Bill Wagner

Hatékony C#

Bill Wagner

Hatékony C#

50 módszer, hogy jobb C# kódot írhassunk

Kiskapu Kiadó
Budapest, 2005

A fordítás a következő angol eredeti alapján készült:

Bill Wagner: Effective C#: 50 Specific Ways to Improve Your C#, 1st edition,
ISBN 0321245660, Pearson Education Inc., Addison Wesley Professional

Copyright © 2005 Pearson Education, Inc. Minden jog fenntartva!

Authorized translation from the English language edition, entitled Effective C#: 50 Specific Ways to Improve Your C#, 1st edition, 0321245660, by Wagner, Bill, published by Pearson Education, Inc, publishing as Addison Wesley Professional, Copyright © 2005 by Pearson Education, Inc.

Translation and Hungarian edition © 2005 Kiskapu Kft. All rights reserved!

All rights reserved. No part of this book, including interior desing, cover design, and icons, may be reproduced or transmitted in any form, by any means (electronic, photocopying, recording, or otherwise) without the prior written permission of the publisher.

Trademarked names appear throughout this book. Rather than list the names and entities that own the trademarks or insert a trademark symbol with each mention of the trademarked name, the publisher states that it is using the names for editorial purposes only and to the benefit of the trademark owner, with no intention of infringing upon that trademark.

Fordítás és magyar változat © 2005 Kiskapu Kft. Minden jog fenntartva!

A könyv egyetlen része sem sokszorosítható semmilyen módszerrel a Kiadó előzetes írásos engedélye nélkül. Ez a korlátozás kiterjed a belső tervezésre, a borítóra és az ikonokra is.
A könyvben bejegyzett védjegyek és márkanevek is felbukkanhatnak. Ahelyett, hogy ezt minden egyes helyen külön jeleznénk, a Kiadó ezennel kijelenti, hogy a műben előforduló valamennyi védett nevet és jelzést szerkesztési célokra, jóhiszeműen, a név tulajdonosának érdekeit szem előtt tartva használja, és nem áll szándékában az azokkal kapcsolatos jogokat megszegni, vagy kétségbe vonni. A szerzők és a kiadó a lehető legnagyobb körültekintéssel járt el e kiadvány elkészítésekor. Sem a szerzők, sem a kiadó nem vállal semminemű felelősséget vagy garanciát a könyv tartalmával, teljességével kapcsolatban. Sem a szerzők, sem a kiadó nem vonható felelősségre bármilyen baleset vagy káresemény miatt, mely közvetve vagy közvetlenül kapcsolatba hozható e kiadvánnyal.

Lektor: *Rézműves László*
Fordítás: *Dr. Ottó István*
Borító: *Bognár Tamás*
Műszaki szerkesztő: *Csutak Hoffmann Levente*

Felelős kiadó a Kiskapu Kft. ügyvezető igazgatója
© 2005 Kiskapu Kft.
1081 Budapest, Népszínház u. 31. I. 7.
Telefon: (+36-1) 477-0443 Fax: (+36-1) 303-1619
www.kiskapukiado.hu
e-mail: kiado@kiskapu.hu

ISBN: 963 9301 94 9

Készült a debreceni Kinizsi Nyomdában
Felelős vezető: *Bördős János*

„Ez a kötet igazolja, hogy Bill remek programozó, és egyben remek író is. Igen röviden képes bemutatni egy problémát, megoldást kínálni rá, és levonni a következtetéseket. Valamennyi fejezet tömör és lényegre törő."

– *Josh Holmes*, független szakértő

„A könyv nagyszerű, gyakorlati szempontú bevezetést nyújt a C# nyelv elemeihez, bemutatva a legjobb fogásokat, és az alapvető nyelvi szabályoktól világosan és logikusan haladva az összetevők létrehozása és a jobb kódok írását segítő ismeretek felé. Mivel minden témakört röviden tekint át, nagyon könnyű olvasni, és gyorsan rájövünk, mennyire hasznos munka."

– *Tomas Restrepo*, Microsoft MVP

„A kötet jól mutatja be az alapokat, különösen ami azokat a döntéseket illeti, amelyeket akkor kell meghoznunk, amikor osztályokat származtatunk a `System.Object`-ből. Olvasmányos, példái világosak, tömörek és jól eltaláltak. Valószínűleg az olvasók nagy része hasznosnak találja majd."

– *Rob Steele*, Central Region Integration COE, vezető tervező, Microsoft

„A *Hatékony C#* minden eszközt megad a C#-fejlesztőnek, ami ahhoz kell, hogy Visual C# 2003-tudását gyorsan gyarapíthassa, miközben betekintést nyújt azokba a fejlesztésekbe is, amelyekkel majd a Visual C# 2005-ben találkozhatunk."

– *Doug Holland*, Precision Objects

„A .NET keretrendszer – és különösképpen a C# nyelv – egyik legfontosabb célja, hogy a fejlesztőnek lehetőséget adjon arra, hogy a megrendelő igényeivel és a termék leszállításával foglalkozhasson, ahelyett, hogy órákat (vagy éppen heteket) töltene unalmas kódolással. Bill Wagner *Hatékony C#* című kötete nem csak azt mutatja meg, mi folyik a színfalak mögött, hanem azt is, hogy miként aknázhatjuk ki a legjobban a C# nyelv egyes kódszerkezeteinek képességeit. A szerző kizárólag a tényekre összpontosít, csak azt tartja szem előtt, hogy miként írhatunk hatékony C# kódot, és ezért olyan gyakorlati példákat mutat be, amelyek tanulmányozása révén könnyebben karbantartható és gyorsabban futó C# alkalmazásokat és programösszetevőket alkothatunk. *NET Bootcamp* és más C# kurzusaim minden hallgatójának melegen ajánlom a *Hatékony C#*-et."

– *Richard Hale Shaw*, `www.RichardHaleShawGroup.com`

„A Hatékony C# nagyon jól szerkesztett, könnyen olvasható könyv, amelyben éppen megfelelő arányban keverednek a kódok a magyarázatokkal ahhoz, hogy az olvasó alapos képet kapjon a témáról. A szerző a C# nyelv és a .NET futásidejű környezet szakértője, de mindenki számára érthetően fogalmaz, és olvasmányos stílusa révén tudását maradéktalanul képes átadni az olvasóknak."

– *Brian Noyes*, az IDesign Inc. vezető szoftverfejlesztője

Tartalomjegyzék

Bevezetés xiii

1. fejezet A C# nyelv elemei

1. tipp
Mindig tulajdonságokat használjunk a hozzáférhető adattagok helyett . . . 1

2. tipp
Használjuk a readonly kulcsszót a const helyett 11

3. tipp
Használjuk az is és az as típusműveleteket a típusátalakítások helyett . . . 16

4. tipp
Használjunk Conditional jellemzőket az #if helyett 23

5. tipp
Mindig írjuk meg a ToString() tagfüggvényt 29

6. tipp
Figyeljünk az értéktípusok és a hivatkozási típusok
közti különbségre 35

7. tipp

Használjunk nem változó, elemi értéktípusokat 41

8. tipp

Gondoskodjunk róla, hogy minden értéktípusnál
érvényes legyen a 0 állapot 48

9. tipp

Értsük meg a ReferenceEquals(), a statikus Equals(),
a példány Equals() és az operator==() közti összefüggéseket 52

10. tipp

Ismerjük meg a GetHashCode() buktatóit 59

11. tipp

Használjunk foreach ciklusokat 66

2. fejezet Erőforrás-kezelés a .NET-ben

12. tipp

Értékadó utasítások helyett használjunk beállító utasításokat 75

13. tipp

Az osztály statikus tagjainak beállításához
használjunk statikus konstruktorokat 78

14. tipp

Használjuk ki a konstruktorláncolás lehetőségét 80

15. tipp

Az erőforrások felszabadításához használjuk
a using és a try/finally kulcsszavakat 85

16. tipp

Ne szemeteljünk! 92

17. tipp

Ne csomagoljunk feleslegesen! 95

18. tipp

Használjuk a szabványos felszabadító minta megvalósítását 101

3. fejezet Tervezés C# alapokon

19. tipp
Használjunk felületeket az öröklés helyett 108

20. tipp
Ne keverjük össze a felületek megvalósítását
a virtuális függvények felülbírálásával 115

21. tipp
Használjunk képviselőket a visszahívások kifejezésére 118

22. tipp
A kimenő felületeket eseményekkel határozzuk meg 120

23. tipp
Kerüljük a belső osztályobjektumokra
mutató hivatkozások visszaadását 126

24. tipp
Felszólító stílus helyett programozzunk kijelentő stílusban 130

25. tipp
Használjunk sorosítható típusokat 135

26. tipp
A rendező relációkat az IComparable
és az IComparer segítségével valósítsuk meg 143

27. tipp
Kerüljük az ICloneable felületet 148

28. tipp
Kerüljük az átalakító műveleteket 152

29. tipp
Csak akkor használjuk a new módosítót,
ha az alaposztály frissítése miatt ez feltétlenül szükséges 156

4. fejezet Bináris összetevők létrehozása

30. tipp
Használjunk CLS-megfelelő szerelvényeket 165

31. tipp
Használjunk rövid, egyszerű függvényeket 170

32. tipp
Készítsünk kisebb, összetartó szerelvényeket 174

33. tipp
Korlátozzuk a típusok láthatóságát 177

34. tipp
Ne aprózzuk el a webes API-kat 181

5. fejezet A keretrendszer használata

35. tipp
Eseménykezelők helyett használjunk felülbírált tagfüggvényeket 187

36. tipp
Használjuk ki a .NET futásidejű hibakeresési lehetőségeit 190

37. tipp
Használjuk a szabványos programbeállítási rendszert 195

38. tipp
Használjuk és támogassuk az adatkötést 199

39. tipp
Használjuk a .NET adatérvényesítést 205

40. tipp
Igazítsuk a gyűjteményeket az igényeinkhez 210

41. tipp
Használjunk DataSet objektumokat az egyedi struktúrák helyett 218

42. tipp
Használjunk jellemzőket a visszatekintés egyszerűsítéséhez 226

43. tipp

Ne vigyük túlzásba a visszatekintés használatát 232

44. tipp

Készítsünk teljes, alkalmazásfüggő kivételosztályokat 237

6. fejezet Egyebek

45. tipp

Részesítsük előnyben az erős kivételgaranciát 243

46. tipp

Bánjunk csínján az együttműködési képességekkel 248

47. tipp

Írjunk biztonságos kódot 255

48. tipp

Ismerjük meg az elérhető segédeszközöket és forrásokat 258

49. tipp

Készüljünk fel a C# 2.0 érkezésére 261

50. tipp

Ismerkedjünk meg az ECMA szabvánnyal 269

Tárgymutató 271

Bevezetés

Ennek a könyvnek az a célja, hogy a programozónak gyakorlati tanácsokat adjon arra nézve, hogy miként fokozhatja a termelékenységet, amikor a C# nyelvet és a .NET programkönyvtárakat használja. A kötet 50 kulcsfontosságú tippel szolgál, amelyek olyan témakörökhöz kapcsolódnak, amelyekkel kapcsolatban a leggyakrabban tesznek fel kérdéseket a C#-közösségben.

A C# nyelvet jómagam 10 évnyi C++-fejlesztés után kezdtem használni, és úgy tűnik, ez elég általános a C# nyelven fejlesztők között. A könyv során többek között azt vizsgálom meg, mikor okozhat gondokat az, ha a C#-ben is a C++-ban megszokott megoldásokat követjük. Vannak azonban olyan fejlesztők is, akik a C#-hez a Java felől érkeznek – számukra az említett részek nem mondanak majd túl sok újat. A Javában működő megoldások némelyike azonban a C#-ben már nem követhető, ezért a Java-fejlesztőknek azt ajánlom, fordítsanak különös figyelmet az értéktípusokról szóló részre (1. fejezet), valamint arra, hogy a .NET szemétgyűjtője másképpen viselkedik, mint a JVM-é (ezzel a 2. fejezet foglalkozik).

A könyvben tárgyalt megoldások alapját azok a javaslatok adják, amelyeket a leggyakrabban szoktam adni a fejlesztőknek. Nem mindegyik általánosan alkalmazható, de a legtöbbjük könnyen felhasználható a mindennapi programozás során, gondolok itt például a tulajdonságok (1), a feltételes fordítás (4), a nem változó típusok (7), az egyenlőség (9), az ICloneable (27) és a `new` módosító (29) tárgyalására. Tapasztalatom szerint a legtöbb esetben a programozónak az az elsődleges célja, hogy csökkentse a fejlesztésre fordított időt, és jó kódot írjon. A rendszer általános teljesítményére a legnagyobb terhet egyes tudományos és mérnöki alkalmazások róják, míg máskor a méretezhetőség a legfontosabb szempont. Céljainktól függően egyes megoldásokat hasznosabbnak, másokat lényegtelenebbnek találunk majd, ezért a célokat megpróbáltam részletesen elmagyarázni. A `readonly` és a `const` kulcsszó használatáról (2), a sorosítható típusokról

(25), a CLS-megfelelőségről (31) a webes tagfüggvényekről (34) és az adathalmazokról (41) írottak bizonyos tervezési célokat feltételeznek, amelyeket az adott tippeknél világosan körül is írok, hogy az Olvasó eldönthesse, melyik vonatkozik rá leginkább.

Bár a kötet megoldásai önállóak, átfogóbb témakörökbe (például C# nyelvtan, erőforráskezelés, objektum- és összetevő-tervezés) rendeztem őket – nem véletlenül. A célom az volt, hogy a tárgyalt megoldásokból az olvasók minél többet sajátítsanak el, ezért egymásra építettem azokat. Ettől függetlenül persze a könyv referenciaként is kiválóan használható, ha éppen egy konkrét kérdésre keressük a választ.

