A C# 2008 és a .NET 3.5

Második kötet (20-33. fejezet)

Andrew Troelsen

SZAK KIADÓ

2009

A C# 2008 és a .NET 3.5 – Második kötet (20-33. fejezet és két függelék)
A negyedik kiadás magyarul, két kötetben
Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition
Andrew Troelsen

Fordította a SZAK Kiadó fordítócsoportja: Domoszlai László, Endrédi Gabriella, Erős Szilvia, Iváncsy Renáta, Szalay Márton, Tiber Melinda, Varga Ágnes, Varga Péter
Lektor: Dávid Zoltán, MVP és Gincsai Gábor, MVP
Magyar nyelvi gondozás: Laczkó Krisztina és Trepák Mónika

ISBN 978-963-9863-10-1

A könyv fordítása a Kilgray Kft. MemoQ (http://www.memoqtm.com) programjával készült, a szöveg helyességét és az elválasztásokat a MorphoLogic Helyesek nevű programjával ellenőriztük.

SZAK Kiadó Kft. ■ Az 1795-ben alapított Magyar Könyvkiadók és Könyvterjesztők Egyesülésének a tagja ■ 2060 Bicske, Diófa u. 3. ■ Tel.: 36-22-350-209 ■ Fax: 36-22-565-311 ■ www.szak.hu ■ e-mail: info@szak.hu ■ Kiadóvezető: Kis Ádám, e-mail: adam.kis@szak.hu ■ Főszerkesztő: Kis Balázs MCSE, MCT, e-mail: balazs.kis@szak.hu

Ezt a kiadást Mikkónak, a csodamacskának, a 412-es szám alatt eltöltött éveknek és csodálatos feleségemnek, Amandának ajánlom, aki türelmesen várta, hogy elkészüljek újabb könyvem megírásával.

Tartalomjegyzék

5. rész: Bevezetés a .NET alaposztálykönyvtáraiba

20. Fájlműveletek és elszigetelt tárolás 3

30. WPF 2D grafikus renderelés, erőforrások és témák 619

8. rész: Függelékek

Köszönetnyilvánítás

Jóllehet egyedül az én nevem jelenik meg a könyv borítóján, de a munka soha nem készülhetett volna el jó néhány tehetséges ember segítsége nélkül. A következőkben azokhoz szólok, akik lehetővé tették, hogy ezt a könyvet megírjam.

Mindenekelőtt köszönet illeti az Apress összes munkatársát, akikkel már több éve van szerencsém együtt dolgozni. Rendkívüli emberek, akik képesek voltak az én eredeti Word-dokumentumaimat tökéletes prózává varázsolni. Hálás köszönet érte. Már most nagyon várom, hogy mikor dolgozhatunk ismét együtt egy következő könyvön (de azért előtte szabadságolom magam egy időre).

Külön köszönöm technikai szerkesztőm, Gavin munkáját, aki annyi bölcs tanáccsal segített, hogy úgy érzem, ez a kiadás sokkal színvonalasabb lett, mint az előzőek. Mint mindig, az esetleges gépelési hibákért vagy technikai tévedésekért egyedül én vagyok a felelős.

Végül, de nem utolsósorban, köszönöm családomnak, barátaimnak és munkatársaimnak, hogy elviselték mogorva viselkedésemet, amely a kézirat elkészítésének utolsó szakaszaiban jó néhányszor jellemzett engem.

Bevezetés

Ez a könyv nagyjából 2001 nyara óta létezik, ha nem is ebben a formában: a C# és a .NET platform első kiadása ugyanis akkor jelent meg, a .NET 1.0 Beta 2 verziójával karöltve. Nagy örömmel és hálával tölt el, hogy munkám kedvező fogadtatásra lelt a sajtóban, és ami még fontosabb, az olvasók körében is. Az évek során jelölték a Jolt Award döntőjére (vesztettem... kellemetlen) és a 2003-as Referenceware Excellence Awardon a programozási könyvek kategóriájában (nyertem?... de jó!)

Azóta azon dolgoztam, hogy a könyvet frissítsem és aktualizáljam a .NET platform újabb verzióinak megfelelően. Eközben limitált darabszámmal megjelentettem egy speciális kiadványt, amely bemutatta a .NET 3.0 verzióját (Windows Presentation Foundation, Windows Coomunication Foundation, és a Windows Workflow Foundation), valamint áttekintést nyújtott olyan különböző technológiákról, amelyek még megjelenésre vártak, és amelyeket most LINQ néven ismerünk.

Jelenleg a könyv negyedik kiadását tartja a kezében az Olvasó. Ez a kiadás egyrészt tömören megismétli a korábbi ismereteket, azaz a .NET 3.5-ös verziójában található legfontosabb változások magyarázatát, másrészt számos teljesen új fejezetet is tartalmaz, de ezen túlmenően több eddigi rész is jelentősen kibővült.

Ahogy a korábbiakban megszokhattuk, ez a kiadás is egyszerű és érthető nyelven ismerteti a C# nyelvet és a .NET-alaposztálykönyvtárakat. Soha nem értettem azokat a szakírókat, akik úgy adják elő mondandójukat, hogy az jobban hasonlít egy GRE-szószedetről írt tanulmányra, mint egy szakkönyvre. Munkám továbbra is arra fekteti a hangsúlyt, hogy a benne található információk segítségével jól használható szoftverrendszereket lehessen készíteni, így nem töltök túl sok időt az ezoterikus részletek tanulmányozásával, hiszen ezekkel csak nagyon kevés olvasóm fog valaha is foglalkozni.

Mi egy csapat vagyunk, az Olvasó és én

A szakírók az emberek igényes csoportjának írnak (nekem már csak tudnom kell - egy vagyok közülük). Tisztában vagyunk azzal, hogy egy szoftver elkészítése - bármelyik platformmal történjen is (.NET, J2EE, COM stb.) - rendkívül aprólékos munka, ugyanakkor nagyon specifikus a részleg, a cég, az ügyfél vagy a tárgykör szerint. Hiszen lehet, hogy épp az elektronikus kiadói iparágban dolgozunk, vagy éppenséggel rendszert fejlesztünk az államnak, esetleg a helyi önkormányzatnak, a NASA-nak vagy egy katonai részlegnek. Én a magam részéről gyermekeket oktató szoftvert fejlesztettem, számos n-rétegű rendszert, valamint projekteket a gyógyászati és a pénzügyi ágazatokban. Majdnem nulla az esélye, hogy annak a kódnak, amelyet egyikünk ír a munkájához, köze lesz ahhoz a kódhoz, amelyet a másikunk ír a saját munkájához (kivéve, ha véletlen épp együtt dolgoztunk korábban!).

Ezért szándékosan kerülöm az olyan mintapéldákat, amelyeket konkrét iparághoz vagy programozási feladathoz köthetnénk: a C#, az objektumorientált programozás és a .NET 3.5 alaposztály könyvtárait ipari alkalmazásoktól független példákon keresztül mutatom be. Ahelyett, hogy minden példám adatok tömkelegét szolgáltatná, bérlistát számolna, vagy ehhez hasonlókkal szolgálna, az én példáimhoz mindenkinek van valami köze: autókról szólnak (ráadásként néhány geometriai struktúrával és alkalmazottakkal). És ez az a pont, ahol összetalálkozunk.

Az *én feladatom*, hogy legjobb tudásom szerint elmagyarázzam a C# programozási nyelvet és a .NET platform alapvető jellegzetességeit. Ugyanígy mindent megteszek azért, hogy ellássam a kedves Olvasót különféle eszközökkel és stratégiákkal, amelyek lehetővé teszik, hogy a könyv alapján folytatni lehessen a téma tanulmányozását.

Az *olvasó feladata* pedig, hogy a megszerezett tudást képes legyen alkalmazni saját programozási feladatában. Persze egyértelmű, hogy a projektek nem állatnevekkel ellátott automobilok körül forognak, de hát pont erről szólna az alkalmazott ismeret. Biztos lehet benne az Olvasó, hogy ha egyszer megértette a könyvben bemutatott elveket, ismereteinek birtokában olyan .NET-alkalmazásokat írhat, amelyek használhatók saját programozási környezetükben.

A könyv áttekintése

A *Pro C# 2008 and the .NET 3.5 Platform (A C# és a Microsoft .NET 3.5.)* negyedik kiadása logikailag nyolc különálló részből áll, s ezek mindegyike további fejezeteket tartalmaz. Jó néhány témakör, például az alapvető C#-szerkezetek, az objektumorientált programozás és a platformfüggetlen .NET fejlesztése kibővült a korábbiakhoz képest, saját fejezetet kapott. Az új kiadás a .NET 3.0–3.5 programozás tulajdonságait (LINQ , WCF , WPF , WF stb.) összefoglaló fejezeteket is tartalmaz.

Megjegyzés A magyar változat kiadásának előkészítése során a SZAK Kiadó úgy döntött, hogy az eredeti könyv tartalmát két kötetben jelenteti meg. Ezt a viszonylag nagy terjedelem indokolja. Az alábbiakban mind a két kötet tartalmát ismertetjük, így biztosítva, hogy az is képet kapjon a mű egészéről, akinek csak az egyik kötet kerül a kezébe. (Ezt a Bevezetést változatlan formában mind a két kötetben szerepeltetjük.)

Első kötet

1. rész: Bevezetés a C#-ba és a .NET platformba

Az első rész bemutatja a .NET platform alapvető természetét és számos olyan fejlesztőeszközt (köztük több nyílt forráskodút), amelyeket .NET-alkalmazások fejlesztéséhez használhatunk. Mindeközben néhány alap C# programozási nyelvi részlettel és a .NET-típusrendszerrel is megismerkedünk.

1. fejezet: A .NET filozófiája

Az első fejezet voltaképpen a könyv további részeinek a gerincét képezi. A hagyományos Windows-fejlesztések világának vizsgálatával kezdjük, és feltárjuk a korábbi hátrányokat is. A fejezet elsődleges célja azonban az, hogy megismerjünk jó néhány .NET-központú építőkövet, például az egységes nyelvi futtatót (Common Language Runtime – CLR), a közös típusrendszert (Common Type System – CTS), az egységes nyelvi specifikációt (Common Language Specification – CLS) és az alaposztálykönyvtárakat. Egyszerű betekintést nyerhetünk a C# programozási nyelvbe és a .NET-szerelvényformátumába, valamint áttekintést kapunk a .NET platform platformfüggetlen természetéről (a B függelék részletesen tárgyalja ezt a témakört).

2. fejezet: C#-alkalmazások készítése

A fejezet célja, hogy különféle eszközök és technikák ismertetésével bevezessen minket a C#-forráskódfájl fordításának folyamatába. Először a parancssoros fordító (`csc.exe`) és a C#-eredményfájlok használatát ismerhetjük meg. A fejezet további részében pedig számos forráskódszerkesztőt és integrált fejlesztési környezetet (integrated development environments - IDEs) tanulmányozhatunk, többek között a következőket: TextPad, Notepad++, SharpDevelop, Visual C# 2008 Express és a Visual Studio 2008. Ezenkívül számos olyan további programozási eszközt ismertetünk, amelyet bármely .NET-fejlesztőnek azonnal elő kell tudnia húzni a farzsebéből.

2. rész: A C# alapvető építőelemei

A 2. részben tárgyalt témakörök kiemelten fontosak, hiszen az itt megjelenő tudnivalókat folyamatosan használjuk függetlenül attól, hogy milyen jellegű .NET-szoftvert szándékozunk fejleszteni (webes alkalmazás, asztali grafikus alkalmazás, forráskódkönyvtárak, Windows-szolgáltatások stb.). Itt derül ki, hogy hogyan működnek a C# nyelv alapvető építőkövei, beleértve az objektumorientált programozás (OOP) részleteit is. Ez a rész ismerteti továbbá, hogy hogyan kell kezelni a futásidejű kivételeket, és bevezet minket a .NET szemétgyűjtő szolgáltatásának részleteibe is.

3. fejezet: A C# alapvető építőelemei, I. rész

Ebben a fejezetben kezdjük el részletesen tanulmányozni a C# programozási nyelvet. Megismerkedünk a `Main()` metódus szerepével és számos részlettel a .NET platform belső adattípusaival kapcsolatban, beleértve a szövegesadatkezelést a `System.String` és a `System.Text.StringBuilder` névterek alkalmazásával. Megvizsgáljuk továbbá az iterációs és a döntési szerkezeteket, a szűkítő és a bővítő műveleteket, valamint az `unchecked` kulcsszó használatát.

4. fejezet: A C# alapvető építőelemei, II. rész

A 4. fejezet befejezi az alapvető C#-szemléletek meghatározását, a típustúlterhelt metódusok létrehozásától az `out`, `ref` és `params` kulcsszavakon keresztüli paraméterek definiálásáig. Vizsgáljuk, hogyan kell adathalmazt létrehozni és kezelni, hogyan kell olyan adattípust készíteni, amelynek `null` is lehet az értéke (a `?` és a `??` operátorokkal), és megmutatjuk az *értéktípus* és a *referenciatípus* közötti különbséget is.

5. fejezet: Egységbe zárt osztálytípusok definiálása

Ebben a fejezetben az objektumorientált programozással (OOP) ismerkedhetünk meg a C# programozási nyelv használatán keresztül. Először meghatározzuk az OOP pilléreit (beágyazás, származtatás és polimorfizmus), majd a fejezet további részében azt nézzük meg, hogyan kell konstruktorok, tulajdonságok, statikus tagváltozók, konstansok és csak olvasható fájlok használatával robusztusosztály-típusokat létrehozni. Végezetül ismertetjük a részleges típus definícióját és a C# XML-kódjának dokumentációs szintaktikáját.

6. fejezet: A származtatás és a polimorfizmus

Ebben a részben az OOP maradék pilléreit (származtatás és polimorfizmus) vizsgáljuk meg; ezek a segítségével összetartozó osztálytípusok családját hozhatjuk létre. Megnézzük a virtuális metódusok és absztrakt metódusok (és az absztrakt alaposztályok) szerepét, valamint a *polimorf interfész* természetét. Végül, de nem utolsósorban, ez a fejezet ismerteti a .NET platform legfontosabb alaposztályának, a `System.Object` osztálynak a szerepét is.

7. fejezet: Strukturált hibakezelés

Ez a fejezet elsősorban azt mutatja be, mit kell tenni a strukturális kivételkezelés segítségével a futási időben keletkezett anomáliákkal. Nemcsak az ilyen jellegű problémák megoldására szolgáló C#-kulcsszavakat ismerjük itt meg (`try`, `catch`, `throw` és `finally`), hanem az alkalmazásszintű és a rendszerszintű kivételek közötti különbséget is. Megismerhetünk továbbá többféle, a Visual Studio 2008-ban található eszközt, amelyek segítségével hibakeresést végezhetünk a figyelmen kívül hagyott kivételek között.

8. fejezet: Az objektumok életciklusa

Ennek a résznek az utolsó fejezete azt tárgyalja, hogy hogyan kezeli a CLR a memóriát a .NET szemétgyűjtőjének a segítségével. Megérthetjük az alkalmazásgyökerek, az objektumnemzedékek és a `System.GCType` szerepét. Az alapok megismerése után, a fejezet további része az *eldobható objektumokkal* (az `IDisposable` interfészen keresztül) és a véglegesítő folyamattal (a `System.Object.Finalize()` metóduson keresztül) foglalkozik.

3. rész: Haladó programozási szerkezetek a C#-ban

A könyv 3. része ismerteti a C# nyelv haladóbb jellegű (de ugyancsak nagyon fontos) elveit. Az interfészek és metódusreferenciák vizsgálatával befejezzük a .NET típusrendszerével való ismerkedést. A továbbiakban betekintést nyerhetünk a generikus típusok és a számos új C# 2008 nyelvi elem szerepébe, valamint a LINQ-ba is.

9. fejezet: Interfészek használata

Ennek a fejezetnek az anyagát az interfészalapú programozást is magába foglaló objektumalapú fejlesztés alkotja. Megismerhetjük belőle, hogy hogyan kell olyan típusokat definiálni, amelyek többszörös viselkedést támogatnak, hogyan lehet ezeket a viselkedéseket futásidőben feltérképezni, és hogyan lehet szelektíven elrejteni bizonyos viselkedéseket az *explicit interfészmegvalósítás* segítségével. Továbbá a számos előre definiált .NET-interfésztípus megismeréséhez el kell sajátítanunk azoknak az egyedi interfészeknek a használatát, amelyek ad hoc jellegű eseményarchitektúra készítéséhez szükségesek.

10. fejezet: Gyűjtemények és generikus típusok

Ez a fejezet a `System.Collections` névtérgyűjtemény típusainak tanulmányozásával kezdődik; ezek már a .NET platform kezdeti kiadásának is részei voltak. Noha a .NET 2.0 megjelenése óta a C# programozási nyelv lehetővé teszi *generikus* típusok használatát, ahogy a későbbiekben kiderül, a generikus programozás lényegesen megnöveli az alkalmazás teljesítményét és a típusbiztonságot. Nemcsak azt vizsgáljuk meg, hogy milyen különböző generikus típusok léteznek a `System.Collections.Generic` névtérben, de azt is, hogyan készíthetjük el saját generikus metódusainkat és típusainkat (megszorításokkal vagy anélkül).

11. fejezet Metódusreferenciák, események és lambdák

A 11. fejezetnek az a célja, hogy érthetővé tegye a *metódusreferencia*-típusokat. Leegyszerűsítve: a .NET-metódusreferencia olyan objektum, amely az alkalmazás más metódusaira mutat. Ez a minta olyan rendszerek készítését teszi lehetővé, amelyek több objektum összekapcsolódását engedik meg kétirányú együttműködés segítségével. A .NET-metódusreferenciák vizsgálata után megismerkedünk a .NET `event` kulcsszavával, amelyet arra használunk, hogy egyszerűsítsük a nyers metódusreferencia programozásának kezelését. Végezetül megvizsgáljuk a C# 2008 lambda-operátorát (=>), és feltárjuk a kapcsolatot a metódusreferenciák, a névtelen metódusok és a lambda-kifejezések között.

12. fejezet: Indexerek, operátorok és mutatók

Ez a fejezet a C# programozási ismereteinket mélyíti el számos haladó programozási technika segítségével. Megismerhetjük, hogyan kell operátorokat túlterhelni, és hogyan kell a típusaink számára egyedi konverziós rutinokat írni (akár implicit, akár explicit formában). Megtanulhatjuk azt is, hogy hogyan kell létrehozni és használni a *típusindexelőket*, és hogyan kell kezelni a C-stílusú mutatókat a „nem biztonságos" kódkörnyezetben.

13. fejezet: C# 2008 nyelvi újdonságai

A .NET 3.5 kiadásában a C# nyelv számos új programozási eszközzel bővült, amelyek közül jó néhány arra szolgál, hogy lehetővé tegye a LINQ API használatát (erről a 14. fejezetben többet is megtudhatunk). Megismerjük a lokális változók implicit típussal történő ellátásának a szerepét, a részleges metódus, az automatikus tulajdonságok, a bővítő metódusok, a névtelen típusok és az objektuminicializáló szintaxis használatát.

14. fejezet: Bevezetés a nyelvbe ágyazott lekérdezésekbe (LINQ)

Ez a fejezet ismerteti a LINQ-et (Language Integrated Query - nyelvi integrált lekérdezés), amelyet nyugodtan nevezhetünk a .NET 3.5 legérdekesebb részének. Ahogy a fejezet későbbi részéből kiderül, a LINQ segítségével lehetőségünk nyílik erősen típusos *lekérdezéseket* írni, amelyeket számos LINQ-célpontra alkalmazhatunk, hogy a szó legtágabb értelmében „adatot" kezelhessünk. Megtanuljuk a LINQ objektumokra történő alkalmazását, ennek segítségével a LINQ-kifejezéseket adattárolókra (tömbök, kollekciók, egyedi típusok) fogjuk tudni alkalmazni. Ez a tudás nélkülözhetetlen lesz akkor, amikor azt vizsgáljuk meg, hogyan kell LINQ-kifejezéseket relációs adatbázisokon (a LINQ az ADO-n keresztül) és XML-dokumentumokon (a LINQ az XML-en keresztül) használni.

4. rész: Programozás .NET-szerelvényekkel

A negyedik rész a .NET-szerelvényformátum részleteit taglalja. Nemcsak azt tanulhatjuk meg, hogyan kell .NET-kódkönyvtárakat telepíteni és konfigurálni, de azt is, hogy mi a .NET bináris kép belső összeállítása. Ebben a részben lesz szó a .NET-attribútumok szerepéről és a többszálú alkalmazások létrehozásának módjáról is. A későbbi fejezetek néhány haladó témát érintenek, például az objektumkörnyezetet, a köztes nyelvi kódot és a dinamikus szerelvényeket.

15. fejezet: A .NET-szerelvények

Első megközelítésben a *szerelvény* nem más, mint egy felügyelt `*.dll` vagy `*.exe` bináris fájl leírására szolgáló kifejezés. Ám a .NET-szerelvények története sokkal több ennél. Ebben a fejezetben megtudjuk, mi az egyfájlos és a többfájlos szerelvények közötti különbség, és, hogy ezeket hogyan kell készíteni és telepíteni. Megvizsgáljuk, hogyan kell privát és megosztott szerelvényeket konfigurálni XML-alapú `*.config` fájlokkal és kibocsátó házirenddel rendelkező szerelvényekkel. Mindeközben megtudhatjuk, hogy milyen a GAC (Global Assembly Cache – globális szerelvénytár) belső struktúrája, és mi a .NET keretrendszer konfigurációs segédprogramjainak a szerepe.

16. fejezet: Típusreflexió, késői kötés és attribútumalapú programozás

A 16. fejezetben folytatjuk az ismerkedést a .NET-szerelvényekkel. A `System.Reflection` névtér használatával kapcsolatban azt vizsgáljuk, a futási időben hogyan valósul meg a típusmeghatározás folyamata. Az ilyen jellegű típusok használatával lehetőségünk van olyan alkalmazások létrehozására, amelyek futás közben képesek a szerelvények metaadatait olvasni. Kiderül, hogyan lehet a *késői kötés* segítségével futási időben dinamikusan betölteni és létrehozni típusokat. A fejezet utolsó témaköre – mind a szabványos, mind az egyedi – .NET-attribútumok szerepének ismertetése. Az egyes témakörök hasznosságának szemléltetésére a fejezet konklúziójaként bemutatunk egy kiterjeszthető Windows Forms-alkalmazást.

17. fejezet: Folyamatok, alkalmazástartományok és objektumkörnyezetek

A szerelvényekről szóló tudnivalók után ez a fejezet a betöltött .NET végrehajtható fájlok felépítését ismerteti. Az elsődleges cél a folyamatok, az alkalmazástartományok és a kontextusbeli határok közötti kapcsolat illusztrálása. Ezek alapozzák meg a következő fejezet témáját: a többszálú alkalmazások készítésének módját.

18. fejezet: Többszálú alkalmazások készítése

Ez a fejezet azt mutatja be, hogy hogyan kell többszálú alkalmazásokat készíteni, és számos olyan technikát ismertet, amelyeket a szálbiztos kód készítéséhez alkalmazhatunk. A fejezet elején átismételjük a .NET metódusreferencia-típusait, hogy megértsük, valójában hogyan támogatják a metódusreferenciák

az aszinkronfüggvény-hívásokat. Majd megvizsgáljuk a `System.Thread` névtérben található típusokat. Ezek között számos olyan van (`Thread`, `ThreadStart`), amelynek segítségével könnyen lehet újabb végrehajtási szálakat készíteni. Végül megismerkedünk a `BackgroundWorker` típussal, amely nagyban megkönnyíti a felhasználói felületről történő szálkészítést.

19. fejezet: A köztes nyelv (CIL) és a dinamikus szerelvények

A 4. rész utolsó fejezetének szerepe kettős. Először a korábbinál részletesebben megvizsgáljuk CIL szintaktikáját és szemantikáját, majd a `System.Reflection.Emit` névtér szerepét mutatjuk be. Ezeknek a típusoknak a használatával lehetőségünk nyílik olyan szoftver készítésére, amely képes .NET-szerelvényeket futásidőben generálni a memóriába. Az olyan szerelvényeket, amelyeket a memóriában definiálunk és futtatunk, *dinamikus szerelvényeknek* nevezzük.

Második kötet

5. rész: Bevezetés a .NET alaposztálykönyvtáraiba

Ha elérkeztünk ehhez a részhez, akkor már biztos tudásunk van a C# nyelvről és a .NET- szerelvényformátumok részleteiről. Az ötödik rész továbbfejleszti ezeket az ismereteket azáltal, hogy számos általánosan használt alaposztálybeli szolgáltatást mutat be, többek között a fájl I/O-t és az adatbáziselérést az ADO.NET segítségével. Ez a rész tárgyalja továbbá az elosztott alkalmazások készítésének témakörét a Windows Communication Foundation (WCF) segítségével, valamint azokat a munkafolyamat-engedélyezett alkalmazások készítését, amelyek a Windows Workflow Foundation (WF) API-t használják.

20. fejezet: Fájlműveletek és elkülönített tárolás

A `System.IO` névtér segítségével lehetőségünk van a számítógép fájl- és könyvtárszerkezetének elérésére. Ebben a fejezetben megismerjük, hogyan kell programozottan könyvtárrendszert létrehozni (és törölni), valamint, adatokat különböző folyamokba (fájlalapú, sztringalapú, memóriaalapú stb.) be-, illetve kimozgatni. A fejezet utolsó része megvizsgálja az *elkülönített tárolást*, amelynek segítségével lehetőségünk van biztonságos helyen külön tárolni az adatokat függetlenül attól, hogy a célgép biztonsági beállításaitól. Hogy megértsük a `System.IO.Isolated` API bizonyos elveit, áttekintést kell kapnunk a kódhozzáférés-szabályozásról (Code Access Security - CAS).

21. fejezet: Bevezetés az objektumsorosítás világába

Ez a fejezet a .NET-platform objektumsorosító szolgáltatásába nyújt betekintést. Egyszerűen szólva a *sorosítás* segítségével lehetőségünk nyílik arra, hogy egy objektum (vagy a kapcsolódó objektumok egy halmazának) állapotát egy folyamban tároljuk a későbbi használat céljából. A *deszerializálás* (a sorosítás ellentétes művelete) az a folyamat, amelynek segítségével kivehetjük az objektumot a folyamból, és a memóriában az alkalmazás számára használható formába hozhatjuk. Az alapok elsajátítása után megtanulhatjuk, hogyan kell egyedivé tenni a sorosítási folyamatot az `ISerializableinterface` és a .NET-attribútumok halmazának segítségével.

22. fejezet: ADO.NET 1. rész: Az élő kapcsolat

Az adatbázisokról szóló két fejezet közül az elsőben az ADO.NET programozási API-ról kaptunk ismereteket. Ez a rész bevezeti a .NET adatszolgáltató szerepét, valamint azt, hogy hogyan kell a relációs adatbázissal kommunikálni az ADO.NET *élő kapcsolatán* keresztül. Ez a kapcsolati objektumokat, a parancsobjektumokat és az adatolvasó objektumokat jelenti Ezenkívül ez a fejezet mutatja be az egyedi adatbázis létrehozásának módját, és ismerteti azokat az adatelérési könyvtárakat, amelyeket majd a könyv további részében használunk.

23. fejezet: ADO.NET 2. rész: A bontott kapcsolat

Ez a fejezet az adatbázis-kezelés különböző módokon történő manipulálási lehetőségeit tárgyalja tovább, jelen esetben az ADO.NET *bontott kapcsolaton* keresztül. Megismerkedhetünk a `DataSet` típus és az adatfeldolgozó objektumok szerepével, valamint számos Visual Studio 2008 eszközzel, amelyek nagyban megkönnyítik az adatvezérelt alkalmazások létrehozását. Mindeközben rávilágítunk arra, hogyan kell a `DataTable`-objektumokat felhasználói felületi elemekhez kapcsolni, olyanokhoz, mint például a Windows Forms API `GridView` típusa.

24. fejezet: A LINQ API programozása

A 24. fejezet a LINQ programozási modellt mutatja be, pontosabban a LINQ „az objektum irányába" részét. Megvizsgáljuk, hogyan kell LINQ-lekérdezéseket alkalmazni relációs adatbázisokon, `DataSet`-objektumokon és XML-dokumentumokon. Mindeközben megtanuljuk az adatkontextus-objektum, valamint az `sqlmetal.exe` segédprogram szerepét, és megvizsgáljuk a Visual Studio 2008 különböző LINQ-támogató funkcióit.

25. fejezet: A WCF

A .NET 3.0 bevezetett egy teljesen új API-t, a WCF-et, amely lehetőséget ad arra, hogy szimmetrikus módon, függetlenül az alatta található rétegek felépítésétől elosztott alkalmazást fejlesszünk. Ez a fejezet feltárja a WCF-szolgáltatások, -gazdagépek és -kliensek készítésének lehetőségeit. A WCF-szolgáltatások rendkívül flexibilisek, a gazdagépek és a kliensek XML-alapú konfigurációs fájlokat használhatnak, hogy deklaratív módon adjanak meg címeket, kötéseket és szerződéseket.

26. fejezet: A Windows Workflow Foundation — Bevezetés

A .NET 3.0 a WCF mellett bevezette a WF API-t is, amely lehetőséget ad arra, hogy *munkafolyamatokat* definiáljunk, futtassunk és monitorozzunk. Ennek segítségével komplex üzleti folyamatok modellezhetők. Megtanuljuk a WF általános célját, ahogy azt is, hogy mi a szerepe az aktivitásoknak, a munkafolyamat tervezőnek, a munkafolyamat futtatási motornak, és a munkafolyamat-engedélyezett forráskód könyvtáraknak.

6. rész: Felhasználói felületek

Általánosan hibás feltételezés a .NET platformmal kapcsolatban az, hogy ez a keretrendszer kizárólag a webalapú felhasználói felületekkel foglalkozik (ez feltehetően a „.NET" kifejezés miatt van, hiszen felidézi az „Internet" szót, azaz a „webprogramokat"). A .NET kiemelkedő módon támogatja a webalkalmazások készítését, a könyv ezen része azonban a hagyományos felhasználói felületekre koncentrál két grafikus felület keretrendszerén, a Windows Forms és a WPF használatán keresztül.

27. fejezet: Windows Forms-programozás

Az eredeti asztali grafikus felhasználói felület, amely a .NET platformmal együtt kapható, a *Windows Forms*. Ez a fejezet ismerteti a felhasználói felület keretrendszerének szerepét, és bemutatja, hogyan kell főablakokat, párbeszédablakokat és menürendszereket készíteni. Feltárja továbbá az űrlapszármaztatás szerepét, és azt, hogy hogyan kell kétdimenziós adatot rajzolni a `System.Drawing` névtér segítségével. A témakör illusztrálására a fejezet végén készítünk egy (korlátozott funkcionalitással bíró) rajzolóalkalmazást.

28. fejezet: A WPF és az XAML

A .NET 3.0 teljesen új grafikus felhasználói felületet vezetett be, amelyet *WPF*-nek nevezett el. Röviden, a WPF kimagaslóan interaktív és médiában gazdag front endek készítését teszi lehetővé asztali alkalmazásokhoz (és indirekt módon webalkalmazásokhoz). A Windows Formsszal ellentétben, ez a túltelített felhasználóifelület-keretrendszer számos kulcsfontosságú szolgáltatást (2D-s és 3D-s grafika, animációk, gazdag dokumentumok stb.) integrál egyetlen egységesített objektummodellbe. Ebben a fejezetben megkezdjük az ismerkedést a WPF-fel és a kiterjeszthető alkalmazásjelölő nyelvvel (Extendable Application Markup Language - XAML). Megtanuljuk, hogyan kell XAML-mentes, csak XAML-t használó és a kettő kombinációjából felépülő WPF-programokat létrehozni. A fejezet végén készítünk egy egyedi XAML-nézegetőt, amelyet a további WPF-fel foglalkozó fejezetekben is használni fogunk.

29. fejezet: Programozás WPF-vezérlőelemekkel

Ebben a fejezetben megtudjuk, hogy hogyan kell dolgozni a WPF-vezérlőelem tartalommodelljével, és érintünk számos vezérlőelemekkel kapcsolatos témát is, például a függőségi tulajdonságokat és irányított eseményeket. Jó néhány WPF-vezérlőelem használatát bemutatjuk, s szintén ebben a fejezetben magyarázzuk el az elrendezéskezelő, -vezérlő utasítások és a WPF-adatkötés modelljének a használatát.

30. fejezet: WPF 2D grafikus renderelés, erőforrások és témák

Az utolsó fejezet a WPF tárgyalását három látszólag független téma vizsgálatával zárja. A WPF grafikus renderelő szolgáltatásának általában szüksége van arra, hogy egyedi erőforrásokat definiáljunk hozzá. Ezeknek az erőforrásoknak a használatával lehetőségünk van egyedi WPF-animációk készítésére, és grafikát, erőforrásokat és animációkat alkalmazva egyedi témák készítését fogjuk tudni megoldani WPF-alkalmazások számára. A különböző témák összegzéseképpen a fejezet utolsó részében azt illusztráljuk, hogyan kell futásidőben egyedi grafikus témákat alkalmazni.

7. rész: Webes alkalmazások fejlesztése ASP.NET segítségével

A 7. rész az ASP.NET programozási API használatával készített webes alkalmazásokat vizsgálja. Az ASP.NET szándékosan úgy lett kialakítva, hogy az asztali felhasználói felület létrehozását modellezze, úgy, hogy szabványos HTTP kérésekre/válaszokra helyez egy eseményvezérelt, objektumorientált keretrendszert.

31. fejezet: ASP.NET weboldalak készítése

Ez a fejezet vezet be bennünket azASP.NET használatával történő webes alkalmazásfejlesztésbe. A kiszolgálóoldali szkripteket felváltják az igazi objektumorientált nyelvek (mint a C#, VB.NET és hasonlók). Bemutatjuk egy ASP.NET weboldal készítését, valamint az alatta található programozási modellt és az ASP.NET egyéb kulcsfontosságú elveit, például azt, hogy hogyan kell webkiszolgálót választani, valamint a `Web.config` fájlok használatát.

32. fejezet: ASP.NET-vezérlőelemek, -témák és -mesteroldalak

Ez a fejezet azokkal a vezérlőelemekkel foglalkozik, amelyek a belső vezérlőfát töltik fel adattal. Megismerjük az alapvető ASP.NET webes vezérlőelemeket, többek között a ellenőrző vezérlőelemet, a belső oldali navigációs vezérlőelemeket és a különböző adatkötő műveleteket. Ugyancsak ennek a fejezetnek a feladata, hogy bemutassa a *mesteroldalak* és az ASP.NET témamotorjának a szerepét, amely a hagyományos stíluslapok szerveroldali megfelelője.

33. fejezet: ASP.NET állapotkezelési technikák

Ez a fejezet kiterjeszti az eddigi ASP.NET-ismereteinket, ugyanis megvizsgálja, milyen különböző módokon lehet állapotot kezelni a .NET alatt. Ahogy a klasszikus ASP, az ASP.NET is egyszerűen teszi lehetővé sütik létrehozását, ahogy alkalmazásszintű és munkafolyamat-szintű változók használatát is. Az ASP.NET azonban bevezet egy új állapotkezelő technikát: az alkalmazásgyorsítótárat. Ha már tudjuk, milyen módokon lehet állapotot kezelni az ASP.NET alatt, megismerkedünk a `System.HttpApplication` alaposztály szerepével (a `Global.asax` fájlon belül), valamint azzal, hogy hogyan kell dinamikusan megváltoztatni a webalkalmazásunk futásidejű viselkedését a `Web.config` fájl segítségével.

8. rész: Függelékek

A könyv utolsó része két fontos témát vizsgál meg, amelyek valójában nem illeszkedtek szervesen a könyv felépítésébe, ezért a függelékbe kerültek. A függelék kiteljesíti a C#-pal és .NET platformmal kapcsolatos ismereteinket, megvizsgáljuk ugyanis, hogyan kell régebbi kódokat integrálni a .NET-alkalmazásainkba, valamint hogyan kell a .NET-fejlesztést a Windows operációs rendszer családján kívül használni.

A függelék: A COM és a .NET együttműködése

Azok, akik már programoztak Windows operációs rendszer alatt a .NET platform használata előtt, nagy valószínűséggel ismerik a Component Object Modelt (COM). Noha a COM-nak és a .NET-nek semmi köze egymáshoz (azon túl, hogy mindkettő a Microsoft szerzeménye), a .NET platform teljes névteret biztosít ahhoz (`System.Runtime.InteropServices`), hogy .NET-szoftverek COM-komponenseket használjanak, és fordítva. Ez a függelék az együttműködési réteget mutatja be egészen részletesen; ez a téma azért nagyon fontos, hogy a korábban megírt programjainkat át tudjuk menteni az újonnan készített .NET-alkalmazásainkba.

B függelék: Platformfüggetlen .NET-fejlesztés a Monóval

Végül, de nem utolsó sorban a B függelék a .NET nyílt forráskódú implementációjával, a *Monóval* foglalkozik. A Mono segítségével lehetőségünk van gazdag tulajdonságokkal rendelkező .NET-alkalmazás létrehozására, telepítésére és futtatására különböző operációs rendszereken, mint például a Max OS X, Solaris, AIX és számos Linux-disztribúció. Mivel a Mono nagyjából kompatibilis a Microsoft .NET platformjával, így már tisztában vagyunk azzal, hogy mit tud nyújtani számunkra. Ezért a függelék a Mono installálását tárgyalja, valamint azt, hogy, milyen fejlesztői eszközök állnak a rendelkezésünkre, és hogyan működik a Mono futtatómotorja.

Öt szabadon letölthető fejezet – még több információ

Azoknak, akiknek a 33 fejezet és a két függelék nem lenne elég, szabadon letölthetnek további öt fejezetet. A könyv korábbi verziói három olyan fejezetet tartalmaztak, amelyeket a Windows Forms fejlesztésének szenteltünk (egyedi vezérlők vizsgálatával egyetemben), egy másik fejezetet a .NET remoting rétegének (`System.Runtime.Remoting` és társai), és a végül egy fejezet a hagyományos XML-webszolgáltatások készítésével foglalkozott az ASP.NET Web Service projekt sablonjainak a segítségével .

A könyv jelen kiadása nem tartalmazza ezt az öt fejezetet. Ennek az a legfőbb oka, hogy a .NET 3.0-ban a WCF és a WFP API-k rendre a .NET-remoting/XML-webszolgáltatások és a Windows Forms API-k örökösei. Ha mélyebben szeretnénk elmerülni a Windows Forms világában (azon túl, amit a 27. fejezet nyújtani tud), vagy meg akarjuk nézni, hogyan kell használni a (örökségül hagyott) .NET-remotingot és az XML-webszolgáltatás-API-t, egyszerűen csak meg kell nézni az Apress honlapján a megfelelő részt:

```
http://apress.com/book/view/1590598849.
```

Az itt található linkre kattintva digitális formában letölthetjük a könyv további fejezeteit, ha sikerült válaszolni a könyv szövegéből véletlenszerűen feltett kérdésekre. (A letölthető fejezetek csak angol nyelven érhetők el - a SZAK Kiadó megjegyzése).

A könyv forráskódjának igénylése

A könyvben szereplő összes forráskód-mintapélda (az öt szabadon letölthető további fejezetben levők is) szabadon és azonnal elérhető az Apress honlapjának Source Code/Download részlegéről. Egyszerűen gépeljük be a http://www.apresss.com címet, válasszuk ki a Source Code/Download linket, és keressük meg a megfelelő címet, és, töltsük le az önmagát kicsomagoló `*.zip` fájlt. A forráskódokat fejezetenként osztottuk fel.

Figyelem: a fejezetekben a Forráskód megjelölés, mint amilyen az alábbi is, arra utal, hogy a szóban forgó példákat be lehet tölteni a Visual Studio 2008-ba további vizsgálódások és módosítások céljából:

Forráskód Ez a forráskód-megjelölés egy kiválasztott könyvtárra hivatkozik.

Egyszerűen nyissuk meg a `*.sln` fájlt a megfelelő alkönyvtárból. Ha nem használjuk a Visual Studio 2008-at (lásd a 2. fejezetet további integrált fejlesztői környezetekért), akkor manuálisan töltsük be a felkínált forráskódfájlokat a választott fejlesztői eszközbe.

Megjegyzés A magyar nyelvű változatban a forráskódok megjegyzéseit lefordítottuk. A letölthető forráskódok fordítása nem állt módunkban, így azok teljesen angol nyelvűek. Ilyen módon inkább megfelelnek a letölthető kódok használati céljának – a SZAK Kiadó megjegyzése.

A lehetséges javítások

Elképzelhető, hogy a könyv végigolvasása során nyelvtani vagy forráskódbeli hibákat vél felfedezni a kedves Olvasó, bár remélem, hogy erre nem kerül sor. Ha mégis, elnézést kérek. Az aktuális hibalistát az Apress weboldalról lehet igényelni (amely szintén a könyv honlapján található meg), és ugyancsak itt található az elérhetőségem, amelyen értesíteni tudnak.

Elérhetőségem

Ha bármilyen kérdés merül fel a könyvhöz tartozó forráskódokkal kapcsolatban, vagy szükség van valamelyik példa tisztázására, vagy egyszerűen csak véleményt szeretne nyilvánítani a .NET platformról, nyugodtan küldjön levelet a következő e-mail címre (hogy biztosak legyünk benne, hogy a levele nem kerül a kéretlen levelek közé, kérem, tegye a tárgymezőbe a „C#FE" szöveget): atroelsen@Intertech.com.

Annak ellenére, hogy megpróbálok minden levélre lehetőség szerint válaszolni, sajnos, ahogy mindnyájan, időnként nagyon elfoglalt vagyok. Ha nem válaszolok egy-két héten belül, akkor ez nem jelenti azt, hogy nem akarok foglalkozni a problémával, csak épp nagyon nem érek rá (vagy ha szerencsém van, valahol épp nyaralok).

Nos, akkor, köszönöm, hogy megvásárolta a könyvemet (vagy legalábbis, hogy belenéz a könyvesboltban, miközben azon gondolkodik, hogy megvegye-e). Remélem, élvezni fogja a kedves Olvasó a tanulmányozását, és hasznosítani tudja az újonnan szerzett ismereteket.

