmaz, amelyeket a beépülő modulok használnak, és amelyekre a Windows Forms-alkalmazás közvetlenül hivatkozik.
- `CSharpSnapin.dll`: C# nyelven megírt beépülő modul, amely használja a `CommonSnapableTypes.dll` típusait.
- `VbNetSnapin.dll`: Visual Basic nyelven megírt beépülő modul, amely kihasználja a `CommonSnapableTypes.dll` típusait.
- `MyExtendableApp.exe`: Ez az a Windows Forms-alkalmazás, amelyet az egyes beépülő modulok funkcionálitása kiterjeszthet.

Ez az alkalmazás dinamikus betöltés, reflexió és késői kötés segítségével dinamikusan fér hozzá azoknak a szerezvényeknek a funkcionálitásához, amelyekről nem rendelkezik előzetes ismeretekkel.

A CommonsnapableTypes.dll létrehozása

Elsőként hozzunk létre egy olyan szerelvényt, amely tartalmazza egy adott beépülő modul által használni kívánt típusokat. A beépülő modult a típusok segítségével kapcsolhatjuk a kiterjeszthető Windows Forms-alkalmazáshoz. A commonsnapableTypes osztálykönyvtárprojektt két típust definiál:

```
namespace CommonsnapableTypes
{
    public interface IAppFunctionality
    {
        void DoIt();
    }

    [AttributeUsage(AttributeTargets.Class)]
    public sealed class CompanyInfoAttribute : System.Attribute
    {
        private string companyName;
        private string companyUrl;
        public CompanyInfoAttribute(){}

        public string Name
        {
            get { return companyName; }
            set { companyName = value; }
        }

        public string Url
        {
            get { return companyUrl; }
            set { companyUrl = value; }
        }
    }
}
```

Az IAppFunctionality interfész minden beépülő modul számára polimorf implementációt biztosít, amelyet a kiterjeszthető Windows Forms-alkalmazás felhasználhat. Mivel a példa pusztán bemutató jellegű, kizárólag a DoIt() metódust határozzuk meg. Egy valós igényeket kielégítő interfész (vagy interfészgyűjtemény) lehetővé tehetné, hogy az objektum szkripteket generáljon, az alkalmazás eszköztárában képet jelenítsen meg, vagy a hosztoló alkalmazás tömenüjébe integrálódjon.

A CompanyInfoAttribute típus egyedi attribútum, amelyet bármely osztálytípuson alkalmazhatunk, ha az szeretne a tárolóba beépülni. Az osztály definíciójából kitérünk, hogy a [CompanyInfo] attribútum lehetővé teszi a beépülő modul fejlesztőjének, hogy az összetevő származási helyéről néhány alapvető információval szolgáljon.

A külső gyártó – aki a C# helyett a Visual Basic nyelvet részesíti előnyben – szerepének bemutatásához hozzunk létre egy új Visual Basic-kódkönyvtárat (vbnetnapin), amely ugyanazokra a külső szereklvényekre hivatkozik, mint az előző példában a csharpnapin projekt. A kód (ismét) szándékosan egyszerű:

Visual Basic beépülő modul készítése

Az `IAppFunctionality` interfész támogatásához explicit interfészmegvalósítást alkalmazzunk. Ez nem kötelező; a lényeg, hogy a rendszer egyetlen része, amelynek közvetlenül együt kell működnie az interfésztípussal, a hosztoló Windows-alkalmazás. Az interfész explicit megvalósításával a `DoIt()` metódus nem közvetlenül a `csharpmodule` típusból nyerjük ki.

```
using System;
using CommonsnapinTypes;
using System.Windows.Forms;

namespace Csharpnapin
{
    [CompanyInfo(Name = "InterTech Training",
        Url = "www.intertech.com")]
    public class Csharpmodule : IAppFunctionality
    {
        void IAppFunctionality.DoIt()
        {
            MessageBox.Show("You have just used the C# snap in!");
        }
    }
}
```

Az alábbiakban elkészítünk egy olyan típust, amely az `IAppFunctionality` interfészt valósítja meg. A kiterjeszthető alkalmazás általános szerkezetének megismeréséhez egy egyszerű típus is tökéletesen megfelel. Tétellezzük fel, hogy a `csharpnapin` osztálykönyvtárprojekt definícióját a `csharpmodule` osztálytípust. Mivel az osztálynak alkalmaznia kell a `CommonsnapinTypes` projektben definiált típusokat, ellenőrizzük, hogy a referenciát erre a bináris fájlra állítottuk-e be. (Arról is meg kell győződnünk, hogy a `System.Windows.Forms` is állítottunk-e be referenciát, mert használni fogjuk: a segítségével egy egyszerű üzenetet jelenítünk majd meg.) Ezután nézzük meg a forráskódot:

C# beépülő modul létrehozása

```
Imports System.Windows.Forms
Imports CommonsnapablTypes

<CompanyInfo(Name:="Chucky's Software",
url:="www.ChuckySoft.com")> _
Public Class VbNetSnapIn
    Implements IAppFunctionality

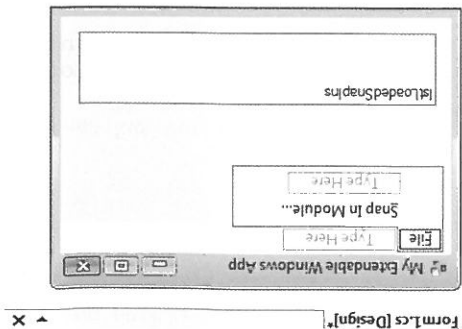
    Public Sub DoIt() Implements
        CommonsnapablTypes.IAppFunctionality.DoIt
        MessageBox.Show("You have just used the VB.NET snap in!")
    End Sub
End Class
```

A Visual Basic szintaxisában az attribútumok alkalmazásához a szükséges zárójelek (]) helyett a relációs jeleket (< >) kell használni. Meghatalozott osztályon vagy struktúrán az interfésztypusok megvalósításához az `implements` kulcsszót alkalmazzuk.

Kiterjeszthető Windows Forms-alkalmazás készítése

Végül készítsük el az új Windows Forms-alkalmazást (`MyExtendableApp`), amely lehetővé teszi a felhasználónak, hogy a standard Windows Open párbeszédpanel segítségével válasszon beépülő modult. Ezt követően állítsuk be a `CommonSnapablTypes.dll` szerezvény referenciáját, de *ne* hivatkozzunk a `CommonSnapablTypes.dll` vagy a `VbNetSnapIn.dll` kódkönyvtárakra. Az alkalmazás célja a késői kötés és a reflexió használata. Ezek segítségével határozzuk meg a külső gyártótól származó független bináris fájlok „beépíthetőségét”. He-lyezzünk egy `MenuStrip` komponenst a Visual Studio úralaptervezőben a `Form`-ra, és definiáljunk egy `File` nevű legfelső szintű menüelemet egyetlen almenüponttal, amelyet nevezzünk `Snap In Module`-nak. A főablak tartalmaz egy `ListBox`-ot is, amelyet `toolStriploadedsnapins`-nek nevezünk. Ezt fogjuk a felhasználó által betöltött beépülő modulok megjelenítésére használni. A 16.11. ábrán a végleges grafikus felület látható.

A `File > Snap In Module` menüelem eseménykezelője (amelyet tervezési időben a menüelem dupla kattintásával készíthetünk el) megjelenít egy `File Open` ablakot, és kinyeri belőle a felhasználó által kiválasztott fájl elérési út-vonalát. Ha a felhasználó nem a `CommonSnapablTypes.dll` szerezvényt választotta (hiszen az pusztán infrastruktúraszerezvény), a kód az elérési út-vonalat a `LoadExternalModule()` segédfüggvénynek küldi további feldolgozásra. A módszer hamis értéket ad vissza, ha nem talál olyan osztályt, amely az `IAppFunctionality` interfészt valósítja meg.

16.11. ábra: A *MyExtendableApp* grafikus felülete

```
private void snapInModuleToolStripMenuItem_Click(object sender,
    EventArgs e)
{
    // Lehetővé teszi a felhasználónak betöltendő szerezési
    // kiállítását.
    openFileDialog dlg = new OpenFileDialog();
    if (dlg.ShowDialog() == DialogResult.OK)
    {
        if (dlg.FileName.Contains("CommonSnapInTypes"))
        {
            MessageBox.Show("CommonSnapInTypes has no snap-ins");
        }
        else if (!LoadExternalModule(dlg.FileName))
        {
            MessageBox.Show("Nothing implements IAppFunctionality");
        }
    }
}
```

A `LoadExternalModule()` metódus a következő feladatokat hajtja végre:

- A kiválasztott szerezési dinamikusan betölti a memóriába.
- El dönti, hogy a szerezési rendelkezik-e olyan típusal, amely megvalósítja az `IAppFunctionality` interfészt.
- Késői kötéssel létrehozza a típust.

Ha olyan típust talál, amely megvalósítja az `IAppFunctionality` interfészt, meghívja a `DoIt()` metódust, majd a típus teljesen meghatározott nevét a listához adja. LINQ-lekérdezés segítségével keressük meg az `IAppFunctionality`-kompatibilis osztálytípusokat.

```

private bool LoadExternalModule(string path)
{
    bool foundSnapshot = false;
    Assembly theSnapshotasm = null;

    try
    {
        theSnapshotasm = Assembly.LoadFrom(path);
        catch(Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
        return foundSnapshot;
    }
    // A kíválasztott szerezvény dinamikus betöltése.
    theSnapshotasm = Assembly.LoadFrom(path);
    // A szerezvény összes IAppFunctionality-kompatibilis
    // osztályának megszerzése.
    var theClassTypes = from t in theSnapshotasm.GetType()
        where t.IsClass &&
        (t.GetInterface("IAppFunctionality") != null)
        select t;
    // Az objektum létrehozása és a DoIt() módszer meghívása.
    foreach (Type t in theClassTypes)
    {
        foundSnapshot = true;
        // A típus létrehozása késő később használattal.
        IAppFunctionality itfApp =
        (IAppFunctionality)theSnapshotasm.CreateInstance(t.FullName,
        true);
        itfApp.DoIt();
        loadedSnapshot.Items.Add(t.FullName);
    }
    return foundSnapshot;
}

Ekkor már futtathatjuk az alkalmazásunkat. Ha a csarpsnapin.dll vagy a
vbnetcsnapin.dll szerezlevényt választjuk, a megfelelő üzenetnek kell megje-
lennie. Az utolsó feladat az, hogy megjelentsük a [companyinfo] attribútum
által biztosított metaadatokat. Ehhez módosítsuk a LoadExternalModule() me-
tódust. Mielőtt kilépne a foreach hatókörből, a módszer meghívja a Display-
CompanyData() segédfüggvényt. A módszer egyetlen System.Type paraméter-
rel rendelkezik.

```

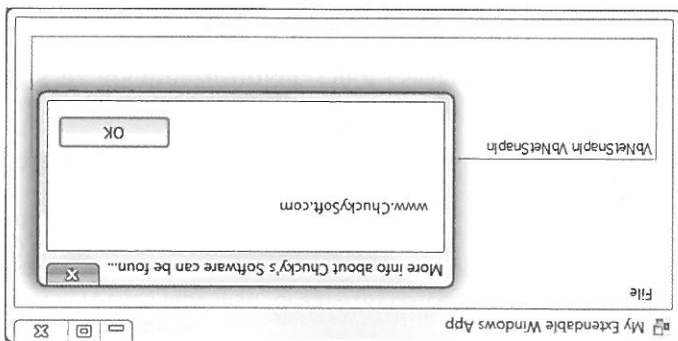
```
// Céginformáció mutatója.
displayCompanyData(t);
}
return foundSnapshot;
}
```

A bejövő típuson alkalmazzuk a `reflexió` a `[CompanyInfo]` attribútum kinyerésére:

```
private void DisplayCompanyData(Type t)
{
    // A [CompanyInfo] adatok megszerzése.
    var compInfo = from ci in t.GetCustomAttributes(false) where
        (ci.GetType() == typeof(CompanyInfoAttribute))
        select ci;

    // Az adatok mutatója.
    foreach (CompanyInfoAttribute c in compInfo)
    {
        MessageBox.Show(c.Url,
            string.Format("More info about {0} can be found at", c.Name));
    }
}
```

A 16.12. ábrán a futtatás eredménye látható.



16.12. ábra: Külső szerezőcények beépítése

A példa megmutatta, hogy a fent tárgyalt témakörök hogyan segíthetik mindennapi munkánkat, és minden szoftverfejlesztőnek hasznosak lehetnek.

Forráskód A `CommonSnapIn` típus, a `VbNetSnapIn`, és a `MyExtendableApp` projektek a 16. fejezet alkönyvtárában találhatók.

Összefoglalás

A reflexió a robosztus OO környezetek érdekes oldala. A .NET világában a reflexió szolgáltatások sárokköve a `System.Type` osztály és a `System.Reflection` névtér. A reflexió segítségével a típust futásidőben vizsgáljuk, és próbáljuk megértetni az adott elemre vonatkozó ki?, mi?, hol?, mikor?, miért? és hogyan? kérdéseket.

A késői kötés folyamatában létrehozunk a típust, és a tagok nevének előzetes ismerete nélkül hívjuk meg a típus említett tagjait. A késői kötés gyakran a *dynamicus betöltés* közvetlen eredménye, amely lehetővé teszi, hogy a .NET-szerelvényeket programozottan töltsük a memóriába. A kiterjeszthető alkalmazás példájában láthattuk, ezt a hatékony módszert az eszközök fejlesztői, valamint azok felhasználói is használják. Megvizsgáltuk az attribútumalapú programozás szerepét. Ha típusainkat attribútumokkal látjuk el, akkor a mögöttes szerelvény metaadatait gyarapítjuk.

Folyamatok, alkalmazástartományok és objektumkörnyezetek

Az előzőekben a CLR-nek azokat a lépéseit tanulmányoztuk, amelyekkel feloldhatunk egy külsőleg hivatkozott szerelvény helyét, valamint a .NET-metadatok szerepét. Ebben a fejezetben megnézzük, hogy a CLR hogyan hosztol egy szerelvényt, és megvizsgáljuk a folyamatok, az alkalmazástartományok és az objektumkörnyezetek közötti kapcsolatot.

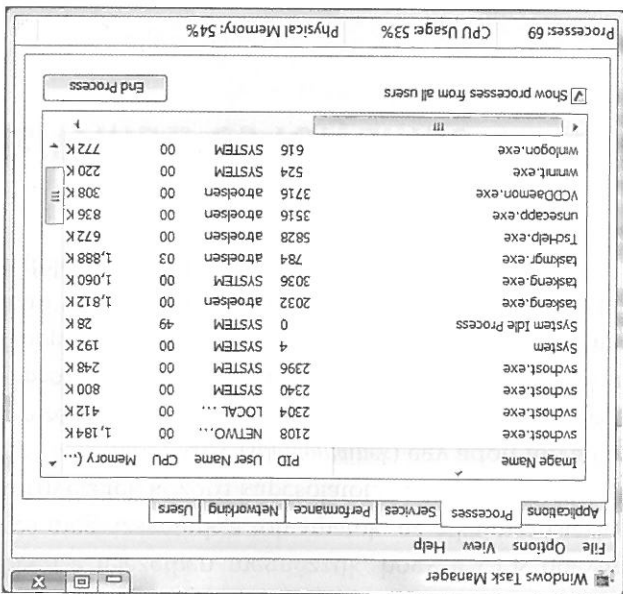
Röviden: az *alkalmazástartományok* (vagy *AppDomain*-ek) egy adott folyamaton belül olyan logikai alrészek, amelyek kapcsolódó .NET-szerelvények készletét hosztolják. Az *AppDomain*-in tovább tagolható *környezetfüggetlen határokra*, amelyek a hasonló .NET-objektumokat csoportosítják. A környezet fogalmát használva a CLR biztosítani tudja, hogy a speciális futási idejű igényeket támasztó objektumokat megfelelően kezelhessük.

A tradicionális Win32-es folyamat áttekintése

A „folyamat” fogalma már jóval a .NET platform kiadása előtt létezett a Windows operációs rendszerekben. A *folyamat* kifejezés a futó alkalmazással használt erőforráskészletet (mint a külső kódkönyvtárak és az elsődleges szál), valamint a szükséges memóriafoglalást írja le. Minthogy minden *.exe fájl betöltődik a memóriába, az operációs rendszer létrehoz számukra egy saját, elkülönített folyamatot, amelyet bezárásukig használ. Az alkalmazás-izoláció ezen formáját használva az eredmény egy sokkal robusztusabb és stabilabb futási környezet lesz, ugyanis egy folyamat hibája nem befolyásolja a többi működését.

Minden Win32-folyamathoz hozzá van rendelve egy egyedi folyamatazonosító (PID), és az operációs rendszer szükség esetén egymástól függetlenül betöltheti vagy eltávolíthatja ezeket a folyamatokat (illetve ez programozottan a Win32 API hívásainak a segítségével is történhet). A Windows Taskmanager (feladatkezelő) (amely a Ctrl + Shift + Esc billentyűparanccsal hívható elő) Processes (folyamatok) lapja lehetővé teszi az adott gépen futó folyamatokkal kapcsolatos különböző statisztikák megtekintését, beleértve a PID-et és a programkódnevet (lásd 17.1. ábra).

Megjegyzés A Windows-feladatkezelő Nézet > Oszlopok kiválasztása menüjében kiválasztathatjuk, hogy mely oszlopok (PID, felhasználó neve stb.) jelenjenek meg.



17.1. ábra: A Windows-feladatkezelő

A szálak áttekintése

Minden Win32-folyamat pontosan egy „szállal” rendelkezik, amely az alkalmazás belépési pontjaként szolgál. A következő fejezetben lesz arról szó, hogy hogyan hozhatunk létre szálakat a .NET platformban a System.Threading névtér használatával; ám néhány definícióra már most szükségünk lesz. Először is: a szál egy folyamaton belüli futási útvonal. A folyamat belépési pontja általában létrehozott első szál neve hivatalosan *elsődleges szál*. A Win32 API-ra épülő

grafikus asztali alkalmazások határozzák meg a `winmain()` metódust mint az alkalmazás belépési pontját. Másrészt: egy konzolalapú program ugyanerre a célra a `main()` metódust biztosítja.

Az olyan folyamatokat, amelyek egyetlen elsődleges végrehajtási szálát tartalmazzák, *szálbiztosnak* nevezzük, ugyanis csak egy olyan szál van, amely adott időben hozzáfér az alkalmazás adataihoz. Egy egyszerű folyamat (különösen egy GUI-alapú) azonban gyakran nem válaszol, ha ez az egy szál valamilyen összetett műveletet hajt végre (pl. egy hosszú szöveges fájl nyomat, intenzív matematikai számolást végez, vagy egy több ezer kilométerre lévő távoli kiszolgálóhoz próbál csatlakozni).

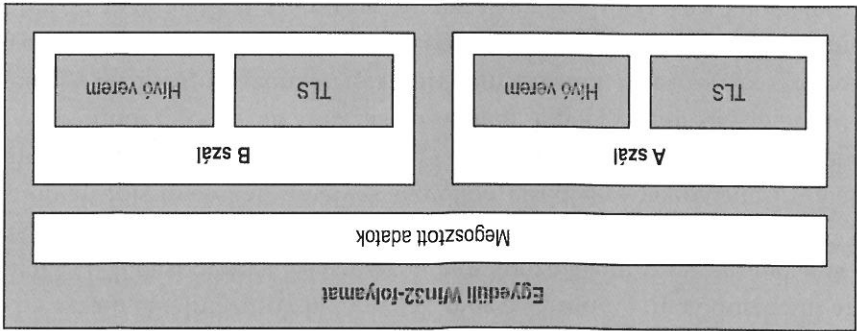
Mint ahogy az egyszerű alkalmazásoknak megvan ez a lehetséges hátránya, a Win32 API (és a .NET platform) lehetővé teszi, hogy az elsődleges szál továbbbi másodlagos szálakat (más néven: *dolgozó szálakat*) származtasson, még hozzá olyan Win32 API funkciók használatával, mint a `createthread()`. Minden szál (elsődleges és másodlagos) egyedi futási útvonalra válik a folyamatban, és egyidejűleg hozzáfér minden megosztott adathoz.

A fejlesztők általában további szálakat is létrehozhatnak, hogy javítsák a program általános reagálókészségét. A többszálú folyamatok azt az érzést keltik, hogy számos tevékenység történik nagyjából ugyanakkor. Új szál indítható például egy nagy munkai igényű feladat (pl. egy nagy szövegfájl) nyomtatásának) elvégzésére.

Ha ez a másodlagos szál intenzíven dolgozik, a fő szál még mindig válaszol a bevitelre, amely az egész folyamatnak nagyobb teljesítményt képes adni.

Am nem feltétlenül ez a helyzet: ugyanis túl sok szál használata egy folyamatban valóban *csökkentheti* a teljesítményt, hiszen a processzornak valtozgatnia kell a folyamat aktív szálai között (és ez időt vesz igénybe).

Igy tehát a többszálság többnyire csak az operációs rendszer által nyújtott illúzió. Azok a gépek, amelyek egyetlen (nem hyperthreading) processzort használnak, nem is képesek egyszerre több szálát kezelni ugyanabban az időpontban. Ehelyett az egyetlen processzor egy időegységben (neve: *időszlet*) egyetlen szálát fog futtatni, ez pedig részben a szál prioritási szintjén alapul. Amikor egy szál időszlete lejár, akkor a szál felülgésztesődik, hogy más szál számára is lehetővé tegye a futtatást. Ahhoz, hogy minden szál emlékezzen arra, hogy mi történt a „felrealizálásuk” előtt, rendelkeznek azzal a képességgel, hogy írjanak a Thread Local Storage-ba (TLS), valamint külön hívásveremmel is, ahogy az a 17.2. ábrán látható.



17.2. ábra: A Win32 folyamat/szál kapcsolat

A szál tehát egy egyedi futási útvonal egy Win32-folyamatban. Minden folyamat rendelkezik elsődleges szállal (amely a végrehajtható fájl belepési pontján keresztül jön létre), valamint tartalmazhat programozottan létrehozott további szálakat is.

Folyamatok kezelése a .NET platformban

Maguk a folyamatok és a szálak nem jelentenek újdonstágot, a .NET platformban azonban ezeknek a primitíveknek a kezelése jelentősen javult. Először nézzük meg, hogy hogyan lehet kezelni a folyamatokat a .NET-alapoztatálykönyvtárak használatával (a többszálú szerverelvények építésének mikéntjét lásd a 18. fejezetben).

A System.Diagnostics névtér számos olyan típust meghatároz, amelyek lehetővé teszik a folyamatok és a különböző, diagnosztikához kapcsolódó típusok (pl. a rendszer-eseménynapló és a teljesítményszámlálók) programozott kezelését. Ebben a fejezetben csak a 17.1. táblázatban található folyamatközpon

A System.Diagnostics névtér folyamatközpontú típusai	Valós jelentés
--	----------------

Process
A Process osztály hozzáférést biztosít a helyi és távoli folyamatokhoz, valamint programozottan lehetővé teszi folyamatok indítását és leállítását.

A System.Diagnostics névtér	
Folyamatközpontú típusai	
Valós jelentés	

ProcessModule
Ez a típus olyan modult (*.dll vagy *.exe) ábrázol, amelyet egy bizonyos folyamatba töltött-tünk be. A ProcessModule típus bármilyen modult – COM-alapú, .NET-alapú vagy hagyományos C-alapú binárisokat – jelképezhet.

ProcessModuleCollection
Biztosítja a ProcessModule objektumok erősen típusos gyűjteményét.

ProcessStartInfo
Megad egy olyan értékkészletet, amely egy folyamat Process.Start() metódus általi indításakor használatos.

ProcessThread
Egy szálát ábrázol egy adott folyamatban. A ProcessThread típust arra használjuk, hogy diagnosztizáljuk egy folyamat szálkészletét, nem pedig arra, hogy új végrehajtási szálakat származtassunk a folyamaton belül.

ProcessThreadCollection
Biztosítja a ProcessThread objektumok erősen típusos gyűjteményét.

17.1. táblázat: A System.Diagnostics névtér tagjai

A system.Diagnostics.Process típus biztosítja az adott (helyi vagy távoli) gépen futó folyamatok analízálását. A Process osztály emellett olyan tagokat is biztosít, amelyek lehetővé teszik folyamatok programozott indítását és befejezését, egy folyamat prioritási szintjének ellenőrzését, valamint az adott folyamat aktív számainak és/vagy betöltött moduljainak listázását. A 17.2. táblázat mutatja a system.Diagnostics.Process néhány (de nem az összes) kulcstagját.

Tagok	Valós jelentés
-------	----------------

ExitCode
Ezen a tulajdonságon keresztül a folyamat kilépési kódja kérdezhető le. Ennek az értéknek a megszerzéséhez az ExitCode.Escaped eseményt kell kezelniünk (az aszinkron értesítéshez), vagy meg kell hívunk a WaitForExit() metódust (a szinkron értesítéshez).

ExitTime
Ez a tulajdonság lekérdezi a befejeződött folyamathoz társított időbélyegyet (ez a tulajdonság a folyamat leállításának idejét mutatja).

Tagok		Valós jelentés
Handle	Ez a tulajdonság visszaadja az operációs rendszer által a folyamathoz társított azonosítót.	
HandleCount	Ez a tulajdonság visszaadja a folyamat által megnyitott azonosítók számát.	
Id	Ez a tulajdonság lekérdezi a társított folyamat PID-jét.	
MachName	Ez a tulajdonság lekérdezi annak a számlítógépnek a nevét, amelyen a társított folyamat fut.	
ModuleName	Ez a tulajdonság lekérdezi a ProcessModule típust, amely egy adott folyamat fő modulját jelképezi.	
MatInWindwTitle	A MatInWindwTitle lekérdezi a folyamat fő ablakának a címét (ha a folyamatnak nincsen fő ablaka, akkor üres stringet kapunk). A MatInWindwTitle lekérdezi a társított ablak alapjául szolgáló azonosítót (a System, IntPtr típus révén). Ha a folyamatnak nincs főablaka, akkor az IntPtr típus a System, IntPtr.Zero értéket kapja.	
Modules	Ez a tulajdonság hozzáférést biztosít az erősen tipizált ProcessModuleCollection típushoz, amely az aktuális folyamat során betöltött modulkészletet (*, dll vagy *.exe) tartalmazza.	
PriorityBoostEnabled	Ezen a tulajdonságon keresztül beállítható, hogy az operációs rendszer több erőforrást biztosítson-e az alkalmazásnak arra az időre, amíg a hozzátartozó ablakon van a fókusz.	
PriorityClass	Ez a tulajdonság lehetővé teszi a társított folyamatok általános prioritásának a megtekintését és a módosítását.	
ProcessName	Ez a tulajdonság lekérdezi a folyamat nevét (amely magának az alkalmazásnak a neve).	
Responding	Ez a tulajdonság azt mutatja meg, hogy az alkalmazás felhasználói felülete pillanatnyilag tud-e reagálni a felhasználói akciókra (vagy éppen „kifagyott”).	
StartTime	Ez a tulajdonság lekérdezi a társított folyamat indításának idejét (a DateTime típus révén).	

Tagok		Valós jelentés
Threads	Ez a tulajdonság megkapja a társított folyamatban futó szálak készletét (amelyet a ProcessThread típusok egy tömbje jelképez).	
CloseMainWindow()	Ez a módszer úgy zárja be a felhasználói felületet al-kalmazó folyamatot, hogy bezárási üzenetet küld a főablakának.	
GetCurrentProcess()	Ez a statikus módszer visszaad egy olyan új Process típust, amely a jelenleg aktív folyamatot képviseli.	
GetProcesses()	Ez a statikus módszer visszaadja egy adott gépen futó új Process komponensek tömbjét.	
Kill()	Ez a módszer azonnal leállítja a társított folyamatot.	
Start()	Ez a módszer elindít egy folyamatot.	

17.2. táblázat: A Process típus tagjai

A futó folyamatok listázása

A process példányok kezelési folyamatának bemutatásához tetelezzük fel, hogy van egy olyan, ProcessManager névű C# parancssori alkalmazásunk, amely az alábbi statikus segédfüggvényt határozza meg a program osz-tályon belül (mindenképpen importáljuk a System.Diagnostics névtérrel):

```
static void ListAllRunningProcesses()
{
    // Lekérdezi a helyi gépen futó összes folyamatot.
    Process[] runningProcs = Process.GetProcesses("");
    // Kijelent minden folyamat azonosítóját (PID) és nevét.
    foreach (Process p in runningProcs)
    {
        string info = string.Format("-> PID: {0} \tName: {1}",
            p.Id, p.ProcessName);
        Console.WriteLine(info);
    }
    Console.WriteLine("*****\n");
}
```

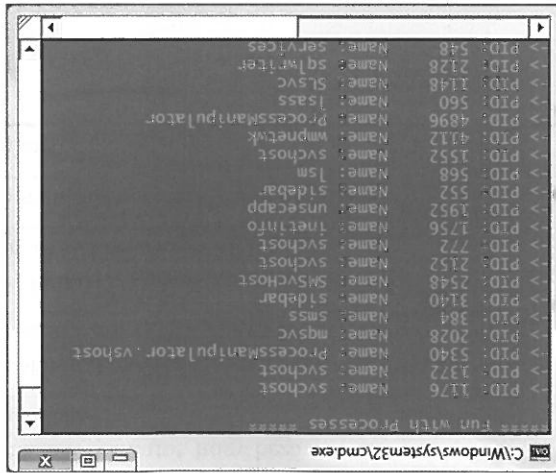
Figyeljük meg, hogy a statikus Process.GetProcesses() módszer hogyan adja vissza a célgépen (a pont jelölés a helyi számítógépet jelzi) futó folyamatokat egy Process típusú tömbben. Ha megszerezünk a Process típusok tömbjét,

```
// Ha nem létezik 987-es azonosítóval folyamat,
// akkor futási idejű kivételt kapunk..
static void GetSpecificProcess()
{
    Process theProc = null;
    try
    {
        theProc = Process.GetProcessById(987);
    }
}
```

Amellett, hogy megszerezhetjük egy adott gépen futó összes folyamat listáját, a statikus `Process.GetProcessById()` módszer lehetővé teszi egyetlen `Process` típus megszerzését is a hozzáátírtított folyamatazonosító (PID) révén. Ha nem létező folyamatazonosítóhoz próbálunk hozzáférni, akkor a rendszert `ArgumentException` kivételt dob. Ha például meg szeretnénk szerezni a 987-es azonosítójú folyamatot jelképező `Process` objektumot, akkor a következőt írhatjuk:

Egy adott folyamat vizsgálata

17.3. ábra: A futó folyamatok listája




```

catch // Az egyszerűség kedvéért általános catch szerepel.
{
    Console.WriteLine("<-> Sorry...bad PID!");
}
}

```

Egy folyamat szálkészletének vizsgálata

A process osztálytípus arra is módot ad, hogy programozottan megvizsgáljuk az adott folyamat által használt összes szálát. A szálkészletet az erősen típusos `ProcessThreadCollection` gyűjtemény jelképezi, amely néhány önálló `ProcessThread` típust tartalmaz. Tegyük fel, hogy a következő statikus segéd-függvényt adtuk hozzá az aktuális alkalmazásunkhoz:

```

static void EnumThreadsForPid(int pid)
{
    Process theProc = null;
    try
    {
        theProc = Process.GetProcessById(pid);
    }
    catch
    {
        Console.WriteLine("<-> Sorry...bad PID!");
    }
    Console.WriteLine("*****\n");
    return;
}

// A megadott folyamat összes szálához tartozó
// statiszika listázása.
Console.WriteLine("Here are the threads used by: {0}",
    theProc.ProcessName);
ProcessThreadCollection theThreads = theProc.Threads;
foreach(ProcessThread pt in theThreads)
{
    string info =
        string.Format("<-> Thread ID: {0}\tstart time {1}\tpriority {2}", pt.Id, pt.StartTime.ToShortTimeString(), pt.PriorityLevel);
    Console.WriteLine(info);
}
Console.WriteLine("*****\n");
}

```

A `System.Diagnostics.Process` típus `Threads` tulajdonsága hozzáférést biztosít a `ProcessThreadCollection` osztályhoz. Ekkor kiterítjük a hozzáférhető szálaazonosítót, indítási időt, valamint a kliens által meghatározott prioritási szinteket. Így, ha az alábbiak szerint módosítjuk programunk `Main()` metódusát, hogy kérje be a vizsgálni kívánt folyamat azonosítóját:

Tag	Valós jelentés
-----	----------------

ProcessAffinity	Lekérdezi a processzorokat, amelyeken a társított szál futhat.
StartAddress	Lekérdezi annak a függvénynek a memóriacímét, amelyet az operációs rendszer hívott meg a szál elindítására.
StartTime	Lekérdezi azt az időpontot, amikor az operációs rendszer elindította a szálát.
ThreadState	Lekérdezi a szál aktuális állapotát.
TotalProcessorTime	Lekérdezi azt a teljes időmennyiséget, ameddig a szál a processzort használta.
WaitReason	Lekérdezi annak okát, hogy miért várakozik a szál.

17.3. táblázat: A *ProcessThread* típus tagjai

A *ProcessThread* típus *nem* arra használatos, hogy létrehozzon, felfüggeszsen vagy leállítson szálakat a .NET platformban, hanem arra szolgál, hogy diagnosztikai információkat szerezhessünk egy futó folyamatban lévő aktív Win32-es szálakról. (A többszálú alkalmazások készítését a *System.Thread* névtér használatával lásd a 18. fejezetben.)

Egy folyamat modulkészletének vizsgálata

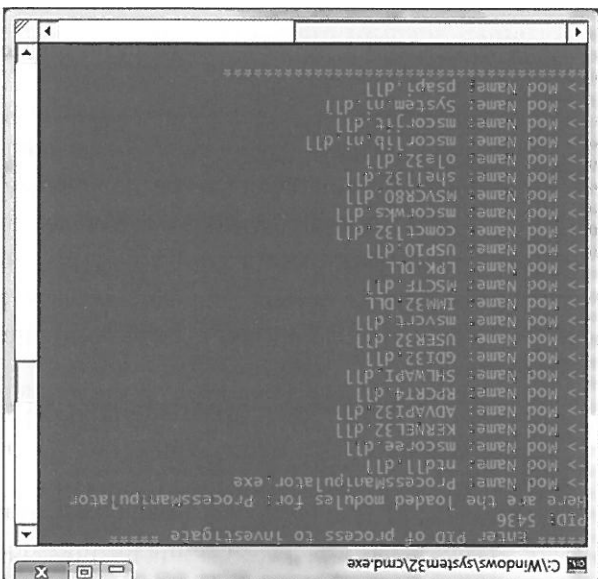
Nézünk meg, hogyan iterálhatunk végig a betöltött modulokon, amelyeket egy adott folyamat hosztol. A *modul* olyan általános név, amely egy bizonyos folyamat által hosztolt adott *.dll-t vagy magát az *.exe-t írja le. Amikor a *Process.Module* tulajdonság révén férünk hozzá a *ProcessModuleCollection* gyűjteményhez, akkor listázhatjuk az adott folyamatban hosztolt összes *modul*. NET-alapú, COM-alapú vagy tradicionális C-alapú könyvtárat. Az alábbi segédfüggvény a PID alapján kilistázza egy adott folyamat moduljait:

```
static void EnumModForPid(int pid)
{
    process theProc = null;
    try
    {
        theProc = Process.GetProcessById(pid);
    }
```

```

        catch
        {
            Console.WriteLine("> Sorry...bad PID!");
            return;
        }
        Console.WriteLine("Here are the loaded modules for: {0}",
            theProc.ProcessName);
        try
        {
            ProcessModuleCollection theMods = theProc.Modules;
            foreach(ProcessModule pm in theMods)
            {
                string info = string.Format("> Mod Name: {0}",
                    pm.ModuleName);
                Console.WriteLine(info);
            }
            Console.WriteLine("*****\n");
        }
        catch
        {
            Console.WriteLine("No mods!");
        }
    }
}

```



17.5. ábra: A futó folyamatban betöltött modulok listázása

Nézzünk egy lehetséges kimenetet: Írassuk ki a hosszoló folyamat (ProcessMani-
pulator) betöltött moduljait. Írdítsuk el az alkalmazást, és nézzük meg a PID-jét
a TaskManagerben, majd adjuk meg ezt az értéket az alkalmazásunknak. (Ko-

rábban úgy módosítottuk a `main()` metódust, hogy az továbbadja a PID-et az `enummódosforpid()` metódusnak.) Ha mindezt megcsejtük, megcsejtve láthatjuk egy egyszerű parancssori alkalmazás során felhasználó *.dll-ek listáját (`GD132.dll, USER32.dll, o1e32.dll stb.`). A 17.5. ábra mutatja a tesztutást fenn.