Ne feledjük, hogy nem nyelvi oktatóanyagról vagy kézikönyvről van szó, tehát a kötet nem a C# nyelvtanát és nyelvi szerkezeteit tanítja meg, hanem arra összpontosít, hogy a különböző helyzetekben alkalmazható lehető legjobb megoldásokkal ismertessen meg.

Kinek szánjuk a könyvet?

A *Hatékony C#* a profi fejlesztőknek íródott, tehát azoknak a programozóknak, akik a C# nyelvet napi munkájuk során használják. Feltételezi, hogy rendelkezünk tapasztalatokkal az objektumközpontú (objektumorientált) programozás terén, és jól ismerünk legalább egy nyelvet a C családjából (C, C++, C# és Java). A Visual Basic 6-ban programozóknak a könyv elolvasása előtt célszerű mind a C# nyelvtanával, mind az objektumközpontú tervezéssel megismerkedniük.

Ezen kívül nem árt, ha van tapasztalatunk a .NET főbb területein (webszolgáltatások, ADO.NET, webes és Windows-ablakok), mert ezekre a könyv során többször is hivatkozom.

Ahhoz, hogy tökéletesen megértsük a könyvben foglaltakat, értenünk kell, hogyan kezeli a .NET környezet a szerelvényeket, a Microsoft köztes nyelvét (Microsoft Intermediate Language, MSIL) és a futtatható kódot. A C# fordítóprogramja olyan szerelvényeket állít elő, amelyek tartalmazzák a MSIL-t, amelyet én gyakran IL-nek rövidítek. Amikor egy szerelvény betöltésére sor kerül, a JIT (Just In Time) fordító a MSIL-t a gép által futtatható kóddá alakítja. A C# fordító végez némi optimalizálást, de a hatékonyság komolyabb fokozását (például a helyben – inline – kifejtést) nagyrészt a JIT fordító végzi. A könyvben mindenütt elmondom, mikor melyik folyamat vesz részt az ilyen műveletekben, a kétlépéses fordítás ugyanis jelentősen befolyásolja, hogy az egyes helyzetekben mely szerkezetek nyújtják a legjobb teljesítményt.

A könyv tartalma

Az 1. fejezet (A C# nyelv elemei) a nyelv alapvető elemeit és a `System.Object` tagfüggvényeit tárgyalja, amelyek az általunk írt valamennyi típusnak részei. Amikor tehát C# kódot írunk, minden esetben a következőkkel dolgozunk: deklarációk, utasítások, algoritmusok, illetve a `System.Object` felület. Ezeken kívül a fejezetben mindazon elemekkel is

foglalkozunk, amelyek közvetlenül kapcsolódnak az értéktípusok és a hivatkozási típusok megkülönböztetéséhez, és amelyek viselkedése attól függően eltérő lehet, hogy hivatkozási típust (osztályt) vagy értéktípust (struktúrát) használunk. Mielőtt jobban belemélyednénk a könyvbe, erősen ajánlott elolvasni az érték- és hivatkozási típusokkal foglalkozó részeket (6–8).

A 2. fejezet (Erőforrás-kezelés a .NET-ben) a C# és a .NET erőforrás-kezelésével foglalkozik. Megtanuljuk, hogyan optimalizálhatjuk az erőforrások lefoglalását és a használati mintákat a .NET által kezelt végrehajtási környezetben. Igen, a .NET szemétgyűjtője egyszerűbbé teszi az életünket; a memóriakezelés a környezet felelőssége, nem a miénk, de amit mi csinálunk, az nagy hatással lehet arra, hogy mennyire jól teljesít a szemétgyűjtő az alkalmazásunkban. Ha a memória nem is okoz gondot, az egyéb erőforrásokra még mindig gondolnunk kell: ezeket az `IDisposable` felületen keresztül kezelhetjük. Tehát itt a .NET-ben követendő erőforrás-kezelési módszerekről lesz szó.

A 3. fejezet (Tervezés C# alapokon) az objektumközpontú tervezést tárgyalja, a C# szemszögéből. A C# nyelv az eszközök széles körét biztosítja; számos feladat sokféleképpen – felületek, képviselők, események, tulajdonságok vagy éppen a visszatekintő felület segítségével – megoldható vele. Az, hogy melyik megoldást választjuk, hatalmas mértékben befolyásolja, hogy rendszerünk mennyire lesz módosítható a jövőben. Ha a program felépítésének leginkább megfelelő szerkezeteket használjuk, a programozók könnyebben használatba vehetik típusainkat, a természetes ábrázolás pedig világosabbá teszi szándékainkat, így a típusok helytelen használatára kisebb lesz az esély. A 3. fejezet témái a tervezési döntéseket állítják középpontba, illetve azt, hogy a C# egyes nyelvi szerkezeteit milyen helyzetekben célszerű használni.

A 4. fejezet (Bináris összetevők létrehozása) a programösszetevőkkel és a nyelvek közötti átjárhatósággal foglalkozik. Megtanuljuk, hogyan készíthetünk olyan összetevőket, amelyeket a .NET más nyelveiben is fel lehet használni, anélkül, hogy kedvenc C#-szolgáltatásainkról le kellene mondanunk. Azt is megtudjuk, hogyan oszthatjuk fel osztályainkat összetevőkre, hogy alkalmazásaink egyes részeit frissíthessük. Ez azért fontos, mert képesnek kell lennünk arra, hogy a teljes alkalmazás újraterjesztése nélkül adjunk ki új változatokat az egyes összetevőkből.

Az 5. fejezet (A keretrendszer használata) témáját a .NET keretrendszer kevésbé kiaknázott területei jelentik. Megfigyelhető, hogy sok fejlesztő szívesebben alkot saját szoftvert, mintsem hogy a már meglevőkre építsen: ennek oka lehet, hogy a .NET keretrendszer méretében keresendő, vagy abban, hogy a keretrendszer még újnak számít. Ebben a fejezetben a keretrendszer azon részeivel foglalkozunk, amelyeknek a kiaknázása helyett a fejlesztők inkább újra feltalálják a kereket. Ha megtanuljuk, hogyan használhatjuk hatékonyabban a keretrendszert, időt takaríthatunk meg.

A 6. fejezet (Egyebek) azoknak a kérdéseknek a tárgyalásával zárja a könyvet, amelyek nem fértek be a többi kategóriába, valamint kitekint a jövőre is. Itt kell tehát keresni a C# 2.0-ra vonatkozó információkat, illetve a szabványokkal, a kivételbiztos kódokkal, a biztonsággal és a rendszerek együttműködésével kapcsolatos kérdéseket.

Néhány szó az egyes tippekről

A szándékom az, hogy világos és használható tanácsokat adjak a C# nyelvű szoftverfejlesztéshez. A kötet egyes irányelvei általánosan alkalmazhatók, mert a programhelyességre vonatkoznak, például az adattagok megfelelő előkészítésére (lásd a 2. fejezetet), mások viszont kevésbé nyilvánvalóak, és a .NET közösségen belül sok vitát gerjesztettek; ilyen például az a kérdés, hogy használjuk-e az ADO.NET `DataSet` adathalmazait. A magam részéről úgy vélem, rengeteg időt takaríthatnak meg nekünk (lásd a 41. tippet), de más profi programozók, akiket tisztelek, nem értenek egyet velem. Valójában minden attól függ, milyen programot készítünk. Álláspontom kialakításakor én az időtakarékosságot tartottam szem előtt, de azok számára, akik gyakran írnak olyan programokat, amelyek adatokat mozgatnak .NET és Java alapú rendszerek között, a `DataSet`-ek használata valóban nem jó ötlet. Javaslataimat a könyvben mindenhol megindokoltam – ha az indoklásról úgy érezzük, nem vonatkozik ránk, a javasolt megoldás sem lesz alkalmazható. Ahol a tanács általános érvényű volt, többnyire kihagytam a nyilvánvaló indokot, ami általában ez: „a program másképp nem fog működni".

Stílus és kódolás

A programozási nyelvekről szóló könyvek írása közben az egyik fő gondot az jelenti, hogy a nyelvek tervezői létező szavakat ruháznak fel különleges új jelentéssel. Emiatt olyan nehezen értelmezhető mondatok fordulhatnak elő, mint a „fejlesszünk felületeket felületekkel". A nyelvi kulcsszavakat ezért mindenütt `kód` stílussal írtuk, míg a C# egyes fogalmai nagybetűvel kezdődnek (például: „Hozzunk létre `Felületeket` az osztályaink által támogatott felület ábrázolására"). Ez a megoldás sem tökéletes, de reményeim szerint könnyebben olvashatóvá teszi a szöveget.

A könyvben számos, a C# nyelvvel kapcsolatos szakkifejezést használok. Amikor típustagokra hivatkozom, ezalatt bármilyen olyan meghatározást értek, ami része lehet egy típusnak, tehát lehet szó tagfüggvényről, tulajdonságról, mezőről, indexelőről, eseményről, felsorolásról vagy képviselőről. Ha a megállapítás ezek közül csak az egyikre vonatkozik, konkrétan megjelölöm, hogy melyikre. A kifejezések némelyike már ismerős lehet, míg mások nem; az első előfordulást **vastag** betűs kiemelés jelzi, és ott megtaláljuk a kifejezés meghatározását is.

A kötet példái rövid kódokból állnak, amelyek az adott megoldást és annak előnyeit hivatottak illusztrálni. Nem teljes kódok, amelyeket beépíthetnénk a programjainkba, tehát nem lehet egyszerűen átmásolni őket és lefordítani. Sok részlet hiányzik belőlük, a `using` záradékok jelenlétét például minden esetben adottnak tételeztem fel:

```
using System;
using System.IO;
using System.Collections;
using System.Data;
```

Ahol kevésbé közönséges névtereket használtam, ügyeltem rá, hogy a lényeges névtér látható legyen. A rövid példák teljes alakú osztályneveket tartalmaznak, míg a hosszabb példákba belefoglaltam a kevésbé nyilvánvaló `using` utasításokat.

A példákon belüli kódokat hasonló szabadsággal kezeltem. Vegyük például azt, amikor ezt írom:

```
string s1 = GetMessage();
```

Ha a tárgyalás szempontjából lényegtelen, a `GetMessage()` eljárás törzsét nem mutatom be. Amikor kihagyok egy kódrészletet, minden esetben nyugodtan feltételezhetjük, hogy a hiányzó függvény valami nyilvánvaló és értelmes dolgot csinál. Azért teszek így, mert az adott témára igyekszem összpontosítani; a kihagyott kód pedig nem része annak, és csak elvonná a figyelmet. Emellett a tippek így elég rövidek maradhatnak ahhoz, hogy egyhuzamban végig lehessen olvasni őket.

A C# 2.0

A C# közelgő 2.0-s kiadásáról nem mondok túl sokat, és erre két okom van. Először is, a könyvben bemutatott megoldások a jelenlegi és a 2.0-s változatra egyaránt érvényesek. Bár a C# 2.0 jelentős frissítés, a C# 1.0-ra épül, és ezért nem teszi érvénytelenné a mai megállapításokat. Ha mégis más lenne a követendő módszer, azt a szövegben jeleztem.

A második ok, hogy még korai lenne a C# 2.0 új szolgáltatásainak leghatékonyabb felhasználásáról írni. A könyv azokra a tapasztalatokra épül, amelyeket én és kollégáim a C# 1.0 használata során gyűjtöttünk, és egyikünk sem rendelkezik még olyan mélyreható ismeretekkel a C# 2.0-ról, hogy tudnánk, hogyan építhetők be a legjobban annak új szolgáltatásai a munkánkba. Mindezek miatt úgy gondolom, félrevezető lenne a C# 2.0-ról beszélni, amikor erre még egyszerűen nem jött el az idő.

Javaslatok, visszajelzés és frissítések

A kötet a saját tapasztalataimra és a munkatársaimmal folytatott vitákra épül. Ha az Olvasónak más tapasztalatai vannak, esetleg kérdéseket tenne fel, vagy megjegyzéseket fűzne a könyvhöz, forduljon bátran hozzám elektronikus levélben a `wwagner@srtsolutions.com` címen. A megjegyzéseket közzéteszem az Interneten (`www.srtsolutions.com/EffectiveCSharp`).

Köszönetnyilvánítás

Bár az írás magányos tevékenységnek tűnhet, a kötet elkészítésében egy egész csapat működött közre. Szerencsésnek mondhatom magam, hogy munkámat két csodálatos szerkesztő, Stephane Nakib és Joan Murray segítette. Stephane Nakib valamivel több, mint egy éve keresett meg azzal, hogy írjak az Addison-Wesley számára, de nekem kétségeim voltak. A könyvesboltok polcai tömve vannak a .NET-ről és a C#-ről szóló könyvekkel, ezért nem láttam értelmét egy újabb referenciakötetnek vagy oktatóanyagnak a témáról, amíg a C# 2.0 elég ideje nem lesz a piacon ahhoz, hogy átfogó ismertetést lehessen írni róla. Számos ötlet felmerült, és mindig oda lyukadtunk ki, hogy a leghatékonyabb megoldásokat lenne érdemes elemezni egy kötetben. Stephane-tól megtudtam, hogy Scott Meyers útnak indította az *Effective* sorozatot, saját *Effective C++* könyvei nyomán. Scott három könyvét magam is rongyosra olvastam, és minden profi C++-programozónak ajánlottam, akit csak ismerek. Scott stílusa tiszta és összeszedett, javaslatait jól megalapozott érvekkel támasztja alá. Az *Effective* könyvek nagyszerűen használhatók, a formátum pedig nagyban segíti, hogy emlékezzünk a tanácsokra. Jónéhány C++-fejlesztőt ismerek, akik lemásolták a tartalomjegyzéket, és kifüggesztették irodájuk falára, hogy mindig szem előtt legyen. Így aztán amikor Stephane megpendítette, hogy megírhatnám az *Effective C#*-et, kapva kaptam az ajánlaton. A kötet egybegyűjti az összes tanácsot, amit a C# nyelven fejlesztőknek adni szoktam. Büszke vagyok, hogy a sorozat részévé válhattam, és sokat tanultam a Scottal való munka során. Remélem, ezt a könyvet is éppen olyan hasznosnak találják majd a C#-programozók, mint amilyen hasznosnak én találtam Scott könyveit a C++-ban végzett munkához.