Minden jót!
Andrew Troelsen

5. rész

Bevezetés a .NET alaposztály-könyvtáraiba

HUSZADIK FEJEZET

Fájlműveletek és elszigetelt tárolás

Teljes értékű asztali alkalmazás készítésekor fontos, hogy a program képes legyen eltárolni a felhasználói munkamenetek információit. Ez a fejezet a .NET keretrendszer szemszögéből mutat be számos, I/O-val kapcsolatos témakört. Elsőként megismerkedünk a `System.IO` névtérben definiált alapvető típusokkal, és megvizsgáljuk, hogy hogyan módosíthatjuk programozottan a számítógép mappa- és fájlstruktúráját, majd a különböző karakter-, bináris, sztring- és memóriaalapú adattárolók olvasásával és írásával foglalkozunk.

Miután megismertük a fájlok és könyvtárak kezelését az alapvető I/O típusok segítségével, bemutatjuk az *elszigetelt tároló* használatát (a `System.IO.IsolatedStorage` névtéren keresztül). Ennek segítségével a szigorúbb biztonsági környezetben futó alkalmazások a felhasználói és az alkalmazásadatokat biztonságosan, korlátozott fájlműveletekkel hozhatják létre. Ennek az API-nak a bemutatásához először a .NET platform egyik biztonsági megoldását, a CAS-t (Code Access Security, kóderedet-alapú biztonság) kell megvizsgálnunk. A CAS-t gyakran az elszigetelt tárolóval együtt használjuk.

A System.IO névtér

A .NET keretrendszerben a fájlalapú (és a memóriaalapú) be- és kiviteli (I/O) szolgáltatásokat biztosító alapvető osztálykönyvtárak helye a `System.IO` névtér. Mint minden névtér, a `System.IO` is számos osztályt, interfészt, felsorolt típust, struktúrát és metódusreferenciát határoz meg, amelyek túlnyomó többsége az `mscorlib.dll` fájlban található. A `System.IO` névtér további tagjait a `System.dll` szerelvény definiálja (a Visual Studio 2008 projektek az alapértelmezés szerint mindkét szerelvényre hivatkoznak, így az itt említett típusokat további teendők nélkül, azonnal használhatjuk).

A `System.IO` névtér több típusa a fizikai mappák és fájlok programozott kezelésére összpontosít. Vannak azonban további típusok, amelyek a sztring- és memóriamanipulációkat támogatják. A 20.1. táblázat bemutatja a `System.IO` alapvető (nem absztrakt) osztályait.

Nem absztrakt I/O osztálytípusok	Jelentés
`BinaryReader`, `BinaryWriter`	Primitív adattípusok (egész számok, logikai értékek, sztringek stb.) bináris értékként történő tárolását és visszaolvasását teszik lehetővé.
`BufferedStream`	Byte-folyamok ideiglenes tárolójaként szolgál, ezeket a későbbiek során más tárolókra menthetjük.
`Directory`, `DirectoryInfo`	Ezekkel a számítógép könyvtárstruktúráját módosíthatjuk. A `Directory` típus a funkcionalitást *statikus tagok* segítségével teszi elérhetővé. A `DirectoryInfo` típus hasonló működést eredményez az érvényes *objektumreferenciából*.
`DriveInfo`	A számítógép meghajtóiról szolgáltat részletes információt.
`File`, `FileInfo`	Ezekkel a számítógépen található fájlokat módosíthatjuk. A `File` típus a funkcionalitást *statikus tagok* segítségével teszi elérhetővé. A `FileInfo` típus hasonló működést eredményez az érvényes *objektumreferenciából*.
`FileStream`	Véletlen fájlhozzáférést (pl. keresési képességeket) tesz lehetővé, és az adatokat byte-folyamként jeleníti meg.
`FileSystemWatcher`	Egy megadott könyvtárban található külső fájlok megváltozásának figyelését teszi lehetővé.
`MemoryStream`	Véletlen hozzáférést biztosít a memóriában tárolt adatfolyamhoz (nem pedig egy fizikai fájlhoz).
`Path`	Olyan `System.String` típusú objektumokon hajt végre műveleteket, amelyek platformfüggetlen fájl-, illetve könyvtárelérési utakról tartalmaznak információkat.
`StreamWriter`, `StreamReader`	Ezekkel fájlokban tárolhatunk (illetve azokból visszaolvashatunk) szöveges információkat. A véletlen fájlhozzáférést nem támogatják.

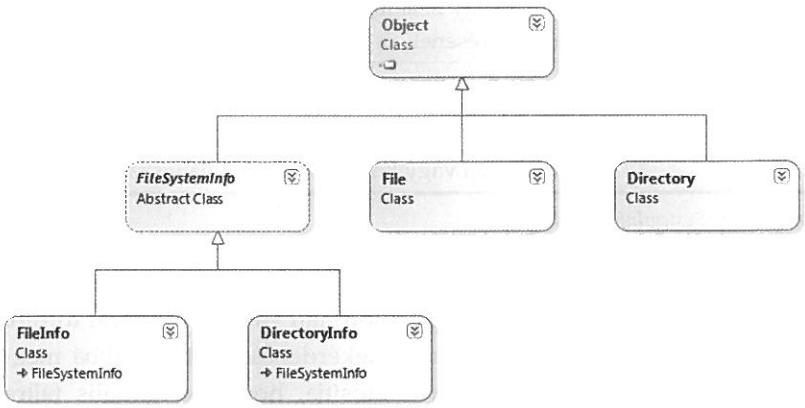
Nem absztrakt I/O osztálytípusok	Jelentés
`StringWriter`, `StringReader`	A `StreamReader` és `StreamWriter` típusokhoz hasonlóan ezek az osztályok is szöveges információkat kezelnek, háttértárolóik azonban nem fizikai fájlok, hanem sztringpufferek.

20.1. táblázat: *A System.IO névtér kulcsfontosságú tagjai*

Az imént felsorolt konkrét osztálytípusok mellett a `System.IO` több felsorolt típust és olyan absztrakt osztályt (`Stream`, `TextReader`, `TextWriter` stb.) is definiál, amelyek egy megosztott polimorf interfészt definiálnak az összes leszármazott számára. A későbbiekben számos ilyen típust bemutatunk.

A Directory(Info) és File(Info) típusok

A `System.IO` névtér négy típus segítségével teszi lehetővé a számítógép könyvtárstruktúrájának és különálló fájljainak a kezelését. Az első két típus, a `Directory` és a `File`, különböző statikus tagokon keresztül biztosítja az objektumok létrehozását, törlését, másolását és mozgatását. Az ezekhez szorosan kapcsolódó `FileInfo` és `DirectoryInfo` típusok hasonló működést tesznek elérhetővé példányszintű metódusokkal (így ezeket a `new` kulcsszóval le kell foglalni). A 20.1. ábrán látható, hogy a `Directory` és `File` típusok közvetlenül a `System.Object` osztályt bővítik ki, míg a `DirectoryInfo` és `FileInfo` osztályok az absztrakt `FileSystemInfo` típusból származnak.



20.1. ábra: *A File- és Directory-központú típusok*

Általánosságban elmondhatjuk, hogy a `FileInfo` és a `DirectoryInfo` osztályokat célszerűbb használni, ha egy fájlról vagy könyvtárról részletes információkra (pl. létrehozás ideje, írhatóság/olvashatóság stb.) van szükségünk, ugyanis ezek tagjai erősen típusos objektumokat adnak vissza. Ezzel szemben a `Directory` és a `File` osztályok tagjai erősen típusos objektumok helyett inkább egyszerű sztringértékekkel térnek vissza.

Az absztrakt FileSystemInfo ősosztály

A `DirectoryInfo` és a `FileInfo` típusok az absztrakt `FileSystemInfo` ősosztálytól öröklik számos képességüket. Legnagyobbrészt arra használjuk `FileSystemInfo` osztály tagjait, hogy lekérdezzük egy fájl vagy egy könyvtár általános tulajdonságait (pl. a létrehozás idejét, s különböző attribútumokat stb). A 20.2. táblázat felsorolja az érdekesebb alaptulajdonságokat.

Tulajdonság	Jelentés
`Attributes`	Visszaadja vagy beállítja az aktuális fájlnak azokat az attribútumait, amelyeket a `FileAttributes` felsorolás ír le.
`CreationTime`	Visszaadja vagy beállítja az aktuális fájl vagy könyvtár létrehozásának az idejét.
`Exists`	Segítségével megállapíthatjuk, hogy egy adott fájl vagy könyvtár létezik-e.
`Extension`	Visszaadja a fájl kiterjesztését.
`FullName`	Visszaadja a fájl vagy könyvtár teljes elérési útvonalát.
`LastAccessTime`	Visszaadja vagy beállítja az aktuális fájl vagy könyvtár legutolsó elérésének az időpontját.
`LastWriteTime`	Visszaadja vagy beállítja az aktuális fájl vagy könyvtár utolsó módosításának az idejét.
`Name`	Az aktuális fájl vagy könyvtár nevét adja vissza.

20.2. táblázat: *A FileSystemInfo tulajdonságai*

A `FileSystemInfo` osztály definiálja a `Delete()` metódust is. Ezt a származtatott típusok valósítják meg azért, hogy egy adott fájlt vagy könyvtárat törölni lehessen a merevlemezről. Az attribútumok lekérdezése előtt továbbá meghívhatjuk a `Refresh()` metódust, amely biztosítja, hogy az aktuális fájlra (vagy könyvtárra) vonatkozó statisztikák ne legyenek elavultak.

A DirectoryInfo típus használata

Az első, I/O műveletekkel kapcsolatos típus, amelyet közelebbről megvizsgálunk, a `DirectoryInfo` osztály. Tagjainak a segítségével létrehozhatunk, áthelyezhetünk, törölhetünk és kilistázhatunk könyvtárakat és alkönyvtárakat. Az ősosztályától (`FileSystemInfo`) örökölt funkcionalitáson túl a `DirectoryInfo` a 20.3. táblázatban feltüntetett kulcsfontosságú tagokkal rendelkezik.

Tag	Jelentés
`Create()`, `CreateSubdirectory()`	Egy könyvtár (vagy több alkönyvtár) létrehozása a megadott útvonalon.
`Delete()`	Egy adott könyvtárnak és teljes tartalmának a törlése.
`GetDirectories()`	Visszaad egy, az aktuális könyvtár összes alkönyvtárának nevét tartalmazó sztringtömböt.
`GetFiles()`	`FileInfo` típusok tömbjével tér vissza, amely az adott könyvtár fájljait tartalmazza.
`MoveTo()`	A könyvtárat és a tartalmát új elérési út alá mozgatja.
`Parent`	Az adott elérési út szülőkönyvtárát adja vissza.
`Root`	Megkapja az elérési útvonal gyökérrészét.

20.3. táblázat: *A DirectoryInfo típus kulcsfontosságú tagjai*

A `DirectoryInfo` típus használatához az elérési útvonalat kell megadnunk konstruktorparaméterként. Ha az aktuális könyvtárhoz (pl. a futó alkalmazás könyvtárához) szeretnénk hozzáférni, használjuk a „." jelölést. Az alábbiakban bemutatunk néhány példát:

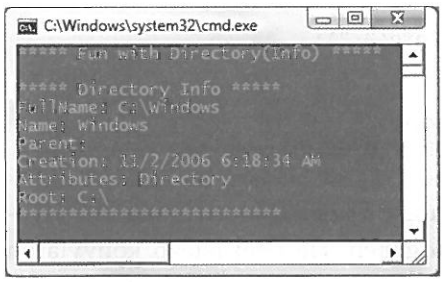
```
// Az aktuális könyvtárhoz kötés.
DirectoryInfo dir1 = new DirectoryInfo(".");

// A C:\Windows könyvtárhoz kötés,
// szó szerinti sztring használatával.
DirectoryInfo dir2 = new DirectoryInfo(@"C:\Windows");
```

A második példában azt feltételezzük, hogy a konstruktornak átadott útvonal (C:\Windows) már létezik a számítógépen. Ha nem létező könyvtáron próbálunk műveleteket végrehajtani, a rendszer egy `System.IO.DirectoryNotFoundException` kivételt dob. Így, ha még nem létező könyvtárat adunk meg, először meg kell hívnunk a `Create()` metódust:

```
// Kössük hozzá egy nem létező könyvtárhoz, majd hozzuk létre.
DirectoryInfo dir3 = new DirectoryInfo(@"C:\MyCode\Testing");
dir3.Create();
```

Miután létrehoztuk a `DirectoryInfo` objektumot, megvizsgálhatjuk a könyvtár tartalmát bármely, a `FileSystemInfo` típusból örökölt tulajdonsággal. Példaként hozzunk létre egy új, `DirectoryApp` nevű konzolalkalmazást. A `Program` osztályt egészítsük ki egy új statikus metódussal, amely létrehoz egy olyan új `DirectoryInfo` objektumot a C:\Windows könyvtárra leképezve (szükség esetén módosítsuk az útvonalat), amely számos érdekes statisztikát jelenít meg (a kimenet a 20.2. ábrán látható):

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Directory(Info) *****\n");
    ShowWindowsDirectoryInfo();
    Console.ReadLine();
  }

  static void ShowWindowsDirectoryInfo()
  {
    // A könyvtárinformációk kiírása.
    DirectoryInfo dir = new DirectoryInfo(@"C:\Windows");
    Console.WriteLine("***** Directory Info *****");
    Console.WriteLine("FullName: {0}", dir.FullName);
    Console.WriteLine("Name: {0}", dir.Name);
    Console.WriteLine("Parent: {0}", dir.Parent);
    Console.WriteLine("Creation: {0}", dir.CreationTime);
    Console.WriteLine("Attributes: {0}", dir.Attributes);
    Console.WriteLine("Root: {0}", dir.Root);
    Console.WriteLine("**************************\n");
  }
}
```



20.2. ábra: *Információk megjelenítése a Windows-könyvtárról*

Fájlok listázása a DirectoryInfo típus segítségével

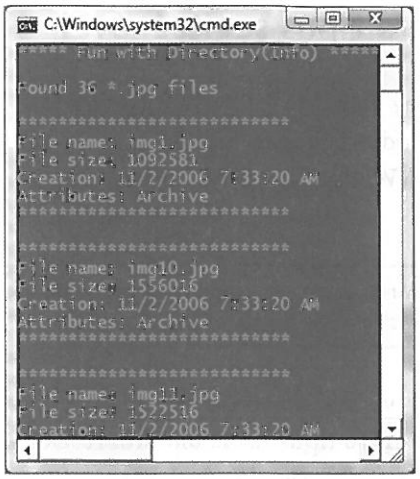
A fenti példát kiegészíthetjük azzal, hogy az alapvető információk lekérdezésén túl a `DirectoryInfo` típus néhány metódusát is használjuk. Először kérdezzünk le adatokat a C:\Windows\Web\Wallpaper könyvtárban található `*.jpg`-fájlokról a `GetFiles()` metódussal.

Megjegyzés Ha a számítógépen nem található C:\Windows\Web\Wallpaper könyvtár, alakítsuk át megfelelően a kódot, például úgy, hogy a C:\Windows könyvtár `*.bmp` fájljait olvassa be.

Ez a metódus olyan `FileInfo` tömbbel tér vissza, amelyek mindegyike egy adott fájlról tartalmaz információkat (a `FileInfo` típus használatáról részletesen később lesz szó). Vegyük a `Program` osztály következő, a `Main()` függvényből meghívott, statikus metódusát:

```
static void DisplayImageFiles()
{
  DirectoryInfo dir =
                new DirectoryInfo(@"C:\Windows\Web\Wallpaper");
  // Az összes *.jpg kiterjesztésű fájl lekérdezése.
  FileInfo[] imageFiles = dir.GetFiles("*.jpg");

  // Mennyit találtunk?
  Console.WriteLine("Found {0} *.jpg files\n", imageFiles.Length);

  // Minden fájlról kiírjuk az információkat.
  foreach (FileInfo f in imageFiles)
  {
    Console.WriteLine("***************************");
    Console.WriteLine("File name: {0}", f.Name);
    Console.WriteLine("File size: {0}", f.Length);
    Console.WriteLine("Creation: {0}", f.CreationTime);
    Console.WriteLine("Attributes: {0}", f.Attributes);
    Console.WriteLine("***************************\n");
  }
}
```

Ha futtatjuk az alkalmazást, akkor a 20.3. ábrához hasonló listát kell látnunk. (A képek nevei különbözhetnek.)



20.3. ábra: *Információk képfájlokról*

Alkönyvtárak létrehozása a DirectoryInfo segítségével

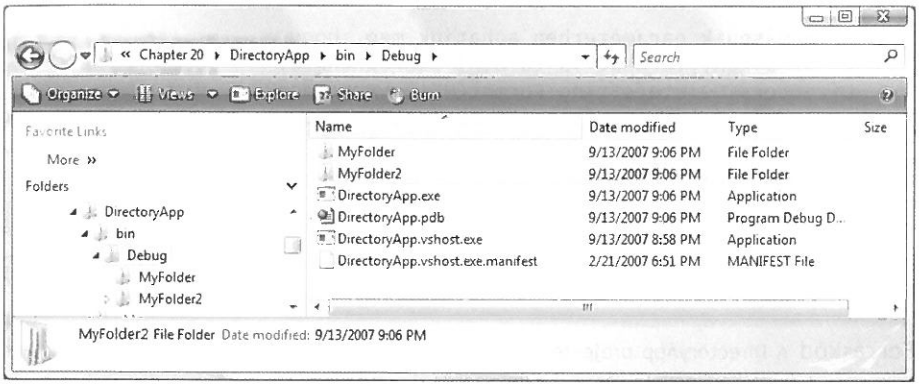
Egy könyvtár szerkezetét a `DirectoryInfo.CreateSubdirectory()` metódussal programozottan bővíthetjük. Ezzel a metódussal akár egy, akár több egymásba ágyazott alkönyvtárat is létrehozhatunk egyszeres függvényhívással. A következő metódus például kibővíti az alkalmazás telepítési útvonalának könyvtárszerkezetét néhány alkönyvtárral:

```
static void ModifyAppDirectory()
{
  DirectoryInfo dir = new DirectoryInfo(".");

  // A \MyFolder létrehozása az alkalmazás könyvtárában.
  dir.CreateSubdirectory("MyFolder");

  // A \MyFolder2\Data létrehozása az alkalmazás könyvtárában.
  dir.CreateSubdirectory(@"MyFolder2\Data");
}
```

Ha meghívjuk ezt a metódust a `Main()` függvényből, és Windows Intézővel megnézzük a Windows-könyvtárat, látható, hogy megjelentek az új alkönyvtárak (lásd a 20.4. ábrát).

A `CreateSubdirectory()` metódus visszatérési értékéről annyit kell tudni, hogy sikeres végrehajtás esetén a metódus visszaad egy `DirectoryInfo` objektumot, amely az újonnan létrehozott könyvtárat jelképezi. Nézzük meg az alábbi példát:

```
static void ModifyAppDirectory()
{
  DirectoryInfo dir = new DirectoryInfo(".");

  // A \MyFolder létrehozása a kezdőkönyvtárban.
  dir.CreateSubdirectory("MyFolder");

  // A visszaadott DirectoryInfo objektum tárolása.
  DirectoryInfo myDataFolder =
              dir.CreateSubdirectory(@"MyFolder2\Data");

  // A ..\MyFolder2\Data elérési útjának kiírása.
  Console.WriteLine("New Folder is: {0}", myDataFolder);
}
```



20.4. ábra: *Alkönyvtárak létrehozása*

A Directory típus használata

A következőkben ismerkedjünk meg a `Directory` típussal. A legtöbb esetben a `Directory` osztály statikus tagjainak működése megfelel annak a funkcionalitásnak, amelyet a `DirectoryInfo` példányszintű tagjai nyújtanak. Ne feledjük azonban, hogy a `Directory` tagjai általában sztringtípusokat adnak vissza, nem pedig erősen típusos `FileInfo`/`DirectoryInfo` típusokat.

A `Directory` típus működésének szemléltetésére ez a végső segédfüggvény megjeleníti a számítógépen található összes meghajtó nevét (a `Directory.GetLogicalDrives()` metódussal), és a `Directory.Delete()` statikus metódussal letörli a korábban létrehozott \MyFolder és \MyFolder2\Data alkönyvtárakat:

```
static void FunWithDirectoryType()
{
  // A számítógép meghajtóinak kilistázása.
  string[] drives = Directory.GetLogicalDrives();
  Console.WriteLine("Here are your drives:");
  foreach (string s in drives)
    Console.WriteLine("--> {0} ", s);

  // Letöröljük, amit létrehoztunk.
  Console.WriteLine("Press Enter to delete directories");
  Console.ReadLine();
  try
  {
    Directory.Delete(string.Format(@"{0}\MyFolder",
                     Environment.CurrentDirectory));

    // A második paraméterben adhatjuk meg, hogy
    // az alkönyvtárakat is törölni szeretnénk-e.
    Directory.Delete(string.Format(@"{0}\MyFolder2",
                     Environment.CurrentDirectory), true);
  }
  catch (IOException e)
  {
    Console.WriteLine(e.Message);
  }
}
```

Forráskód A DirectoryApp projektet a forráskódkönyvtárban a 20. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

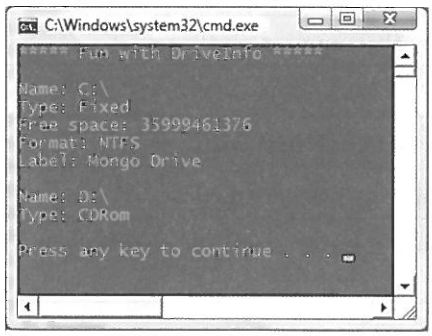
A DriveInfo osztálytípus használata

A `System.IO` névtérben található a `DriveInfo` nevű osztály. A `Directory.GetLogicalDrives()` metódushoz hasonlóan a statikus `DriveInfo.GetDrives()` metódus is a számítógép meghajtóinak a nevét adja vissza, de a `Directory.GetLogicalDrives()` metódussal ellentétben a `DriveInfo` számos egyéb részletet is megad a meghajtókról (pl. a meghajtó típusát, a szabad területet, a kötetcímkét stb.). Nézzük a következő, `DriveInfoApp` nevű új konzolalkalmazásban definiált `Program` osztályt:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with DriveInfo *****\n");
```

```
        // Információk lekérése minden meghajtóról.
        DriveInfo[] myDrives = DriveInfo.GetDrives();

        // A meghajtók statisztikáinak kiírása.
        foreach(DriveInfo d in myDrives)
        {
          Console.WriteLine("Name: {0}", d.Name);
         Console.WriteLine("Type: {0}", d.DriveType);

          // Ellenőrizzük, hogy a meghajtó csatlakoztatva van-e.
          if (d.IsReady)
          {
            Console.WriteLine("Free space: {0}", d.TotalFreeSpace);
            Console.WriteLine("Format: {0}", d.DriveFormat);
            Console.WriteLine("Label: {0}", d.VolumeLabel);
            Console.WriteLine();
          }
        }
        Console.ReadLine();
      }
    }
```

A 20.5. ábra egy lehetséges kimenetet mutat.



20.5. ábra: *Meghajtók adatainak lekérdezése a DriveInfo használatával*

A `Directory`, a `DirectoryInfo` és a `DriveInfo` osztályok alapvető működésének áttekintése után a következőkben megvizsgáljuk, hogyan hozhatunk létre, nyithatunk meg, zárhatunk be és törölhetünk adott könyvtárban lévő fájlokat.

Forráskód A DriveInfoApp projektet a forráskódkönyvtárban a 20. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A FileInfo osztály használata

Ahogy a DirectoryApp-példában is láttuk, a FileInfo osztály segítségével a számítógépen található fájlokról kérdezhetünk le adatokat (létrehozás ideje, méret, attribútumok stb.), továbbá az osztály segítségével létrehozhatunk, másolhatunk, áthelyezhetünk és törölhetünk fájlokat. A FileSystemInfo típusból örökölt funkcionalitáson túl a FileInfo osztály néhány egyedi alaptagját a 20.4. táblázatban foglaltuk össze.

Tag	Jelentés
AppendText()	Egy StreamWriter típust hoz létre (lásd később), amelynek segítségével szöveget fűzhetünk a fájlhoz.
CopyTo()	A meglévő fájlt átmásolja egy új fájlba.
Create()	Új fájlt hoz létre, és egy FileStream típussal (lásd később) tér vissza, amellyel az újonnan létrehozott fájlt kezelhetjük.
CreateText()	Egy StreamWriter típust hoz létre, amely új szövegfájlt ír.
Delete()	Törli a FileInfo példány által reprezentált állományt.
Directory	A szülőkönyvtár egy példányát adja vissza.
DirectoryName	Visszaadja a szülőkönyvtár teljes elérési útját.
Length	Visszaadja az aktuális fájl vagy könyvtár méretét.
MoveTo()	A fájlt áthelyezi, és lehetőséget ad arra, hogy új nevet adjunk neki.
Name	Visszaadja a fájl nevét.
Open()	Megnyitja a fájlt különböző olvasási/írási és megosztási jogosultságokkal.
OpenRead()	Csak olvasható FileStream típust hoz létre.
OpenText()	Egy StreamReader típust hoz létre (lásd később), amellyel egy létező szövegfájlból olvashatunk.
OpenWrite()	Csak írható FileStream típust hoz létre.

20.4. táblázat: *FileInfo alapvető tagjai*

A `FileInfo` osztály metódusainak többsége tehát egy specifikus I/O-központú objektumot (`FileStream`, `StreamWriter` stb.) ad vissza. Ezek segítségével a fájlból többféleképpen olvashatunk, illetve írhatunk adatokat. A példák tanulmányozása előtt azonban vizsgáljuk meg azokat a különböző módszereket, amelyekkel a `FileInfo` osztállyal megszerezhetjük a fájlazonosítóját.

A FileInfo.Create() metódus

A fájlazonosító létrehozásának első módja a `FileInfo.Create()` metódus használata:

```
static void Main(string[] args)
{
  // Létrehozunk egy új fájlt a C meghajtón.
  FileInfo f = new FileInfo(@"C:\Test.dat");
  FileStream fs = f.Create();

  // Használjuk a FileStream objektumot...

  // Zárjuk le a fájlfolyamot.
  fs.Close();
}
```

A `FileInfo.Create()` metódus `FileStream` típussal tér vissza, amely a fájlon mind szinkron, mind aszinkron írási/olvasási műveleteket tesz lehetővé (a részleteket lásd később). A `FileInfo.Create()` által visszaadott `FileStream` objektum minden felhasználó számára teljes olvasási/írási hozzáférést biztosít.

Miután befejeztük a `FileStream` objektummal a munkát, zárjuk le a fájlazonosítót, hiszen így a rendszer felszabadítja az adatfolyam nem felügyelt erőforrásait. Minthogy a `FileStream` megvalósítja az `IDisposable` interfészt, a C# `using` blokkjának a segítségével a fordítóra bízhatjuk a „takarító"-logika létrehozását (a részleteket lásd az első kötet 8. fejezetében):

```
static void Main(string[] args)
{
  // Egy 'using' hatókör definiálása
  // ideális a fájl I/O típusokhoz.
  FileInfo f = new FileInfo(@"C:\Test.dat");
  using (FileStream fs = f.Create())
  {
    // Használjuk a FileStream objektumot...
  }
}
```

A FileInfo.Open() metódus

A `FileInfo.Open()` metódust a `FileInfo.Create()` által nyújtott lehetőségeknél precízebben használhatjuk a fájlok megnyitására és létrehozására, ugyanis az `Open()` jellemzően több paramétert vesz fel a fájl kezelésének a leírására. Az `Open()` meghívása után egy `FileStream` objektumot kapunk vissza. Nézzük meg a következő logikát:

```
static void Main(string[] args)
{
  // Létrehozunk egy új fájlt a FileInfo.Open() metódussal.
  FileInfo f2 = new FileInfo(@"C:\Test2.dat");
  using(FileStream fs2 = f2.Open(FileMode.OpenOrCreate,
        FileAccess.ReadWrite, FileShare.None))
  {
    // Használjuk a FileStream objektumot...
  }
}
```

A túlterhelt `Open()` metódus ezen verziójának három paraméterre van szüksége. Az első paraméter az I/O-kérés típusát határozza meg (pl. új fájl létrehozása, létező fájl megnyitása, hozzáfűzés fájlhoz stb.), amelyet a `FileMode` felsorolt típussal határozhatunk meg (lásd 20.5. táblázat):

```
public enum FileMode
{
  CreateNew,
  Create,
  Open,
  OpenOrCreate,
  Truncate,
  Append
}
```

Tag	Jelentés
`CreateNew`	Az operációs rendszert új fájl létrehozására utasítja. Ha a fájl már létezik, `IOException` kivételt dob.
`Create`	Az operációs rendszert új fájl létrehozására utasítja. Ha a fájl már létezik, akkor felülírja azt.
`Open`	Megnyit egy már létező fájlt. Ha a fájl nem létezik, `FileNotFoundException` kivételt dob.
`OpenOrCreate`	Megnyitja a fájlt, ha az már létezik; ellenkező esetben új fájlt hoz létre.

A C# 2008 és a .NET 3.5

Második kötet (20-33. fejezet)

Andrew Troelsen

SZAK KIADÓ

2009

A C# 2008 és a .NET 3.5 - Második kötet (20-33. fejezet és két függelék)
A negyedik kiadás magyarul, két kötetben
Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition
Andrew Troelsen

Fordította a SZAK Kiadó fordítócsoportja: Domoszlai László, Endrédi Gabriella, Erős Szilvia, Iváncsy Renáta, Szalay Márton, Tiber Melinda, Varga Ágnes, Varga Péter
Lektor: Dávid Zoltán, MVP és Gincsai Gábor, MVP
Magyar nyelvi gondozás: Laczkó Krisztina és Trepák Mónika

ISBN 978-963-9863-10-1

A könyv fordítása a Kilgray Kft. MemoQ (http://www.memoqtm.com) programjával készült, a szöveg helyességét és az elválasztásokat a MorphoLogic Helyesek nevű programjával ellenőriztük.

SZAK Kiadó Kft. ■ Az 1795-ben alapított Magyar Könyvkiadók és Könyvterjesztők Egyesülésének a tagja ■ 2060 Bicske, Diófa u. 3. ■ Tel.: 36-22-350-209 ■ Fax: 36-22-565-311 ■ www.szak.hu ■ e-mail: info@szak.hu ■ Kiadóvezető: Kis Ádám, e-mail: adam.kis@szak.hu ■ Főszerkesztő: Kis Balázs MCSE, MCT, e-mail: balazs.kis@szak.hu

Ezt a kiadást Mikkónak, a csodamacskának, a 412-es szám alatt eltöltött éveknek és csodálatos feleségemnek, Amandának ajánlom, aki türelmesen várta, hogy elkészüljek újabb könyvem megírásával.

Tartalomjegyzék

5. rész: Bevezetés a .NET alaposztálykönyvtáraiba

20. Fájlműveletek és elszigetelt tárolás 3

30. WPF 2D grafikus renderelés, erőforrások és témák 619

8. rész: Függelékek

Köszönetnyilvánítás

Jóllehet egyedül az én nevem jelenik meg a könyv borítóján, de a munka soha nem készülhetett volna el jó néhány tehetséges ember segítsége nélkül. A következőkben azokhoz szólok, akik lehetővé tették, hogy ezt a könyvet megírjam.

Mindenekelőtt köszönet illeti az Apress összes munkatársát, akikkel már több éve van szerencsém együtt dolgozni. Rendkívüli emberek, akik képesek voltak az én eredeti Word-dokumentumaimat tökéletes prózává varázsolni. Hálás köszönet érte. Már most nagyon várom, hogy mikor dolgozhatunk ismét együtt egy következő könyvön (de azért előtte szabadságolom magam egy időre).

Külön köszönöm technikai szerkesztőm, Gavin munkáját, aki annyi bölcs tanáccsal segített, hogy úgy érzem, ez a kiadás sokkal színvonalasabb lett, mint az előzőek. Mint mindig, az esetleges gépelési hibákért vagy technikai tévedésekért egyedül én vagyok a felelős.

Végül, de nem utolsósorban, köszönöm családomnak, barátaimnak és munkatársaimnak, hogy elviselték mogorva viselkedésemet, amely a kézirat elkészítésének utolsó szakaszaiban jó néhányszor jellemzett engem.

Bevezetés

Ez a könyv nagyjából 2001 nyara óta létezik, ha nem is ebben a formában: a C# és a .NET platform első kiadása ugyanis akkor jelent meg, a .NET 1.0 Beta 2 verziójával karöltve. Nagy örömmel és hálával tölt el, hogy munkám kedvező fogadtatásra lelt a sajtóban, és ami még fontosabb, az olvasók körében is. Az évek során jelölték a Jolt Award döntőjére (vesztettem... kellemetlen) és a 2003-as Referenceware Excellence Awardon a programozási könyvek kategóriájában (nyertem?... de jó!)

Azóta azon dolgoztam, hogy a könyvet frissítsem és aktualizáljam a .NET platform újabb verzióinak megfelelően. Eközben limitált darabszámmal megjelentettem egy speciális kiadványt, amely bemutatta a .NET 3.0 verzióját (Windows Presentation Foundation, Windows Coomunication Foundation, és a Windows Workflow Foundation), valamint áttekintést nyújtott olyan különböző technológiákról, amelyek még megjelenésre vártak, és amelyeket most LINQ néven ismerünk.

Jelenleg a könyv negyedik kiadását tartja a kezében az Olvasó. Ez a kiadás egyrészt tömören megismétli a korábbi ismereteket, azaz a .NET 3.5-ös verziójában található legfontosabb változások magyarázatát, másrészt számos teljesen új fejezetet is tartalmaz, de ezen túlmenően több eddigi rész is jelentősen kibővült.

Ahogy a korábbiakban megszokhattuk, ez a kiadás is egyszerű és érthető nyelven ismerteti a C# nyelvet és a .NET-alaposztálykönyvtárakat. Soha nem értettem azokat a szakírókat, akik úgy adják elő mondandójukat, hogy az jobban hasonlít egy GRE-szószedetről írt tanulmányra, mint egy szakkönyvre. Munkám továbbra is arra fekteti a hangsúlyt, hogy a benne található információk segítségével jól használható szoftverrendszereket lehessen készíteni, így nem töltök túl sok időt az ezoterikus részletek tanulmányozásával, hiszen ezekkel csak nagyon kevés olvasóm fog valaha is foglalkozni.

Mi egy csapat vagyunk, az Olvasó és én

A szakírók az emberek igényes csoportjának írnak (nekem már csak tudnom kell - egy vagyok közülük). Tisztában vagyunk azzal, hogy egy szoftver elkészítése - bármelyik platformmal történjen is (.NET, J2EE, COM stb.) - rendkívül aprólékos munka, ugyanakkor nagyon specifikus a részleg, a cég, az ügyfél vagy a tárgykör szerint. Hiszen lehet, hogy épp az elektronikus kiadói iparágban dolgozunk, vagy éppenséggel rendszert fejlesztünk az államnak, esetleg a helyi önkormányzatnak, a NASA-nak vagy egy katonai részlegnek. Én a magam részéről gyermekeket oktató szoftvert fejlesztettem, számos n-rétegű rendszert, valamint projekteket a gyógyászati és a pénzügyi ágazatokban. Majdnem nulla az esélye, hogy annak a kódnak, amelyet egyikünk ír a munkájához, köze lesz ahhoz a kódhoz, amelyet a másikunk ír a saját munkájához (kivéve, ha véletlen épp együtt dolgoztunk korábban!).

Ezért szándékosan kerülöm az olyan mintapéldákat, amelyeket konkrét iparághoz vagy programozási feladathoz köthetnénk: a C#, az objektumorientált programozás és a .NET 3.5 alaposztály könyvtárait ipari alkalmazásoktól független példákon keresztül mutatom be. Ahelyett, hogy minden példám adatok tömkelegét szolgáltatná, bérlistát számolna, vagy ehhez hasonlókkal szolgálna, az én példáimhoz mindenkinek van valami köze: autókról szólnak (ráadásként néhány geometriai struktúrával és alkalmazottakkal). És ez az a pont, ahol összetalálkozunk.

Az *én feladatom*, hogy legjobb tudásom szerint elmagyarázzam a C# programozási nyelvet és a .NET platform alapvető jellegzetességeit. Ugyanígy mindent megteszek azért, hogy ellássam a kedves Olvasót különféle eszközökkel és stratégiákkal, amelyek lehetővé teszik, hogy a könyv alapján folytatni lehessen a téma tanulmányozását.

Az *olvasó feladata* pedig, hogy a megszerezett tudást képes legyen alkalmazni saját programozási feladatában. Persze egyértelmű, hogy a projektek nem állatnevekkel ellátott automobilok körül forognak, de hát pont erről szólna az alkalmazott ismeret. Biztos lehet benne az Olvasó, hogy ha egyszer megértette a könyvben bemutatott elveket, ismereteinek birtokában olyan .NET-alkalmazásokat írhat, amelyek használhatók saját programozási környezetükben.

A könyv áttekintése

A *Pro C# 2008 and the .NET 3.5 Platform (A C# és a Microsoft .NET 3.5.)* negyedik kiadása logikailag nyolc különálló részből áll, s ezek mindegyike további fejezeteket tartalmaz. Jó néhány témakör, például az alapvető C#-szerkezetek, az objektumorientált programozás és a platformfüggetlen .NET fejlesztése kibővült a korábbiakhoz képest, saját fejezetet kapott. Az új kiadás a .NET 3.0–3.5 programozás tulajdonságait (LINQ , WCF , WPF , WF stb.) összefoglaló fejezeteket is tartalmaz.

Megjegyzés A magyar változat kiadásának előkészítése során a SZAK Kiadó úgy döntött, hogy az eredeti könyv tartalmát két kötetben jelenteti meg. Ezt a viszonylag nagy terjedelem indokolja. Az alábbiakban mind a két kötet tartalmát ismertetjük, így biztosítva, hogy az is képet kapjon a mű egészéről, akinek csak az egyik kötet kerül a kezébe. (Ezt a Bevezetést változatlan formában mind a két kötetben szerepeltetjük.)

Első kötet

1. rész: Bevezetés a C#-ba és a .NET platformba

Az első rész bemutatja a .NET platform alapvető természetét és számos olyan fejlesztőeszközt (köztük több nyílt forráskodút), amelyeket .NET-alkalmazások fejlesztéséhez használhatunk. Mindeközben néhány alap C# programozási nyelvi részlettel és a .NET-típusrendszerrel is megismerkedünk.

1. fejezet: A .NET filozófiája

Az első fejezet voltaképpen a könyv további részeinek a gerincét képezi. A hagyományos Windows-fejlesztések világának vizsgálatával kezdjük, és feltárjuk a korábbi hátrányokat is. A fejezet elsődleges célja azonban az, hogy megismerjünk jó néhány .NET-központú építőkövet, például az egységes nyelvi futtatót (Common Language Runtime – CLR), a közös típusrendszert (Common Type System – CTS), az egységes nyelvi specifikációt (Common Language Specification – CLS) és az alaposztálykönyvtárakat. Egyszerű betekintést nyerhetünk a C# programozási nyelvbe és a .NET-szerelvényformátumába, valamint áttekintést kapunk a .NET platform platformfüggetlen természetéről (a B függelék részletesen tárgyalja ezt a témakört).

2. fejezet: C#-alkalmazások készítése

A fejezet célja, hogy különféle eszközök és technikák ismertetésével bevezessen minket a C#-forráskódfájl fordításának folyamatába. Először a parancssoros fordító (`csc.exe`) és a C#-eredményfájlok használatát ismerhetjük meg. A fejezet további részében pedig számos forráskódszerkesztőt és integrált fejlesztési környezetet (integrated development environments - IDEs) tanulmányozhatunk, többek között a következőket: TextPad, Notepad++, SharpDevelop, Visual C# 2008 Express és a Visual Studio 2008. Ezenkívül számos olyan további programozási eszközt ismertetünk, amelyet bármely .NET-fejlesztőnek azonnal elő kell tudnia húzni a farzsebéből.

2. rész: A C# alapvető építőelemei

A 2. részben tárgyalt témakörök kiemelten fontosak, hiszen az itt megjelenő tudnivalókat folyamatosan használjuk függetlenül attól, hogy milyen jellegű .NET-szoftvert szándékozunk fejleszteni (webes alkalmazás, asztali grafikus alkalmazás, forráskódkönyvtárak, Windows-szolgáltatások stb.). Itt derül ki, hogy hogyan működnek a C# nyelv alapvető építőkövei, beleértve az objektumorientált programozás (OOP) részleteit is. Ez a rész ismerteti továbbá, hogy hogyan kell kezelni a futásidejű kivételeket, és bevezet minket a .NET szemétgyűjtő szolgáltatásának részleteibe is.

3. fejezet: A C# alapvető építőelemei, I. rész

Ebben a fejezetben kezdjük el részletesen tanulmányozni a C# programozási nyelvet. Megismerkedünk a `Main()` metódus szerepével és számos részlettel a .NET platform belső adattípusaival kapcsolatban, beleértve a szövegesadat-kezelést a `System.String` és a `System.Text.StringBuilder` névterek alkalmazásával. Megvizsgáljuk továbbá az iterációs és a döntési szerkezeteket, a szűkítő és a bővítő műveleteket, valamint az `unchecked` kulcsszó használatát.

4. fejezet: A C# alapvető építőelemei, II. rész

A 4. fejezet befejezi az alapvető C#-szemléletek meghatározását, a típustúlterhelt metódusok létrehozásától az `out`, `ref` és `params` kulcsszavakon keresztüli paraméterek definiálásáig. Vizsgáljuk, hogyan kell adathalmazt létrehozni és kezelni, hogyan kell olyan adattípust készíteni, amelynek `null` is lehet az értéke (a `?` és a `??` operátorokkal), és megmutatjuk az *értéktípus* és a *referenciatípus* közötti különbséget is.

5. fejezet: Egységbe zárt osztálytípusok definiálása

Ebben a fejezetben az objektumorientált programozással (OOP) ismerkedhetünk meg a C# programozási nyelv használatán keresztül. Először meghatározzuk az OOP pilléreit (beágyazás, származtatás és polimorfizmus), majd a fejezet további részében azt nézzük meg, hogyan kell konstruktorok, tulajdonságok, statikus tagváltozók, konstansok és csak olvasható fájlok használatával robusztusosztály-típusokat létrehozni. Végezetül ismertetjük a részleges típus definícióját és a C# XML-kódjának dokumentációs szintaktikáját.

6. fejezet: A származtatás és a polimorfizmus

Ebben a részben az OOP maradék pilléreit (származtatás és polimorfizmus) vizsgáljuk meg; ezek a segítségével összetartozó osztálytípusok családját hozhatjuk létre. Megnézzük a virtuális metódusok és absztrakt metódusok (és az absztrakt alaposztályok) szerepét, valamint a *polimorf interfész* természetét. Végül, de nem utolsósorban, ez a fejezet ismerteti a .NET platform legfontosabb alaposztályának, a `System.Object` osztálynak a szerepét is.