Folyamatok programozott indítása és leállítása

A `System.Diagnostics.Process` típus itt vizsgált utolsó aspektusai a `start()` és `kill()` metódusok. Ezek a tagok egy folyamat elindításának, illetve leállításának programozott módját biztosítják. Figyeljünk meg például a statikus `startAndKillProcess()` segédmetódust:

```
static void startAndKillProcess()
{
    // Internet Explorer elindítása.
    Process pProc = Process.Start("IExplore.exe",
    "www.intertech.com");
    Console.WriteLine("--> Hit enter to kill {0}...", pProc.ProcessName);
    try
    {
        pProc.Kill();
    }
    catch { } // Ha a felhasználó már leállította volna...
```

A statikus `Process.Start()` metódus többszörösen túlterhelt. Legálább az elindítást követően a folyamatot meg kell adnunk (mint a Microsoft Internet Explorer, `IExplore.exe`). Ez a példa a `start()` metódus egy olyan variációját használja, amellyel bármennyi további paramétert megadhatunk továbbadva a program belepési pontjának (vagyis a `main()` metódusnak). A `start()` metódus azt is lehetővé teszi, hogy egy `System.Diagnostics.ProcessStartInfo` típust adjunk át, amely további információkat ad a folyamat elindulásával kapcsolatban. A `ProcessStartInfo` formális definíciója a következő (részletekért lásd a .NET Framework 3.5 SDK dokumentációját):

A.NET-alkalmazástartomány

Forráskód A `ProcessManager` projekt megtalálható a 17. fejezet alkönyvtárában.

A `Process.start()` metódus bármely verzióját hívjuk meg, az az újonnan aktivált folyamat referenciájával tér vissza. Ha le akarunk állítani egy folyamatot, egyszerűen hívjuk meg a példányszintű `kill()` metódust.

```
public sealed class System.Diagnostics.ProcessStartInfo :
    object
{
    public ProcessStartInfo();
    public ProcessStartInfo(string fileName);
    public ProcessStartInfo(string fileName, string arguments);
    public string Arguments { get; set; }
    public bool CreateWindow { get; set; }
    public stringIdictionary EnvironmentVariables { get; }
    public bool ErrorDialog { get; set; }
    public IntPtr ErrorDialogParentHandle { get; set; }
    public string FileName { get; set; }
    public bool RedirectStandardError { get; set; }
    public bool RedirectStandardInput { get; set; }
    public bool RedirectStandardOutput { get; set; }
    public bool UseShellExecute { get; set; }
    public string Verb { get; set; }
    public string[] Verbs { get; }
    public ProcessWindowStyle WindowStyle { get; set; }
    public string WorkingDirectory { get; set; }
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
}
```

A következőkben vizsgáljuk meg a .NET-alkalmazástartomány koncepcióját. A .NET platformban a futtatható fájlok nem közvetlenül egy folyamaton belül vannak hosztolva (ellentétben a hagyományos Win32-alkalmazásokkal). Ehelyett, a .NET végrehajtható fájlt egy logikai partíció hosztolja az *alkalmazástartomány* nevű folyamaton belül. Egyetlen folyamat több alkalmazástartományt is tartalmazhat, ezek mindegyike egy .NET végrehajtható fájlt hosztol. A tradicionális Win32-folyamat ezen további felosztása több tekintetben is előnyös; nézzünk meg néhányat:

- Az alkalmazástartományok a .NET platform operációsrendszer-függően természetének egyfajta kulcselemei, hiszen ez a logikai felosztás megszűnteti a különbséget arra nézve, hogy az alapul szolgáló operációs rendszer hogyan ábrázol egy betöltött végrehajtható fájlt.
- Az alkalmazástartományok kevesebb processzoridőt és memóriát igényelnek, mint a teljes folyamatok. Ezért a CLR gyorsabban képes betölteni és eltávolítani őket, mint egy formális folyamatot.
- Az alkalmazástartományok melyebb izolációs szintet biztosítanak a betöltött alkalmazás hosztolására. Ha az egyik folyamat alkalmazástartománya leáll, a többi tovább működik.

Egyetlen folyamat bármennyi alkalmazástartományt hosztolhat, ezek mindegyike teljesen elkülönül a folyamat egyéb alkalmazástartományaitól (és más folyamatoktól is). Emiatt egy alkalmazástartományban futó alkalmazás semmilyen adatot (globális változót vagy statikus mezőt) nem tud megsebezni egy másik alkalmazástartományból, hacsak nem használnak elosztott programozási protokollt (pl. a Windows Communication Foundationt). Egy folyamat több alkalmazástartományt *hosztolhat*, általában azonban nem ez történik. Az operációs rendszer egyik folyamata fogja hosztolni azt, amit *alapértelmezett alkalmazástartománynak* nevezünk. Ezt a specifikus alkalmazástartományt a CLR automatikusan létrehozza a folyamat indításakor. Ezután ha szükséges, további alkalmazástartományokat is létrehoz. Ha igény van rá (valószínűleg *nem* az lesz a helyzet a .NET használatakor), akkor futásidőben is létrehozhatunk alkalmazástartományokat egy adott folyamaton belül, programozottan, a `system.AppDomain` osztály statikus metódusainak felhasználásával. Ez az osztály az alkalmazástartományok aláacsony szintű vezérléséhez is hasznos. Az osztály kulcstagjai a 17.4. táblázatban láthatók.

Tag	Valós jelentés
<code>CreateDomain()</code>	Ez a statikus metódus új alkalmazástartományt hoz létre az aktuális folyamaton. Mivel a CLR szükség esetén gondoskodik az új alkalmazástartományok létrehozásáról, ezért valószínűen, hogy valaha is meg kell hív-nunk ezt a metódust.
<code>GetCurrentThread()</code>	Ez a statikus metódus visszaadja az aktuális alkalmazástartományban az aktív szál azonosítóját.

Tag	Valós jelentés
-----	----------------

unload() Ez a statikus módszer lehetővé teszi a megadott alkalmazástartomány eltávolítását egy folyamattól.

Basediractory Ez a tulajdonság lekérdezi a függő szerelvények keresésére használt alapkönyvtárat.

CreateInstance() Ez a módszer létrehozza az adott szerelvényfájlból meghatározott típus egy példányát.

ExecuteAssembly() Ez a módszer egy szerelvényt hajt végre egy alkalmazástartományon belül, megadva a fajlnevet.

GetAssembly() Ez a módszer lekérdezi az alkalmazástartományba betöltött .NET-szerelvényeket (a COM- és C-alapú szerelvényeket figyelmen kívül hagyja).

Load () Ez a módszer dinamikusan betölt egy szerelvényt az aktuális alkalmazástartományba.

17.4. táblázat: Az alkalmazástartomány tagjai

Emellett az AppDomain típus olyan kis eseménykészletet is meghatároz, amely megfelel az alkalmazástartomány életciklusára jellemző aspektusoknak (lásd 17.5. táblázatot).

Esemény	Valós jelentés
---------	----------------

AssemblyLoad Szerelvény betöltésekor következik be.

AssemblyResolve Szerelvény feloldásának hibájakor fordul elő.

DomainUnload Alkalmazástartomány eltávolítása előtt fordul elő.

ProcessExit Az alapértelmezett alkalmazástartományon fordul elő akkor, amikor annak a szülőfolyamata kilép.

ResourceResolve Egy erőforrás feloldásának hibájakor fordul elő.

TypeResolve Egy típus feloldásának hibájakor fordul elő.

UnhandledException Akkor fordul elő, ha egy kivételt nem kap egyetlen eseménykezelő sem.

17.5. táblázat: Az AppDomain típus eseményei

Folyamat alkalmazástartományainak felsorolása

A .NET-alkalmazástartomány

A .NET-alkalmazástartományok programozott kezelésének bemutatásához tegyük fel, hogy létrehoztunk egy új C# parancssori alkalmazást AppDomainManipulator néven, amely meghatároz egy PrintAllAssembliesInAppDomain() nevű metódust a Program típuson belül. Ez a segédmetódus az AppDomain.GetAssemblies() használataival szerzi meg a szóban forgó alkalmazástartományban hosztolt összes .NET bináris fájl listáját.

Ezt a listát a System.Reflection.Assembly típusok egy tömbje reprezentálja, így a System.Reflection névteret kell használnunk (lásd a 16. fejezetet). Ha megszerezjük a szerelvénytömböt, végigigiterálunk a tömbön, és kiterítjük az egyes szerelvények barátságos nevét, valamint verzióját:

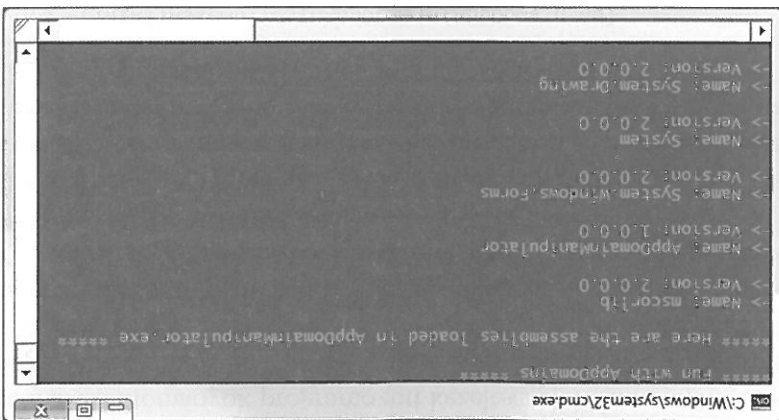
```
static void PrintAllAssembliesInAppDomain(AppDomain ad)
{
    Assembly[] loadedAssemblies = ad.GetAssemblies();
    Console.WriteLine("***** Here are the assemblies loaded in {0}
    ", ad.FriendlyName);
    foreach(Assembly a in loadedAssemblies)
    {
        Console.WriteLine("-> Name: {0}", a.GetName().Name);
        Console.WriteLine("-> Version: {0}\n", a.GetName().Version);
    }
}
```

Módosítsuk úgy a main() metódust, hogy megszerezze a referenciát az aktuális alkalmazástartományra, mielőtt az AppDomain.CurrentDomain tulajdonság használataival meghívna a PrintAllAssembliesInAppDomain() metódust.

A main() metódus elindít egy Windows Forms üzenetdobozt, hogy rákényszerítse a CLR-t a System.Windows.Forms.dll, a System.Drawing.dll és a System.dll szerelvények betöltésére (mindenképpen állítsunk be egy hivatkozást ezekre a szerelvényekre, és frissítsük ennek megfelelően a using utasításokat):

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with AppDomains *****\n");
    // Az aktuális alkalmazástartomány információinak beolvasása.
    AppDomain defaultAD = AppDomain.CurrentDomain;
    // A hívás csak további szerelvényeket tölt be
    // az alkalmazástartományba.
    MessageBox.Show("Hello");
    PrintAllAssembliesInAppDomain(defaultAD);
    Console.ReadLine();
}
```

A 17.6. ábra mutatja a kimenetet.



17.6. ábra: Szerelevények felsorolása az aktuális alkalmazástartományban

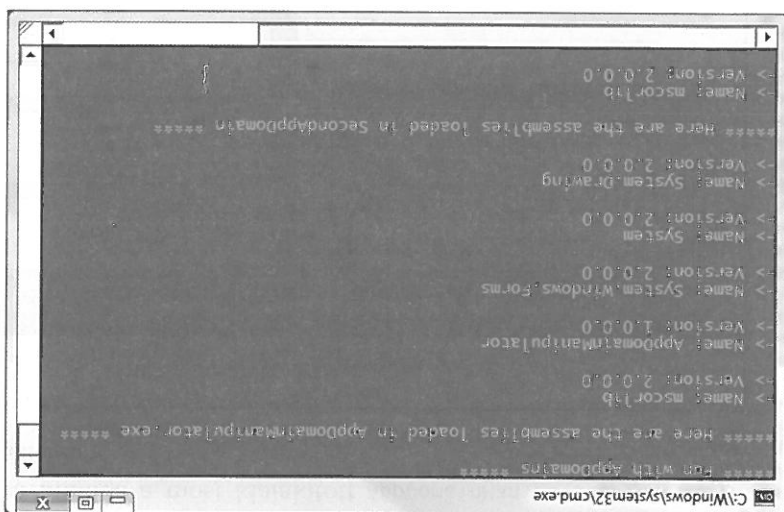
Új alkalmazástartományok programozott létrehozása

Egyetlen folyamat több alkalmazástartományt is hosztolhat. Meglehetősen ritkán kell manuálisan alkalmazástartományokat létrehozunk, ezt mégis megtehejük a statikus `CreateDomain()` metódus hívásával. Az `AppDomain.CreateDomain()` többszörösen túlterhelt metódus. Legálább az új alkalmazástartomány barátságos nevét meg kell adnunk a következőképpen:

```

static void Main(string[] args)
{
    ...
    // új alkalmazástartomány létrehozása az aktuális folyamatban.
    AppDomain anotherAD = AppDomain.CreateDomain("SecondAppDomain");
    PrintAllAssembliesInAppDomain(anotherAD);
}
Console.ReadLine();
  
```

Ha ismét futtatjuk az alkalmazást (lásd a 17.7. ábrát), figyeljük meg, hogy a csak az alapértelmezett alkalmazástartományban töltődnek be. Egy Win32-programozó szemszögéből nézve ez meglepő lehet (azt várnánk ugyanis, hogy mindkét alkalmazástartomány ugyanazokhoz a szerelevényekhez fér hozzá). Egy szerelevény azonban az *alkalmazástartományba* töltődik, nem pedig közvetlenül magába a folyamatba.



17.7. ábra: Egyetlen folyamat két alkalmazástartományával

A `secondAppDomain` alkalmazástartomány magában foglalja az `mscorlib.dll`-ről készített saját másolatát, ugyanis ezt a kulcsszerelevényt a CLR automatikusan betölti minden egyes alkalmazástartományba. Ez felveti a kérdést: Hogyan lehet programozottan betölteni egy szerelevényt egy alkalmazástartományba? Egyyszerűen az `AppDomain.Load()` módszerrel (vagy az `AppDomain.ExecuteAssembly()` módszerrel), amely betölti és végrehajtja egy `*.exe` szerelvény `main()` módszerét).

Tételezzük fel, hogy átmásoltuk a `CarLibrary.dll`-t az `AppDomainManipulator.exe` alkalmazás könyvtárába; ekkor ezt a következők szerint tölthetjük be a `secondAppDomain` alkalmazástartományba:

```

static void main(string[] args)
{
    ...
    // CarLibrary.dll betöltése az új alkalmazástartományba.
    try
    {
        anotherAD.Load("CarLibrary");
        PrintAllAssembliesInAppDomain(anotherAD);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.ReadLine();
}
  
```

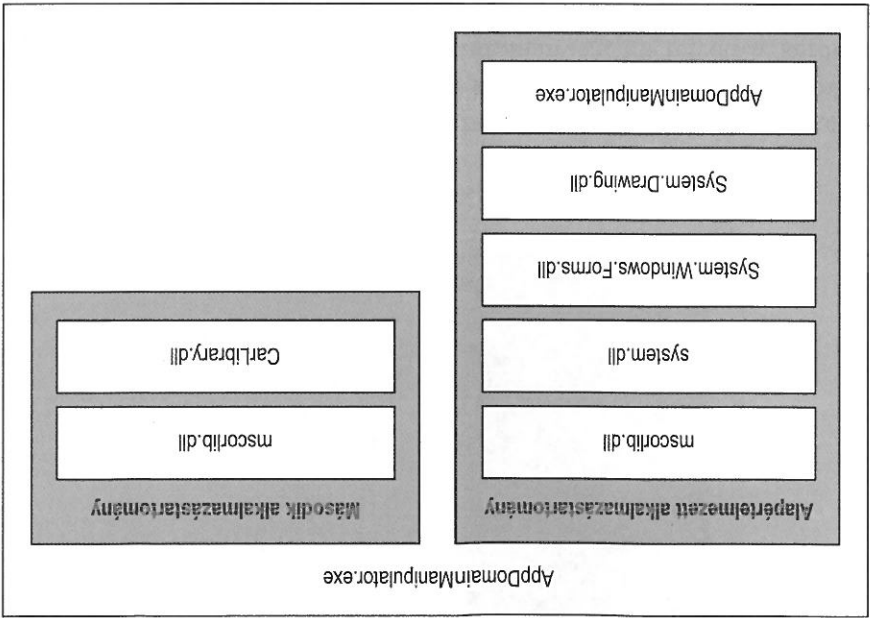
A CLR nem engedélyezi önálló .NET-szerelvények eltávolítását. Az AppDomain unload() módszer használataval azonban szelektíven eltávolíthatunk egy adott alkalmazástartományt a hosztolófolyamattól. Az alkalmazástartomány ilyenkor cserebe minden szerelvényt eltávolít.

Az AppDomain típus azoknak az eseményeknek a kis készletét határozza meg, amelyek egyike a domainunload. Ezt az eseményt akkor süttük el, amikor egy (nem alapértelmezett) alkalmazástartományt eltávolítunk az öt tartalmazó folyamattól. A másik érdekes esemény a processexit, amely akkor süti el, amikor az alapértelmezett alkalmazástartomány eltávolítódik a folyamattól (ez nyilvánvalóan magával hozza magának a folyamatnak a beárátását is).

Alkalmazástartományok programozott eltávolítása

Megjegyzés Ha hibakereső módban indítjuk az alkalmazást (az F5 lenyomásával), azt tapasztalhatjuk, hogy a Visual Studio hibakeresője által használt számos további szerelvény is betöltődik az alkalmazástartományba. A projekt futtatása (Ctrl + F5) egyes alkalmazástartományonként közvetlenül csak a szerelvényeket jeleníti meg.

17.8. ábra: Az AppDomainManipulator.exe folyamat belülről



A 17.8. ábra mutatja a most kialakított AppDomainManipulator.exe folyamat belső szerkezetét.

Ezért, ha programozottan szeretnénk eltávolítani az anotherad tartományt az AppDomainManager.pul ator.exe folyamatból, valamint szeretnénk értesítést kapni a társított alkalmazástartomány eltávolításáról, akkor a következő esemény-logikát írhatjuk:

```
static void Main(string[] args)
{
    // Feliratkozás DomainUnload eseményre.
    anotherad.DomainUnload +=
        new EventHandler(anotherad.DomainUnload);
    // anotherad eltávolítása.
    AppDomain.Unload(anotherad);
    Console.ReadLine();
}
```

A DomainUnload eseményhez System.EventHandler típusú módszerreferencia tartozik, ezért az anotherad.DomainUnload() paraméterlistája a következő:

```
static void anotherad_DomainUnload(object sender, EventArgs e)
{
    Console.WriteLine("***** unloaded anotherad! *****\n");
}
```

Ha szeretnénk értesítést kapni akkor, amikor az alapértelmezett alkalmazás-tartomány eltávolítódik, módosítsuk a Main() metódust, hogy kezelje az alapértelmezett alkalmazástartomány processzevent eseményét:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with AppDomain *****\n");
    AppDomain defaultAD = AppDomain.CurrentDomain;
    defaultAD.ProcessExit += new EventHandler(defaultAD.ProcessExit);
    ...
}

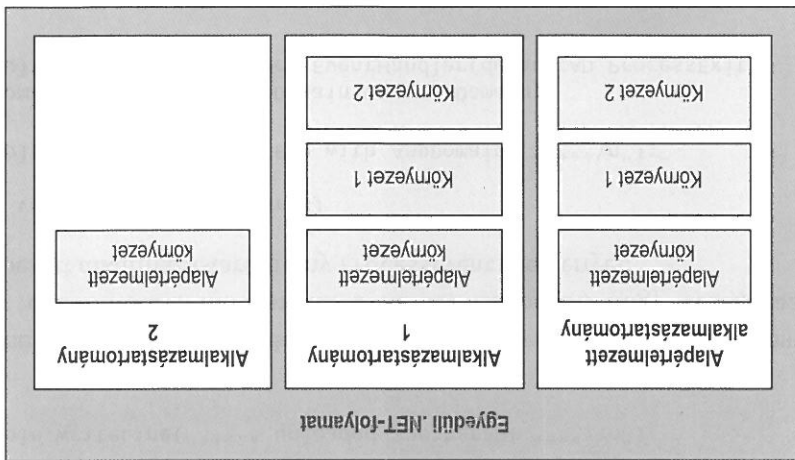
és definiáljunk egy megfelelő eseménykezelőt:
static void defaultAD_ProcessExit(object sender, EventArgs e)
{
    Console.WriteLine("***** unloaded defaultAD! *****\n");
}
```

Forráskód Az AppDomainManager.pul ator projekt megtalálható a 17. fejezet alkönyvtárában.

Az objektumkontextus határai

Az alkalmazástartományok egy folyamaton belüli logikai partíciók a .NET-szerű objektumkontextusra. Röviden: egy .NET-kontextus egyetlen alkalmazástartományhoz tartozik. Biztosít lehetőséget arra, hogy külön környezetet létesítsen egy adott objektumnak.

A kontextus használatával a CLR úgy képes biztosítani azt, hogy a speciális futásidőbeli követelményekkel bíró objektumok kezelése megfelelő és következetes legyen, hogy elfogja a metodushívásokat, amelyek adott kontextusból indulnak, és abba érkeznék. Ez az elfogórét lehetőséget tesz a CLR számára, hogy úgy állítsa be az aktuális metodushívást, hogy az megfelelően az adott objektum kontextusbeli beállításainak. Ha például egy olyan C#-osztályt definiálunk, amely automatikus szalibiztonságot követel meg (a [Synchronization] attribútum használatával), akkor a CLR a példányosítás során létrehoz egy „szinkronizált kontextust”.



17.9. ábra: Folyamatok, alkalmazástartományok és kontextushatárok

Ahogy egy folyamat meghatároz egy alapértelmezett alkalmazástartományt, úgy minden alkalmazástartományban van egy alapértelmezett kontextusa. Ez az alapértelmezett kontextus (amelyre néha úgy utalnak, hogy *context 0*, mivel mindig ez az alkalmazásban létrehozott első kontextus) csoportosítja azokat a .NET-objektumokat, amelyeknek nincs saját vagy egyedi kontextus-igényük. A .NET-objektumok nagy része a *context 0*-ba töltődik be. Ha a

CLR úgy dönt, hogy újonnan létrehozott objektumnak speciális igényei vannak, akkor új kontextushatárokat hoz létre a hosztoló alkalmazástartományon belül. A 17.9. ábra (lásd fenn) illusztrálja a folyamat/alkalmazástartomány/kontextus kapcsolatot.

Kontextusváltó és kötött típusok

Azokat a .NET-típusokat, amelyek nem igényelnek speciális kontextust, *kontextusváltó* objektumoknak nevezzük. Ezeket bárhonnan el lehet érni a hosztoló alkalmazástartományból anélkül, hogy zavart okoznának az objektumfutásidejű követelményeiben. A kontextusváltó objektumok létrehozása nem különösebben nehéz, hiszen voltaképpen semmit nem kell csinálni (pontosabban: nem cifrázzuk az objektumokat semmilyen kontextusattribútummal, és nem származtatjuk a `system.contextboundobject` osztályból):

```
// A kontextusváltó objektum a context 0-ba töltődik.
class Sportscar{
```

Másrésztől, azokat az objektumokat, amelyeknek kontextuslefoglalásra van szükségük, *kötött* objektumoknak nevezzük, és ezeknek a `system.contextboundobject` alaposztályból *kell* származniuk. Ez az alaposztály megőrösi azt, hogy a kérdéses objektum csak abban a kontextusban működhet megfelelően, amelyben létrehozták. A .NET-kontextus szerepe miatt, ha egy kötött objektum egy másik kontextusba kerül, az garantáltan valamilyen problémát fog okozni (természetesen a legrosszabbkor).

Amellett, hogy a `system.contextboundobject` típusból kell származnia, a kontextusérzékeny típus kiegészül a .NET-attribútumok egy speciális kategóriájával, amelynek kontextusattribútum a neve. Minden kontextusattribútum a `contextattributed` osztályból származik, amely a `system.runtime.remoting.contexts` névtérben található:

```
public class ContextAttribute :
    Attribute, IContextAttribute, IContextProperty
{
    public ContextAttribute(string name);
    public string Name { virtual get; }
    public object TypeId { virtual get; }
    public virtual bool Equals(object o);
    public virtual void
        Freeze(System.Runtime.Remoting.Contexts.Context newContext);
    public virtual int GetHashCode();
}
```

A [synchronization] attribútummal jelölt típusok szálbiztos kontextusba kerülnek. A myThreadSafeObject osztálytípus speciális kontextusigényeit figyelembe véve, képviseljük el, hogy mi történne akkor, ha egy lefoglalt objektumot szinkronizált kontextusból nem szinkronizált kontextusba helyeznénk át. Az objektum hirtelen nem lenne szálbiztos többé, és így az adatai hamar

```
// Ez a kötött típus csak szinkronizált,
// és emiatt szálbiztos kontextusba fog betöltődni.
[ synchronization ]
class SportsCars : ContextBoundObject
{ }
```

Tételezzük fel, hogy olyan osztályt szeretnénk megvalósítani (SportsCars), amely természeténél fogva szálbiztos, noha nem kódoltunk szálszinkronizációt a tagimplementációkba. Ehhez származtassuk a ContextBoundObject osztályból, és alkalmazzuk a [synchronization] attribútumot a következők szerint:

Kötött objektum meghatározása

Megjegyzés Nem részletezzük az egyedi objektumkontextusok építését; erről a Jason Bock és Tom Barnaby által jegyzett Applied .NET Attributes (Apress, 2003) című könyvben bővebben is olvashatunk.

Mivel a ContextAttribute osztály nincs lezárva, készíthetünk saját kontextus-attribútumot (amely egyszerűen a ContextAttribute osztályból származik, és felüldefiniálja a szükséges virtuális metódusokat). Ezután készíthetünk egy olyan szoftvereszközt, amely válaszol a kontextusbeállítási kérésokra.

```
public virtual void GetPropertiesForNewContext(
    System.Runtime.Remoting.Activation.
    IConstructionalMessage ctorMsg);
public Type GetType();
public virtual bool IsContextOK(
    System.Runtime.Remoting.Contexts.Context ctx,
    System.Runtime.Remoting.Activation.
    IConstructionalMessage ctorMsg);
public virtual bool IsDefaultAttribute();
public virtual bool IsNewContextOK(
    System.Runtime.Remoting.Contexts.Context newCtx);
public virtual bool Match(object obj);
public virtual string ToString(); }
```


inkonzisztenssé válhatnak, hiszen egyszerre több szál próbálna a most már nem szálbiztos objektum adatát manipulálni. Ahhoz, hogy biztosítsuk azt, hogy a CLR nem helyezi a sportscarts objektumokat szinkronizált kontextuson kívülre, egyszerűen származtatassuk a contextboundobject osztályból.

Egy objektum kontextusának vizsgálata

Noha nagyon kevés alkalmazást kell programozottan kapcsolni a kontextussal, álljon itt mégis egy szemléltető példa. Hozzunk létre egy új példarancsori alkalmazást contextmanipulator névvel. Ez az alkalmazás egy kontextusválogató osztályt (sportscar) és egy kötött típust (sportscarts) definiál:

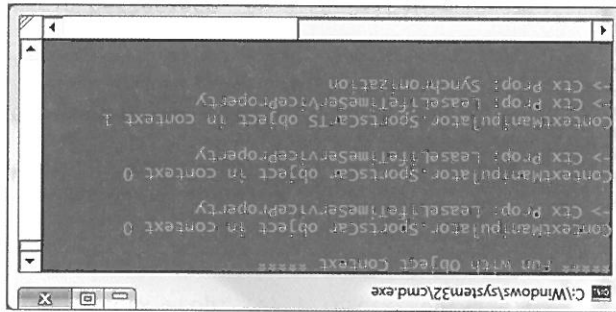
```
using System.Runtime.Remoting.Contexts; // For context type.
using System.Threading; // For thread type.

// A sportscar típusnak nincs speciális kontextusigénye,
// és ezért az alkalmazástartomány
// alapértelmezett kontextusába fog betölteni.
class Sportscar
{
    public Sportscar()
    {
        // context információ beszerzése, és a context ID kinyomtatása.
        context ctx = Thread.CurrentContext;
        Console.WriteLine("{0} object in context {1}",
            this.ToString(), ctx.ContextID);
        foreach (IContextProperty ctxProp in ctx.ContextProperties)
            Console.WriteLine("-> ctx Prop: {0}", ctxProp.Name);
    }
}

// sportscarts típus megköveteli, hogy
// szinkronizált kontextusba töltsön.
class Sportscarts : ContextBoundObject
{
    public Sportscarts()
    {
        // context információ beszerzése, és a context ID kinyomtatása.
        context ctx = Thread.CurrentContext;
        Console.WriteLine("{0} object in context {1}",
            this.ToString(), ctx.ContextID);
        foreach (IContextProperty ctxProp in ctx.ContextProperties)
            Console.WriteLine("-> ctx Prop: {0}", ctxProp.Name);
    }
}
```

Mivel a sportcar osztály nincs ellátva kontextusattribútummal, a CLR a sport és sport2 elemeket a context 0-ba foglalta le (vagyis az alapértelmezett környezetbe).

17.10. ábra: Egy objektum kontextusának vizsgálata



Amint az objektum létrejön, az osztálykonstruktorok kiírják a kontextuscentrikus információk különböző részeit (lásd a 17.10. ábrát).

```
static void main(string[] args)
{
    Console.WriteLine("***** Fun with Object Context *****\n");
    // Az objektumok a létrehozás során kontextusinformációkat
    // jelenítenek meg.
    Sportcar sport = new Sportcar();
    Console.WriteLine();
    Sportcar sport2 = new Sportcar();
    Console.WriteLine();
    SportsCarTS synchrosport = new SportsCarTS();
    Console.WriteLine();
}
```

Minden konstruktor megszerez egy context típust az aktuálisan futó számtól a statikus Thread.CurrentContext tulajdonság révén. A context objektum használatával kiteríthetjük a kontextushatár statisztikáit, például a hozzárendelt azonosítót, valamint a context.ContextProperties révén megszerzett leírók gyűjteményét. Ez a tulajdonság egy olyan objektumot ad vissza, amely implementálja az IContextProperty interfészt, amely a Name tulajdonság révén feltárja az egyes leírókat. Módosítsuk a main() metódust úgy, hogy minden osztálypéldány egy példányt foglalt le:

A sportsCarTs objektum azonban egyedül kontextusúthatárok közé került (amely az 1-es kontextuszazonosítót kapta), hiszen ez a kötött típus rendelkezik a [synchronization] attribútummal.

Forráskód A ContextManiPulátor projekt megtalálható a 17. fejezet alkönyvtárában.

A folyamat, alkalmazástartomány és a kontextus összefoglalása

A kulcspontok a CLR .NET-szerelvényének hosztolásához a következőképpen foglaltak össze:

- Egy .NET-folyamat egy vagy több alkalmazástartományt hosztolhat. Minden alkalmazástartomány képes bármennyi kapcsolódó .NET-szerelvényt hosztolni. A CLR egy mástól függetlenül betöltheti és el távolíthatja az alkalmazástartományokat (akár programozottan, a system.AppDomain típus révén).
- Egy adott alkalmazástartomány egy vagy több kontextusból áll. A kontextus használataival a CLR képes elhelyezni egy „speciális igényű” objektumot egy logikai tárolóba, hogy biztosítsa annak futásidőű követelményeit.

Az esetek többségében a .NET-futtatórendszer automatikusan kezeli a folyamatok, az alkalmazástartományok és a kontextusok részleteit. Ezek az információk szilárd alapot biztosítanak a többszálú programozás megértéséhez a .NET platformban.

Összefoglalás

Ebben a fejezetben megvizsgáltuk, hogy egy .NET végrehajtható fájl kód hosztolása hogyan történik a .NET platformon. Hogy a processzsek kezelése alkalmazkodjon a CLR igényeihez, a Win32 sokáig uralkodó koncepciója megváltozott.

Egy folyamat több alkalmazástartományt is hosztolhat, amelyek mind-egyike képes bármennyi kapcsolódó szerelvény hosztolására és végrehajtására. Továbbá egy szimpla alkalmazástartomány bármennyi kontextushatárt is tartalmazhat. Ezt a további izolációs szintet használva, a CLR képes a speciális igényű objektumok helyes kezelését biztosítani.

TIZENNYOLCADIK FEJEZET

Többcsálú alkalmazások létrehozása

Az előző fejezetben a folyamatok, az alkalmazástartományok és a kontextusok közötti kapcsolatot vizsgáltuk meg. Ebben a fejezetben azzal foglalkozunk, hogy a .NET platform hogyan teszi lehetővé többcsálú alkalmazások készítését, és hogyan tarthatjuk szálbiztosan a megosztott erőforrásokat. Újra megvizsgáljuk a .NET-metódusreferencia típusát, majd megnézzük, hogy valójában hogyan támogatja az aszinkron függvényhívásokat. Ezzel a módszerrel automatikusan meghívhatunk egy metódust egy másodlagos végrehajtási szálon. Ezután a `system.threading` névtérben található típusokat tanulmányozzuk. Számos olyan típust (`thread`, `threadstart` stb.) megvizsgálunk, amelyekkel könnyedén létrehozhatunk további végrehajtási szálakat. Ezenfelül megismerkedünk a `backgroundworker` típus használatával is, amely leegyszerűsíti a GUI-alapú alkalmazáskörnyezetben futó háttérműveletek végrehajtását. A többcsálú fejlesztés bonyolultsága persze nem a szálak készítésében rejlik, hanem annak a biztosításában, hogy a kód bármikor jól fel van-e szerelve a megosztott erőforrások parhuzamos hozzáféréseinek a kezeléséhez. Így a fejezet a .NET-keretrendszer különféle szinkronizációs primitívjeit is bemutatja.

A folyamat, az alkalmazástartomány, a kontextus és a szál kapcsolata

Az előzőekben a *szálat* egy végrehajtható alkalmazás futási útvonalaként definiáltuk. Sok .NET-alkalmazás egyszerű, ám egy szereplővel elsődleges szála (a CLR hozza létre a `main()` lefutásakor) másodlagos végrehajtási szálakat hozhat létre további munkaegységek készítésére. A kiegészítő szálak megvalósításával könnyebben kezelhető (de az egymagos gépeken nem feltétlenül gyorsabb) alkalmazásokat építhetünk fel.

Megjegyzés Manapság az új számítógépek gyakran többmagos processzorral (de legalábbis hiperszálas egymagos processzorral) vannak felszerelve. A több szál párhuzamos futása nélkül a fejlesztők nem tudnák kihasználni a többmagos gépek maximális teljesítményét.

A `System.Threading` névtér olyan típusokat tartalmaz, amelyek segítségével többszálú alkalmazásokat készíthetünk. A `Thread` osztály az alap típus, hiszen az egy adott szálát reprezentál. Ha programozottan szeretnénk hivatkozni egy adott tagot éppen végrehajtó szálra, hívjuk meg a statikus `Thread.CurrentThread` tulajdonságot:

```
static void ExecuteCuttingThread()
{
    // megkapjuk a metódust éppen
    // futató szálát.
    Thread currThread = Thread.CurrentThread;
}
```

A .NET platformon *nem* közvetlen egy-egy kapcsolat van az alkalmazástartományok és a szálak között. Egy adott alkalmazástartományban valójában jó néhány szál lefuthat bármely időpontban. Továbbá egy adott szál élete során nincs egyetlen alkalmazástartományra korlátozva. A szálak szabadon átléphetik az alkalmazástartományok határait, ahogy azt a Win32-szálútervező és a CLR jónak látja.