A kötet kialakításában Stephane nagy segítségemre volt, mind a szerkezetet, mind a szöveget illetően, amikor pedig az ő helyét Joan Murray vette át, új szerkesztőm fennakadás nélkül vitte tovább a munkát. Mindvégig segítette a szerkesztői munkát Ebony Haight, a programozói szakzsargont pedig Krista Hansing fordította hétköznapi halandók által is érthető nyelvre. A Worddokumentumok könyvvé formálásának munkáját Christy Hackerd végezte.

Ami hiba a könyvben maradt, az mind az enyém, de ezek túlnyomó többségét, valamint a kifelejtett szavakat és zavaros magyarázatokat egy nagyszerű csapat segített helyrepofozni. Brian Noyes, Rob Steel, Josh Holmes és Doug Holland nevét mindenképpen meg kell említenem: a korai vázlatokból nekik köszönhetően lett kevesebb hibát tartalmazó és hasznos szöveg. Az Ann Arbor Computing Society minden tagjának is szeretnék köszöne-

tet mondani, valamint a Great Lakes Area .NET User Groupnak, a Greater Lansing User Groupnak és a West Michigan .NET User Groupnak, akik hasznos megjegyzéseket fűztek az anyaghoz.

A könyv végleges formája legnagyobbrészt Scott Meyer részvételének köszönhető. Amikor az első vázlatokot megbeszéltem vele, akkor jöttem rá, miért is nyűttem el az ő *Effective C++* könyveit. Figyelő szemét szinte semmi nem kerüli el.

Köszönettel tartozom Andy Seidlnek és Bill Frenchnek is a MyST Technology Partnerstől (`myst-technology.com`), ugyanis a vázlatokat egy MyST alapú biztonságos webnaplóban tettem közzé. Így a munka hatékonyabbá vált, és lerövidült az egyes változatok közötti idő. Azóta a hely egyes részeit megnyitottuk a nagyközönség előtt, így a könyv egyes részei megtekinthetők az Interneten (lásd a `www.srtsolutions.com/EffectiveCSharp` címet).

Jónéhány éve írok már újságcikkeket, és szeretném megköszönni annak, aki elindított ezen a pályán, Richard Hale Shawnak. Ő biztosított nekem, a tapasztalatlan szerzőnek egy rovatot az eredeti *Visual C++ Developer's Journal*-ben, amelynek alapításában ő is részt vett. Nélküle soha nem jöttem volna rá, mennyire szeretek írni, és a kezdeti lökés nélkül, amit ő adott, nem lett volna lehetőségem írni a *Visual Studio Magazine*, a *C# Pro* vagy az *ASP.NET Pro* számára.

Munkám során a különböző folyóiratoknál számos nagyszerű szerkesztővel volt szerencsém együtt dolgozni. Szívem szerint felsorolnám mindnyájukat, de erre nem elegendő a hely. Egy valaki azonban külön említést érdemel, mégpedig Elden Nelson, aki az írói stíusomra nagy hatással volt, és akivel mindig élveztem a közös munkát.

Üzlettársaim, Josh Holmes és Dianne Marsh is köszönetet érdemelnek, amiért elnézték, hogy kevésbé vettem részt a cég életében, amíg a könyvet írtam. A kéziratokat ők is elolvasták és véleményezték, és hasznos megjegyzéseket fűztek hozzájuk.

Írás közben mindvégig szem előtt tartottam szüleim, Bill és Alice Wagner útmutatását, miszerint ha elkezdek valamit, fejezzem is be. Ez a legfőbb oka annak, amiért az Olvasó most kézben tarthatja a kész könyvet.

Mindenekfelett azonban családomnak, Marlene-nek, Larának, Sarah-nak és Scottnak tartozom köszönettel, amiért nem szűnő türelemmel viseltettek irántam az alatt a hosszú idő alatt, amíg ez a könyv elkészült.

1

A C# nyelv elemei

Miért változtassunk a jól bevált munkamódszerünkön? A válasz egyszerű: azért, hogy még jobbak legyünk. Azért cseréljük le az eszközöket és a nyelveket, mert ezáltal hatékonyabbak lehetünk. A várt eredményt azonban csak akkor érhetjük el, ha bizonyos szokásainkon változtatunk. Ez különösen nehéz, ha az új nyelv, a C#, ennyire hasonlít egy már ismert nyelvre, mint amilyen a C++ vagy a Java. Ilyenkor könnyen újra előkerülhetnek a régi beidegződések, amivel általában nincs is baj: a C# nyelvet úgy tervezték, hogy képesek legyünk felhasználni a korábban használt nyelveken szerzett ismereteinket. Néhány elemet azonban hozzáadtak vagy módosítottak, hogy jobban illeszkedjen a Common Language Runtime (CLR) futási környezethez, és hogy jobb támogatást biztosítson az összetevő alapú fejlesztéshez. Ebben a fejezetben azokról a szokásainkról lesz szó, amelyeket meg kell változtatnunk, és arról, hogy mit kell alkalmaznunk helyettük.

1. tipp

Mindig tulajdonságokat használjunk a hozzáférhető adattagok helyett

A C# nyelvben a tulajdonságok alkalmi eszközökből elsőrangú nyelvi elemekké nőtték ki magukat. Ha még mindig nyilvános változókat használunk a típusainkon belül, akkor itt az ideje, hogy leszokjunk erről. Ha még mindig magunk írjuk meg a `get` és a `set` tagfüggvényeket, akkor itt az ideje, hogy erről is leszokjunk. A tulajdonságok segítségével az adattagokat úgy tehetjük elérhetővé a nyilvános felületeken keresztül, hogy közben megőrizzük az objektumközpontú környezetben elvárt betokozást (encapsulation). A **tulajdonságok** olyan nyelvi elemek, amelyeket adattagokként érhetünk el, de megvalósításuk tagfüggvényként történik.

A típusok tagjai közül néhányat leginkább adatként érdemes ábrázolni: ilyen egy vásárló neve, egy pont x,y koordinátái vagy éppen a tavalyi jövedelmünk. A tulajdonságokkal olyan felületeket hozhatunk létre, amelyek adatelérési pontként működnek, de a függvények min-

den előnyével is rendelkeznek. Az ügyfélkód úgy érheti el a tulajdonságokat, mintha azok nyilvános változók lennének. A tényleges megvalósítás azonban tagfüggvényeket használ, amelyekben mi határozzuk meg a tulajdonságokat elérő elemek viselkedését.

A .NET keretrendszer feltételezi, hogy a nyilvános adattagok helyett tulajdonságokat használunk. A .NET keretrendszer adatkapcsoló kódot tartalmazó osztályai csak a tulajdonságokat támogatják, a nyilvános adattagokat nem. Az adatkapcsolás az objektumok tulajdonságait kapcsolja hozzá a felhasználói felület vezérlő elemeihez, ami webes vagy Windows Forms vezérlő lehet. Az adatkapcsolási rendszer visszatekintéssel (reflection) találja meg a típus névvel rendelkező tulajdonságait:

```
textBoxCity.DataBindings.Add( "Text",
  address, "City" );
```

A fenti kód a `textBoxCity` vezérlő `Text` tulajdonságát az `address` objektum `City` nevű tulajdonságához kapcsolja. (A további részleteket lásd a 38. tippnél.) Egy `City` nevű nyilvános adattaggal nem működne. A keretrendszer osztálykönyvtárának (Framework Class Library) tervezői nem támogatták ezt a gyakorlatot; a nyilvános adattagok használatát rossz szokásnak tartották. Ez a döntésük még inkább arra kell hogy ösztönözzön bennünket, hogy a helyes objektumközpontú megoldásokat alkalmazzuk. Hadd tegyek egy megjegyzést azon C++ és Java programozók kedvéért, akiknek a hajkoronája őszbe fordult az imént: az adatkapcsoló kód nem keresi a `get` és a `set` függvényeket sem. Ezeknek a nyelveknek a hagyományos `get_` és `set_` függvényei helyett is tulajdonságokat kell használnunk.

Igen, az adatkapcsolás csak azokra az osztályokra vonatkozik, amelyek olyan elemeket tartalmaznak, amelyek megjelennek a felhasználói felületen. Ez azonban nem jelenti azt, hogy csak a felhasználó felületekhez használhatunk tulajdonságokat. Más osztályokkal és szerkezetekkel is bátran használhatjuk őket. A tulajdonságok módosítása sokkal egyszerűbb, ha az idők során új követelmények merülnek fel, vagy új jelenségekbe ütközünk. Előfordulhat, hogy rájövünk, hogy a vevőt megtestesítő típusnál a névnek sohasem szabad üresen maradnia. Ha a `Name` nyilvános tulajdonság, akkor ezt könnyedén elintézhetjük egyetlen módosítással:

```
public class Customer
{
  private string _name;
  public string Name
  {
    get
    {
      return _name;
    }
    set
    {
```

```
        if (( value == null ) ||
          ( value.Length == 0 ))
          throw new ArgumentException( "Name cannot be blank",
            "Name" );
        _name = value;
      }
    }

    // ...
  }
```

Ha nyilvános adattagokat használtunk volna, akkor most töviről-hegyire átnézhetnénk a kódot a vevő nevét módosító részek után kutatva, és minden egyes helyen el kellene végeznünk a módosítást, ami bizony eltartana egy ideig. Nagyon, nagyon, nagyon hosszú ideig.

Mivel a tulajdonságok megvalósítása tagfüggvényekkel történik, sokkal egyszerűbben támogathatjuk a többszálas működést. Egyszerűen a `get` és a `set` tagfüggvények megvalósításának bővítésével adjunk lehetőséget az adatok összehangolt elérésére:

```
public string Name
{
  get
  {
    lock( this )
    {
      return _name;
    }
  }
  set
  {
    lock( this )
    {
      _name = value;
    }
  }
}
```

A tulajdonságok a tagfüggvények összes nyelvi szolgáltatásával rendelkeznek. A tulajdonságok lehetnek virtuálisak:

```
public class Customer
{
  private string _name;
  public virtual string Name
  {
    get
    {
```

```
      return _name;
    }
    set
    {
      _name = value;
    }
  }

  // a további megvalósítás elhagyva
}
```

Könnyen belátható, hogy a tulajdonságok elvontak, sőt akár egy felület részei is lehetnek:

```
public interface INameValuePair
{
  object Name
  {
    get;
  }

  object Value
  {
    get;
    set;
  }
}
```

Végül, de nem utolsósorban, a tulajdonságok segítségével létrehozhatjuk a felületek `const` és `nonconst` típusú változatait is:

```
public interface IConstNameValuePair
{
  object Name
  {
    get;
  }

  object Value
  {
    get;
  }
}
public interface INameValuePair
{
  object Value
  {
    get;
```

```
    set;
  }
}

// Használat:
public class Stuff : IConstNameValuePair, INameValuePair
{
  private string _name;
  private object _value;

  #region IConstNameValuePair Members
  public object Name
  {
    get
    {
      return _name;
    }
  }

  object IConstNameValuePair.Value
  {
    get
    {
      return _value;
    }
  }

  #endregion

  #region INameValuePair Members

  public object Value
  {
    get
    {
      return _value;
    }
    set
    {
     _value = value;
    }
  }
  #endregion
}
```

A tulajdonságok teljes értékű, elsőrangú nyelvi elemek, amelyek olyan tagfüggvények bővítései, amelyekkel belső adatokat érhetünk el és módosíthatunk. Bármit, amit megtehetünk tagfüggvényekkel, azt megtehetjük tulajdonságokkal is.

A tulajdonságelérők két önálló függvény, amelyeket a nyelv a típus részeként fordít le. A C# 2.0-ban a tulajdonságok `get` és `set` elérőihez más elérésmódosítókat is megadhatunk. Ezzel még pontosabban szabályozhatjuk azoknak az adatelemeknek a láthatóságát, amelyeket tulajdonságként teszünk elérhetővé:

```
// Szabályos C# 2.0:
public class Customer
{
  private string _name;
  public virtual string Name
  {
    get
    {
      return _name;
    }
    protected set
    {
      _name = value;
    }
  }

  // a további megvalósítás elhagyva
}
```

A tulajdonságok nyelvtana túlmutat az egyszerű adatmezőkön. Ha egy típus felületének indexelt elemeket kell tartalmaznia, akkor használhatjuk az **indexelőket** (vagyis a paraméterrel rendelkező tulajdonságokat). Ez olyan tulajdonságok létrehozásánál hasznos, amelyek valamilyen sorrendben adnak vissza elemeket:

```
public int this [ int index ]
{
  get
  {
    return _theValues [ index ] ;
  }
  set
  {
    _theValues[ index ] = value;
  }
}

// hozzáférés egy indexelőhöz
int val = MyObject[ i ];
```

Az indexelők is élvezik mindazt a nyelvi támogatást, amit az egyedülálló tulajdonságok. Megvalósításuk éppen úgy történik, mint az általunk írt tagfüggvényeké, tehát bármilyen ellenőrzést és számítást elvégezhetünk az indexelők belsejében. Az indexelők lehetnek

virtuálisak és elvontak, bevezethetők felületekben, és lehetnek csak olvashatók vagy írhatók-olvashatók is. Az egydimenziós, számparaméterrel rendelkező indexelők részt vehetnek adatkapcsolásban. Más indexelők nem egész paramétereket használva térképeket és szótárakat határozhatnak meg:

```
public Address this [ string name ]
{
  get
  {
    return _theValues[ name ] ;
  }
  set
  {
    _theValues[ name ] = value;
  }
}
```

A C# nyelv többdimenziós tömbjeihez hasonlóan többdimenziós indexelőket is készíthetünk, minden tengelyen hasonló vagy eltérő típusokkal:

```
public int this [ int x, int y ]
{
  get
  {
    return ComputeValue( x, y );
  }
}

public int this[ int x, string name ]
{
  get
  {
    return ComputeValue( x, name );
  }
}
```

Figyeljük meg, hogy mindegyik indexelőt a `this` kulcsszó segítségével vezettük be. Az indexelőknek nem adhatunk nevet, ezért minden típusban ugyanazzal a paraméterlistával csak egy indexelő létezhet.