7. fejezet: Strukturált hibakezelés

Ez a fejezet elsősorban azt mutatja be, mit kell tenni a strukturális kivételkezelés segítségével a futási időben keletkezett anomáliákkal. Nemcsak az ilyen jellegű problémák megoldására szolgáló C#-kulcsszavakat ismerjük itt meg (`try`, `catch`, `throw` és `finally`), hanem az alkalmazásszintű és a rendszerszintű kivételek közötti különbséget is. Megismerhetünk továbbá többféle, a Visual Studio 2008-ban található eszközt, amelyek segítségével hibakeresést végezhetünk a figyelmen kívül hagyott kivételek között.

8. fejezet: Az objektumok életciklusa

Ennek a résznek az utolsó fejezete azt tárgyalja, hogy hogyan kezeli a CLR a memóriát a .NET szemétgyűjtőjének a segítségével. Megérthetjük az alkalmazásgyökerek, az objektumnemzedékek és a `System.GCType` szerepét. Az alapok megismerése után, a fejezet további része az *eldobható objektumokkal* (az `IDisposable` interfészen keresztül) és a véglegesítő folyamattal (a `System.Object.Finalize()` metóduson keresztül) foglalkozik.

3. rész: Haladó programozási szerkezetek a C#-ban

A könyv 3. része ismerteti a C# nyelv haladóbb jellegű (de ugyancsak nagyon fontos) elveit. Az interfészek és metódusreferenciák vizsgálatával befejezzük a .NET típusrendszerével való ismerkedést. A továbbiakban betekintést nyerhetünk a generikus típusok és a számos új C# 2008 nyelvi elem szerepébe, valamint a LINQ-ba is.

9. fejezet: Interfészek használata

Ennek a fejezetnek az anyagát az interfészalapú programozást is magába foglaló objektumalapú fejlesztés alkotja. Megismerhetjük belőle, hogy hogyan kell olyan típusokat definiálni, amelyek többszörös viselkedést támogatnak, hogyan lehet ezeket a viselkedéseket futásidőben feltérképezni, és hogyan lehet szelektíven elrejteni bizonyos viselkedéseket az *explicit interfészmegvalósítás* segítségével. Továbbá a számos előre definiált .NET-interfésztípus megismeréséhez el kell sajátítanunk azoknak az egyedi interfészeknek a használatát, amelyek ad hoc jellegű eseményarchitektúra készítéséhez szükségesek.

10. fejezet: Gyűjtemények és generikus típusok

Ez a fejezet a `System.Collections` névtérgyűjtemény típusainak tanulmányozásával kezdődik; ezek már a .NET platform kezdeti kiadásának is részei voltak. Noha a .NET 2.0 megjelenése óta a C# programozási nyelv lehetővé teszi *generikus* típusok használatát, ahogy a későbbiekben kiderül, a generikus programozás lényegesen megnöveli az alkalmazás teljesítményét és a típusbiztonságot. Nemcsak azt vizsgáljuk meg, hogy milyen különböző generikus típusok léteznek a `System.Collections.Generic` névtérben, de azt is, hogyan készíthetjük el saját generikus metódusainkat és típusainkat (megszorításokkal vagy anélkül).

11. fejezet Metódusreferenciák, események és lambdák

A 11. fejezetnek az a célja, hogy érthetővé tegye a *metódusreferencia*-típusokat. Leegyszerűsítve: a .NET-metódusreferencia olyan objektum, amely az alkalmazás más metódusaira mutat. Ez a minta olyan rendszerek készítését teszi lehetővé, amelyek több objektum összekapcsolódását engedik meg kétirányú együttműködés segítségével. A .NET-metódusreferenciák vizsgálata után megismerkedünk a .NET `event` kulcsszavával, amelyet arra használunk, hogy egyszerűsítsük a nyers metódusreferencia programozásának kezelését. Végezetül megvizsgáljuk a C# 2008 lambda-operátorát (=>), és feltárjuk a kapcsolatot a metódusreferenciák, a névtelen metódusok és a lambda-kifejezések között.

12. fejezet: Indexerek, operátorok és mutatók

Ez a fejezet a C# programozási ismereteinket mélyíti el számos haladó programozási technika segítségével. Megismerhetjük, hogyan kell operátorokat túlterhelni, és hogyan kell a típusaink számára egyedi konverziós rutinokat írni (akár implicit, akár explicit formában). Megtanulhatjuk azt is, hogy hogyan kell létrehozni és használni a *típusindexelőket*, és hogyan kell kezelni a C-stílusú mutatókat a „nem biztonságos" kódkörnyezetben.

13. fejezet: C# 2008 nyelvi újdonságai

A .NET 3.5 kiadásában a C# nyelv számos új programozási eszközzel bővült, amelyek közül jó néhány arra szolgál, hogy lehetővé tegye a LINQ API használatát (erről a 14. fejezetben többet is megtudhatunk). Megismerjük a lokális változók implicit típussal történő ellátásának a szerepét, a részleges metódus, az automatikus tulajdonságok, a bővítő metódusok, a névtelen típusok és az objektuminicializáló szintaxis használatát.

14. fejezet: Bevezetés a nyelvbe ágyazott lekérdezésekbe (LINQ)

Ez a fejezet ismerteti a LINQ-et (Language Integrated Query - nyelvi integrált lekérdezés), amelyet nyugodtan nevezhetünk a .NET 3.5 legérdekesebb részének. Ahogy a fejezet későbbi részéből kiderül, a LINQ segítségével lehetőségünk nyílik erősen típusos *lekérdezéseket* írni, amelyeket számos LINQ-célpontra alkalmazhatunk, hogy a szó legtágabb értelmében „adatot" kezelhessünk. Megtanuljuk a LINQ objektumokra történő alkalmazását, ennek segítségével a LINQ-kifejezéseket adattárolókra (tömbök, kollekciók, egyedi típusok) fogjuk tudni alkalmazni. Ez a tudás nélkülözhetetlen lesz akkor, amikor azt vizsgáljuk meg, hogyan kell LINQ-kifejezéseket relációs adatbázisokon (a LINQ az ADO-n keresztül) és XML-dokumentumokon (a LINQ az XML-en keresztül) használni.

4. rész: Programozás .NET-szerelvényekkel

A negyedik rész a .NET-szerelvényformátum részleteit taglalja. Nemcsak azt tanulhatjuk meg, hogyan kell .NET-kódkönyvtárakat telepíteni és konfigurálni, de azt is, hogy mi a .NET bináris kép belső összeállítása. Ebben a részben lesz szó a .NET-attribútumok szerepéről és a többszálú alkalmazások létrehozásának módjáról is. A későbbi fejezetek néhány haladó témát érintenek, például az objektumkörnyezetet, a köztes nyelvi kódot és a dinamikus szerelvényeket.

15. fejezet: A .NET-szerelvények

Első megközelítésben a *szerelvény* nem más, mint egy felügyelt `*.dll` vagy `*.exe` bináris fájl leírására szolgáló kifejezés. Ám a .NET-szerelvények története sokkal több ennél. Ebben a fejezetben megtudjuk, mi az egyfájlos és a többfájlos szerelvények közötti különbség, és, hogy ezeket hogyan kell készíteni és telepíteni. Megvizsgáljuk, hogyan kell privát és megosztott szerelvényeket konfigurálni XML-alapú `*.config` fájlokkal és kibocsátó házirenddel rendelkező szerelvényekkel. Mindeközben megtudhatjuk, hogy milyen a GAC (Global Assembly Cache – globális szerelvénytár) belső struktúrája, és mi a .NET keretrendszer konfigurációs segédprogramjainak a szerepe.

16. fejezet: Típusreflexió, késői kötés és attribútumalapú programozás

A 16. fejezetben folytatjuk az ismerkedést a .NET-szerelvényekkel. A `System.Reflection` névtér használatával kapcsolatban azt vizsgáljuk, a futási időben hogyan valósul meg a típusmeghatározás folyamata. Az ilyen jellegű típusok használatával lehetőségünk van olyan alkalmazások létrehozására, amelyek futás közben képesek a szerelvények metaadatait olvasni. Kiderül, hogyan lehet a *késői kötés* segítségével futási időben dinamikusan betölteni és létrehozni típusokat. A fejezet utolsó témaköre – mind a szabványos, mind az egyedi – .NET-attribútumok szerepének ismertetése. Az egyes témakörök hasznosságának szemléltetésére a fejezet konklúziójaként bemutatunk egy kiterjeszthető Windows Forms-alkalmazást.

17. fejezet: Folyamatok, alkalmazástartományok és objektumkörnyezetek

A szerelvényekről szóló tudnivalók után ez a fejezet a betöltött .NET végrehajtható fájlok felépítését ismerteti. Az elsődleges cél a folyamatok, az alkalmazástartományok és a kontextusbeli határok közötti kapcsolat illusztrálása. Ezek alapozzák meg a következő fejezet témáját: a többszálú alkalmazások készítésének módját.

18. fejezet: Többszálú alkalmazások készítése

Ez a fejezet azt mutatja be, hogy hogyan kell többszálú alkalmazásokat készíteni, és számos olyan technikát ismertet, amelyeket a szálbiztos kód készítéséhez alkalmazhatunk. A fejezet elején átismételjük a .NET metódusreferencia-típusait, hogy megértsük, valójában hogyan támogatják a metódusreferenciák

az aszinkronfüggvény-hívásokat. Majd megvizsgáljuk a `System.Thread` névtérben található típusokat. Ezek között számos olyan van (`Thread`, `ThreadStart`), amelynek segítségével könnyen lehet újabb végrehajtási szálakat készíteni. Végül megismerkedünk a `BackgroundWorker` típussal, amely nagyban megkönnyíti a felhasználói felületről történő szálkészítést.

19. fejezet: A köztes nyelv (CIL) és a dinamikus szerelvények

A 4. rész utolsó fejezetének szerepe kettős. Először a korábbinál részletesebben megvizsgáljuk CIL szintaktikáját és szemantikáját, majd a `System.Reflection.Emit` névtér szerepét mutatjuk be. Ezeknek a típusoknak a használatával lehetőségünk nyílik olyan szoftver készítésére, amely képes .NET-szerelvényeket futásidőben generálni a memóriába. Az olyan szerelvényeket, amelyeket a memóriában definiálunk és futtatunk, *dinamikus szerelvényeknek* nevezzük.

Második kötet

5. rész: Bevezetés a .NET alaposztálykönyvtáraiba

Ha elérkeztünk ehhez a részhez, akkor már biztos tudásunk van a C# nyelvről és a .NET- szerelvényformátumok részleteiről. Az ötödik rész továbbfejleszti ezeket az ismereteket azáltal, hogy számos általánosan használt alaposztálybeli szolgáltatást mutat be, többek között a fájl I/O-t és az adatbáziselérést az ADO.NET segítségével. Ez a rész tárgyalja továbbá az elosztott alkalmazások készítésének témakörét a Windows Communication Foundation (WCF) segítségével, valamint azokat a munkafolyamat-engedélyezett alkalmazások készítését, amelyek a Windows Workflow Foundation (WF) API-t használják.

20. fejezet: Fájlműveletek és elkülönített tárolás

A `System.IO` névtér segítségével lehetőségünk van a számítógép fájl- és könyvtárszerkezetének elérésére. Ebben a fejezetben megismerjük, hogyan kell programozottan könyvtárrendszert létrehozni (és törölni), valamint, adatokat különböző folyamokba (fájlalapú, sztringalapú, memóriaalapú stb.) be-, illetve kimozgatni. A fejezet utolsó része megvizsgálja az *elkülönített tárolást*, amelynek segítségével lehetőségünk van biztonságos helyen külön tárolni az adatokat függetlenül attól, hogy a célgép biztonsági beállításaitól. Hogy megértsük a `System.IO.Isolated` API bizonyos elveit, áttekintést kell kapnunk a kódhozzáférés-szabályozásról (Code Access Security - CAS).

21. fejezet: Bevezetés az objektumsorosítás világába

Ez a fejezet a .NET-platform objektumsorosító szolgáltatásába nyújt betekintést. Egyszerűen szólva a *sorosítás* segítségével lehetőségünk nyílik arra, hogy egy objektum (vagy a kapcsolódó objektumok egy halmazának) állapotát egy folyamban tároljuk a későbbi használat céljából. A *deszerializálás* (a sorosítás ellentétes művelete) az a folyamat, amelynek segítségével kivehetjük az objektumot a folyamból, és a memóriában az alkalmazás számára használható formába hozhatjuk. Az alapok elsajátítása után megtanulhatjuk, hogyan kell egyedivé tenni a sorosítási folyamatot az `ISerializableinterface` és a .NET-attribútumok halmazának segítségével.

22. fejezet: ADO.NET 1. rész: Az élő kapcsolat

Az adatbázisokról szóló két fejezet közül az elsőben az ADO.NET programozási API-ról kaptunk ismereteket. Ez a rész bevezeti a .NET adatszolgáltató szerepét, valamint azt, hogy hogyan kell a relációs adatbázissal kommunikálni az ADO.NET *élő kapcsolatán* keresztül. Ez a kapcsolati objektumokat, a parancsobjektumokat és az adatolvasó objektumokat jelenti Ezenkívül ez a fejezet mutatja be az egyedi adatbázis létrehozásának módját, és ismerteti azokat az adatelérési könyvtárakat, amelyeket majd a könyv további részében használunk.

23. fejezet: ADO.NET 2. rész: A bontott kapcsolat

Ez a fejezet az adatbázis-kezelés különböző módokon történő manipulálási lehetőségeit tárgyalja tovább, jelen esetben az ADO.NET *bontott kapcsolaton* keresztül. Megismerkedhetünk a `DataSet` típus és az adatfeldolgozó objektumok szerepével, valamint számos Visual Studio 2008 eszközzel, amelyek nagyban megkönnyítik az adatvezérelt alkalmazások létrehozását. Mindeközben rávilágítunk arra, hogyan kell a `DataTable`-objektumokat felhasználói felületi elemekhez kapcsolni, olyanokhoz, mint például a Windows Forms API `GridView` típusa.

24. fejezet: A LINQ API programozása

A 24. fejezet a LINQ programozási modellt mutatja be, pontosabban a LINQ „az objektum irányába" részét. Megvizsgáljuk, hogyan kell LINQ-lekérdezéseket alkalmazni relációs adatbázisokon, `DataSet`-objektumokon és XML-dokumentumokon. Mindeközben megtanuljuk az adatkontextus-objektum, valamint az `sqlmetal.exe` segédprogram szerepét, és megvizsgáljuk a Visual Studio 2008 különböző LINQ-támogató funkcióit.

25. fejezet: A WCF

A .NET 3.0 bevezetett egy teljesen új API-t, a WCF-et, amely lehetőséget ad arra, hogy szimmetrikus módon, függetlenül az alatta található rétegek felépítésétől elosztott alkalmazást fejlesszünk. Ez a fejezet feltárja a WCF-szolgáltatások, -gazdagépek és -kliensek készítésének lehetőségeit. A WCF-szolgáltatások rendkívül flexibilisek, a gazdagépek és a kliensek XML-alapú konfigurációs fájlokat használhatnak, hogy deklaratív módon adjanak meg címeket, kötéseket és szerződéseket.

26. fejezet: A Windows Workflow Foundation — Bevezetés

A .NET 3.0 a WCF mellett bevezette a WF API-t is, amely lehetőséget ad arra, hogy *munkafolyamatokat* definiáljunk, futtassunk és monitorozzunk. Ennek segítségével komplex üzleti folyamatok modellezhetők. Megtanuljuk a WF általános célját, ahogy azt is, hogy mi a szerepe az aktivitásoknak, a munkafolyamat tervezőnek, a munkafolyamat futtatási motornak, és a munkafolyamat-engedélyezett forráskód könyvtáraknak.

6. rész: Felhasználói felületek

Általánosan hibás feltételezés a .NET platformmal kapcsolatban az, hogy ez a keretrendszer kizárólag a webalapú felhasználói felületekkel foglalkozik (ez feltehetően a „.NET" kifejezés miatt van, hiszen felidézi az „Internet" szót, azaz a „webprogramokat"). A .NET kiemelkedő módon támogatja a webalkalmazások készítését, a könyv ezen része azonban a hagyományos felhasználói felületekre koncentrál két grafikus felület keretrendszerén, a Windows Forms és a WPF használatán keresztül.

27. fejezet: Windows Forms-programozás

Az eredeti asztali grafikus felhasználói felület, amely a .NET platformmal együtt kapható, a *Windows Forms*. Ez a fejezet ismerteti a felhasználói felület keretrendszerének szerepét, és bemutatja, hogyan kell főablakokat, párbeszédablakokat és menürendszereket készíteni. Feltárja továbbá az űrlapszármaztatás szerepét, és azt, hogy hogyan kell kétdimenziós adatot rajzolni a `System.Drawing` névtér segítségével. A témakör illusztrálására a fejezet végén készítünk egy (korlátozott funkcionalitással bíró) rajzolóalkalmazást.

28. fejezet: A WPF és az XAML

A .NET 3.0 teljesen új grafikus felhasználói felületet vezetett be, amelyet *WPF*-nek nevezett el. Röviden, a WPF kimagaslóan interaktív és médiában gazdag front endek készítését teszi lehetővé asztali alkalmazásokhoz (és indirekt módon webalkalmazásokhoz). A Windows Formsszal ellentétben, ez a túltelített felhasználóifelület-keretrendszer számos kulcsfontosságú szolgáltatást (2D-s és 3D-s grafika, animációk, gazdag dokumentumok stb.) integrál egyetlen egységesített objektummodellbe. Ebben a fejezetben megkezdjük az ismerkedést a WPF-fel és a kiterjeszthető alkalmazásjelölő nyelvvel (Extendable Application Markup Language - XAML). Megtanuljuk, hogyan kell XAML-mentes, csak XAML-t használó és a kettő kombinációjából felépülő WPF-programokat létrehozni. A fejezet végén készítünk egy egyedi XAML-nézegetőt, amelyet a további WPF-fel foglalkozó fejezetekben is használni fogunk.

29. fejezet: Programozás WPF-vezérlőelemekkel

Ebben a fejezetben megtudjuk, hogy hogyan kell dolgozni a WPF-vezérlőelem tartalommodelljével, és érintünk számos vezérlőelemekkel kapcsolatos témát is, például a függőségi tulajdonságokat és irányított eseményeket. Jó néhány WPF-vezérlőelem használatát bemutatjuk, s szintén ebben a fejezetben magyarázzuk el az elrendezéskezelő, -vezérlő utasítások és a WPF-adatkötés modelljének a használatát.

30. fejezet: WPF 2D grafikus renderelés, erőforrások és témák

Az utolsó fejezet a WPF tárgyalását három látszólag független téma vizsgálatával zárja. A WPF grafikus renderelő szolgáltatásának általában szüksége van arra, hogy egyedi erőforrásokat definiáljunk hozzá. Ezeknek az erőforrásoknak a használatával lehetőségünk van egyedi WPF-animációk készítésére, és grafikát, erőforrásokat és animációkat alkalmazva egyedi témák készítését fogjuk tudni megoldani WPF-alkalmazások számára. A különböző témák összegzéseképpen a fejezet utolsó részében azt illusztráljuk, hogyan kell futásidőben egyedi grafikus témákat alkalmazni.

7. rész: Webes alkalmazások fejlesztése ASP.NET segítségével

A 7. rész az ASP.NET programozási API használatával készített webes alkalmazásokat vizsgálja. Az ASP.NET szándékosan úgy lett kialakítva, hogy az asztali felhasználói felület létrehozását modellezze, úgy, hogy szabványos HTTP kérésekre/válaszokra helyez egy eseményvezérelt, objektumorientált keretrendszert.

31. fejezet: ASP.NET weboldalak készítése

Ez a fejezet vezet be bennünket azASP.NET használatával történő webes alkalmazásfejlesztésbe. A kiszolgálóoldali szkripteket felváltják az igazi objektumorientált nyelvek (mint a C#, VB.NET és hasonlók). Bemutatjuk egy ASP.NET weboldal készítését, valamint az alatta található programozási modellt és az ASP.NET egyéb kulcsfontosságú elveit, például azt, hogy hogyan kell webkiszolgálót választani, valamint a `Web.config` fájlok használatát.

32. fejezet: ASP.NET-vezérlőelemek, -témák és -mesteroldalak

Ez a fejezet azokkal a vezérlőelemekkel foglalkozik, amelyek a belső vezérlőfát töltik fel adattal. Megismerjük az alapvető ASP.NET webes vezérlőelemeket, többek között a ellenőrző vezérlőelemet, a belső oldali navigációs vezérlőelemeket és a különböző adatkötő műveleteket. Ugyancsak ennek a fejezetnek a feladata, hogy bemutassa a *mesteroldalak* és az ASP.NET témamotorjának a szerepét, amely a hagyományos stíluslapok szerveroldali megfelelője.

33. fejezet: ASP.NET állapotkezelési technikák

Ez a fejezet kiterjeszti az eddigi ASP.NET-ismereteinket, ugyanis megvizsgálja, milyen különböző módokon lehet állapotot kezelni a .NET alatt. Ahogy a klasszikus ASP, az ASP.NET is egyszerűen teszi lehetővé sütik létrehozását, ahogy alkalmazásszintű és munkafolyamat-szintű változók használatát is. Az ASP.NET azonban bevezet egy új állapotkezelő technikát: az alkalmazásgyorsítótárat. Ha már tudjuk, milyen módokon lehet állapotot kezelni az ASP.NET alatt, megismerkedünk a `System.HttpApplication` alaposztály szerepével (a `Global.asax` fájlon belül), valamint azzal, hogy hogyan kell dinamikusan megváltoztatni a webalkalmazásunk futásidejű viselkedését a `Web.config` fájl segítségével.

8. rész: Függelékek

A könyv utolsó része két fontos témát vizsgál meg, amelyek valójában nem illeszkedtek szervesen a könyv felépítésébe, ezért a függelékbe kerültek. A függelék kiteljesíti a C#-pal és .NET platformmal kapcsolatos ismereteinket, megvizsgáljuk ugyanis, hogyan kell régebbi kódokat integrálni a .NET-alkalmazásainkba, valamint hogyan kell a .NET-fejlesztést a Windows operációs rendszer családján kívül használni.

A függelék: A COM és a .NET együttműködése

Azok, akik már programoztak Windows operációs rendszer alatt a .NET platform használata előtt, nagy valószínűséggel ismerik a Component Object Modelt (COM). Noha a COM-nak és a .NET-nek semmi köze egymáshoz (azon túl, hogy mindkettő a Microsoft szerzeménye), a .NET platform teljes névteret biztosít ahhoz (`System.Runtime.InteropServices`), hogy .NET-szoftverek COM-komponenseket használjanak, és fordítva. Ez a függelék az együttműködési réteget mutatja be egészen részletesen; ez a téma azért nagyon fontos, hogy a korábban megírt programjainkat át tudjuk menteni az újonnan készített .NET-alkalmazásainkba.

B függelék: Platformfüggetlen .NET-fejlesztés a Monóval

Végül, de nem utolsó sorban a B függelék a .NET nyílt forráskódú implementációjával, a *Monóval* foglalkozik. A Mono segítségével lehetőségünk van gazdag tulajdonságokkal rendelkező .NET-alkalmazás létrehozására, telepítésére és futtatására különböző operációs rendszereken, mint például a Max OS X, Solaris, AIX és számos Linux-disztribúció. Mivel a Mono nagyjából kompatibilis a Microsoft .NET platformjával, így már tisztában vagyunk azzal, hogy mit tud nyújtani számunkra. Ezért a függelék a Mono installálását tárgyalja, valamint azt, hogy, milyen fejlesztői eszközök állnak a rendelkezésünkre, és hogyan működik a Mono futtatómotorja.

Öt szabadon letölthető fejezet – még több információ

Azoknak, akiknek a 33 fejezet és a két függelék nem lenne elég, szabadon letölthetnek további öt fejezetet. A könyv korábbi verziói három olyan fejezetet tartalmaztak, amelyeket a Windows Forms fejlesztésének szenteltünk (egyedi vezérlők vizsgálatával egyetemben), egy másik fejezetet a .NET remoting rétegének (`System.Runtime.Remoting` és társai), és a végül egy fejezet a hagyományos XML-webszolgáltatások készítésével foglalkozott az ASP.NET Web Service projekt sablonjainak a segítségével .

A könyv jelen kiadása nem tartalmazza ezt az öt fejezetet. Ennek az a legfőbb oka, hogy a .NET 3.0-ban a WCF és a WFP API-k rendre a .NET-remoting/XML-webszolgáltatások és a Windows Forms API-k örökösei. Ha mélyebben szeretnénk elmerülni a Windows Forms világában (azon túl, amit a 27. fejezet nyújtani tud), vagy meg akarjuk nézni, hogyan kell használni a (örökségül hagyott) .NET-remotingot és az XML-webszolgáltatás-API-t, egyszerűen csak meg kell nézni az Apress honlapján a megfelelő részt:

`http://apress.com/book/view/1590598849.`

Az itt található linkre kattintva digitális formában letölthetjük a könyv további fejezeteit, ha sikerült válaszolni a könyv szövegéből véletlenszerűen feltett kérdésekre. (A letölthető fejezetek csak angol nyelven érhetők el - a SZAK Kiadó megjegyzése).

A könyv forráskódjának igénylése

A könyvben szereplő összes forráskód-mintapélda (az öt szabadon letölthető további fejezetben levők is) szabadon és azonnal elérhető az Apress honlapjának Source Code/Download részlegéről. Egyszerűen gépeljük be a http://www.apresss.com címet, válasszuk ki a Source Code/Download linket, és keressük meg a megfelelő címet, és, töltsük le az önmagát kicsomagoló `*.zip` fájlt. A forráskódokat fejezetenként osztottuk fel.

Figyelem: a fejezetekben a Forráskód megjelölés, mint amilyen az alábbi is, arra utal, hogy a szóban forgó példákat be lehet tölteni a Visual Studio 2008-ba további vizsgálódások és módosítások céljából:

Forráskód Ez a forráskód-megjelölés egy kiválasztott könyvtárra hivatkozik.

Egyszerűen nyissuk meg a `*.sln` fájlt a megfelelő alkönyvtárból. Ha nem használjuk a Visual Studio 2008-at (lásd a 2. fejezetet további integrált fejlesztői környezetekért), akkor manuálisan töltsük be a felkínált forráskódfájlokat a választott fejlesztői eszközbe.

Megjegyzés A magyar nyelvű változatban a forráskódok megjegyzéseit lefordítottuk. A letölthető forráskódok fordítása nem állt módunkban, így azok teljesen angol nyelvűek. Ilyen módon inkább megfelelnek a letölthető kódok használati céljának – a SZAK Kiadó megjegyzése.

A lehetséges javítások

Elképzelhető, hogy a könyv végigolvasása során nyelvtani vagy forráskódbeli hibákat vél felfedezni a kedves Olvasó, bár remélem, hogy erre nem kerül sor. Ha mégis, elnézést kérek. Az aktuális hibalistát az Apress weboldalról lehet igényelni (amely szintén a könyv honlapján található meg), és ugyancsak itt található az elérhetőségem, amelyen értesíteni tudnak.

Elérhetőségem

Ha bármilyen kérdés merül fel a könyvhöz tartozó forráskódokkal kapcsolatban, vagy szükség van valamelyik példa tisztázására, vagy egyszerűen csak véleményt szeretne nyilvánítani a .NET platformról, nyugodtan küldjön levelet a következő e-mail címre (hogy biztosak legyünk benne, hogy a levele nem kerül a kéretlen levelek közé, kérem, tegye a tárgymezőbe a „C#FE" szöveget): atroelsen@Intertech.com.

Annak ellenére, hogy megpróbálok minden levélre lehetőség szerint válaszolni, sajnos, ahogy mindnyájan, időnként nagyon elfoglalt vagyok. Ha nem válaszolok egy-két héten belül, akkor ez nem jelenti azt, hogy nem akarok foglalkozni a problémával, csak épp nagyon nem érek rá (vagy ha szerencsém van, valahol épp nyaralok).

Nos, akkor, köszönöm, hogy megvásárolta a könyvemet (vagy legalábbis, hogy belenéz a könyvesboltban, miközben azon gondolkodik, hogy megvegye-e). Remélem, élvezni fogja a kedves Olvasó a tanulmányozását, és hasznosítani tudja az újonnan szerzett ismereteket.

Minden jót!
Andrew Troelsen

5. rész

Bevezetés a .NET alaposztály-könyvtáraiba

HUSZADIK FEJEZET

Fájlműveletek és elszigetelt tárolás

Teljes értékű asztali alkalmazás készítésekor fontos, hogy a program képes legyen eltárolni a felhasználói munkamenetek információit. Ez a fejezet a .NET keretrendszer szemszögéből mutat be számos, I/O-val kapcsolatos témakört. Elsőként megismerkedünk a `System.IO` névtérben definiált alapvető típusokkal, és megvizsgáljuk, hogy hogyan módosíthatjuk programozottan a számítógép mappa- és fájlstruktúráját, majd a különböző karakter-, bináris, sztring- és memóriaalapú adattárolók olvasásával és írásával foglalkozunk.

Miután megismertük a fájlok és könyvtárak kezelését az alapvető I/O típusok segítségével, bemutatjuk az *elszigetelt tároló* használatát (a `System.IO.IsolatedStorage` névtéren keresztül). Ennek segítségével a szigorúbb biztonsági környezetben futó alkalmazások a felhasználói és az alkalmazásadatokat biztonságosan, korlátozott fájlműveletekkel hozhatják létre. Ennek az API-nak a bemutatásához először a .NET platform egyik biztonsági megoldását, a CAS-t (Code Access Security, kóderedet-alapú biztonság) kell megvizsgálnunk. A CAS-t gyakran az elszigetelt tárolóval együtt használjuk.

A System.IO névtér

A .NET keretrendszerben a fájlalapú (és a memóriaalapú) be- és kiviteli (I/O) szolgáltatásokat biztosító alapvető osztálykönyvtárak helye a `System.IO` névtér. Mint minden névtér, a `System.IO` is számos osztályt, interfészt, felsorolt típust, struktúrát és metódusreferenciát határoz meg, amelyek túlnyomó többsége az `mscorlib.dll` fájlban található. A `System.IO` névtér további tagjait a `System.dll` szerelvény definiálja (a Visual Studio 2008 projektek az alapértelmezés szerint mindkét szerelvényre hivatkoznak, így az itt említett típusokat további teendők nélkül, azonnal használhatjuk).

A `System.IO` névtér több típusa a fizikai mappák és fájlok programozott kezelésére összpontosít. Vannak azonban további típusok, amelyek a sztring- és memóriamanipulációkat támogatják. A 20.1. táblázat bemutatja a `System.IO` alapvető (nem absztrakt) osztályait.

Nem absztrakt I/O osztálytípusok	Jelentés
`BinaryReader`, `BinaryWriter`	Primitív adattípusok (egész számok, logikai értékek, sztringek stb.) bináris értékként történő tárolását és visszaolvasását teszik lehetővé.
`BufferedStream`	Byte-folyamok ideiglenes tárolójaként szolgál, ezeket a későbbiek során más tárolókra menthetjük.
`Directory`, `DirectoryInfo`	Ezekkel a számítógép könyvtárstruktúráját módosíthatjuk. A `Directory` típus a funkcionalitást *statikus tagok* segítségével teszi elérhetővé. A `DirectoryInfo` típus hasonló működést eredményez az érvényes *objektumreferenciából*.
`DriveInfo`	A számítógép meghajtóiról szolgáltat részletes információt.
`File`, `FileInfo`	Ezekkel a számítógépen található fájlokat módosíthatjuk. A `File` típus a funkcionalitást *statikus tagok* segítségével teszi elérhetővé. A `FileInfo` típus hasonló működést eredményez az érvényes *objektumreferenciából*.
`FileStream`	Véletlen fájlhozzáférést (pl. keresési képességeket) tesz lehetővé, és az adatokat byte-folyamként jeleníti meg.
`FileSystemWatcher`	Egy megadott könyvtárban található külső fájlok megváltozásának figyelését teszi lehetővé.
`MemoryStream`	Véletlen hozzáférést biztosít a memóriában tárolt adatfolyamhoz (nem pedig egy fizikai fájlhoz).
`Path`	Olyan `System.String` típusú objektumokon hajt végre műveleteket, amelyek platformfüggetlen fájl-, illetve könyvtárelérési utakról tartalmaznak információkat.
`StreamWriter`, `StreamReader`	Ezekkel fájlokban tárolhatunk (illetve azokból visszaolvashatunk) szöveges információkat. A véletlen fájlhozzáférést nem támogatják.

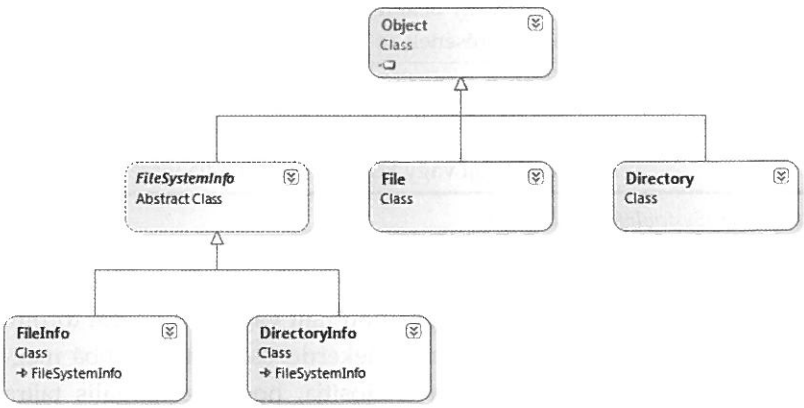
Nem absztrakt I/O osztálytípusok	Jelentés
`StringWriter`, `StringReader`	A `StreamReader` és `StreamWriter` típusokhoz hasonlóan ezek az osztályok is szöveges információkat kezelnek, háttértárolóik azonban nem fizikai fájlok, hanem sztringpufferek.

20.1. táblázat: *A System.IO névtér kulcsfontosságú tagjai*

Az imént felsorolt konkrét osztálytípusok mellett a `System.IO` több felsorolt típust és olyan absztrakt osztályt (`Stream`, `TextReader`, `TextWriter` stb.) is definiál, amelyek egy megosztott polimorf interfészt definiálnak az összes leszármazott számára. A későbbiekben számos ilyen típust bemutatunk.

A Directory(Info) és File(Info) típusok

A `System.IO` névtér négy típus segítségével teszi lehetővé a számítógép könyvtárstruktúrájának és különálló fájljainak a kezelését. Az első két típus, a `Directory` és a `File`, különböző statikus tagokon keresztül biztosítja az objektumok létrehozását, törlését, másolását és mozgatását. Az ezekhez szorosan kapcsolódó `FileInfo` és `DirectoryInfo` típusok hasonló működést tesznek elérhetővé példányszintű metódusokkal (így ezeket a `new` kulcsszóval le kell foglalni). A 20.1. ábrán látható, hogy a `Directory` és `File` típusok közvetlenül a `System.Object` osztályt bővítik ki, míg a `DirectoryInfo` és `FileInfo` osztályok az absztrakt `FileSystemInfo` típusból származnak.



20.1. ábra: *A File- és Directory-központú típusok*

Általánosságban elmondhatjuk, hogy a `FileInfo` és a `DirectoryInfo` osztályokat célszerűbb használni, ha egy fájlról vagy könyvtárról részletes információkra (pl. létrehozás ideje, írhatóság/olvashatóság stb.) van szükségünk, ugyanis ezek tagjai erősen típusos objektumokat adnak vissza. Ezzel szemben a `Directory` és a `File` osztályok tagjai erősen típusos objektumok helyett inkább egyszerű sztringértékekkel térnek vissza.

Az absztrakt FileSystemInfo ősosztály

A `DirectoryInfo` és a `FileInfo` típusok az absztrakt `FileSystemInfo` ősosztálytól öröklik számos képességüket. Legnagyobbrészt arra használjuk `FileSystemInfo` osztály tagjait, hogy lekérdezzük egy fájl vagy egy könyvtár általános tulajdonságait (pl. a létrehozás idejét, s különböző attribútumokat stb). A 20.2. táblázat felsorolja az érdekesebb alaptulajdonságokat.

Tulajdonság	Jelentés
`Attributes`	Visszaadja vagy beállítja az aktuális fájlnak azokat az attribútumait, amelyeket a `FileAttributes` felsorolás ír le.
`CreationTime`	Visszaadja vagy beállítja az aktuális fájl vagy könyvtár létrehozásának az idejét.
`Exists`	Segítségével megállapíthatjuk, hogy egy adott fájl vagy könyvtár létezik-e.
`Extension`	Visszaadja a fájl kiterjesztését.
`FullName`	Visszaadja a fájl vagy könyvtár teljes elérési útvonalát.
`LastAccessTime`	Visszaadja vagy beállítja az aktuális fájl vagy könyvtár legutolsó elérésének az időpontját.
`LastWriteTime`	Visszaadja vagy beállítja az aktuális fájl vagy könyvtár utolsó módosításának az idejét.
`Name`	Az aktuális fájl vagy könyvtár nevét adja vissza.

20.2. táblázat: *A FileSystemInfo tulajdonságai*

A `FileSystemInfo` osztály definiálja a `Delete()` metódust is. Ezt a származtatott típusok valósítják meg azért, hogy egy adott fájlt vagy könyvtárat törölni lehessen a merevlemezről. Az attribútumok lekérdezése előtt továbbá meghívhatjuk a `Refresh()` metódust, amely biztosítja, hogy az aktuális fájlra (vagy könyvtárra) vonatkozó statisztikák ne legyenek elavultak.

A DirectoryInfo típus használata

Az első, I/O műveletekkel kapcsolatos típus, amelyet közelebbről megvizsgálunk, a `DirectoryInfo` osztály. Tagjainak a segítségével létrehozhatunk, áthelyezhetünk, törölhetünk és kilistázhatunk könyvtárakat és alkönyvtárakat. Az ősosztályától (`FileSystemInfo`) örökölt funkcionalitáson túl a `DirectoryInfo` a 20.3. táblázatban feltüntetett kulcsfontosságú tagokkal rendelkezik.

Tag	Jelentés
`Create()`, `CreateSubdirectory()`	Egy könyvtár (vagy több alkönyvtár) létrehozása a megadott útvonalon.
`Delete()`	Egy adott könyvtárnak és teljes tartalmának a törlése.
`GetDirectories()`	Visszaad egy, az aktuális könyvtár összes alkönyvtárának nevét tartalmazó sztringtömböt.
`GetFiles()`	`FileInfo` típusok tömbjével tér vissza, amely az adott könyvtár fájljait tartalmazza.
`MoveTo()`	A könyvtárat és a tartalmát új elérési út alá mozgatja.
`Parent`	Az adott elérési út szülőkönyvtárát adja vissza.
`Root`	Megkapja az elérési útvonal gyökérrészét.