Habár az aktív szálakat át lehet mozgatni az alkalmazástartományok határain, egy adott szál bármely időpillanatban csak egyetlen alkalmazástartományon belül futhat (más szóval, lehetetlen, hogy egy szál egy időben több alkalmazástartományban is működjön). Ha programozottan szeretnénk elérni az aktuális szálát hosztoló alkalmazástartományt, hívjuk a statikus `Thread.GetDomain()` metódust:

```
static void ExtractAppDomainInHostThread()
{
    // megszerezzük az aktuális szálát hosztoló alkalmazástartományt.
    AppDomain ad = Thread.GetDomain();
}
```

Egy szálát bármikor át is helyezhetünk egy adott környezetbe, de a CLR is átmozgathatja egy új környezetbe. Ha meg akarjuk kapni azt az aktuális környezetet, amelyikben a szál éppen fut, használjuk a statikus `Thread.CurrentContext` tulajdonságot:

A CLR-nek az a feladata, hogy a szálakat alkalmazástartományokba és környezetekbe helyezze el (illetve vegye ki onnan). .NET-fejlesztőként szerencsére nem kell számon tartanunk, hogy egy szál hol fut (illetve, hogy mikor vált alkalmazástartományt). Ettől függetlenül meg kell ismernünk a mögöttes primitívek elérésének különféle módjait.

A konkurencia problémája

A többszálú programozás egyik kellőmértékben tudjuk befolysolni azt, hogy a mögöttes operációs rendszer vagy a CLR hogyan használja a szálakat. Ha például egy olyan kódblokkot készítünk, amely új végrehajtási szálhoz létre, nem garantálható, hogy a szál azonnal elindul. A kód csak arra utasítja az operációs rendszert, hogy amint lehet, futtasson a szálakat (ez általában akkor történik meg, amikor a szálütemező hozzáfog). Továbbá, mivel a szálak a CLR igényeinek megfelelően áthelyezhetők az alkalmazások és a környezetek határain, figyelniünk kell arra, hogy az alkalmazásunk mely szempontból *nem szálbiztos* (pl. többszálú hozzáférések alávett), és mely műveletek *szálilekony műveletek* (a szálilekony műveletek a veszélyesek). Tétellezzük fel például, hogy egy szál bizonyos objektum metódusát hívja. A szálütemező azonban a tevékenység felfüggesztésére utasítja ezt a szálakat azért, hogy egy másik szál ugyanennek az objektumnak az azonos metódusát elérhesse. Ha az eredeti szál nem fejezte be teljesén a műveletet, a második bemeneti szál egy részben módosított objektummal találkozhat. Ekkor a második szál alapvetően hamis adatot olvas, ez pedig furcsa (és nehezen megtalálható) hibához vezet, amelyeket még nehezebb reprodukálni és megkeresni. Másrészt, az atomi műveletek többszálú környezetben mindig biztonságosak. Sajnos nagyon kevés olyan művelet van a .NET-alapoztálykönyvtárakban, amely garantáltan atomi. Még az értékek tagváltozókhoz rendelése sem atomi. Amíg a .NET-keretrendszer 3.5 SDK dokumentációja kifejezetten azt nem mondja, hogy egy művelet atomi, azt kell feltételeznünk, hogy szálilekony, és így óvintézkedéseket kell tennünk.

```
static void ExtractCurrentThreadContext()
{
    // Szerezünk meg azt a környezetet, amelyben
    // az aktuális szál működik.
    context ctx = Thread.CurrentContext;
}
```

A folyamat, az alkalmazástartomány, a kontextus és a szál kapcsolata

A szálszinkronizáció szerepe

A többszálú alkalmazástartományok önmagukban nem szálbiztosak, ugyanis (többé-kevésbé) egy időben több szál is működhet egy szerepben. Azért, hogy az alkalmazás erőforrásait a lehetséges adatszerűléstől megóvják, a .NET-fejlesztőknek szálkezelési primitíveket (pl. zárolások [lock], megfigyelők [monitors] és a [synchronisation] attribútum) kell igénybe venniük a végrehajtási szálakhoz való hozzáférés vezérléséhez.

Jóllehet a .NET platform nem képes teljes mértékben megszüntetni a nagy teljesítményű, többszálú alkalmazások létrehozásának nehézségeit, a folyamatot nagymértékben leegyszerűsíti. A `system.threading` névtérben definiált típusok használatával különösebb nehézségek nélkül hozhatunk létre további szálakat. Ugyanígy, amikor zároljuk az adatok megosztott pontjait, találunk olyan kiegészítő típusokat, amelyek ugyanazt a működést biztosítják, mint a Win32 API szálkezelési primitívei (amelyek jóval tisztább objektummodellt alkalmaznak).

Nem csak a `system.threading` névtér segítségével készíthetünk azonban többszálú .NET-programokat. A .NET-metódusreferencia vizsgálatakor (lásd 11. fejezet) már láttuk, hogy minden metódusreferencia képes a tagok aszinkron hívására. Ez az egyik fő előnye a .NET platformnak, hiszen a szálak létrehozásának az egyik leggyakoribb célja az, hogy nem blokkoló (aszinkron) módon hívassuk a metódusokat. Habár a `system.threading` névtérrel hasonló eredménnyre jutunk, a metódusreferencia az egész folyamatot megkönnyíti.

A .NET-metódusreferenciák rövid áttekintése

A .NET-metódusreferencia típusa lényegében egy típusbiztos, objektumorientált függvénymutató. A .NET-metódusreferenciák deklarálásakor a `C#-for-díto` úgy válaszol, hogy létrehoz egy, a `system.multicastdelegate`-ből (amely a `system.delegate`-ből származik) származó, lezárt osztályt. Ezek az osztályok biztosítják a metódusreferenciáknak, hogy metóduscsimlistát tarthassanak fenn a később hívható metódusokról. Nézzük meg a 11. fejezetben definiált `BinaryOp` metódusreferenciát:

```
// Egy C# metódusreferencia-típus.  
public delegate int BinaryOp(int x, int y);
```


Definíciója szerint a `BinaryOp` bármely olyan metódusra rá tud mutatni, amely vesz két egész számot (érték szerint) paraméterként, és egész számot ad vissza. Ha egyszer lefordítottuk, a definiáló szerelvény egy teljesen kiakartított osztálydefiniációt fog tartalmazni, amelyet a metódusreferencia deklarációja alapján dinamikus generálódott. A `BinaryOp` esetében ez az osztály többnyire a következőképpen néz ki (pszeudokód):

```
public sealed class BinaryOp : System.MulticastDelegate
{
    public BinaryOp(object target, int functionAddress);
    public void Invoke(int x, int y);
    public IAsyncResult BeginInvoke(int x, int y,
        AsyncCallback cb, object state);
    public int EndInvoke(IAsyncResult result);
}
```

A `generalInvoke()` metódust a metódusreferencia-objektum metódusainak szinkron hívására használjuk. Ezért a hívószálmak (pl. az alkalmazás elsődleges szála) meg kell várnia, hogy a metódusreferencia meghívása befejeződjön. A C#-ban az `Invoke()` metódust nem szükséges közvetlenül kóddal meghívni, hanem közvetett módon is ellindítható a háttérben, normál metódushívó szintaxissal. Nézzük meg a következő konzolprogramot (`syncDelegateReview`), amely a statikus `Add()` metódust hívja, szinkron (blokkoló) módon:

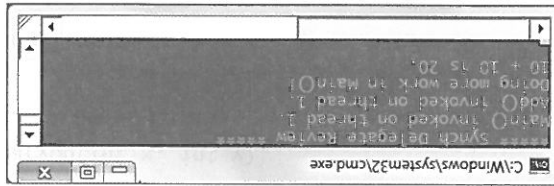
```
// Szükség van erre a Thread.Sleep() meghíváshoz.
using System.Threading;
using System;

namespace SyncDelegate
{
    public delegate int BinaryOp(int x, int y);

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Sync Delegate Review *****");
            // A futó szál azonosítójának kitérása.
            Console.WriteLine("main() invoked on thread {0}.",
                Thread.CurrentThread.ManagedThreadId);
            // Az Add() metódus aktivizálása szinkron módon.
            BinaryOp b = new BinaryOp(Add);
            // Lehetne írni azt is, hogy b.Invoke(10, 10);
            int answer = b(10, 10);
        }
    }
}
```

A program futtatásakor láthatjuk, hogy a `console.WriteLine()` logika `main()` végrehajtása öt másodpercet késik. Habár sok (ha nem a legtöbb) metódust problémamentesen hívhatunk szinkron módon, a `NFT`-metódusreferenciákat utasíthatjuk, hogy ha szükséges, aszinkron módon hívják meg a metódusait.

18.1. ábra: Metódusok szinkron (blokkoló) hívása



Ezután a `main` metódus lekérdezi az aktuális szálat (a `thread.current-thread` révén), és a `managedthread` tulajdonság segítségével kírja a szál azonosítóját. Ugyanez a logika ismétlődik meg a statikus `add()` metódusban. Mivel csak az elsődleges szál dolgozik az alkalmazásban, a konzolon ugyanaz az azonosítóérték fog megjelenni (lásd 18.1. ábra).

Először is ez a program a `system.threading` névtérben keresztül éri el a `thread` típust. Az `add()` metóduson belül meghívjuk a statikus `thread.sleep()` metódust a hívószál kb. öt másodperces felfüggesztésére, hogy egy hosszadalmas tevékenységet szimuláljunk. Mivel az `add()` metódust *szinkron* módon hívjuk, a `main()` metódus addig nem írja ki a művelet eredményét, amíg az `add()` metódus be nem fejeződik.

```
// Ezek a sorok nem hajtodnak végre, amíg
// az add() metódus be nem fejeződik.
console.WriteLine("Doing more work in main()");
console.WriteLine("10 + 10 is {0}.", answer);
console.ReadLine();
}

static int add(int x, int y)
{
    // A futó szál azonosítójának kírása.
    console.WriteLine("add() invoked on thread {0}.",
        thread.CurrentThread.ManagedThreadId);
    // Szünet egy hosszú művelet szimulálásra.
    thread.Sleep(5000);
    return x + y;
}
}
```

A metódusreferenciák aszinkron viselkedése

Mitől is szól pontosan az *aszinkron* függvényhívás? Egyes programozási műveletek időbe telnek. Képzeljük el, hogy egy olyan egyszerű alkalmazást készítsünk, amely távoli objektum metódusát hívja meg, sokáig futó adatbázis-lekérdezést hajt végre, letölt egy nagyméretű dokumentumot, vagy egy 500 soros szöveget ír egy külső fájlba. A műveletek végrehajtása alatt kis időig úgy fog tűnni, hogy az alkalmazás lefagyott. Az aktuális feladat feldolgozásáig a program egyéb dolgokra (pl. a menü aktiválása, eszköztárra kattintás vagy konzolos megjelenítés) nem válaszol.

Ezért a kérdés az, hogy hogyan utasítsuk a metódusreferenciát egy metódus külön végrehajtási szálon történő hívására, hogy több, egy időben lefutó feladatot szimuláljunk. Minden .NET metódusreferencia-típus automatikusan rendelkezik ezzel a képességgel. És ehhez *nem* szükséges közvetlenül ismerni a `System.Threading` névtér részleteit sem.

A `BeginInvoke()` és az `EndInvoke()` metódusok

Amikor a C#-fordító feldolgozza a delegate kulcsszót, a dinamikusan generált osztály definiálja a `BeginInvoke()` és az `EndInvoke()` metódusokat. A `BeginInvoke()` metódusreferencia alapján az előbbi metódusok prototípusa a következő:

```
public sealed class Binaryop : System.MulticastDelegate
{
    ...
    // metódus aszinkron hívására használatos.
    public IAsyncResult BeginInvoke(int x, int y,
        AsyncCallback cb, object state);
    // megszerzi az aktivizált metódus
    // visszatérési értékét.
    public int EndInvoke(IAsyncResult result);
}
```

A `beginInvoke()` metódusba elsőként beadott paraméterlista a `C#`-metódusrencia formátumán alapul (a `binaryop` esetében ez két egész szám). A két utolsó argumentum mindig a `system.AsyncCallback` és a `system.Object`. Hamarosan tanulmányozni fogjuk ezeknek a paramétereknek a szerepét; addig azonban állítsuk nulla mindkettőt. Az `endInvoke()` visszaadott értéke a `binaryop` definíciója alapján egy egész szám, paraméterre pedig az `IAsyncResult` típus.

A `System.IAsyncResult` interfész

A `beginInvoke()` metódus mindig az `IAsyncResult` interfészt megvalósító objektumot adja vissza, míg az `endInvoke()` egyedüli paraméterként egy `IAsyncResult`-kompatibilis típust igényel. A `beginInvoke()` metódusból visszatérő `IAsyncResult`-kompatibilis objektum alapján véve egy társító mechanizmus, amely lehetővé teszi, hogy a hívó szál az `endInvoke()` révén később elérje az aszinkron függvényhívás eredményét. A `system` névtérben definiált `IAsyncResult` interfész a következőképpen néz ki:

```
public interface IAsyncResult
{
    object AsyncState { get; }
    WaitHandle AsyncWaitHandle { get; }
    bool CompletedSynchronously { get; }
    bool IsCompleted { get; }
}
```

A legegyszerűbb eset az, amikor el tudjuk kerülni ezeknek a tagoknak a közvetlen meghívását. Csak annyit kell tennünk, hogy eltaroljuk a `beginInvoke()` metódusból visszatérő `IAsyncResult`-kompatibilis objektumot, majd továbbíthetjük az `endInvoke()` metódusba, amikor készen állunk a függvényhívás eredményének az eléréséhez. Ha pontosabban szeretnénk szabályozni egy vissza-térési érték kiolvasását, lehetőségünk van erre az `IAsyncResult`-kompatibilis objektum tagjainak a meghívásával.

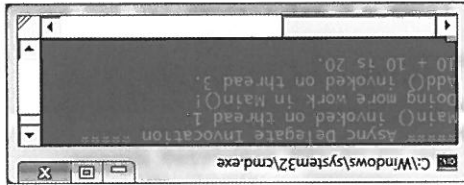
Megjegyzés Ha aszinkron módon hívunk egy `void` értéket visszaadó metódust, egyszerűen „tedd meg, és lépj tovább”. Ilyen esetekben soha nincs szükség az `IAsyncResult`-kompatibilis objektum gyorsítottárazására vagy az `endInvoke()` első helyen történő hívására (mivel nincs visszakerekeshető érték).

Metódushívás aszinkron módon

Ahhoz, hogy utasítsuk a `BinaryOp` metódusreferenciát az `Add()` aszinkron meghívására, módosítsuk az előző `Main()` metódust az alábbiak szerint:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Async Delegate Invocation *****");
    // A futó szál azonosítójának kitírása.
    Console.WriteLine("Main() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
    // Az Add() meghívása egy másodlagos szálon.
    BinaryOp b = new BinaryOp(Add);
    TaskAsyncResult tftar = b.BeginInvoke(10, 10, null, null);
    // A többi feladatot az elsődleges szálon végezzük el...
    Console.WriteLine("Doing more work in Main()!");
    // Amikor kész, szerezzük meg
    // az Add() metódus eredményét.
    int answer = b.EndInvoke(tftar);
    Console.WriteLine("10 + 10 is {0}.", answer);
    Console.ReadLine();
}
```

Ha lefuttatjuk ezt az alkalmazást, megfigyelhetjük, hogy két egyedi szálazo-nosító jelenik meg, hiszen az aktuális alkalmazástartományban valójában több szál működik (lásd 18.2. ábra).



18.2. ábra: Aszinkron módon hívott metódusok

Az alkalmazás futása közben az egyedi azonosítókön kívül az is megfigyelhető, hogy a „Doing more work in Main()!” üzenet azonnal megjelenik, miután a másodlagos szál a feladatával van elfoglalva.

A hívó szál szinkronizálása

A `main()` aktuális megvalósításán elgondolkodva, bizonyára nyilvánvalóvá vált, hogy a `beginInvoke()` és az `endInvoke()` hívása között eltelt idő kevesebb, mint öt másodperc. Így, miután a „Doing more work in Main()!” megjelent a konzolon, a hívó szál blokkolódik, és várja, hogy a másodlagos szál befejeződjön, majd hozzájárulhasson az `add()` módszer eredményéhez. Ezért teljesen még egy *szinkron hívást* csinálunk:

```
static void Main(string[] args)
{
    ...
    BinaryOp b = new BinaryOp(Add);
    AsyncResult r = b.BeginInvoke(10, 10, null, null);

    // Ez a hívás jóval kevesebb ideig tart, mint öt másodperc!
    Console.WriteLine("Doing more work in Main()!");

    // A hívó szál most blokkolva van, amíg
    // a BeginInvoke() be nem fejeződik.
    int answer = b.EndInvoke(r);
    ...
}
```

Az aszinkron módszerreferenciák nyilvánvalóan elvesztenék az előnyüket, ha a hívó szál bizonyos körülmények között blokkolódna. Az `AsyncResult` interfész az `IsCompleted` tulajdonsága révén teszi lehetővé a hívó szál számára, hogy ellenőrizze az aszinkron módon hívott módszer feladatának a teljesülését. Ennek a tagnak a segítségével a hívó szál meg tudja határozni, hogy az aszinkron hívás valóban befejeződött-e az `EndInvoke()` hívása előtt. Ha a módszer nem fejeződött be, az `IsCompleted` értéke hamis, a hívó szál pedig folytathatja a feladatát. Ha az `IsCompleted` értéke igaz, a hívó szál „a legkevesebb blokkoló módon” érte el az eredményeket. Gondoljunk végig a `main()` módszer módosítását:

```
static void Main(string[] args)
{
    ...
    BinaryOp b = new BinaryOp(Add);
    AsyncResult r = b.BeginInvoke(10, 10, null, null);

    // Ez az üzenet folyamatosan kitródik, amíg
    // az Add() módszer be nem fejeződik.
    while(!r.IsCompleted)
    {
        Console.WriteLine("Doing more work in Main()!");
        Thread.Sleep(1000);
    }
}
```

getConstructors()	Ezek a módszerek teszik lehetővé, hogy hozzáférjünk a számmunkra érdekes elemeket (interfész, metódus, tulajdonság stb.) bemutató tömbhöz. Mindegyik metódus a kapcsolódó tömböt adja vissza (pl. a getFields() metódus a fieldInfo tömböt, a getMethods() a methodInfo tömböt). Ezeknek a metódusoknak létezik egyes számuformája (például getMethod(), getProperty() stb.) is, amely azt teszi lehetővé, hogy név szerint keressünk ki egy elemet az összes elem leképezése helyett.
findMembers()	Ez a metódus a keresési feltételek alapján memberInfo típusok tömbjét adja vissza.
getType()	Ez a statikus metódus visszaadja a megadott típusnévhez tartozó Type példányt.
invokeMember()	Ez a metódus biztosítja az adott elemhez történő késői költését.

16.2. táblázat: A System.Type osztály néhány tagja

Típusreferencia megszerzése a System.Object.GetType() segítségével

A Type osztály egy példányt többféleképpen is megszerezhetjük. Mivel a Type absztrakt osztály, az egyetlen dolog, amely nem működik, ha a new kulcsszóval közvetlenül létrehozzuk a Type objektumot. A System.Object viszont definiálja a GetType() metódust, amely az aktuális objektum metaadatát megjelenítő Type osztálypéldányt adja vissza:

```
// Típusinformáció megszerzése egy Sportscar példány használatával.
Sportscar sc = new Sportscar();
Type t = sc.GetType();
```

Ez a megközelítés nyilvánvalóan csak akkor működik, ha a típusról (ebben az esetben a sportscar-ról) fordítási idejű információkkal rendelkezünk, és a típus egy példánya pillanatnyilag a memóriában található. Ezzel a megszorítással érthetővé válik, hogy az olyan eszközök, mint az ilдам.exe, a típusinformációkhoz nem jutnak hozzá úgy, hogy a System.Object.GetType() metódust közvetlenül meghívják minden egyes típushoz, hiszen az ilдам.exe fordítása nem az egyedi szerezvények használatával történt.

Típusreferencia megszerzése a System.Type.GetType() segítségével

Egyszerűbben hozzáférhetünk a típusinformációkhoz, ha a `System.Type` osztály statikus `GetType()` tagját hívjuk, és megadjuk a vizsgálni kívánt típus teljes meghatározott nevét. Ezzel a módszerrel *nincs* szükség arra, hogy fordítási idejű információkkal rendelkezünk arról a típusról, amelynek metaadattát vizsgáljuk, hiszen a `Type.GetType()` módszer a mindenütt jelenlevő `System.String` osztály példányát várja paraméterként.

Megjegyzés Amikor azt állítjuk, hogy a `Type.GetType()` hívása során nincs szükségünk fordítási idejű ismeretekre, arra utalunk, hogy (erősen típusos változó helyett) a módszer bármilyen stringértéket kaphat. A típus stringformátumú nevét természetesen ismernünk kell.

A `Type.GetType()` módszer túlterhelése lehetővé teszi, hogy meghatározzunk két logikai paramétert, amelyek egyike azt szabályozza, hogy a rendszer kivétel dobjon, ha nem találja a típust, a másik pedig a string kis- és nagybetűs érzékenységet állítja be. Ehhez nézzük meg a következő kódot:

```
// Típusinformáció megszerzése a statikus Type.GetType() módszerrel
// használataival (nem dob kivételt, ha nem talál Sportscar típust,
// valamint figyelmen kívül hagyja a kis- és nagybetűket).
Type t = Type.GetType("CarLibary.Sportscar", false, true);
```

A `GetType()` módszernek átadott string nem említi azt a szerelvényt, amely magában foglalja a típust. Ekkor az az alapértelmezett feltevés, hogy az ép-pén végrehajtás alatt álló szerelvény definiálja a típust. Ha azonban kívülső privát szerelvény metaadathoz szeretnénk hozzáférni, a stringparamétert a típus teljesen meghatározott nevével, valamint a típust tartalmazó szerelvény barátságos nevével kell megadni (a neveket vessző választja el egymástól):

```
// Típusinformáció megszerzése egy külső szerelvény típusáról.
Type t = Type.GetType("CarLibary.Sportscar, CarLibary");
```

A `Type.GetType()` módszernek átadott string pluszjel token (+) tartalmazhat, amellyel a *beágyazott típust* jelöljük. Tétellezzük fel, hogy a `JamesBondCar` osztályba ágyazott felsorolt típus (`spyoptions`) típusinformációhoz szeretnénk hozzájutni. Ehhez a következő kód szükséges:

```
// Típusinformáció megszerzése egy olyan beágyazott felsorolt
// típushoz, amely az aktuális szerelvényben található.
Type t = Type.GetType("CarLibary.JamesBondCar+spyoptions");
```


Típusreferencia készítése a typeof() segítségével

A típusinformációk megszerzésének utolsó lehetséges módja a C# typeof operátorának az alkalmazása:

```
// Típus megszerzése a typeof használatával.  
Type t = typeof(Sportscar);
```

A Type.GetType() módszerhez hasonlóan, ha a typeof operátorral szeretnénk megkapni a típusinformációkat, nem kell létrehozunk objektum példányt. Az alapködnak azonban fordítási idejű ismeretekkel kell rendelkeznie a vizsgálni kívánt típusról, ugyanis a typeof operátornak a szöveges reprezentáció helyett a típus erősen típusált neve kell.

Egyedi metaadat-nézegető létrehozása

A reflexió alapvető folyamatainak (és a System.Type osztály lehetőségeinek) bemutatásához hozzuk létre a MyTypeVierer parametrsorral alkalmazást. A program megjeleníti az mscorlib.dll-ben (a .NET-alkalmazások automatikusan hozzáférnek ehhez a kulcsfontosságú keretrendszer-osztálykönyvtárhoz), illetve a MyTypeVierer alkalmazásban definiált típusok metódusait, tulajdonságait, mezőit és az itt található interfészek (továbbá egyéb érdekes összetevők) részleteit. Az alkalmazás létrehozása után mindenképpen importáljuk a System.Reflection névteret:

```
using System.Reflection;
```

A metódusok reflexiója

A program osztályt számos olyan statikus metódussal bővíti, amelyek közül mindegyik egyetlen System.Type paramétert vár, és a visszatérési értéke void. Előként nézzük meg a ListMethods() metódust, amely a bejövő típus által definiált metódusok nevét listázza. A Type.GetMethods() a System.Reflection.MethodInfo típusok tömbjét adja vissza:

```

    }
    Console.WriteLine();
    Console.WriteLine(">{0}", prop.Name);
    foreach(PropertyInfo prop in pi)
        PropertyInfo[] pi = t.GetProperties();
    Console.WriteLine("***** Properties *****");
}
static void ListProps(Type t)
// Megjeleníti a típus tulajdonságainak a nevét.

```

A típus tulajdonságait megjelenítő logika hasonló:

```

    }
    Console.WriteLine();
    Console.WriteLine(">{0}", field.Name);
    foreach(FieldInfo fi in fi)
        FieldInfo[] fi = t.GetFields();
    Console.WriteLine("***** Fields *****");
}
static void ListFields(Type t)
// Megjeleníti a típus mezőinek a nevét.

```

zök nevét listázzuk:

A `ListFields()` módszer megvalósítása hasonló a fentiekhez. Az egyetlen figyelmeztetés az, hogy a `GetType()` módszer hívása, valamint az eredményként kapott `FieldInfo` tömb. Az egyszerűség kedvéért csak a me-

Mezők és tulajdonságok reflexiója

A példában a `MethodInfo.Name` tulajdonság segítségével kiírjuk a módszer nevét. A módszer további tagokkal rendelkezik, s ezekkel meghatározhatjuk azt, hogy a módszer statikus, virtuális, generikus vagy absztrakt legyen. A `MethodInfo` típus lehetővé teszi azt is, hogy hozzáférjünk a módszer visszatérési értékéhez és paraméterkészletéhez.

```

    }
    Console.WriteLine();
    Console.WriteLine(">{0}", m.Name);
    foreach(MethodInfo mi in mi)
        MethodInfo[] mi = t.GetMethods();
    Console.WriteLine("***** Methods *****");
}
static void ListMethods(Type t)
// Megjeleníti a típus módszereinek nevét.

```

A típus által megvalósított interfészek reflexiója

A következő példában elkészítjük a `ListInterfaces()` metódust, amely a bejövő típus által támogatott interfészek nevét listázza. Az egyetlen igazán érdekes pont az, hogy a `getInterfaces()` a `System.Type` osztályok tömbjét adja vissza. Ez azért nem meglepő, mert az interfészek valójában típusok:

```
// A megvalósított interfészek megjelenítése.
static void ListInterfaces(Type t)
{
    Console.WriteLine("***** Interfaces *****");
    Type[] ifaces = t.GetInterfaces();
    foreach(Type i in ifaces)
        Console.WriteLine("<->{0}", i.Name);
}
```

Egyéb érdekességek megjelenítése

Végül, de nem utolsósorban, rendelkezésünkre áll egy olyan segédmetódus, amely a bejövő típusról különböző statisztikákat jelenít meg (ezek azt jelzik, hogy a típus generikus-e, lezár-e, mi a típus alaposztálya, és így tovább):

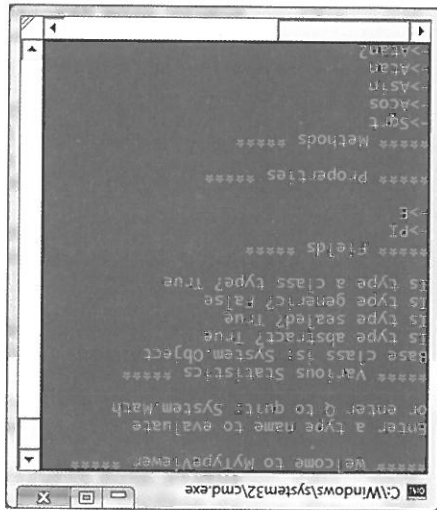
```
// Pusztán ráadásként.
static void ListVariousStats(Type t)
{
    Console.WriteLine("***** Various Statistics *****");
    Console.WriteLine("Base class is: {0}", t.BaseType);
    Console.WriteLine("Is type abstract? {0}", t.IsAbstract);
    Console.WriteLine("Is type sealed? {0}", t.IsSealed);
    Console.WriteLine("Is type generic? {0}", t.IsGenericType);
    Console.WriteLine("Is type generic? {0}", t.IsGenericType);
    Console.WriteLine("Is type a class type? {0}", t.IsClass);
    Console.WriteLine();
}
```

A Main() metódus megvalósítása

A program osztály `Main()` metódusa a típus teljesen meghatározott nevét kéri a felhasználótól. Mivel ezt a sztringadat a bitokunkban van, átadjuk a `Type.GetType()` metódusnak, a kinyert `System.Type` osztályt pedig elküldjük az összes segédmetódusnak. A folyamatot addig ismétli a rendszer, amíg a felhasználó a `q` betrássával leállítja az alkalmazást:

- System.IO.BinaryWriter
- System.Math
- System.Console
- MyTypeViewer.Program

A 16.2. ábrán a `System.Math` osztály részleges kimenete látható.



16.2. ábra: A `System.Math` reflexiója

Generikus típusok reflexiója

Amikor generikus típusok metaadat-leírásának megszerzéséhez hívjuk a `Type.GetType()` metódust, speciális szintaxist kell alkalmaznunk a „visszafelé aposztrofál” (`?`), amelyet egy numerikus érték követ. A numerikus érték jelzi, hogy a típus hány típusparamétert támogat. Ha például a `List<T>` metaadat leírását szeretnénk kinyerni, a következő sztringet kell betennünk az alkalmazásunkba:

```
System.Collections.Generic.List`1
```

A numerikus érték az `1` lesz, hiszen a `List<T>` egyetlen típusparaméterrel rendelkezik. Ha azonban a `Dictionary<TKey, TValue>` osztállyal dolgozunk, a numerikus érték `2`:

```
System.Collections.Generic.Dictionary`2
```

Metódusparaméterek és visszatérési értékek reflexiója

Módosítsuk a `ListMethods()` segédfüggvényt, hogy az ne csak a meghatározott metódus nevét listázza, de a visszatérési értéket és a bejövő paramétereket is. Ezt a `MethodInfo` típus `ReturnType` tulajdonságával és `GetParameters()` metódusával oldhatjuk meg. Az alábbi forráskódban beágyazott `foreach ciklus` segítségével olyan sztringtípust készítünk, amely az összes paraméter típusát és nevét tartalmazza:

```
static void ListMethods(Type t)
{
    Console.WriteLine("***** Methods *****");
    MethodInfo[] mi = t.GetMethods();
    foreach (MethodInfo m in mi)
    {
        // A visszatérési érték megszerzése.
        string retVal = m.ReturnType.FullName;
        string paramInfo = "(" +
            // A paraméterek megszerzése.
            foreach (ParameterInfo pi in m.GetParameters())
            {
                paramInfo += string.Format("{0} {1} ", pi.ParameterType,
                    pi.Name);
            }
            paramInfo += ")";
        // Most jelenítsük meg az egyszerű metódus-aláírást.
        Console.WriteLine("<-> {0} {1} {2}", retVal, m.Name, paramInfo);
    }
    Console.WriteLine();
}
```

Ha a módosított alkalmazást futtatjuk, azt tapasztaljuk, hogy egy adott típus metódusai sokkal több információt hordoznak. A 16.3. ábrán a `System.Globalization.GregorianCalendar` típus metódusainak metaadatai láthatók. A `ListMethods()` fenti megvalósítása azért hasznos, mert a `System.Reflection` objektummodell segítségével közvetlenül megvizsgálhatjuk a paramétereket és a metódusok visszatérési értékét. Az `xxxInfo` típusok mindegyike (`MethodInfo`, `PropertyInfo`, `EventInfo` stb.) felülbírálja a `ToString()` metódust a kért elem aláírásának a megjelenítéséhez. A `ListMethods()` metódust tehát megvalósíthatjuk a következőképpen:

Dinamikusan betöltődő szereplvények

16. fejezet: Típusreflexió, késői kötés és attribútumalapú programozás

A CLR a szereplvény manifestumához fordul, amikor külső hívatközött szereplvényt keres (lásd a 15. fejezetben). Sokszor előfordulhat, hogy programozottan, menet közben kell betölteniünk szereplvényeket, még akkor is, ha az említett szereplvényről a manifestum említést sem tesz. Szakmai szempontból a külső szereplvények igény szerinti betöltésének folyamataát *dinamikus betöltésnek* nevezzük.

A `System.Reflection` névtér definiálja az `Assembly` osztályt. A típus segítségével dinamikusan tölthetünk be egy szereplvényt, valamint ismerhetjük meg annak tulajdonságait. Az `Assembly` típussal dinamikusan betölthetünk privát és megosztott szereplvényeket, valamint tesztelges helyen tárolt szereplvényt is. Az `Assembly` osztály lenyegében metódusokat (nevezetesen a `Load()` és a `LoadFrom()` metódusokat) biztosít, amelyekkel programozottan beszerezhetjük az ügyféloldali konfigurációs fájlban található információkat. A dinamikus betöltés bemutatásához készítsük el az újabb paramcsorti alkalmazásunkat, az `ExternalAssemblyReflector`. A feladatunk az, hogy olyan `main()` metódust készítsünk, amely a dinamikusan betölteni kívánt szerelvény barátságos nevet kérdezi meg. Az `Assembly` referenciát a `DisplayTypes()` segédmetódusunk adjuk át, amely a benne foglalt osztályok, interfészek, struktúrák, felsorolások és metódusreferenciák nevét listázza. A forráskód meglehetősen egyszerű:

```
using System;
using System.Reflection;
using System.IO; // A FileNotFoundException definíciójához.

namespace ExternalAssemblyReflector
{
    class Program
    {
        static void DisplayTypesInAsm(Assembly asm)
        {
            Console.WriteLine("\n***** Types in Assembly *****");
            Console.WriteLine("<->{0}", asm.FullName);
            Type[] types = asm.GetTypes();
            foreach (Type t in types)
            {
                Console.WriteLine("Type: {0}", t);
            }
            Console.WriteLine("");
        }
    }
}
```



```

static void Main(string[] args)
{
    Console.WriteLine("***** External Assembly Viewer *****");
    string asmName = "";
    bool userIsDone = false;
    Assembly asm = null;

    do
    {
        Console.WriteLine("\nEnter an assembly to evaluate");
        Console.WriteLine("or enter q to quit: ");
        // A típus nevének megszerzése.
        asmName = Console.ReadLine();
        // A felhasználó ki akar lépni?
        if (asmName.ToUpper() == "q")
        {
            userIsDone = true;
            break;
        }
        // Próbáljuk meg betölteni a szerelvényt.
        try
        {
            asm = Assembly.Load(asmName);
            DisplayTypeInAsm(asm);
        }
        catch
        {
            Console.WriteLine("Sorry, can't find assembly.");
        }
        while (!userIsDone)
        {
        }
    }
}

```

A statikus Assembly.Load() metódusnak kizárólag a memóriába betölteni kívánt szerelvény barátságos nevét adjuk át. Ha tehát a CarLibrary.dll szerelvényt reflexiót szeretnénk alkalmazni, a CarLibrary.dll bináris fájlt a program futtatásához az ExternalAssemblyReflector alkalmazás \bin\Debug könyvtárába kell másolnunk. A másolás után a 16.4. ábrán látható kimenethez hasonló eredményt kapunk.

Ha rugalmasabba szeretnénk tenni az ExternalAssemblyReflector alkalmazmazást, módosíthatjuk a forráskódot úgy, hogy az Assembly.Load() metódus helyett az Assembly.LoadFrom() metódussal töltsé be a kívűs szerelvényt. Így megadhatjuk a megtekinteni kívánt szerelvény abszolút elérési útját (pl.: C:\MyApp\MyAsm.dll). Lényegében az Assembly.LoadFrom() metódussal programozottan adhatjuk meg a <codebase> értéket.