Ez az egész tulajdonságos működés szép és jó, és komoly fejlődés. De még mindig csábító lehet, hogy kezdetben adattagokat használva hozzunk létre egy megvalósítást, és majd csak később cseréljük le az adattagokat tulajdonságokra, amikor szükségünk lesz azok előnyeire. Ez nem hangzik rossz ötletnek, pedig az.

Nézzük meg például az alábbi részletet egy osztály meghatározásából:

```
// nyilvános adattagok használata - rossz szokás:
public class Customer
{
  public string Name;

  // a további megvalósítás elhagyva
}
```

Az osztály egy névvel rendelkező vásárlót ír le. A nevet a már ismerős tagjelöléssel olvashatjuk ki vagy adhatjuk meg:

```
string name = customerOne.Name;
customerOne.Name = "This Company, Inc.";
```

Ez így pofonegyszerű. Azt gondolnánk, hogy később lecserélhetjük a `Name` adattagot egy tulajdonságra, és a kód továbbra is működni fog. Nos, ez csak nagyjából igaz.

A tulajdonságokat úgy tervezték meg, hogy az adattagokhoz hasonlítsanak, amikor megpróbáljuk elérni őket. Ez a célja az új nyelvtannak. A tulajdonságok azonban nem adatok. A hozzáférés egy tulajdonsághoz nem ugyanazt az MSIL kódot eredményezi, mintha egy adathoz férnénk hozzá. Az előbbi vásárlótípus az alábbi MSIL kódot eredményezi a `Name` mezőhöz:

```
.field public string Name
```

A mező elérése az alábbi utasításokat eredményezi:

```
ldloc.0
ldfld      string NameSpace.Customer::Name
stloc.1
```

Ha egy értéket mentünk a mezőbe, akkor ezt kapjuk:

```
ldloc.0
ldstr      "This Company, Inc."
stfld      string NameSpace.Customer::Name
```

Ne aggódjunk, nem fogunk egész álló nap IL kódot nézegetni, de itt azért fontos ez, mert mindjárt látni fogjuk, hogyan teszi tönkre a bináris megfelelőséget, ha tulajdonságra cserélünk egy adattagot.

Nézzük meg a vásárlótípusnak az alábbi változatát, amit tulajdonságokkal hoztunk létre:

```
public class Customer
{
  private string _name;
  public string Name
  {
    get
    {
      return _name;
    }
    set
    {
      _name = value;
    }
  }

  // a további megvalósítás elhagyva
}
```

C# nyelven a name tulajdonság elérése éppen úgy történik, ahogy az előbb:

```
string name = customerOne.Name;
customerOne.Name = "This Company, Inc.";
```

A C# fordító azonban teljesen más MSIL kódot állít elő ehhez a változathoz. A Customer típusból ez lesz:

```
.property instance string Name()
{
  .get instance string NameSpace.Customer::get_Name()
  .set instance void NameSpace.Customer::set_Name(string)
} // end of property Customer::Name

.method public hidebysig specialname instance string
        get_Name() cil managed
{
  // Code size       11 (0xb)
  .maxstack  1
  .locals init ([0] string CS$00000003$00000000)
  IL_0000:  ldarg.0
  IL_0001:  ldfld      string NameSpace.Customer::_name
  IL_0006:  stloc.0
  IL_0007:  br.s       IL_0009
  IL_0009:  ldloc.0
  IL_000a:  ret
} // end of method Customer::get_Name
```

```
.method public hidebysig specialname instance void
        set_Name(string 'value') cil managed
{
  // Code size       8 (0x8)
  .maxstack  2
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  stfld      string NameSpace.Customer::_name
  IL_0007:  ret
} // end of method Customer::set_Name
```

Két lényeges dolog van, amit meg kell értenünk a tulajdonságok meghatározásainak MSIL-re fordításával kapcsolatban. Először is, a `.property` utasítás meghatározza a tulajdonság típusát és azokat a függvényeket, amelyek megvalósítják a tulajdonság `get` és `set` elérőit. A két függvény a `hidebysig` és a `specialname` jelölést kapta. Nekünk most elég annyi, hogy ezek a jelölések azt jelentik, hogy ezeknek a függvényeknek a meghívása nem közvetlenül a C# forráskódban történik, és hogy nem szabad őket a formális típusmeghatározás részeként kezelni. Ehelyett a tulajdonságon keresztül érhetjük el őket.

Arra persze számítottunk, hogy a tulajdonság leírására más MSIL kód születik. Ami azonban jóval fontosabb, hogy a `get` és a `set` hozzáférését leíró MSIL is megváltozott:

```
// get
ldloc.0
callvirt    instance string NameSpace.Customer::get_Name()
stloc.1

// set
ldloc.0
ldstr       "This Company, Inc."
callvirt    instance void NameSpace.Customer::set_Name(string)
```

Attól függően, hogy a `Name` tag tulajdonság vagy adattag, ugyanaz a C# forráskód, amivel a vásárló nevéhez hozzáférünk, más-más MSIL kódot eredményez. A tulajdonságok és az adattagok eléréséhez ugyanazt a C# forráskódot használjuk. A C# fordító feladata, hogy a forrást különböző IL kódokra fordítsa, attól függően, hogy tulajdonságot vagy adattagot használtunk.

Bár a tulajdonságok és az adattagok a forrás szintjén csereszabatosak, a bináris kód szintjén nem azok. Ez nyilvánvalóan azt jelenti, hogy amikor egy nyilvános adattagról átváltunk egy vele egyenértékű tulajdonságra, akkor az összes olyan kódot újra kell fordítani, ami használta a nyilvános adattagot. A 4. fejezetben (*Bináris összetevők létrehozása*) részletesen szólunk majd a bináris összetevőkről, de ne feledjük, hogy a bináris megfelelőséget már azzal az egyszerű művelettel is tönkretehetjük, ha egy adattagot egy tulajdonságra cserélünk. Ez rendkívül megnehezíti a korábban már kiadott egyedi szerelvények frissítését.

Miközben vizsgálgattuk a tulajdonsághoz tartozó IL kódot, valószínűleg elgondolkoztunk a tulajdonságok és az adattagok egymáshoz viszonyított teljesítményén. A tulajdonságok nem lesznek gyorsabbak az adattagoknál, de lassabbak sem kell, hogy legyenek. A JIT fordító valóban beszúr néhány tagfüggvényhívást, beleértve a tulajdonságelérők hívását. Amikor a JIT beszúrja a tulajdonságok elérését, akkor az adattagok és a tulajdonságok teljesítménye megegyezik. És még ha nem is szúrja be a JIT fordító a tulajdonságok elérését, a tényleges teljesítménybeli különbség egy elhanyagolható függvényhívás lesz. Ez a különbség csak az esetek kis részében mérhető.

Ha a típusok nyilvános vagy védett felületében teszünk elérhetővé adatokat, akkor mindig használjunk tulajdonságokat. Sorozatokhoz és szótárakhoz használjunk indexelőket. Az adattagok kivétel nélkül legyenek privátak. Így rögtön támogatást kapunk az adatkapcsoláshoz, és később sokkal könnyebben végezhetünk módosításokat a tagfüggvényeken. Az a kis plusz gépelés, amivel a tulajdonságban lévő összes változó elrejtése jár, legfeljebb egy-két percet vesz el a napi munkaidőnkből. Ha csak később jövünk rá, hogy tulajdonságokat kellene használnunk az adott feladathoz, az több órányi munkát jelenthet. Ha tehát idejében szánunk erre egy kis időt, azzal később sok-sok fáradságot takaríthatunk meg.

2. tipp

Használjuk a readonly kulcsszót a const helyett

A C# nyelv kétféle állandót ismer: a **fordítási idejű állandókat** és a **futásidejű állandókat**. Ezek viselkedése jelentősen eltér egymástól, és ha nem a megfelelőt használjuk, az a programunk teljesítményének és helyes működésének rovására mehet. Egyik sem kellemes, de ha muszáj választani, akkor egy lassabb, de helyesen működő program még mindig jobb, mint egy gyors program, ami hibás. Ennélfogva mindig részesítsük előnyben a futásidejű állandókat a fordítási idejű állandókkal szemben. A fordítási idejű állandók kissé gyorsabbak, de lényegesen rugalmatlanabbak, mint a futásidejű állandók. Csak akkor használjunk fordítási idejű állandókat, ha a teljesítmény kulcsfontosságú, és ha az állandó értéke bizonyosan nem változik meg a program futása közben.

A futásidejű állandókat a `readonly`, a fordítási idejű állandókat pedig a `const` kulcsszóval vezethetjük be:

```
// fordítási idejű állandó:
public const int _Millennium = 2000;

// futásidejű állandó:
public static readonly int _ThisYear = 2004;
```

A fordítási idejű és a futásidejű állandók működése közti eltérés az elérésük módjából adódik. A fordítási idejű állandók helyére az állandó értéke kerül a tárgykódban.

Az alábbi szerkezet

```
if ( myDateTime.Year == _Millennium )
```

éppen azt az IL kódot eredményezi, mintha ezt írtuk volna:

```
if ( myDateTime.Year == 2000 )
```

A futásidejű állandók kiértékelése futásidőben történik. Ha egy csak olvasható állandóra hivatkozunk, akkor a létrejövő IL kód a `readonly` típusú változóra hivatkozik, és nem az értékére.

Ez a különbség számos megkötéssel jár arra nézve, hogy mikor melyik típusú állandót használhatjuk. A fordítási idejű állandókat csak az alapvető típusokkal (a beépített egész és lebegőpontos típusokkal), a felsorolásokkal és karakterláncokkal használhatjuk. Kizárólag ezekhez a típusokhoz adhatunk meg értelmes állandóértékeket kezdőértékként, csak ezeket az egyszerű típusokat lehet literálokra cserélni a fordító által előállított IL kódban. Az alábbi szerkezetet nem lehet lefordítani, ugyanis fordítási idejű állandónak nem adhatunk kezdőértéket a `new` függvénnyel, még akkor sem, ha a típusa egy értéktípus:

```
// Nem lehet lefordítani - használjuk a readonly kulcsszót helyette:
private const DateTime _classCreation = new
  DateTime( 2000, 1, 1, 0, 0, 0 );
```

A fordítási idejű állandók csak számok és karakterláncok lehetnek. A csak olvasható értékek abból a szempontból szintén állandók, hogy a konstruktor futása után már nem lehet módosítani őket. A csak olvasható értékek viszont annyiban mások, hogy futásidőben rendelődnek az állandóhoz. A futásidejű állandók tehát sokkal rugalmasabban kezelhetők, már csak azért is, mert bármilyen típusúak lehetnek. Egy konstruktoron belül kell kezdőértéket adni nekik, de használhatunk előkészítő (inicializáló) utasítást is. A `DateTime` szerkezetekből készíthetünk `readonly` állandókat, `const` típusúakat viszont nem.

A `readonly` értékek lehetnek példányállandók, amelyek egy osztálytípus minden példányához különböző értékeket tárolnak. A fordítási idejű állandók a meghatározásukból következően mindig statikus állandók.

A legfontosabb különbség, hogy a `readonly` állandók feloldása futásidőben történik. Ha egy `readonly` állandóra hivatkozunk, akkor a létrejövő IL kód a `readonly` típusú változóra hivatkozik, és nem az értékére. Ennek messzemenő következményei vannak a későbbi karbantartást illetően. A fordítási idejű állandókból ugyanaz az IL kód áll elő, mintha magát a számértéket használtuk volna a kódunkban. Ez akár több különböző szerelvényre is igaz. Ha egy szerelvényben használunk egy állandót, annak helyére akkor is az adott érték kerül, ha az állandót egy másik szerelvényben is használtuk.