20.3. táblázat: *A DirectoryInfo típus kulcsfontosságú tagjai*

A `DirectoryInfo` típus használatához az elérési útvonalat kell megadnunk konstruktorparaméterként. Ha az aktuális könyvtárhoz (pl. a futó alkalmazás könyvtárához) szeretnénk hozzáférni, használjuk a „." jelölést. Az alábbiakban bemutatunk néhány példát:

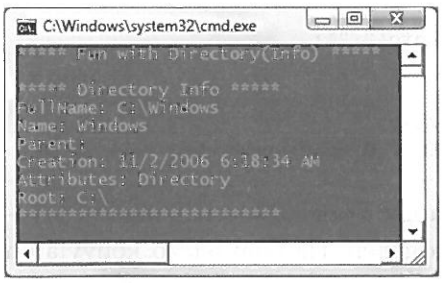
```
// Az aktuális könyvtárhoz kötés.
DirectoryInfo dir1 = new DirectoryInfo(".");

// A C:\Windows könyvtárhoz kötés,
// szó szerinti sztring használatával.
DirectoryInfo dir2 = new DirectoryInfo(@"C:\Windows");
```

A második példában azt feltételezzük, hogy a konstruktornak átadott útvonal (C:\Windows) már létezik a számítógépen. Ha nem létező könyvtáron próbálunk műveleteket végrehajtani, a rendszer egy `System.IO.DirectoryNotFoundException` kivételt dob. Így, ha még nem létező könyvtárat adunk meg, először meg kell hívnunk a `Create()` metódust:

```
// Kössük hozzá egy nem létező könyvtárhoz, majd hozzuk létre.
DirectoryInfo dir3 = new DirectoryInfo(@"C:\MyCode\Testing");
dir3.Create();
```

Miután létrehoztuk a `DirectoryInfo` objektumot, megvizsgálhatjuk a könyvtár tartalmát bármely, a `FileSystemInfo` típusból örökölt tulajdonsággal. Példaként hozzunk létre egy új, `DirectoryApp` nevű konzolalkalmazást. A `Program` osztályt egészítsük ki egy új statikus metódussal, amely létrehoz egy olyan új `DirectoryInfo` objektumot a C:\Windows könyvtárra leképezve (szükség esetén módosítsuk az útvonalat), amely számos érdekes statisztikát jelenít meg (a kimenet a 20.2. ábrán látható):

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Directory(Info) *****\n");
    ShowWindowsDirectoryInfo();
    Console.ReadLine();
  }

  static void ShowWindowsDirectoryInfo()
  {
    // A könyvtárinformációk kiírása.
    DirectoryInfo dir = new DirectoryInfo(@"C:\Windows");
    Console.WriteLine("***** Directory Info *****");
    Console.WriteLine("FullName: {0}", dir.FullName);
    Console.WriteLine("Name: {0}", dir.Name);
    Console.WriteLine("Parent: {0}", dir.Parent);
    Console.WriteLine("Creation: {0}", dir.CreationTime);
    Console.WriteLine("Attributes: {0}", dir.Attributes);
    Console.WriteLine("Root: {0}", dir.Root);
    Console.WriteLine("*************************\n");
  }
}
```



20.2. ábra: *Információk megjelenítése a Windows-könyvtárról*

Fájlok listázása a DirectoryInfo típus segítségével

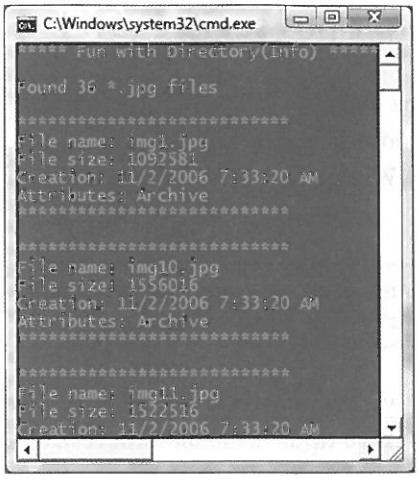
A fenti példát kiegészíthetjük azzal, hogy az alapvető információk lekérdezésén túl a `DirectoryInfo` típus néhány metódusát is használjuk. Először kérdezzünk le adatokat a C:\Windows\Web\Wallpaper könyvtárban található `*.jpg`-fájlokról a `GetFiles()` metódussal.

Megjegyzés Ha a számítógépen nem található C:\Windows\Web\Wallpaper könyvtár, alakítsuk át megfelelően a kódot, például úgy, hogy a C:\Windows könyvtár `*.bmp` fájljait olvassa be.

Ez a metódus olyan `FileInfo` tömbbel tér vissza, amelyek mindegyike egy adott fájlról tartalmaz információkat (a `FileInfo` típus használatáról részletesen később lesz szó). Vegyük a `Program` osztály következő, a `Main()` függvényből meghívott, statikus metódusát:

```
static void DisplayImageFiles()
{
  DirectoryInfo dir =
                new DirectoryInfo(@"C:\Windows\Web\Wallpaper");
  // Az összes *.jpg kiterjesztésű fájl lekérdezése.
  FileInfo[] imageFiles = dir.GetFiles("*.jpg");

  // Mennyit találtunk?
  Console.WriteLine("Found {0} *.jpg files\n", imageFiles.Length);

  // Minden fájlról kiírjuk az információkat.
  foreach (FileInfo f in imageFiles)
  {
    Console.WriteLine("***************************");
    Console.WriteLine("File name: {0}", f.Name);
    Console.WriteLine("File size: {0}", f.Length);
    Console.WriteLine("Creation: {0}", f.CreationTime);
    Console.WriteLine("Attributes: {0}", f.Attributes);
    Console.WriteLine("***************************\n");
  }
}
```

Ha futtatjuk az alkalmazást, akkor a 20.3. ábrához hasonló listát kell látnunk. (A képek nevei különbözhetnek.)



20.3. ábra: *Információk képfájlokról*

Alkönyvtárak létrehozása a DirectoryInfo segítségével

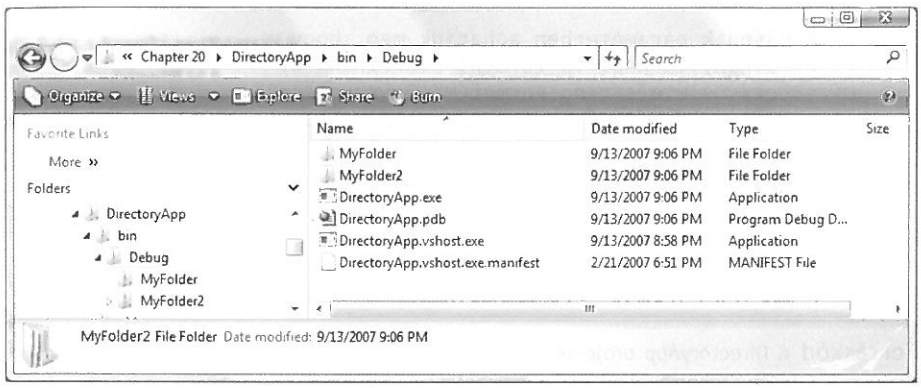
Egy könyvtár szerkezetét a `DirectoryInfo.CreateSubdirectory()` metódussal programozottan bővíthetjük. Ezzel a metódussal akár egy, akár több egymásba ágyazott alkönyvtárat is létrehozhatunk egyszeres függvényhívással. A következő metódus például kibővíti az alkalmazás telepítési útvonalának könyvtárszerkezetét néhány alkönyvtárral:

```
static void ModifyAppDirectory()
{
  DirectoryInfo dir = new DirectoryInfo(".");

  // A \MyFolder létrehozása az alkalmazás könyvtárában.
  dir.CreateSubdirectory("MyFolder");

  // A \MyFolder2\Data létrehozása az alkalmazás könyvtárában.
  dir.CreateSubdirectory(@"MyFolder2\Data");
}
```

Ha meghívjuk ezt a metódust a `Main()` függvényből, és Windows Intézővel megnézzük a Windows-könyvtárat, látható, hogy megjelentek az új alkönyvtárak (lásd a 20.4. ábrát).

A `CreateSubdirectory()` metódus visszatérési értékéről annyit kell tudni, hogy sikeres végrehajtás esetén a metódus visszaad egy `DirectoryInfo` objektumot, amely az újonnan létrehozott könyvtárat jelképezi. Nézzük meg az alábbi példát:

```
static void ModifyAppDirectory()
{
  DirectoryInfo dir = new DirectoryInfo(".");

  // A \MyFolder létrehozása a kezdőkönyvtárban.
  dir.CreateSubdirectory("MyFolder");

  // A visszaadott DirectoryInfo objektum tárolása.
  DirectoryInfo myDataFolder =
               dir.CreateSubdirectory(@"MyFolder2\Data");

  // A ..\MyFolder2\Data elérési útjának kiírása.
  Console.WriteLine("New Folder is: {0}", myDataFolder);
}
```



20.4. ábra: *Alkönyvtárak létrehozása*

A Directory típus használata

A következőkben ismerkedjünk meg a `Directory` típussal. A legtöbb esetben a `Directory` osztály statikus tagjainak működése megfelel annak a funkcionalitásnak, amelyet a `DirectoryInfo` példányszintű tagjai nyújtanak. Ne feledjük azonban, hogy a `Directory` tagjai általában sztringtípusokat adnak vissza, nem pedig erősen típusos `FileInfo`/`DirectoryInfo` típusokat.

A `Directory` típus működésének szemléltetésére ez a végső segédfüggvény megjeleníti a számítógépen található összes meghajtó nevét (a `Directory.GetLogicalDrives()` metódussal), és a `Directory.Delete()` statikus metódussal letörli a korábban létrehozott \MyFolder és \MyFolder2\Data alkönyvtárakat:

```
static void FunWithDirectoryType()
{
  // A számítógép meghajtóinak kilistázása.
  string[] drives = Directory.GetLogicalDrives();
  Console.WriteLine("Here are your drives:");
  foreach (string s in drives)
    Console.WriteLine("--> {0} ", s);

  // Letöröljük, amit létrehoztunk.
  Console.WriteLine("Press Enter to delete directories");
  Console.ReadLine();
  try
  {
    Directory.Delete(string.Format(@"{0}\MyFolder",
                     Environment.CurrentDirectory));

    // A második paraméterben adhatjuk meg, hogy
    // az alkönyvtárakat is törölni szeretnénk-e.
    Directory.Delete(string.Format(@"{0}\MyFolder2",
                     Environment.CurrentDirectory), true);
  }
  catch (IOException e)
  {
    Console.WriteLine(e.Message);
  }
}
```

Forráskód A DirectoryApp projektet a forráskódkönyvtárban a 20. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

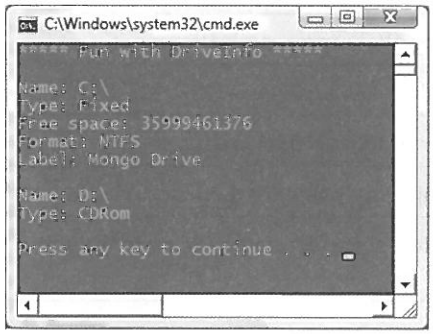
A DriveInfo osztálytípus használata

A `System.IO` névtérben található a `DriveInfo` nevű osztály. A `Directory.GetLogicalDrives()` metódushoz hasonlóan a statikus `DriveInfo.GetDrives()` metódus is a számítógép meghajtóinak a nevét adja vissza, de a `Directory.GetLogicalDrives()` metódussal ellentétben a `DriveInfo` számos egyéb részletet is megad a meghajtókról (pl. a meghajtó típusát, a szabad területet, a kötetcímkét stb.). Nézzük a következő, `DriveInfoApp` nevű új konzolalkalmazásban definiált `Program` osztályt:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with DriveInfo *****\n");
```

```
        // Információk lekérése minden meghajtóról.
        DriveInfo[] myDrives = DriveInfo.GetDrives();

        // A meghajtók statisztikáinak kiírása.
        foreach(DriveInfo d in myDrives)
        {
          Console.WriteLine("Name: {0}", d.Name);
         Console.WriteLine("Type: {0}", d.DriveType);

          // Ellenőrizzük, hogy a meghajtó csatlakoztatva van-e.
          if (d.IsReady)
          {
            Console.WriteLine("Free space: {0}", d.TotalFreeSpace);
            Console.WriteLine("Format: {0}", d.DriveFormat);
            Console.WriteLine("Label: {0}", d.VolumeLabel);
            Console.WriteLine();
          }
        }
        Console.ReadLine();
      }
    }
```

A 20.5. ábra egy lehetséges kimenetet mutat.



20.5. ábra: *Meghajtók adatainak lekérdezése a DriveInfo használatával*

A `Directory`, a `DirectoryInfo` és a `DriveInfo` osztályok alapvető működésének áttekintése után a következőkben megvizsgáljuk, hogyan hozhatunk létre, nyithatunk meg, zárhatunk be és törölhetünk adott könyvtárban lévő fájlokat.

Forráskód A DriveInfoApp projektet a forráskódkönyvtárban a 20. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A FileInfo osztály használata

Ahogy a `DirectoryApp`-példában is láttuk, a `FileInfo` osztály segítségével a számítógépen található fájlokról kérdezhetünk le adatokat (létrehozás ideje, méret, attribútumok stb.), továbbá az osztály segítségével létrehozhatunk, másolhatunk, áthelyezhetünk és törölhetünk fájlokat. A `FileSystemInfo` típusból örökölt funkcionalitáson túl a `FileInfo` osztály néhány egyedi alaptagját a 20.4. táblázatban foglaltuk össze.

Tag	Jelentés
`AppendText()`	Egy `StreamWriter` típust hoz létre (lásd később), amelynek segítségével szöveget fűzhetünk a fájlhoz.
`CopyTo()`	A meglévő fájlt átmásolja egy új fájlba.
`Create()`	Új fájlt hoz létre, és egy `FileStream` típussal (lásd később) tér vissza, amellyel az újonnan létrehozott fájlt kezelhetjük.
`CreateText()`	Egy `StreamWriter` típust hoz létre, amely új szövegfájlt ír.
`Delete()`	Törli a `FileInfo` példány által reprezentált állományt.
`Directory`	A szülőkönyvtár egy példányát adja vissza.
`DirectoryName`	Visszaadja a szülőkönyvtár teljes elérési útját.
`Length`	Visszaadja az aktuális fájl vagy könyvtár méretét.
`MoveTo()`	A fájlt áthelyezi, és lehetőséget ad arra, hogy új nevet adjunk neki.
`Name`	Visszaadja a fájl nevét.
`Open()`	Megnyitja a fájlt különböző olvasási/írási és megosztási jogosultságokkal.
`OpenRead()`	Csak olvasható `FileStream` típust hoz létre.
`OpenText()`	Egy `StreamReader` típust hoz létre (lásd később), amellyel egy létező szövegfájlból olvashatunk.
`OpenWrite()`	Csak írható `FileStream` típust hoz létre.

20.4. táblázat: *FileInfo alapvető tagjai*

A FileInfo osztály metódusainak többsége tehát egy specifikus I/O-központú objektumot (FileStream, StreamWriter stb.) ad vissza. Ezek segítségével a fájlból többféleképpen olvashatunk, illetve írhatunk adatokat. A példák tanulmányozása előtt azonban vizsgáljuk meg azokat a különböző módszereket, amelyekkel a FileInfo osztállyal megszerezhetjük a fájlazonosítóját.

A FileInfo.Create() metódus

A fájlazonosító létrehozásának első módja a FileInfo.Create() metódus használata:

```
static void Main(string[] args)
{
  // Létrehozunk egy új fájlt a C meghajtón.
  FileInfo f = new FileInfo(@"C:\Test.dat");
  FileStream fs = f.Create();

  // Használjuk a FileStream objektumot...

  // Zárjuk le a fájlfolyamot.
  fs.Close();
}
```

A FileInfo.Create() metódus FileStream típussal tér vissza, amely a fájlon mind szinkron, mind aszinkron írási/olvasási műveleteket tesz lehetővé (a részleteket lásd később). A FileInfo.Create() által visszaadott FileStream objektum minden felhasználó számára teljes olvasási/írási hozzáférést biztosít.

Miután befejeztük a FileStream objektummal a munkát, zárjuk le a fájlazonosítót, hiszen így a rendszer felszabadítja az adatfolyam nem felügyelt erőforrásait. Minthogy a FileStream megvalósítja az IDisposable interfészt, a C# using blokkjának a segítségével a fordítóra bízhatjuk a „takarító"-logika létrehozását (a részleteket lásd az első kötet 8. fejezetében):

```
static void Main(string[] args)
{
  // Egy 'using' hatókör definiálása
  // ideális a fájl I/O típusokhoz.
  FileInfo f = new FileInfo(@"C:\Test.dat");
  using (FileStream fs = f.Create())
  {
    // Használjuk a FileStream objektumot...
  }
}
```

A FileInfo.Open() metódus

A `FileInfo.Open()` metódust a `FileInfo.Create()` által nyújtott lehetőségeknél precízebben használhatjuk a fájlok megnyitására és létrehozására, ugyanis az `Open()` jellemzően több paramétert vesz fel a fájl kezelésének a leírására. Az `Open()` meghívása után egy `FileStream` objektumot kapunk vissza. Nézzük meg a következő logikát:

```
static void Main(string[] args)
{
  // Létrehozunk egy új fájlt a FileInfo.Open() metódussal.
  FileInfo f2 = new FileInfo(@"C:\Test2.dat");
  using(FileStream fs2 = f2.Open(FileMode.OpenOrCreate,
        FileAccess.ReadWrite, FileShare.None))
  {
    // Használjuk a FileStream objektumot...
  }
}
```

A túlterhelt `Open()` metódus ezen verziójának három paraméterre van szüksége. Az első paraméter az I/O-kérés típusát határozza meg (pl. új fájl létrehozása, létező fájl megnyitása, hozzáfűzés fájlhoz stb.), amelyet a `FileMode` felsorolt típussal határozhatunk meg (lásd 20.5. táblázat):

```
public enum FileMode
{
  CreateNew,
  Create,
  Open,
  OpenOrCreate,
  Truncate,
  Append
}
```

Tag	Jelentés
`CreateNew`	Az operációs rendszert új fájl létrehozására utasítja. Ha a fájl már létezik, `IOException` kivételt dob.
`Create`	Az operációs rendszert új fájl létrehozására utasítja. Ha a fájl már létezik, akkor felülírja azt.
`Open`	Megnyit egy már létező fájlt. Ha a fájl nem létezik, `FileNotFoundException` kivételt dob.
`OpenOrCreate`	Megnyitja a fájlt, ha az már létezik; ellenkező esetben új fájlt hoz létre.

Tag	Jelentés
`Truncate`	Megnyitja és 0 byte méretűre vágja le a fájlt.
`Append`	Megnyitja a fájlt, a végére pozicionál, és írási műveletet kezd meg (ez a kapcsoló kizárólag csak írható folyamatokkal használható). Ha a fájl nem létezik, új fájlt hoz létre.

20.5. táblázat: *A FileMode felsorolt típus tagjai*

A második paraméter a `FileAccess` felsorolt típus egy olyan értéke, amellyel a folyam olvasási/írási viselkedését határozhatjuk meg:

```
public enum FileAccess
{
  Read,
  Write,
  ReadWrite
}
```

Végül a harmadik paraméter (`FileShare`) azt határozza meg, hogy milyen módon osztjuk meg a fájlt más fájlkezelőkkel. Az alapnevek a következők:

```
public enum FileShare
{
  None,
  Read,
  Write,
  ReadWrite
}
```

A FileInfo.OpenRead() és a FileInfo.OpenWrite() metódusok

Habár a `FileInfo.Open()` metódussal rendkívül rugalmasan kezelhetők a fájlazonosítók, a `FileInfo` osztály további, `OpenRead()` és `OpenWrite()` nevű metódusokat is tartalmaz. Ezek megfelelően konfigurált írásvédett, illetve csak írható `FileStream` típusokat adnak vissza anélkül, hogy különféle felsorolásértékeket kellene ehhez megadnunk. A `FileInfo.Create()` és a `FileInfo.Open()` metódusokhoz hasonlóan az `OpenRead()` és az `OpenWrite()` is `FileStream` objektummal tér vissza (a következő kódban feltételezzük, hogy a C meghajtón létezik a Test3.dat és Test4.dat nevű fájl):

```
static void Main(string[] args)
{
  // Írásvédett FileStream objektum megszerzése.
  FileInfo f3 = new FileInfo(@"C:\Test3.dat");
  using(FileStream readOnlyStream = f3.OpenRead())
  {
    // Használjuk a FileStream objektumot...
  }

  // Most pedig a csak írható FileStream objektumot kapjuk vissza.
  FileInfo f4 = new FileInfo(@"C:\Test4.dat");
  using(FileStream writeOnlyStream = f4.OpenWrite())
  {
    // Használjuk a FileStream objektumot...
  }
}
```

A FileInfo.OpenText() metódus

Az `OpenText()` a `FileInfo` típus egy másik, a fájlmegnyitással kapcsolatos tagja. A `Create()`, az `Open()`, az `OpenRead()` és az `OpenWrite()` metódusokkal szemben az `OpenText()` metódus a `StreamReader` típus egy példányát adja vissza a `FileStream` típus helyett. Feltételezve, hogy a C meghajtón található egy `boot.ini` nevű fájl, annak tartalmát a következőképpen olvashatjuk ki:

```
static void Main(string[] args)
{
  // Visszakapunk egy StreamReader objektumot.
  FileInfo f5 = new FileInfo(@"C:\boot.ini");
  using(StreamReader sreader = f5.OpenText())
  {
    // Használjuk a StreamReader objektumot...
  }
}
```

A `StreamReader` típus segítségével karakteradatokat olvashatunk a mögöttes fájlból.

A FileInfo.CreateText() és a FileInfo.AppendText() metódusok

A két utolsó érdekes metódus a `CreateText()` és az `AppendText()`, amelyek `StreamWriter` referenciát adnak vissza:

```
static void Main(string[] args)
{
  FileInfo f6 = new FileInfo(@"C:\Test5.txt");
  using(StreamWriter swriter = f6.CreateText())
  {
    // Használjuk a StreamWriter objektumot...
  }

  FileInfo f7 = new FileInfo(@"C:\FinalTest.txt");
  using(StreamWriter swriterAppend = f7.AppendText())
  {
    // Használjuk a StreamWriter objektumot...
  }
}
```

A `StreamWriter` típussal karakteradatokat írhatunk a mögöttes fájlba.

A File típus használata

A `File` típus a `FileInfo` osztályhoz hasonló funkcionalitást biztosít a statikus tagokon keresztül. Ugyanúgy, mint a `FileInfo` típusban, a `File` típusban is megtalálhatók az `AppendText()`, a `Create()`, a `CreateText()`, az `Open()`, az `OpenRead()`, az `OpenWrite()` és az `OpenText()` metódusok. Sok esetben a `File` és a `FileInfo` típusok egymással felcserélhetők. Az előző, a `FileStream` használatát bemutató példákat a `File` típus alkalmazásával például egyszerűbbé tehetjük:

```
static void Main(string[] args)
{
  // FileStream objektum megszerzése a File.Create() metódus révén.
  using(FileStream fs = File.Create(@"C:\Test.dat"))
  {
  }

  // FileStream objektum megszerzése a File.Open() metódus révén.
  using(FileStream fs2 = File.Open(@"C:\Test2.dat",
        FileMode.OpenOrCreate,
        FileAccess.ReadWrite, FileShare.None))
  {
  }

  // Írásvédett FileStream objektum megszerzése.
  using(FileStream readOnlyStream = File.OpenRead(@"Test3.dat"))
  {
  }
```

```
    // Csak írható FileStream objektum megszerzése.
    using(FileStream writeOnlyStream = File.OpenWrite(@"Test4.dat"))
    {
    }

    // Visszakapunk egy StreamReader objektumot.
    using(StreamReader sreader = File.OpenText(@"C:\boot.ini"))
    {
    }

    // Néhány StreamWriter megszerzése.
    using(StreamWriter swriter = File.CreateText(@"C:\Test3.txt"))
    {
    }

    using(StreamWriter swriterAppend =
                        File.AppendText(@"C:\FinalTest.txt"))
    {
    }
}
```

További fájlközpontú tagok

A `File` típus rendelkezik néhány egyedi taggal is (lásd 20.6. táblázat), amelyek jelentősen megkönnyítik a szöveges adatok olvasását és írását.

Metódus	Jelentés
`ReadAllBytes()`	Megnyitja a megadott fájlt, visszaadja a bináris adatokat byte-tömbként, majd lezárja a fájlt.
`ReadAllLines()`	Megnyitja a megadott fájlt, visszaadja a karakteradatokat sztringtömbként, majd lezárja a fájlt.
`ReadAllText()`	Megnyitja a megadott fájlt, visszaadja a karakteradatokat `System.String` típusként, majd lezárja a fájlt.
`WriteAllBytes()`	Megnyitja a megadott fájlt, kiírja a byte-tömböt, majd lezárja a fájlt.
`WriteAllLines()`	Megnyitja a megadott fájlt, kiírja a sztringtömböt, majd lezárja a fájlt.
`WriteAllText()`	Megnyitja a megadott fájlt, kiírja a karakteradatokat, majd lezárja a fájlt.

20.6. táblázat: *A File típus metódusai*

A `File` típus új metódusaival, néhány sornyi kóddal olvashatunk vagy írhatunk adatcsomagokat. Sőt mi több, ezek a tagok a művelet végén automatikusan le is zárják a mögöttes fájlazonosítót. A következő konzolalkalmazás (`SimpleFileIO`) például a lehető legkevesebb munkával menti a sztringadatokat egy új fájlba a C meghajtón (majd onnan beolvassa memóriába):

```
using System;
using System.IO;

class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Simple IO with the File Type *****\n");
    string[] myTasks = {
              "Fix bathroom sink", "Call Dave",
              "Call Mom and Dad", "Play Xbox 360"};

    // Az összes adatot kiírjuk egy fájlba a C meghajtón.
    File.WriteAllLines(@"C:\tasks.txt", myTasks);

    // Visszaolvassuk az egészet, és kiírjuk a képernyőre.
    foreach (string task in File.ReadAllLines(@"C:\tasks.txt"))
    {
      Console.WriteLine("TODO: {0}", task);
    }
    Console.ReadLine();
  }
}
```

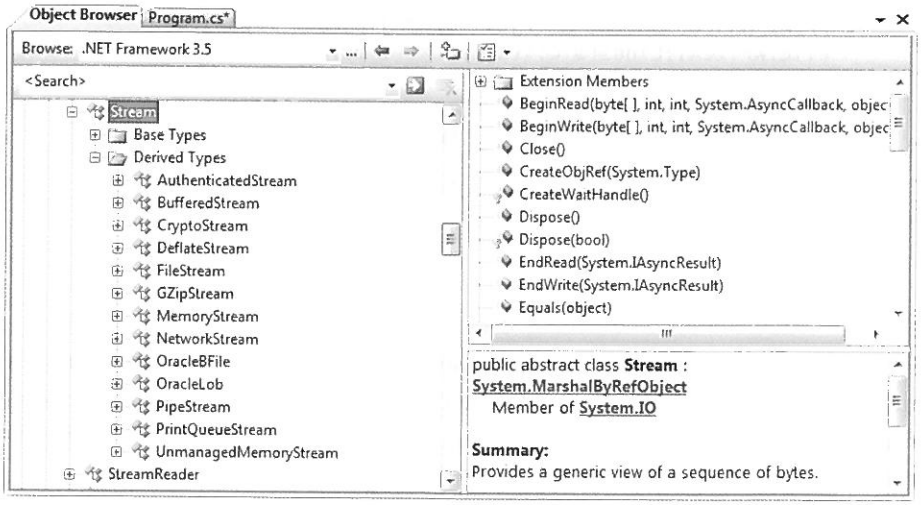
Ha gyorsan szeretnénk fájlazonosítóhoz jutni, a `File` típus egyértelműen megkönnyíti a dolgunkat. Ha pedig előbb létrehozunk egy `FileInfo` objektumot, ennek megvan az az előnye, hogy megvizsgálhatjuk a fájlt az absztrakt `FileSystemInfo` alaposztály tagjainak a segítségével.

Forráskód A SimpleFileIO projekt megtalálható a 20. fejezet alkönyvtárában. A forráskód-könyvtárról lásd a Bevezetés xlv. oldalát.

Az absztrakt Stream osztály

Eddig a `FileStream`, a `StreamReader` és a `StreamWriter` objektumokat még nem használtuk fájlok adatainak írására vagy olvasására. Hogy megértsük ennek menetét, meg kell ismernünk az adatfolyam koncepcióját. Az I/O kezelésében az *adatfolyam* olyan adattöredéket jelképez, amely a forrás és a cél között folyik. Az adatfolyamokkal egyformán kezelhetünk egy *byte-sorozatot* attól függetlenül, hogy milyen eszköz (fájl, hálózati kapcsolat, nyomtató stb.) tárolja vagy jeleníti meg azt.

Az absztrakt `System.IO.Stream` osztály több tagot definiál, amelyek szinkron és aszinkron adatkezelést tesznek lehetővé a tárolóeszközön (pl. mögöttes fájl- vagy memóriahelyen). A 20.6. ábra a `Stream` típus különböző leszármazottait mutatja a Visual Studio 2008 objektumböngészőjében.



20.6. ábra: *A Streamből származó típusok*

Megjegyzés Az adatfolyamok segítségével nem csak fájlokat kezelhetünk. A .NET-könyvtárak adatfolyam-alapú hozzáférést biztosítanak a hálózatokhoz, a memóriaterületekhez és az egyéb adatfolyam-alapú absztrakciókhoz.

A `Stream` osztály leszármazottai tehát az adatokat nyers byte-folyamként jelenítik meg; így gyakran elég nehézkes a nyers adatfolyamokkal dolgozni. Bizonyos `Stream`-leszármazottak támogatják a *keresést*, és ez azt jelenti, hogy lekérdezhetjük vagy beállíthatjuk az adatfolyamban lévő pozíciót.

A `Stream` osztály által nyújtott funkcionalitás megértéséhez nézzük meg annak alapvető tagjait a 20.7. táblázatban.

Tag	Jelentés
`CanRead`, `CanWrite`, `CanSeek`	Megadja, hogy az aktuális adatfolyam támogatja-e az olvasást, a keresést és/vagy az írást.
`Close()`	Lezárja az aktuális adatfolyamot, és felszabadít minden, az adatfolyammal kapcsolatos erőforrást (pl. csatlakozópontokat és fájlazonosítókat). Belsőleg ez tulajdonképpen a `Dispose()` metódus álneve; vagyis az „adatfolyam lezárása" funkcionálisan megegyezik az „adatfolyam eldobásával".
`Flush()`	Frissíti a mögöttes adatforrást vagy adattárat a puffer jelenlegi állapotával, majd törli a puffert. Ha egy adatfolyamhoz nem tartozik puffer, ez a metódus nem csinál semmit.
`Length`	Byte-ban visszaadja az adatfolyam hosszát.
`Position`	Meghatározza a pozíciót az aktuális adatfolyamban.
`Read()`, `ReadByte()`	Kiolvas egy byte-sorozatot (vagy egyetlen byte-ot) az adatfolyamból, és az olvasott byte-ok számával előrébb lépteti az aktuális pozíciót az adatfolyamban.
`Seek()`	Pozicionál az aktuális adatfolyamban.
`SetLength()`	Beállítja az aktuális adatfolyam hosszát.
`Write()`, `WriteByte()`	Kiír egy byte-sorozatot (vagy egyetlen byte-ot) az adatfolyamba, és a kiírt byte-ok számával előrébb lépteti az aktuális pozíciót az adatfolyamban.

20.7. táblázat: *A Stream absztrakt tagjai*

A FileStream típusok használata

A `FileStream` osztály úgy valósítja meg az absztrakt `Stream` típus tagjait, ahogy az a fájlalapú folyamok számára a legmegfelelőbb. Ez egy elég egyszerű adatfolyam; csak egy byte-ot vagy byte-ok tömbjét tudja olvasni. A valóságban közvetlenül nem túlságosan gyakran használjuk a `FileStream` típus tagjait. Ehelyett inkább különböző *adatfolyam-burkolókat* alkalmazunk, amelyek megkönnyítik a szöveges adatokkal vagy a .NET-típusokkal végzett munkát. Ennek ellenére példaként nézzük meg a `FileStream` típus szinkron olvasási/írási lehetőségeit.

Tételezzük fel, hogy van egy új, FileStreamApp nevű konzolalkalmazásunk. A célunk az, hogy egy egyszerű szöveges üzenetet írjuk a myMessage.dat nevű új fájlba. Mivel a FileStream csak nyers byte-okkal tud dolgozni, a System.String típusú adatot előbb egy megfelelő byte-tömbbé kell alakítani. A System.Text névtér egy Encoding nevű típust definiál, amelynek tagjai sztringeket tudnak byte-tömbbe kódolni vagy abból dekódolni (az Encoding típus részletes leírását a .NET Framework 3.5 SDK dokumentációja tartalmazza).

A kódolás végeztével a byte-tömböt a FileStream.Write() metódussal írhatjuk ki a fájlba. Ha a byte-okat vissza szeretnénk olvasni a memóriába, először alaphelyzetbe kell állítanunk az adatfolyam belső pozícióját (a Position tulajdonság segítségével), majd meg kell hívnunk a ReadByte() metódust. Végül megjelenítjük a konzolon a nyers byte-tömböt és a dekódolt sztringet. A teljes Main() metódus a következő:

```
// Ne felejtsük el importálni a System.Text és
// a System.IO névtereket.
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with FileStreams *****\n");

  // A FileStream objektum megszerzése.
  using(FileStream fStream = File.Open(@"C:\myMessage.dat",
        FileMode.Create))
  {
    // A sztring byte-tömbbe történő kódolása.
    string msg = "Hello!";
    byte[] msgAsByteArray = Encoding.Default.GetBytes(msg);

    // A byte[] fájlba írása.
    fStream.Write(msgAsByteArray, 0, msgAsByteArray.Length);

    // Az adatfolyam belső pozíciójának alaphelyzetbe állítása.
    fStream.Position = 0;

    // A típusok kiolvasása a fájlból és megjelenítésük a konzolon.
    Console.Write("Your message as an array of bytes: ");
    byte[] bytesFromFile = new byte[msgAsByteArray.Length];
    for (int i = 0; i < msgAsByteArray.Length; i++)
    {
      bytesFromFile[i] = (byte)fStream.ReadByte();
      Console.Write(bytesFromFile[i]);
    }

    // Dekódolt üzenetek megjelenítése.
    Console.Write("\nDecoded Message: ");
    Console.WriteLine(Encoding.Default.GetString(bytesFromFile));
  }
  Console.ReadLine();
}
```

Bár a fenti példa valóban feltölti a fájlt adatokkal, kiderül a `FileStream` típus közvetlen használatának legnagyobb hátránya is: nyers byte-okkal kell dolgoznunk. Más `Stream`-leszármazott típusok is hasonlóképpen működnek. Ha például egy byte-sorozatot szeretnénk egy adott memóriaterületre kiírni, lefoglalhatunk egy `MemoryStream` objektumot. Hasonlóképp, ha egy byte-tömböt egy hálózati kapcsolaton keresztül kell továbbítanunk, a `NetworkStream` típus lesz a segítségünkre.

A `System.IO` névtér tartalmaz számos „olvasó" és „író" típust, amelyek magukba foglalják a `Streamből` származó típusok működésének részleteit.

Forráskód A FileStreamApp projekt megtalálható a 20. fejezet alkönyvtárában. A forráskód-könyvtárról lásd a Bevezetés xlv. oldalát.

A StreamWriter és StreamReader típusok használata

A `StreamWriter` és `StreamReader` osztályok segítségével karakteralapú adatokat (pl. sztringeket) írhatunk vagy olvashatunk. Az alapértelmezés szerint mindkét típus Unicode-karakterekkel dolgozik, de ezt megváltoztathatjuk, ha megadunk egy megfelelően konfigurált `System.Text.Encoding` objektum-referenciát. Az egyszerűség kedvéért tegyük fel, hogy az alapértelmezett Unicode-kódolás megfelelő.

A `StreamReader` az absztrakt `TextReader` típusból származik ugyanúgy, mint a kapcsolódó `StringReader` típus (erről a későbbiekben még lesz szó). A `TextReader` alaposztály nagyon korlátozott funkcionalitást tesz lehetővé a leszármazottainak, különösen a karakterfolyam olvasásában és megtekintésében.

A `StreamWriter` típus (a `StringWriterhez` hasonlóan, ezt lásd később) az absztrakt `TextWriter` ősosztályból származik. Ez az osztály olyan tagokat definiál, amelyekkel a származtatott típusok szöveges adatot írhatnak egy megadott karakterfolyamba.

A `StreamWriter` és a `StringWriter` osztályok jobb megértésének érdekében a 20.8. táblázat összefoglalja a `TextWriter` ősosztály legfontosabb tagjait.

Tag	Jelentés
`Close()`	A metódus lezárja az írót, és felszabadít minden kapcsolódó erőforrást. A folyamat során a puffer automatikusan kiürül (ez a tag funkcionálisan megegyezik a `Dispose()` metódus meghívásával).
`Flush()`	Ez a metódus kiüríti a puffereket az aktuális íróból, és minden pufferelt adatot kiír a mögöttes eszközre, de nem zárja le az írót.
`NewLine`	Ez a tulajdonság jelöli a származtatott író osztály újsor-konstansát. A Windows operációs rendszer alapértelmezett sorvégjelzője a „kocsivissza", amelyet soremelés követ (`\r\n`).
`Write()`	Ez a túlterhelt metódus adatokat ír ki a szövegfolyamba újsorkonstans nélkül.
`WriteLine()`	Ez a túlterhelt metódus adatokat ír ki a szövegfolyamba újsorkonstanssal.

20.8. táblázat: *A TextWriter alapvető tagjai*

Megjegyzés A `TextWriter` osztály utolsó két tagja ismerős lehet. A `System.Console` típusnak is van `Write()` és `WriteLine()` tagja, amelyek az alapértelmezett kimeneti eszközre írnak szöveges adatot. Valójában a `Console.In` tulajdonság egy `TextWriter`, a `Console.Out` tulajdonság pedig egy `TextReader` objektum.

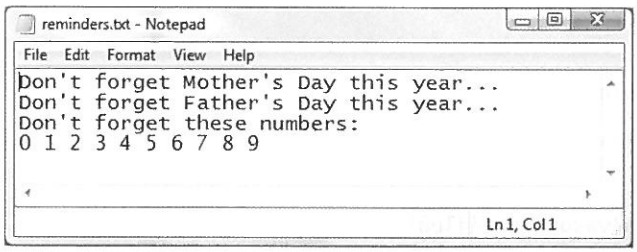
A származtatott `StreamWriter` osztály megfelelő `Write()`, `Close()` és `Flush()` metódusokat valósít meg, emellett egy `AutoFlush` tulajdonsággal is rendelkezik. Ha ezt a tulajdonságot igaz értékre állítjuk, akkor arra kényszeríti a `StreamWriter` típust, hogy kiürítsen minden adatot az írási műveletek után. Jobb teljesítményt érhetünk el, ha az `AutoFlush` tulajdonságot nem kapcsoljuk be, ugyanis ez esetben meghívjuk a `Close()` metódust, miután befejeztük az írást a `StreamWriter` típussal.

Szövegfájl írása

A `StreamWriter` típus működésének tanulmányozásához készítsünk egy új, `StreamWriterReaderApp` nevű konzolalkalmazást. A következő `Main()` metódus létrehoz egy `reminders.txt` nevű fájlt a `File.CreateText()` metódus használatával. A visszakapott `StreamWriter` objektummal szöveges adatot írunk ki az új fájlba:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with StreamWriter /
                    StreamReader *****\n");

  // Megkapjuk a StreamWritert, majd kiírjuk a sztringadatot.
  using(StreamWriter writer = File.CreateText("reminders.txt"))
  {
    writer.WriteLine("Don't forget Mother's Day this year...");
    writer.WriteLine("Don't forget Father's Day this year...");
    writer.WriteLine("Don't forget these numbers:");
    for(int i = 0; i < 10; i++)
      writer.Write(i + " ");

    // Új sort szúrunk be.
    writer.Write(writer.NewLine);
  }
  Console.WriteLine("Created file and wrote some thoughts...");
  Console.ReadLine();
}
```

Ha lefuttatjuk a programot, megvizsgálhatjuk az új fájl tartalmát (lásd 20.7. ábra). A fájl az aktuális alkalmazás bin\Debug mappájában található, feltéve, ha nem abszolút elérési útvonalat adtunk meg a `CreateText()` függvénynek.



20.7. ábra: *A *.txt fájl tartalma*

Olvasás szövegfájlból

A következőkben megnézzük, hogyan olvashatunk programozottan adatot fájlból a megfelelő `StreamReader` típus segítségével. Ez az osztály az absztrakt `TextReader` osztályból származik, amely a 20.9. táblázatban összegzett funkcionalitást nyújtja.

Tag	Jelentés
`Peek ()`	Visszaadja a következő elérhető karaktert anélkül, hogy az olvasó pozícióját módosítaná. Ha `-1` értékkel tér vissza, az azt jelenti, hogy az adatfolyam végére értünk.
`Read ()`	Adatot olvas a bemeneti adatfolyamból.
`ReadBlock()`	Legfeljebb annyi karaktert olvas be az aktuális adatfolyamból, amennyit a `count` paraméterben megadunk, az adatokat pedig kiírja a pufferbe a megadott indextől kezdve.
`ReadLine()`	Beolvas egysornyi karaktert az aktuális adatfolyamból, és sztringként adja vissza az adatot (`null` sztringgel tér vissza, ha a fájl végén vagyunk).
`ReadToEnd()`	Beolvassa az adatfolyamban található valamennyi karaktert az aktuális pozíciótól kezdve, és egyetlen sztringként adja vissza őket.

20.9. táblázat: *A TextReader alapvető tagjai*

Ha úgy módosítjuk a `MyStreamWriterReader` osztályt, hogy az a `StreamReadert` használja, akkor az alábbiak szerint olvashatjuk be a `reminders.txt` fájl szöveges tartalmát:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with StreamWriter /
                    StreamReader *****\n");
  ...
  // Most adatokat olvasunk a fájlból.
  Console.WriteLine("Here are your thoughts:\n");
  using(StreamReader sr = File.OpenText("reminders.txt"))
  {
    string input = null;
    while ((input = sr.ReadLine()) != null)
    {
      Console.WriteLine (input);
    }
  }
  Console.ReadLine();
}
```

Miután lefuttatjuk a programot, a `reminders.txt` fájl tartalmát látjuk a konzolon.

A StreamWriter/StreamReader típusok közvetlen létrehozása

A `System.IO` névtér típusait használva zavaró lehet, hogy ugyanazt az eredményt gyakran többféle megközelítéssel is elérhetjük. Megszerezhetünk például egy `StreamWriter` objektumot a `File` vagy a `FileInfo` révén, a `CreateText()` metódussal. Van azonban még egy módszer, amelyet a `StreamWriter` és a `StreamReader` típusokkal használhatunk: ha közvetlenül hozzuk létre őket. Az aktuális alkalmazást például a következőképpen is módosíthatjuk:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with StreamWriter /
                    StreamReader *****\n");

  // Megkapunk egy StreamWritert, és kiírjuk a sztringadatot.
  using(StreamWriter writer = new StreamWriter("reminders.txt"))
  {
    ...
  }

  // Most adatokat olvasunk a fájlból.
  using(StreamReader sr = new StreamReader("reminders.txt"))
  {
    ...
  }
}
```

Bár zavaró lehet, hogy a fájl I/O-műveleteit ennyiféleképpen végrehajthatjuk, ám ezek mind a nagyobb rugalmasságot biztosítják. A következőkben a `StringWriter` és a `StringReader` osztályok szerepét vizsgáljuk meg.

Forráskód A StreamWriterReaderApp projektet a forráskódkönyvtárban a 20. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A StringWriter és StringReader típusok használata

A `StringWriter` és a `StringReader` típusok segítségével a szöveges információt memóriában tárolt karakterek folyamaként kezelhetjük. Ez akkor lehet hasznos, ha egy mögöttes pufferhez karakteralapú információt kell hozzáfűz-

nünk. Vegyük példaként azt, amikor a StringReaderWriterApp nevű konzolalkalmazás a helyi merevlemezen lévő fájl helyett egy StringWriter objektumba ír sztringadatblokkot.

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with StringWriter /
                    StringReader *****\n");
  // Létrehozzuk a StringWritert, és memóriába írjuk
  // a karakteradatot.
  using(StringWriter strWriter = new StringWriter())
  {
    strWriter.WriteLine("Don't forget Mother's Day this year...");
    // Megkapjuk a tartalom másolatát (sztringben tárolva),
    // és kiírjuk a konzolra.
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);
  }
  Console.ReadLine();
}
```

Mivel a StringWriter és a StreamWriter ugyanabból az ősosztályból (TextWriter) származik, az írás logikája többé-kevésbé megegyezik. A StringWriter osztályból azonban - természeténél fogva - kinyerhető egy System.Text.StringBuilder objektum, amelyet a GetStringBuilder() metódussal kérhetünk le:

```
using (StringWriter strWriter = new StringWriter())
{
  strWriter.WriteLine("Don't forget Mother's Day this year...");
  Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);

  // Megkapjuk a belső StringBuildert.
  StringBuilder sb = strWriter.GetStringBuilder();
  sb.Insert(0, "Hey!! ");
  Console.WriteLine("-> {0}", sb.ToString());
  sb.Remove(0, "Hey!! ".Length);
  Console.WriteLine("-> {0}", sb.ToString());
}
```

Ha egy szöveges adatfolyamból szeretnénk olvasni, használjuk a megfelelő StringReader típust, amely (ahogy az várható) a kapcsolódó StreamReader osztályhoz hasonlóan működik. A StringReader osztály tulajdonképpen nem tesz mást, mint felüldefiniálja az örökölt tagokat, hogy azok ne fájlból, hanem szöveges adatból olvassanak:

```
using (StringWriter strWriter = new StringWriter())
{
  strWriter.WriteLine("Don't forget Mother's Day this year...");
  Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);

  // Kiolvassuk az adatot a StringWriter-ből.
  using (StringReader strReader =
                      new StringReader(strWriter.ToString()))
  {
    string input = null;
    while ((input = strReader.ReadLine()) != null)
    {
      Console.WriteLine(input);
    }
  }
}
```

Forráskód A StringReaderWriterApp projektet a forráskódkönyvtárban a 20. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A BinaryWriter és BinaryReader osztályok használata

Végezetül megvizsgálunk még két írással/olvasással kapcsolatos osztályt, a `BinaryReadert` és a `BinaryWritert`. Mindkettő közvetlenül a `System.Object` osztályból származik. Ezekkel a típusokkal különálló adattípusokat írhatunk és olvashatunk tömör bináris formátumban a mögöttes adatfolyamból. A `BinaryWriter` osztály egy többszörösen túlterhelt `Write()` metódust határoz meg, amellyel egy adattípust írhatunk a mögöttes adatfolyamba. A `Write()` metódus mellett a `BinaryWriter` további tagokkal is rendelkezik, ezek segítségével lekérdezhetjük vagy beállíthatjuk a `Streamből` származó típust, továbbá a `BinaryWriter` véletlen elérést biztosít az adathoz (lásd 20.10. táblázat).