```
// A CarLibrary 1.0.982.23972 verziójának betöltése
// az alapértelmezett kultúra használatával.
Assembly a = Assembly.Load(
    @"CarLibrary, Version=1.0.982.23972, PublicKeyToken=null,
    Culture=""""");
```

A megjelenített név összeállításakor a `PublicKeyToken = null` konvenció jelzi, hogy erős névvel nem rendelkező szerelvényhez történő kötés és egyeztetés szükséges. Továbbá a `Culture = ""` érték mutatja, hogy a rendszernek a cél-számtípus alapértelmezett kultúraazonosítójával kell egyeztetnie:

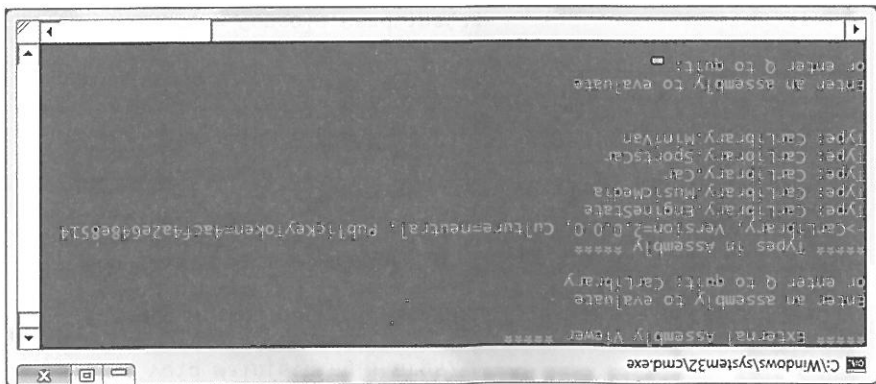
```
name(,version = major.minor.build.revision)
(,culture = culture token) (,public key token)
```

Az `Assembly.Load()` többszörösen túlterhelt metódus. Az egyik változat lehetővé teszi kultúraazonosító (lokallizált szerelvények esetén), valamint verziószám és nyilvános kulcs értékének (megosztott szerelvények számára) a meghatározását. A szerelvényt azonosító elemek gyűjteménye a `// megjelenítési név`. A megjelenített név formátuma a név/érték páros vesszővel elválasztott sztringje. A sztring a szerelvény barátságos nevével kezdődik, amelyet (tetszőleges sorrendben) opcionális szerelvényminősítők követnek. A következő sablon a következő (az opcionális minősítők zárójelben jelennek meg):

Megosztott szerelvények

Forráskód Az `ExternalAssemblyReflector` projekt megtalálható a 16. fejezet alkönyvtárában.

16.4. ábra: A külső `CarLibrary` szerelvény reflexiója



16. fejezet: Típusreflexió, késői kötés és attribútumalapú programozás

A `System.Reflection` névter által biztosított `AssemblyName` típus lehetővé teszi, hogy az előbb összeállított sztringinformációt egy jól kezelhető objektummá változáltsuk. Ezt az osztályt rendszert a `System.Version` osztállyal együtt alkalmazzuk, amely a szerverlány verziószámának OO-burkolója. A megjelölt név létrehozása után ezt átadhatjuk a `Assembly.Load()` módszernek:

```
// Használjuk az AssemblyName típust a megjelölt név
// definiálásához.
AssemblyName assemblyName;
assemblyName = new AssemblyName();
assemblyName.Name = "CarLibrary";
Version v = new Version("1.0.982.23972");
assemblyName.Version = v;
Assembly a = Assembly.Load(assemblyName);
```

Há a globális szerverlánytárból (GAC) szerverlányt betölteni egy megosztott szerverlányt, az `Assembly.Load()` paraméternek meg kell határoznia a `public-keytoken` értékét. Töltsük fel például, hogy a .NET-alapoztatlykönyvtárak `System.Windows.Forms.dll` szerverlányének 2.0.0.0 verzióját töltsük be. Mivel a szerverlány rengeteg típust foglal magában, a következő alalkalmazás, egyszerű LINQ-lekérdezés segítségével, csak a nyilvános felsorolások nevét listázza:

```
using System;
using System.Reflection;
using System.IO;
using System.Linq;

namespace SharedAssemblyReflector
{
    public class SharedAssemblyReflector
    {
        private static void DisplayInfo(Assembly a)
        {
            Console.WriteLine("***** Info about Assembly *****");
            Console.WriteLine("Loaded from GAC? {0}",
                a.GlobalAssemblyCache);
            Console.WriteLine("Assembly Name: {0}", a.GetName().Name);
            Console.WriteLine("Assembly Version: {0}", a.GetName().Version);
            Console.WriteLine("Assembly Culture: {0}",
                a.GetName().CultureInfo.DisplayName);
            Console.WriteLine("\nHere are the public enums:");
        }

        // Használjunk LINQ-lekérdezést a nyilvános felsorolások
        // kitűztetéséhez.
        Type[] types = a.GetTypes();
        var publicEnums = from pe in types where pe.IsEnum &&
            pe.IsPublic select pe;
```

```
foreach (var pe in publicenums)
{
    Console.WriteLine(pe);
}
}
static void Main(string[] args)
{
    Console.WriteLine("***** The Shared Asm Reflector App *****\n");
    // Töltsük be a System.Windows.Forms.dll szerializált
    // a GAC-ból.
    string displayName = null;
    displayName = "System.Windows.Forms, " +
        "version=2.0.0.0, " +
        "publickeytoken=b77a5c561934e089, " +
        @"culture=""""";
    Assembly asm = Assembly.Load(displayName);
    DisplayInfo(asm);
    Console.WriteLine("Done!");
    Console.ReadLine();
}
}
```

Forráskód A SharedAsmReflector projekt megtalálható a 16. fejezet alkönyvtárában.

Megismerkedtünk tehát a `System.Reflection` névtér alapfüggvényeivel a metaadatok futásidejű vizsgálatahoz. Annak ellenére, hogy rendkívül hasznos lehetőség, munkánk során valószínűleg nem kell egyedi objektumbőngészőket készítenünk. A reflexió szolgáltatások képezik azonban több általános programozási tevékenység, például a *késői kötés* alapját.

A késői kötés

A *késői kötés* olyan technika, amelynek segítségével futásidőben létrehozhatjuk egy adott típus egy példányát, és meghívhatjuk a típus tagjait anélkül, hogy az alkalmazásba kódolt fordítási idejű ismeretekkel rendelkeznenk a levezéséről. Ha az alkalmazás, amelyet készítettünk, késői kötést használ egy külső szerializált típusán, nem kell a szerializálásra hívatkoznunk; így a hívó felületre nem listázza a szerializált.

Mi a késői kötés használata? Igaz, hogy ha alkalmaznánk a típus korai kötését (pl. beállítunk szerelevényreferenciát, és a `C# new` kulcsszavának segítségével lefoglaljuk a típust), akkor érdemes valóban használni. Egyrészt a késői kötés lehetővé teszi, hogy a hibákat a futásidő helyett már a fordítás során kiszűrjük. Másrészt kritikus szereppel rendelkezik a kiterjeszhető alkalmazások készítésében. A fejezet későbbi részében alkalmunk nyílik kiterjeszhető program létrehozására; addig is azonban vizsgáljuk meg az aktivator típus szerepét.

A System.Activator osztály

A `System.Activator` osztály a .NET késői kötési folyamatának a kulcsa. A `System.Object` osztálytól örökölt metódusokon túl az aktivator a tagok olyan ki-sebb gyűjteményét definiálja, amelyek többsége a .NET-remoting API-hoz kapcsolódik. Ezek közül az `Activator.CreateInstance()` metódus segítségével-vel, késői kötéssel létrehozhatjuk a típus egy példányát.

A metódus többszörösen túlterhelt, így rendkívül rugalmas funkcionali-tást biztosít. A `CreateInstance()` tag legegyszerűbb változata érvényes `Type` objektummal dolgozik, amely a menetközben lefoglalni kívánt entitást írja le. Hozzuk létre a `latebindingapp` parancssori alkalmazást, és a következőkép-pen módosítsuk a `Main()` metódust (ne felejtsük el a `CarLibrary.dll` másola-tát a projekt `\bin\Debug` könyvtárba helyezni):

```
// Típus dinamikus létrehozása.
public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Late Binding *****");
        // Próbáljuk meg betölteni a CarLibrary helyi másolatát.
        Assembly a = null;
        try
        {
            a = Assembly.Load("CarLibrary");
        }
        catch (FileNotFoundException e)
        {
            Console.WriteLine(e.Message);
            return;
        }
        // A minivan típus metaadatainak kiolvásása.
        Type minivan = a.GetType("CarLibrary.Minivan");
    }
}
```

```
// A minivan létrehozása menet közben.
object obj = Activator.CreateInstance(minivan);
console.WriteLine("Created a {0} using late binding!", obj);
console.ReadLine();
}
```

Az `Activator.CreateInstance()` metódus az erősen típusos `minivan` helyett a `system.object` típust adja vissza. Ezért, ha az `obj` változóra a pont operátort alkalmazzuk, nem látjuk a `minivan` típus egyetlen tagját sem. A problémát explicit kasszt segítségével orvosolhatjuk; ennek a programnak azonban fogalma sincs arról, hogy a `minivan` egyáltalán micsoda (ha megoldhatnánk a problémát, mi szükségünk lenne a késői kötésre?).

A késői kötés lényege, hogy a fordítási időben a rendszer számára ismeretlen objektumok példányait tudjuk létrehozni. Ennek fényében felmerül a kérdés: Hogyan hívhatjuk meg a `system.object` változóban tárolt `minivan` objektum mögöttes metódusait? Természetesen reflexió segítségével.

Paraméter nélküli metódusok hívása

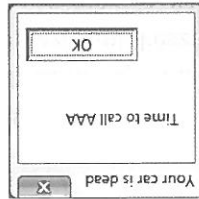
Tételezzük fel, hogy a `minivan` típus `turboboost()` metódusát szeretnénk meghívni. Ez a metódus a motor állapotát „tönkrement” értékre állítja, és tájékoztat a üzenetdobozt jelent meg. Az első lépés, hogy a `type.getmethod()` segítségével kinyerjük a `turboboost()` metódushoz tartozó metódusinfót. Az eredményt kapott metódusinfót példányon keresztül meghívhatjuk a `minivan` példány `turboboost` tagját. A `methodinfo.invoke()` metódus számára az összes olyan paramétert meg kell határoznunk, amelyeket a metódusinfó által képviselt metódusnak átadunk. A paramétereket a `system.object` típusok tömbje képviseli (ugyanis a metódus paraméterei között több különböző entitás szerepelhet). Mivel a `turboboost()` nem igényel paramétereket, null értéket adhatunk át (ez jelzi, hogy a metódus nem rendelkezik paraméterekkel). Módosítsuk `main()` metódusunkat a következőképpen:

```
static void main(string[] args)
{
    ...
    // A minivan típus metaadatainak kiolvasása.
    type minivan = a.GetType("CarLibrary.minivan");
    // A minivan létrehozása menet közben.
    object obj = Activator.CreateInstance(minivan);
    console.WriteLine("Created a {0} using late binding!", obj);
}
```

```
// Információk megszerzése a TurboBoost módszerrel.
MethodInfo mi = minivan.GetType().GetMethod("TurboBoost");

// A módszer meghívása ('null' akkor, ha nincs paraméter).
mi.Invoke(obj, null);
Console.ReadLine();
}
```

Ekkor a 16.5. ábrán látható üzenetdoboz tartalma a következő lesz:



16.5. ábra: Módushívás késői kötéssel

Paraméterekkel rendelkező módszerek hívása

Nézzük meg, hogyan lehet dinamikusan hívni paraméterekkel rendelkező módszereket. Tételizzük fel, hogy módosítottuk az előzőekben elkészített minivan típust, és a `Te11ch1dToBequie1()` módszerrel bővítettük:

```
// A csapat lecsendesítése...
public void Te11ch1dToBequie1(string kidName, int shameIntensity)
{
    for(int i = 0 ; i < shameIntensity; i++)
        MessageBox.Show(string.Format("Be quiet {0} !!", kidName));
}
```

A `Te11ch1dToBequie1()` módszernek két paramétere van: `szt1ng` tárolja a gyermek nevét, és egész szám képviseli az érzelmi pillanatnyi mértékét. Ha késői kötéssel alkalmazzuk, a bemenő paraméterek `system.object` tömbben adhatók át. Az új módszer hívásához a következőképpen módosítsuk a `main()` módszert:

```
static void main(string[] args)
{
    ...
    // A minivan típus metaadatainak kiolvasása.
    Type minivan = a.GetType("CarLibrary.Minivan");
}
```

```
// A minivan létrehozása menet közben.
object obj = Activator.CreateInstance(minivan);
console.WriteLine("Created a {0} using late binding!", obj);

// késői kötés paraméterekkel rendelkező módszerhez.
object[] paramArray = new object[2];
paramArray[0] = "Fred"; // Gyermekek neve.
paramArray[1] = 4; // Érzélem mértéke.
mi = minivan.GetMethod("TellChildToBeQuiet");
mi.Invoke(obj, paramArray);
console.ReadLine();
}
```

Ha futtatjuk a programot, négy üzenetdoboz jelenik meg az ifjú Fred megszégyenítésére. A reflexió, a dinamikus betöltés és a késői kötés közötti kapcsolatot már megvilágítottuk, ám még kérdéses lehet, hogy az alkalmazásunk készítése során pontosan *mikor* vehetjük hasznát ezeknek a módszereknek. A választ a fejezet összefoglalásában adjuk meg. A következőkben a .NET-attribútumok szerepét vizsgáljuk.

Forráskód A `lateBindingApp` projekt megtalálható a 16. fejezet alkönyvtárában.

Programozás attribútumokkal

A .NET-fordító egyik szerepe az, hogy elkészítse a definiált és hivatkozott típusok metaadat-leírását. A szerelvényben tárolt standard metaadatokon kívül a .NET platform lehetőséget biztosít a fejlesztőknek arra, hogy *attribútumok* segítségével további metaadatokat ágyazzanak a szerelvénybe. Röviden: az attribútumok forráskódjegyzetek, amelyeket egy adott típusra (osztály, interfész, struktúra stb.), tagra (tulajdonság, módszer stb.), szerelvényre vagy modorra alkalmazhatunk.

Az attribútumok által jegyzetekkel ellátott forráskód ötlete nem újdonosság. A COM IDL több olyan előre definiált attribútumot biztosított, amelyek segítségével a fejlesztők leírhatták az adott COM-kiszolgálón található típusokat. A COM-attribútumok azonban alig jelentettek többet kulcsszavak gyűjteményénél. Ha a COM-fejlesztőknek egyedi attribútumot kellett létrehozniuk, ezt ugyan nyugodtan megtehetné, de a rendszer a forráskódban az attribútumra csak egy 128 bites számmal (GUID) hivatkozott, és ez nagyon kényelmetlen megoldás.

A COM IDL attribútumoktól eltérően (amelyek pusztán kulcsszavakat jelentettek) a .NET-attribútumok osztálytípusok, amelyek a `System.Attribute` osztályból származó új típusok létrehozásával készíteni, hogy tovább pontosítsuk a típusaink viselkedését az `Attribute` osztályból származó új típusok létrehozásával.

Amikor a forráskódunkban attribútumokat alkalmazunk, a beágyazott metaadat egészen addig hasznavehetetlen, amíg egy másik kódreszt elplicit módon alkalmazott reflexióval ki nem nyeri azt. Ennek hiányában a szelvénybe ágyazott metaadatról senki sem szerez tudomást, így annak semmilyen hatása nincs.

Attribútumok feldolgozása

A .NET Framework 3.5 SDK több olyan segédprogramot tartalmaz, amelyek a különböző .NET-attribútumok alapján dolgoznak. A fordítási ciklus során magára a C#-fordító (`csc.exe`) is észleli a különféle attribútumok jelenlétét. Ha például a C#-fordító a `[CLSCompliant]` attribútummal találkozik, automatikusan ellenőrzi azt az elemet, amelyen az attribútumot alkalmazzuk, és megbizonyosodik róla, hogy az kizárólag CLS-kompatibilis szerkezetekkel dolgozik. Egy másik esetben, ha a C#-fordító `[Obsolete]` attribútummal ellátott elemet talál, a Visual Studio 2008 hibahírlékában fordítási figyelmeztetést jelent meg.

A fejlesztői eszközökön kívül a .NET-alapoztálykönyvtár számos metódusa is jellegzetes attribútumokat figyel. Ha például fájlban szeretnénk rögzíteni az objektum állapotát, akkor egyetlen tennivalónk az, hogy az osztályunkat ellátsuk a `[Serializable]` attribútummal. Ha a `BinaryFormatter` osztály `Serialize()` metódusa ezzel az attribútummal találkozik, az objektum mentése automatikusan egy tömör binárisformátumban történik.

A .NET CLR attribútuma ugyancsak bizonyos attribútumok jelenlétét kutatja. A legismertebb .NET-attribútum talán a `WebMethod`. Ha szeretnénk értéket automatikusan XML-formátumban kódolni, alkalmazzuk a `WebMethod` attribútumot, és a CLR minden egyéb részletről gondoskodik. A webszolgáltatás fejlesztésén túl az attribútumok a .NET biztonsági rendszer, a Windows Communication Foundation és a COM/.NET együttműködési képesség számára is egyaránt szükségesek.

Bátran készíthetünk olyan alkalmazásokat, amelyekben saját egyedi attribútumainkon, illetve a .NET-alapoztálykönyvtárak tetszőleges attribútumain alkalmazunk reflexiót. Attribútumok készítésével gyakorlatilag szerelevé-nyek számára érthető kulcsszavakat hozhatunk létre.

Attribútumok alkalmazása a C# nyelvben

A .NET-alapoztálykönyvtár különböző névterekben több attribútumot biztosít. A 16.3. táblázatban néhány (de *nem* az összes) előre definiált attribútumot mot találunk.

Attribútum	Valós jelentése
[CLSCompliant]	Kényszeríti a jeggzettel ellátott elemet, hogy megfeleljen a CLS szabályainak. A CLS-kompatibilis típusokat zökkenőmentesen használhadjuk bármely .NET programozási nyelvben.
[DllImport]	Lehetővé teszi a .NET-kód számára, hogy bármilyen nem felügyelt C- vagy C++-alapú kódkönyvtárat meghívjon – beleértve a mögöttes operációs rendszer API-ját is. A [DllImport] nem használatos COM-alapú programokkal folytatott kommunikációban.
[Obsolete]	Elavult típust vagy tagot jelöl. Ha a programozó ilyen elemet próbál használni, a fordító figyelmeztetést küld, amelyben a probléma leírása található.
[Serializable]	„Sorosítható” osztályt vagy szerkezetet jelöl. Ez annyit jelent, hogy az elem aktuális állapotja adatfolyamban rögzíthető.
[NonSerialized]	Meghatározza, hogy az osztály vagy szerkezet adott mezőjét a sorostási folyamat ne tárolja.
[WebMethod]	Jelzi, hogy a metódust hívhatják HTTP-kérések, és utasítja a CLR-t, hogy a függvény visszatérési értékét XML-formátumba sorosítsa.

16.3. táblázat: Néhány előre definiált attribútum

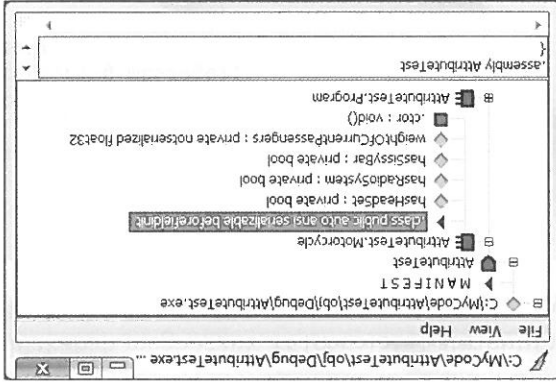
Az attribútumok alkalmazásának bemutatásához tételezzük fel, hogy szeretnénk elkészíteni a bináris formátumban tárolható `Motorcycle` osztályt. Ennek érdekében alkalmazzuk az osztálydefinição a `[Serializable]` attribútumot. Ha olyan mezővel dolgozunk, amelyet nem szeretnénk tárolni, alkalmazzuk a `[NonSerialized]` attribútumot:

```
// Ez az osztály elmenthető.
[Serializable]
public class Motorcycle
{
    // Ez a mező azonban nem marad meg.
    [NonSerialized]
    float weightOfCurrentPassengers;

    // Ezek a mezők még sorosíthatók.
    bool hasRadioSystem;
    bool hasHeadset;
    bool hasSuspensionBar;
}
```

Megjegyzés Az attribútum kizárólag a „közvetlenül mellette álló” elemre érvényes. A `Motorcycle` osztály egyetlen nem sorosított mezője a `weightOfCurrentPassengers` mező. A többi mező sorosítható, hiszen a teljes osztályt megjelöltük a `[Serializable]` attribútummal.

Ha attribútumot szeretnénk alkalmazni, az attribútum nevét szögletes zárójelbe kell foglalnunk. Az osztály fordítását követően az `ildasm.exe` segítségével megnevezhetjük az újabb metaadatokat. A fenti attribútumokat a `serializable` és a `nonserialized` tokenek rögzítik (lásd 16.6. ábrát).



16.6. ábra: Attribútumok az `ildasm.exe` segédprogramban

Egyetlen elemhez több attribútumot társíthatunk. Tétélezzük fel, hogy rendelkezünk egy sorosíthatóként megjelölt, örökölt C#-osztálytípussal (horse-andbuggy), amely az aktuális fejlesztés szempontjából már elavultnak számít. Ahelyett, hogy az alapkódból töröljünk az osztály definícióját (és esetleg tönkretennénk a programot), megjelölhetjük az osztályt az [obsolete] attribútummal. Ha egyetlen elemen több attribútumot szeretnénk alkalmazni, állítsuk össze az attribútumok vesszővel elválasztott listáját:

```
[Serializable, Obsolete("use another vehicle")]
public class HorseAndBuggy
{
    // ...
}
```

Másik megoldás az, ha úgy alkalmazzunk több attribútumot egyetlen elemen, hogy az egyes attribútumokat egymásra halmozzuk (a végeredmény ugyanaz):

```
[Serializable
[Obsolete("use another vehicle")]
public class HorseAndBuggy
{
    // ...
}
```

Attribútumok konstruktorparamétereinek meghatározása

Figyeljük meg, hogy az [obsolete] olyan paramétereket is fogad, amelyek leginkább a konstruktorok bemenő paramétereire hasonlítanak. Ha a Visual Studio 2008 Code Definition ablakában megnézzük az [obsolete] attribútum formális definícióját, láthatjuk, hogy az osztály valóban rendelkezik egy olyan konstruktorral, amely a system.string típust kapja:

```
public sealed class ObsoleteAttribute : System.Attribute
{
    public bool IsError { get; }
    public string Message { get; }
    public ObsoleteAttribute(string message, bool error);
    public ObsoleteAttribute();
}
```

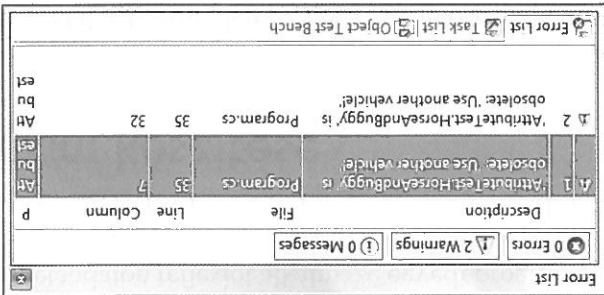
Ha konstruktorparaméterekkel látjuk el az attribútumot, a rendszer egészen addig *nem* foglalja le az attribútumot a memóriában, amíg a paramétereken más típus vagy külső eszköz nem alkalmaz reflexiót. Az attribútumszinten definiált sztringadatokat a szereplvény metaadat formájában tárolja.

Az Obsolete attribútum működés közben

Ha elavultként jelöltük a HorseAndBuggy típust, és szeretnénk lefoglalni a típus egy példányát:

```
static void Main(string[] args)
{
    HorseAndBuggy muLe = new HorseAndBuggy();
}
```

láthatnánk, hogy „az obsolete attribútum paramétereként megadott” sztring-adat a Visual Studio 2008 hibalistáblakában jelenik meg (lásd a 16.7. ábrát).



16.7. ábra: Attribútumok működés közben

Jelen esetben az a szoftver, amely reflexiót alkalmaz, maga a C#-fordító.

C#-attribútum rövidített formája

Az [obsolete] attribútum tényleges neve obsoleteAttribute, nem pedig obsolete. Névadási konvencióként minden .NET-attribútum (a saját készítésű egyedi attribútumokat is beleértve) utótagja az Attribute token. Az attribútumok alkalmazásának egyszerűsítésére a C# nyelvben nem szükséges beírunk az Attribute utótagot. Ennek megfelelően a HorseAndBuggy típus követ-kező verziója megegyezik az előzővel (csak kicsit többet kell gépeelnünk):

```
[SerializedAttribute]
[ObsoleteAttribute("Use another vehicle")]
public class HorseAndBuggy
{
    // ...
}
```

Ez a lehetőség a C# révén áll a rendelkezésünkre. A .NET-kompatibilis nyelvek közül csak néhány támogatja ezt az attribútumgyorsazonosító-szintaxist. A .NET-attribútumokkal kapcsolatos fontos tények :

- Az attribútumok a `System.Attribute` származnak.
- Az attribútumok beágyazott metadatot generálnak.
- Az attribútumok addig használhatóak, amíg egy másik program nem alkalmaz reflexiót rájuk.
- Az attribútumokat a C# nyelvben szögletes zárójelek között használjuk.

A következőkben vizsgáljuk meg, hogyan készíthetjük el saját egyedi attribútumainkat és a beágyazott metadaton reflexiót alkalmazó egyedi programot.

Egyedi attribútum készítése

Az egyedi attribútumok készítésének első lépése az, hogy a `System.Attribute` osztályból új származtatott osztályt hozunk létre. Az eddigi gépjárműpéldánál maradva tegyük fel, hogy létrehoztuk az `AttributedCarLibrary` új C#-osztálykönyvtárat. A szerelvény több járművet definiál (amelyek közül néhányat már találkoztunk), és ezek mindegyikét a `VehicleDescriptionAttribute` egyedi attribútum írja le:

```
// Egy egyedi attribútum.
public sealed class VehicleDescriptionAttribute : System.Attribute
{
    private string msgData;

    public VehicleDescriptionAttribute(string description)
    { msgData = description; }

    public VehicleDescriptionAttribute() {}
}
```

```

public string Description
{
    get { return msgData; }
    set { msgData = value; }
}

```

A VehicleDescriptionAttribute belül egy privát sztringet tárol, amelyet a típus konstruktorán keresztül állíthatunk be, és a típus tulajdonságán keresztül módosíthatunk. Azon túl, hogy az osztály a System.Attribute osztályból származik, az osztály definíciója semmilyen különlegességet nem tartalmaz.

Megjegyzés Biztonsági okokból a .NET gyakorlati ajánlatos az egyedi attribútumokat lezárt attribútumként létrehozni. A Visual Studio egy Attribute nevű kódsnippetet biztosít, amely egy SystemAttribute-ből származó kész osztályt generál a forráskódunkba. A kódrészletek alkalmazásának magyarázatát lásd a 2. fejezetben.

Egyedi attribútumok alkalmazása

Mivel a VehicleDescriptionAttribute a System.Attribute osztályból származik, tetszés szerinti jegyzetekkel láthatjuk el a járműveinket. Tesztelésre adjuk a következő osztálydefiníciókat az új osztálykönyvtárunkhoz:

```

// Létfás hozzáadása "nevesített tulajdonság" használatával.
[Serializable]
[VehicleDescription("My rocking Harley")]
public class Motorcycle
{
}

[Serializable]
[ObsoleteAttribute("Use another vehicle")]
[VehicleDescription("The old gray mare, she ain't what she used to be...")]
public class HorseAndBuggy
{
}

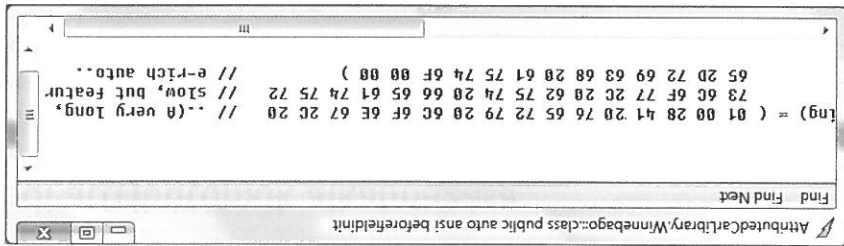
[VehicleDescription("A very long, slow, but feature-rich auto")]
public class Winnebago
{
}

```

Nevesített tulajdonság szintaxisa

A `Motorcycle` `description` tulajdonságát az új attribútum-központú szintaxis segítségével adtuk meg: ezt nevezzük *nevesített tulajdonságnak*. Az első `Vehicle` `description` attribútum konstruktorában a `new/érték` páros segítségével beállítjuk a mögöttes `system.string` osztaílyt. Ha külső beavatkozással reflexiót alkalmazunk az attribútumon, a rendszer a `description` tulajdonságnak adja az értéket (a nevesített tulajdonság szintaxisa kizárólag akkor megengedett, ha az attribútum írható .NET-tulajdonsággal rendelkezik).

Ezzel szemben a `HouseAndBuggy` és a `WineBago` típusok nem alkalmazzák a nevesített tulajdonság szintaxisát, egyszerűen egyedi konstruktorral adják az sztringadatokat. Az `AttributedCarLibrary` szerelvény fordítását követően az `tlasm.exe` segítségével megtekinthetjük a típusban elhelyezett újabb metaadat-leírásokat. A 16.8. ábrán például a `WineBago` típus beágyazott leírását láthatjuk.



16.8. ábra: Beágyazott járműleíró adatok

Az attribútumok használatának korlátozása

Az alapértelmezés szerint az egyedi attribútumokat a forráskód bármely elemére (metódusokra, osztályokra, tulajdonságokra stb.) alkalmazhatjuk. Ha tehát van értelme, akkor a `Vehicle` `description` attribútummal minősíthetjük a metódusokat, a tulajdonságokat vagy a mezőket (és egyéb elemeket):

```
[VehicleDescription("A very long, slow, but feature-rich auto")]
public class WineBago
{
    [VehicleDescription("My rocking CD player")]
    public void PlayMusic(bool on)
    ...
}
```


Néhányszor pontosan erre a viselkedésre van szükség. Máskor azonban előfordulhat, hogy olyan egyedi attribútumot kell készítenünk, amelyet csak bizonyos kódlemléken alkalmazhatunk. Ha szeretnénk korlátozni az egyedi attribútum hatókörét, az egyedi attribútum definícióján az [AttributeUsage] attribútumot kell alkalmaznunk. Az [AttributeUsage] lehetővé teszi, hogy az AttributeUsage felsorolásból (or operátor segítségével) az értékek tetszőleges kombinációját határozzuk meg:

```
// Ez a felsorolás definiálja egy tulajdonság lehetséges céljait.
public enum AttributeTargets
{
    All, Assembly, Class, Constructor,
    Delegate, Enum, Event, Field,
    Interface, Method, Module, Parameter,
    Property, ReturnValue, Struct
}
```

Az [AttributeUsage] lehetővé teszi, hogy beállítsunk egy nevesített tulajdonságot (AttributeUsage). Ezzel a tulajdonsággal meghatározza, hogy az attribútum többször is alkalmazható-e ugyanazon az elemen. Az [AttributeUsage] segítségével beállíthatjuk, hogy az Inherited nevesített tulajdonság révén a származtatott osztályok öröklhessék-e az attribútumot. Ha biztosítani szeretnénk, hogy a [AttributeUsage] attribútumot az osztályon vagy szerkezetben csak egyszer lehessen alkalmazni (és az érték ne öröklődjön tovább a származtatott típusokra), a [AttributeUsage] definícióját a következőképpen módosíthatjuk:

```
// Ez alkalmazza az AttributeUsage attribútummal
// látjuk el egyedi attribútumunkat.
[AttributeUsage(AttributeTargets.Struct,
    AllowMultiple = false, Inherited = false)]
public sealed class AttributeUsage : System.Attribute
{
    ...
}
```

Ekkor, ha a fejlesztő megkísérel osztály vagy egyéb struktúra helyett más elemen alkalmazni a [AttributeUsage] attribútumot, az eredmény fordítási idejű hiba lesz.

Tipp Szokjuk meg, hogy a létrehozott egyedi attribútumokat a használatukra vonatkozó jelölésekkel látjuk el, ugyanis nem minden .NET-nyelv részései előnyben a nem minősített attribútumok használatát.

Alkalmazásszintű (és modulszintű) attribútumok

Meghatározott modul minden típusán (többfajlos szerelvény esetén) vagy meghatározott szerelvény összes modulján alkalmazhatunk attribútumokat a [module:], illetve az [assembly:] címkek segítségével. Tételizzuk fel például, hogy biztosítani szeretnénk a szerelvényünkben definiált összes nyilvános típusCLS-kompatibilitását. Ennek érdekében a következő sort kell az egyik C#-forráskóddájlunkhoz adni (a szerelvény szintű attribútumokat a névtérdefiniáció hatókörén kívül kell elhelyeznünk):

```
//CLS-megfeleltetés kényszerítése a szerelvény összes nyilvános
// típusa számára.
[assembly: System.CLSCompliantAttribute(true)]
```

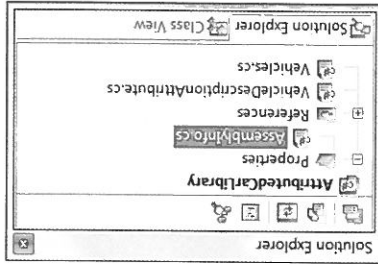
Ha a CLS-specifikációnak nem megfelelő kódreszt helyezzünk el az alkalmazásban (például előjel nélküli publikus adatot):