A fordítási idejű és a futásidejű állandók kiértékelésének módja a futásidejű megfelelőségre is hatással van. Tegyük fel, hogy egy `Infrastructure` nevű szerelvényben megadtunk egy `const` és egy `readonly` típusú mezőt is:

```
public class UsefulValues
{
  public static readonly int StartValue = 5;

  public const int EndValue = 10;
}
```

Egy másik szerelvényben pedig hivatkozunk ezekre az értékekre:

```
for ( int i = UsefulValues.StartValue;
  i < UsefulValues.EndValue;
  i++ )
  Console.WriteLine( "value is {0}", i );
```

Ha lefuttatjuk a kis tesztünket, akkor az alábbi egyértelmű kimenetet kapjuk:

```
Value is 5
Value is 6
...
Value is 9
```

Majd eltelik egy kis idő, és kiadjuk az `Infrastructure` szerelvény új változatát, a következő módosításokkal:

```
public class UsefulValues
{
  public static readonly int StartValue = 105;

  public const int EndValue = 120;
}
```

Elkezdjük terjeszteni az `Infrastructure` szerelvényt, anélkül, hogy újrafordítanánk az `Application` szerelvényt. Azt gondolnánk, hogy az eredmény a következő lesz:

```
Value is 105
Value is 106
...
Value is 119
```

Valójában azonban semmilyen kimenetet nem kapunk, a ciklus ugyanis most a `105` értéket fogja induló értékként használni, kilépési feltétele pedig a `10` lesz. A C# fordító a `const` típusú `10` értéket helyezi az `Application` szerelvénybe, vagyis nem az `EndValue` értékét tároló helyre mutató hivatkozást. Hasonlítsuk össze ezt a `StartValue` értékkel, amit `readonly` típusúként vezettünk be, vagyis a feloldása futásidőben történik. Az `Application` szerelvény ezért anélkül is az új értéket fogja használni, hogy újrafordítanánk. Elég, ha telepítjük az `Infrastructure` szerelvény új változatát, és máris minden olyan ügyfélkód viselkedése megváltozik, amely használja az értéket. A nyilvános állandók frissítését felületváltozásként kell értékelnünk, ami azt jelenti, hogy minden olyan kódot újra kell fordítanunk, ami hivatkozik az adott értékre. Egy `readonly` típusú állandó értékének módosítása a megvalósításban bekövetkezett változásnak tekintendő, ami binárisan összeegyeztethető a már létező ügyfélkóddal. Ha megvizsgáljuk az előző ciklushoz tartozó MSIL kódot, akkor pontosan láthatjuk, hogy miért van ez így:

```
IL_0000:  ldsfld     int32 Chapter1.UsefulValues::StartValue
IL_0005:  stloc.0
IL_0006:  br.s       IL_001c
IL_0008:  ldstr      "value is {0}"
IL_000d:  ldloc.0
IL_000e:  box        [mscorlib]System.Int32
IL_0013:  call       void [mscorlib]System.Console::WriteLine
    (string,object)
IL_0018:  ldloc.0
IL_0019:  ldc.i4.1
IL_001a:  add
IL_001b:  stloc.0
IL_001c:  ldloc.0
IL_001d:  ldc.i4.s   10
IL_001f:  blt.s      IL_0008
```

Látható, hogy a `StartValue` értéket dinamikusan tölti be a program az MSIL lista elején. A kilépési feltétel `10`-es értéke azonban ott van bedrótozva az MSIL kód végén.

Persze az is előfordul, hogy már a fordításkor meg akarjuk határozni egy állandó értékét. Tegyük fel például, hogy van egy sor állandónk, amelyek egy objektum különböző változatszámait jelzik sorban egymás után (lásd a 25. tippet). Az egyes változatokat jelző állandóértékekként fordítási idejű állandókat válasszunk, hiszen az értékük sohasem változik. Az aktuális változatszám azonban futásidejű állandó lesz, amelynek értéke minden egyes kiadásnál más és más lesz.

```
private const int VERSION_1_0 = 0x0100;
private const int VERSION_1_1 = 0x0101;
private const int VERSION_1_2 = 0x0102;
```

```
// új kiadás:
private const int VERSION_2_0 = 0x0200;

// ellenőrizzük az aktuális változatszámot
private static readonly int CURRENT_VERSION =
  VERSION_2_0;
```

A futásidejű változattal raktározhatjuk el az aktuális változatszámot minden mentett fájlba:

```
// Olvasunk az állandó tárolóból, és összehasonlítjuk
// a tárolt változatszámot a fordítási idejű állandóval:
protected MyType( SerializationInfo info,
  StreamingContext cntxt )
{
  int storedVersion = info.GetInt32( "VERSION" );
  switch ( storedVersion )
  {
  case VERSION_2_0:
    readVersion2( info, cntxt );
    break;
  case VERSION_1_1:
    readVersion1Dot1( info, cntxt );
    break;

  // stb.
  }
}

// Kiírjuk az aktuális változatszámot:
[ SecurityPermissionAttribute( SecurityAction.Demand,
  SerializationFormatter =true ) ]
void ISerializable.GetObjectData( SerializationInfo inf,
    StreamingContext cxt )
{
  // A futásidejű állandót használjuk az aktuális változatszámhoz
  inf.AddValue( "VERSION", CURRENT_VERSION );

  // Kiírjuk a többi elemet...
}
```

A `const` utolsó előnye a `readonly` típussal szemben a teljesítmény. Az ismert állandóértékekből valamivel hatékonyabb kódot lehet előállítani, mint amit a `readonly` típusú állandókhoz szükséges változó-hozzáférés engedne. Az így nyert sebességnövekedés azonban igen csekély, és mérlegelnünk kell, hogy megéri-e azt, hogy a program ezáltal veszít rugalmasságából. Ezért mindig végezzünk elemzést a teljesítményt illetően, mielőtt feláldoznánk a rugalmasságot.

A `const` kulcsszót akkor kell használni, ha az értéknek már fordításkor elérhetőnek kell lennie: jellemzőparamétereknél, felsorolás-meghatározásoknál, és azokban a ritka esetekben, amikor egy olyan értéket akarunk megadni, ami nem változik a program újabb és újabb kiadásai során. Minden más esetben használjuk a sokkal rugalmasabb `readonly` állandókat.

3. tipp

Használjuk az is és az as típusműveleteket a típusátalakítások helyett

A C# erősen típusos nyelv. A helyes programozási gyakorlat azt is jelenti, hogy amennyiben elkerülhető, nem próbáljuk meg az egyik típust átpasszírozni egy másikba. Néha viszont mindenképpen szükséges a típusok futásidőben való ellenőrzése. A C# nyelvben sokszor írunk olyan függvényeket, amelyek `System.Object` típusú paramétereket kapnak, mivel a keretrendszer meghatározza helyettünk a tagfüggvény aláírását. Ezeket az objektumokat aztán valószínűleg át kell alakítanunk valamilyen más típusra, ami lehet egy osztály vagy egy felület. Ezt kétféleképpen tehetjük meg. Vagy az `as` típusművelettel, vagy a jó öreg C nyelvben megszokott hagyományos típusátalakítással. Van még egy óvatos megoldás is. Ellenőrizhetjük az átalakítást az `is` kulcsszóval, majd az `as` kulcsszóval vagy egy típusátalakítással elvégezhetjük a műveletet.

A helyes választás mindig az `as`, ha a használatára lehetőségünk van, mivel sokkal biztonságosabb, mint egy vakon elvégzett típusátalakítás, és futásidőben sokkal hatékonyabb is. Az `as` és az `is` kulcsszavak nem végeznek felhasználó által meghatározott átalakításokat. Csak akkor járnak sikerrel, ha a futásidőben kapott típus megegyezik a kívánt típussal. Új szerkezeteket sosem hoznak létre azért, hogy eleget tegyenek egy ilyen kérésnek.

Nézzünk meg egy példát. Írunk egy kis kódrészletet, aminek az a feladata, hogy egy tetszőleges objektumot átalakítson a `MyType` objektum egy példányává. Ezt megtehetjük például így:

```
object o = Factory.GetObject( );

// Első változat:
MyType t = o as MyType;

if ( t != null )
{
  // t egy MyType példány, tehát használhatjuk
} else
{
  // jelezzük a hibát
}
```

Vagy írhatjuk ezt is:

```
object o = Factory.GetObject( );

// Második változat:
try {
  MyType t;
  t = ( MyType ) o;
  if ( t != null )
  {
    // t egy MyType példány, tehát használhatjuk
  } else
  {
    // jelezzük a null típusra hivatkozást
  }
} catch
{
  // jelezzük az átalakítás sikertelenségét
}
```

Bizonyára egyetértünk abban, hogy az első változat egyszerűbb, és sokkal jobban követhető. Nem kell hozzá a `try` és a `catch`, így nincs az ezzel járó többletmunka és a plusz kód. Figyeljük meg, hogy a hagyományos típusátalakítást használó változatban még a `null` hivatkozásokra is figyelnünk kell a kivételek elfogása mellett. A `null` bármilyen hivatkozási típusra átalakítható egy típusátalakítással, az `as` viszont `null` értéket ad vissza, amikor egy `null` hivatkozással használjuk. A hagyományos típusátalakításnál tehát figyelnünk kell a `null` hivatkozásokra, és a kivételeket is el kell fognunk. Az `as` kulcsszóval csak azt kell ellenőriznünk, hogy `null`-e a visszaadott hivatkozás.

A legnagyobb különbség az `as` típusművelet és a hagyományos típusátalakítás között a felhasználó által meghatározott típusátalakításoknál jelentkezik. Az `as` és az `is` műveletek mindössze annyit tesznek, hogy futásidőben megvizsgálják az átalakítandó objektum típusát. Más művelet végrehajtására nem alkalmasak. Ha egy adott objektum nem a kívánt típusú, vagy ha a típusnak csak egy leszármazottja, akkor sikertelen lesz a vizsgálat. A hagyományos típusátalakítással viszont bármely objektumot átalakíthatunk a kívánt típusúra. Ugyanez áll a számok közötti, beépített típusátalakításokra is. Ha egy `long` típusú értéket `short` típusúra alakítunk, akkor az információvesztéssel járhat. Hasonló veszélyek leselkednek ránk, ha egy általunk létrehozott típusra szeretnénk átalakítani egy értéket. Vegyük például az alábbi típust:

```
public class SecondType
{
  private MyType _value;

  // az egyéb részleteket mellőztük
```

```
  // Átalakító művelet
  // Egy SecondType típust
  // MyType típusúra alakít, lásd a 29. tippet
  public static implicit operator
    MyType( SecondType t )
  {
    return t._value;
  }
}
```

Tegyük fel, hogy az első kódrészletben a `Factory.GetObject()` függvény egy `SecondType` típusú objektumot ad vissza:

```
object o = Factory.GetObject( );

// Az o SecondType típusú:
MyType t = o as MyType; // sikertelen, az o nem MyType típusú

if ( t != null )
{
  // t egy MyType példány, tehát használhatjuk
} else
{
  // jelezzük a hibát
}

// Második változat:
try {
  MyType t1;
  t = ( MyType ) o; // sikertelen, az o nem MyType típusú
  if ( t1 != null )
  {
    // t1 egy MyType példány, tehát használhatjuk
  } else
  {
    // jelezzük a null típusra hivatkozást
  }
} catch
{
  // jelezzük az átalakítás sikertelenségét
}
```

Egyik változat sem működik, pedig azt mondtuk, hogy a hagyományos típusátalakítással elvégezhetők a felhasználó által megadott átalakítások. Azt gondolhatnák tehát, hogy a hagyományos módszer működik, és ezt a logikát követve valóban működnie kellene. De nem fog, mert a fordító az o objektum fordítási idejű típusa alapján hozza létre a kódot.

A fordító nem tudja, hogy mi lesz az o típusa futásidőben, és egy System.Object példánynak veszi. A fordító látja, hogy nem létezik a felhasználó által meghatározott átalakítás a System.Object típusról a MyType típusra. Megnézi a System.Object és a MyType típusok meghatározását. Mivel nem talál a felhasználó által meghatározott átalakítást a típusok között, a fordító által előállított kód futásidőben megvizsgálja az o típusát, és ellenőrzi, hogy MyType típusú-e. Persze nem az lesz, mivel az o egy SecondType típusú objektum. A fordító azt nem ellenőrzi, hogy az o tényleges típusa futásidőben átalakítható-e MyType típusra.

Ha az alábbi módon írnánk meg a kódrészletet, akkor sikerrel járna a SecondType típus átalakítása MyType típusra:

```
object o = Factory.GetObject( );

// Harmadik változat:
SecondType st = o as SecondType;
try {
  MyType t;
  t = ( MyType ) st;
  if ( t != null )
  {
    // t egy MyType példány, tehát használhatjuk
  } else
  {
    // jelezzük a null típusra hivatkozást
  }
} catch
{
  // jelezzük az átalakítás sikertelenségét
}
```

Ilyen visszataszító kódot persze sosem írunk, de itt jól szemléltet egy gyakran előforduló gondot. Ugyan az alábbi kódot sem írnánk meg soha, de ha egy függvény számol a megfelelő átalakításokkal, akkor ahhoz megadhatunk egy System.Object paramétert:

```
object o = Factory.GetObject( );

DoStuffWithObject( o );

private void DoStuffWithObject( object o2 )
{
  try {
    MyType t;
    t = ( MyType ) o2; // sikertelen, az o nem MyType típusú
    if ( t != null )
    {
      // t egy MyType példány, tehát használhatjuk
```

```
    } else
    {
      // jelezzük a null típusra hivatkozást
    }
  } catch
  {
    // jelezzük az átalakítás sikertelenségét
  }
}
```

Ne feledjük, hogy a felhasználói típusátalakító műveletek az objektumok fordítási idejű típusával képesek csak dolgozni, a futásidejű típusokkal nem. Nem számít, hogy létezik futásidejű átalakítás az `o2` és a `MyType` típusok között. A fordító ezt egyszerűen nem tudja, és nem is érdekli. Az alábbi utasítás másképp viselkedik attól függően, hogy milyen típusúként vezettük be az `st` változót.

```
t = ( MyType ) st;
```

A következő utasítás viszont mindenképpen ugyanazt eredményezi, függetlenül az `st` típusától. Tehát mindig az `as` kulcsszót részesítsük előnyben a hagyományos típusátalakításokkal szemben, mert annak működése sokkal következetesebb. Ami azt illeti, ha a típusok között nincs öröklődési kapcsolat, de létezik egy felhasználói típusátalakítás, akkor az alábbi utasítás fordítói hibát eredményez:

```
t = st as MyType;
```

Most, hogy már tudjuk, hogyan kell használni az `as` kulcsszót, amikor ez lehetséges, eljött az idő, hogy azt is megbeszéljük, hogy mikor nem használhatjuk. Az `as` típusművelet értéktípusokkal nem működik. Az alábbi utasítást tehát nem tudjuk lefordítani:

```
object o = Factory.GetValue( );
int i = o as int; // Fordítási hibához vezet
```

Ez azért van, mert az egészek értéktípusok, tehát sosem lehet `null` az értékük. Milyen értéket adnánk az `i` változónak, ha `o` nem egész? Bármilyen értéket is válasszunk, az éppen úgy lehet egy érvényes egész szám is, ezért aztán az `as` ilyenkor nem használható. Marad a hagyományos típusátalakítás:

```
object o = Factory.GetValue( );
int i = 0;
try {
  i = ( int ) o;
} catch
{
  i = 0;
}
```

A típusátalakítások viselkedését azonban nem feltétlenül kell megőriznünk. Az is kulcsszót használva kivédhetjük a kivételeket és az átalakításokat:

```
object o = Factory.GetValue( );
int i = 0;
if ( o is int )
  i = ( int ) o;
```

Ha az o típusát nem lehet egésszé alakítani, mert például double típusú, akkor az is művelet hamis (false) értéket ad vissza. Az is a null argumentumoknál mindig hamis értéket ad vissza.