Tag	Jelentés
`BaseStream`	Ez az írásvédett tulajdonság hozzáférést biztosít a `BinaryWriter` objektummal használt mögöttes adatfolyamhoz.
`Close()`	Ez a metódus lezárja a bináris adatfolyamot.
`Flush()`	Ez a metódus kiüríti a bináris adatfolyam pufferét.

Tag	Jelentés
`Seek()`	Ez a metódus beállítja a pozíciót az aktuális adatfolyamban.
`Write()`	Ez a metódus kiír egy értéket az aktuális adatfolyamba.

20.10. táblázat: *A BinaryWriter legfontosabb tagjai*

A `BinaryReader` osztály a `BinaryWriter` által nyújtott működést a 20.11. táblázatban szereplő tagokkal egészíti ki.

Tag	Jelentés
`BaseStream`	Ez az írásvédett tulajdonság hozzáférést biztosít a `BinaryReader` objektummal használt mögöttes adatfolyamhoz.
`Close()`	Ez a metódus lezárja a bináris olvasót.
`PeekChar()`	Ez a metódus visszaadja a következő elérhető karaktert anélkül, hogy módosítaná a pozíciót az adatfolyamban.
`Read ()`	Ez a metódus kiolvas egy adott byte- vagy karaktersorozatot, és eltárolja őket a bemeneti tömbben.
`ReadXXXX()`	A `BinaryReader` osztály számos olyan olvasási metódust definiál, amelyek kiolvassák a következő típust az adatfolyamból (`ReadBoolean()`, `ReadByte()`, `ReadInt32()` stb.).

20.11. táblázat: *A BinaryReader alapvető tagjai*

A következő példa (`BinaryWriterReader` nevű konzolalkalmazás) többféle adattípust ír ki egy új `*.dat` fájlba:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Binary Writers /
                    Readers *****\n");

  // Megnyitunk egy bináris írót egy fájl számára.
  FileInfo f = new FileInfo("BinFile.dat");
  using(BinaryWriter bw = new BinaryWriter(f.OpenWrite()))
  {
    // Kiírjuk a BaseStream tulajdonság típusát.
    // (Jelen esetben System.IO.FileStream.)
    Console.WriteLine("Base stream is: {0}", bw.BaseStream);

   // Létrehozunk néhány adatot, amelyet elmenthetünk a fájlba.
   double aDouble = 1234.67;
   int anInt = 34567;
   string aString = "A, B, C";
```

```
    // Az adatok írása.
    bw.Write(aDouble);
    bw.Write(anInt);
    bw.Write(aString);
  }
  Console.ReadLine();
}
```

Figyeljük meg, hogy a `FileInfo.OpenWrite()` által visszaadott `FileStream` objektumot hogyan adjuk át a `BinaryWriter` típus konstruktorának. A módszer segítségével könnyen közbeiktathatunk egy másik „adatfolyamréteget" az adatok kiírása előtt. A `BinaryWriter` konstruktora bármilyen, `Stream`-leszármazott típust (pl. `FileStream`, `MemoryStream` vagy `BufferedStream`) kezelhet. Vagyis, ha a memóriába bináris adatot szeretnénk írni, egyszerűen adjunk meg egy érvényes `MemoryStream` objektumot.

A bináris adatok kiolvasására a `BinFile.dat` fájlból a `BinaryReader` típus több lehetőséget is ad. Meghívunk több olvasásközpontú tagot, hogy kiolvassanak minden adattöredéket a fájlfolyamból:

```
static void Main(string[] args)
{
  ...
  FileInfo f = new FileInfo("BinFile.dat");
  ...
  // Kiolvassuk a bináris adatokat az adatfolyamból.
  using(BinaryReader br = new BinaryReader(f.OpenRead()))
  {
    Console.WriteLine(br.ReadDouble());
    Console.WriteLine(br.ReadInt32());
    Console.WriteLine(br.ReadString());
  }
  Console.ReadLine();
}
```

Forráskód A BinaryWriterReader projektet a forráskódkönyvtárban a 20. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Fájlok programozott „figyelése"

A következőkben nézzük meg a `FileSystemWatcher` osztály szerepét. Ez a típus meglehetősen hatékonyan segít a rendszerben található fájlok programozott felügyeletében (vagy megfigyelésében). Ez azt jelenti, hogy a `FileSystemWatcher` típust utasíthatjuk a fájlok felügyeletére a `System.IO.NotifyFilters` által felsorolt műveletek tekintetében (bár a tagok többsége egyértelmű, célszerű elolvasni a .NET Framework 3.5 SDK dokumentációjának idevágó részeit):

```
public enum NotifyFilters
{
  Attributes, CreationTime,
  DirectoryName, FileName,
  LastAccess, LastWrite,
  Security, Size,
}
```

A `FileSystemWatcher` használatának első lépéseként be kell állítanunk a `Path` tulajdonságot a megfigyelendő fájlokat tartalmazó mappa elérési útvonalára, valamint a `Filter` tulajdonságban meg kell adnunk a felügyelendő fájlok kiterjesztéseit.

Itt eldönthetjük, hogy kezeljük-e a `Changed`, `Created` és `Deleted` eseményeket, amelyek mindegyike a `FileSystemEventHandler` metódusreferenciával dolgozik együtt. Ez a metódusreferencia bármely, a következő mintát követő metódust meghívhatja:

```
// A FileSystemEventHandler metódusreferenciának egy,
// a következő szignatúrának megfelelő metódust kell meghívnia.
void MyNotificationHandler(object source, FileSystemEventArgs e)
```

A `Renamed` eseményt is kezelhetjük a `RenamedEventHandler` metódusreferenciatípussal, amely a következő szignatúrának megfelelő metódusokat hívhatja meg:

```
// A RenamedEventHandler metódusreferenciának egy,
// a következő szignatúrának megfelelő metódust kell meghívnia.
void MyNotificationHandler(object source, RenamedEventArgs e)
```

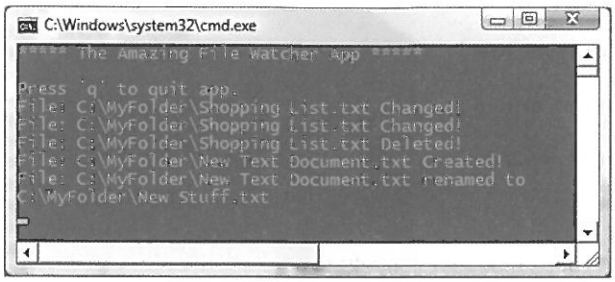
A fájlok megfigyelésének folyamatát a következő példa szemlélteti; ehhez azt feltételezzük, hogy a C meghajtón létrehoztunk egy `MyFolder` nevű új könyvtárat, amelyben különböző `.txt` fájlok találhatók (tetszés szerint elnevezve). Az alábbi (`MyDirectoryWatcher` nevű) konzolalkalmazás a `MyFolder` könyvtárban van. A `*.txt` fájlokat fogja felügyelni, és üzenetet ír ki a konzolra, ha új fájlt hozunk létre, illetőleg meglévő fájlt törlünk, módosítunk vagy átnevezünk:

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Amazing File Watcher App *****\n");

  // Megadjuk a megfigyelendő mappa elérési útját.
  FileSystemWatcher watcher = new FileSystemWatcher();
  try
  {
    watcher.Path = @"C:\MyFolder";
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
    return;
  }

  // Beállítjuk a monitorozandó eseményeket.
  watcher.NotifyFilter = NotifyFilters.LastAccess
                       | NotifyFilters.LastWrite
                       | NotifyFilters.FileName
                       | NotifyFilters.DirectoryName;

  // Csak szövegfájlokat figyelünk.
  watcher.Filter = "*.txt";

  // Hozzáadjuk az eseménykezelőket.
  watcher.Changed += new FileSystemEventHandler(OnChanged);
  watcher.Created += new FileSystemEventHandler(OnChanged);
  watcher.Deleted += new FileSystemEventHandler(OnChanged);
  watcher.Renamed += new RenamedEventHandler(OnRenamed);

  // Megfigyeljük a könyvtárat.
  watcher.EnableRaisingEvents = true;

  // Várunk, amíg a felhasználó ki nem lép.
  Console.WriteLine(@"Press 'q' to quit app.");
  while(Console.Read()!='q');
}
```

A két eseménykezelő egyszerűen kiírja az aktuális fájl módosításait:

```
static void OnChanged(object source, FileSystemEventArgs e)
{
  // Mi történjen, ha a fájl megváltozik, létrehozzák vagy letörlik?
  Console.WriteLine("File: {0} {1}!", e.FullPath, e.ChangeType);
}

static void OnRenamed(object source, RenamedEventArgs e)
{
  // Mi történjen, ha a fájlt átnevezik?
  Console.WriteLine("File: {0} renamed to\n{1}", e.OldFullPath,
                    e.FullPath);
}
```

A program kipróbálásához indítsuk el az alkalmazást, és nyissuk meg a Windows Intézőt. Próbáljuk meg átnevezni a fájlokat, hozzunk létre és töröljünk egy `*.txt` fájlt stb. A `MyFolder` könyvtárban lévő szövegfájlok állapotával kapcsolatos különböző információk láthatóvá válnak (lásd 20.8. ábrát).



20.8. ábra: *Szövegfájlok megfigyelése*

Forráskód A MyDirectoryWatcher alkalmazást a forráskódkönyvtárban a 20. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Aszinkron fájlolvasás és -írás

A `System.IO` névtér áttekintésének befejezéseként nézzük meg, miként hajthatunk végre aszinkron műveleteket a `FileStream` típussal. Az első kötetben, a többszálú feldolgozás elemzésekor már láthattuk, hogyan támogatja a .NET keretrendszer az aszinkron műveleteket (lásd a 18. fejezetet). Mivel a nagy fájlok írása és olvasása időigényes művelet lehet, a `System.IO.Stream` osztályból származó típusok több, az aszinkron adatfeldolgozást lehetővé tevő metódussal rendelkeznek. Ezek a metódusok az `IAsyncResult` típust használják:

```
public abstract class Stream :
      MarshalByRefObject, IDisposable
{
  ...
  public virtual IAsyncResult BeginRead(byte[] buffer, int offset,
        int count, AsyncCallback callback, object state);
  public virtual IAsyncResult BeginWrite(byte[] buffer, int offset,
        int count, AsyncCallback callback, object state);
  public virtual int EndRead(IAsyncResult asyncResult);
  public virtual void EndWrite(IAsyncResult asyncResult);
}
```

A Stream osztályból származtatott típusok aszinkron viselkedésével végzett munka hasonló ahhoz, mint amikor aszinkron metódusreferenciákkal és aszinkron távoli metódushívásokkal dolgozunk. Bár nem valószínű, hogy az aszinkron viselkedés nagyban meggyorsítja a fájlok elérését, más adatfolyamok (pl. a csatlakozópont-alapúak) aszinkron kezelése során jelentősebb teljesítménynövekedést tapasztalhatunk. A következő (AsyncFileStream nevű) konzolalkalmazás bemutatja a FileStream típus aszinkron kezelésének egyik módját (mindenképpen importáljuk a System.Threading és a System.IO névtereket):

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Async File I/O *****\n");

    Console.WriteLine("Main thread started. ThreadID = {0}",
    Thread.CurrentThread.GetHashCode());

    // Egy FileStream aszinkron olvasási vagy írási hozzáféréséhez
    // ezt a konstruktort kell használnunk.
    FileStream fs = new FileStream("logfile.txt", FileMode.Append,
                                   FileAccess.Write,
                                   FileShare.None, 4096, true);

    string msg = "this is a test";
    byte[] buffer = Encoding.ASCII.GetBytes(msg);

    // Az aszinkron írás elkezdése. A program a WriteDone-t hívja
    // meg, ha végzett. Figyeljük meg, hogy a visszahívási metódus
    // a FileStream objektumot kapja meg állapotinformációként.
    fs.BeginWrite(buffer, 0, buffer.Length,
    new AsyncCallback(WriteDone), fs);
  }

  private static void WriteDone(IAsyncResult ar)
  {
    Console.WriteLine("AsyncCallback method on ThreadID = {0}",
                      Thread.CurrentThread.GetHashCode());

    Stream s = (Stream)ar.AsyncState;
    s.EndWrite(ar);
    s.Close();
  }
}
```

A fenti példa egyetlen érdekessége az, hogy a FileStream típus aszinkron viselkedésének engedélyezéséhez csak a példában szereplő konstruktort használhatjuk. A legutolsó, System.Boolean típusú paraméter (ha igaz értéket kap) arra utasítja a FileStream objektumot, hogy a munkát egy másik szálon végezze.

Forráskód A AsyncFileStream alkalmazást a forráskódkönyvtárban a 20. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Az elszigetelt tároló szerepe

Minden előző, fájl I/O-val kapcsolatos példánkban az alkalmazás végrehajtásakor azt feltételeztük, hogy a program a CLR-től *mindent megengedő (full trust) biztonsági privilégiumot* kap. Egy számítógép merevlemezéről olvasni vagy a lemezre írni potenciális biztonsági fenyegetést jelenthet az alkalmazás eredetétől függően. A .NET platform támogatja, hogy különböző helyekről szerelvényeket töltsünk le, többek között külső webhelyekről, az intranetről, de akár a `System.Reflection.Emit` névtér típusainak segítségével dinamikusan, a memóriában létrehozott új szerelvénnyel is megtehetjük ezt (lásd az előző kötet 19. fejezetét).

Az első kötet 15. fejezetében szó volt arról, hogy az ügyfél `*.config` fájljának `<codeBase>` eleme lehetővé teszi, hogy deklaratív módon tetszőleges elérési utat határozzunk meg, ahonnan a CLR külső szerelvényeket tölthet be, míg az `Assembly.LoadFrom()` metódussal pedig (lásd a 16. fejezetet) programozottan tölthetünk be külső URI-ról szerelvényt.

Bizalom kérdése

Figyelembe véve, hogy .NET-szerelvényeket a számítógép helyi merevlemezén kívül számos különböző helyről tölthetünk be, a bizalom kérdése igen fontossá válik. Tegyük fel például, hogy van egy olyan alkalmazásunk, amely távoli helyről letöltött kódot futtat. Ha ez a kódkönyvtár a helyi számítógépről fájlokat próbál olvasni, vajon hogyan bizonyosodhatunk meg arról, hogy nem akar érzékeny adatokat olvasni? Hasonlóképpen, ha távoli számítógépről töltünk le végrehajtható fájlt, mi garantálja, hogy a szerelvény nem próbál érzékeny adatokat kiolvasni a rendszerleíró adatbázisból, nem intéz (rosszindulatú) hívásokat az operációs rendszer mögöttes API-jához, vagy nem jelent egyéb potenciális biztonsági kockázatot?

Erre a .NET platform válasza a kóderedet-alapú biztonság (CAS) elnevezésű, .NET-központú biztonsági mechanizmus. A CAS segítségével a CLR biztonsági kiváltságokat adhat a végrehajtó szerelvénynek; többek között az alábbiakat:

- a számítógép könyvtár- és fájlrendszerének módosítása,
- hálózati, internet- és adatbáziskapcsolatok kezelése,
- új alkalmazástartományok és dinamikus szerelvények létrehozása,
- a .NET reflexiós szolgáltatás használata,
- nem felügyelt kód meghívása a PInvoke segítségével.

Az I/O-specifikus biztonsági kérdésekre koncentrálva tételezzük fel, hogy olyan alkalmazást fejlesztünk, amelyet egy távoli, webalapú URL-en keresztül telepítünk külső felhasználóknak. Attól függően, hogy a telepítőszkriptet hogyan állítjuk be (és hogy a felhasználó számítógépén milyen biztonsági házirend van érvényben), lehetséges, hogy a program a helyi fájlrendszer elérését *tiltó* szigorú biztonsági környezetben fut. Ha az alkalmazás felhasználói vagy alkalmazásbeállításokat próbál eltárolni a `System.IO` névtérrel, a CLR futásidejű biztonsági kivételeket fog dobni.

A `System.IO.IsolatedStorage` névtér típusai használhatók olyan alkalmazás létrehozására, amely a .NET-et futtató számítógép speciális, *elszigetelt tárolónak* nevezett területén tárolja az alkalmazás adatait. Ez valójában egy biztonságos futtatókörnyezet (angolul: „sandbox"), amelyben a CLR lehetővé teszi az írási/olvasási műveleteket még akkor is, ha az alkalmazást külső URL-ről töltötték le, vagy a rendszergazda más okból helyezte ebbe a virtuális környezetbe.

Az elszigetelt tárolóhoz tartozó API egyéb felhasználási módjai

Az elszigetelt tárolóhoz tartozó API-t nem kizárólag a távoli számítógépekről letöltött kódok olvasási/írási műveleteinek szabályozására használhatjuk. Az API-val azt is könnyen megvalósíthatjuk, hogy a felhasználói adatokat más alkalmazás közvetetten (vagy közvetlenül) ne módosíthassa. Írhatunk például olyan egyszerű alkalmazást, amely az adott munkaállomás összes felhasználójának adatát elkülönített mappákban tárolja.

Az elszigetelt tárolóhoz tartozó API használatának másik előnye az, hogy a kódnak nincs szüksége bedrótozott elérési utakra vagy könyvtárnevekre a programban. Ehelyett az elszigetelt tároló használatakor az alkalmazás indirekt módon menti a kód azonosítójával (pl. URL, erős név vagy X509 digitális aláírás) valamilyen módon összefüggésbe hozható egyedi adattároló területre az adatokat (a kódazonosítók ismertetését lásd később).

Szerencsére az elszigetelt tároló kezelése igen egyszerű azoknak, akik értik az alapvető fájlműveleteket. Az elszigetelt tároló működésének tanulmányozása előtt nézzük át a .NET kódhozzáférés-szabályozási modelljét.

Megjegyzés A CAS teljes bemutatása legalább egy (vagy két) teljes fejezetet igényelne, ezért a CAS működését kizárólag olyan szinten tekintjük át, hogy megérthessük az elszigetelt tároló-API szerepének alapjait. A CAS működésének további részleteit a .NET Framework 3.5 SDK tartalmazza.

Bevezetés a kóderedet-alapú biztonságba

A távoli .NET-szerelvények letöltése és futtatása által jelentett biztonsági problémák megelőzésének érdekében a CLR automatikusan azonosítja a szerelvényt, és besorolja az előre meghatározott kódcsoportok egyikébe. Leegyszerűsítve a *kódcsoport* egy olyan szerelvénygyűjtemény, amelynek tagjai ugyanazon feltételeknek felelnek meg (pl. ugyanaz az eredetük).

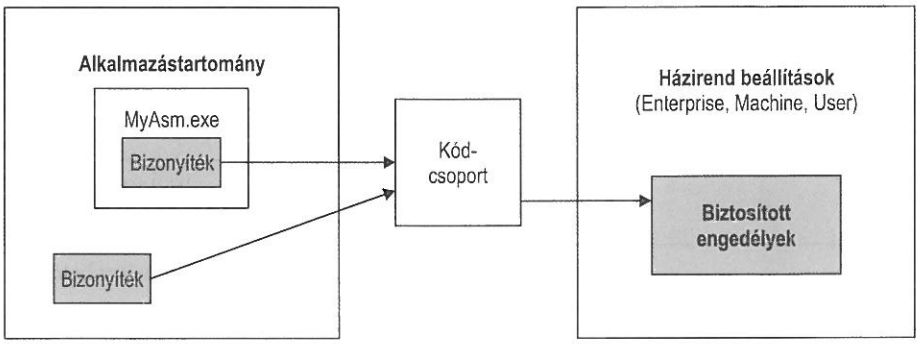
Azt a feltételt, amely alapján a CLR eldönti egy szerelvényről, hogy az melyik kódcsoportba tartozik, *bizonyítéknak* nevezzük. A szerelvényeket eredetükön kívül más bizonyíték alapján is besorolhatjuk egy kódcsoportba. Ilyen lehet a szerelvény erős neve, egy beágyazott X509 digitális tanúsítvány vagy bármely, általunk programozottan elkészített egyedi feltétel.

Megjegyzés Szigorúan véve kétféle bizonyítékról beszélhetünk: ezek a szerelvényalapú és a hosztalapú bizonyítékok. Míg a szerelvényalapú bizonyítékot lefordítjuk a szerelvénybe, a hosztalapú bizonyítékot programozottan, az `AppDomain` típus segítségével határozhatjuk meg.

Miután a CLR kiértékelte a szerelvény bizonyítékait, és eldöntötte, hogy melyik kódcsoportba kerüljön, annak érdekében, hogy meghatározhassa, a szerelvény mit tehet, és főként mit *nem tehet*, lekérdezi a kódcsoporthoz társított *engedélykészletet* (ez egyszerűen a különálló engedélyek nevesített készlete).

A kódcsoportok és a hozzájuk tartozó engedélyek együtt alkotják a *biztonsági házirendet*, amelyet három nagyobb szintre oszthatunk (vállalati, gép-, illetve felhasználószintű). Ennek a réteges megközelítésnek a használatával a rendszergazdának lehetősége van arra, hogy egyedi házirendet alakítson ki a vállalat, illetve az egyes számítógépek/felhasználók szintjén.

Miután mind a három (vállalati, számítógép- és felhasználói) biztonsági házirend alkalmazása megtörtént, a szerelvény lefut a .NET futásidejű környezetében. Ha a szerelvény a jogosultsági készletébe nem tartozó kódot próbál futtatni, a CLR futásidejű biztonsági kivételt fog dobni. A 20.9. ábra mutatja be, hogy a CAS ezen építőelemei (bizonyíték, kódcsoportok/engedélykészletek és házirendek) hogyan fonódnak össze.



20.9. ábra: *A CAS építőelemei*

A bizonyíték kiértékelésének, a szerelvény kódcsoportba sorolásának és a jogosultsági készletek hozzárendelésének folyamata minden alkalommal, egy .NET-alkalmazás futtatásakor a háttérben, észrevehetetlenül zajlik le. Legtöbbször az alapértelmezett CAS biztonsági beállítás, valamint a CLR és a CAS közötti interakció a háttérben történik, felhasználói beavatkozás nélkül. Érdemes azonban kicsit részletesebben szemügyre venni a CAS építőelemeit, kezdve a szerelvénybizonyíték fogalmával.

A bizonyítékok szerepe

Annak érdekében, hogy a CLR meghatározza, hogy egy szerelvény melyik kódcsoportba kerüljön, először kiolvassa a megadott bizonyítékot. A bizonyíték tehát olyan adat, amelyet a szerelvényből (vagy az azt hosztoló alkalmazástartományból) szerzünk meg, amikor az betöltődik a memóriába. A 20.12. táblázat írja le azokat a főbb bizonyítéktípusokat, amelyet a szerelvény bemutathat a CLR-nek.

Hosztbizonyíték-típus	Jelentés
Alkalmazáskönyvtár	A szerelvény telepítési könyvtára.
Szerelvényhashkód	A szerelvény tartalmának hashértéke.
Kibocsátó tanúsítványa	A szerelvényhez rendelt Authenticode X509-es digitális tanúsítvány (ha van).
Hely	Az a forráswebhely, amelyről a szerelvényt betöltötték (nem érvényes a helyi gépről betöltött szerelvények esetében).
A szerelvény erős neve	A szerelvény erős neve (ha van ilyen).
URL	Az az URL, ahonnan a szerelvény betöltődött (HTTP, FTP, fájl stb.).
Zóna	Annak a zónának a neve, ahonnan a szerelvény betöltődött.

20.12. táblázat: *Különböző típusú szerelvénybizonyítékok*

Bár a bizonyítékok beolvasása automatikusan történik, arra is van lehetőségünk, hogy a bizonyítékokat programozottan olvassuk be a reflexiós API és a `System.Security.Policy` névtér `Evidence` típusának a segítségével. A bizonyítékok megértéséhez hozzunk létre egy új, `MyEvidenceViewer` nevű konzolalkalmazást. Ne felejtsük el importálni a `System.Reflection`, a `System.Collections` és a `System.Security.Policy` névtereket.

Ezután egy olyan egyszerű alkalmazást írunk, amely megkéri a felhasználót, hogy adja meg a betöltendő szerelvény nevét. Számba vesszük valamennyi szerelvénybizonyítékot, majd az eredményt kiírjuk a konzolablakba. Először is, a `Program` osztály `Main()` metódusa lehetővé teszi, hogy a felhasználó megadja a kiértékelendő szerelvény teljes elérési útját. Ha az L lehetőséget választja, meghívunk egy segédmetódust, amely megpróbálja betölteni a megadott szerelvényt a memóriába. Ha sikerült, átadjuk az `Assembly` referenciát egy másik, `DisplayAsmEvidence()` nevű segédmetódusnak. A megvalósítás a következő:

```
class Program
{
  static void Main(string[] args)
  {
    bool isUserDone = false;
    string userOption = "";
    Assembly asm = null;
```

```
    Console.WriteLine("***** Evidence Viewer *****\n");
    do
    {
      Console.Write("L (load assembly) or Q (quit): ");
      userOption = Console.ReadLine();
      switch (userOption.ToLower())
      {
        case "l":
          asm = LoadAsm();
          if (asm != null)
          {
            DisplayAsmEvidence(asm);
          }
          break;

        case "q":
          isUserDone = true;
          break;

        default:
          Console.WriteLine("I have no idea what you want!");
          break;
      }
    } while (!isUserDone);
  }
}
```

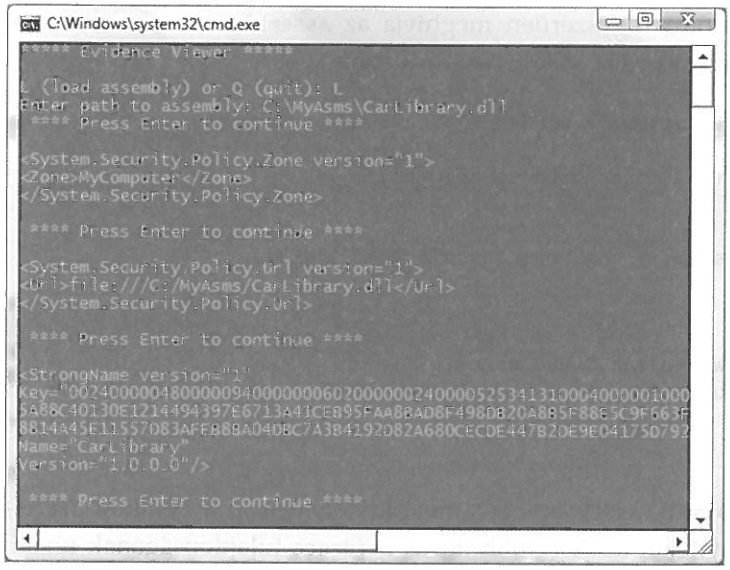
A `LoadAsm()` metódus egyszerűen meghívja az `Assembly.LoadForm()` metódust, hogy beállítsa a privát `Assembly` tagváltozót:

```
private static Assembly LoadAsm()
{
  Console.Write("Enter path to assembly: ");
  try
  {
    return Assembly.LoadFrom(Console.ReadLine());
  }
  catch
  {
    Console.WriteLine("Load error...");
    return null;
  }
}
```

Végül pedig a `DisplayAsmEvidence()` metódus olvassa ki a betöltött szerelvényből a bizonyítékot az Assembly típus `Evidence` tulajdonságának révén. Ezután egy felsoroló segítségével (az `Evidence` típus `GetHostEvidence()` metódusával) kiírjuk az összes jelenlévő bizonyítéktípust:

```
private static void DisplayAsmEvidence(Assembly asm)
{
  // A bizonyítékgyűjtemény és a mögöttes felsoroló lekérése.
  Evidence e = asm.Evidence;
  IEnumerator itfEnum = e.GetHostEnumerator();