```
//Az ulong típusok nem felelnek meg a CLS-nek.
public class Winnebago
{
    public ulong notCompliant;
```

akkor a fordításkor figyelmeztetés jelenik meg.

A Visual Studio 2008 AssemblyInfo.cs fájlja

Az alapértelmezés szerint a Visual Studio 2008 projektek rendelkeznek egy AssemblyInfo.cs fájljal, amelyet a Solution Explorer Properties ikonjának segítségével nézhetünk meg (lásd a 16.9. ábrát).



16.9. ábra: Az AssemblyInfo.cs fájl

Alkalmazásszintű (és modulszintű) attribútumok

A fájlból kényelmesen elhelyezhető azokat az attribútumokat, amelyeket szerelvény szinten szeretnénk alkalmazni. A 15. fejezetben a „NET-szerelvényeket vizsgáljuk. Az önleírás szerelvény szintű metaadatokat foglal magában, amelyek nagy része a 16.4. táblázatban összefoglaló szerelvény szintű attribútumokból származik.

Attribútum		Valós jelentése
AssemblyCompanyAttribute	A vállalatra vonatkozó alapvető információkat tárol.	
AssemblyCopyrightAttribute	A termék vagy szerelvény szerzői jogai kapcsolatos információt tárolja.	
AssemblyCultureAttribute	A szerelvény által támogatott kultúrákról vagy nyelvekről nyújt információt.	
AssemblyDescriptionAttribute	Azon termékek vagy modulok barátságos leírását foglalja magában, amelyekből a szerelvény felépül.	
AssemblyKeyFileAttribute	Meghatározza a fájl nevét, amely a szerelvény aláírásához szükséges kulcspárt tárolja (például a szerelvény erős nevének elkészítéséhez).	
AssemblyOperationalSystemAttribute	Információt biztosít az operációs rendszerről, amelyet a szerelvény támogat.	
AssemblyProcessorAttribute	Információt biztosít a processzorról, amelyet a szerelvény támogat.	
AssemblyProductAttribute	Termékinformációkat biztosít.	
AssemblyTrademarkAttribute	Védjegy-információkat biztosít.	
AssemblyVersionAttribute	Meghatározza a szerelvény verzióinformációt <major.minor.build.revision> formában.	

16.4. táblázat: Néhány szerelvény szintű attribútum

Forráskód Az AttributedCarLibrary projekt megtalálható a 16. fejezet alkönyvtárában.

Attribútumok korai kötés használataival

16. fejezet: Típusreflexió, késői kötés és attribútumalapú programozás

Az attribútum addig felesleges, amíg valamilyen szoftverösszetevő reflexiót nem alkalmaz annak értékein. Ha a szóban forgó szoftver felleldezi az adott attribútumot, tetszése szerint bármilyen műveletet végrehajthat. A többi alkalmazáshoz hasonlóan korai kötés vagy késői kötés segítségével ez a „szoftverösszetevő” is felleldezheti az egyedi attribútum jelenlétét. Ha a korai kötést előnyeit szeretnénk kihasználni, gondoskodnunk kell arról, hogy az ügyfelalkalmazás rendelkezzen a kérdéses attribútum (ebben az esetben a vehicleDescriptionAttribute) fordítási idejű definíciójával. Mivel az AttributeLibrary szereplvény az egyedi attribútumot nyilvános osztályként definiálta, a korai kötés a lehető legjobb megoldás.

Az egyedi attribútumokon alkalmazott reflexió bemutatására hozzuk létre a vehicleDescriptionAttributeReader C# parancssori alkalmazást, majd készítsük el az AttributeLibrary szereplvény referenciáját. Végül frissítsük a kezdeti *.cs fájljunkat a következő kóddal:

```
// Reflexió az egyedi attribútumokon korai kötés használatával.
using System;
using AttributeLibrary;

public class Program
{
    static void Main(string[] args)
    {
        // Szerezünk meg a Winnebago képviselő típust.
        Type t = typeof(Winnebago);

        // Szerezünk meg a Winnebago összes attribútumát.
        object[] customatts = t.GetCustomAttributes(false);

        // Irjuk ki a listát.
        Console.WriteLine("***** Value of VehicleDescriptionAttribute *****\n");
        foreach (VehicleDescriptionAttribute v in customatts)
        {
            Console.WriteLine("<-> {0}\n", v.Description);
        }
    }
}
```

Ahogy a neve is sugallja, a `Type.GetCustomAttributes()` objektumtömböt ad vissza, amely a `Type` által tartalmazott tagon alkalmazott összes attribútumot megjeleníti (a logikai paraméter szabályozza, hogy a keresést a rendszer bővítse-e felfelé a származtatási láncban). Ha megszereltük az attribútumok listáját, iteráljunk végig a `VehicleDescriptionAttribute` osztályokon, és listázzuk a `Description` tulajdonság segítségével megszerzett értéket.

Forráskód A `VehicleDescriptionAttributeReader` projekt megtalálható a 16. fejezet alkönyvtárában.

Az előző példa a korai kötés segítségével listázta a `WineBago` típusú jármű leíró adatait. Ezt az tette lehetővé, hogy a `VehicleDescriptionAttribute` osztálytípust az `AttributeReader` szerelvényben nyilvántartott tagként definiáltuk. A dinamikus betöltés és a késői kötés segítségével reflexiót alkalmazhatunk az attribútumokon.

Hozzuk létre a `VehicleDescriptionAttributeReader` projektet, és másoljuk az `AttributeReader.d11` fájlt a projekt `bin\Debug` könyvtárába. Módosítsuk a `Main()` metódusunkat a következőképpen:

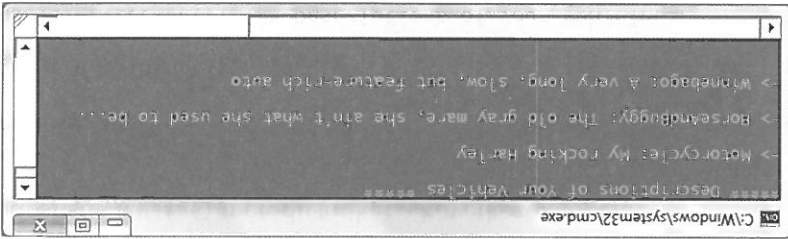
```
using System.Reflection;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Descriptions of your Vehicles *****\n");
        // Töltsük be az AttributeReader helyi másolatát.
        Assembly asm = Assembly.Load("AttributeReader");
        // Szerezzük meg a VehicleDescriptionAttribute
        // típusinformációját.
        Type vehicleDesc =
            asm.GetType("AttributeReader.
            VehicleDescriptionAttribute");
        // Szerezzük meg a Description tulajdonság típusinformációját.
        PropertyInfo propDesc = vehicleDesc.GetProperty("Description");
```

```
//Szerezünk meg a szereplvény összes típusát.
Type[] types = asm.getTypes();

// Iteráljunk végig minden típuson, és szerezzük meg az összes
// VehicleDescriptionAttributes attribútumot.
foreach (Type t in types)
{
    object[] objs = t.GetCustomAttributes(VehicleDesc, false);

    // Iteráljunk végig az összes VehicleDescriptionAttribute
    // attribútumon, és a késői kötés használatával írassuk ki
    // a leírást.
    foreach (object o in objs)
    {
        console.WriteLine("-> {0}: {1}\n",
            t.Name, propDesc.GetValue(o, null));
    }
}
console.WriteLine();
}
```

Ez a main() metódus már (többé-kevésbé) magától értődő és világos. Az egyetlen érdekesség a PropertyInfo.GetValue() metódus alkalmazása, amelynek segítségével a Description tulajdonság olvasó tagját futtathatjuk. A 16.10. ábrán láthatjuk a kimenetet.



16.10. ábra: Attribútumok késői kötés használatával

Forráskód A VehicleDescriptionAttributeReaderLateBinding projekt megtalálható a 16. fejezet alkönyvtárában.

Reflexió, késői kötés és egyedi attribútumok kifejtése

Reflexió, késői kötés és egyedi attribútumok kifejtése

Annak ellenére, hogy több példát láttuk a címben szereplő jelenségekre működés közben, továbbra is lehetnek kétségeink a reflexió, a dinamikus betöltés, a késői kötés és az egyedi attribútumok alkalmazásával kapcsolatban. Annak érdekében, hogy életszerű szituációhoz tudjunk ezeket kapcsolni, egy összetettebb példára van szükség. Tegyük fel, hogy egy olyan programozói csapatban dolgozunk, amely az alábbi követelménynek megfelelően alkalmazást készít:

- A termék külső gyártótól származó eszközök segítségével legyen kiterjeszthető.

Pontosan mit kell a *kiterjeszthetőség* értelmünk? Gondoljunk a Visual Studio 2008 integrált fejlesztői környezetre. Az alkalmazás fejlesztése során különböző „kampókat” helyeztek el a programban, hogy más szoftvergyártók egyedi modulokat kapcsolhassanak az integrált fejlesztői környezethez. A Visual Studio 2008 fejlesztői csapata nyilvánvalóan nem tudott olyan külső .NET-sze-relvényreferenciákat beállítani, amelyek még el sem készültek (tehát a korai kötés értelmét veszti), akkor viszont hogyan is biztosíthatná az alkalmazás a szükséges kapcsolódási pontokat? A probléma megoldásának egyik lehetséges módja a következő:

- Először is a kiterjeszthető alkalmazásnak valamilyen bemeneti felületet kell biztosítania, amely lehetővé teszi a felhasználó számára a beépített kívánt modul meghatározását (ez lehet párbeszédpanel vagy parancssori kapcsoló). Ezt *dinamikus betöltéssel* oldhatjuk meg.
- Másodszorban a kiterjeszthető alkalmazásnak meg kell tudnia határozni, hogy a modul a megfelelő funkcionalitást (pl. a szükséges interfészek gyűjteményt) támogatja, és így csatlakoztatható a környezethez. Ezt *reflexióval* oldhatjuk meg.

- Végül a kiterjeszthető alkalmazásnak meg kell szereznie a szükséges infrastruktúra (például interfésztípusok gyűjteményének) referenciáját, és a mögöttes funkcionalitás elindításához meg kell hívnia a tagokat. Ezt *késői kötéssel* oldhatjuk meg.

Ha a kiterjeszthető alkalmazás meghatározott interfészek lekérdezésére készült, futásidőben képes meghatározni azt, hogy a típus aktíválható vagy sem. Ha az ellenőrzés sikeres, a kérdéses típus további interfészeket támogat, amelyek a funkcionálisukhoz sokoldalú szerkezetet biztosítanak. A Visual Studio 2008 fejlesztői pontosan ezt a megközelítést követték.

Kiterjeszthető alkalmazás készítése

A következőkben megvizsgálunk egy teljes példát, amely bemutatja egy kiterjeszthető Windows Forms-alkalmazás készítésének a folyamatát. Az alkalmazást külső szereplők funkcionálisával gyarapíthatjuk. A példában nem foglalkozunk a Windows Forms-alkalmazások programozásával, hanem a Windows Forms alkalmazói felületét a II. kötet 27. fejezetében fogjuk áttekinteni. Aki tehát nem rendelkezik kellő jártassággal ezen a területen, ott talál majd tanulmányozható példát. A kiterjeszthető alkalmazásunk a következők szerelvényekkel dolgozik:

- `commonSnappableTypes.dll`: Ez a szerelvény olyan típusdefiniciókat tartalmaz, amelyeket a beépülő modulok használnak, és amelyekre a Windows Forms-alkalmazás közvetlenül hivatkozik.

- `CsharpSnappin.dll`: C# nyelven megírt beépülő modul, amely használja a `commonSnappableTypes.dll` típusait.

- `vbnetSnappin.dll`: Visual Basic nyelven megírt beépülő modul, amely kihasználja a `commonSnappableTypes.dll` típusait.

- `MyExtendableApp.exe`: Ez az a Windows Forms-alkalmazás, amelyet az egyes beépülő modulok funkcionálitása kiterjeszthet.

Ez az alkalmazás dinamikus betöltés, reflexió és késői kötés segítségével dinamikusan fér hozzá azoknak a szerelvényeknek a funkcionálitásához, amelyekről nem rendelkezik előzetes ismeretekkel.

A CommonsnapableTypes.dll létrehozása

Elsőként hozzunk létre egy olyan szerelvényt, amely tartalmazza egy adott beépülő modul által használni kívánt típusokat. A beépülő modul a típusok segítségével kapcsolható a kiterjeszthető Windows Forms-alkalmazáshoz. A CommonsnapableTypes osztálykönyvtárprojektek két típust definiál:

```
namespace CommonsnapableTypes
{
    public interface IAppFunctionality
    {
        void DoIt();
    }

    [AttributeUsage(AttributeTargets.Class)]
    public sealed class CompanyInfoAttribute : System.Attribute
    {
        private string companyName;
        private string companyUrl;
        public CompanyInfoAttribute() {}

        public string Name
        {
            get { return companyName; }
            set { companyName = value; }
        }

        public string Url
        {
            get { return companyUrl; }
            set { companyUrl = value; }
        }
    }
}
```

Az IAppFunctionality interfész minden beépülő modul számára polimorf implementációt biztosít, amelyet a kiterjeszthető Windows Forms-alkalmazás felhasználhat. Mivel a példa pusztán bemutató jellegű, kizárólag a DoIt() metódust határozzuk meg. Egy valós igényeket kielégítő interfész (vagy interfészgyűjtemény) lehetővé tehetné, hogy az objektum szkripteket generáljon, az alkalmazás eszköztárában képet jelenítsen meg, vagy a hoszoló alkalmazás főmenüjébe integrálódjon.

A CompanyInfoAttribute típus egyedi attribútum, amelyet bármely osztálytípuson alkalmazhatunk, ha az szeretne a tárolóba beépülni. Az osztály definíciójából kitűnik, hogy a [CompanyInfo] attribútum lehetővé teszi a beépülő modul fejlesztőjének, hogy az összetevő származási helyéről néhány alapvető információval szolgálgjon.

A külső gyártó – aki a C# helyett a Visual Basic nyelvet részesíti előnyben – szerepének bemutatásához hozzunk létre egy új Visual Basic-kódkönyvtárat (vbnetcsnapin), amely ugyanazokra a külső szereklényekre hivatkozik, mint az előző példában a cssharpcsnapin projekt. A kód (ismét) szándékosan egyszerű:

Visual Basic beépülő modul készítése

Az IAppFunctionality interfész támogatásához explicit interfészmegvalósítást alkalmazzunk. Ez nem kötelező; a lényeg, hogy a rendszer egyetlen része, amelynek közvetlenül együtt kell működnie az interfésztípussal, a hosztoló Windows-alkalmazás. Az interfész explicit megvalósításával a DoIt() metódust nem közvetlenül a cssharpmodule típusból nyerjük ki.

```
using System;
using CommonsnapableTypes;
using System.Windows.Forms;

namespace Csharpcsnapin
{
    [CompanyInfo(Name = "Intertech Training",
        Url = "www.intertech.com")]
    public class Csharpmodule : IAppFunctionality
    {
        void IAppFunctionality.DoIt()
        {
            MessageBox.Show("You have just used the C# snap in!");
        }
    }
}
```

Az alábbiakban elkészítünk egy olyan típust, amely az IAppFunctionality interfészt valósítja meg. A kiterjeszthető alkalmazás általános szerkezetének megismeréséhez egy egyszerű típus is tökéletesen megfelel. Tervezzük fel, hogy a cssharpcsnapin osztálykönyvtárprojekt definícióját a cssharpmodule osztálytípust. Mivel az osztálynak alkalmaznia kell a commonsnapableTypes projektben definiált típusokat, ellenőrizzzuk, hogy a referenciát erre a bináris fájlra állítottuk-e be. (Arról is meg kell győződnünk, hogy a System.Windows.Forms is állítottunk-e be referenciát, mert használni fogjuk: a segítségével egy egyszerű üzenetet jelentünk majd meg.) Ezután nézzük meg a forráskódot:

C# beépülő modul létrehozása

```
Imports System.Windows.Forms
Imports CommonsnapppableTypes

<CompanyInfo(Name:="Chucky's Software",
url:="www.chuckysoft.com")> _
Public Class VbNetSnapIn
Implements IAppFunctionality

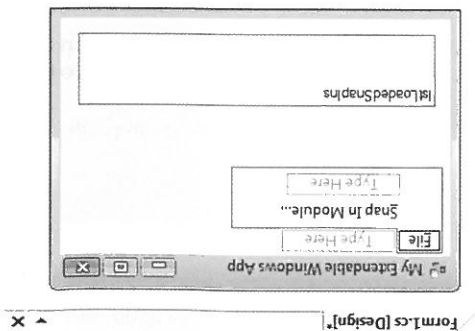
Public Sub DoIt() Implements
CommonsnappableTypes.IAppFunctionality.DoIt
MessageBox.Show("You have just used the VB .NET snap in!")
End Sub
End Class
```

A Visual Basic szintaxisában az attribútumok alkalmazásához a szükséges zárlójelek ([]) helyett a relációs jeleket (< >) kell használni. Meghatalozott osztályon vagy struktúrán az interfésztípusok megvalósításához az `implements` kulcsszót alkalmazzuk.

Kiterjeszthető Windows Forms-alkalmazás készítése

Végül készítsük el az új Windows Forms-alkalmazást (`MyExtendableApp`), amely lehetővé teszi a felhasználónak, hogy a standard Windows Open párbeszédpanel segítségével válasszon beépülő modult. Ezt követően állítsuk be a `CommonsnappableTypes.dll` szerelvény referenciáját, de *ne* hivatkozzunk a `csnapin.dll` vagy a `vbnetnapin.dll` kódkönyvtárra. Az alkalmazás célja a késői kötés és a reflexió használata. Ezek segítségével határozzuk meg a külső gyártótól származó független bináris fájlok „beépíthetőségét”. He-lyezzünk egy `MenuStrip` komponenst a Visual Studio úrlaptervezőben a `Form`-ra, és definiáljunk egy `File` nevű legfelső szintű menüelemet egyetlen almenüponttal, amelyet nevezzünk `Snap In Module`-nak. A főablak tartalmaz egy `Listbox`-ot is, amelyet `loadedsnapins`-nek nevezünk. Ezt fogjuk a felhasználó által betöltött beépülő modulok megjelenítésére használni. A 16.11. ábrán a végleges grafikus felület látható.

A `File > Snap In Module` menüelem eseménykezelője (amelyet tervezési időben a menüelem dupla kattintásával készíthetünk el) megjeleníti egy `File Open` ablakot, és kinyeri belőle a felhasználó által kiválasztott fájlt elérési útvonalat. Ha a felhasználó nem a `CommonsnappableTypes.dll` szerelvényt választotta (hiszen az pusztán infrastruktúraszerelevény), a kód az elérési útvonalat a `LoadExternalModule()` segédfüggvénynek küldi további feldolgozásra. A módus hamis értéket ad vissza, ha nem talál olyan osztályt, amely az `IAppFunctionality` interfészt valósítja meg.



16.11. ábra: A *MyExtendableApp* grafikus felülete

```
private void snapInModuleToolStripMenuItem_Click(object sender,
    EventArgs e)
{
    // Lehetővé teszi a felhasználónak betöltendő szereplvény
    // kiválasztását.
    openFileDialog dlg = new OpenFileDialog();
    if (dlg.ShowDialog() == DialogResult.OK)
    {
        if (dlg.FileName.Contains("CommonSnapableTypes"))
        {
            MessageBox.Show("CommonSnapableTypes has no snap-ins");
        }
        else if (!LoadExternalModule(dlg.FileName))
        {
            MessageBox.Show("Nothing implements IAppFunctionality");
        }
    }
}
```

A `LoadExternalModule()` metódus a következő feladatokat hajja végre:

- A kiválasztott szereplvényt dinamikusan betölti a memóriába.
- Eldönti, hogy a szereplvény rendelkezik-e olyan típusal, amely megvalósítja az `IAppFunctionality` interfészt.
- Késői kötéssel létrehozza a típust.

Ha olyan típust talál, amely megvalósítja az `IAppFunctionality` interfészt, meghívja a `DoIt()` metódust, majd a típus teljesen meghatározott nevét a listához adja. LINQ-lekérdezés segítségével keressük meg az `IAppFunctionality`-kompatibilis osztálytípusokat.

```

private bool LoadExternalModule(string path)
{
    bool foundSnapshot = false;
    Assembly theSnapshotasm = null;
    try
    {
        // A kiválasztott szereplővény dinamikus betöltése.
        theSnapshotasm = Assembly.LoadFrom(path);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
    return foundSnapshot;
}

// A szereplővény összes IAppFunctionality-kompatibilis
// osztályának megszerzése.
var theClassTypes = from t in theSnapshotasm.GetTypees()
    where t.IsClass &&
    (t.GetInterface("IAppFunctionality") != null)
    select t;

// Az objektum létrehozása és a DoIt() módszer meghívása.
foreach (Type t in theClassTypes)
{
    foundSnapshot = true;
    // A típus létrehozása késő később használattal.
    IAppFunctionality itfApp =
    (IAppFunctionality)theSnapshotasm.CreateInstance(t.FullName,
    true);
    itfApp.DoIt();
    loadedSnapshot.Items.Add(t.FullName);
}
return foundSnapshot;
}

```

Ekkor már futtathatjuk az alkalmazásunkat. Ha a csarpsnapin.dll vagy a vbnetnapin.dll szereplővényt választjuk, a megfelelő üzenetek kell megjelenjenie. Az utolsó feladat az, hogy megjelenítsük a [CompanyInfo] attribútum által biztosított metaadatokat. Ehhez módosítsuk a LoadExternalModule() metódust. Mielőtt kilépne a foreach hatókörből, a metódus meghívja a DisplayCompanyData() segédfüggvényt. A metódus egyetlen System.Type paraméterrel rendelkezik.

```

private bool LoadExternalModule(string path)
{
    ...
    foreach (Type t in theClassTypes)
    {
        ...
    }
}

```

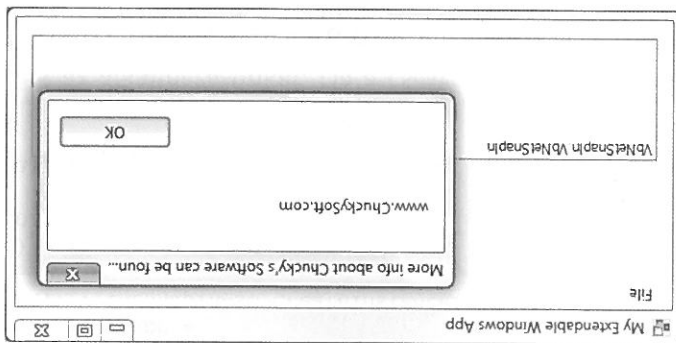
```
// céginformáció mutató.
displayCompanyData(t);
}
return foundSnapIn;
}
```

A bejövő típuson alkalmazzukunk reflexiót a [companyInfo] attribútum kinyerésére:

```
private void DisplayCompanyData(Type t)
{
    // A [companyInfo] adatok megszerzése.
    var compInfo = from ci in t.GetCustomAttributes(false) where
        (ci.GetType() == typeof(CompanyInfoAttribute))
        select ci;

    // Az adatok mutatója.
    foreach (CompanyInfoAttribute c in compInfo)
    {
        MessageBox.Show(c.Url,
            string.Format("More info about {0} can be found at", c.Name));
    }
}
```

A 16.12. ábrán a futtatás eredménye látható.



16.12. ábra: Külső szerverek beépítése

A példa megmutatta, hogy a fent tárgyalt témakörök hogyan segíthetik mindennapi munkánkat, és minden szoftverfejlesztőnek hasznosak lehetnek.

Forráskód A CommonsnapablETypes, a VbNetSnapIn, a CSharpSnapIn, és a MyExtensibleApp projektek a 16. fejezet alkönyvtárában találhatók.

Összefoglalás

A reflexió a robosztus OO környezetek érdekes oldala. A .NET világában a reflexió szolgáltatások sárolkköve a `System.Type` osztály és a `System.Reflection` névtér. A reflexió segítségével a típust futásidőben vizsgáljuk, és próbáljuk megérteni az adott elemre vonatkozó ki?, mi?, hol?, mikor?, miért? és hogyan? kérdéseket.

A késői kötés folyamataiban létrehozzuk a típust, és a tagok nevének előzetes ismerete nélkül hívjuk meg a típust említett tagjait. A késői kötés gyakran a *dinamikus betöltés* közvetlen eredménye, amely lehetővé teszi, hogy a .NET-szerelvényeket programozottan töltsük a memóriába. A kiterjeszthető alkalmazás példájában láthattuk, ezt a hatékony módszert az eszközök fejlesztői, valamint azok felhasználói is használják. Megvizsgáltuk az attribútumalapú programozás szerepét. Ha típusainkat attribútumokkal látjuk el, akkor a mögöttes szerelvény metaadatait gyarapítjuk.

Folyamatok, alkalmazástartományok és objektumkörnyezetek

Az előzőekben a CLR-nek azokat a lépéseit tanulmányoztuk, amelyekkel feloldhattuk egy külsőleg hivatkozott szerverlány helyét, valamint a .NET-metadatok szerepét. Ebben a fejezetben megnézzük, hogy a CLR hogyan hoztol egy szerverlányt, és megvizsgáljuk a folyamatok, az alkalmazástartományok és az objektumkörnyezetek közötti kapcsolatot.

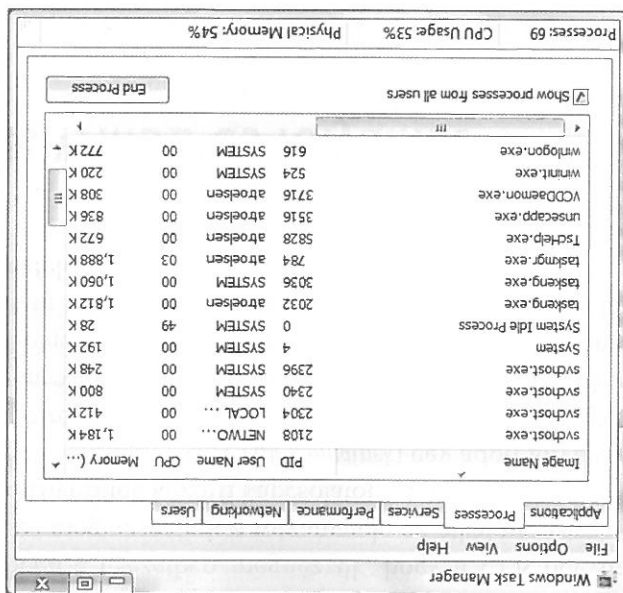
Röviden: az *alkalmazástartományok* (vagy *AppDomainek*) egy adott folyamaton belül olyan logikai alrészek, amelyek kapcsolódó .NET-szerverlányek készletét hozzólják. Az AppDomain tovább tagolható *környezetfüggetlen határoknak*, amelyek a hasonló .NET-objektumokat csoportosítják. A környezet fogalmát használva a CLR biztosítani tudja, hogy a speciális futási idejű igényeket támasztó objektumokat megfelelően kezelhessük.

A tradicionális Win32-es folyamat áttekintése

A „folyamat” fogalma már jóval a .NET platform kiadása előtt létezett a Windows operációs rendszerekben. A *folyamat* kifejezés a futó alkalmazással használt erőforráskészletet (mint a külső kódkönyvtárak és az elsődleges szál), valamint a szükséges memóriafoglalást írja le. Minthogy minden *.exe fajt beülődik a memóriába, az operációs rendszer létrehoz számukra egy saját, elkülönített folyamatot, amelyet bezárásukig használ. Az alkalmazás-izoláció ezen formáját használva az eredmény egy sokkal robosztusabb és stabilabb futási környezet lesz, ugyanis egy folyamat hibája nem befolyásolja a többi működését.

Minden Win32-folyamathoz hozzá van rendelve egy egyedi folyamatazonosító (PID), és az operációs rendszer szükség esetén egymástól függetlenül betöltheti vagy eltávolíthatja ezeket a folyamatokat (illetve ez programozottan a Win32 API hívásainak a segítségével is történhet). A Windows Taskmanager (feladatkezelő) (amely a Ctrl + Shift + Esc billentyűparanccsal hívható el) Processes (folyamatok) lapja lehetővé teszi az adott gépen futó folyamatokkal kapcsolatos különböző statisztikák megtekintését, beleértve a PID-et és a programkódnevét (lásd 17.1. ábra).

Megjegyzés A Windows-feladatkezelő Nézet > Oszlopok kiválasztása menüjében kiválasztathatjuk, hogy mely oszlopok (PID, felhasználó neve stb.) jelenjenek meg.



17.1. ábra: A Windows feladatkezelő

A szálak áttekintése

Minden Win32-folyamat pontosan egy „szállal” rendelkezik, amely az alkalmazás belépési pontjaként szolgál. A következők fejezetben lesz arról szó, hogy hogyan hozhatunk létre szálakat a .NET platformban a System.Threading névter használatával, ám néhány definícióra már most szükségünk lesz. Először is: a szál egy folyamaton belüli futási útvonál. A folyamat belépési pontja általában létrehozott első szál neve hivatalosan *elsődleges szál*. A Win32 API-ra épülő

grafikus asztali alkalmazások határozzák meg a `winmain()` metódust mint az alkalmazás belépési pontját. Másrészt: egy konzolalapú program ugyanerre a célra a `main()` metódust biztosítja.

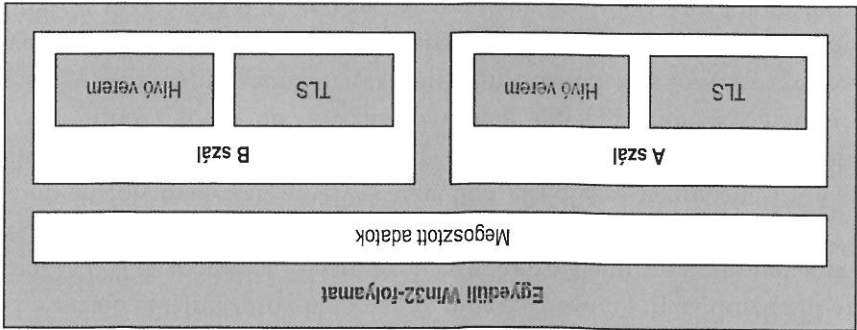
Az olyan folyamatokat, amelyek egyetlen elsődleges végrehajtható szálát tartalmazzák, *szálbiztosnak* nevezzük, ugyanis csak egy olyan szál van, amely adott időben hozzáfér az alkalmazás adataihoz. Egy egyszerű folyamat (különösen egy GUI-alapú) azonban gyakran nem választ, ha ez az egy szál valamilyen összetett műveletet hajt végre (pl. egy hosszú szöveges fájl nyomat, intenzív matematikai számolást végez, vagy egy több ezer kilométerre levő távoli kiszolgálóhoz próbál csatlakozni).

Mint ahogy az egyszerű alkalmazásoknak megvan ez a lehetséges hátránya, a Win32 API (és a .NET platform) lehetővé teszi, hogy az elsődleges szál továbbra is másodlagos szálakat (más néven: *dolgozó szálakat*) származtasson, meghozza olyan Win32 API funkciók használatával, mint a `createthread()`. Minden szál (elsődleges és másodlagos) egyedi futási útvonalra válik a folyamatban, és egyidejűleg hozzáfér minden megosztott adathoz.

A fejlesztők általában további szálakat is létrehozhatnak, hogy javítsák a program általános reakciókészségét. A többszálú folyamatok azt az érzést kelthetik, hogy számos tevékenység történik nagyjából ugyanakkor. Új szál indítható például egy nagy munkai igényű feladat (pl. egy nagy szövegfájl) nyomtatásának) elvégzésére.

Ha ez a másodlagos szál intenzíven dolgozik, a fő szál még mindig válaszol a bevitelre, amely az egész folyamatnak nagyobb teljesítményt képes adni. Am nem feltétlenül ez a helyzet: ugyanis túl sok szál használata egy folyamatban valószínűleg *csökkentheti* a teljesítményt, hiszen a processzornak valtozathnia kell a folyamat aktív szála között (és ez időt vesz igénybe).

Így tehát a többszálúság többnyire csak az operációs rendszer által nyújtott illúzió. Azok a gépek, amelyek egyetlen (nem hyperthreading) processzort használnak, nem is képesek egyszerre több szálát kezelni ugyanabban az időpontban. Ehelyett az egyetlen processzor egy időegységben (neve: *időszlet*) egyetlen szálát fog futtatni, ez pedig részben a szál prioritási szintjén alapul. Amikor egy szál időszlete lejár, akkor a szál felülgésztesődik, hogy más szál számára is lehetővé tegye a futtatást. Ahhoz, hogy minden szál emlékezzzen arra, hogy mi történt a „felrealizálásuk” előtt, rendelkeznek azzal a képességgel, hogy írjanak a Thread Local Storage-ba (TLS), valamint külön hívásveremmel is, ahogy az a 17.2. ábrán látható.



17.2. ábra: A Win32 folyamat/szál kapcsolat

A szál tehát egy egyedi futási útvonal egy Win32-folyamatban. Minden folyamat rendelkezik elsődleges szállal (amely a végrehajtható fájl beépési pontján keresztül jön létre), valamint tartalmazhat programozottan létrehozott további szálakat is.

Folyamatok kezelése a .NET platformban

Maguk a folyamatok és a szálak nem jelentenek újtonságot, a .NET platformon azonban ezeknek a primitíveknek a kezelése jelentősen javult. Először nézzük meg, hogy hogyan lehet kezelni a folyamatokat a .NET-alapozottalkönyvtárak használatával (a többszálú szerveralkalmazások építésénél miként lád a 18. fejezetben).

A System.Diagnostics névtér számos olyan típust meghatároz, amelyek lehetővé teszik a folyamatok és a különböző, diagnosztikaihoz kapcsolódó típusok (pl. a rendszer-eseménynapló és a teljesítményszámítók) programozott kezelését. Ebben a fejezetben csak a 17.1. táblázatban található folyamat-központú típusokkal foglalkozunk.

A System.Diagnostics névtér folyamatközpontú típusai	Valós jelentés
--	----------------

Process

A Process osztály hozzáférést biztosít a helyi és távoli folyamatokhoz, valamint programozottan lehetővé teszi a folyamatok indítását és leállítását.

A System.Diagnostics névtér		Folyamatközpontú típusai	
Valós jelentés		Valós jelentés	
ProcessModule	Ez a típus olyan modul (*.dll vagy *.exe) ábrázol, amelyet egy bizonyos folyamatba töltöttünk be. A ProcessModule típus bármilyen módon - COM-alapú, .NET-alapú vagy hagyományos C-alapú binárisokat - jellepezhet.	ProcessModuleCollection	Biztosítja a ProcessModule objektumok erősen típusos gyűjteményét.
ProcessStartInfo	Megad egy olyan értékkeszletet, amely egy folyamat Process.Start() metódus általi indításakor használatos.	ProcessThread	Egy szálát ábrázol egy adott folyamatban. A ProcessThread típus arra használjuk, hogy debug-nosztiáljuk egy folyamat szálkészletét, nem pedig arra, hogy új végrehajtási szálakat származtassunk a folyamaton belül.
ProcessThreadCollection	Biztosítja a ProcessThread objektumok erősen típusos gyűjteményét.	ProcessThreadCollection	Biztosítja a ProcessThread objektumok erősen típusos gyűjteményét.

17.1. táblázat: A System.Diagnostics névtér tagjai

A system.Diagnostics, process típus biztosítja az adott (helyi vagy távoli) gépen futó folyamatok analízálását. A process osztály emellett olyan tagokat is biztosít, amelyek lehetővé teszik folyamatok programozott indítását és befejezést, egy folyamat prioritási szintjének ellenőrzését, valamint az adott folyamat aktív számainak és/vagy betöltött moduljainak listázását. A 17.2. táblázat mutatja a system.Diagnostics, process néhány (de nem az összes) kulcstagját.

Tagok		Valós jelentés	
ExitCode	Ezen a tulajdonságon keresztül a folyamat kilépési kódja kérdezhető le. Ennek az értéknek a megszerzéséhez az ExitCode eseményt kell kezelniünk (az aszinkron értesítéshez), vagy meg kell hívniunk a WaitForExit() metódust (a szinkron értesítéshez).	ExitTime	Ez a tulajdonság lekérdezi a befejeződött folyamatához társított időbélyegget (ez a tulajdonság a folyamat leállásának idejét mutatja).

Tagok		Valós jelentés
Handle	Ez a tulajdonság visszaadja az operációs rendszer által a folyamathoz társított azonosítót.	
HandleCount	Ez a tulajdonság visszaadja a folyamat által megnyitott azonosítók számát.	
Id	Ez a tulajdonság lekérdezi a társított folyamat PID-jét.	
MachInName	Ez a tulajdonság lekérdezi annak a számtílogépnek a nevét, amelyen a társított folyamat fut.	
MainModule	Ez a tulajdonság lekérdezi a ProcessModule típust, amely egy adott folyamat fő modulját jelképezi.	
MainWindowTitle	A mainWindowTitle lekérdezi a folyamat fő ablakának a címét (ha a folyamatnak nincsen fő ablaka, akkor üres sztringet kapunk). A mainWindowHandle lekérdezi a társított ablak alapjául szolgáló azonosítót (a System.IntPtr típus révén). Ha a folyamatnak nincs főablaka, akkor az IntPtr típus a System.IntPtr zero értéket kapja.	
Modules	Ez a tulajdonság hozzáférést biztosít az erősen tipizált ProcessModuleCollection típushoz, amely az aktuális folyamat során betöltött modulkészletet (*.*dll vagy *.*exe) tartalmazza.	
PriorityBoostEnabled	Ezen a tulajdonságon keresztül beállítható, hogy az operációs rendszer több erőforrást biztosítson-e az alkalmazásnak arra az időre, amíg a hozzátartozó ablakon van a fókusz.	
PriorityClass	Ez a tulajdonság lehetővé teszi a társított folyamatok általános prioritásának a megtekintését és a módosítását.	
ProcessName	Ez a tulajdonság lekérdezi a folyamat nevét (amely magának az alkalmazásnak a neve).	
Responding	Ez a tulajdonság azt mutatja meg, hogy az alkalmazás felhasználói felülete pillanatnyilag tud-e reagálni a felhasználói akciókra (vagy éppen „kifagyott”).	
StartTime	Ez a tulajdonság lekérdezi a társított folyamat indításának idejét (a DateTime típus révén).	

Tagok	Valós jelentés
Threads	Ez a tulajdonság megkapja a társított folyamatban futó szálak készletét (amelyet a ProcessThread típusok egy tömbje jelképez).
CloseMainWindow()	Ez a módszer úgy zárja be a felhasználói felületet al-kalmazó folyamatot, hogy bezárási üzenetet küld a főablakának.
GetCurrentProcess()	Ez a statikus módszer visszaad egy olyan új Process típust, amely a jelenleg aktív folyamatot képviseli.
GetProcesses()	Ez a statikus módszer visszaadja egy adott gépen futó új Process komponensek tömbjét.
Kill()	Ez a módszer azonnal leállítja a társított folyamatot.
Start()	Ez a módszer elindít egy folyamatot.

17.2. táblázat: A Process típus tagjai

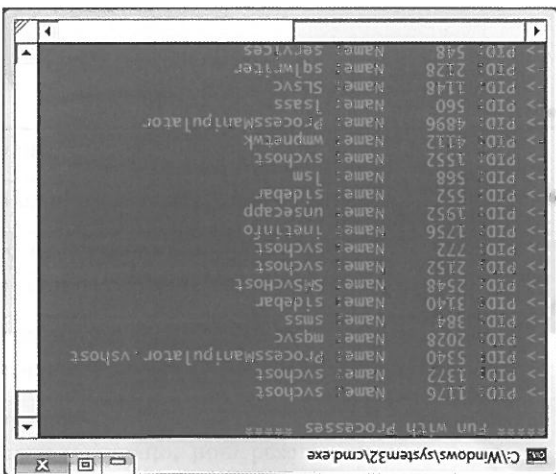
A futó folyamatok listázása

A process példányok kezelési folyamatainak bemutatásához tételezzük fel, hogy van egy olyan, processzamanipulátor nevű C# paramcssori alkalmazásunk, amely az alábbi statikus segédfüggvényt határozza meg a program osz-tályon belül (mindenképpen importáljuk a System.Diagnostics névtérét):