Az is műveletet csak akkor használjuk, ha a típust az as művelettel nem lehet átalakítani. Különben a használata felesleges:

```
// helyes, de felesleges
object o = Factory.GetObject( );

MyType t = null;
if ( o is MyType )
  t = o as MyType;
```

A fenti kódrészlet pont ugyanaz, mintha ezt írtuk volna:

```
// helyes, de felesleges
object o = Factory.GetObject( );

MyType t = null;
if ( ( o as MyType ) != null )
  t = o as MyType;
```

Ez így nem túl hatékony, és teljesen felesleges. Ha az as használatával akarunk átalakítani egy típust, akkor az is művelettel elvégzett összehasonlításra egyszerűen semmi szükség. Sokkal egyszerűbb, ha ellenőrizzük, hogy nem null-e az as által visszaadott érték.

Most, hogy már ismerjük az is, az as, és a hagyományos típusátalakítások közti különbségeket, próbáljuk meg kitalálni, hogy melyiket használják a foreach ciklusok:

```
public void UseCollection( IEnumerable theCollection )
{
  foreach ( MyType t in theCollection )
    t.DoStuff( );
}
```

A `foreach` egy típusátalakítás segítségével alakítja át az objektumokat a ciklusban használt típusra. A `foreach` által előállított kód megegyezik az alábbi változattal:

```
public void UseCollection( IEnumerable theCollection )
{
  IEnumerator it = theCollection.GetEnumerator( );
  while ( it.MoveNext( ) )
  {
    MyType t = ( MyType ) it.Current;
    t.DoStuff( );
  }
}
```

A `foreach` azért használ típusátalakítást, mert támogatnia kell az értéktípusokat és a hivatkozási típusokat is. A típusátalakítás használata miatt a `foreach` utasítás mindig ugyanúgy viselkedik, a céltípustól függetlenül, viszont éppen ezért előfordlhat, hogy a `foreach` ciklus kivált egy `BadCastException` kivételt.

Mivel az `IEnumerator.Current` egy `System.Object` típust ad vissza, amihez nem tartozik típusátalakító művelet, semmi nem tesz eleget az összehasonlításnak. Egy `SecondType` objektumokból álló gyűjteményt sem használhatunk a fenti `UseCollection()` függvényben, mert amint azt már láttuk, az átalakítás sikertelen lesz. A `foreach` utasítás (ami típusátalakítást használ) nem vizsgálja a gyűjteményben lévő objektumok futásidejű típusát. Csak a `System.Object` osztályhoz (vagyis az `IEnumerator.Current` által visszaadott típushoz) tartozó típusátalakításokat vizsgálja, illetve a ciklusváltozó megadott típusát veszi figyelembe (ebben az esetben ez a `MyType`).

Befejezésül, néha előfordul, hogy egy objektum pontos típusára vagyunk kíváncsiak, és nem csak arra, hogy a jelenlegi típusát át lehet alakítani egy adott céltípusra. Az `as` típusművelet a céltípusból származtatott bármely típusra igaz (`true`) értéket ad vissza. A `GetType()` tagfüggvény az objektumok futásidejű típusát adja vissza. Ez sokkal szigorúbb ellenőrzést hajt végre, mint az `is` és az `as` műveletek. A `GetType()` az objektum típusát adja vissza, amit aztán összehasonlíthatunk egy adott típussal.

Nézzük meg ismét ezt a függvényt:

```
public void UseCollection( IEnumerable theCollection )
{
  foreach ( MyType t in theCollection )
    t.DoStuff( );
}
```

Ha létrehoznánk egy `NewType` osztályt, ami a `MyType` leszármazottja, akkor egy `NewType` objektumokból álló gyűjtemény szépen működne a `UseCollection` függvényben:

```
public class NewType : MyType
{
  // contents elided.
}
```

Ha olyan függvényt akarunk írni, ami a `MyType` összes példányával működik, akkor ez rendben is van. Ha viszont olyan függvényt akarunk, ami kizárólag a `MyType` objektumokkal működik, akkor a pontos típust kell használnunk az összehasonlításnál. Ebben az esetben ezt a `foreach` ciklus belsejében tehetnénk meg. A leggyakoribb eset, amikor fontos a pontos futásidejű típus, ha egy azonosságot ellenőrzünk (lásd a 9. tippet). A legtöbb más esetben az `as` és az `is` által nyújtott `.isinst` összehasonlítások szemantikailag helytállóak.

A helyes objektumközpontú programozás ugyan azt is jelenti, hogy kerülnünk kell a típusok közti átalakításokat, de néha nincs más választásunk. Amikor ez elkerülhetetlen, akkor mindig az `as` és az `is` típusműveleteket használjuk, hogy a szándékaink minél világosabbak legyenek. A típusok átalakítására több módszer is létezik, és ezeknek eltérő szabályai vannak. A leghelyesebb szinte kivétel nélkül az `is` és az `as` műveletek használata. Ezek a műveletek csak akkor járnak sikerrel, ha az adott objektum a megfelelő típusú. Tehát ha lehet, ezeket részesítsük előnyben a hagyományos típusátalakító műveletekkel szemben, amelyeknek számtalan akaratlan mellékhatása lehet, és olyankor járhatnak sikerrel vagy vallhatnak kudarcot, amikor a legkevésbé számítunk rá.

4. tipp

Használjunk Conditional jellemzőket az #if helyett

Az `#if/#endif` blokkokat máig használják arra, hogy a forráskódból különböző változatokat építsenek fel, a leggyakrabban hibakeresés vagy valamilyen programváltozat létrehozásának céljából. Ezekkel az eszközökkel azonban sosem volt öröm a munka. Az `#if/#endif` blokkok használatát könnyű túlzásba vinni, ami nehezebben érthető kódot eredményez, amiben a hibák felkutatása elég nehézkes. A programnyelvek tervezői olyan tökéletesebb eszközök létrehozásával próbálnak megfelelni ennek a kihívásnak, amelyek a különböző környezetekhez különböző gépi kódot állítanak elő. A C# nyelvben a `Conditional` jellemző megjelenése jelenti ezt a fejlődést, ami azt jelzi, hogy meghívható-e egy tagfüggvény az adott környezettől függően. Ez sokkal rendezettebb lehetőséget nyújt a feltételes fordításra, mint az `#if/#endif` utasítások. A fordító ismeri a `Conditional` jellemzőt, így jobban el tudja végezni a kód ellenőrzését, amikor valamilyen feltételt alkalmazunk. A feltételes jellemzőket a tagfüggvények szintjén kell alkalmaznunk, így rákényszerülünk, hogy külön tagfüggvényekbe tegyük a feltételhez kötött kódot. Ha valamilyen feltételhez szeretnénk kötni egy kódrészletet, akkor az `#if/#endif` utasítások helyett mindig használjuk a `Conditional` jellemzőt.

A legtöbb veterán programozóval már előfordult, hogy feltételes fordítással ellenőrzött elő- és utófeltételeket egy objektumban. Általában írtunk egy privát tagfüggvényt, amivel ellenőriztük az osztályok és objektumok állandóit. Ezt a tagfüggvényt aztán feltételesen fordítottuk, hogy csak a hibakereséshez készült változatokban jelenjen meg.

```
private void CheckState( )
{
// A régi módszer:
#if DEBUG
  Trace.WriteLine( "Entering CheckState for Person" );

  // Elcsípjük a hívó eljárás nevét:
  string methodName =
    new StackTrace( ).GetFrame( 1 ).GetMethod( ).Name;

  Debug.Assert( _lastName != null,
    methodName,
    "Last Name cannot be null" );

  Debug.Assert( _lastName.Length > 0,
    methodName,
    "Last Name cannot be blank" );

  Debug.Assert( _firstName != null,
    methodName,
    "First Name cannot be null" );

  Debug.Assert( _firstName.Length > 0,
    methodName,
    "First Name cannot be blank" );

  Trace.WriteLine( "Exiting CheckState for Person" );
#endif
}
```

Az `#if` és az `#endif` utasítások segítségével egy üres tagfüggvényt készítettünk a végleges változatokhoz. A `CheckState()` tagfüggvényt minden változatban, tehát a hibakeresésre szolgáló és a végleges változatokban is meghívja a program. Bár a végleges változatokban semmit nem csinál, a tagfüggvény meghívása időt vesz igénybe. Csekély, de további árat fizetünk az üres eljárás betöltéséért és JIT fordításáért is.

Ez a fajta gyakorlat általában jól működik, de alig észrevehető hibákhoz vezethet a végleges változatokban.

Az alábbi gyakori hiba megmutatja, hogy mi történhet, ha direktívákat használunk a feltételes fordításhoz:

```
public void Func( )
{
  string msg = null;

#if DEBUG
  msg = GetDiagnostics( );
#endif
  Console.WriteLine( msg );
}
```

A hibakereső változatban minden tökéletesen működik, a végleges változatokban azonban a program kiír egy üres sort. A kész változat tehát vidáman kiírja az üres üzenetet, annak ellenére, hogy ez nem állt szándékunkban. Hülyeséget csináltunk, és ezen a fordító sem tudott segíteni. Olyan kódot tettünk egy feltételes blokkba, ami alapvető részét képezi a gondolatmenetünknek. Ha teletűzdeljük a kódunkat `#if/#endif` blokkokkal, akkor nehezen lehet majd követni, hogyan viselkednek a program különböző változatai.

A C# nyelv jobb megoldást is kínál erre a problémára: a `Conditional` jellemzőt. A `Conditional` jellemző segítségével megjelölhetjük azokat a függvényeket, amelyeknek csak akkor kell az osztályainkban szerepelniük, amikor egy adott környezeti változó meghatározása létezik, vagy egy bizonyos értéket kap. Ezt a szolgáltatást leginkább arra használjuk, hogy hibakereső utasításokat helyezzünk el a kódunkban. A .NET keretrendszer könyvtára már tartalmazza ehhez az alapvető szolgáltatásokat. Az alábbi példa bemutatja, hogyan használhatjuk ki a .NET keretrendszer könyvtárának hibakereső képességeit, és látni fogjuk a `Conditional` jellemzők működését, és azt, hogy mikor és hova kell a kódunkba illeszteni őket.

A `Person` objektum fordításakor beszúrunk egy tagfüggvényt, ami ellenőrzi az objektum értékeit:

```
private void CheckState( )
{
  // Elcsípjük a hívó eljárás nevét:
  string methodName =
    new StackTrace( ).GetFrame( 1 ).GetMethod( ).Name;

  Trace.WriteLine( "Entering CheckState for Person:" );
  Trace.Write( "\tcalled by " );
  Trace.WriteLine( methodName );

  Debug.Assert( _lastName != null,
    methodName,
    "Last Name cannot be null" );
```

```
  Debug.Assert( _lastName.Length > 0,
    methodName,
    "Last Name cannot be blank" );

  Debug.Assert( _firstName != null,
    methodName,
    "First Name cannot be null" );

  Debug.Assert( _firstName.Length > 0,
    methodName,
    "First Name cannot be blank" );

  Trace.WriteLine( "Exiting CheckState for Person" );
}
```

Előfordulhat, hogy a fenti tagfüggvényben használt könyvtári függvényekkel még nem találkoztunk, ezért szaladjunk végig rajtuk gyorsan. A `StackTrace` osztály visszatekintéssel (lásd a 43. tippet) kapja meg a hívó eljárás nevét. Elég költséges megoldás, de nagyon leegyszerűsíti az olyan feladatokat, mint amikor a program menetével kapcsolatos információkat kell előállítanunk. Itt annak a tagfüggvénynek a nevét deríti ki, amelyik meghívta a `CheckState()` függvényt. A többi tagfüggvény vagy a `System.Diagnostics.Debug` vagy a `System.Diagnostics.Trace` osztályhoz tartozik. A `Debug.Assert` tagfüggvény ellenőriz egy feltételt, és ha a feltétel teljesül, akkor leállítja a programot. A további paramétereiben azokat az üzeneteket adhatjuk meg, amelyeket akkor ír ki a program, ha a feltétel hamisnak bizonyul. A `Trace.WriteLine` hibaüzeneteket ír ki a hibakereső ablakba. Ez a tagfüggvény tehát üzeneteket ír ki, és leállítja a programot, ha a program érvénytelen `Person` objektummal találkozik. Ezt a tagfüggvényt minden nyilvános tagfüggvénybe és tulajdonságba bele kellene írnunk elő- és utófeltételként:

```
public string LastName
{
  get
  {
    CheckState( );
    return _lastName;
  }
  set
  {
    CheckState( );
    _lastName = value;
    CheckState( );
  }
}
```

A `CheckState()` jelenti, amint valaki egy üres karakterláncot vagy egy `null` értéket ad meg családnévként. Ekkor kijavítjuk a `set` elérőt, hogy ellenőrizze a `LastName` értékét. Minden megy annak rendje és módja szerint.