  // A bizonyíték kiírása.
  while (itfEnum.MoveNext())
  {
    Console.WriteLine(" **** Press Enter to continue ****");
    Console.ReadLine();
    Console.WriteLine(itfEnum.Current);
  }
}
```

Az alkalmazás teszteléséhez érdemes a C meghajtó gyökerében létrehozni egy `MyAsms` mappát, majd ebbe a mappába belemásolni az erős névvel rendelkező `CarLibrary.dll` szerelvényt (lásd az első kötet 15. fejezetét), végül elindítani a programot. Feltételezve, hogy az L parancs mellett döntünk, adjuk meg a szerelvény teljes elérési útvonalát, majd nyomjuk meg az Enter billentyűt. Az alkalmazás kiírja a konzolra az összes bizonyítékot, ahogy az a 20.10. ábrán is látható (fontos, hogy az alkalmazás konstrukciója szerint minden egyes bizonyítéktípus megjelenítése után Entert kell ütni).



20.10. ábra: *A CarLibrary.dll bizonyítékainak megtekintése*

Látható, hogy a `CarLibrary.dll` a `MyComputer` zónába került a C:\MyAsms\CarLibrary.dll URL-ről, és rendelkezik erős névvel. Ha a szerelvényt más helyről töltjük be (mondjuk távoli webhelyről), akkor teljesen más kimenetet fogunk kapni. A lényeg az, hogy a szerelvény memóriába való betöltésekor a CLR megvizsgálja a szerelvény által szolgáltatott bizonyítékokat.

Forráskód A MyEvidenceViewer projektet a forráskódkönyvtárban a 20. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A kódcsoportok szerepe

A bizonyítékok használatával a CLR a szerelvényeket kódcsoportokba sorolhatja. Minden kódcsoportot egy biztonsági zónához rendeltek, amelyek gyárilag meghatározott biztonsági beállításokkal rendelkeznek. A CLR ezek segítségével határozza meg, hogy a szerelvény mit tehet meg, és mit nem. A 20.13. táblázat a leggyakoribb kódcsoportokat mutatja be az alapértelmezés szerint hozzájuk rendelt engedélykészlettel együtt (ezekről a következő részben lesz szó).

Alapértelmezett kódcsoport	Hozzárendelt engedélyek	Jelentés
My_Computer_Zone	Full Trust	Közvetlenül a helyi merevlemezről betöltött szerelvényt jelképez.
LocalIntranet_Zone	LocalIntranet	A helyi intranet egy megosztási pontjáról letöltött szerelvényt jelez.
Internet_Zone	Internet	A webről letöltött szerelvényt jelez.

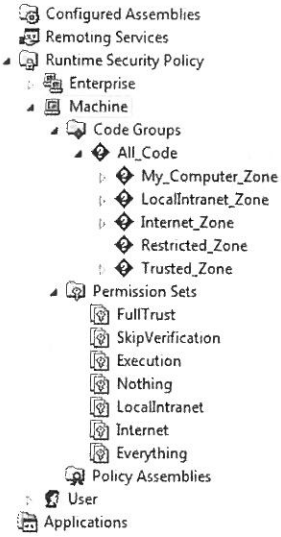
20.13. táblázat: *Néhány gyakoribb kódcsoport*

A .NET keretrendszer 3.5 SDK egy olyan grafikus felügyeleti eszközt biztosít, amellyel a rendszergazdák megnézhetik és módosíthatják a meglévő kódcsoportokat, vagy szükség szerint újat hozhatnak létre. Az eszköz segítségével beállíthatjuk például, hogy a CLR minden, adott címről (pl. http://www.intertech.com) letöltött külső szerelvényt egy testre szabott biztonsági környezetben futtasson.

Megjegyzés A .NET Framework 3.5 SDK egy `caspol.exe` nevű parancssori programot is tartalmaz, amellyel ugyanezeket a beállításokat érhetjük el.

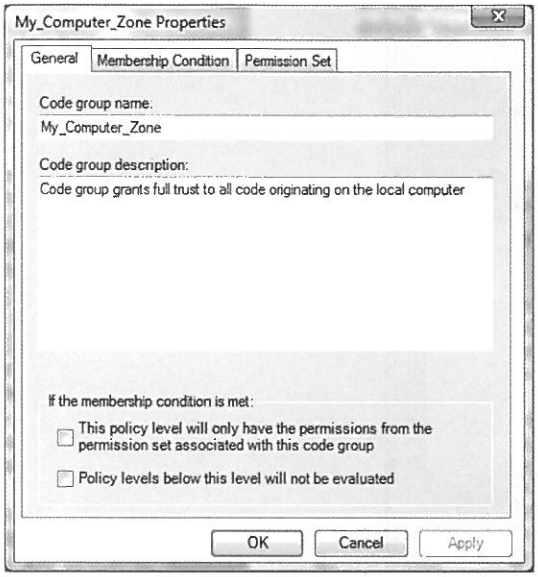
Ezt az eszközt a Control Panel Administrative Tools mappájában találjuk Microsoft .NET Framework Configuration néven. Az eszköz elindítása után a CAS-beállításokat három szinten módosíthatjuk: vállalati szinten (pl. minden hálózatba kapcsolt gép), a helyi gép szintjén, illetve felhasználónként. (A CAS-t alkalmazástartományi szinten is beállíthatjuk, de ezt csak programozottan tehetjük meg.) A 20.11. ábra a CAS alapértelmezett „számítógépszintű házirend" beállításainak lenyitott csomópontját mutatja aktuálisan.

Az All_Code kódcsoport (amely az összes .NET-szerelvényt jelenti) több olyan zónát is meghatároz, amelybe a szerelvény tartozhat (pl. My_Computer_Zone stb.). Ha jobb gombbal kattintunk az All_Code gyökér alatt található bármelyik csomópontra, megnyithatjuk az adott objektum tulajdonságlapját, amely további részletekkel szolgál. A 20.12. ábra például a My_Computer_Zone kódcsoport (amely a közvetlenül a helyi merevlemezről betöltött szerelvényeket tartalmazza) tulajdonságait mutatja.

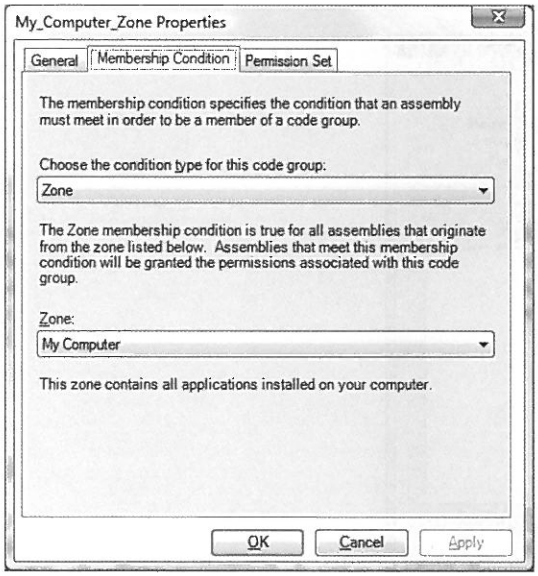


20.11. ábra: *A CAS-központú számítógépházirend*

A Membership Condition fülre kattintva látható, hogy a CLR mi alapján dönti el, hogy egy adott .NET-szerelvénynek ebbe a zónába kell-e tartoznia. A 20.13. ábra mutatja, hogy a tagsági feltétel a nem túl beszédes Zone-érték, amely azt a helyet jelöli, ahonnan a szerelvényt betöltöttük.

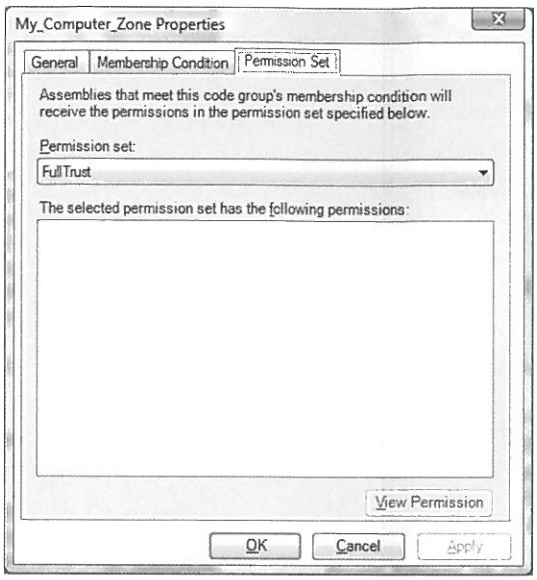


20.12. ábra: *A My_Computer_Zone kódcsoport részletei*

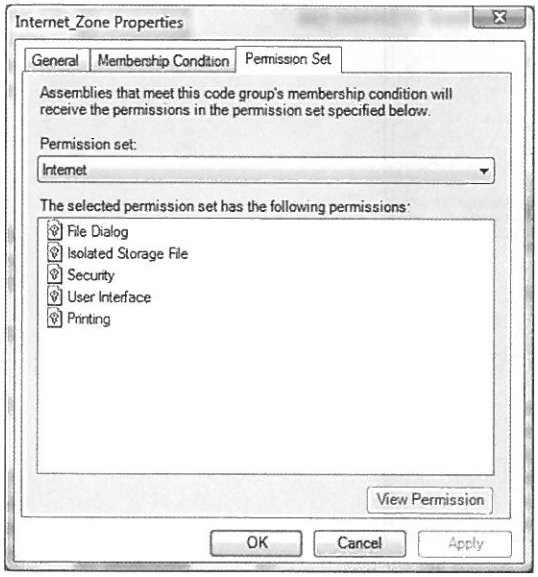


20.13. ábra: *A My_Computer_Zone kódcsoport tagsági feltételei*

Ha a My_Computer_Zone kódcsoport Permission Set fülére kattintunk, azt látjuk, hogy minden, a helyi merevlemezről betöltött szerelvény a Full Trust elnevezésű biztonsági engedélykészletet kapja meg (lásd a 20.14. ábrát).



20.14. ábra: *Az alapértelmezés szerint a helyi merevlemezről betöltött szerelvények Full Trust engedélyt kapnak*

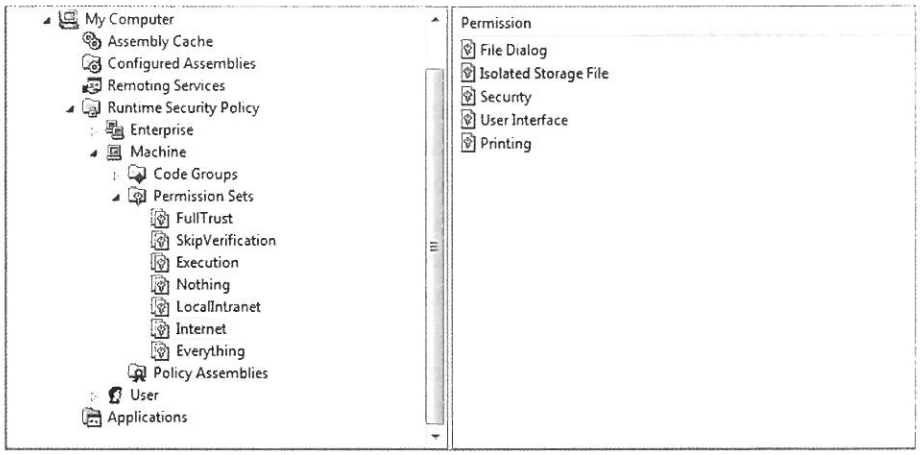


20.15. ábra: *A külső URL-ről betöltött szerelvények az alapértelmezés szerint nem kapják meg a Full Trust engedélykészletet*

Ezzel szemben a külső URL-ről (a céges intraneten kívülről) betöltött szerelvények az Internet_Zone kódcsoportba kerülnek, amely jóval szigorúbb jogosultságcsoportba tartozik. Ahogy a 20.15. ábrán is látható, az Internet_Zone csoportba sorolt szerelvények nem a Full Trust engedélyt kapják meg, hanem az Internet nevű engedélykészlettel fognak rendelkezni.

Az engedélykészletek szerepe

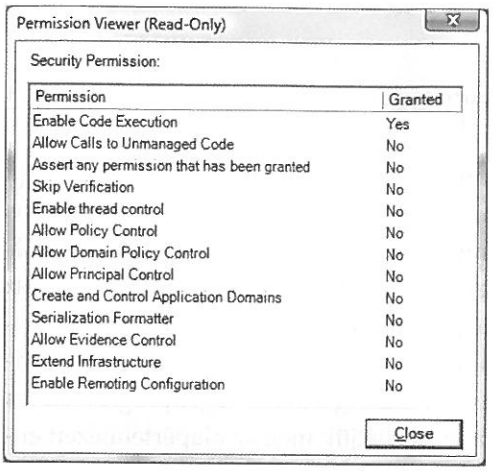
A CLR különböző feltételek alapján sorolja a szerelvényeket kódcsoportokba (a betöltés eredete, erős név stb. alapján). A kódcsoporthoz pedig egy barátságos névvel rendelkező (Full Trust, Internet stb.) engedélykészlet tartozik. Mindegyik engedélykészlet (mint azt a neve is mutatja) olyan egyedileg beállított engedélyek gyűjteménye, amelyek különböző biztonsági beállításokat vezérelnek (nyomtatóhoz vagy fájlrendszerhez való hozzáférés, reflexiós API használata stb.). A Microsoft .NET Framework Configuration segédprogramban a View Permissions hivatkozásra kattintva nézhetjük meg az alapértelmezett engedélykészletek beállításait. A 20.16. ábra az Internet-engedélykészlet beállításait mutatja a View Permissions hivatkozásra való kattintás után.



20.16. ábra: *Internet engedélykészlet*

Az itt látható ikonok (File Dialog, Isolated Storage File, Security stb.) mind a készlet egy-egy specifikus engedélyét képviselik, amelyeket még részletesebben beállíthatunk, ha duplán kattintunk rájuk. Ha például kétszer kattintunk a Security engedélyre (amely az általános biztonsági beállítások gyűjtőengedélye), azt láthatjuk, hogy az Internet_Zone alatt futó szerelvény (amelyet

ezért az Internet engedélykészlet korlátoz) elindítható, de az alapértelmezés szerint *nem* hajthat végre számos egyéb alapvető műveletet, nem használhatja például a platformhívási szolgáltatást arra, hogy meghívja az operációs rendszer API-ját (lásd a 20.17. ábrát).



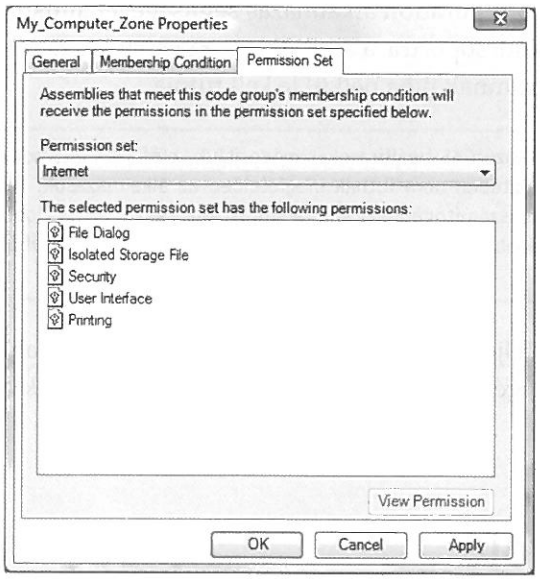
20.17. ábra: *Az engedélykészlet egyes engedélyeinek megtekintése*

A CAS működése

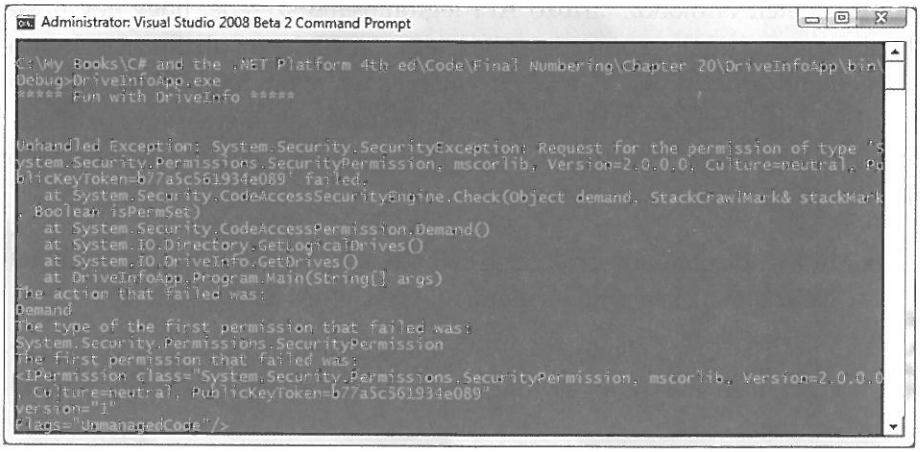
A CAS-műveletek bemutatása érdekében megváltoztatjuk a My_Computer kódcsoport néhány alapértelmezett beállítását, majd megnézzük az eredményt. A kapcsolódó tulajdonságlap megnyitásához kattintsunk jobb gombbal a Microsoft .NET Framework Configuration alkalmazásban a számítógép-házirendhez tartozó My_Computer_Zone kódcsoportra. Ezután kattintsunk a Permission Set fülre, majd módosítsuk az engedélykészletet Full Trustról Internetre (lásd a 20.18. ábrát).

Az OK gombra kattintva a program módosítja a helyi számítógép biztonsági házirendjét. Ez ebben az esetben azt jelenti, hogy a merevlemezről betöltött .NET végrehajtható fájlok többé nem fogják tudni elérni a helyi merevlemezt a `System.IO` névtér típusainak felhasználásával. Ellenőrzésképpen indítsuk el valamelyik korábban készített IO-intenzív alkalmazásunkat. Használjuk a parancssort arra, hogy a korábban készített `DriveInfoApp.exe` alkalmazást tartalmazó könyvtárra váltsunk, és próbáljuk meg elindítani a programot. Ha megjelenik a futásidejű hibát jelző párbeszédpanel, válasszuk a Close the Program lehetőséget, majd figyeljük meg a parancssori kimenetet.

A 20.19. ábrán látszik, hogy az alkalmazás nem tudott hozzáférni a helyi merevlemezhez, és a CLR biztonsági kivételt dobott.



20.18. ábra: *A My_Computer_Zone kódcsoport engedélykészletének módosítása*



20.19. ábra: *Biztonsági veszély. A My_Computer_Zone kódcsoport nem engedélyezi IO-műveletek végrehajtását*

Full Trust jogosultság visszaállítása a My_Computer_Zone kódcsoportra

A Microsoft .NET Framework Configuration alkalmazás segítségével állítsuk vissza a My_Computer_Zone kódcsoportra a Full Trust engedélykészletet. Ezután a `DriveInfoApp.exe` programnak hiba nélkül le kell futnia.

Megjegyzés Ha a számítógépre vonatkozó CAS-beállításokat módosítjuk, előfordulhat, hogy egy véletlen helytelen beállítás következtében nem futnak megfelelően az alkalmazások. Ha vissza kell állítani az alapértelmezett számítógépházirend-beállításokat, kattintsunk jobb gombbal a Runtime Security Policy mappában található Machine ikonra, majd a helyi menüből válasszuk a Reset parancsot.

A fenti rövid leírás nem adhat teljes áttekintést erről a rendkívül sokoldalú API-ról, de ez alapján jobban megérthetjük az elszigetelt tároló szerepének a fontosságát.

Az elszigetelt tároló

Az elszigetelt tárolóhoz tartozó API legfontosabb szerepe, hogy egy olyan biztonságos virtuális környezetet hoz létre, amelybe az alkalmazások attól függetlenül írhatnak vagy abból olvashatnak adatokat, hogy melyik kódcsoportban találhatók, anélkül, hogy az alapértelmezett biztonsági házirendet módosítani kellene. Ezenkívül ennek a technológiának a használata az alábbi körülmények között is célszerű lehet.

- El kell menteni az alkalmazás felhasználói beállításait vagy egyéb felhasználóhoz kapcsolódó adatokat (DataSet-ek, XML-fájlok stb.).
- Egy ClickOnce alkalmazást telepítünk, amely biztonságos futtatókörnyezetben működik, és nincs korlátlan hozzáférése a helyi fájlrendszerhez.
- Olyan Windows Forms-vezérlőelemet töltöttünk le az internetről, amely egy webalkalmazásba integrálódik, és az ügyfélgépen kell beállításokat tárolnia.
- XAML böngészőalkalmazást (XBAP, XAML Browser Application – lásd a 28. fejezetet) telepítünk, amely a felhasználó számítógépén tárol adatokat.

Az elszigetelt tároló használata nem jelenti azt, hogy ne tárolhatnánk adatot a `.config` fájlban (például kapcsolatsztringeket), vagy helyezhetnénk el különböző alkalmazásbeállításokat a rendszerleíró adatbázisban. Mint minden technológiának, az elszigetelt tárolónak is vannak hátrányai. Első és legfontosabb az, hogy az elszigetelt tárolóba helyezett adat nincs automatikusan titkosítva. Ezért csakúgy, mint a hagyományos IO-műveletek esetén, az érzékeny adatokat (pl. hitelkártya-információkat) manuálisan kell titkosítanunk (és visszafejtenünk).

Megjegyzés A .NET titkosító API-jának használata nem témája ennek a könyvnek. Erről részletes információval szolgál a .NET Framework 3.5 SDK dokumentációjának Cryptographic Services című része.

Csakúgy, mint a fájlrendszer más részein tárolt adatokat, a végfelhasználó az elszigetelt tárolóba helyezett fájlokat is átmásolhatja, áthelyezheti vagy letörölheti. A `System.IO.IsolatedStorage` névtér olyan, dinamikusan generált könyvtárstruktúrában tárolja az adatokat, amely el van rejtve a merevlemez egy adott helyén. Így a legtöbb felhasználó soha sem találkozik közvetlenül a számítógép elszigetelt tárolójával.

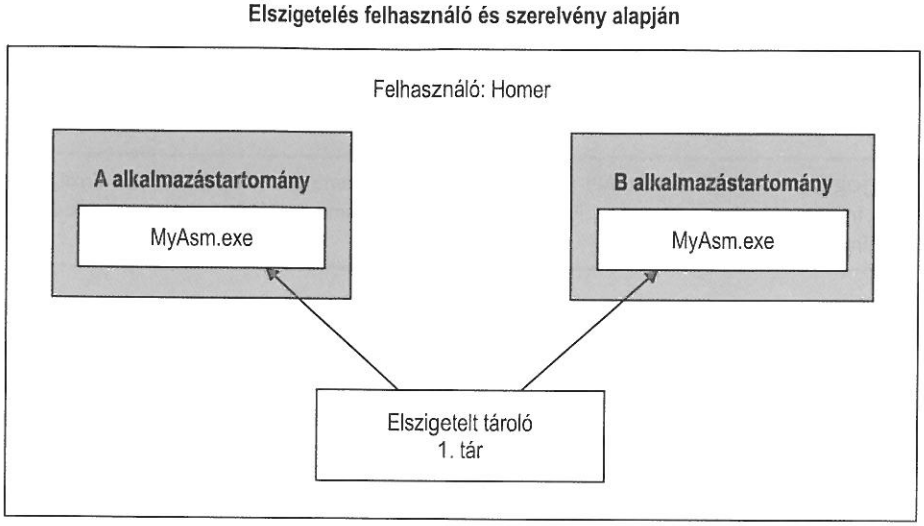
A rendszergazda korlátozhatja az elszigetelt tároló által elfoglalható lemezterület nagyságát. Ha az alkalmazásunk mellett más alkalmazások is nagy mennyiségű adatot tárolnak, lehetséges, hogy a tároló mérete eléri a megengedett felső korlátot. Tanácsos a strukturált kivételkezelés segítségével felkészülni erre az esetre.

Az elszigetelt tároló hatóköre

Természeténél fogva az elszigetelt tároló (legalább) felhasználónként elszigeteli egymástól az adatokat. Ezért, ha a programunk az elszigetelt tárolóba ment adatokat, azokat a .NET az aktuálisan bejelentkezett felhasználó szintjén tárolja. Ha ugyanarra a munkaállomásra egy másik felhasználó is bejelentkezik, és ugyanabból az alkalmazásból adatokat ment el, az eltárolt adatokat egy, az adott felhasználó számára egyedi helyen rögzíti.

A felhasználószintű elszigetelésen kívül az elszigetelt tároló szerelvény és/vagy alkalmazástartomány-azonosító szintjén is képes az adatokat izolálni. Ha az elszigetelést felhasználónként és szerelvényenként is engedélyezzük, az alkalmazás ugyanazt a tárolót fogja használni függetlenül attól, hogy a programot melyik alkalmazástartomány hosztolja.

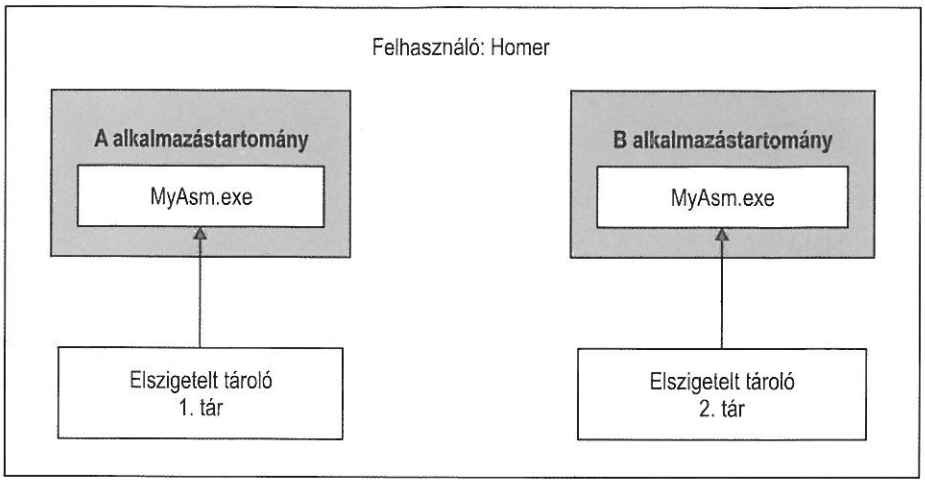
Ebben az esetben az elérhető legjellegzetesebb bizonyíték alapján (pl. az erős név) jön létre a tároló neve. Így elérhetjük, hogy ugyanazon a gépen egy időben a program több példánya fusson, amelyek mind ugyanazt a tárolót használják (lásd a 20.20. ábrát).



20.20. ábra: *Felhasználó- és szerelvényszintű izoláció*

Másrészről, ha felhasználó + szerelvény + alkalmazástartomány elkülönítési szintet állítunk be, akkor a .NET az alkalmazástartományt is figyelembe veszi. A legjellegzetesebb szerelvénybizonyítékokon kívül a legjellegzetesebb alkalmazástartomány-bizonyítékot (jellemzően a Site) is felhasználja a rendszer a tároló létrehozásához. Ekkor, még ha ugyanazt a szerelvényt is használjuk több egyedi alkalmazástartományból, az alkalmazás adatai külön tárolókba kerülnek (lásd a 20.21. ábrát).

Az elszigetelt tároló használatával vándorló (roaming) profilok is kialakíthatók. A vándorló hatókör lehetővé teszi, hogy bár a felhasználó különböző számítógépekre jelentkezik be, mégis ugyanazt az alkalmazásadatot kapja meg. Ilyen esetben az adattárolás hálózati helyen történik, és igény szerint letöltődik, ha a felhasználó bejelentkezik egy adott munkaállomásra. Ennek részleteiért olvassuk el a .NET Framework 3.5 SDK dokumentációjának ide vonatkozó részét.



20.21. ábra: *Felhasználó-, szerelvény- és alkalmazástartomány-szintű izoláció*

Az elszigetelt tároló helye

Az elszigetelt tároló nem más, mint a .NET-et futtató számítógép fájlrendszerének erre a célra szánt része, vagyis nem különbözik a C:\Windows, C:\Program Files vagy bármely más, a merevlemezen lévő könyvtártól. Mindazonáltal az elszigetelt tároló pontos helye az operációs rendszertől is függ. Egy adott tároló gyökerének a helye egy Vistát futtató számítógépen:

C:\Users\<*felhasználó*>\App Data\Local\IsolatedStorage

Egy Windows XP számítógépen viszont:

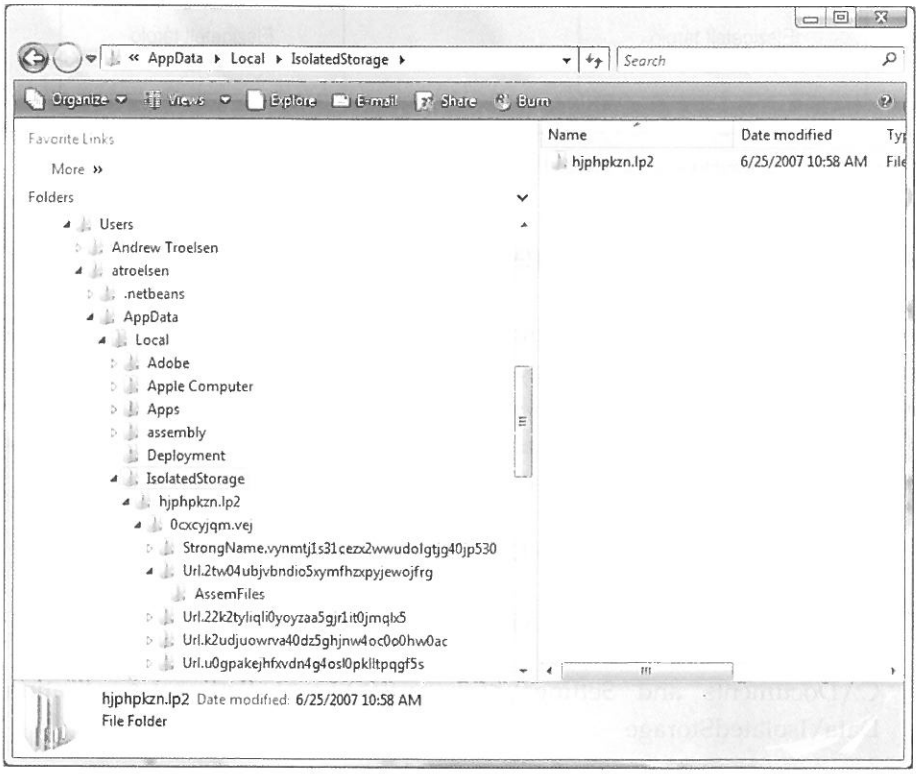
C:\Documents and Settings\<*felhasználó*>\Local Settings\Application Data\IsolatedStorage

Ez alatt található számtalan (teljesen kimondhatatlan nevű) alkönyvtár, amelyeket az elszigetelt tárolóhoz tartozó API hoz létre és tart karban. A 20.22. ábra a tároló helyét mutatja egy Vistát futtató számítógépen.

A könyvtárak neveivel vagy azzal, hogy ezeket programozottan megadjuk a tárolók létrehozása során, nem kell foglalkoznunk. Ezek a nevek automatikusan jönnek létre a felhasználó azonossága és a szükséges szerelvény és/vagy alkalmazástartomány-bizonyítékok alapján.

A bizonyítékok gyűjtése során a CLR az alábbi listát ellenőrzi (a legspecifikusabb bizonyítékkal kezdve), ebben a sorrendben:

- Kibocsátó tanúsítványa
- Erős név
- URL
- Hely
- Zóna



20.22. ábra: *Elszigetelt tároló egy Vistát futtató számítógépen*

Az elszigetelt tároló kezelése a storeadm.exe segítségével

A .NET Framework 3.5 SDK tartalmaz egy `storeadm.exe` nevű parancssori eszközt is, amellyel a számítógép aktuális tárolórendszerét kezelhetjük. Az eszköz segítségével megtekinthetjük az éppen bejelentkezett felhasználó tárolóit (a `/list` paraméterrel), és törölhetjük az aktuális felhasználó tárolóit (a `/remove` argumentummal). A 20.23. ábrán a `/list` kapcsoló kimenete látható.

Mint azt az előző ábra is mutatja, ez az eszköz megjeleníti az izoláció szintjét (szerelvény, alkalmazástartomány) és a tároló létrehozásához használt bizonyítékokat is. De *nem* mutatja meg az egyes tárolókban található fájlok tartalmát. Ehhez meg kell keresnünk a létrehozott tárolóhelyeket a Windows Intézővel, vagy programozottan kell beolvasnunk az adatokat.

20.23. ábra: *Az aktuális felhasználó tárolóinak megjelenítése a storeadm.exe programmal*

A System.IO.IsolatedStorage típusa

Mielőtt megnéznénk, hogy írhatunk (és olvashatunk) adatokat az elszigetelt tárolóba, tekintsük át a 20.14. táblázatot, amely az elszigetelt tároló alapvető típusait ismerteti. Ez a névtér meglehetősen kicsi, mivel típusait a `System.IO` alapvető típusaival együtt használjuk.

System.IO.IsolatedStorage típus	Jelentés
`IsolatedStorage`	Ez a típus az az absztrakt ősosztály, amelyből valamennyi elszigetelt tárolónak származnia kell.
`IsolatedStorageScope`	Ennek a felsorolt típusnak a segítségével állíthatjuk be a használandó izoláció szintjét (szerelvény, alkalmazástartomány vagy vándorló).
`IsolatedStorageException`	Ez a típus határozza meg, hogy milyen kivételt dobjon a program, ha egy művelet nem sikerül az elszigetelt tárolóban.
`IsolatedStorageFile`	Ez a tag jelöli azt a területet, ahol az elszigetelt tároló a fájlokat és könyvtárakat tartja.
`IsolatedStorageFileStream`	Ez a típus az elszigetelt tárolóban található fájlok olvasását, írását, létrehozását teszi lehetővé.

20.14. táblázat: *A System.IO.IsolatedStorage típusai*

Tároló létrehozása az IsolatedStorageFile objektummal

Ha az elszigetelt tárolóban szeretnénk alkalmazásadatot tárolni, első lépésként el kell döntenünk, hogy milyen szintű izolációt szeretnénk használni. A .NET a tárolókat legalább felhasználónként mindenképp elkülöníti, de beállíthatunk szerelvényenkénti vagy alkalmazástartományonkénti izolációt is (valamint vándorló profilt is). A megfelelő elkülönítési szint beállításakor egyik lehetőségünk, hogy az `IsolatedStorageScope` felsorolás egy tagjával beállítjuk az értékeket, majd meghívjuk az `IsolatedStorageFile.GetStore()` metódust. Az `IsolatedStorageFile` típus statikus `GetUserStoreForDomain()` vagy `GetUserStoreForAssembly()` metódusai is használhatók. Vizsgáljuk meg a következő példákat:

```
static void GetAppDomainStorageForUser()
{
  // Elszigetelt tároló megnyitása szerelvény- és
  // alkalmazástartomány-azonosító alapján (rövid).
  IsolatedStorageFile store =
          IsolatedStorageFile.GetUserStoreForDomain();
```

```
    // Vagy egyesítjük a jelzőket, és meghívjuk a GetStore() metódust.
    IsolatedStorageFile store2 =
            IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Domain, null, null);
}

static void GetAssemblyStorageForUser()
{
    // Megnyitunk egy tárolót
    // szerelvényazonosító alapján (rövid).
    IsolatedStorageFile store2 =
            IsolatedStorageFile.GetUserStoreForAssembly();

    // Vagy egyesítjük a jelzőket, és meghívjuk a GetStore() metódust.
    IsolatedStorageFile store2 =
            IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Assembly, null, null);
}
```

Bármelyik módszert is választjuk, az eredményként egy `IsolatedStorageFile` objektumot kapunk. Ennek a típusnak a segítségével a tárolóba adatokat írhatunk, abból olvashatunk, vagy létrehozhatunk tetszőleges egyedi könyvtárszerkezetet az aktuális felhasználó tárolóján belül. A 20.15. táblázat az `IsolatedStorageFile` néhány fontosabb tagját mutatja be.

Tag	Jelentés
`CurrentSize`, `MaximumSize`	Ezek az írásvédett tulajdonságok visszaadják az elszigetelt tároló méretével kapcsolatos adatokat.
`Scope`	Megadja az izoláció hatókörét (felhasználó, szerelvény vagy alkalmazástartomány).
`CreateDirectory()`	Létrehoz egy új könyvtárat a tárolóban.
`DeleteDirectory()`	Töröl egy könyvtárat a tárolóból.
`DeleteFile()`	Töröl egy fájlt a megadott könyvtárból.
`GetDirectoryNames()`	Ez a metódus lehetővé teszi, hogy végigiteráljunk a nevesített könyvtárakon.
`GetEnumerator()`	Egy hatókör-specifikus `IEnumerator` típussal tér vissza.
`GetFiles()`	Visszaadja a tárolóban található fájlok listáját.
`GetStore()`	Ez a túlterhelt metódus a megadott alkalmazástartomány- és szerelvénybizonyítékoknak és az izolációs hatókörnek megfelelő elszigetelt tárolót adja vissza.

Tag	Jelentés
`GetUserStoreForAssembly()`	Ez a metódus megszerzi a hívó szerelvényazonosítójának megfelelő elszigetelt tárolót.
`GetUserStoreForDomain()`	Ez a metódus megszerzi az alkalmazástartomány- és szerelvényazonosítónak megfelelő elszigetelt tárolót.
`Remove()`	Ez a metódus eltávolítja a tárolókat.

20.15. táblázat: *Az IsolatedStorageFile tagjai*

Adat írása a tárolóba

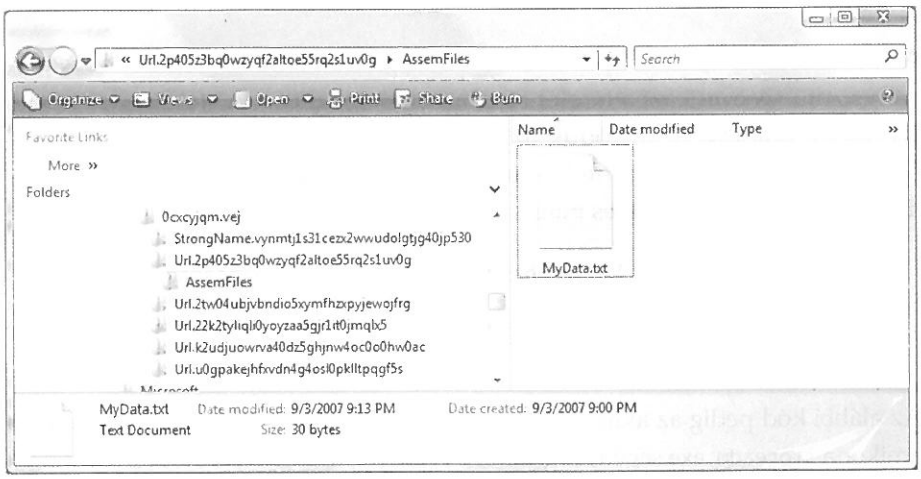
Miután létrehoztuk a tárolót, következő lépésként létre kell hoznunk az `IsolatedStorageFileStream` típus egy példányát, amely az adattároláshoz használt fájlt jelenti a tárolóban. A többi IO-adatfolyamhoz hasonlóan ez a típus is beállítható a korábban ismertetett `System.IO.FileMode` felsorolással. Ennek bemutatásához hozzunk létre egy konzolalkalmazást `SimpleIsoStorage` néven, amelybe importáljuk a `System.IO` és a `System.IO.IsolatedStorage` névtereket. Ezután módosítsuk úgy a `Main()` metódust, hogy a `Program` osztály alábbi segédmetódusát hívja meg:

```
static void WriteTextToIsoStorage()
{
  // Elszigetelt tároló megnyitása szerelvény- és
  // felhasználói bizonyítékok alapján.
  using (IsolatedStorageFile store =
                IsolatedStorageFile.GetUserStoreForAssembly())
  {
    // Létrehozzuk az IsolatedStorageFileStream típust.
    using (IsolatedStorageFileStream isStream =
                  new IsolatedStorageFileStream("MyData.txt",
                              FileMode.OpenOrCreate, store))
    {
      // Az adatfolyamot "becsomagoljuk" egy StreamWriter
      // objektumba, és kiírunk valamilyen szöveget.
      using (StreamWriter sw = new StreamWriter(isStream))
      {
        sw.WriteLine("This is my data.");
        sw.WriteLine("Cool, huh?");
      }
    }
  }
}
```

Először megszerzünk egy tárolót a felhasználó számára a végrehajtó szerelvény (`SimpleIsoStorage.exe`) azonossága alapján az `IsolatedStorageFile.GetUserStoreForAssembly()` metódussal. Ezután létrehozunk egy új `IsolatedStorageFileStream` objektumot, és megadjuk, hogy az újonnan létrehozandó fájl neve `MyData.txt` legyen a most megszerzett tárolóban (ha a fájl már létezik, akkor megnyitjuk). Végül az `IsolatedStorageFileStream` objektumot egy `System.IO.StreamWriter` íróba csomagoljuk, és kiírunk néhány sornyi szöveget.

Ha lefuttatjuk az alkalmazást, belenézhetünk a számítógépen lévő elszigetelt tárolóba, és (némi kutakodás után) megtalálhatjuk a `MyData.txt` fájlt (lásd a 20.24. ábrát). A `MyData.txt` fájlban, ha jegyzettömbbel megnyitjuk, az abba kiírt két sort látjuk.

Természetesen az `IsolatedStorageFileStream` objektumot más, a `Stream` osztályból származó típusba is becsomagolhatjuk. Ha például inkább bináris formában szeretnénk kiírni az adatokat, használhatjuk a `StreamWriter` helyett a `BinaryWriter` osztályt.



20.24. ábra: *Az elszigetelt tárolóba helyezett szövegfájl*

Adat olvasása a tárolóból

A felhasználó tárolójában elhelyezett adat olvasása is hasonlóan egyszerű. Nézzük meg a következő metódust (amelyet célszerű szintén a `Main()` függvényből meghívni a `WriteTextToIsoStorage()` metódus után), amely a `MyData.txt` fájlban található információt kiolvassa, és megjeleníti a konzolon:

```
private static void ReadTextFromIsoStorage()
{
  using (IsolatedStorageFile store =
                IsolatedStorageFile.GetUserStoreForAssembly())
  {
    using (IsolatedStorageFileStream isStream =
                  new IsolatedStorageFileStream("MyData.txt",
                      FileMode.Open, FileAccess.Read, store))
    {
      // "Becsomagoljuk" StreamReaderbe.
      using (StreamReader sr = new StreamReader(isStream))
      {
        string allTheData = sr.ReadToEnd();
        Console.WriteLine(allTheData);
      }
    }
  }
}
```

Felhasználói adat törlése a tárolóból

Az `IsolatedStorageFile` típus két módszert biztosít a tárolók törlésére. A példányszintű `Remove()` azt a tárolót törli, amelyik meghívja. A statikus `IsolatedStorageFile.Remove()` metódus az `IsolatedStorageScope.User` értéket veszi fel, és törli a kódot futtató felhasználó összes tárolóját. A következő kód például törli az aktív tárolót, és megsemmisíti annak tartalmát:

```
// Töröljük az aktuális felhasználó aktuális tárolójának adatait.
IsolatedStorageFile store =
                    IsolatedStorageFile.GetUserStoreForAssembly();
store.Remove();
```

Az alábbi kód pedig az aktuális felhasználó *összes tárolóját* törli (ugyanúgy, mint amikor a `storeadm.exe` segédprogramot a `/remove` paraméterrel indítjuk):

```
// Töröljük a felhasználó ÖSSZES tárolóját.
IsolatedStorageFile.Remove(IsolatedStorageScope.User);
```

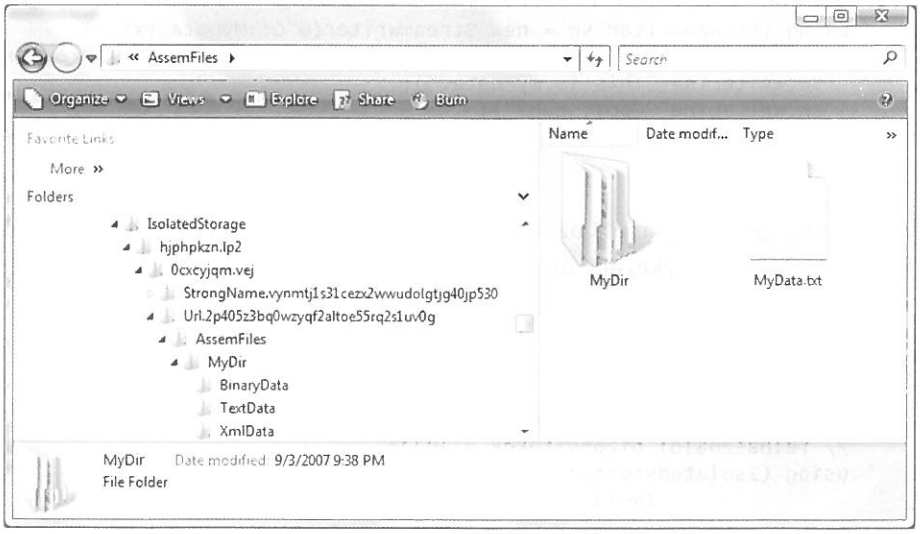
Egyedi könyvtárstruktúra létrehozása

Az előző példákban nem hoztunk létre egyedi hierarchiát a különböző adatfájlok tárolására. Ehelyett a `MyData.txt` fájlt közvetlenül a tároló gyökerében helyeztük el (a legtöbb esetben pontosan erre is van szükség). Egyedi könyvtárakat a példányszintű `CreateDirectory()` metódussal hozhatunk létre. Az

elszigetelt tárolóban az alkönyvtárakat nem képezhetjük le objektumokra, hanem olyan sztringeket kell átadnunk, amelyek a létrehozni kívánt könyvtárakat jelzik. Ha ez megvan, a `CreateDirectory()` metódus egy `IsolatedStorageFile` típussal tér vissza, amely a legbelső könyvtárhoz nyújt hozzáférését. Nézzük meg a következő kódot:

```
private static void CreateStorageDirectories()
{
  // Hagyományos és fordított perjelet is használhatunk.
  using (IsolatedStorageFile store =
                IsolatedStorageFile.GetUserStoreForAssembly())
  {
    store.CreateDirectory(@"MyDir\XmlData");
    store.CreateDirectory("MyDir\\BinaryData");
    store.CreateDirectory("MyDir/TextData");
  }
}
```

Ha a fenti metódust meghívjuk a `Main()` függvényből, megtalálhatjuk a létrehozott könyvtárstruktúrát az elszigetelt tároló területén (lásd a 20.25. ábrát).



20.25. ábra: *Könyvtárstruktúra létrehozása a megadott tárolóban*

Forráskód A SimpleIsoStorage projektet a forráskódkönyvtárban a 20. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Az elszigetelt tároló működés közben: ClickOnce-telepítés

Meglehet, hogy az elszigetelt tároló ez idáig pusztán a felhasználószintű adattárolás egyedi módjának tűnt (ez önmagában is hasznos dolog). Ám az egyik probléma, amit ez az API orvosol, az az, hogy a nem Full Trust jogosultsággal futó alkalmazások is biztonságosan tárolhassanak adatokat.

Tegyük fel, hogy van egy Windows Forms-alkalmazásunk (neve: `FileOrIsoStorageWinApp`), amely egy két gombot tartalmazó űrlapból áll. (A Windows Forms API-ról részletesen a 27. fejezetben lesz szó, de az egyes `Button` típusok kattintási eseményét már itt is le fogjuk kezelni.)

Az egyik gomb a helyi merevlemezre próbál adatot menteni hagyományos IO-megoldásokkal (ne felejtsük el importálni a `System.IO` és a `System.IO.IsolatedStorage` névtereket a kódfájlunkba):

```
private void btnFileIO_Click(object sender, EventArgs e)
{
  using (StreamWriter sw = new StreamWriter(@"C:\MyData.txt"))
  {
    sw.WriteLine("This is my data.");
    sw.WriteLine("Cool, huh?");
  }
}
```

A második gomb ugyanazt az adatot egy elszigetelt tárolóban lévő fájlba írja ki. A `Click` eseménykezelő megvalósítása a korábbi projektben már megírt `WriteTextToIsoStorage()` metódushoz hasonlít:

```
private void btnIsoStorage_Click(object sender, EventArgs e)
{
  // Elszigetelt tároló megnyitása szerelvény- és
  // felhasználói bizonyítékok alapján.
  using (IsolatedStorageFile store =
                IsolatedStorageFile.GetUserStoreForAssembly())
  {
    // Létrehozzuk az IsolatedStorageFileStream típust.
    using (IsolatedStorageFileStream isStream =
                  new IsolatedStorageFileStream("MyData.txt",
                              FileMode.OpenOrCreate, store))
    {
      // Az adatfolyamot "becsomagoljuk" egy StreamWriter
      // objektumba, és kiírunk valamilyen szöveget.
```

```
            using (StreamWriter sw = new StreamWriter(isStream))
            {
              sw.WriteLine("This is my data.");
              sw.WriteLine("Cool, huh?");
            }
          }
        }
      }
```

Az IsolatedStorageFilePermission attribútum

Mielőtt kipróbálnánk az alkalmazást, adjuk hozzá az alábbi using direktívát a kezdeti `Form`-leszármazott osztályt definiáló C#-fájlhoz:

```
using System.Security.Permissions;
```

Ebben a névtérben számos olyan biztonsággal kapcsolatos, az alkalmazáson használható attribútum található, amelyek segítségével értesíthetjük a biztonsági alrendszert arról, hogy az adott szerelvény megfelelő működéséhez milyen biztonsági beállításokra van szükség (az egyéb részletek mellett). A példában arról értesítjük a CLR-t, hogy az alkalmazásunknak legalább szerelvényszintű, elszigetelt tárolási izolációs jogosultságra van szüksége. Ezt úgy érhetjük el, ha a következő szerelvényszintű attribútumot hozzáadjuk a `Form`-leszármazott típusunk kódfájljához:

```
[assembly:
 IsolatedStorageFilePermission(SecurityAction.RequestMinimum,
 UsageAllowed = IsolatedStorageContainment.AssemblyIsolationByUser)]
```

Ha a programot közvetlenül Visual Studio 2008-ban (a Ctrl + F5 billentyűkombinációval) fordítjuk le, és futtatjuk, azt tapasztaljuk, hogy bármelyik gombra is kattintunk, létrejön a megfelelő szöveges adatot tartalmazó új fájl. Ennek oka az, hogy az alkalmazást a My_Computer_Zone kódcsoportból töltöttük be, amely a Full Trust jogosultságot biztosítja a szerelvény számára.

A biztonsági zóna korlátozása

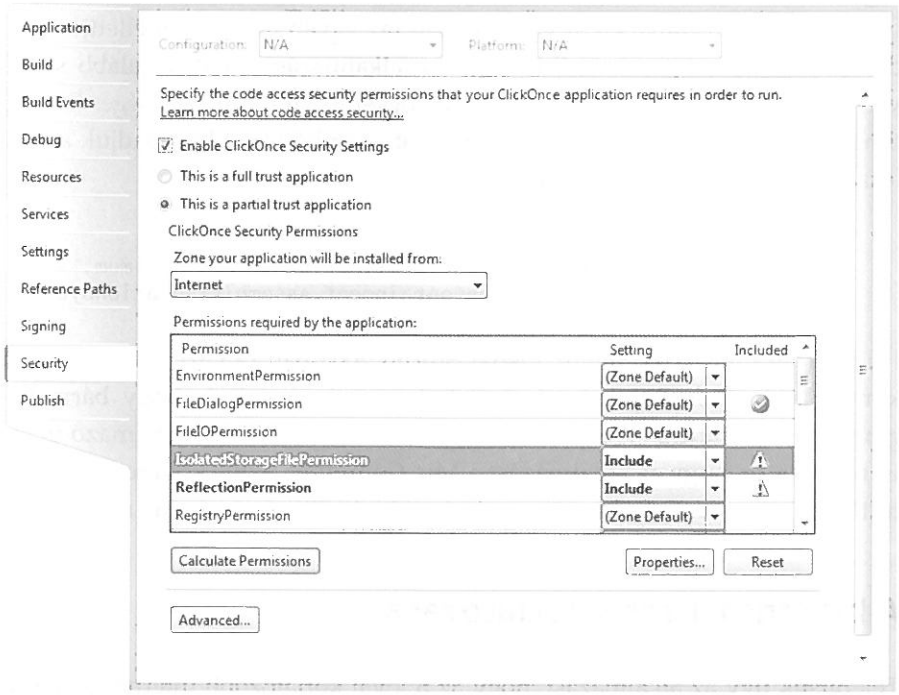
Telepítsük úgy az alkalmazást, hogy az a jóval korlátozóbb Internet_Zone engedélykészlettel fusson. Ennek érdekében az alkalmazásunkat a ClickOnce-szal telepítjük. A ClickOnce segítségével telepíthetünk távoli webkiszolgálóról futtatható alkalmazást a végfelhasználó számítógépére.

A távoli alkalmazás egy IIS virtuális könyvtárban található, ahonnan egyszerűen az URL-böngészőben történő megadásával letölthetjük, és telepíthetjük a helyi számítógépre.

Megjegyzés A ClickOnce teljes ismertetése túlmutat ennek a fejezetnek a keretein. Ha valaki még nem telepített ily módon alkalmazást, csak kövesse az alábbi utasításokat (és igény szerint olvassa el a .NET Framework 3.5 SDK dokumentációját).

Először nyissuk meg a projekt Properties lapját úgy, hogy a Solution Explorerben duplán kattintunk a Properties ikonra. Majd kattintsunk a Security fülre. Az alapértelmezés szerint a ClickOnce alkalmazások Full Trust engedéllyel települnek, ezért ugyanolyan biztonsági jogosultságokkal rendelkeznek, mint a hagyományosan telepített programok.

Olyan telepítőszkriptet kell írnunk, amely arra kényszeríti a programot, hogy az Internet zónában fusson, amely nem enged hozzáférést a merevlemezhez hagyományos IO-műveletek felhasználásával.



20.26. ábra: *Az alkalmazás biztonsági zónájának korlátozása*

Ehhez kattintsunk az Enable ClickOnce Security Settings jelölőnégyzetre, válasszuk a This is a partial trust application rádiógombot, majd válasszuk az Internet lehetőséget a zónákat felsoroló legördülő listamezőből. Végül, de nem utolsósorban kattintsunk a Calculate Permissions gombra a biztonsági beállításokat tartalmazó oldal alján. Ez meghatározza az alkalmazás által igényelt végső engedélykészletet (lásd a 20.26. ábrát).

Az alkalmazás közzététele webkiszolgálón

Kattintsunk a tulajdonságszerkesztőben a Publish fülre. Majd kattintsunk az oldal alsó részében található Publish Wizard gombra, és a varázsló első lapján írjuk be az alábbi URL-t, hogy létrehozzuk az alkalmazást tároló új IIS virtuális könyvtárat a helyi gépen:

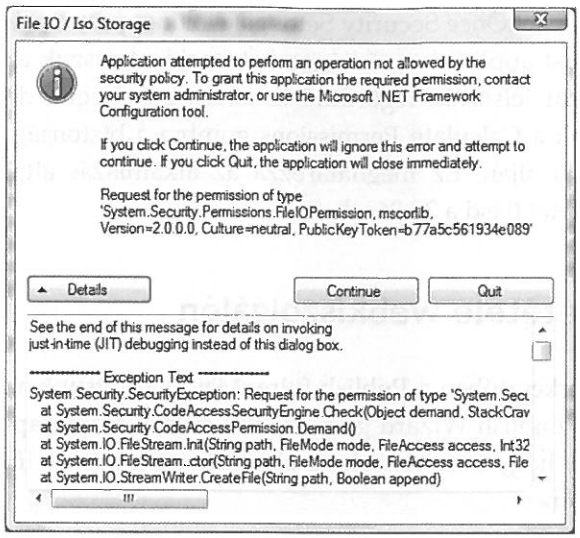
```
http://localhost/MyIsolatedStorageApp/
```

Az eszköz további beállításait alapértelmezett értéken hagyhatjuk. Mikor a Finish gombra kattintunk, a webböngésző betölti az automatikusan generált `publish.htm` fájlt. Ezt fogják látni a végfelhasználók is, amikor a letöltendő és a helyileg telepítendő alkalmazást tároló távoli webkiszolgálóra csatlakoznak (a weboldalt kedvünk szerint testre szabhatjuk; a fájlt és a kapcsolódó tartalmakat a projekt bin\Debug mappájában találjuk). Kattintsunk az Install gombra, majd futtassuk a `setup.exe` alkalmazást (és fogadjunk el minden biztonsági kérdést). Kis idő múlva az alkalmazás feltelepül a számítógépre, a ClickOnce gyorsítótárnak nevezett területre, és a program elindul.

Az eredmény megtekintése

Mivel az alkalmazást úgy állítottuk be, hogy korlátozott környezetben fusson, ha arra a gombra kattintunk, amely a `System.IO` típusok használatával próbál adatot menteni, akkor a 20.27. ábrán lévő biztonsági kivétel fog megjelenni.

Ellenben, ha arra a gombra kattintunk, amely az elszigetelt tároló segítségével ment adatot, az alkalmazás sikeresen lefut.



20.27. ábra: *Biztonsági veszély. Nem lehet hozzáférni a helyi fájlrendszerhez az Internet zónából*

Forráskód A FileOrIsoStorageWinApp projektet a forráskódkönyvtárban a 20. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Összefoglalás

A fejezet elején megismertük a `Directory(Info)` és `File(Info)` típusok használatát. Megtudtuk, hogy ezek az osztályok lehetővé teszi a merevlemezen található fizikai fájlok és könyvtárak kezelését. Ezután megismerkedtünk több, az absztrakt `Stream` osztályból származó típussal, amelyek közül a `FileStream` típussal bővebben foglalkoztunk. Mivel a `Stream`-leszármazott típusok nyers byte-folyamokkal végeznek műveleteket, a `System.IO` névtér számos olyan olvasó/író típust (`StreamWriter`, `StringWriter`, `BinaryWriter` stb.) rendelkezésünkre bocsát, amelyek megkönnyítik ezt a folyamatot. Ezek során a `DriveType` által nyújtott funkcionalitást is megnéztük, és megvizsgáltuk, hogyan figyelhetünk meg fájlokat a `FileSystemWatcher` segítségével, és hogyan kezelhetjük az adatfolyamokat aszinkron módon.

A fejezet második része az elszigetelt tárolót mutatta be. Ez az API lehetővé teszi a programok számára, hogy egy biztonságos virtuális környezetben végezzenek olvasási és írási műveleteket még akkor is, ha az alkalmazást korlátozott biztonsági környezetben töltötték be. Bár a programozási modell

meglehetősen egyértelmű (ha megértjük az alapvető fájlműveleteket), a kapcsolódó témakörök valamelyest megbonyolítják. Ennek fényében röviden áttekintettük a kódhozzáférés-szabályzást is.

Ebben megtudtuk, hogy a szerelvények alkalmazástartományba való betöltésük során a CLR részére bizonyítékokat szolgáltatnak. Ez alapján kerülnek bele az alapértelmezett engedélykészletekkel rendelkező kódcsoportokba. A CAS fájlműveletek szempontjából fontos tulajdonsága, hogy ha az alkalmazás nem kap Full Trust jogosultságot, a hagyományos IO-műveletek biztonsági kivételt eredményeznek. Ezzel szemben az elszigetelt tároló segítségével a programok felhasználónként elszigetelten és biztonságosan tárolhatnak adatokat.

HUSZONEGYEDIK FEJEZET

Bevezetés az objektum-sorosítás világába

Az előző fejezetben megismerkedtünk a `System.IO` névtér funkcionalitásával és az elszigetelt tároló (`System.IO.IsolatedStorage`) szerepével.

Ezek a névterek lehetővé teszik, hogy az adatokat számos olvasó és író használatával egy adott helyre (és egy adott formátumban) mentsük. A következőkben az ehhez a témához kapcsolódó *objektumsorosítás* kérdéskörét tárgyaljuk. Az objektumsorosítás segítségével elmenthetjük és visszaállíthatjuk az objektum állapotát bármely `System.IO.Streamből` származó típusból vagy típusba (beleértve az `IsolatedStorageFileStream` típust is).

A típusok sorosítása akkor rendkívül fontos, amikor egy objektumot különböző remoting technológiákkal, például a .NET remoting réteggel, az XML-webszolgáltatással vagy a Windows Communication Foundation segítségével távoli gépre szeretnénk másolni. A sorosítás önmagában is nagyon hasznos, és minden bizonnyal nagy szerepet fog játszani jó néhány .NET-alkalmazásunkban (akár elosztott, akár nem). A továbbiakban megismerkedünk a .NET sorosítási séma számos aspektusával, köztük olyan attribútumokkal (és interfészekkel), amelyek lehetővé teszik a folyamat testreszabását.

Az objektumsorosítás

A *sorosítás* kifejezés az objektumok folyamba (fájlfolyamba, memóriafolyamba stb.) történő mentését (és lehetőség szerinti átvitelét) jelenti. Az elmentett adatsorozatok minden szükséges információt tartalmaznak az objektumok állapotának visszaállításához (vagy *deszerializációjához*) a későbbi használatra. Egyértelmű választás, ha nagy mennyiségű (különböző formátumú) adatot szeretnénk elmenteni a lehető legkisebb ráfordítással. Ha a sorosítószolgáltatást használjuk, akkor az alkalmazásadatok mentése sok esetben kevesebb kódot eredményez, mint ha a `System.IO` névtér olvasóinak/íróinak használatával tennénk.

Tételezzük fel például, hogy egy grafikus asztali alkalmazást hoztunk létre, és lehetővé szeretnénk tenni, hogy a felhasználók elmenthessék a beállításaikat (ablak színe, betűméret stb.). Ehhez definiálhatunk egy `UserPrefs` nevű osztályt, amely körülbelül 20 adatmezőt foglal magába. Ha a `System.IO.BinaryWriter` típust használnánk, a `UserPrefs` objektum minden egyes mezőjét *manuálisan* kellene elmenteni. Ugyanígy, ha az adatokat a fájlokból vissza akarnánk tölteni a memóriába, szükség lenne a `System.IO.BinaryReaderre`, és az új `UserPrefs` objektum újrakonfigurálásához (ismét csak) *manuálisan* kellene beolvasni minden egyes értéket.

Bár a fenti megoldás teljesen működőképes, rengeteg időt spórolhatunk meg saját magunknak, ha a `UserPrefs` osztályt egyszerűen a `[Serializable]` attribútummal jelöljük meg:

```
[Serializable]
public class UserPrefs
{
  // Különböző adatok...
}
```

Így az objektum teljes állapotát csupán néhány soros forráskóddal is elmenthetjük. Nézzük meg az alábbi `Main()` metódust:

```
static void Main(string[] args)
{
  // Tételezzük fel, hogy a UserPrefs az alábbi tulajdonságokat
  // határozza meg.
  UserPrefs userData= new UserPrefs();
  userData.WindowColor = "Yellow";
  userData.FontSize = "50";

  // A BinaryFormatter az állapotadatokat bináris formátumban menti.
  BinaryFormatter binFormat = new BinaryFormatter();

  // Az objektumot egy helyi fájlban tárolja.
  using(Stream fStream = new FileStream("user.dat",
  FileMode.Create, FileAccess.Write, FileShare.None))
  {
    binFormat.Serialize(fStream, userData);
  }
  Console.ReadLine();
}
```

A .NET-objektumsorosítással az objektum állapotának mentése meglehetősen egyszerűnek mondható, ám a háttérben zajló folyamatok eléggé összetettek. Ha egy objektumot például egy folyamba mentünk, minden társított adat (ősosztályadatok, tárolt objektumok stb.) sorosítása is automatikusan megtör-

ténik. Ezért ha származtatott osztályt szeretnénk elmenteni, minden, a származási láncban lévő adat tárolódik. Az egymással kapcsolatban levő objektumok egy objektumgráffal ábrázolhatók.

A .NET-sorosítószolgáltatás az objektumgráfok különböző formátumú mentését is lehetővé teszi. Az előző példakód a `BinaryFormatter` típust alkalmazta, tehát a `UserPrefs` objektum állapotát tömörített bináris formátumban mentette. Az objektumgráfokat más típusokat használó SOAP- vagy XML-formátumban is elmenthetjük. Ezek a formátumok meglehetősen hasznosnak bizonyulnak, ha biztosítani akarjuk, hogy az elmentett objektumokat zavartalanul lehessen használni a különböző operációs rendszerekben, nyelvekben és architektúrákban.

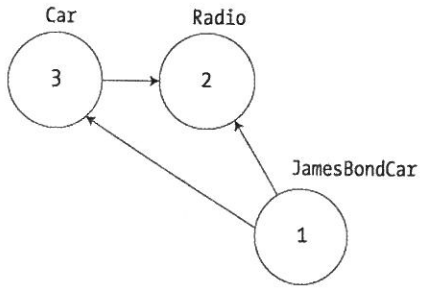
Az objektumgráfokat *bármilyen* `System.IO.Streamból` származó típusba elmenthetjük. Az előző példában a `UserPrefs` objektumot a `FileStream` típus segítségével mentettük a helyi fájlba. Ha azonban az objektumokat inkább egy meghatározott memóriaterületen szeretnénk tárolni, használhatjuk a `MemoryStream` típust is. A lényeg az, hogy az adatsorozat megfelelően ábrázolja a gráfon található objektumok állapotát.

Az objektumgráfok szerepe

Ha egy objektumot sorosítunk, a CLR valamennyi kapcsolódó objektumot számításba veszi annak biztosítására, hogy az adatok elmentése megfelelően történjen meg. A kapcsolódó objektumoknak ezt a halmazát *objektumgráfnak* nevezzük. Az objektumgráfok egyszerűen azt dokumentálják, hogy az objektumok miként kapcsolódnak egymáshoz. Nem feltétlenül képezik le a klasszikus OO kapcsolatokat (mint például az „az egy" vagy a „van neki egy" kapcsolatokat) annak ellenére, hogy ezt a paradigmát is meglehetősen jól modellezik.

Az objektumgráfokban található objektumok egyéni numerikus értékkel rendelkeznek. Az objektumgráf tagjaihoz rendelt számok tetszőlegesek, és semmilyen kapcsolatban nem állnak a külvilággal. Ha minden egyes objektumhoz hozzárendelődik egy numerikus érték, az objektumgráf minden egyes objektum függőséghalmazát képes elmenteni.

Ennek egyszerű szemléltetésére tételezzük fel, hogy olyan osztálykészletet hozunk létre, amely néhány gépkocsit modellez. Van tehát egy `Car` nevű ősosztály, amelynek „van egy" `Radio` osztálya. Egy másik, `JamesBondCar` nevű osztály kibővíti a `Car` alaptípust. A 21.1. ábra ezeket a lehetséges kapcsolatokat modellező objektumgráfot ábrázolja.



21.1. ábra: *Egyszerű objektumgráf*

Az objektumgráfok olvasásakor a nyilak összekapcsolásában használhatjuk a „függőség" vagy a „hivatkozás" kifejezéseket. Tehát a 21.1 ábrán azt látjuk, hogy a `Car` osztály hivatkozik a `Radio` osztályra (tekintettel a „van neki egy" kapcsolatra). A `JamesBondCar` hivatkozik a `Car` osztályra (mivel adott az „az egy" kapcsolat), valamint a `Radio` osztályra (mivel az örökli ezt a védett tagváltozót).

A CLR természetesen nem rajzol képeket a memóriába a kapcsolódó objektumgráfok ábrázolásához. Az előző diagramban dokumentált kapcsolatot inkább egy, az alábbihoz hasonló matematikai formula segítségével ábrázolja:

```
[Car 3, ref 2], [Radio 2], [JamesBondCar 1, ref 3, ref 2]
```

A formula elemzésekor ismét láthatjuk, hogy a 3-as objektum (a `Car`) függőségi kapcsolatban áll a 2-es objektummal (a `Radio`). A 2-es objektum, a `Radio`, „magányos farkas", nincs szüksége senkire. Végül pedig a 1-es objektum (a `JamesBondCar`) függőségi viszonyban áll mind a 3-as, mind a 2-es objektummal. Mindenesetre ha sorosítjuk vagy visszaállítjuk a `JamesBondCar` osztály egy példányát, az objektumgráf biztosítja, hogy a `Radio` és a `Car` típusok is részt vegyenek a folyamatban.

A sorosítási folyamatban az objektumok közötti kapcsolatokat ábrázoló gráf automatikusan, a háttérben jön létre. A későbbiekben is látni fogjuk, hogy lehetőség van arra, hogy irányítsuk az objektumgráf létrejöttét a sorosítási folyamat attribútumain és interfészeken keresztüli testreszabással.

Megjegyzés Az `XmlSerializer` típus (lásd később) nem az objektumgráfokkal menti el az állapotot, hanem továbbra is prediktív módon sorosítja és állítja vissza a kapcsolódó objektumokat.

Objektumok konfigurálása sorosításhoz

Annak érdekében, hogy egy adott objektum a .NET-sorosítószolgáltatások számára elérhető legyen, nem kell mást tennünk, mint minden egyes kapcsolódó osztályt (vagy struktúrát) fel kell ruháznunk a `[Serializable]` attribútummal. Ha egy adott típus olyan adattagokkal rendelkezik, amelyeknek nem kellene szerepelnie (vagy nem szerepelhetnek) a sorosítási sémában, jelöljük meg ezeket a mezőket a `[NonSerialized]` attribútummal. Ez akkor lehet hasznos, ha olyan tagváltozók vannak egy sorosítható osztályban, amelyekre nem kell „emlékezni" (pl. fix értékek, véletlenszerű értékek, átmeneti adatok stb.), és szeretnénk a mentett adatok méretét csökkenteni.

Sorosítható típusok meghatározása

Kiindulásként hozzunk létre egy új konzolalkalmazást SimpleSerialize névvel. Szúrjunk be egy új osztályt `Radio` névvel, amelyet a `[Serializable]` attribútummal jelöltünk, kivéve egy tagváltozót (`radioID`), amelyet pedig a `[NonSerialized]` attribútummal jelöltünk, ezért ez nem mentődik el a meghatározott adatfolyamban:

```
[Serializable]
public class Radio
{
  public bool hasTweeters;
  public bool hasSubWoofers;
  public double[] stationPresets;

  [NonSerialized]
  public string radioID = "XF-552RR6";
}
```

Ezt követően adjunk hozzá két osztályt (`JamesBondCar` és a `Car`), lássuk el őket a `[Serializable]` attribútummal, és definiáljuk az alábbi tagokat:

```
[Serializable]
public class Car
{
  public Radio theRadio = new Radio();
  public bool isHatchBack;
}
```

```
[Serializable]
public class JamesBondCar : Car
{
  public bool canFly;
  public bool canSubmerge;
}
```

A `[Serializable]` attribútum nem öröklődik a szülőosztályból. Ha tehát egy osztályt egy `[Serializable]` attribútummal jelölt típusból származtatunk, a gyermekosztályt is a `[Serializable]` attribútummal kell megjelölnünk, különben nem tudjuk elmenteni. Valójában az objektumgráf minden tagját a `[Serializable]` attribútummal kell jelölnünk. Ha a `BinaryFormatter` vagy a `SoapFormatter` segítségével nem sorosítható objektumot próbálunk sorosítani, futásidőben a `SerializationException` jelenik meg.

Megjegyzés Mivel a `XmlSerializer` típus nem használja az objektumgráfokat, technikailag nem szükséges a típusokat a `[Serializable]` attribútummal megjelölni annak érdekében, hogy az objektum állapotát XML-ként mentsük el. Ha azonban azt szeretnénk, hogy a típusok minden lehetséges formátumban menthetők legyenek, érdemes minden sorosítandó típust a `[Serializable]` attribútumokkal megjelölni.

Nyilvános mezők, privát mezők és nyilvános tulajdonságok

Az egyszerűség kedvéért ezekben az osztályokban az adatmezőket nyilvánosként határoztuk meg. A nyilvános tulajdonságokkal felruházott privát adatok természetesen jobbak lennének az OO szempontjából. Az egyszerűség kedvéért azonban ezekre a típusokra nem határoztam meg semmilyen egyedi konstruktort, ezért azok az adatmezők, amelyeknek nem adtunk értéket, az ilyenkor várható alapértelmezett értékeket kapják.

Eltekintve az OO tervezési elvektől, kérdés lehet, hogy a különböző formázók hogyan várják a típusok adatmezőjére vonatkozó hozzáférés-módosítókat annak érdekében, hogy azok sorosíthatók legyenek. A válasz: attól függ. Ha a `BinaryFormatterrel` vagy a `SoapFormatterrel` mentjük el az objektumok állapotát, abszolút nem számít, milyen hozzáférés-módosítót használunk. Ez a két típus képes sorosítani *minden* sorosítható mezőt attól függetlenül, hogy ezek nyilvános mezők, privát mezők vagy nyilvános tulajdonságokat használó privát mezők. Vannak azonban olyan adatok, amelyeket nem szeretnénk az objektumgráfba menteni, ekkor a nyilvános és a privát mezőket szelektíven a `[NonSerialized]` attribútummal jelölhetjük, ahogy azt a `Radio` típus sztringmezőjével is tettük.

A helyzet azonban más, ha az `XmlSerializer` típust használjuk. Ez a típus *csak* a nyilvános adatmezőket vagy a nyilvános tulajdonságok által kiajánlott privát adatokat sorosítja, míg a tulajdonságon keresztül nem elérhető privát adatok figyelmen kívül maradnak. Példaként vegyük az alábbi sorosítható `Person` típust:

```
[Serializable]
public class Person
{
  // Nyilvános mező.
  public bool isAlive = true;

  // Privát mező.
  private int personAge = 21;

  // Nyilvános tulajdonság/privát adat.
  private string fName = string.Empty;
  public string FirstName
  {
    get { return fName; }
    set { fName = value; }
  }
}
```

Ha a `BinaryFormatter` vagy a `SoapFormatter` típust használtuk, megfigyelhetjük, hogy az `isAlive`, a `personAge` és a `fName` mezők mindegyikét elmentettük a kijelölt folyamba. Az `XmlSerializer` azonban *nem* menti el a `personAge` értékét, ugyanis ezt a privát adatelemet nem ajánlottuk ki nyilvános típustulajdonsággal. Ha a személy életkorát az `XmlSerializerrel` szeretnénk elmenteni, a mezőt nyilvánosként kell meghatároznunk, vagy be kell ágyaznunk a privát tagot egy nyilvános tulajdonságba.

A sorosító formázó kiválasztása

Ha a szükséges attribútumokkal konfiguráltuk a .NET sorosítási sémájában szereplő típusokat, a következő lépésben ki kell választanunk, melyik formátumot (bináris, SOAP vagy XML) használjuk az objektumok állapotának elmentésére. Az egyes lehetőségeket az alábbi osztályok mutatják:

- BinaryFormatter
- SoapFormatter
- XmlSerializer

A `BinaryFormatter` típus az objektumok állapotát tömörített bináris formátum segítségével sorosítja. A típust az `mscorlib.dll` részét képező `System.Runtime.Serialization.Formatters.Binary` névtér definiálja. Ha tehát ehhez a típushoz szeretnénk hozzáférni, egyszerűen csak az alábbi, C# `using` direktívát kell megadnunk:

```
// Hozzáférés a mscorlib.dll szerelvényben lévő
// BinaryFormatter típushoz.
using System.Runtime.Serialization.Formatters.Binary;
```

A `SoapFormatter` típus az objektumok állapotát SOAP-üzenetként menti el. Ezt a típust a *különálló szerelvényben* definiált `System.Runtime.Serialization.Formatters.Soap` névtérben határozhatjuk meg. Ha tehát az objektumgráfot SOAP-üzenetbe szeretnénk sorosítani először a Visual Studio 2008 Add Reference párbeszédpaneljével hivatkozzunk a `System.Runtime.Serialization.Formatters.Soap.dll` szerelvényre, majd adjuk meg az alábbi C# `using` direktívát:

```
// Kötelező referencia
// a System.Runtime.Serialization.Formatters.Soap.dll szerelvényre.
using System.Runtime.Serialization.Formatters.Soap;
```

Végül pedig, ha egy objektumfát XML-dokumentumként kívánunk elmenteni, használjuk a `XmlSerializer` típust. Ehhez csak határozzuk meg a névteret, majd állítsuk be a `System.Xml.dll` szerelvényre vonatkozó referenciát. Szerencsére minden Visual Studio 2008 projektsablon automatikus referenciával rendelkezik a `System.Xml.dll` szerelvényre, ezért nekünk csak az alábbi névteret kell használnunk:

```
// Definiálva a System.Xml.dll szerelvényben.
using System.Xml.Serialization;
```

Az IFormatter és az IRemotingFormatter interfészek

Függetlenül attól, hogy melyik formázót választjuk, ne feledjük, hogy azok mindegyike közvetlenül a `System.Object` névtérből származik, tehát egy sorosítás központú alaposztállyal *nincsen* közös tagkészletük. A `BinaryFormatter` és a `SoapFormatter` azonban az `IFormatter` és az `IRemotingFormatter` interfészek implementációján keresztül támogatja a gyakori tagokat (bármilyen furcsának is tűnik, a `XmlSerializer` azonban egyáltalán nem).

A `System.Runtime.Serialization.IFormatter` határozza meg azokat a legfontosabb `Serialize()` és `Deserialize()` metódusokat, amelyek az objektumgráfokat egy adott folyamba helyezik, vagy onnan kiveszik. Ezeken a tagokon kívül az `IFormatter` meghatározza még azt a pár tulajdonságot is, amelyeket az implementáló típus a háttérben használ:

```
public interface IFormatter
{
  SerializationBinder Binder { get; set; }
  StreamingContext Context { get; set; }
  ISurrogateSelector SurrogateSelector { get; set; }
  object Deserialize(System.IO.Stream serializationStream);
  void Serialize(System.IO.Stream serializationStream,
                 object graph);
}
```

A `System.Runtime.Remoting.Messaging.IRemotingFormatter` interfész (amelyet a .NET-remoting a háttérben használ) a `Serialize()` és a `Deserialize()` tagokat az elosztott sorosítások szempontjából sokkal hasznosabban terheli túl. Az `IRemotingFormatter` a sokkal általánosabb `IFormatter` interfészből származik:

```
public interface IRemotingFormatter : IFormatter
{
  object Deserialize(Stream serializationStream,
                     HeaderHandler handler);
  void Serialize(Stream serializationStream, object graph,
                 Header[] headers);
}
```

Bár a legtöbb sorosítási feladatban valószínűleg nem lesz közvetlenül dolgunk ezekkel az interfészekkel, érdemes tudni, hogy az interfészalapú polimorfizmus lehetővé teszi, hogy egy `IFormatter` referencia segítségével megtartsuk a `BinaryFormatter` vagy a `SoapFormatter` egy példányát. Tehát ha olyan metódust kívánunk építeni, amely ezen osztályok valamelyikének felhasználásával sorosítja az objektumgráfot, a következőket írjuk:

```
static void SerializeObjectGraph(IFormatter itfFormat,
                                  Stream destStream, object graph)
{
  itfFormat.Serialize(destStream, graph);
}
```

A formázók közötti típuspontosság

A három formázó közötti legnyilvánvalóbb különbség az objektumgráf folyamba mentésének módjában keresendő (bináris, SOAP vagy XML). Létezik még néhány apróbb pont is, amelyben különböznek egymástól, ezek közül kiemelkedik a formázók *típuspontosság* szempontja. A `BinaryFormatter` típus nemcsak az objektumgráfban található objektumok adatmezőit menti el, hanem minden egyes típus teljesen meghatározott nevét, valamint a meghatározó szerelvény teljes nevét is (név, verzió, nyilvános kulcstoken és kultúra). Ezek miatt az extra adatpontok miatt a `BinaryFormatter` ideális választásnak tűnik, ha az objektumokat érték szerint (pl. teljes másolatként), a gépek közötti korlátok átlépésével kívánjuk áthelyezni a .NET-központú alkalmazásokba.

A `SoapFormatter` az eredeti szerelvény nyomait egy XML-névtér segítségével tárolja el. Gondoljunk a korábban említett `Person` típusra. Ha a típust SOAP-üzenetként mentettük volna el, a `Person` kezdő elemét a generált xmlns minősítené. Nézzük meg az alábbi definíciórészletet, s közben figyeljünk az XML-névtérre:

```
<a1:Person id="ref-1" xmlns:a1=
    "http://schemas.microsoft.com/clr/nsassem/SimpleSerialize/
    MyApp%2C%20Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20
    PublicKeyToken%3Dnull">
  <isAlive>true</isAlive>
  <personAge>21</personAge>
  <fName id="ref-3"></fName>
</a1:Person>
```

Az `XmlSerializer` azonban *nem* törekszik a teljes típuspontosság megtartására, ezért nem tárolja el a típusok teljesen meghatározott nevét, sem az eredeti szerelvényt. Bár ez első ránézésre korlátozásnak tűnhet, az ok az XML-adatok megjelenítésének nyílt természetében keresendő. Nézzük meg a `Person` típus egy lehetséges XML-ábrázolását:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <isAlive>true</isAlive>
  <PersonAge>21</PersonAge>
  <FirstName />
</Person>
```

Ha az objektumok állapotát úgy szeretnénk elmenteni, hogy bármely operációs rendszer (Windows XP, Mac OS X és a különböző Linux-disztribúciók), alkalmazási keretrendszer (.NET, J2EE, COM stb.) vagy programozási nyelv használhassa, ne ragaszkodjunk a teljes típuspontossághoz, hiszen nem tételezhetjük fel, hogy minden lehetséges befogadó ismeri a .NET-specifikus adattípusokat. Ennek alapján a `SoapFormatter` és az `XmlSerializer` a legmegfelelőbb választás, ha az elmentett objektumfának a lehető legszélesebb elérhetőséget szeretnénk biztosítani.

Objektumok sorosítása a BinaryFormatterrel

Annak szemléltetésére, hogy milyen egyszerűen menthetjük el a `JamesBondCar` egy példányát fizikai fájlokba, először használjuk a `BinaryFormatter` típust. Ismét a `BinaryFormatter` típus két legfontosabb metódusára, a `Serialize()` és a `Deserialize()` metódusra kell figyelnünk:

- `Serialize()`: Az objektumgráfokat byte-ok sorozataként egy adott folyamba menti.
- `Deserialize()`: Az elmentett byte-sorozatot objektumgráffá konvertálja.

Tételezzük fel, hogy létrehoztuk a `JamesBondCar` egy példányát, módosítottunk néhány állapotadatot, és egy `*.dat` fájlba szeretnénk elmenteni. Először magát a `*.dat` fájlt kell létrehoznunk. Ezt a `System.IO.FileStream` típus példányosításával érhetjük el (lásd a 20. fejezetet). Ezután egyszerűen hozzuk létre a `BinaryFormatter` egy példányát, és adjuk át a `FileStremet` és az objektumgráfot a sorosításhoz. Vizsgáljuk meg a következő `Main()` metódust:

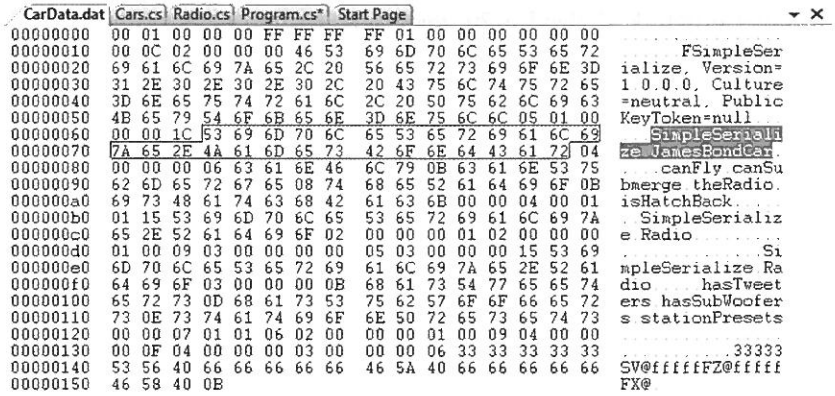
```
// Feltétlenül importáljuk
// a System.Runtime.Serialization.Formatters.Binary
// és a System.IO névtereket.
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Serialization *****\n");

  // Hozzunk létre egy JamesBondCar osztályt, és határozzuk meg
  // az állapotát.
  JamesBondCar jbc = new JamesBondCar();
  jbc.canFly = true;
```

```
    jbc.canSubmerge = false;
    jbc.theRadio.stationPresets = new double[]{89.3, 105.1, 97.1};
    jbc.theRadio.hasTweeters = true;

    // Most mentsük el az autót egy adott, bináris formátumú fájlba.
    SaveAsBinaryFormat(jbc, "CarData.dat");
    Console.ReadLine();
  }
```

A `SaveAsBinaryFormat()` metódust az alábbiak szerint valósítjuk meg:

```
static void SaveAsBinaryFormat(object objGraph, string fileName)
{
  // Mentsük az objektumot egy CarData.dat nevű bináris fájlba.
  BinaryFormatter binFormat = new BinaryFormatter();

  using(Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
  {
    binFormat.Serialize(fStream, objGraph);
  }
  Console.WriteLine("=> Saved car in binary format!");
}
```

Láthatjuk, hogy a `BinaryFormatter.Serialize()` metódus felelős az objektumgráf összeállításáért és a byte-sorozat valamilyen *Stream*ből származtatott típusba helyezéséért. Ebben az esetben a folyam egy fizikai fájl. Az objektumtípusokat bármilyen *Stream*ből származó típusba, például memóriaterületre, hálózati folyamba stb. sorosíthatjuk. A program futtatása után a projekt \bin\ Debug mappájához tallózva megnézhetjük annak a `CarData.dat` fájlnak a tartalmát, amely a `JamesBondCarnak` ezt a példányát jeleníti meg. A 21.2. ábra ezt a Visual Studio 2008-ban megnyitott fájlt ábrázolja.



21.2. ábra: *A BinaryFormatter használatával sorosított JamesBondCar*

Objektumok visszaállítása a BinaryFormatterrel

Tételezzük fel, hogy a bináris fájlba sorosított `JamesBondCar` típust szeretnénk objektumváltozókba beolvasni. Miután programozottan megnyitottuk a `CarData.dat` fájlt (a `File.OpenRead()` metóduson keresztül), egyszerűen csak használjuk a `BinaryFormatterDeserialize()` metódusát. A `Deserialize()` egy generikus `System.Object` típust ad vissza, így meg kell adnunk egy explicit kasztot az alábbiak szerint:

```
static void LoadFromBinaryFile(string fileName)
{
  BinaryFormatter binFormat = new BinaryFormatter();

  // Olvassuk be a JamesBondCar típust a bináris fájlból.
  using(Stream fStream = File.OpenRead(fileName))
  {
    JamesBondCar carFromDisk =
                (JamesBondCar)binFormat.Deserialize(fStream);
    Console.WriteLine("Can this car fly? : {0}",
                      carFromDisk.canFly);
  }
}
```

A `Deserialize()` hívásakor az elmentett objektumgráf helyét jelző *Stream*ből származtatott típust adunk meg.. Miután az objektumot visszakasztoltuk a megfelelő típusba, látható, hogy az állapotadatok az utolsó mentési állapotot tükrözik.

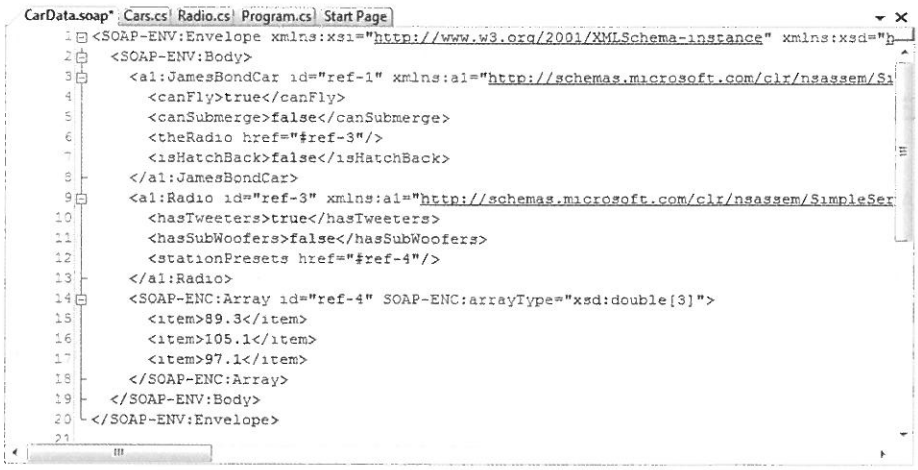
Objektumok sorosítása a SoapFormatterrel

A következő választható formázó a `SoapFormatter` típus. A `SoapFormatter` az objektumgráfokat SOAP-üzenetként menti el, és ez jó választásnak tűnik abban az esetben, ha az objektumokat távoli helyeken, különböző környezeteken keresztül szeretnénk megosztani. Ha nem ismerjük a Simple Object Access Protocol (SOAP) specifikációját, röviden annyit azért érdemes tudni róla, hogy a SOAP egy olyan standard folyamatot határoz meg, amelyben a metódusokat platform- és operációsrendszer-független módon hívhatjuk meg.

Ha beállítottunk egy referenciát a `System.Runtime.Serialization.Formatters.Soap.dll` szerelvényre (és importáltuk a `System.Runtime.Serialization.Formatters.Soap` névteret), SOAP-üzenetként sorosíthatunk és állíthatunk vissza egy `JamesBondCar` objektumot azáltal, hogy a `BinaryFormatter` minden egyes előfordulását a `SoapFormatter` típussal helyettesítjük. Nézzük meg a `Program` osztály alábbi, új metódusát, amely az objektumokat helyi fájlba sorosítja:

```
// Feltétlenül importáljuk
// a System.Runtime.Serialization.Formatters.Soap névteret,
// és állítsunk fel referenciát a
// System.Runtime.Serialization.Formatters.Soap.dll szerelvényre.
static void SaveAsSoapFormat (object objGraph, string fileName)
{
  // Mentsük az objektumot SOAP-formátumban egy CarData.soap
  // nevű fájlba.
  SoapFormatter soapFormat = new SoapFormatter();

  using(Stream fStream = new FileStream(fileName,
                FileMode.Create, FileAccess.Write, FileShare.None))
  {
    soapFormat.Serialize(fStream, objGraph);
  }
  Console.WriteLine("=> Saved car in SOAP format!");
}
```

Ahogy azt az előbb is tettük, használjuk a `Serialize()` és a `Deserialize()` metódusokat az objektumgráf folyamba helyezésére és az onnan történő kivételre. Ha a metódust a `Main()` metódusból hívjuk meg, és futtatjuk az alkalmazást, megnyithatjuk az eredményként létrejött `*.soap` fájlt.



21.3. ábra: *A SoapFormatter használatával sorosított JamesBondCar*

Itt találhatjuk azokat az XML-elemeket, amelyek az aktuális `JamesBondCar` állapotára vonatkozó értékeket, valamint a gráfban lévő objektumokat a `#ref` tokeneken keresztül jelölik (lásd 21.3. ábra).

Objektumok sorosítása az XmlSerializerrel

A SOAP- és a bináris formázók mellett a `System.Xml.dll` szerelvény egy harmadik formázót is biztosít: a `System.Xml.Serialization.XmlSerializer` egy adott objektum *nyilvános* állapotát SOAP-üzenetbe ágyazott XML-adatok helyett tiszta XML-ként tárolja. Ennek a típusnak a használata némiképp különbözik a `SoapFormatterétől` vagy a `BinaryFormatterétől`. Nézzük meg az alábbi forráskódot, amely feltételezi, hogy a `System.Xml.Serialization` névteret már importáltuk:

```
static void SaveAsXmlFormat(object objGraph, string fileName)
{
  // Mentsük az objektumot XML-formátumban egy CarData.xml
  // nevű fájlba.
  XmlSerializer xmlFormat = new XmlSerializer(typeof(JamesBondCar),
                new Type[] { typeof(Radio), typeof(Car) });

  using(Stream fStream = new FileStream(fileName,
  FileMode.Create, FileAccess.Write, FileShare.None))
  {
    xmlFormat.Serialize(fStream, objGraph);
  }
  Console.WriteLine("=> Saved car in XML format!");
}
```

A `XmlSerializer` típus alkalmazásakor felmerülő leglényegesebb különbség, hogy ehhez meg kell határoznunk minden egyes, a gyökérbe ágyazott alelem típusára vonatkozó információt. Az `XmlSerializer` első konstruktorargumentuma meghatározza az XML-fájl gyökérelemét, míg a második argumentum a `System.Type` típusok tömbje, amely az alelemekre vonatkozó metaadatokat tartalmazza. Ha belenézünk az újonnan generált XML-fájlba (feltételezve, hogy ezt az új metódust valóban a `Main()` metódusban hívtuk meg), a 21.4. ábrán látható XML-adatokat találjuk.

CarData.xml | Cars.cs | Radio.cs | Program.cs | Start Page

```
<?xml version="1.0"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <theRadio>
    <hasTweeters>true</hasTweeters>
    <hasSubWoofers>false</hasSubWoofers>
    <stationPresets>
      <double>89.3</double>
      <double>105.1</double>
      <double>97.1</double>
    </stationPresets>
    <radioID>XF-552RR6</radioID>
  </theRadio>
  <isHatchBack>false</isHatchBack>
  <canFly>true</canFly>
  <canSubmerge>false</canSubmerge>
</JamesBondCar>
```

21.4. ábra: *A XmlSerializer használatával sorosított JamesBondCar*

Megjegyzés Az `XmlSerializer` típus egyik feltétele, hogy minden, az objektumgráfban található sorosított típus támogasson egy alapértelmezett konstruktort (tehát mindenképpen adjuk hozzá az egyedi konstruktorok meghatározásakor). Ha ez nem történik meg, futásidőben `InvalidOperationException` jelenik meg.

Generált XML-adatok szabályozása

Az XML-technológiákra vonatkozó háttérismeretekből tudjuk, hogy gyakran létfontosságú annak biztosítása, hogy az XML-dokumentum adatai megfeleljenek az adatok *érvényességét* meghatározó szabályoknak. Tudjuk, hogy az „érvényes" XML-dokumentumoknak semmi köze az XML-elemek szintaktikai jólétéhez (pl. minden nyitóelemhez tartoznia kell egy záróelemnek is). Az érvényes dokumentumoknak inkább olyan – megegyezésen alapuló – formázási szabályoknak kell megfelelniük (pl. az X mezőt attribútumként, nem pedig alelemként kell kifejezni), amelyeket jellemzően egy XML-sémában vagy egy DTD- (dokumentum-típusdefiníció) fájlban határozunk meg.

Az alapértelmezés szerint egy `[Serializable]` típus minden nyilvános adatát alelemként és nem XML-attribútumként formázzuk meg. Ha szeretnénk meghatározni, hogy az `XmlSerializer` hogyan generálja az eredményként létrejövő XML-dokumentumot, a `[Serializable]` típusokat tetszőleges számú további, a `System.Xml.Serialization` névtérből származó attribútummal ruházhatjuk fel. A 21.1. táblázat néhány (de nem minden) olyan attribútumot tartalmaz, amelyek befolyásolják az XML-adatok folyamba történő kódolásának a módját.

Attribútum	Jelentés
`XmlAttributeAttribute`	A tagot XML-attribútumként sorosítja.
`XmlElementAttribute`	A mezőt vagy tulajdonságot XML-elemként sorosítja.
`XmlEnumAttribute`	Meghatározza, hogy egy felsorolt típus egy tagjához milyen nevű elem tartozzon a sorosítás során nyert XML-ben.
`XmlRootAttribute`	Ez az attribútum határozza meg, hogy a gyökérelem milyen szerkezetű legyen (névtér és elemnév).
`XmlTextAttribute`	A mező vagy tulajdonság sorosításának XML-szövegként kell történnie.
`XmlTypeAttribute`	Az XML-típus neve és névtere.

21.1. táblázat: *A System.Xml.Serialization névtér néhány attribútuma*

Először egy egyszerű példán figyeljük meg, hogy a `JamesBondCar` adatmezőinek elmentése milyen XML-t eredményez:

```
<?xml version="1.0" encoding="utf-8"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  ...
  <canFly>true</canFly>
  <canSubmerge>false</canSubmerge>
</JamesBondCar>
```

Ha olyan egyedi XML-névteret szeretnénk megadni, amely meghatározza a `JamesBondCar` osztályt, valamint a `canFly` és a `canSubmerge` értékeket XML-attribútumként kódolja, módosítsuk a `JamesBondCar` C#-definícióját az alábbiak szerint:

```
[Serializable,
XmlRoot(Namespace = "http://www.intertech.com")]
public class JamesBondCar : Car
{
  [XmlAttribute]
  public bool canFly;
  [XmlAttribute]
  public bool canSubmerge;
}
```

Ennek eredménye a következő XML-dokumentum lesz (figyeljük meg a kezdő `<JamesBondCar>` elemet):

```
<?xml version="1.0"""?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  canFly="true" canSubmerge="false"
  xmlns="http://www.intertechtraining.com">
  ...
</JamesBondCar>
```

Természetesen számos attribútumot használhatunk annak vezérlésére, hogy az `XmlSerializer` hogyan generálja a létrejövő XML-dokumentumokat. További részletekért tekintsük át a .NET Framework 3.5 SDK dokumentációjában a `System.Xml.Serialization` névtérre vonatkozó információkat.

Objektumgyűjtemények sorosítása

A továbbiakban azt vizsgáljuk meg, hogyan menthetjük el az objektumok egy halmazát. Az `IFormatter` interfész `Serialize()` metódusa nem teszi lehetővé, hogy inputként tetszőleges számú objektumot adjunk meg (csak egy önálló `System.Object` objektumot). Ennek megfelelően a `Deserialize()` visszatérési értéke szintén csak egy önálló `System.Object` (az `XmlSerializer` típusra ugyanezek az alapvető korlátozások érvényesek):

```
public interface IFormatter
{
  ...
  object Deserialize(System.IO.Stream serializationStream);
  void Serialize(System.IO.Stream serializationStream,
                 object graph);
}
```

A `System.Object` valójában az objektum teljes fáját képviseli. Ez alapján, ha átadunk egy `[Serializable]` attribútummal megjelölt objektumot, amely további `[Serializable]` objektumokat tartalmaz, akkor az objektumok teljes halmaza egyetlen metódushívással rögzül. A `System.Collections` és a `System.Collections.Generic` névterekben található típusok legnagyobb része már rendelkezik `[Serializable]` jelöléssel. Ezért ha sorosítani szeretnénk az objektumok egy halmazát, egyszerűen adjuk hozzá a halmazt egy tárolóhoz (pl. egy `ArrayListhez` vagy egy `List<T>`-hez), majd sorosítsuk az objektumot a kívánt folyamhoz.