```
static void ListAllRunningProcesses()
{
    // Lekérdezi a helyi gépen futó összes folyamatot.
    Process[] runningProcs = Process.GetProcesses(".");
    // Kijelöl minden folyamat azonosítóját (PID) és nevét.
    foreach(Process p in runningProcs)
    {
        string info = string.Format("-> PID: {0} \tName: {1}",
            p.Id, p.ProcessName);
        Console.WriteLine(info);
    }
    Console.WriteLine("*****\n");
}
```

Figyeljük meg, hogy a statikus Process.GetProcesses() módszer hogyan adja vissza a célgépen (a pont jelölés a helyi számítógépet jelzi) futó folyamatokat egy Process típusú tömbben. Ha megszerezettük a Process típusok tömbjét,

akkor meghívhatjuk azoknak a tagoknak a bármelyikét, amelyeket a 17.2. táblázatban láthattunk. Jelenleg csak a folyamatok azonosítóját (PID-jét) és nevét jelentettük meg. Feltételezve, hogy úgy módosítottuk a main() metódust, hogy meghívja a listAllRunningProcesses() metódust, a 17.3. ábrán mutatott kimenethez hasonlóan fogunk látni.



17.3. ábra: A futó folyamatok listája

Egy adott folyamat vizsgálata

Amellett, hogy megszerezhetjük egy adott gépen futó összes folyamat listáját, a statikus Process.GetProcessById() metódus lehetővé teszi egyetlen Process típus megszerzését is a hozzájárított folyamatazonosító (PID) révén. Ha nem létező folyamatazonosítóhoz próbálunk hozzáférni, akkor a rendszer argument-exception kivételt dob. Ha például meg szeretnénk szerezni a 987-es azonosítója folyamatot jelképező Process objektumot, akkor a következőket írhatjuk:

```
// Ha nem létezik 987-es azonosítóval folyamat,
// akkor futási idejű kivételt kapunk..
static void GetSpecifiProcess()
{
    Process theProc = null;
    try
    {
        theProc = Process.GetProcessById(987);
    }
}
```



```

        catch // Az egyszerűség kedvéért általános catch szerepel.
        {
            Console.WriteLine("-> Sorry...bad PID!");
        }
    }
}

```

Egy folyamat szálképletének vizsgálata

A process osztálytípus arra is módot ad, hogy programozottan megvizsgáljuk az adott folyamat által használt összes szálát. A szálképletet az erősen típusos `ProcessThreadCollection` gyűjtemény jelképezi, amely néhány önálló `ProcessThread` típust tartalmaz. Tegyük fel, hogy a következő statikus segéd-függvényt adtuk hozzá az aktuális alkalmazásunkhoz:

```

static void EnumThreadsForPid(int pid)
{
    Process theProc = null;
    try
    {
        theProc = Process.GetProcessById(pid);
    }
    catch
    {
        Console.WriteLine("-> Sorry...bad PID!");
    }
    Console.WriteLine("*****\n");
    return;
}

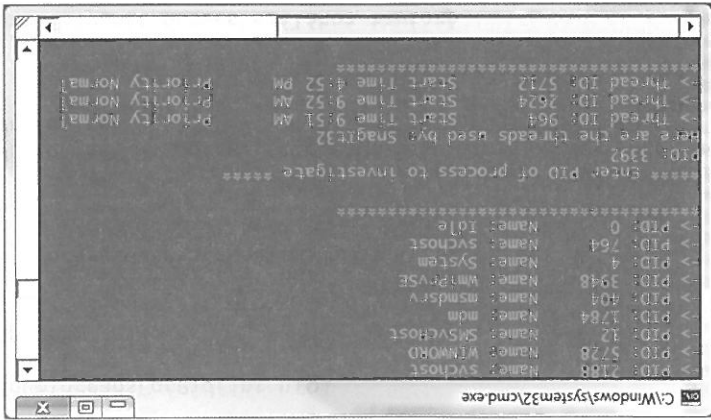
// A megadott folyamat összes szálához tartozó
// statisztika listázása.
Console.WriteLine("Here are the threads used by: {0}",
    theProc.ProcessName);
ProcessThreadCollection theThreads = theProc.Threads;
foreach(ProcessThread pt in theThreads)
{
    string info =
        string.Format("-> Thread ID: {0}\tstart time {1}\tpriority {2}",
            pt.Id, pt.StartTime.ToShortTimeString(),
            pt.PriorityLevel);
    Console.WriteLine(info);
}
Console.WriteLine("*****\n");
}

```

A `System.Diagnostics.Process` típus `Threads` tulajdonsága hozzáférést biztosít a `ProcessThreadCollection` osztályhoz. Ekkor kiterítjük a hozzárendelt szálaazonosítót, indítási időt, valamint a kliens által meghatározott prioritási szinteket. Így, ha az alábbiak szerint módosítjuk programunk `Main()` metódusát, hogy kérje be a vizsgálni kívánt folyamat azonosítóját:

```
static void main(string[] args)
{
    ...
    // PID bekérése és a hozzátartozó aktív szájak kitérítése.
    Console.WriteLine("***** Enter PID of process to investigate
    *****");
    Console.WriteLine("PID: ");
    string pid = Console.ReadLine();
    int theProcID = int.Parse(pid);
    EnumThreadObjects(theProcID);
    Console.ReadLine();
}
```

akkor hasonló kimenetet fogunk látni, mint a 17.4. ábrán.



17.4. ábra: Egy futó folyamatban lévő szájak listázása

A `ProcessThread` típus további tagokkal is rendelkezik az `Id`, `StartTime` és `PriorityLevel` tagok mellett. A 17.3. táblázat ezek közül sorol fel néhányat.

Tag		Valós jelentés
<code>BasePriority</code>	Lekérdezi a szál alapprioritását.	
<code>CurrentPriority</code>	Lekérdezi a szál aktuális prioritását.	
<code>Id</code>	Lekérdezi a szál egyedi azonosítóját.	
<code>IdealProcessor</code>	Beállítja, hogy az adott szállehetőleg melyik processzoron fusson.	
<code>PriorityLevel</code>	Lekérdezi vagy beállítja a szál prioritási szintjét.	

Tag	Valós jelentés
ProcessAffinity	Lekérdezi a processzorokat, amelyeken a társított szál futhat.
StartAddress	Lekérdezi annak a függvénynek a memóriacímét, amelyet az operációs rendszer hívott meg a szál elindítására.
StartTime	Lekérdezi azt az időpontot, amikor az operációs rendszer elindította a szálát.
ThreadState	Lekérdezi a szál aktuális állapotát.
TotalProcessorTime	Lekérdezi azt a teljes időmennyiséget, ameddig a szál a processzort használta.
WaitReason	Lekérdezi annak okát, hogy miért várakozik a szál.

17.3. táblázat: A *ProcessThread* típus tagjai

A *ProcessThread* típus *nem* arra használatos, hogy létrehozzon, felfüggeszsen vagy leállítson szálakat a .NET platformban, hanem arra szolgál, hogy diagnosztikai információkat szerezhessünk egy futó folyamatban lévő aktív Win32-es szálakról. (A többszálu alkalmazások készítését a *system.Thread* névtér használatával lásd a 18. fejezetben.)

Egy folyamat modulkészletének vizsgálata

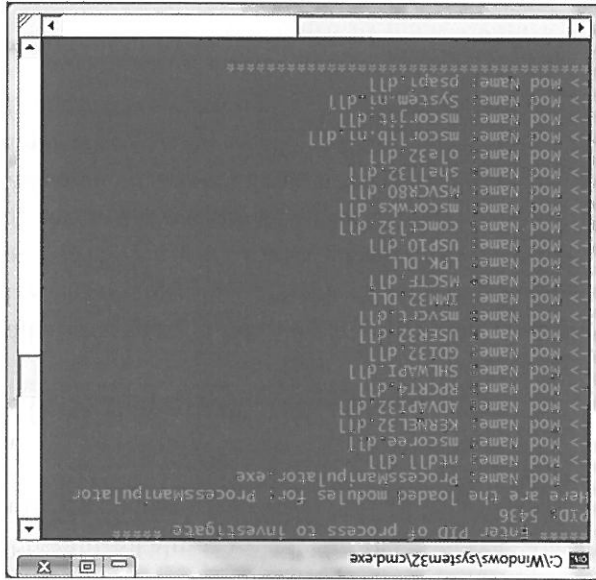
Nézünk meg, hogyan iterálhatunk végig a betöltött modulokon, amelyeket egy adott folyamat hozott. A *modul* olyan általános név, amely egy bizonyos folyamat által hozott adott *.dll-t vagy magát az *.exe-t írja le. Amikor a *Process.Module* tulajdonság révén férünk hozzá a *ProcessModuleCollection* gyűjteményhez, akkor listázhatjuk az adott folyamatban hozott összes *modul*. .NET-alapú, COM-alapú vagy tradicionális C-alapú könyvtárat. Az alábbi segédfüggvény a PID alapján kilistázza egy adott folyamat moduljait:

```
{
    static void EnumModulesForPid(int pid)
    {
        Process theProc = null;
        try
        {
            theProc = Process.GetProcessById(pid);
        }
    }
}
```

```

        catch
        {
            Console.WriteLine("-> Sorry..bad PID!");
            Console.WriteLine("*****\n");
            return;
        }
        Console.WriteLine("Here are the loaded modules for: {0}",
            theProc.ProcessName);
        try
        {
            ProcessModuleCollection theMods = theProc.Modules;
            foreach(ProcessModule pm in theMods)
            {
                string info = string.Format("-> Mod Name: {0}",
                    pm.ModuleName);
                Console.WriteLine(info);
            }
            Console.WriteLine("*****\n");
        }
        catch
        {
            Console.WriteLine("No mods!");
        }
    }
}

```



17.5. ábra: A futó folyamatban betöltött modulok listázása

Nézzünk egy lehetséges kimenetet: Irassuk ki a hosztoló folyamat (ProcessMani-
pulator) betöltött moduljait. Indítsuk el az alkalmazást, és nézzük meg a PID-jét
a TaskManagerben, majd adjuk meg ezt az értéket az alkalmazásunknak. (Ko-

rábban úgy módosítottuk a `main()` metódust, hogy az továbbadja a PID-et az `enummodsforsid()` metódusnak.) Ha mindezt megcsejtük, megcsejtve láthatjuk egy egyszerű parancssori alkalmazás során felhasználó *.dll-ek listáját (`gdi32.dll`, `user32.dll`, `ole32.dll`, `stb`). A 17.5. ábra mutatja a tesztelést fenn.

Folyamatok programozott indítása és leállítása

A `System.Diagnostics.Process` típus itt vizsgált utolsó aspektusai a `start()` és `kill()` metódusok. Ezek a tagok egy folyamat elindításának, illetve leállításának programozott módját biztosítják. Figyeljünk meg például a statikus `startAndKillProcess()` segédmetódust:

```
static void startAndKillProcess()
{
    // Internet Explorer elindítása.
    Process iProc = Process.Start("IExplore.exe",
    "www.intertech.com");
    Console.WriteLine("--> Hit enter to kill {0}...", iProc.ProcessName);
    // Az IExplore.exe folyamat leállítása.
    try
    {
        iProc.Kill();
    }
    catch { } // Ha a felhasználó már leállította volna...
}
```

A statikus `Process.Start()` metódus többszörösen túlterhelt. Legalább az elindítani kívánt folyamat barátságos nevét meg kell adnunk (mint a Microsoft Internet Explorer, `IExplore.exe`). Ez a példa a `start()` metódus egy olyan variációját használja, amellyel bármennyi további paramétert megadhatunk továbbadásra a program belepési pontjának (vagyis a `main()` metódusnak). A `start()` metódus azt is lehetővé teszi, hogy egy `System.Diagnostics.ProcessStartInfo` típust adjunk át, amely további információkat ad a folyamat elindulásával kapcsolatban. A `ProcessStartInfo` formális definíciója a következő (részletekért lásd a .NET Framework 3.5 SDK dokumentációját):

```
public sealed class System.Diagnostics.ProcessStartInfo :
```

```
    object
    {
        public ProcessStartInfo(string fileName);
        public ProcessStartInfo(string fileName, string arguments);
        public string Arguments { get; set; }
        public bool CreateNoWindow { get; set; }
        public stringDictionary EnvironmentVariables { get; }
        public bool ErrorDialog { get; set; }
        public IntPtr ErrorDialogParentHandle { get; set; }
        public string FileName { get; set; }
        public bool RedirectStandardError { get; set; }
        public bool RedirectStandardInput { get; set; }
        public bool RedirectStandardOutput { get; set; }
        public bool UseShellExecute { get; set; }
        public string Verb { get; set; }
        public string[] Verbs { get; }
        public ProcessWindowStyle WindowStyle { get; set; }
        public string WorkingDirectory { get; set; }
        public virtual bool Equals(object obj);
        public virtual int GetHashCode();
        public Type GetType();
        public virtual string ToString();
    }
```

A `Process.Start()` módusus bármely verzióját hívjuk meg, az az újonnan ak-
tívált folyamat referenciájával tér vissza. Ha le akarunk állítani egy folyama-
tot, egyszerűen hívjuk meg a példányszintű `Kill()` metódust.

Forráskód A `ProcessManager` projekt megtalálható a 17. fejezet alkönyvtárában.

A.NET-alkalmazástartomány

A következőkben vizsgáljuk meg a .NET-alkalmazástartomány koncepcióját.
A .NET platformban a futtatható fájlok nem közvetlenül egy folyamaton be-
ül vannak hosztolva (ellentétben a hagyományos Win32-alkalmazásokkal).
Ehelyett, a .NET végrehajtható fájlt egy logikai partíció hosztolja az *alkalma-
zástartomány* nevű folyamaton belül. Egyetlen folyamat több alkalmazástar-
tományt is tartalmazhat, ezek mindegyike egy .NET végrehajtható fájlt
hosztol. A tradicionális Win32-folyamat ezen további felosztása több tekin-
tetben is előnyös; nézzünk meg néhányat:

- Az alkalmazástartományok a .NET platform operációsrendszer-függően természetének egyfajta kulcselemei, hiszen ez a logikai felosztás megszünteti a különbséget arra nézve, hogy az alapul szolgáló operációs rendszer hogyan ábrázol egy betöltött végrehajtható fájlt.

- Az alkalmazástartományok kevesebb processzoridőt és memóriát igényelnek, mint a teljes folyamatok. Ezért a CLR gyorsabban képes betölteni és eltávolítani őket, mint egy formális folyamatot.
- Az alkalmazástartományok melyebb izolációs szintet biztosítanak a betöltött alkalmazás hosztolására. Ha az egyik folyamat alkalmazástartománya leáll, a többi tovább működik.

Egyetlen folyamat bármennyi alkalmazástartományt hosztolhat, ezek mindegyike teljesen elkülönül a folyamat egyéb alkalmazástartományaitól (és más folyamatoktól is). Emiatt egy alkalmazástartományban futó alkalmazás semmilyen adatot (globális változót vagy statikus mezőt) nem tud megsejteni egy másik alkalmazástartományból, hacsak nem használunk elosztott programozási protokollt (pl. a Windows Communication Foundationt). Egy folyamat több alkalmazástartományt *hosztolhat*, általában azonban nem ez történik. Az operációs rendszer egyik folyamata fogja hosztolni azt, amit *alapértelmezett alkalmazástartomány*nak nevezünk. Ezt a specifikus alkalmazástartományt a CLR automatikusan létrehozza a folyamat indításakor. Ezután ha szükséges, további alkalmazástartományokat is létrehoz. Ha igény van rá (valószínűleg *nem* az lesz a helyzet a .NET használatakor), akkor futásidőben is létrehozhatunk alkalmazástartományokat egy adott folyamaton belül, programozottan, a `system.appdomain` osztály statikus módszereinek felhasználásával. Ez az osztály az alkalmazástartományok alá-csony szintű vezérléséhez is hasznos. Az osztály kulcstagjai a 17.4. táblázatban láthatók.

Tag	Valós jelentés
-----	----------------

<code>CreateDomain()</code>	Ez a statikus módszer új alkalmazástartományt hoz létre az aktuális folyamatban. Mivel a CLR szükség esetén gondoskodik az új alkalmazástartományok létrehozásáról, ezért valószínűleg, hogy valaha is meg kell hívniunk ezt a módszert.
<code>GetCurrentThread()</code>	Ez a statikus módszer visszaadja az aktuális alkalmazástartományban az aktív szál azonosítóját.

Tag	Valós jelentés
-----	----------------

unload() Ez a statikus módszer lehetővé teszi a megadott alkalmazástartomány eltávolítását egy folyamattól.

Basedirectory Ez a tulajdonság lekérdezi a függő szerelvények kérésére használt alapkönyvtárat.

CreateInstance() Ez a módszer létrehozza az adott szerelvényfájlból meghatározott típus egy példányát.

ExecuteAssembly() Ez a módszer egy szerelvényt hajt végre egy alkalmazástartományon belül, megadva a fájlnévet.

GetAssembly() Ez a módszer lekérdezi az alkalmazástartományba betöltött .NET-szerelvényeket (a COM- és C-alapú szerelvényeket figyelmen kívül hagyja).

Load () Ez a módszer dinamikusan betölt egy szerelvényt az aktuális alkalmazástartományba.

Emellett az AppDomain típus olyan kis eseménykészletet is meghatároz, amely megfelel az alkalmazástartomány életciklusára jellemző aspektusoknak (lásd 17.5. táblázatot).

Esemény	Valós jelentés
---------	----------------

AssemblyLoad Szerelvény betöltésekor következik be.

AssemblyResolve Szerelvény feloldásának hibájakor fordul elő.

DomainLoad Alkalmazástartomány eltávolítása előtt fordul elő.

ProcessExit Az alapértelmezett alkalmazástartományon fordul elő akkor, amikor annak a szülőfolyamata kilép.

ResourceResolve Egy erőforrás feloldásának hibájakor fordul elő.

TypeResolve Egy típus feloldásának hibájakor fordul elő.

UnhandledException Akkor fordul elő, ha egy kivételt nem kap egyetlen eseménykezelő sem.

17.5. táblázat: Az AppDomain típus eseményei

Folyamat alkalmazástartományainak felsorolása

A .NET-alkalmazástartományok programozott kezelésének bemutatásához tegyük fel, hogy létrehozunk egy új C# parancssori alkalmazást AppDomainManipulator néven, amely meghatároz egy PrintAllAssembliesInAppDomain() nevű metódust a Program típuson belül. Ez a segédmetódus az AppDomain.GetAssemblies() használatával szerzi meg a szóban forgó alkalmazástartományban hosztolt összes .NET bináris fájl listáját.

Ezt a listát a System.Reflection.Assembly típusok egy tömbje reprezentálja, így a System.Reflection névteret kell használnunk (lásd a 16. fejezetet). Ha megszerztük a szerelvénytömböt, végigiterálunk a tömbön, és kiterítjük az egyes szerelvények barátságos nevét, valamint verzióját:

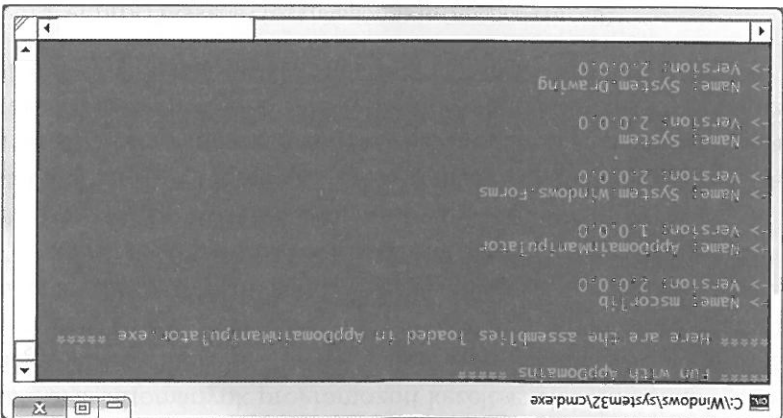
```
static void PrintAllAssembliesInAppDomain(AppDomain ad)
{
    Assembly[] loadedAssemblies = ad.GetAssemblies();
    Console.WriteLine("***** Here are the assemblies loaded in {0}
    ",
        ad.FriendlyName);
    foreach(Assembly a in loadedAssemblies)
    {
        Console.WriteLine("-> Name: {0}", a.GetName().Name);
        Console.WriteLine("-> Version: {0}\n", a.GetName().Version);
    }
}
```

Módosítsuk úgy a main() metódust, hogy megszerezze a referenciát az aktuális alkalmazástartományra, mielőtt az AppDomain.CurrentDomain tulajdonság használatával meghívna a PrintAllAssembliesInAppDomain() metódust.

A main() metódus elindít egy Windows Forms üzenetdobozt, hogy rákényszerítse a CLR-t a System.Windows.Forms.dll, a System.Drawing.dll és a System.dll szerelvények betöltésére (mindenképpen állítsunk be egy hivalkozást ezekre a szerelvényekre, és frissítsük ennek megfelelően a using utasításokat):

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with AppDomains *****\n");
    // Az aktuális alkalmazástartomány információinak beolvasása.
    AppDomain defaultAD = AppDomain.CurrentDomain;
    // A hívás csak további szerelvényeket tölt be
    // az alkalmazástartományba.
    MessageBox.Show("Hello");
    PrintAllAssembliesInAppDomain(defaultAD);
    Console.ReadLine();
}
```

A 17.6. ábra mutatja a kimenetet.



17.6. ábra: Szerelevények felsorolása az aktuális alkalmazástartományban

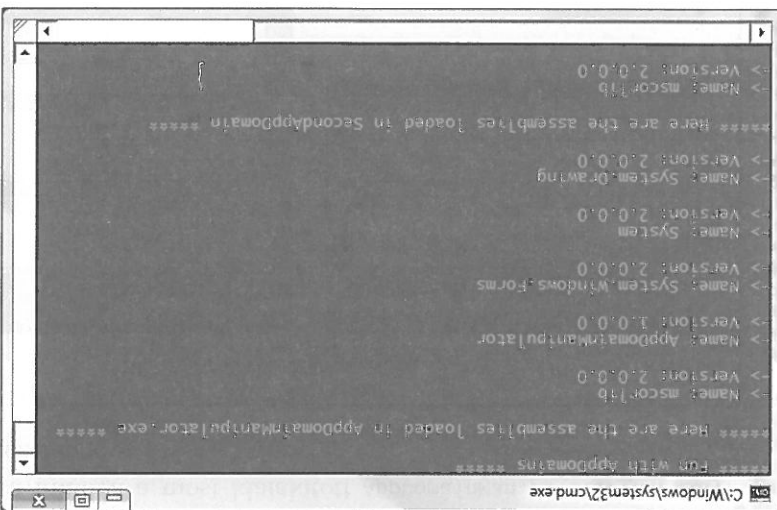
Új alkalmazástartományok programozott létrehozása

Egyetlen folyamat több alkalmazástartományt is hosztolhat. Meglehetősen ritkán kell manuálisan alkalmazástartományokat létrehozunk, ezt mégis megtehetjük a statikus `CreateDomain()` módszer hívásával. Az `AppDomain.CreateDomain()` többszörösen túlterhelt módszer. Legálább az új alkalmazástartomány barátságos nevet meg kell adnunk a következőképpen:

```

static void Main(string[] args)
{
    ...
    // új alkalmazástartomány létrehozása az aktuális folyamatban.
    AppDomain anotherAD = AppDomain.CreateDomain("SecondAppDomain");
    PrintAllAssembliesInAppDomain(anotherAD);
    Console.ReadLine();
}
  
```

Ha ismét futtatjuk az alkalmazást (lásd a 17.7. ábrát), figyeljük meg, hogy a `system.windows.forms.dll`, a `system.drawing.dll` és a `system.dll` szerelevények csak az alapértelmezett alkalmazástartományban töltődnek be. Egy Win32-programozó szemszögéből nézve ez meglepő lehet (azt várnánk ugyanis, hogy mindkét alkalmazástartomány ugyanazokhoz a szerelevényekhez fér hozzá). Egy szerelevény azonban az *alkalmazástartományba* töltődik, nem pedig közvetlenül magába a folyamatba.



17.7. ábra: Egyetlen folyamattal két alkalmazástartomány

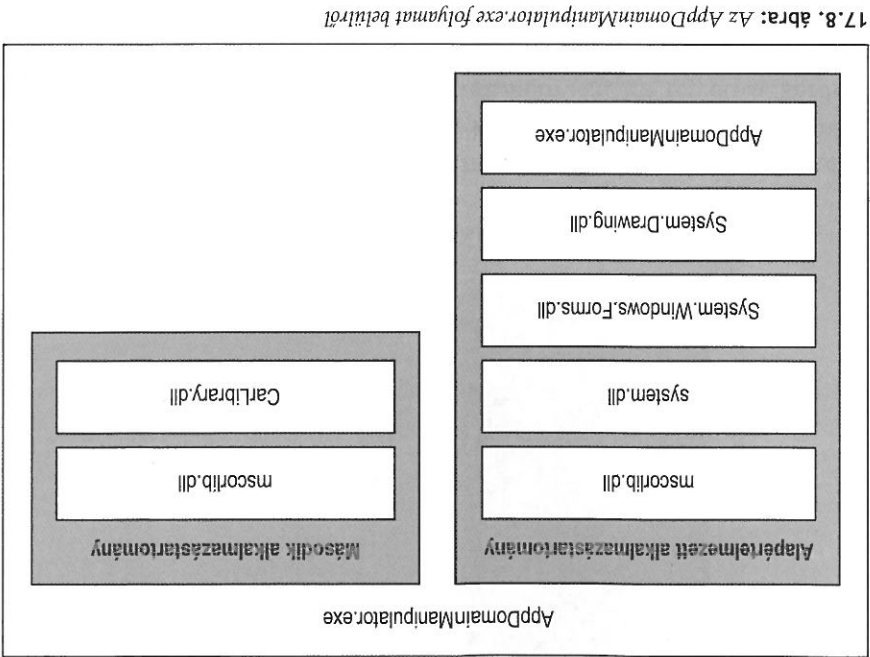
A secondAppDomain alkalmazástartomány magában foglalja az mscorlib.dll-től készített saját másolatát, ugyanis ezt a kulcsszerelevényt a CLR automatikusan betölti minden egyes alkalmazástartományba. Ez felveti a kérdést: Hogyan lehet programozottan betölteni egy szerelevényt egy alkalmazástartományba? Egyszerűen az AppDomain.Load() módszerrel (vagy az AppDomain.ExecuteAssembly() módszerrel, amely betölti és végrehajtja egy *.exe szerelevény main() módszerét).

vény main() metódusát).
Tételezzük fel, hogy átmasoltuk a carLibary.d11-t az AppomatiManiPulator.exe alkalmasz könyvtárába; ekkor ezt a következők szerint tölthetjük be a secondAppomati alkalmaszstartományba:

a SecondAppDomain alkalmazástörténet:

```
static void Main(string[] args)
{
    ...
    // carLibrary.dll betöltése az új alkatrészstartományba.
    AppDomain anotherAD = AppDomain.CreateDomain("SecondAppDomain");
    try
    {
        anotherAD.Load("CarLibrary");
        PrintAllAssembliesInAppDomain(anotherAD);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.ReadLine();
}
```

A 17.8. ábra mutatja a most kialakított AppDomainManipulator.exe folyamat belső szerkezetét.



Megjegyzés Ha hibakereső módban indítjuk az alkalmazást (az F5 lenyomásával), azt tapasztalhatjuk, hogy a Visual Studio hibakeresője által használt számos további szerverlány is betöltődik az alkalmazásstartományba. A projekt futtatása (Ctrl + F5) egyes alkalmazásstartományonként közvetlenül csak a szerverlányeket jeleníti meg.

Alkalmazásstartományok programozott eltávolítása

A CLR nem engedélyezi önálló .NET-szerverlányek eltávolítását. Az AppDomainUnload() módszer használataival azonban szelektíven eltávolíthatunk egy adott alkalmazásstartományt a hosztolófolyamattól. Az alkalmazásstartomány ilyenkor cserebe minden szerverlányt eltávolított.

Az AppDomain típus azoknak az eseményeknek a kis készletét határozza meg, amelyek egyike a domainunload. Ezt az eseményt akkor süttük el, amikor egy (nem alapértelmezett) alkalmazásstartományt eltávolítunk az öt tartalomzóról. A másik érdekes esemény a processexit, amely akkor sütt el, amikor az alapértelmezett alkalmazásstartomány eltávolítódik a folyamattól (ez nyilvánvalóan magával hozza magának a folyamatnak a befejezését is).

Ezért, ha programozottan szeretnénk eltávolítani az anotherad tartományt az AppDomainManipulator.exe folyamatból, valamint szeretnénk értesítést kapni a társított alkalmazástartomány eltávolításáról, akkor a következő eseménylogikát írhatjuk:

```
static void Main(string[] args)
{
    ...
    // Feliratkozás domainunload eseményre.
    anotherad.DomainUnload +=
        new EventHandler(anotherad.DomainUnload);
    // anotherad eltávolítása.
    AppDomain.Unload(anotherad);
    Console.ReadLine();
}
```

A domainunload eseményhez System.EventHandlerer típusú módszerreferencia tartozik, ezért az anotherad.DomainUnload() paraméterlistája a következő:

```
static void anotherad_DomainUnload(object sender, EventArgs e)
{
    Console.WriteLine("***** unloaded anotherad! *****\n");
}
```

Ha szeretnénk értesítést kapni akkor, amikor az alapértelmezett alkalmazás-tartomány eltávolítódik, módosítsuk a Main() metódust, hogy kezelje az alapértelmezett alkalmazástartomány ProcessEvent eseményét:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with AppDomains *****\n");
    AppDomain defaultAD = AppDomain.CurrentDomain;
    defaultAD.ProcessExit += new EventHandler(defaultAD.ProcessExit);
    ...
}
```

és definiáljunk egy megfelelő eseménykezelőt:

```
static void defaultAD_ProcessExit(object sender, EventArgs e)
{
    Console.WriteLine("***** unloaded defaultAD! *****\n");
}
```

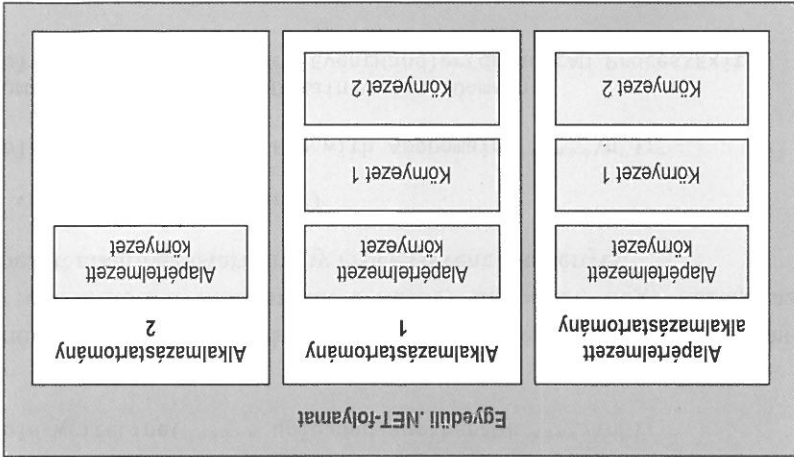
Forráskód Az AppDomainManipulator projekt megtalálható a 17. fejezet alkönyvtárában.

Az objektumkontextus határai

17. fejezet: Folyamatok, alkalmazástartományok és objektumkörnyezetek

Az alkalmazástartományok egy folyamaton belüli logikai partíciók a .NET-szerű objektumkontextusra. Röviden: egy .NET-kontextus egyetlen alkalmazástartományhoz tartozik. Biztosít lehetőséget arra, hogy külön környezetet létesítsen egy adott objektumnak.

A kontextus használatával a CLR úgy képes biztosítani azt, hogy a speciális futásidőbeli követelményekkel bíró objektumok kezelése megfelelő és következetes legyen, hogy elfogja a metódushívásokat, amelyek egy adott kontextusból indulnak, és abba érkezők. Ez az elfogórészt lehetővé teszi a CLR számára, hogy úgy állítsa be az aktuális metódushívást, hogy az megfelelően az adott objektum kontextusbeli beállításainak. Ha például egy olyan C#-osztályt definiálunk, amely automatikus szálbiztonságot követel meg (a [Synchronization] attribútum használatával), akkor a CLR a példányosítás során létrehoz egy „szinkronizált kontextust”.



17.9. ábra: Folyamatok, alkalmazástartományok és kontextushatárok

Ahogy egy folyamat meghatároz egy alapértelmezett alkalmazástartományt, úgy minden alkalmazástartományban van egy alapértelmezett kontextusa. Ez az alapértelmezett kontextus (amelyre néha úgy utalnak, hogy *context 0*, mivel mindig ez az alkalmazásban létrehozott első kontextus) csoportosítja azokat a .NET-objektumokat, amelyeknek nincs saját vagy egyedi kontextus-igényük. A .NET-objektumok nagy része a *context 0*-ba töltődik be. Ha a

CLR úgy dönt, hogy egy újonnan létrehozott objektumnak speciális igényei vannak, akkor új kontextushatárokat hoz létre a használt alkalmazástartományon belül. A 17.9. ábra (lásd fenn) illusztrálja a folyamat/alkalmazástartomány/kontextus kapcsolatot.

Kontextusváltozó és kötött típusok

Azokat a NET-típusokat, amelyek nem igényelnek speciális kontextust, *kontextusátlan* objektumoknak nevezzük. Ezeket bárhonnan el lehet érni a hozzátoló alkalmazástartományból anélkül, hogy zavart okoznának az objektum futásidejű követelményeiben. A kontextusátlan objektumok létrehozása nem különösebben nehéz, hiszen voltaképpen semmit nem kell csinálni (pontosabban: nem círázzuk az objektumokat semmilyen kontextusattribútummal, és nem származtatjuk a `system.contextboundobject` (összaltályból))

```
// A kontextusvártó objektum a context 0-ba töltődik.
class Sportscar{
```

Másrészt, azokat az objektumokat, amelyeknek kontextuslefoglalásra van szükségük, *kötött* objektumoknak nevezzük, és ezeknek a `system.context-boundobject` alaposztályból *kell* származniuk. Ez az alaposztály megőrösi azt, hogy a kérdéses objektum csak abban a kontextusban működhet megfelelően, amelyben létrehozták. A `.NET`-kontextus szerepe miatt, ha egy kötött objektum egy másik kontextusba kerül, az garantáltan valamilyen problémát fog okozni (természetesen a legrosszabbkor).