A minden nyilvános eljárásban végrehajtott kiegészítő ellenőrzés azonban időbe telik, pedig igazán csak a hibakereséshez készült programváltozatokban van szükségünk rá. Itt jön el a `Conditional` jellemző ideje:

```
[ Conditional( "DEBUG" ) ]
private void CheckState( )
{
  // ugyanaz a kód, mint az előbb
}
```

A `Conditional` jellemzőt látva a C# fordító tudja, hogy csak akkor kell meghívni ezt a tagfüggvényt, amikor a fordító érzékeli a `DEBUG` környezeti változót. A `Conditional` jellemző nem befolyásolja a `CheckState()` függvényből előállított kódot, csak a függvény hívását szabályozza. Ha megadtuk a `DEBUG` szimbólumot, akkor ezt kapjuk:

```
public string LastName
{
  get
  {
    CheckState( );
    return _lastName;
  }
  set
  {
    CheckState( );
    _lastName = value;
    CheckState( );
  }
}
```

Ha nem, akkor pedig ezt:

```
public string LastName
{
  get
  {
    return _lastName;
  }
  set
  {
    _lastName = value;
  }
}
```

A CheckState() függvény törzse mindig ugyanaz lesz, a környezeti változó állapotától függetlenül. Ez jó példa arra, hogy miért kell tisztában lennünk a sima fordítás és a JIT fordítás közti különbséggel a .NET keretrendszerben. A CheckState() tagfüggvény fordítása megtörténik, és az a szerelvény részévé válik, attól függetlenül, hogy megadtuk-e a DEBUG környezeti változót vagy sem. Lehet, hogy ez nem tűnik hatékony megoldásnak, de csak lemezhellyel kell fizetnünk érte. A CheckState() függvény nem töltődik a memóriába, és a JIT fordítására sem kerül sor, hacsak meg nem hívjuk. Az, hogy ott van a szerelvényben, teljesen lényegtelen. Ez a megközelítés növeli a program rugalmasságát, és szinte nem kerül semmibe a teljesítményt illetően. Ha mélyrehatóbban szeretnénk megismerkedni a módszerrel, akkor nézzük meg a .NET keretrendszer Debug osztályát. A System.dll szerelvényben ott van a Debug osztály összes tagfüggvényének a kódja az összes olyan gépen, amire telepítjük a .NET keretrendszert. Azt, hogy egy program meghívja-e őket, már a környezeti változók szabályozzák a hívó kód fordításakor.

Olyan tagfüggvényeket is létrehozhatunk, amelyek több környezeti változótól is függnek. Ha egyszerre több feltételes jellemzőt alkalmazunk, akkor az OR kulcsszóval kombinálhatjuk azokat. Az alábbi CheckState() függvény hívására például akkor kerül sor, ha vagy a DEBUG, vagy a TRACE értéke igaz:

```
[ Conditional( "DEBUG" ),
  Conditional( "TRACE" ) ]
private void CheckState( )
```

Ha az AND kulcsszóval akarunk létrehozni egy szerkezetet, akkor magunknak kell meghatároznunk az előfeldolgozó számára a szimbólumot a forráskódban, az előfeldolgozó direktíváinak segítségével:

```
#if ( VAR1 && VAR2 )
#define BOTH
#endif
```

Igen, ahhoz, hogy létrehozzunk egy olyan feltételes eljárást, ami egynél több környezeti változó meglétén alapul, vissza kell nyúlnunk a korábban használt, kiérdemesült #if utasításhoz. Az #if direktívát azonban itt mindössze egy új szimbólum meghatározásához használjuk. Végrehajtandó kódot továbbra se helyezzünk az utasítás belsejébe.

A Conditional jellemzőt csak teljes tagfüggvényekre lehet alkalmazni. Kikötés még, hogy a Conditional jellemzővel ellátott tagfüggvények visszatérési értéke nem lehet más, mint void. Tagfüggvények belsejében lévő kódblokkokkal és olyan tagfüggvényekkel, amelyek valamilyen értéket adnak vissza, nem használhatjuk a Conditional jellemzőt. Ehelyett mindig figyelmesen tervezzük meg a feltételes tagfüggvényeket, és a feltételes viselkedéseket különítsük el ezekbe a függvényekbe. Ettől persze még át kell néznünk

ezeket a feltételes tagfüggvényeket, hogy ne legyen semmilyen mellékhatásuk az objektumok állapotára, de a Conditional jellemzővel sokkal hatékonyabban találhatjuk meg ezeket a pontokat, mint az #if/#endif utasításokkal. Az #if és az #endif blokkokkal akaratlanul is fontos függvényhívásokat és értékadásokat távolíthatunk el a programból.

A fent bemutatott példák az előre meghatározott DEBUG és TRACE szimbólumokat használták, de a módszert kiterjeszthetjük bármilyen általunk megadott szimbólumra is. A Conditional jellemzőt számos különböző módon meghatározott szimbólummal szabályozhatjuk. A szimbólumokat meghatározhatjuk a fordító parancssorában, az operációs rendszer héjának környezeti változóival, vagy a forráskódban elhelyezett direktívákkal.

A Conditional jellemző hatékonyabb IL kódot állít elő, mint az #if/#endif utasítások. További előnye, hogy kizárólag a tagfüggvények szintjén alkalmazható, ami rákényszeríti a programozót a feltételes kód szerkezetének gondos megtervezésére. A fordító a Conditional jellemzővel segít bennünket, hogy elkerüljük azokat a gyakori hibákat, amelyeket mindnyájan elkövettünk már, amikor rossz helyre tettünk egy #if vagy egy #endif utasítást. A Conditional jellemző sokkal hatékonyabban támogatja a feltételes kód különválasztását, mint ahogyan azt az előfeldolgozó tette.

5. tipp

Mindig írjuk meg a ToString() tagfüggvényt

A System.Object.ToString() a .NET környezet egyik leggyakrabban használt tagfüggvénye. Illik megírni egy elfogadható változatát az osztályunk összes ügyfele számára. Máskülönben az osztályt felhasználókat arra kényszerítjük, hogy az osztály tulajdonságait használva maguk készítsék el az osztály emberek számára is érthető ábrázolását. A típus szöveges ábrázolásával megkönnyítjük az információk kiírását egy objektumról a felhasználóknak a Windows ablakokban, a webes űrlapokon, illetve a konzolon. A szöveges ábrázolás a hibakeresés szempontjából is hasznos lehet. Készítsük el minden típusnál ennek a függvénynek a felülbírálatát. Ha bonyolultabb típusokat hozunk létre, akkor készítsük el az IFormattable.ToString() megvalósítását. Legyünk őszinték: ha nem bíráljuk felül ezt az eljárást, vagy ha egy gyenge változatot készítünk, azzal az ügyfeleinkre hárítjuk a megoldást.

A System.Object változat a típus nevét adja vissza. Ez elég haszontalan információ, hiszen nem azt akarjuk kiírni a felhasználóknak, hogy "Rect", "Point" vagy "Size". Viszont ha nem bíráljuk felül a ToString() tagfüggvényt az osztályainkban, akkor pontosan ezt kapjuk. Egy osztályt csak egyszer kell megírnunk, az ügyfeleink viszont számtalanszor fogják használni azt. Ha egy kicsivel több időt fordítunk az osztály megírására, az minden alkalommal megtérül, amikor mi magunk vagy valaki más használni fogja.

Nézzük a legegyszerűbb elvárást, a `System.Object.ToString()` tagfüggvény felülbírálását. Minden létrehozott típusnak tartalmaznia illik a `ToString()` felülbírálatát, hogy az megadja a típus leggyakoribb szöveges leírását. Vegyünk például egy `Customer` nevű osztályt, aminek három mezője van:

```
public class Customer
{
  private string  _name;
  private decimal _revenue;
  private string  _contactPhone;
}
```

Az `Object.ToString()` öröklött változata a `"Customer"` karakterlánccal tér vissza. Ennek senki nem veszi sok hasznát. Ennél még akkor is komolyabbnak kell lennie a `ToString()` tagfüggvénynek, ha csak hibakeresésre használjuk. Az `Object.ToString()` tagfüggvény felülbírált változatának azt a szöveges leírást kell visszaadnia, amire az objektum ügyfeleinek valószínűleg a leginkább szükségük lesz. A `Customer` esetében ez valószínűleg a megrendelő neve:

```
public override string ToString()
{
  return _name;
}
```

Ha mást nem is fogadunk meg ebből a tippből, ezt a gyakorlatot mindenképpen alkalmazzuk az összes típusnál, amit meghatározunk. Ezzel mindenkinek időt takarítunk meg. Ha megírjuk az `Object.ToString()` egy elfogadható megvalósítását, akkor ennek az osztálynak az objektumait sokkal könnyebben adhatjuk hozzá Windows ablakelemekhez, webes űrlapelemekhez, vagy nyomtatott kimenethez. A .NET FCL könyvtár az `Object.ToString()` felülbírált változatát használja az objektumok megjelenítéséhez a különféle vezérlőkben: a lenyíló listákban, a listamezőkben, a szövegmezőkben és más elemekben is. Ha létrehozunk egy megrendelő objektumokból álló listát egy Windows ablakban vagy egy webes űrlapon, akkor a név jelenik meg az elem szövegeként. Mindezt a `System.Console.WriteLine()`, a `System.String.Format()` és a `ToString()` belseje végzi el. Ha a .NET FCL egy megrendelő nevére kíváncsi, akkor a megrendelő típusunk megadja az adott megrendelő nevét. Ezeknek az alapvető igényeknek egy egyszerű, háromsoros tagfüggvény segítségével teszünk eleget.

Ez az egyszerű tagfüggvény, a `ToString()`, sok olyan kívánalom kielégítésére képes, ahol a felhasználói típust szövegként kell megjeleníteni. Néha azonban ennél többre van szükség. Az előbbi megrendelő típusnak három mezője volt: név, jövedelem, és egy telefonszám. A `System.ToString()` felülbírálata ezek közül csak a nevet használja. Ezt a hiányosságot azzal orvosolhatjuk, ha a típusunkhoz elkészítjük az `IFormattable` felület

megvalósítását. Az IFormattable felületnek van egy túlterhelt ToString() tagfüggvénye, amivel formázási információkat adhatunk meg a típushoz. Ezt a felületet használjuk, amikor különböző formátumú szöveges kimenetet kell előállítanunk. A megrendelő osztály jó példa erre. A felhasználók valószínűleg el akarnak majd készíteni egy olyan jelentést, amiben a megrendelők neve mellett ott szerepel a tavalyi jövedelmük is táblázatos formában. Az IFormattable.ToString() tagfüggvénnyel olyan eszközt kapunk a kezünkbe, amivel lehetővé tehetjük a felhasználók számára, hogy a típusból származó szöveges információt megformázzák. Az IFormattable.ToString() tagfüggvény aláírása egy formátumleíró karakterláncból és egy formátum-szolgáltatóból áll:

```
string System.IFormattable.ToString( string format,
  IFormatProvider formatProvider )
```

A formátumleíró karakterlánccal saját formátumokat adhatunk meg az általunk létrehozott típusokhoz. A formátumleírókhoz saját helyettesítő karaktereket is megadhatunk. A megrendelő esetében az n karakterrel jelölhetjük a nevet (name), az r karakterrel a jövedelmet (revenue), illetve a p karakterrel a telefonszámot (phone). Ha azt is megengedjük, hogy a felhasználó ezek kombinációját is használja, akkor például az alábbi módon készíthetjük el az IFormattable.ToString() függvényt:

```
#region IFormattable Members
// támogatott formátumok:
// n legyen a név
// r legyen a jövedelem
// p legyen a telefonszám
// a kombinációkat is támogatjuk: nr, np, npr stb.
// a "G" az általános eset (general)
string System.IFormattable.ToString( string format,
  IFormatProvider formatProvider )
{
  if ( formatProvider != null )
  {
    ICustomFormatter fmt = formatProvider.GetFormat(
      this.GetType( ) )
      as ICustomFormatter;
    if ( fmt != null )
      return fmt.Format( format, this, formatProvider );
  }

  switch ( format )
  {
    case "r":
      return _revenue.ToString( );
    case "p":
      return _contactPhone;
    case "nr":
```

```
        return string.Format( "{0,20}, {1,10:C}",
          _name, _revenue );
      case "np":
        return string.Format( "{0,20}, {1,15}",
          _name, _contactPhone );
      case "pr":
        return string.Format( "{0,15}, {1,10:C}",
          _contactPhone, _revenue );
      case "pn":
        return string.Format( "{0,15}, {1,20}",
          _contactPhone, _name );
      case "rn":
        return string.Format( "{0,10:C}, {1,20}",
          _revenue, _name );
      case "rp":
        return string.Format( "{0,10:C}, {1,20}",
          _revenue, _contactPhone );
      case "nrp":
        return string.Format( "{0,20}, {1,10:C}, {2,15}",
          _name, _revenue, _contactPhone );
      case "npr":
        return string.Format( "{0,20}, {1,15}, {2,10:C}",
          _name, _contactPhone, _revenue );
      case "pnr":
        return string.Format( "{0,15}, {1,20}, {2,10:C}",
          _contactPhone, _name, _revenue );
      case "prn":
        return string.Format( "{0,15}, {1,10:C}, {2,15}",
          _contactPhone, _revenue, _name );
      case "rpn":
        return string.Format( "{0,10:C}, {1,15}, {2,20}",
          _revenue, _contactPhone, _name );
      case "rnp":
        return string.Format( "{0,10:C}, {1,20}, {2,15}",
          _revenue, _name, _contactPhone );
      case "n":
      case "G":
      default:
        return _name;
    }
  }
  #endregion
```

Ennek a függvénynek a megírásával lehetővé tesszük az ügyfeleink számára, hogy ők maguk határozzák meg, hogy miként kívánják megjeleníteni a megrendelőik adatait:

```
IFormattable c1 = new Customer();
Console.WriteLine( "Customer record: {0}",
  c1.ToString( "nrp", null ) );
```

Az `IFormattable.ToString()` megvalósítása az adott típustól függ, de van néhány eset, amit mindig figyelembe kell vennünk az `IFormattable` felület megírásánál. Először is, mindig támogatnunk kell az általános, `"G"` formátumot. Másodszor, támogatnunk kell az üres formátumot, mindkét változatában: az egyik a `""`, a másik a `null`. Mindhárom formátumleírónak ugyanazt a karakterláncot kell visszaadnia, amit az `Object.ToString()` tagfüggvény felülbírált változatának. A .NET FCL az `IFormattable.ToString()` tagfüggvényt hívja meg az `Object.ToString()` helyett az összes olyan típusnál, amelyiknél létezik az `IFormattable` megvalósítása. A .NET FCL általában egy üres formátumleíróval hívja meg az `IFormattable.ToString()` függvényt, de vannak olyan helyek is, ahol a `"G"` formátumleírót használják az általános formátum jelölésére. Ha támogatjuk az `IFormattable` felületet, de megfeledkezünk ezekről a szabványos formátumokról, azzal elrontjuk az FCL automatikus karakterlánc-átalakításait.