Tételezzük fel, hogy néhány állapotadat meghatározásához kétargumentumú konstruktorral frissítettük a `JamesBondCar` osztályt (láthatjuk, hogy az `XmlSerializer` követelményeinek megfelelően az alapértelmezett konstruktort is hozzáadjuk):

```
[Serializable,
XmlRoot(Namespace = "http://www.intertech.com")]
public class JamesBondCar : Car
{
  public JamesBondCar(bool skyWorthy, bool seaWorthy)
  {
    canFly = skyWorthy;
    canSubmerge = seaWorthy;
  }
  // Az XmlSerializer megkövetel egy alapértelmezett konstruktort!
  public JamesBondCar(){}
...
}
```

Így most már az alábbiak szerint akárhány `JamesBondCar` osztályt menthetünk el:

```
static void SaveListOfCars()
{
  // Most sorosítsuk a JamesBondCar List<T> objektumot.
  List<JamesBondCar> myCars = new List<JamesBondCar>();
  myCars.Add(new JamesBondCar(true, true));
  myCars.Add(new JamesBondCar(true, false));
  myCars.Add(new JamesBondCar(false, true));
  myCars.Add(new JamesBondCar(false, false));

  using(Stream fStream = new FileStream("CarCollection.xml",
        FileMode.Create, FileAccess.Write, FileShare.None))
  {
    XmlSerializer xmlFormat =
                  new XmlSerializer(typeof(List<JamesBondCar>),
    new Type[] { typeof(JamesBondCar), typeof(Car),
                 typeof(Radio) });
    xmlFormat.Serialize(fStream, myCars);
  }
  Console.WriteLine("=> Saved list of cars!");
}
```

Minthogy az `XmlSerializer` típust használtuk, meg kell határoznunk a típusinformációt minden egyes, a gyökérobjektumban található alobjektumra vonatkozóan (a gyökérobjektum jelen esetben a `List<JamesBondCar>`). Ha a `BinaryFormatter` vagy a `SoapFormatter` típust használtuk volna, még egyértelműbb logikai úton haladnánk, például:

```
static void SaveListOfCarsAsBinary()
{
  // Mentsük el az ArrayList objektumot (myCars) bináris
  // objektumként.
  List<JamesBondCar> myCars = new List<JamesBondCar>();

  BinaryFormatter binFormat = new BinaryFormatter();
  using(Stream fStream = new FileStream("AllMyCars.dat",
                FileMode.Create, FileAccess.Write, FileShare.None))
  {
    binFormat.Serialize(fStream, myCars);
  }
  Console.WriteLine("=> Saved list of cars in binary!");
}
```

Forráskód A SimpleSerialize projektet a forráskódkönyvtárban a 21. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A sorosítási folyamat testreszabása

A legtöbb esetben az alapértelmezett .NET platform sorosítási séma pontosan olyan lesz, amilyet szeretnénk. Egyszerűen használjuk a `[Serializable]` attribútumot a kapcsolódó típusokhoz, és adjuk át a feldolgozandó objektumfát a kívánt formázónak. Néhányszor előfordulhat, hogy nagyobb beleszólást szeretnénk a fa szerkesztésekor és kezelésekor a sorosítási folyamatba. Ha például olyan üzleti szabállyal rendelkezünk, amely szerint minden adatmezőt adott formátumban kell elmenteni, vagy további olyan adatelemeket szeretnénk a folyamhoz hozzáadni, amelyek nem képeződnek le közvetlenül a rögzítendő objektumban található mezőkbe (időbélyegek, egyedi azonosítók stb.).

Ha jobban be szeretnénk kapcsolódni az objektumsorosítás folyamatába, a `System.Runtime.Serialization` névtér számos olyan típust bocsát a rendelkezésünkre, amelyek mindezt lehetővé teszik. A 21.2. táblázat a legfontosabb típusok közül ismertet néhányat.

Típus	Jelentés
`ISerializable`	Ennek az interfésznek a `[Serializable]` típusra történő implementálásával vezérelhetjük annak sorosítását és visszaállítását.
`ObjectIDGenerator`	Ez a típus generálja az objektumgráf tagjainak azonosítóját.

Típus	Jelentés
`[OnDeserialized]`	Ez az attribútum teszi lehetővé, hogy specifikáljuk azt a metódust, amelyet közvetlenül az objektum visszaállítása után hívunk.
`[OnDeserializing ]`	Ez az attribútum teszi lehetővé, hogy specifikáljuk azt a metódust, amelyet az objektum visszaállítása előtt hívunk.
`[OnSerialized]`	Ez az attribútum teszi lehetővé, hogy specifikáljuk azt a metódust, amelyet közvetlenül az objektum sorosítása után hívunk.
`[OnSerializing ]`	Ez az attribútum teszi lehetővé, hogy specifikáljuk azt a metódust, amelyet a sorosítási folyamat előtt hívunk.
`[OptionalField]`	Ez az attribútum teszi lehetővé, hogy meghatározzuk a típusoknak azokat a mezőit, amelyek kimaradhatnak az adott folyamból.
`SerializationInfo`	Ez az osztály egy ún. „tulajdonságkosár", amely a sorosítási folyamat során az objektumok állapotát jelző név/érték párokat képviseli.

21.2. táblázat: *A System.Runtime.Serialization névtér alapvető típusai*

Az objektumsorosítás háttérrészletei

Mielőtt megvizsgálnánk a sorosítási folyamat testreszabásának különböző módjait, érdemes a háttérrel is foglalkozni. Amikor a `BinaryFormatter` sorosítja az objektumgráfot, az alábbi információkat továbbítja az adott folyamba:

- a gráfban található objektum teljesen meghatározott nevét (pl. `MyApp.JamesBondCar`),
- az objektumgráfot meghatározó szerelvény nevét (pl. `MyApp.exe`),
- az `SerializationInfo` osztály egy példányát, amely minden, az objektumgráf tagjai által fenntartott állapotadatot tartalmaz.

A `BinaryFormatter` a visszaállítási folyamat során az objektum egy azonos másolatának létrehozásához ugyanazt az információt használja, amelyet az alatta lévő folyamból nyer ki. A `SoapFormatter` által alkalmazott folyamat meglehetősen hasonló.

Megjegyzés Az XmlSerializer, annak érdekében, hogy a lehető legnagyobb mértékben megtartsa az objektum állapotának hordozhatóságát, nem menti el a típusok teljesen meghatározott nevét, sem a meghatározó szerelvény nevét. Ez a típus csak a megjelenített nyilvános adatok mentésével foglalkozik.

Azon túl, hogy a szükséges adatokat a folyamba helyezik, illetve onnan kiveszik, a formázók az alábbi infrastrukturális adatok szempontjából elemzik is az objektumgráfban található tagokat:

- Ellenőrzik, hogy az objektumok meg vannak-e jelölve a `[Serializable]` attribútummal. Ha az objektum nincs megjelölve, `SerializationException` jelenik meg.
- Ha az objektum meg van jelölve a `[Serializable]` attribútummal, ellenőrzik és meghatározzák, hogy az objektum implementálja-e az `ISerializable` interfészt. Ebben az esetben a `GetObjectData()` metódust hívják az objektumra.
- Ha az objektum nem implementálja az `ISerializable` interfészt, az alapértelmezett sorosítási folyamat sorosítja az összes, `[NonSerialized]` attribútummal meg nem jelölt mezőt.

Azon kívül, hogy meghatározzák, hogy a típus támogatja-e az `ISerializable` interfészt, a formázók azért is felelősek, hogy felfedjék, vajon a kérdéses típusok támogatják-e azokat a tagokat, amelyeket az `[OnSerializing]`, az `[OnSerialized]`, az `[OnDeserializing]` vagy az `[OnDeserialized]` attribútumban határoztunk meg. Mielőtt megvizsgálnánk ezeknek az attribútumoknak a szerepét, nézzük meg közelebbről az `ISerializable` funkcióját.

Sorosítások testreszabása az ISerializable használatával

A `[Serializable]` attribútummal megjelölt objektumoknak lehetősége van implementálni az `ISerializable` interfészt. Ezzel „részévé válhatunk" a sorosítási folyamatnak, és az adatokat előzetesen vagy utólagosan formázhatjuk.

Megjegyzés A .NET 2.0 megjelenése óta a sorosítási folyamat testreszabásának preferált módja a sorosítási attribútumok használata (ennek leírását lásd alább). A meglévő rendszerek fenntartása érdekében azonban fontos, hogy ismerjük az `ISerializable` interfészt is.

Az `ISerializable` interfész meglehetősen egyszerű, hiszen csak egy metódust határoz meg, ez pedig a `GetObjectData()`:

```
// Ha a sorosítási folyamat finomhangolására törekszünk,
// implementáljuk az ISerializable interfészt.
public interface ISerializable
{
  void GetObjectData(SerializationInfo info,
                     StreamingContext context);
}
```

A `GetObjectData()` metódust az adott formázó a sorosítási folyamat során automatikusan hívja. Az implementációja feltölti a bemeneti `SerializationInfo` paramétert név/érték párok sorozatával, amelyek (jellemzően) az elmentendő objektum adatmezőire képződnek le. A `SerializationInfo` számos variációt határoz meg a felüldefiniált `AddValue()` metódusra, valamint a tulajdonságok egy kis gyűjteményére, ezek lehetővé teszik, hogy az adott típus lekérdezze és módosítsa a típus nevét, a meghatározó szerelvényt és a tagok számát. Nézzünk meg egy kódrészletet:

```
public sealed class SerializationInfo : object
{
  public SerializationInfo(Type type,
                           IFormatterConverter converter);
  public string AssemblyName { get; set; }
  public string FullTypeName { get; set; }
  public int MemberCount { get; }
  public void AddValue(string name, short value);
  public void AddValue(string name, UInt16 value);
  public void AddValue(string name, int value);
  ...
}
```

Az `ISerializable` interfészt implementáló típusoknak egy, az alábbi szignatúrával rendelkező, speciális konstruktort is meg kell határozni:

```
// Az egyedi konstruktort el kell látni ezzel az aláírással
// annak érdekében, hogy a futtatómotor meg tudja határozni
// az objektum állapotát.
[Serializable]
class SomeClass : ISerializable
{
  protected SomeClass (SerializationInfo si,
                       StreamingContext ctx) {...}
  ...
}
```

Ennek a konstruktornak a láthatósága *védettre* van beállítva. Ez azért lehetséges, mert a formázó a láthatóságtól függetlenül hozzáfér ehhez a taghoz. Ezek a speciális konstruktorok általában védett (vagy privát) jelölést kapnak, hogy megakadályozzák, hogy az alkalmi objektumfelhasználók ilyen módon hozzanak létre objektumokat. A konstruktor első paramétere a `SerializationInfo` típus egy példánya (ahogy azt korábban is láttuk).

A speciális konstruktor második paramétere egy `StreamingContext` típus, amely a bitek forrásával vagy rendeltetési helyével kapcsolatban tartalmaz információt. Ennek a típusnak a leginformatívabb tagja a `State` tulajdonság, amely egy, a `StreamingContextStates` felsorolt típusból származó értéket képvisel. A felsorolt típus értékei az aktuális folyam alapösszeállítását ábrázolják.

Ha csak nem implementálunk néhány alacsony szintű, egyedi remoting-szolgáltatást, aligha kell közvetlenül foglalkoznunk ezzel a típussal. Ennek ellenére megadjuk a `StreamingContextStates` felsorolt típus lehetséges értékeit (a részletek a .NET Framework 3.5 SDK dokumentációjában találhatók):

```
public enum StreamingContextStates
{
  CrossProcess,
  CrossMachine,
  File,
  Persistence,
  Remoting,
  Other,
  Clone,
  CrossAppDomain,
  All
}
```

A `ISerializable` interfészt használó sorosítási folyamat testreszabásának illusztrálásához tételezzük fel, hogy olyan új konzolalkalmazást hozunk létre (`CustomSerialization` névvel), amely két sztringadatot tartalmazó osztálytípust határoz meg. Továbbá tegyük fel, hogy a sztringobjektumokat csupa nagybetűvel kell sorosítani, és csupa kisbetűvel kell visszaállítani. Ezeknek a szabályoknak az érdekében a következőképpen implementáljuk az `ISerializable` interfészt (ne felejtsük el importálni a `System.Runtime.Serialization` névteret):

```
[Serializable]
class StringData : ISerializable
{
  public string dataItemOne = "First data block";
  public string dataItemTwo= "More data";
```

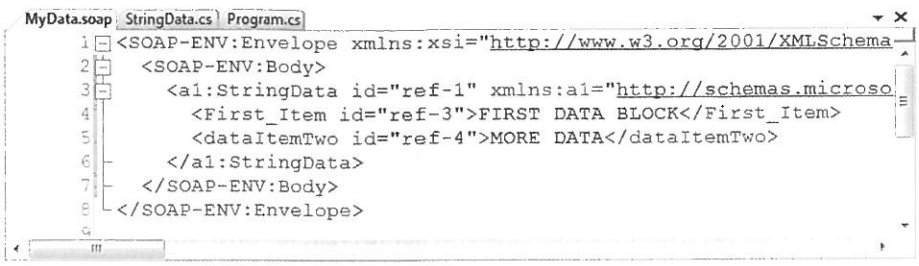
```
  public StringData(){}
  protected StringData(SerializationInfo si, StreamingContext ctx)
  {
    // Töltsük fel újra a tagváltozókat a folyamból.
    dataItemOne = si.GetString("First_Item").ToLower();
    dataItemTwo = si.GetString("dataItemTwo").ToLower();
  }

  void ISerializable.GetObjectData(SerializationInfo info,
                                   StreamingContext ctx)
  {
    // Töltsük fel a SerializationInfo objektumot
    // a formázott adatokkal.
    info.AddValue("First_Item", dataItemOne.ToUpper());
    info.AddValue("dataItemTwo", dataItemTwo.ToUpper());
  }
}
```

Amikor a `GetObjectData()` metódusból töltjük fel a `SerializationInfo` típust, az adatoknak *nem* kell ugyanazt a nevet adnunk, mint a típus belső tagváltozóinak. Ez akkor hasznos, ha a típust és a perszisztálási formátumot még jobban függetleníteni szeretnénk.

A testreszabás teszteléséhez tételezzük fel, hogy elmentettük a `MyStringData` egy példányát egy `SoapFormatter` használatával:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Custom Serialization *****");

  // Emlékszünk, hogy ez a típus implementálja az ISerializable
  // interfészt.
  StringData myData = new StringData();

  // Mentsük el SOAP-formátumban egy helyi fájlba.
  SoapFormatter soapFormat = new SoapFormatter();
  using(Stream fStream = new FileStream("MyData.soap",
        FileMode.Create, FileAccess.Write, FileShare.None))
  {
    soapFormat.Serialize(fStream, myData);
  }
  Console.ReadLine();
}
```

Ha megnézzük az így létrejövő `*.soap` fájlt, látjuk, hogy a sztringmezőket nagybetűsen tároljuk (lásd a 21.5.ábrát).



21.5. ábra: *A sorosítás testreszabása az ISerializable használatával*

Sorosítások testreszabása attribútumokkal

Bár az `ISerializable` interfész implementációja a sorosítási folyamat testreszabásának egy lehetséges módja, a .NET 2.0 megjelenése óta a sorosítási folyamat testreszabása inkább az új sorosítás-központú attribútumok segítségével: `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]` vagy `[OnDeserialized]` megadott metódusokkal történik. Ezeknek az attribútumoknak a használata kevésbé nehézkes, mint az `ISerializable` implementálása, hiszen nem kell manuálisan közreműködnünk a bemeneti `SerializationInfo` paraméterekkel. Ehelyett, miközben a formázó elvégzi a típussal kapcsolatos munkát, közvetlenül módosíthatjuk az állapotadatokat.

Megjegyzés Ezek a sorosítási attribútumok a `System.Runtime.Serialization` névtéren belül vannak definiálva.

Ezeknek az attibútumoknak a használatához egy olyan metódust kell definiálni, amely megkapja a `StreamingContext` paramétert, és nem ad vissza semmit (különben futásidejű kivételt kapunk). Nem kell minden egyes sorosítás-központú attribútumot számba vennünk, elég csak a sorosítás számunkra érdekes állapotaival foglalkoznunk. Ennek illusztrálására nézzünk meg egy új `[Serializable]` típust, amely ugyanazokkal a követelményekkel rendelkezik, mint a `StringData`, és ez alkalommal az `[OnSerializing]` és az `[OnDeserialized]` attribútumokat használja:

```
[Serializable]
class MoreData
{
  public string dataItemOne = "First data block";
  public string dataItemTwo= "More data";
```

```
    [OnSerializing]
    private void OnSerializing(StreamingContext context)
    {
      // Hívás a sorosítási folyamat alatt.
      dataItemOne = dataItemOne.ToUpper();
      dataItemTwo = dataItemTwo.ToUpper();
    }

    [OnDeserialized]
    private void OnDeserialized(StreamingContext context)
    {
      // Hívás a visszaállítási folyamat befejezését követően.
      dataItemOne = dataItemOne.ToLower();
      dataItemTwo = dataItemTwo.ToLower();
    }
}
```

Ha az új típust sorosítani szeretnénk, ismét azt tapasztalnánk, hogy az adatok sorosítása nagybetűs, míg a visszaállítása kisbetűs.

Forráskód A CustomSerialization projektet a forráskódkönyvtárban a 21. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Az előző példa segítségével megismertük az objektumsorosítás legfőbb szolgáltatásait, köztük a folyamat testreszabásának különböző módjait. A sorosítási és a visszaállítási folyamat révén nagy mennyiségű adatot is egyszerűen elmenthetünk, ez sokkal kevesebb munkát igényel, mintha a `System.IO` névtér különböző olvasó/író osztályaival dolgoznánk. Ha közelebbről is meg szeretnénk ismerkedni a .NET platformnak ezzel az aspektusával, olvassuk el a `BinaryFormatter`, a `SoapFormatter` és az `XmlSerializer` típusokról szóló információkat a .NET Framework 3.5 SDK dokumentációjában.

Összefoglalás

Ebben a fejezetben az objektumsorosító szolgáltatások témáját tárgyaltuk. A .NET platform az objektumgráfokat használja azon kapcsolódó objektumok teljes halmazának figyelembevételére, amelyeket egy folyamba kell elmenteni. Mindaddig, amíg minden objektumgráfot a `[Serializable]` attribútummal jelölünk, megválaszthatjuk az adatok sorosításának a formátumát (bináris, SOAP vagy XML). A nem egyedi sorosítási folyamatoknak kétféle testreszabási módja lehetséges.

Először megnéztük az `ISerializable` interfész implementálásának (és egy speciális privát konstruktor támogatásának) a módját annak érdekében, hogy nagyobb befolyással lehessünk arra, hogy a megadott adatokat hogyan mentsék el a formázók. Ezt követően megismertünk néhány olyan .NET-attribútumot, amelyekkel leegyszerűsíthető a sorosítás testreszabása. A `StreamingContext` paraméterrel rendelkező tagoknál az `[OnSerializing]`, az `[OnSerialized]`, az `[OnDeserializing]` vagy az `[OnDeserialized]` attribútumot alkalmazzuk, a formázók pedig ennek megfelelően fogják őket meghívni.

HUSZONKETTEDIK FEJEZET

ADO.NET, 1. rész: Az élő kapcsolat

A .NET platform számos névteret definiál, amelyek segítségével a gépen található lokális vagy távoli relációs adatbázisokat kezelni tudjuk. Ezeket a névtereknek a gyűjtőneve *ADO.NET*. Először körvonalazzuk az ADO.NET általános szerepét, majd az adatszolgáltatók témakörét tárgyaljuk. A .NET platform számos adatszolgáltatót támogat, ezek mindegyikét adott adatbázis-kezelő rendszerrel (Microsoft SQL Server, Oracle, MySQL stb.) történő kommunikációra optimalizálták.

A különböző adatszolgáltatók közös funkcióit megvizsgálva megnézzük a data provider factory minta működését. A `System.Data.Common` névtér használatával (és a hozzátartozó `App.config` fájl segítségével) elkészíthetünk egy olyan egyszerű kódot, amely dinamikusan képes választani és használni a mögöttes adatszolgáltatók közül anélkül, hogy az alkalmazás kódját újra kellene fordítani vagy telepíteni.

Olyan egyedi adathozzáférési könyvtárt fogunk tudni készíteni (`AutoLotDAL.dll`), amely különböző, egy egyedi adatbázison, az AutoLoton végrehajtható adatbázis-műveleteket valósít meg. A könyvtárat a következő fejezetben továbbfejlesztjük, és a továbbiakban többször használni is fogjuk. A fejezet végén megvizsgáljuk, hogyan lehet a Microsoft SQL Serverrel aszinkron módon kommunikálni a `System.Data.SqlClient` névtér típusainak felhasználásával, valamint bevezetjük az adatbázis-tranzakciók fogalmát.

Az ADO.NET magas szintű meghatározása

Ha ismerjük a Microsoft korábbi, COM-alapú adatelérési modelljét (Active Data Objects vagy ADO), tudjuk, hogy az ADO.NET-nek meglehetősen kevés köze van az ADO-hoz. Bár van kapcsolat a két rendszer között (pl. mindkettő tartalmazza a kapcsolat- és utasításobjektumok fogalmát), néhány

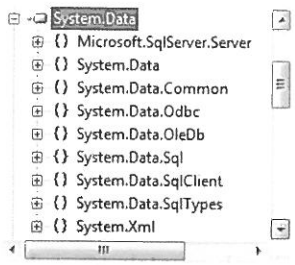
ADO-típus (mint pl. a Recordset) már nem létezik. Továbbá számos olyan ADO.NET-típus van, amelyhez nem találunk a klasszikus ADO-ban megfeleltethető párt (ilyen pl. az adatillesztő).

Ellentétben a klasszikus ADO-val, amelyet a szorosan kapcsolódó kliens/szerver rendszerekre hoztak létre, az ADO.NET a kapcsolat nélküli világot szem előtt tartva készült el a `DataSetek` használatával. A típus tetszőleges számú kapcsolódó adatbázistáblák helyi másolatát reprezentálja, amelyeknek mindegyike sorok és oszlopok gyűjteménye. A `DataSet` segítségével a hívószerelvény (pl. egy weboldal vagy egy asztali végrehajtható fájl) kezelheti és módosíthatja a `DataSet` tartalmát, mialatt a kapcsolata az adatforrással bontva van, és a megfelelő adatillesztőn keresztül bármilyen módosított adatot visszaküldhet további feldolgozásra.

További lényeges különbség a klasszikus ADO és az ADO.NET között az, hogy az ADO.NET messzemenően támogatja az XML-adatok megjelenítését. Tulajdonképpen az adattárolóból kinyert adatot (az alapértelmezés szerint) XML-formátumban sorosítja a rendszer. Mivel az XML-adatokat gyakran szabványos HTTP protokollon keresztül továbbítjuk a rétegek között, így az ADO.NET nem ütközik semmilyen tűzfal által okozott korlátozásba.

Talán a legalapvetőbb különbség a klasszikus ADO és az ADO.NET között az, hogy az ADO.NET nem más, mint egy felügyelt kódkönyvtár, így pontosan a felügyelt könyvtárak szabályai szerint működik. Az ADO.NET-et alkotó típusok a CLR memóriakezelési protokollját használják, ugyanazokat a típusrendszereket alkalmazzák (osztályokat, interfészeket, felsorolásokat, struktúrákat és metódusreferenciákat), és bármely .NET-nyelv által elérhetők.

Programozási szemszögből nézve az ADO.NET lényegét egy alapvető szerelvény képviseli, amelynek `System.Data.dll` a neve. A bináris fájlban jó néhány névtér megtalálható (lásd 22.1. ábra), ezek közül sok az elérhető ADO.NET-adatszolgáltató (ennek definícióját lásd alább) típusait reprezentálja.



22.1. ábra: *A System.Data.dll a legalapvetőbb ADO.NET szerelvény*

A legtöbb Visual Studio 2008 projektsablon automatikusan hivatkozik erre a kulcsfontosságú adatelérési könyvárra. Ennek ellenére módosítani kell a forráskódunkat, hogy azokat a névtereket is importáljuk, amelyeket használni szeretnénk. Például:

```
// Vezessünk be néhány ADO.NET-névteret!
using System.Data;
using System.Data.SqlClient;

namespace MyApp
{
  class Program
  {
    static void Main(string[] args)
    {
    }
  }
}
```

Az említett `System.Data.dll` szerelvényen kívül léteznek más ADO.NET-centrikus szerelvények (mint pl. a `System.Data.OracleClient.dll`), amelyekre manuálisan kell hivatkoznunk az aktuális projektünkben az Add Reference párbeszédpanel segítségével.

Megjegyzés A .NET 3.5 megjelenésével az ADO.NET számos további szerelvénnyel és névtérrel bővült, amelyek az ADO.NET/LINQ integrációt segítik. A 24. fejezetben részletesen megvizsgáljuk az ADO.NET LINQ-központú jellemzőit.

Az ADO.NET két arca

Az ADO.NET könyvtárakat két koncepcionálisan egyedi módon lehet használni: élőkapcsolat-alapú és kapcsolat nélküli módon. Ha a *kapcsolatalapú modellt* használjuk, akkor a forráskódunk expliciten csatlakozik az alatta található adattárolóhoz, és szintén expliciten bontja is a kapcsolatot. Ha ilyen megközelítésben használjuk az ADO.NET-et, akkor tipikusan a kapcsolatobjektumokon, parancsobjektumokon és adatolvasó objektumokon keresztül kommunikálunk az adattárolóval.

A kapcsolat nélküli modell (lásd a 23. fejezetben részletesen) lehetőséget biztosít arra, hogy `DataTable` objektumoknak azt a halmazát kezeljük (ezeket a `DataSet` tárolja), amelyek a külső adatok kliensoldali másolatának felelnek meg. Ha lekérünk egy `DataSet` objektumot a megfelelő adatillesztő objek-

tummal, a kapcsolat automatikusan megnyílik, majd lezáródik. Ez a megközelítés a kapcsolatok gyors felszabadítását segíti más hívók számára, és nagyban hozzájárul a rendszer skálázhatóságának tökéletesítéséhez.

Amikor a hívó megkap egy `DataSetet`, képes bejárni és kezelni a tartalmát, közben pedig nem generál hálózati forgalmat. Hasonlóképpen, amikor a hívó szeretné visszaküldeni a módosításait az adattárolónak, a rendszer az adatillesztő segítségével (SQL-utasítások egy halmazával együttműködésben) módosítja az adatforrást, ezután a kapcsolat azonnal bomlik.

Az ADO.NET-adatszolgáltatók működése

A klasszikus ADO-val ellentétben az ADO.NET nem egyetlen típushalmazt kínál fel a különböző adatbáziskezelő rendszerekkel (Database Management System - DBMS) folytatott kommunikációhoz, helyette *több adatszolgáltatót* támogat, amelyek mindegyike egy meghatározott adatbáziskezelőre van optimalizálva. A megközelítés egyik előnye, hogy ilyen módon az meghatározott adatszolgáltató hozzáférhet az adott adatbáziskezelő egyedi tulajdonságaihoz. A másik előnye az, hogy a specifikus adatszolgáltató közvetlenül csatlakozhat a mögöttes adatbázismotorhoz anélkül, hogy egy köztes leképező réteget kellene igénybe vennie.

Röviden, az *adatszolgáltató* egy adott névtérben definiált típusok egy halmaza, amely tudja, hogyan kell kommunikálnia egy konkrét adatforrással. Függetlenül attól, hogy melyik adatszolgáltatót használjuk, mindegyik egy olyan osztálytípushalmazt definiál, amely az alapvető funkcionalitást biztosítja. A 22.1. táblázat néhány alapvető objektumot mutat be, valamint a hozzájuk tartozó ősosztályt (amelyek mindegyikét a `System.Data.Common` névtér definiálja), és az általuk megvalósított adatközpontú interfészeket (amelyek mindegyikét a `System.Data` névtér definiálja).

Objektum	Ősosztály	Megvalósított interfészek	Jelentés
`Connection`	`DbConnection`	`IDbConnection`	Lehetőséget biztosít az adattárolóhoz való csatlakozásra és a kapcsolat bontására. A kapcsolatobjektumok lehetőséget biztosítanak a kapcsolódó tranzakciós objektumok eléréséhez.

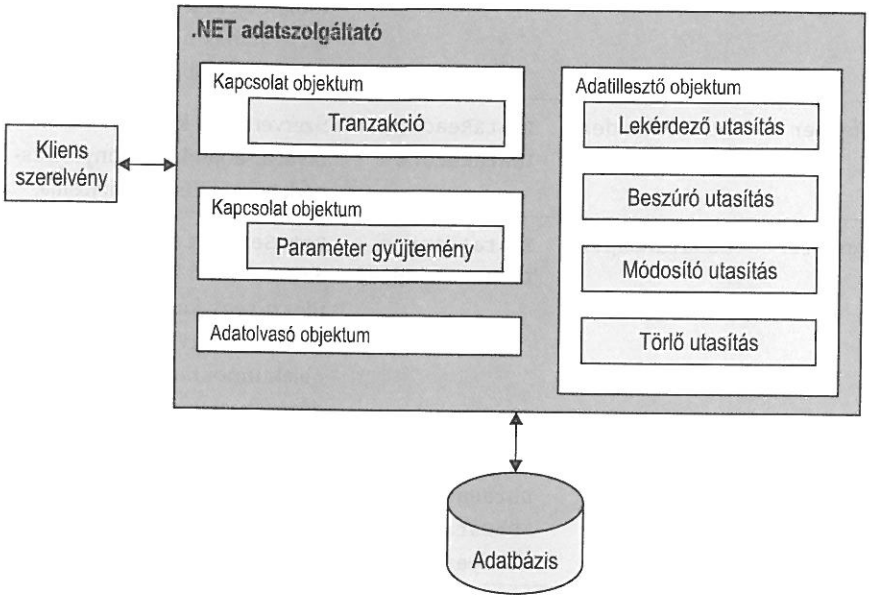
Objektum	Ősosztály	Megvalósított interfészek	Jelentés
`Command`	`DbCommand`	`IDbCommand`	Egy SQL-lekérdezést vagy egy tárolt eljárást képvisel. A kapcsolatobjektumok hozzáférést biztosítanak a szolgáltató adatolvasó objektumaihoz.
`DataReader`	`DbDataReader`	`IDataReader`, `IDataRecord`	Szerveroldali kurzor használatával az adatok egyirányú, írásvédett hozzáférését biztosítja.
`DataAdapter`	`DbDataAdapter`	`IDataAdapter`, `IDbDataAdapter`	`DataSeteket` szállít a hívó és az adattároló között. Az adatillesztő egy kapcsolatobjektumot és négy belső parancsobjektumot tartalmaz a lekérdezés, beszúrás, módosítás és törlés műveleteihez.
`Parameter`	`DbParameter`	`DataParameter`, `IDbData-Parameter`	Egy névvel ellátott paramétert képvisel egy paraméterezett lekérdezésben.
`Transaction`	`DbTransaction`	`IDbTransaction`	Beágyaz egy adatbázis-tranzakciót.

22.1. táblázat: *Egy ADO.NET adatszolgáltató alapvető objektumai*

Noha az alapvető objektumok konkrét neve eltérhet az egyes adatszolgáltatók esetén (pl. az `SqlConnection`, az `OracleConnetion`, az `OdbcConnection` és a `MySqlConnection` viszonylatában), minden objektum ugyanabból az ősosztályból származik (`DbConnection` a kapcsolatobjektumok esetén), ez pedig azonos interfészeket valósít meg. Ennek megfelelően biztosak lehetünk benne, hogy ha egy adatszolgáltató kezelését ismerjük, már nem lesz nehéz más adatszolgáltatókat használni.

Megjegyzés Az ADO.NET-ben, ha „kapcsolatobjektumról" beszélünk, akkor valóban egy konkrét `DBConnectionből` származtatott típusra hivatkozunk, és nincs olyan osztály, amelynek a neve szó szerint „Connection" lenne. Ugyanez érvényes a „parancsobjektumra", az „adatillesztő objektumra" és így tovább. A névadási konvencióknak megfelelően egy adott adatszolgáltató objektumainak a neve a kapcsolódó adatbáziskezelő nevével kezdődik (pl. `SqlConnection`, `OracleConnection`, `SqlDataReader` stb.).

A 22.2. ábra bemutatja az ADO.NET-adatszolgáltatók hátterét. A képen a „Kliens szerelvény" bármilyen .NET-alkalmazás lehet: konzolalkalmazás, Windows Forms-alkalmazás, ASP.NET-weboldal, WCF-szolgáltatás, .NET-kódkönyvtár és így tovább.



22.2. ábra: *Az ADO.NET-adatszolgáltatók adott adatbáziskezelő hozzáférését biztosítják*

Természetesen egy adatszolgáltató az ábrán látható objektumokon kívül más típusokat is tartalmaz. Ám ezek az objektumok egységes alapot képeznek az összes adatszolgáltatóban.

A Microsoft által szállított ADO.NET-adatszolgáltatók

A Microsoft .NET-disztribúciója számos adatszolgáltatóval kerül a piacra, amelyek között találunk szolgáltatót az Oracle felé, az SQL Server felé és az OLE DB/ODBC összekapcsolhatóság felé is. A 22.2. táblázat a Microsoft ADO.NET-adatszolgáltatóihoz tartozó névtereket és a befoglaló szerelvényeket sorolja fel.

Adatszolgáltató	Névtér	Szerelvény
OLE DB	`System.Data.OleDb`	`System.Data.dll`
Microsoft SQL Server	`System.Data.SqlClient`	`System.Data.dll`
Microsoft SQL Server Mobile	`System.Data.SqlServerCe`	`System.Data.SqlServerCe.dll`
ODBC	`System.Data.Odbc`	`System.Data.dll`
Oracle	`System.Data.OracleClient`	`System.Data.OracleClient.dll`

22.2. táblázat: *A Microsoft ADO.NET-adatszolgáltatói*

Megjegyzés Nincsen olyan adatszolgáltató, amelyet kifejezetten a Jet motorhoz lehetne leképezni (és ilyenformán a Microsoft Accesshez). Ha egy Access adatbázisfájt szeretnénk kezelni, akkor ezt az OLE DB vagy az ODBC adatszolgáltatóin keresztül tudjuk megtenni.

Az OLE DB adatszolgáltatóján keresztül, amelyet a `System.Data.OleDB` névtér típusai alkotnak, bármilyen adathoz hozzáférhetünk, amely a klasszikus COM-alapú OLE DB protokollt támogató adattárban található. Ennek a szolgáltatónak a használatával bármely OLE DB- vel kompatibilis adatbázissal kommunikálhatunk, ha a kapcsolatsztring Provider szegmensét beállítjuk.

Az OLE DB szolgáltató számos COM-objektummal működik együtt a színfalak mögött, és ez befolyásolhatja az alkalmazás teljesítményét. Általában elmondható, hogy csak akkor érdemes OLE DB adatszolgáltatót használni, ha olyan adatbázis-kezelővel szeretnénk dolgozni, amelyhez nincsen specifikus .NET-adatszolgáltató. Minthogy napjainkban minden valamire való relációsadatbázis-kezelőnek kell rendelkeznie saját letölthető ADO.NET-adatszolgáltatóval, így a `System.Data.OleDB` olyan korábbi névtér, amelyet igen ritkán használunk a .NET 3.5 világában (és ez egyre inkább így van a data provider factory modell megjelenésével, amely a .NET 2.0 megjelenése óta áll a rendelkezésünkre).

Megjegyzés Egy olyan eset van, amikor a `System.Data.OleDb` névtér típusainak a használatára van szükség: ha a Microsoft SQL Server 6.5 vagy valamelyik régebbi verziójával kell dolgoznunk. A `System.Data.SqlClient` névtér ugyanis csak a Microsoft SQL Server 7.0 vagy ennél magasabb verziójával tud kommunikálni.

A Microsoft SQL Server adatszolgáltató közvetlen elérést biztosít a Microsoft SQL Server adattárakhoz, és *csak* az SQL Server adattárakhoz (7.0 vagy későbbi verziókhoz). A `System.Data.SqlClient` névtér az SQL Server adatszolgáltató által használt típusokat tartalmazza, és ugyanazt az alapfunkcionalitást biztosítja, mint az OLE DB szolgáltató. A legfontosabb különbség az, hogy az SQL Server szolgáltató kikerüli az OLE DB réteget, ilyenformán számos teljesítménybeli előnyre tesz szert. A Microsoft SQL Server adatszolgáltató természetesen lehetőséget biztosít olyan egyedi tulajdonságok eléréséhez, amelyeket ez az adatbázis-kezelő nyújt.

A többi Microsoft által szállított szolgáltató (`System.Data.OracleClient`, `System.Data.Odbc` és `System.Data.SqlClientCe`) elérési lehetőséget nyújt az Oracle-adatbázisokhoz, biztosítják az ODBC-kapcsolatokkal történő kommunikációt, és elérhetővé teszik a SQL Server Mobile DBMS kiadását (általában kézieszközöknél használatos, mint pl. a Pocket PC). A `System.Data.Odbc` névtéren belül definiált ODBC-típusok jellemzően csak olyan esetben hasznosak, ha olyan relációs adatbázissal kell kommunikálnunk, amelyhez nem definiáltak egyedi .NET-adatszolgáltatót (hiszen az ODBC egy általánosan elterjedt modell, amely számos adattárolóhoz szolgáltat hozzáférést).

Harmadik féltől származó ADO.NET-adatszolgáltató beszerzése

A Microsoft által kínált adatszolgáltatókon kívül számos harmadik féltől származó adatszolgáltató létezik számos nyílt forráskódú és piaci adatbáziskezelőhöz. Noha valószínűleg közvetlenül az adatbáziskezelőt szállító cégtől fogjuk beszerezni az ADO.NET-adatszolgáltatót, érdemes megnézni a következő oldalt is (az URL változtatásának jogát a tulajdonos fenntartja): http://www.sqlsummit.com/DataProv.htm.

Ez a webhely egy azok közül a webhelyek közül, amelyek minden ismert ADO.NET-adatszolgáltatót dokumentálnak, valamint hivatkozásokat biztosítanak további információkhoz és letöltésekhez. Számos ADO.NET-adatszolgáltató megtalálható itt, többek között az SQLLite, a DB2, a MySQL, a PostgreSEL és a Sybase.

Mivel nagyon sok ADO.NET-adatszolgáltató létezik, a fejezetben található mintapéldák a Microsoft SQL Server adatszolgáltatót (`System.Data.SqlClient.dll`) használják.

Ez az adatszolgáltató a Microsoft SQL Server 7.0-val és a későbbi verziókkal - az SQL Server 2005 Express Editiont is beleértve - folytatott kommunikációt biztosítja. Ha az ADO.NET-et másmilyen adatbázis-kezelővel szeretnénk használni, akkor sem lesz ezzel problémánk a következő tudnivalók ismeretében.

További ADO.NET-névterek

A .NET-névtereken kívül, amelyek az egyes adatszolgáltatók típusait definiálják a .NET alaposztálykönyvtárai számos további ADO.NET-tel foglalkozó névteret kínálnak, amelyekből néhányat a 22.3. táblázatban láthatunk.

Névtér	Jelentés
`Microsoft.SqlServer.Server`	Ez a névtér olyan típusokat szolgáltat, amelyek segítik a CLR és az SQL Server 2005 integrációs szolgáltatásait.
`System.Data`	Ez a névtér definiálja az ADO.NET-nek azokat az alapvető típusait, amelyeket minden adatszolgáltató használ, beleértve a közös interfészeket és számos típust, amelyek a kapcsolat nélküli modellt képviselik (`DataSet`, `DataTable` stb.).
`System.Data.Common`	Ez a névtér azokat a típusokat tartalmazza, amelyeket minden ADO.NET-adatszolgáltató használ, beleértve a közös absztrakt ősosztályokat.
`System.Data.Sql`	Ez a névtér lehetővé teszi, hogy feltérképezzük az aktuális helyi hálózaton megtalálható Microsoft SQL Server példányokat.
`System.Data.SqlTypes`	Ez a névtér a Microsoft SQL Server által használt natív adattípusokat tartalmazza. Noha bármikor nyugodtan használhatjuk a megfelelő CLR-adattípusokat, az SqlTypes típusai kimondottan az SQL Serverrel működnek optimálisan.

22.3. táblázat: *További ADO.NET-tel foglalkozó névterek*

A System.Data névtér típusai

Az ADO.NET-névterek között a `System.Data` a legkisebb közös nevező. Nem is tudunk ADO.NET-alkalmazást készíteni anélkül, hogy ne hivatkoznánk erre a névtérre. Ez a névtér olyan típusokat tartalmaz, amelyet minden ADO.NET-adatszolgáltató használ, függetlenül a mögöttes adattárolótól. A számos adatbázissal kapcsolatos kivételen kívül (`NoNullAllowedException`, `RowNotInTableException`, `MissingPrimaryKeyException` és hasonlók) a `System.Data` olyan típusokat is tartalmaz, amelyek különböző adatbázis-primitíveket képviselnek (táblák, sorok, oszlopok, megszorítások stb.), valamint ugyancsak tartalmaz az adatszolgáltató objektumok által megvalósított közös interfészeket. A 22.4. táblázat néhány olyan alapvető típust sorol fel, amelyeket ismernünk kell.

Típus	Jelentés
`Constraint`	Az adott `DataColumn` objektum megszorítását képviseli.
`DataColumn`	A `DataTable` objektumon belül egy oszlopot képvisel.
`DataRelation`	Két `DataTable` objektum közötti szülő/gyermek viszonyt képviseli.
`DataRow`	A `DataTable` objektumon belül egy sort képvisel.
`DataSet`	Az adatok memóriabeli gyorsítótárát képviseli, amelyben bármennyi egymással összefüggő `DataTable` objektum található.
`DataTable`	A memóriában található adatok táblázatos formáját képviseli.
`DataTableReader`	Lehetővé teszi, hogy a `DataTable` objektumot egy tömlőkurzorként kezeljük (egyirányú, írásvédett adathozzáférés).
`DataView`	A `DataTable` objektum egy testre szabott nézetét képviseli rendezés, szűrés, keresés, módosítás és navigálás céljára.
`IDataAdapter`	Az adatillesztő objektum alapviselkedését definiálja.
`IDataParameter`	A paraméterobjektum alapviselkedését definiálja.
`IDataReader`	Az adatolvasó objektum alapműködését definiálja.
`IDbCommand`	A parancsobjektum alapműködését definiálja.
`IDbDataAdapter`	Az `IDataAdapter` interfészt bővíti ki, hogy az adatillesztő objektumot további funkcionalitásokkal lássa el.
`IDbTransaction`	A tranzakcióobjektum alapműködését definiálja.

22.4. táblázat: *A System.Data névtér alapvető tagjai*