```
public class ContextAttribute :
    Attribute, IContextAttribute, IContextProperty
{
    public ContextAttribute(string name);
    public string Name { virtual get; }
    public object TypedId { virtual get; }
    public virtual bool Equals(object o);
    public virtual void
        Freeze(System.Runtime.Remoting.Contexts.Context
            public virtual int GetHashCode();
}
```

A [synchronization] attribútummal jelölt típusok szálbiztos kontextusba kerülnek. A myThreadSafeObject osztálytípus speciális kontextusigényeit figyelembe véve, képviseljük el, hogy mi történne akkor, ha egy lefoglalt objektumot szinkronizált kontextusból nem szinkronizált kontextusba helyeznénk át. Az objektum hirtelen nem lenne szálbiztos többé, és így az adatai hamar

```
// Ez a kötött típus csak szinkronizált,
// és emiatt szálbiztos kontextusba fog betöltődni.
[synchronization]
class SportsCars : ContextBoundObject
{
}
```

Tételezzük fel, hogy olyan osztályt szeretnénk meghatározni (SportsCars), amely természeténél fogva szálbiztos, noha nem kódoltunk szálszinkronizációt a tagimplementációkba. Ehhez származtassuk a ContextBoundObject osztályból, és alkalmazzuk a [synchronization] attribútumot a következők szerint:

Kötött objektum meghatározása

Megjegyzés Nem részletezzük az egyedi objektumkontextusok építését; erről a Jason Bock és Tom Barnaby által jegyzett Applied .NET Attributes (Apress, 2003) című könyvben bővebben is olvashatunk.

Mivel a ContextAttribute osztály nincs lezárva, készíthetünk saját kontextus-attribútumot (amely egyszerűen a ContextAttribute osztályból származik, és felülmúlja a szükséges virtuális metódusokat). Ezután készíthetünk egy olyan szoftvereszközt, amely válaszol a kontextusbeállításokra.

```
public virtual void GetPropertyForNewContext(
    System.Runtime.Remoting.Activation.
    IConstructionCallMessage ctorMsg);
public Type GetType();
public virtual bool IsContextOK(
    System.Runtime.Remoting.Contexts.Context ctx,
    System.Runtime.Remoting.Activation.
    IConstructionCallMessage ctorMsg);
public virtual bool IsDefaultAttribute();
public virtual bool IsNewContextOK(
    System.Runtime.Remoting.Contexts.Context newCtx);
public virtual bool Match(object obj);
public virtual string ToString();
}
```


inkonzisztenssé válhatnak, hiszen egy szál próbálna a most már nem szabványos objektum adatát manipulálni. Ahhoz, hogy biztosítsuk azt, hogy a CLR nem helyezi a sportscarts objektumokat szinkronizált kontextuson kívülre, egyszerűen származtassuk a contextboundobject osztályból.

Egy objektum kontextusának vizsgálata

Noha nagyon kevés alkalmazást kell programozottan kapcsolni a kontextussal, álljon itt mégis egy szemléltető példa. Hozzunk létre egy példamancsot, amely az alkalmazás egy kontextusval rendelkező osztályt (sportscart) és egy kötött típust (sportscarts) definiál:

```
using System.Runtime.Remoting.Contexts; // For Context type.
using System.Threading; // For Thread type.

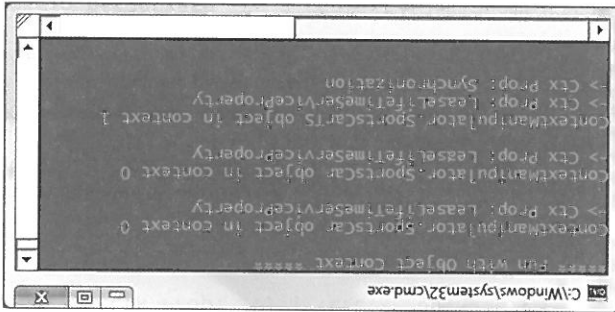
// A sportscar típusnak nincs speciális kontextusigénye,
// és ezért az alkalmazástomany
// alapértelmezett kontextusába fog betölteni.
class Sportscar
{
    public Sportscar()
    {
        // Context információ beszerzése, és a context ID kinyomtatása.
        Context ctx = Thread.CurrentContext;
        Console.WriteLine("{0} objektum in context {1}",
            this.ToString(), ctx.ContextID);
        foreach (IContextProperty ctxProp in ctx.ContextProperties)
            Console.WriteLine("> {0} ctx Prop: {1}", ctxProp.Name);
    }
}

// sportscarts típus megköveteli, hogy
// szinkronizált kontextusba töltsön.
class Sportscarts : ContextBoundObject
{
    public Sportscarts()
    {
        // Context információ beszerzése, és a context ID kinyomtatása.
        Context ctx = Thread.CurrentContext;
        Console.WriteLine("{0} objektum in context {1}",
            this.ToString(), ctx.ContextID);
        foreach (IContextProperty ctxProp in ctx.ContextProperties)
            Console.WriteLine("> {0} ctx Prop: {1}", ctxProp.Name);
    }
}
```

Minden konstruktor megszerez egy context típust az aktuálisan futó száltól a statikus Thread.CurrentContext tulajdonság révén. A context objektum használatával kiríthatjuk a kontextushatár statisztikáit, például a hozzárendelt azonosítót, valamint a context.contextProperties révén megszerzett leírók gyűjteményét. Ez a tulajdonság egy olyan objektumot ad vissza, amely implementálja az IContextProperty interfészt, amely a Name tulajdonság révén fel tárja az egyes leírókat. Módosítsuk a main() metódust úgy, hogy minden osztálypéldány egy példányát foglalja le:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Context *****\n");
    // Az objektumok a létrehozás során kontextusinformációkat
    // jelenítenek meg.
    Sportscar sport = new Sportscar();
    Console.WriteLine();
    Sportscar sport2 = new Sportscar();
    Console.WriteLine();
    SportsCars synchronSport = new SportsCars();
    Console.WriteLine();
}
```

Amint az objektum létrejön, az osztálykonstruktorok kirírták a kontextuscentrikus információk különböző részeit (lásd a 17.10. ábrát).



17.10. ábra: Egy objektum kontextusának vizsgálata

Mivel a Sportscar osztály nincs ellátva kontextusattribútummal, a CLR a sport és sport2 elemeket a context 0-ba foglalta le (vagyis az alapértelmezett környezetbe).

A sportscaRTS objektum azonban egyedi kontextushatárok közé került (amely az 1-es kontextuszazonosítót kapta), hiszen ez a kötött típus rendelkezik a [Synchronization] attribútummal.

Forráskód A contextmanipulátor projekt megtalálható a 17. fejezet alkönyvtárában.

A folyamat, alkalmazástartomány és a kontextus összefoglalása

A kulcspontok a CLR .NET-szerelvényének hosztolásához a következőképpen foglalhatók össze:

- Egy .NET-folyamat egy vagy több alkalmazástartományt hosztolhat. Minden alkalmazástartomány képes bármennyi kapcsolódó .NET-szerelvényt hosztolni. A CLR egy mástól függetlenül betöltheti és eltarthatja az alkalmazástartományokat (akár programozottan, a system.AppDomain típus révén).
- Egy adott alkalmazástartomány egy vagy több kontextusból áll. A kontextus használatával a CLR képes elhelyezni egy „speciális igényű” objektumot egy logikai tárolóba, hogy biztosítsa annak futásidejű követelményeit.

Az esetek többségében a .NET-futtatórendszer automatikusan kezeli a folyamatok, az alkalmazástartományok és a kontextusok részleteit. Ezek az információk szilárd alapot biztosítanak a többszálú programozás megértéséhez a .NET platformban.

Összefoglalás

Ebben a fejezetben megvizsgáltuk, hogy egy .NET végrehajtható fájl kód hosztolása hogyan történik a .NET platformon. Hogy a processzerek kezelése alkalmazkodjon a CLR igényeihez, a Win32 sokáig uralkodó koncepciója megváltozott.

Egy folyamat több alkalmazástartományt is hosztolhat, amelyek mind-egyike képes bármennyi kapcsolódó szerelvény hosztolására és végrehajtására. Továbbá egy szimpla alkalmazástartomány bármennyi kontextushatárt is tartalmazhat. Ezt a további izolációs szintet használva, a CLR képes a speciális igényű objektumok helyes kezelését biztosítani.

TIZENNYOLCADIK FEJEZET

Többszálú alkalmazások létrehozása

Az előző fejezetben a folyamatok, az alkalmazástartományok és a kontextusok közötti kapcsolatot vizsgáltuk meg. Ebben a fejezetben azzal foglalkozunk, hogy a .NET platform hogyan teszi lehetővé többszálú alkalmazások készítését, és hogyan tarthatjuk szálbiztosan a megosztott erőforrásokat.

Újra megvizsgáljuk a .NET-metódusreferencia típusát, majd megnézzük, hogy valójában hogyan támogatja az aszinkron függvényhívásokat. Ezzel a módszerrel automatikusan meghívhatunk egy metódust egy másodlagos végrehajtási szálon. Ezután a `system.threading` névtérben található típusokat tanulmányozzuk. Számos olyan típust (`Thread`, `ThreadStart` stb.) megvizsgálunk, amelyekkel könnyedén létrehozhatunk további végrehajtási szálakat. Ezenfelül megismerkedünk a `BackgroundWorker` típus használatával is, amely leegyszerűsíti a GUI-alapú alkalmazáskönyvezetben futó háttérműveletek végrehajtását.

A többszálú fejlesztés bonyolultsága persze nem a szálak készítésében rejlik, hanem annak a biztosításában, hogy a kódbizsisunk jól fel van-e szervezve a megosztott erőforrások párhuzamos hozzáféréseinek a kezeléséhez. Így a fejezet a .NET-keretrendszer különféle szinkronizációs primitíveit is bemutatja.

A folyamat, az alkalmazástartomány, a kontextus és a szál kapcsolata

Az előzőekben a *szálat* egy végrehajtható alkalmazás futási útvonalaként definiáltuk. Sok .NET-alkalmazás egyszerű, ám egy szereplvény elsődleges szála hozza létre a `Main()` lefutásakor) másodlagos végrehajtási szálakat hozhat létre további munkaegységek készítésére. A kiegészítő szálak megvalósításával könnyebben kezelhető (de az egymagos gépeken nem feltétlenül gyorsabb) alkalmazásokat építhetünk fel.

Megjegyzés Manapság az új számítógépek gyakran többmagos processzorral (de legalábbis hiperaszias egymagos processzorral) vannak felszerelve. A több szál párhuzamos futása nélkül a fejlesztők nem tudnák kihasználni a többmagos gépek maximális teljesítményét.

A `System.Threading` névtér olyan típusokat tartalmaz, amelyek segítségével többszálú alkalmazásokat készíthetünk. A `Thread` osztály az alaptípus, hiszen az egy adott szálát reprezentál. Ha programozottan szeretnénk hívatkozni egy adott tagot éppen végrehajtó szálra, hívjuk meg a statikus `Thread.CurrentThread` tulajdonságot:

```
static void ExecuteExecutingThread()
{
    // Megkapjuk a metódust éppen
    // futató szálát.
    Thread currThread = Thread.CurrentThread;
}
```

A .NET platformon *nem* közvetlen egy-egy kapcsolat van az alkalmazástartományok és a szálak között. Egy adott alkalmazástartományban valóban jó néhány szál lefuthat bármely időpontban. Továbbá egy adott szál élete során nincs egyetlen alkalmazástartományra korlátozva. A szálak szabadon átléphetik az alkalmazástartományok határait, ahogy azt a Win32-szálútemező és a CLR jónak látja.

Habár az aktív szálakat át lehet mozgatni az alkalmazástartományok határain, egy adott szál bármely időpillanatban csak egyetlen alkalmazástartományon belül futhat (más szóval, lehetetlen, hogy egy szál egy időben több alkalmazástartományban is működjön). Ha programozottan szeretnénk elérni az aktuális szálhoz tartozó alkalmazástartományt, hívjuk a statikus `Thread.GetDomain()` metódust:

```
static void ExtractAppDomainHostingThread()
{
    // Megszerezzük az aktuális szálhoz tartozó alkalmazástartományt.
    AppDomain ad = Thread.GetDomain();
}
```

Egy szálát bármikor át is helyezhetünk egy adott környezetbe, de a CLR is átmozgathatja egy új környezetbe. Ha meg akarjuk kapni azt az aktuális környezetet, amelyikben a szál éppen fut, használjuk a statikus `Thread.CurrentContext` tulajdonságot:

```
static void ExtractCurrentThreadContext()
{
    // Szerezzük meg azt a környezetet, amelyben
    // az aktuális szál működik.
    Context ctx = Thread.CurrentContext;
}
```

A CLR-nek az a feladata, hogy a szálakat alkalmazástartományokba és környezetekbe helyezze el (illetve vegye ki onnan). .NET-fejlesztőként szeretnénk csőre nem kell számon tartanunk, hogy egy szál hol fut (illetve, hogy mikor vált alkalmazástartományt). Ettől függetlenül meg kell ismernünk a mögöttes primitívek elérésének különféle módjait.

A konkurencia problémája

A többszálú programozás egyik kellémetlensége az, hogy kísérletekben tudjuk befolyásolni azt, hogy a mögöttes operációs rendszer vagy a CLR hogyan használja a szálakat. Ha például egy olyan kódblokkot készítettünk, amely új végrehajlási szálhoz létre, nem garantálható, hogy a szál azonnal elindul. A kód csak arra utasítja az operációs rendszert, hogy amint lehet, futtassa a szálát (ez általában akkor történik meg, amikor a szálütemező hozzáfoglal). Továbbá, mivel a szálak a CLR igényeinek megfelelően áthelyezhetők az alkalmazások és a környezetek határain, figyelniünk kell arra, hogy az alkalmazásunk mely műveletek *atomik* (a szálillékony műveletek a vesztélyesek). Tetelezzük felünk mely szempontból *nem szálbiztos* (pl. többszálú hozzáféréseknek alávetett), és mely műveletek *atomik* (a szálillékony műveletek a vesztélyesek). Tetelezzük fel például, hogy egy szál egy bizonyos objektum metódusát hívja. A szálütemező azonban a tevékenység felfüggesztésére utasítja ezt a szálát azért, hogy egy másik szál ugyanennek az objektumnak az azonos metódusát elérhesse.

Ha az eredeti szál nem fejezte be teljesén a műveletet, a második bemeneti szál egy részben módosított objektummal találkozhat. Ekkor a második szál alapvetően hamis adatot olvas, ez pedig furcsa (és nehezen megtalálható) hibához vezet, amelyeket még nehezebb reprodukálni és megkeresni. Másrésztől, az atomi műveletek többszálú környezetben mindig biztonságosak. Sajnos nagyon kevés olyan művelet van a .NET-alapoztálykönyvtárakban, amely garantáltan atomi. Még az értékek tagváltozókhoz rendelése sem atomi. Amíg a .NET-keretrendszer 3.5 SDK dokumentációja kifejezetten azt nem mondja, hogy egy művelet atomi, azt kell feltételeznünk, hogy szálillékony, és így óvintézkedéseket kell tennünk.

A szálszinkronizáció szerepe

A többszálú alkalmazástartományok önmagukban nem szálbiztosak, ugyanis (többé-kevésbé) egy időben több szál is működhet egy szerepben. Azért, hogy az alkalmazás erőforrásait a lehetőlegvesd adatátérülésöl megővják, a .NET-fejlesztőknek szálkezelési primitíveket (pl. zárolások [lock], megfigyelők [monitors] és a [synchronization] attribútum) kell igénybe venniük a végrehajtási szálakhoz való hozzáférés vezérléséhez.

Jóllehet a .NET platform nem képes teljes mértékben megszüntetni a nagy teljesítményű, többszálú alkalmazások létrehozásának nehézségeit, a folyamatot nagymértékben leegyszerűsíti. A system.Threading névtérben definiált típusok használatával különösebb nehézségek nélkül hozhatunk létre további szálakat. Ugyanígy, amikor zárójuk az adatok megosztott pontjait, találunk olyan kiegészítő típusokat, amelyek ugyanazt a működést biztosítják, mint a Win32 API szálkezelési primitívjei (amelyek jóval tisztább objektummodell alkalmaznak).

Nem csak a system.Threading névtér segítségével készíthetünk azonban többszálú .NET-programokat. A .NET-metódusreferencia vizsgálatakor (lásd 11. fejezet) már láttuk, hogy minden metódusreferencia képes a tagok aszinkron hívására. Ez az egyik fő előnye a .NET platformnak, hiszen a szálak létrehozásának az egyik leggyakoribb célja az, hogy nem blokkoló (aszinkron) módon hívassuk a metódusokat. Habár a system.Threading névtérrel hasonló eredményre jutunk, a metódusreferencia az egész folyamatot megkönnyíti.

A .NET-metódusreferenciák rövid áttekintése

A .NET-metódusreferencia típusa lényegében egy típusbiztos, objektumorientált függvénymutató. A .NET-metódusreferenciák deklarálásakor a C#-fordító úgy választ, hogy létrehoz egy, a system.MulticastDelegateból (amely a system.Delegateból származik) származó, lezárt osztályt. Ezek az osztályok biztosítják a metódusreferenciáknak, hogy metóduscsimlistát tarthassanak fenn a később hívható metódusokról. Nézzük meg a 11. fejezetben definiált BinaryOp metódusreferenciát:

```
// Egy C# metódusreferencia-típus.  
public delegate int BinaryOp(int x, int y);
```


Definíciója szerint a `BinaryOp` bármely olyan metódusra rá tud mutatni, amely vesz két egész számot (érték szerint) paraméterként, és egész számot ad vissza. Ha egyszer lefordítottuk, a definiáló szerelvény egy teljesen kiállított osztálydefiniációt fog tartalmazni, amelyet a metódusreferencia deklarációja alapján dinamikusán generálódott. A `BinaryOp` esetében ez az osztály többnyire a következőképpen néz ki (pszeudokód):

```
public sealed class BinaryOp : System.MulticastDelegate
{
    public BinaryOp(object target, uint functionAddress);
    public void Invoke(int x, int y);
    public IAsyncResult BeginInvoke(int x, int y,
        AsyncCallback cb, object state);
    public int EndInvoke(IAsyncResult result);
}
```

A `GenerateInvoke()` metódust a metódusreferencia-objektum metódusainak szintron hívására használjuk. Ezért a hívószálmak (pl. az alkalmazás elsődleges szála) meg kell várnia, hogy a metódusreferencia meghívása befejeződjön. A C#-ban az `Invoke()` metódust nem szükségessé közvelelenti kóddal meghívni, hanem közvetett módon is elindítható a háttérben, normál metódushívó szintaxissal. Nézzük meg a következő konzolprogramot (`syncdelegatereview`), amely a statikus `Add()` metódust hívja, szintron (blokkoló) módon:

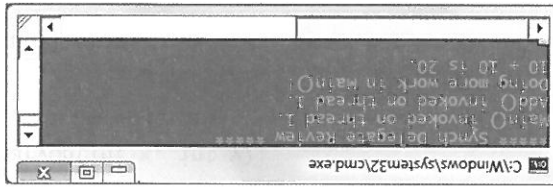
```
// szükség van erre a Thread.Sleep() meghíváshoz.
using System.Threading;
using System;

namespace SyncDelegate
{
    public delegate int BinaryOp(int x, int y);

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Sync Delegate Review *****");
            // A fűtő szálat azonosítójának kitérása.
            Console.WriteLine("Main() invoked on thread {0}.",
                Thread.CurrentThread.ManagedThreadId);
            // Az Add() metódus aktivizálása szintron módon.
            BinaryOp b = new BinaryOp(Add);
            // Lehetne írni azt is, hogy b.Invoke(10, 10);
            int answer = b(10, 10);
        }
    }
}
```

A program futtatásakor láthatjuk, hogy a `Console.WriteLine()` logika `main()` végrehajtása öt másodpercet késik. Habár sok (ha nem a legtöbb) metódust problémamentesen hívhatunk szinkron módon, a .NET-metódusreferenciákat tisztíthatjuk, hogy ha szükséges, aszinkron módon hívják meg a metódusait.

18.1. ábra: Metódusok szinkron (blokkoló) hívása



Először is ez a program a `System.Threading.Thread`-ing névtérben keresztül éri el a `Thread`-típust. Az `Add()` metóduson belül meghívjuk a statikus `Thread.Sleep()` metódust a hívással kb. öt másodperces felfüggesztésre, hogy egy hosszadalmas tevékenységet szimuláljunk. Mivel az `Add()` metódust szinkron módon hívjuk, a `main()` metódus addig nem írja ki a művelet eredményét, amíg az `Add()` metódus be nem fejeződik. Ezután a `main` metódus lekérdezi az aktuális szálat (a `Thread.CurrentThread` révén), és a `ManagedThread` tulajdonság segítségével kinyomtatja a szálat azonosítóját. Ugyanez a logika ismétlődik meg a statikus `Add()` metódusban. Mivel csak az elsődleges szálat dolgozik az alkalmazásban, a konzolon ugyanaz az azonosítóérték fog megjelenni (lásd 18.1. ábra).

```
// Ezek a sorok nem hajtódnak végre, amíg
// az Add() metódus be nem fejeződik.
Console.WriteLine("Doing more work in main()");
Console.WriteLine("10 + 10 is {0}", answer);
Console.ReadLine();
}

static int Add(int x, int y)
{
    // A futó szálat azonosítójának kinyomtatása.
    Console.WriteLine("Add() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
    // Szünet egy hosszú művelet szimulálására.
    Thread.Sleep(5000);
    return x + y;
}
}
```

A metódusreferenciák aszinkron viselkedése

Miért is szól pontosan az *aszinkron* függvényhívás? Egyes programozási műveletek időbe telnek. Képzeljük el, hogy egy olyan egyszerű alkalmazást készítünk, amely egy távoli objektum metódusát hívja meg, sokáig futó adatbázis-lekérdezést hajt végre, letölt egy nagyméretű dokumentumot, vagy egy 500 soros szöveget ír egy külső fájlba. A műveletek végrehajtása alatt kis időig úgy fog tűnni, hogy az alkalmazás lefagyott. Az aktuális feladat feldolgozásáig a program egyéb dolgokra (pl. a menü aktiválása, eszközöztárra kattintás vagy konzolos megjelenítés) nem válaszol.

Ezért a kérdés az, hogy hogyan utasítsuk a metódusreferenciát egy metódus külön végrehajtási szálra történő hívására, hogy több, egy időben lefutó feladatot szimuláljunk. Minden .NET metódusreferencia-típus automatikusan rendelkezik ezzel a képességgel. És ehhez *nem* szükséges közvetlenül ismerni a `System.Threading` névtér részleteit sem.

A `BeginInvoke()` és az `EndInvoke()` metódusok

Amikor a C#-fordító feldolgozza a `delegate` kulcsszót, a dinamikusan generált osztály definiálja a `BeginInvoke()` és az `EndInvoke()` metódusokat. A `BeginInvoke()` metódusreferencia alapján az előbbi metódusok prototípusa a következő:

```
public sealed class BinaryOp : System.MulticastDelegate
{
    ...
    // Metódus aszinkron hívására használatos.
    public AsyncResult BeginInvoke(int x, int y,
        AsyncCallback cb, object state);
    // Megszerezi az aktivizált metódus
    // visszatérési értékét.
    public int EndInvoke(AsyncResult result);
}
```

A `BeginInvoke()` metódusba elsőként beadott paraméterlista a `C#`-metódusreferencia formátumán alapul (a `BinaryOp` esetében ez két egész szám). A két utolsó argumentum mindig a `system.AsyncCallBack` és a `system.Object`. Hamarosan tanulmányozni fogjuk ezeknek a paramétereknek a szerepét; addig azonban állítsuk nulla mindkettőt. Az `EndInvoke()` visszaadott értéke a `BinaryOp` definíciója alapján egy egész szám, paramétere pedig az `IAsyncResult` típus.

A System.IAsyncResult interfész

A `BeginInvoke()` metódus mindig az `IAsyncResult` interfészt megvalósító objektumot adja vissza, míg az `EndInvoke()` egyedüli paraméterként egy `IAsyncResult`-kompatibilis típust igényel. A `BeginInvoke()` metódusból visszatérő `IAsyncResult`-kompatibilis objektum alapján véve egy társító mechanizmus, amely lehetővé teszi, hogy a hívó szál az `EndInvoke()` révén később elérje az aszinkron függvényhívás eredményét. A `system` névtérben definiált `IAsyncResult` interfész a következőképpen néz ki:

```
public interface IAsyncResult
{
    object AsyncState { get; }
    bool IsCompleted { get; }
    bool CompletedSynchronously { get; }
    bool IsCompleted { get; }
```

A leegyszerűbb eset az, amikor el tudjuk kérülni ezeknek a tagoknak a közvetlen meghívását. Csak annyit kell tennünk, hogy eltároljuk a `BeginInvoke()` metódusból visszatérő `IAsyncResult`-kompatibilis objektumot, majd továbbítjuk az `EndInvoke()` metódusba, amikor készen állunk a függvényhívás eredményének az eléréséhez. Ha pontosabban szeretnénk szabályozni egy visszaterési érték kiolvasását, lehetőségünk van erre az `IAsyncResult`-kompatibilis objektum tagjainak a meghívásával.

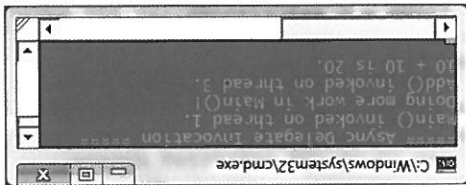
Megjegyzés Ha aszinkron módon hívunk egy void értéket visszaadó metódust, egyszerűen „tedd meg, és lépj tovább”. Ilyen esetekben soha nincs szükség az `IAsyncResult`-kompatibilis objektum gyorsítottárazására vagy az `EndInvoke()` első helyen történő hívására (mivel nincs visszakereshető érték).

Metódushívás aszinkron módon

Ahhoz, hogy utasítsuk a `BinaryOp` metódusreferenciát az `Add` aszinkron meghívására, módosítsuk az előző `Main` metódust az alábbiak szerint:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Async Delegate Invocation *****");
    // A futó szál azonosítójának kitírása.
    Console.WriteLine("Main() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
    // Az Add() meghívása egy másodlagos szálon.
    BinaryOp b = new BinaryOp(Add);
    Task<Result> t = b.BeginInvoke(10, 10, null, null);
    // A többi feladatot az elsődleges szálon végzzük el...
    Console.WriteLine("Doing more work in Main()!");
    // Amikor kész, szerezzük meg
    // az Add() metódus eredményét.
    int answer = b.EndInvoke(t);
    Console.WriteLine("10 + 10 is {0}.", answer);
    Console.ReadLine();
}
```

Ha lefuttatjuk ezt az alkalmazást, megfigyelhetjük, hogy két egyedi szálozostó jelenik meg, hiszen az aktuális alkalmazástartományban valójában több szál működik (lásd 18.2. ábra).



18.2. ábra: Aszinkron módon hívott metódusok

Az alkalmazás futása közben az egyedi azonosítókön kívül az is megfigyelhető, hogy a `Doing more work in Main()` üzenet azonnal megjelenik, miután a másodlagos szál a feladatával van elfoglalva.

A hívó szál szinkronizálása

A `main()` aktuális megvalósításán elgondolkodva, bizonyára nyilvánvalóvá vált, hogy a `BeginInvoke()` és az `EndInvoke()` hívása között eltelt idő kevesebb, mint öt másodperc. Így, miután a `Doing more work in Main()` megjelent a konzolon, a hívó szál blokkolódik, és várja, hogy a másodlagos szál befejeződjön, majd hozzájárulhasson az `Add()` módszer eredményéhez. Ezért ténylegesen még egy *szinkron hívást* csinálunk:

```
{
    static void Main(string[] args)
    {
        ...
        BinaryOp b = new BinaryOp(Add);
        AsyncResult iftar = b.BeginInvoke(10, 10, null, null);

        // Ez a hívás jóval kevesebb ideig tart, mint öt másodperc!
        Console.WriteLine("Doing more work in Main()!");

        // A hívó szál most blokkolva van, amíg
        // a BeginInvoke() be nem fejeződik.
        int answer = b.EndInvoke(iftar);
    }
    ...
}
```

Az aszinkron módszerreferenciák nyilvánvalóan elvesztenék az előnyüket, ha a hívó szál bizonyos körülmények között blokkolódna. Az `AsyncResult` interfész az `ISCOMPLETED` tulajdonsága révén teszi lehetővé a hívó szál számára, hogy ellenőrizze az aszinkron módon hívott módszer feladatának a teljesülését. Ennek a tagnak a segítségével a hívó szál meg tudja határozni, hogy az aszinkron hívás valóban befejeződött-e az `EndInvoke()` hívása előtt. Ha a módszer nem fejeződött be, az `ISCOMPLETED` értéke hamis, a hívó szál pedig folytathatja a feladatát. Ha az `ISCOMPLETED` értéke igaz, a hívó szál „a legkevesebb blokkoló módon” érte el az eredményeket. Gondoljuk végig a `main()` módszer alábbi módosítását:

```
{
    static void Main(string[] args)
    {
        ...
        BinaryOp b = new BinaryOp(Add);
        AsyncResult iftar = b.BeginInvoke(10, 10, null, null);

        // Ez az üzenet folyamatosan kiterődik, amíg
        // az Add() módszer be nem fejeződik.
        while(!iftar.IsCompleted)
        {
            Console.WriteLine("Doing more work in Main()!");
            Thread.Sleep(1000);
        }
    }
}
```

```
// Most már tudjuk, hogy az Add() metódus befejeződött.
int answer = b.EndInvoke(iftar);
...
}
```

Ide betrunk egy ciklust, amely folytatja a `console.WriteLine()` utasítás fel-
dolgozását, amíg a másodlagos szál be nem fejeződik. Ha ez megtörtént,
hozzáférhetünk az `Add()` metódus eredményéhez azzal a biztos tudattal,
hogy a metódus befejeződött. Ennek az alkalmazásnak a megfélelő működé-
séhez nem szükséges a `Thread.Sleep(1000)` hívása; azzal azonban, hogy az
egyres iterációk alatt kb. egy másodperces várakozásra kényszeríti az elsődle-
ges szálát, megelőzi, hogy több százszor megjelenjen ugyanaz az üzenet:

Az `AsyncResult` interfész az `IScriptable` tulajdonság mellett az `Async-
waitable` tulajdonság segítségével biztosít még rugalmasabb várakozási lo-
gikát. Ez a tulajdonság a `waitable` típus egy példányát adja vissza, amely
rendelkezik egy `waitone()` metódussal. A `waitable.waitone()` előnye az,
hogy megadhatjuk a maximális várakozási időt. Ha túllépjük a megadott
időkorlátot, a `waitone()` értéke hamis. Nézzük meg a következő, módosított
`while` ciklust, amely már nem hívja a `Thread.Sleep()` metódust:

```
while (iftar.AsyncWaitable.WaitOne(1000, true))
{
    console.WriteLine("Doing more work in main()");
}
```

Bár az `AsyncResult` fenti tulajdonságai a hívó szál szinkronizálását biztosít-
ják, ez nem a leghatékonyabb megközelítés. Az `IScriptable` tulajdonság sok
tekintetben olyan, mint egy bosszantó főnök, aki állandóan azt kérdi: Kész
vagy már? Szerencsére a metódusreferenciák számos további (és még elegendő-
sabb) technikát biztosítanak az aszinkron módon hívott metódusok eredménye-
nyének az eléréséhez.

Forráskód Az `AsyncDelegate` projekt a 18. fejezet alkönyvtárában található.

Az AsyncCallbac metódusreferencia szerepe

Nem kell lekérdézni a metódusreferenciát annak meghatározására, hogy be-
fejeződött-e egy aszinkron módon hívott metódus, jóval hatékonyabb, ha
utasítjuk a másodlagos szálát, hogy feladata befejezésekor értesítse a hívó
szálát. Ha engedélyezni szeretnénk ezt a viselkedést, létre kell hoznunk a

system.AsyncCallback metódusreferencia egy példányát a BeginInvoke() paramétereként, amely elédig null volt. Amikor azonban létrehozunk egy AsyncCallback objektumot, a metódusreferencia automatikusan meghívja a megadott metódust, ha az aszinkron hívás befejeződött.

Mint bármelyik metódusreferencia, az AsyncCallback is csak olyan metódusokat tud meghívni, amelyek megfelelnek egy adott formátumnak, ez ebben az esetben egy olyan metódus, amelynek egyedüli paramétere az AsyncCallback, és eredményként nem ad vissza semmit:

```
// Az AsyncCallback célpontjainak egyeznieük kell a következő
// formátummal.
void AsyncCallbackMethod(IAsyncResult itfAr)
```

Tételezzük fel, hogy van még egy alkalmazásunk, amelyik használja a BinaryOp metódusreferenciát. Ezúttal azonban nem kérdezzük le a metódusreferenciát, hogy eldöntsük, befejeződött-e az Add() metódus. Ehelyett definiálunk egy AsyncCallback() nevű statikus metódust, hogy értesítsen az aszinkron hívás befejezéséről:

```
namespace AsyncCallbackExample
{
    public delegate int BinaryOp(int x, int y);

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** AsyncCallbackExample *****");
            Console.WriteLine("Main() invoked on thread {0}.",
                Thread.CurrentThread.ManagedThreadId);

            BinaryOp b = new BinaryOp(Add);
            IAsyncResult itfAr = b.BeginInvoke(10, 10,
                new AsyncCallback(AddCallback), null);

            // Egyéb műveletek végrehajtása...

            Console.ReadLine();
        }

        static void AddCallback(IAsyncResult itfAr)
        {
            Console.WriteLine("AddCallback() invoked on thread {0}.",
                Thread.CurrentThread.ManagedThreadId);
            Console.WriteLine("Your addition is complete");
        }
    }
}
```

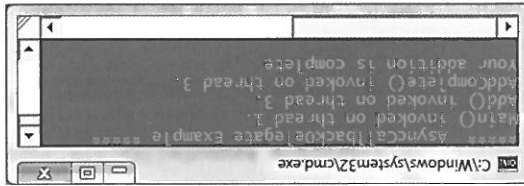


```

static int Add(int x, int y)
{
    Console.WriteLine("Add() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
    Thread.Sleep(5000);
    return x + y;
}
}
}

```

Tehát a statikus `AddComplete()` metódust az `AsyncCallback` metódusreferencia hívja meg, amikor az `Add()` metódus befejeződött. Ha lefuttatjuk a programot, látható, hogy a másodlagos szál hívja meg az `AddComplete()` metódust (lásd 18.3. ábra).



18.3. ábra: Az `AsyncCallback` metódusreferencia működés közben

Az `AsyncResult` osztály szerepe

Az előző példában láttuk, hogy a `Main()` metódus nem tárolja el a `BeginInvoke()` metódusból visszatérő `AsyncResult` típusú, és már nem hívja az `EndInvoke()` metódust. Ennek az az oka, hogy az `AsyncCallback` metódusreferencia célja (ebben a példában az `AddComplete()` nem fér hozzá a `Main()` hatókörében létrehozott, eredeti `BinaryOp` metódusreferenciához. Akár statikusként is deklarálhatnánk a `BinaryOp` változót, hogy mindkét metódus hozzáférjen, sokkal elegánsabb megoldás az `AsyncResult` bemenő paraméter használata.

Az `AsyncCallback` metódusreferencia által mutató metódusnak átadott bemeneti `AsyncResult` paraméter valójában a `System.Runtime.Messaging.Messaging` névtérben definiált `AsyncResult` osztály (hüanyagzik az `I` prefixum) egy példánya. A statikus `AsyncDelegate` tulajdonság az eredeti, máshol létrehozott aszinkron metódusreferenciára való referenciát adja vissza. Ezért, ha a `Main()` metódusban található `BinaryOp` metódusreferencia-objektum referenciáját szeretnénk elérni, egyszerűen kasztooljuk az `AsyncDelegate` tulajdonság által visszaadott `System.Object`-et a `BinaryOp` típusra. Ekkor a várt módon érhetjük el az `EndInvoke()` lefutását:

```
// Ne felejtsük el importálni a
// System.Runtime.Remoting.Messaging névteret
static void AddComplete(IAsyncResult itfAR)
{
    Console.WriteLine("AddComplete() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
    Console.WriteLine("Your addition is complete");
    // Most megkapjuk az eredményt.
    AsyncResult ar = (AsyncResult)itfAR;
    BinaryOp b = (BinaryOp)ar.AsyncDelegate;
    Console.WriteLine("10 + 10 is {0}.", b.EndInvoke(itfAR));
}
```

Egyedi állapotadatok átadása és fogadása

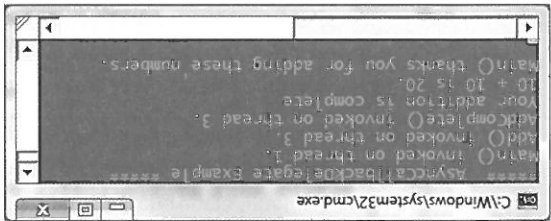
Az aszinkron módszerreferenciákkal kapcsolatosan meg kell még említenünk a `BeginInvoke()` módszer utolsó argumentumát (amely eddig null volt). Ennek a paraméternek a segítségével az elsődleges számból adhatunk át további állapotinformációkat a visszahívott módszerbe. Mivel ez az argumentum a `System.Object` egy prototípusa, bármilyen típusú adatot átadhatunk számára mindaddig, amíg a visszahívott módszer tudja, hogy mire számíthat. Együk fel például, hogy az elsődleges szál egy egyedi szöveges üzenetet szeretne beadni az `AddComplete()` módszerbe:

```
static void Main(string[] args)
{
    ...
    IAsyncResult itfAR = b.BeginInvoke(10, 10,
        new AsyncCallback(AddComplete),
        "Main() thanks you for adding these numbers.");
    ...
}
```

Ahhoz, hogy az `AddComplete()` hatókörén belül érjük el ezt az adatot, használjuk a `bemenni IAsyncResult paraméter AsyncState` tulajdonságát. Ekkor explicit kasztolásra lesz szükségünk, ezért az elsődleges és a másodlagos száalaknak meg kell egyezniük az `AsyncState` eredményeként kapott mögöttes típusban.

```
static void AddComplete(IAsyncResult itfAR)
{
    ...
    // kérjük le az információk objektumot, és kasztoljuk sztringbe.
    string msg = (string)itfAR.AsyncState;
    Console.WriteLine(msg);
}
```

A 18.4. ábra az aktuális alkalmazás kimenetét mutatja be.



18.4. ábra: Egyedi állapotadatokat átadása és fogadása

A következőkben megvizsgáljuk, hogy hogyan történik a System.Threading névtér segítségével a közvetlen szinkronizáció a száakkal.

Forráskód Az AsyncCallbackExample projekt a 18. fejezet alkönyvtárában található.

A System.Threading névtér

A .NET platform system.threading névtére számos olyan típust tartalmaz, amely lehetővé teszi a többszálú alkalmazások közvetlen létrehozását. A CLR-száakkal történő szinkronizációt lehetővé tévő típusok biztosítása mellett ez a névtér olyan típusokat is definiál, amelyek révén hozzáférhetünk a CLR-karbantartású szálkészlethez, egy egyszerű (nem GUI-alapú) timer osztályhoz, illetve számos olyan típushoz, amelyek segítségével szinkron módon elérhetjük a megosztott erőforrásokat. A 18.1. táblázat a névtér néhány meghatározó tagját sorolja fel. (További részletekért tekintsük át a .NET-keretrendszer 3.5 SDK verzió dokumentációját.)

Típus	Szerepe
Interlocked	Ez a típus atomi műveletekkel szolgál a több szál által megosztott típusok számára.
Monitor	Ez a típus szinkronizálja a szálobjektumokat, zárolásokat és várakozási jelek segítségével. A C# Lock kulcsszava a hálterben a Monitor típust használja.
Mutex	Ezt a szinkronizációs primitívet az alkalmazástartományok határainak szinkronizációjára használjuk.

Típus

Szerepe

ParameterizedThreadstart Ez a metódusreferencia lehetővé teszi, hogy a szálak olyan metódusokat hívjanak, amelyek bármennyi argumentumot felvehetnek.

Semaphore Ezzel a típussal korlátozhatjuk egy erőforrást vagy bizonyos típusú erőforrásokat párhuzamosan használó szálak számát.

Thread Ez a típus egy olyan szálát reprezentál, amely a CLR-ben hajtódik végre. Ezzel további szálakat hozhatunk létre az eredeti alkalmazástartományban.

ThreadPool Ez a típus lehetővé teszi, hogy egy adott folyamat során a CLR által fenntartott szálkészlettel kommunikáljunk.

ThreadPriority Ez a felsorolás egy szál prioritásszintjét adja meg (legmagasabb, normál stb.).

Threadstart Ezzel a metódusreferenciával határozzuk meg, hogy egy adott szál melyik metódust hívja. Eltérően a ParameterizedThreadstart metódusreferenciától, a Threadstart által mutatott metódusoknak rögzített prototípusúaknak kell lenniük.

Threadstate Ez a felsorolás határozza meg az érvényes állapotokat, amelyeket egy szál felvehet (fűtő, megszakított stb.).

Timer Ez a típus olyan mechanizmust ad a kezünkbe, amellyel metódusokat futtathatunk előre meghatározott időközönként.

TimerCallback Ezt a metódusreferencia-típust a Timer típusokkal együtt használjuk.

18.1. táblázat: A System.Threading nevű forrásból típusai

A System.Threading.Thread osztály

A System.Threading.Thread névű legprimitívebb típusa a Thread. Ez az osztály egy adott alkalmazástartomány adott futási útvonalat beburkoló objektumorientált csomagolómodul. Ez a típus számos olyan metódust is definiál (statikus és példányszintű is), amelyek révén új szálat hozhatunk létre az aktuális alkalmazástartományban, illetve felfüggeszthetünk, leállíthatunk és megsemmisíthetünk egy adott szál. Nézzük meg a 18.2. táblázatban látható alapvető statikus tagok listáját.

Statikus tag

CurrentContext Ez az írásvédezt tulajdonság a szál aktuális futási környezetét adja vissza.

CurrentThread Ez az írásvédezt tulajdonság az aktuálisan futó szála történő hivatkozást adja vissza.

GetDomain() Ezek a metódusok az aktuális szál futtató aktuális alkalmazástartományra, illetve a tartomány azonosítójára történő hivatkozást adnak vissza.

Sleep() Ez a metódus adott időtartamra felfüggeszti az aktuális szál.

18.2. táblázat: A Thread típus legfontosabb statikus tagjai

A Thread osztály számos példányszintű tagot is támogat, amelyekből néhányat felsorolunk a 18.3. táblázatban.

Példányszintű tag

IsAlive Egy logikai értéket ad vissza, amely azt jelzi, hogy elkezdődött-e a szál futása.

IsBackground Lekérdezi vagy módosítja azt az értéket, amely azt jelzi, hogy a szál a háttérben fut-e vagy sem (erről lásd a fejezet későbbi részét).

Name Segítségével barátságos elnevezést adhatunk a szálnak.

Priority Lekérdezi vagy módosítja egy szál prioritását, amelyhez a ThreadPriority felsorolásból rendelhetünk értéket.

```
// Mindenképpen importáljuk a System.Threading névtérét.
static void Main(string[] args)
{
    Console.WriteLine("***** Primary Thread stats *****\n");
    // Az aktuális szál megszerzése és elnevezése.
    Thread primaryThread = Thread.CurrentThread;
    primaryThread.Name = "ThePrimaryThread";
}
```

Egy végrehajtható szerelevény belépési pontja (pl. a main() módszer) az elsődleges végrehajlási szálon fut. A thread típus alapvető használataának bemutatásához tételezzük fel, hogy van egy új, Threadstats nevű konzolalkalmazásunk. A statikus Thread.CurrentThread tulajdonság az aktuálisan futó szálát reprezentáló Thread típust adja vissza. Ha hozzáférünk az aktuális szálhoz, különféle statisztikákat jelenthetünk meg:

Hozzáférés az aktuális szál statisztikáihoz

Megjegyzés Egy aktív szál megszakítása vagy felfüggesztése általában nem szerencsés. Ha mégis azt tesszük, előfordulhat (bár nem nagy az esélye), hogy a szál terhelése „elszivárog” a megszakítások vagy a befejeződéskor.

18.3. táblázat: A Thread típus fontosabb példányszintű tagjai

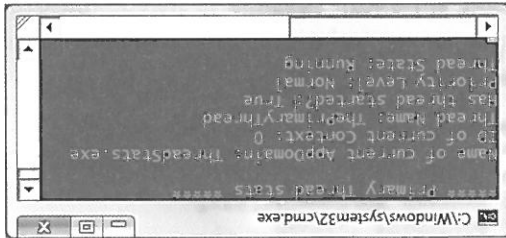
Suspend()	Felfüggeszti a szálát. Ha a szál már fel volt függesztve, a Suspend() hívására semmi sem történik.
Start ()	Útastja a CLR-t, hogy a lehető leghamarabb indítsa el a szálát.
Resume()	Tovább futtassa a korábban felfüggesztett szálát.
Join()	Blokkolja a hívó szálát, amíg az adott szál (amelyikre a join() módszert hívtuk) befejeződik.
Interrupt()	Megszakítja az aktuális szál várakozási periódusát (pl. aktíválja).
Abort()	Útastja a CLR-t, hogy amint lehet, szakítsa meg a szálát.
Threadstate	A szál állapotát adja meg, amelyhez a Threadstate felsorolásból rendelhetünk értéket.

Példányszintű tag	Szerepe
-------------------	---------

18. fejezet: Többszálú alkalmazások létrehozása

```
// A hosztoló alkalmazástomány/környezet részleteinek mutatása.  
Console.WriteLine("Name of current AppDomain: {0}",  
Thread.GetDomain().FriendlyName);  
Console.WriteLine("ID of current Context: {0}",  
Thread.CurrentContext.ContextID);  
  
// Kítr néhány statisztikát erről a szálról.  
Console.WriteLine("Thread Name: {0}",  
primaryThread.Name);  
Console.WriteLine("Has thread started?: {0}",  
primaryThread.IsAlive);  
Console.WriteLine("Priority Level: {0}",  
primaryThread.Priority);  
Console.WriteLine("Thread state: {0}",  
primaryThread.ThreadState);  
Console.WriteLine();  
}
```

A 18.5. ábra az aktuális alkalmazás kimenetét mutatja.



18.5. ábra: Statisztika gyűjtése a szálról

A Name tulajdonság

A Thread osztály támogatja a Name nevű tulajdonságot. Ha nem adjuk meg az értéket, a Name egy üres sztringet ad vissza. Am ha egyszerű hozzáféréseink egy barátságos sztringmonikert egy adott Thread objektumhoz, jóval egyszer-
rűbbé tehetjük a hibakeresési kísérleteket.

ID	Name	Location	Priority	Suspend
2768			0	
960	<No Name>		Highest	
5436	<No Name>		Normal	
5584			0	
5992	<No Name>		Normal	
5716	.NET SystemEvents		Normal	
4752	ThreadStats.Thread		0	
	ThreadStats.Program.Main		Normal	

18.6. ábra: Szálak hibakeresése Visual Studio 2008-ban

Há Visual Studio 2008-at használunk, a hibakeresés folyamán hozzáférhetünk a Threads ablakhoz (Debug > Windows > Threads). Ahogy a 18.6. ábrán fenn látható, a diagnosztizálni kívánt szálát gyorsan beazonosíthatjuk.

A Priority tulajdonság

A thread típus definiálja a priority nevű tulajdonságot. Az alapértelmezés szerint az összes szál prioritásszintje Normal. Ezt azonban bármikor megváltoztathatjuk a szál élettartama alatt a threadpriority tulajdonság és a kapcsolódó System.Threading.Threadpriority felsorolás segítségével:

```
public enum Threadpriority
{
    AboveNormal,
    BelowNormal,
    Highest,
    Idle,
    Lowest,
    Normal, // Alapértelmezett érték.
    Timercritical
}
```

Még ha át is állítjuk egy szál prioritásszintjét, akkor sem tudjuk közvetlenül befolyásolni azt, hogy a szálütemező mikor vált át egyik szárról a másikra. Valójában a szálak prioritásszintjére javaslatot ad a CLR-nek a szál aktivitásának fontosságáról. Így a Threadpriority. Highest értékkel bíró szál nem feltétlenül kap a legmagasabb precedenciát.

Ha szálütemező éppen egy adott feladattal foglalkozik (pl. objektumot szinkronizál, szálakat vált át vagy mozgat), a prioritásszint változásainak feladatnak megfelelően fog változni. Egyenlő feltételek mellett azonban a CLR kiolvassa ezeket az értékeket, és utasítja a szálütemezőt, hogyan lehetne a legjobban kiosztani az időszelvényeket. Az azonos prioritású szálaknak ugyanannyi idejük van a feladat elvégzésére.

Többnyire ritkán van szükség a szál prioritásszintjének közvetlen megváltoztatására. Akár olyan mértékben is felemelhetjük egyes szálak prioritását, hogy az alacsonyabb prioritású szálak nem jutnak processzoridőhöz, ebből következően csak óvatosan használjuk ezt a módszert.

Forráskód A threadstats projekt a 18. fejezet alkönyvtárában található.

Második szál létrehozása kódból

Ha bizonyos feladatok elvégzésére programozottan kívánunk további szálakat létrehozni, egy előre meghatározható folyamatot kell követni:

1. Hozzunk létre egy metódust, amely az új szál belépési pontja lesz.
2. Hozzunk létre egy új ParameterezettThreadstart (vagy Threadstart) metódusreferenciát úgy, hogy az 1. pontban definiált metódus címét továbbítsuk a konstruktornak.
3. Hozzunk létre egy Thread objektumot úgy, hogy konstruktorargumentumként továbbítsuk a ParameterezettThreadstart/Threadstart metódusreferenciát.
4. Adjunk meg kezdeti száljellemzőket (név, tulajdonság stb.).
5. Hívjuk meg a Thread.start() metódust. Ez elindítja a 2. lépésben létrehozott metódusreferencia által hívalkozott metódust, amint erre lehetőség nyílik.

Ahogy a 2. pontban említettük, két különböző metódusreferencia-típus segítséggel „rámutathatunk” arra a metódusra, amelyet a másodlagos szál fog végrehajtani. A Threadstart metódusreferencia a.NET 1.0 verziójának megjelölése óta a System.Threading névtér része, és bármely olyan metódusra rá tud mutatni, amely nem vesz fel argumentumokat, és nincs visszaadott értéke. Ez a metódusreferencia akkor lehet hasznos, ha a metódust csak arra tervezték, hogy a háttérben fusson, semmi más interakcióra.

A Threadstart nyilvántartó korláta az, hogy a segítségével indított szálnak nem tudunk paramétereket átadni. A.NET 2.0 óta létezik a ParameterezettThreadstart metódusreferencia-típus, amely lehetővé teszi egy System.Object típusú egyedi paramétert. Mivel bármit lehet System.Object típusként reprezentálni, egy egyedi osztályon vagy struktúrán keresztül bármennyi paramétert megadhatunk. A ParameterezettThreadstart metódusreferencia azonban csak a visszatérési érték nélküli (void) metódusokra tud rámutatni.

A ThreadStart metódusreferencia használata

Többszálú alkalmazás készítési folyamatainak (valamint hasznáinak) bemutatásához tétellezzük fel, hogy van egy konzolalkalmazásunk (SimpleMultiThreadApp), amely lehetővé teszi, hogy a végfelhasználó eldöntse, hogy az alkalmazás az egyedüli elsődleges szálát használja-e a feladatai elvégzésére, vagy megosztja a terhelést két külön végrehajtható szál között.

Ha importáltuk a System.Threading névtérre, az első lépés, hogy definiáljunk egy metódust, amely végrehajthatja a másodlagos szál feladatát. Ez a metódus ismétlődően kiír egy számsorozatot a konzolablakba, minden menetet körülbelül két másodpercre felfüggesztve. A Printer osztály teljes definíciója a következő:

```
public class Printer
{
    public void PrintNumbers()
    {
        // A szálinformáció megjelentése.
        Console.WriteLine("-> {0} is executing PrintNumbers()",
            Thread.CurrentThread.Name);
        // Számok kitiratása.
        Console.WriteLine("Your numbers: ");
        for(int i = 0; i < 10; i++)
        {
            Console.WriteLine("{0}, ", i);
            Thread.Sleep(2000);
        }
        Console.WriteLine();
    }
}
```

A main() metódusban először felkérjük a felhasználót annak meghatározására, hogy egy vagy két szál hajtása-e végre az alkalmazás feladatát. Ha a felhasználó egy szálát kér, egyszerűen hívjuk a PrintNumbers() metódust az elsődleges szálon belül. Ha azonban a felhasználó két szálát ad meg, hozzunk létre egy ThreadStart metódusreferenciát, amely a PrintNumbers() metódusra mutat, majd ezt a metódusreferencia-objektumot továbbítjuk egy új Thread objektum konstruktorába, és hívjuk a Start() metódust, amellyel értesítjük a CLR-t, hogy ez a szál felkészült a feldolgozásra.

Először adjunk referenciát a System.Windows.Forms.dll szerelvényre, majd a MessageBox.Show() segítségével jelenítsünk meg egy üzenetet a main() metódusban (a program lefuttatásakor ki fog derülni, miért jó ez). A main() metódus teljes megvalósítása a következő:

Ha lefutattuk a programot egy szállal, láthatjuk, hogy az utolsó üzenetdoboz nem jeleníti meg addig az üzenetet, amíg a teljes számsorozatot meg nem jelenít a konzol. Mivel a futás minden egyes szám megjelenése után mintegy két másodpercet felfüggesztődik, a végfelhasználói élmény elég lehangoló. Ha azonban két szálát választunk, az üzenetdoboz azonnal megjelenik, mivel egy egyedi thread objektum lesz felelős azért, hogy a számok a konzolon megjelenjenek (lásd 18.7. ábra).

```
static void main(string[] args)
{
    Console.WriteLine("***** The Amazing Thread App *****\n");
    Console.WriteLine("Do you want [1] or [2] threads? ");
    string threadCount = Console.ReadLine();

    // Nevezzük el az aktuális szálát.
    Thread primaryThread = Thread.CurrentThread;
    primaryThread.Name = "Primary";

    // A szálinformáció megjelenítése.
    Console.WriteLine("<-> {0} is executing main()",
        Thread.CurrentThread.Name);

    // Dolgozószál létrehozása.
    Printer p = new Printer();
    switch(threadCount)
    {
        case "2":
            // Most hozzuk létre a szálát.
            Thread backgroundThread =
                new Thread(new ThreadStart(p.PrintNumbers));
            backgroundThread.Name = "Secondary";
            backgroundThread.Start();
            break;
        case "1":
            p.PrintNumbers();
            break;
        default:
            Console.WriteLine("I don't know what you want...
                you get 1 thread.");
            goto case "1";
    }

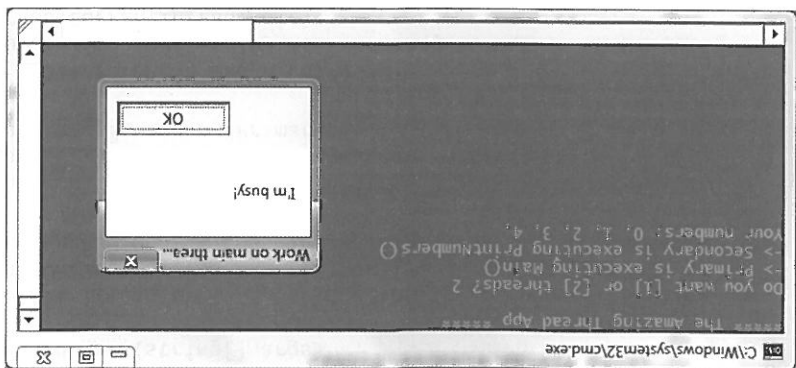
    // Végezzük el a további feladatokat.
    MessageBox.Show("I'm busy!", "Work on main thread...");
    Console.ReadLine();
}
```

A ParameterizedThreadStart módszerreferencia használata

Forráskód A SimpleMultiThreadApp projekt a 18. fejezet alkönyvtárában található.

Amikor egy processzoros gépen készítenünk többszálú alkalmazást (amelybe az aszinkron módszerreferenciák használata is beletartozik), az alkalmazás nem fog semmivel sem gyorsabban futni, mint a gép processzora. Akár egy, akár két szálát használva futtatjuk az alkalmazást, a számok mindig ugyanolyan sebességgel jelennek meg. Valójában a többszálú alkalmazások *válaszképessébb* alkalmazások. A végfelhasználó számára úgy tűnhet, hogy ez a program „gyorsabb”, de nem így van. A szálak nem tudják befolyásolni a foreach ciklusok végrehajtásának vagy a nyomtatásnak a sebességét, illetve nem tudják arra kényszeríteni a számokat, hogy rakétasebességgel adódjanak össze. A többszálú alkalmazások egyszerűen csak lehetővé teszik, hogy a terhelés több szál között legyen elosztva.

18.7. ábra: A többszálú alkalmazások a felhasználói inputokra gyorsabban reagáló alkalmazásokat eredményeznek



18. fejezet: Többszálú alkalmazások létrehozása

A ThreadStart módszerreferencia csak azokra a módszerekre tud rámutatni, amelyek void értéket adnak vissza, és nem vesznek fel argumentumokat. Ez ugyan a legtöbb esetben megfelelő, ám ha a másodlagos szálon futó módszerbe szeretnénk adatokat bevinni, a ParameterizedThreadStart módszerreferencia-típusra lesz szükségünk. Ehhez hozzuk újra létre a fejezet korábbi részében készített AsyncBacklogate projekt logikáját, de most a ParameterizedThreadStart módszerreferencia-típussal.

Elsőször is készítsünk egy új `AddWithThreads` nevű konzolalkalmazást, és importáljuk a `System.Threading` névtérét. Mivel a `ParameterizedThreadStart` bármely olyan metódusra rá tud mutatni, amely egy `System.Object` paramétert vesz fel, létrehozunk egy egyedi típust, amely tartalmazza a hozzáadandó számokat:

```
class AddParams
{
    public int a, b;

    public AddParams(int num1, int num2)
    {
        a = num1;
        b = num2;
    }
}
```

Ezután hozzunk létre egy statikus metódust a program osztályban, amely felvessz egy `AddParams` típust, és az értékek összegét jelenti meg:

```
static void Add(object data)
{
    if (data is AddParams)
    {
        Console.WriteLine("ID of thread in main(): {0}",
            Thread.CurrentThread.ManagedThreadId);
        AddParams ap = (AddParams)data;
        Console.WriteLine("{0} + {1} is {2}",
            ap.a, ap.b, ap.a + ap.b);
    }
}
```

A `main()` metódus forraskódja egyszerű. Egyszerűen csak használjuk a `ParameterizedThreadStart` metódusreferenciát a `ThreadStart` helyett:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Adding with thread objects *****");
    Console.WriteLine("ID of thread in main(): {0}",
        Thread.CurrentThread.ManagedThreadId);
    // készítsünk egy AddParams objektumot, amelyet átadunk
    // a másodlagos szálnak.
    AddParams ap = new AddParams(10, 10);
    Thread t = new Thread(new ParameterizedThreadStart(Add));
    t.Start(ap);
    ...
}
```

Előtérben és háttérben futó szálak

A továbbiakban az előtérbeli és háttérbeli szálak közti különbséget formálisan

- Az *előtérbeli szálak* képesek megakadályozni az aktuális alkalmazás befejezését. A CLR addig nem állít le egy alkalmazást (azaz nem távolítja el a memóriából a hosztoló alkalmazástartományt), amíg az összes előtérbeli szál be nem fejeződik.

- A *háttérszálakat* (más néven *démonszálakat*) a CLR feláldozható végrehajtási útvonalaknak tekintti, amelyeket bármikor figyelmen kívül lehet hagyni (még akkor is, ha éppen munkát végeznek). Így, ha az összes előtérbeli szál befejeződött, a háttérszálak automatikusan leállnak, amikor az alkalmazástartomány kikerül a memóriából.

Az előtérbeli és a háttérbeli szálak nem szinonimái az elsődleges és a dolgozó szálaknak. Az alapértelmezés szerint az összes, thread.start() metódussal létrehozott szál automatikusan *előtérbeli* lesz. Ez azt jelenti, hogy az alkalmazástartomány addig nem kerül ki a memóriából, amíg az összes végrehajtási szál be nem fejezte a munkáját. A legtöbb esetben pontosan ezt a viselkedést várjuk el. Tétellezzük azonban fel, hogy meg szeretnénk hívni a Printer.PrintNumbers() metódust egy másodlagos szálon, amelynek háttérszálként kellene viselkednie. Még egyszer: ez azt jelenti, hogy a metódusnak, amelyre a Thread típus rámutat (a Thread.start() vagy a ParameterezettThread.start() metódusok révén), képesnek kell lennie arra, hogy biztonságosan leálljon, mire az összes előtérbeli szál befejezi a munkáját. Ilyen szálakat konfigurálni pontosan olyan egyszerű, mint igaz értékre állítani az IsBackground tulajdonságot:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Background Threads *****\n");
    Printer p = new Printer();
    Thread bgThread =
        new Thread(new ThreadStart(p.PrintNumbers));
}
```

```
// Ez most egy háttérrel.  
backgroundThread.isBackground = true;  
backgroundThread.start();  
}
```

A `main()` módszer nem hívja a `console.readline()` módszert, hogy arra kény- szerítse a konzolt, hogy az `Enter` billentyű megnyomásáig látható maradjon. Így, amikor lefuttatjuk az alkalmazást, azonnal le is áll, mivel a `thread` objektumot háttérrelként konfiguráltuk. Mivel a `main()` módszer módosul ki az első- leges *előtérbeli* szál létrehozását, amint a `main()` algoritmus befejeződik, az alkalmazásstartomány kikerül a memóriából, mielőtt a másodlagos szál befeje- zette volna a munkáját. Ha azonban egy megfigyeléssel érvénytelenítjük azt a sort, amelyik megadja az `isBackground` tulajdonságot, az összes szám megjele- nik a konzolon, hiszen az összes előtérbeli szálnak be kell fejeznie a feladatát, mielőtt az alkalmazásstartomány kikerül a hosztoló folyamatból.

A legtöbb esetben akkor érdemes egy szálát háttérbeliként konfigurálni, ha a kérdéses dolgozószál olyan, nem kritikus feladatot lát el, amelyre nem lesz már szükség a program fő feladatainak befejezése után.

Forráskód A `backgroundThread` projekt a 18. fejezet alkönyvtárában található.

A konkurencia kérdése

Az összes, példaként használt, többszálú mintaaalkalmazás szálbiztos, hiszen a kérdéses módszereket egyetlen `thread` objektum hajtotta végre. Néhány alkal- mazásunk ilyen egyszerűsített természetű, ám a többszálú alkalmazásaink nagy része számos másodlagos szál is tartalmazhat. Mivel egy alkalmazásstartomány összes szála párhuzamos hozzáféréssel rendelkezik az alkalmazás közös adatai- hoz, kérdés: Mi történne, ha több szál ugyanazt az adatot manipulálná? Mint- hogy a száluitemező arra kényszeríti a szálakat, hogy véletlenszerűen felülgész- szék a munkájukat, mi történik, ha az A szálát leállítjuk, mielőtt teljesen befejezne a munkáját? Ez esetben a B szál inkonzisztens adatokat fog olvasni.

A konkurencia problémájának bemutatásához készítsünk egy újabb multi- threadedPrinting nevű C#-konzolalkalmazást. Ez az alkalmazás ismét a ko- rábban létrehozott `Printer` osztályt használja, de a `PrintNumbers()` módszer ezúttal kényszeríteni fogja az aktuális szálát, hogy egy véletlenszerűen generált időtartamra függessze fel a futását:

```

public class Printer
{
    public void PrintNumbers()
    {
        ...
        for (int i = 0; i < 10; i++)
        {
            // A szál pihentetése véletlenszerű időtartamig.
            Random r = new Random();
            Thread.sleep(1000 * r.Next(5));
            Console.WriteLine("{0}, ", i);
        }
        Console.WriteLine();
    }
}

```

A `main()` metódus felelős egy tíz, egyedi elnevezésű thread objektumot magába foglaló tömb létrehozásáért, amelyek közül mindegyik a `Printer` objektum *ugyanazon példányát* hívja:

```

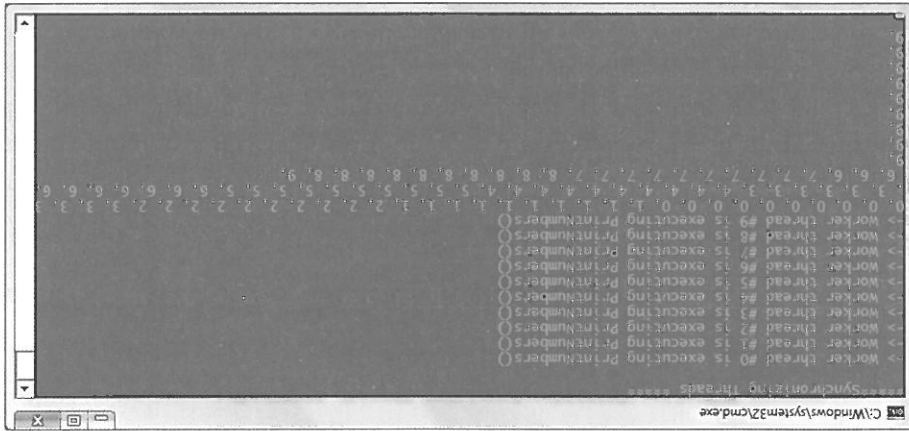
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("*****Synchronizing Threads *****\n");
        Printer p = new Printer();

        // készítünk 10 szálát, amelyek mind ugyanarra
        // a metódusra mutatnak ugyanazon az objektumon.
        Thread[] threads = new Thread[10];
        for (int i = 0; i < 10; i++)
        {
            threads[i] =
                new Thread(new ThreadStart(p.PrintNumbers));
            threads[i].Name = string.Format("Worker thread #{0}", i);
        }
        // Most indítsuk el mindegyiket.
        foreach (Thread t in threads)
            t.Start();
        Console.ReadLine();
    }
}

```

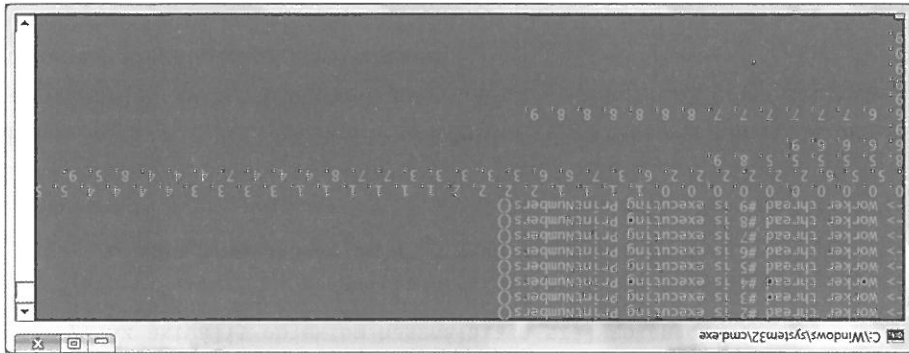
Mielőtt letesztelnénk a programot, gondoljuk végig újra a problémát. Ennek az alkalmazástartománynak az elsődleges szála azzal kezd meg az életét, hogy létrehoz tíz másodlagos dolgozószálát. Minden egyes dolgozószálnak az a feladata, hogy meghívja a `PrintNumbers()` metódust *ugyanazon* a `Printer`

példáyon. Minthogy előzőleg nem zároltuk az objektum megosztott erőforrásait (a konzolt), nagy az esélye annak, hogy az aktuális szál leall, mielőtt a `PrintNumbers()` metódus megjelentíthetné a teljes eredményt. Mivel nem tudjuk pontosan, hogy ez mikor történik meg (ha egyáltalán megtörténik), előre nem látható eredményekhez fogunk jutni. Előfordulhat például, hogy a 18.8. ábrán látható kimenetet kapjuk.



18.8. ábra: Konkurencia a gyakorlatban, első eset

Futtassuk le néhányszor az alkalmazást. A 18.9. ábra egy másik lehetőséget mutat be (az eredmények nyilván eltérőek lesznek).



18.9. ábra: Konkurencia a gyakorlatban, második eset

Megjegyzés Ha nem tudunk előre nem látható kimenetet generálni, növeljük 10-ről 100-ra a szálok számát, vagy vezessünk be egy újabb `Thread.Sleep()`-hívást a programba. Végül min-denképpen szembekeverülünk a konkurencia problémájával.

Nyilvánvalóan akad ezzel némi gond. Ahogy minden egyes szál utasítja a `Printert` a numerikus adatok megjelenítésére, a szálütemező boldogan a hátterbe söpri a szálakat. Az eredményül kapott kimenet nem konzisztens. Arra van szükségünk, hogy a megosztott erőforrásokat programozottan kényszerítsük a szinkronizálai hozzáférésre. A `System.Threading` számos szinkronizáció-központú típust tartalmaz. A `C#` programozási nyelv külön kulcshozval is rendelkezik a megosztott adatok szinkronizálására a többszálú alkalmazásokban.

Szinkronizáció a C# lock kulcsszavával

Az egyik módszer, amelyet a megosztott erőforrások hozzáféréseinek szinkronizálására alkalmazhatunk, a C# lock kulcsszáva. Ezzel a kulcsszóval definiálhatunk egy utasításblokkot, amelyet a szálak között kell szinkronizálni. Így a blokk utasításaira újonnan ráfuto szálak nem tudják megszakkítani az aktuális szálát, ezáltal megakadályozni, hogy az befejezze a feladatát. A lock kulcsszó használatakor meg kell határoznunk egy *token* (objektumreferenciát), amelyhez hozzá kell jutniuk a szálaknak, ha be akarnak lépni a zárolás hatókörébe. Amikor megpróbálunk zárolni egy *private* példányszintű metódust, a locknak adjuk át egyszerűen az aktuális példány referenciáját:

```
private void SomePrivatemethod()  
{  
    // Az aktuális objektum használata a szá1 tokenjeként.  
    lock(this)  
    {  
        // Ezen a hatókörön belül minden kód szá1biztos.  
    }  
}
```

Ha azonban egy kódstartományt zárolunk egy *nyilvános* tagon belül, biztonságosabb (és egyben a legjobb) megoldás, ha deklaráciunk egy privát objektum-tagváltozót a zárolás tokenjeként:

```
public class Printer
{
    // zárolok token.
    private Object threadlock = new Object();
    public void printNumbers()
    {
        // használok a zárolok token.
    }
}
```