Az `IFormattable.ToString()` második paramétere egy objektum, ami az `IFormatProvider` felület megvalósítása. Ez az objektum lehetővé teszi az ügyfelek számára, hogy olyan formázási lehetőségeket is megadjanak, amelyeket mi nem láthattunk előre. Ha megnézzük az `IFormattable.ToString()` előbbi megvalósítását, akkor valószínűleg számtalan olyan formázási lehetőséget ki tudunk találni, ami jó lenne, de nincs megadva. Az emberi olvasásra szánt kimenet természete már csak ilyen. Nem számít, hogy hány formátumot támogatunk, a felhasználónak egy szép napon egyszer csak pont egy olyan formátumra lesz szüksége, amire nem gondoltunk. Ezért van az, hogy a tagfüggvény első néhány sora egy olyan objektum után kutat, ami tartalmazza az `IFormatProvider` megvalósítását, majd továbbadja a feladatot az `ICustomFormatter` objektumának.

Most pedig hagyjuk egy kicsit az osztály szerzőjét, és képzeljük magunkat az osztály „fogyasztójának" helyébe. Tegyük fel, hogy rájövünk, hogy olyan formátumra lenne szükségünk, amit az osztály nem támogat. Például vannak olyan megrendelőink, akiknek a neve több mint 20 karakter hosszú, ezért módosítani szeretnénk a formátumot, hogy a megrendelő neve 50 karakter széles legyen. Erre való az `IFormatProvider` felület. A saját, testreszabott formátumaink elkészítéséhez létre kell hoznunk az `IFormatProvider` megvalósítását tartalmazó osztályt, és mellé az `ICustomFormatter` megvalósítását tartalmazót. Az `IFormatProvider` felület egy tagfüggvény meghatározását tartalmazza. Ez nem más, mint a `GetFormat()`, ami egy olyan objektumot ad vissza, ami tartalmazza az `ICustomFormatter` felület megvalósítását. A tényleges formázást végző tagfüggvény meghatározását az `ICustomFormatter` felület adja meg. Az alábbi pár úgy módosítja a kimenetet, hogy a megrendelő nevének megjelenítéséhez 50 oszlopot használ:

```
// IFormatProvider példa:
public class CustomFormatter : IFormatProvider
{
  #region IFormatProvider Members
  // Az IFormatProvider egy tagfüggvényt tartalmaz
  // Ez a tagfüggvény egy objektumot ad vissza, ami
  // a kért felületet használva formáz
```

```
    // Általában csak az ICustomFormatter
    // felület megvalósítását készítjük el
    public object GetFormat( Type formatType )
    {
      if ( formatType == typeof( ICustomFormatter ))
        return new CustomerFormatProvider( );
      return null;
    }
    #endregion

    // Beágyazott osztály, ami biztosítja
    // a Customer osztály testreszabott formázását
    private class CustomerFormatProvider : ICustomFormatter
    {
      #region ICustomFormatter Members
      public string Format( string format, object arg,
        IFormatProvider formatProvider )
      {
        Customer c = arg as Customer;
        if ( c == null )
          return arg.ToString( );
        return string.Format( "{0,50}, {1,15}, {2,10:C}",
          c.Name, c.ContactPhone, c.Revenue );
      }
      #endregion
    }
  }
```

A `GetFormat()` tagfüggvény létrehozza azt az objektumot, ami megvalósítja az `ICustomFormatter` felületet, az `ICustomFormatter.Format()` tagfüggvény pedig elvégzi a tényleges formázást a kívánt módon. Ez a tagfüggvény alakítja át szöveges formátumra az objektumot. Meghatározhatunk formátumleírókat az `ICustomFormatter.Format()` részére, hogy egyszerre több formátumot is megadhassunk egyetlen függvényben. A `FormatProvider` lesz az az `IFormatProvider` objektum, amelyet a `GetFormat()` tagfüggvénytől kapunk vissza.

A saját formátum megadásához meg kell hívnunk a `string.Format()` függvényt az `IFormatProvider` objektummal:

```
Console.WriteLine( string.Format( new CustomFormatter(),
  "", c1 ));
```

Az `IFormatProvider` és `ICustomFormatter` megvalósításokat attól függetlenül elkészíthetjük az osztályok számára, hogy azok tartalmazták-e az `IFomattable` felület megvalósítását. Tehát, ha az osztály írója nem is írt egy valamirevaló `ToString()` függvényt, mi pótolhatjuk ezt a hiányosságot. Persze az osztályon kívülről csak a nyilvános tulajdonsá-

gokhoz és adattagokhoz férünk hozzá, tehát csak ezeket használhatjuk a karakterláncok elkészítéséhez. Két osztály, az `IFormatProvider` és az `ICustomFormatter` megírása elég sok munka csak azért, hogy szöveget írhassunk ki. De ha ebben a formában valósítjuk meg a szövegünk megjelenítését, az azt jelenti, hogy az a .NET keretrendszerben mindenhol támogatott lesz.

Most bújjunk vissza az osztály írójának bőrébe. Az `Object.ToString()` felülbírálása a legegyszerűbb módja annak, hogy elkészítsük az osztályaink szöveges ábrázolását. Ezt mindig írjuk meg, ha létrehozunk egy típust. Az a fontos, hogy mindig a típus legnyilvánvalóbb, leggyakrabban használt ábrázolását válasszuk. Azokban az igen ritka esetekben, amikor a típusunknak ennél összetettebb kimenetet kell előállítania, használjuk ki az `IFormattable` felület megvalósításával járó előnyöket. Ezzel szabványos lehetőséget adunk az osztályunk felhasználói számára, hogy a típus szöveges megjelenítését a saját igényeiknek megfelelően alakítsák. Ha ezt nem tesszük meg, a felhasználóknak saját maguknak kell megvalósítaniuk az egyedi formázókat. Ez a megoldás sokkal több kódolással jár, és mivel a felhasználók az osztályon kívül dolgoznak, nem áll módjukban megvizsgálni az objektum belsejét.

A típusainkról kapott információk fogyasztói emberek, akik csak a szöveges kimenetet értik meg. Adjuk meg nekik ezt a legegyszerűbb módon: minden típusunkhoz készítsük el a `ToString()` tagfüggvény felülbírált változatát.

6. tipp

Figyeljünk az értéktípusok és a hivatkozási típusok közti különbségre

Értéktípus vagy hivatkozási típus? Struktúrák vagy osztályok? Mikor melyiket használjuk? Ez nem C++, ahol minden típust értéktípusként határozunk meg, és később hivatkozhatunk rájuk. Ez nem Java, ahol minden hivatkozási típus. Már a típusok létrehozásakor el kell tudnunk dönteni, hogy a típusunk példányai miként fognak viselkedni. Fontos döntés ez, amit nem könnyű elsőre helyesen meghozni. A döntésünk következményeit aztán viselnünk kell, mivel a későbbi módosítás komoly mennyiségű kódban okozhat alig észrevehető hibákat. A típus létrehozásakor csak azt kell eldöntenünk, hogy a `struct` vagy a `class` kulcsszót használjuk, de ha később módosítani szeretnénk a döntésünket, akkor rengeteg munkába telik frissíteni az összes ügyfélnél a típust.

A dolog persze nem olyan egyszerű, hogy az egyiket jobban szeretjük a másiknál. A megfelelő döntés azon múlik, hogy mire számítunk az új típus viselkedését illetően. Az értéktípusok nem többalakúak. Ezek sokkal alkalmasabbak az alkalmazás által kezelt adatok tárolására. A hivatkozási típusok többalakúak is lehetnek, és általában az alkalmazás viselkedésének meghatározására kell használnunk őket. Mindig gondoljuk át, hogy milyen feladat hárul majd az új típusokra, és a feladatok alapján döntsük el, hogy milyen típust hozunk létre. A struktúrák adatokat tárolnak, az osztályok az alkalmazás viselkedését határozzák meg.

Az értéktípusok és hivatkozási típusok közötti különbségtételt azért vezették be a C# nyelvben, mert a C++ és a Java nyelvben ennek hiánya gyakran gondokat okozott. A C++ nyelvben a paraméterek és visszatérési értékek átadása érték szerint történt. Az érték szerinti átadás nagyon hatékony, de van vele egy gond, mégpedig a részleges másolás (amit néha az objektum felszeletelésének is hívnak). Ha egy származtatott objektumot olyan helyen használunk, ahol a program alapobjektumot vár, akkor csak az objektum alapjához tartozó rész másolódik át. Ezzel gyakorlatilag tökéletesen elveszítettünk minden információt, ami egy származtatott objektum jelenlétére utalt volna; még a virtuális függvények hívásait is az alaposztályhoz küldi a program.

A Java nyelv megoldása erre az volt, hogy többé-kevésbé száműzte a nyelvből az értéktípusokat. A felhasználói típusok mind hivatkozási típusok lettek. A Java nyelvben a paraméterek és a visszatérési értékek átadása is hivatkozásként történik. Ennek a megközelítésnek nagy előnye, hogy következetes, de rettentően lerontja a teljesítményt. Lássuk be, néhány típus sosem lesz többalakú, mert egyszerűen nem annak szánták őket. A Java-programozók minden egyes változóért egy memóriafoglalással és egy szemétgyűjtéssel fizetnek. Arról nem is beszélve, hogy minden változó esetében további időbe telik a hivatkozások feloldása. Minden változó egy hivatkozás. A C# nyelvben a `struct` vagy a `class` kulcsszavak segítségével mi mondjuk meg, hogy egy típus értéktípus vagy hivatkozási típus legyen. Az értéktípusok mindig legyenek kisméretű típusok. A hivatkozási típusok alkotják az osztályhierarchiát. Ebben a részben a típusok különböző felhasználási módjait vizsgáljuk meg, hogy tisztába tegyük az értéktípusok és a hivatkozási típusok közti összes különbséget.

Elsőként nézzük meg az alábbi típust, ami egy tagfüggvény visszatérési értékének típusa:

```
private MyData _myData;
public MyData Foo()
{
 return _myData;
}

// meghívjuk:
MyData v = Foo();
TotalSum += v.Value;
```

Ha a `MyData` értéktípus, akkor a visszatérési érték a `v` tárolására kijelölt helyre másolódik. A `v` ezen kívül ott lesz a veremben is. Ha azonban a `MyData` hivatkozási típus, akkor egy belső változóra mutató hivatkozást exportáltunk, és ezzel megsértettük a betokozás elvét (lásd a 23. tippet). Vagy nézzük meg ezt a változatot:

```
private MyData _myData;
public MyData Foo()
{
 return _myData.Clone( ) as MyData;
}
```

```
// meghívjuk:
MyData v = Foo();
TotalSum += v.Value;
```

A v most az eredeti _myData másolata. Hivatkozási típusról lévén szó, a program két objektumot hoz létre a dinamikus memóriában (a halmon). Itt tehát nem kell számolnunk a belső adatok exportálásának problémájával, ehelyett egy új objektumot hoztunk létre a halmon. Ha a v helyi változó, akkor hamarosan szemét lesz belőle, a Clone pedig rákényszerít minket a futásidejű típusellenőrzésre. Mindent egybevetve ez nem túl hatékony megoldás.

Azok a típusok, amelyeket adatok nyilvános tagfüggvényekkel való exportálásához használunk, mindig legyenek értéktípusok. Ez persze nem azt jelenti, hogy minden nyilvános tagtól visszakapott típusnak értéktípusnak kell lennie. A korábban bemutatott kódrészletben feltételeztük, hogy a MyData adatokat tárol. A feladata tehát az, hogy tárolja ezeket az adatokat.

De most nézzük meg ezt a másik kódrészletet:

```
private MyType _myType;
public IMyInterface Foo()
{
  return _myType as IMyInterface;
}

// meghívjuk:
IMyInterface iMe = Foo();
iMe.DoWork( );
```

A _myType változót most is a Foo tagfüggvénytől kapjuk vissza, de ezúttal ahelyett, hogy a visszaadott értéken belüli adatokat próbálnánk elérni, az objektumelérés azért történik, hogy egy meghatározott felületen keresztül meghívjunk egy tagfüggvényt. Nem az általa tartalmazott adatok miatt nyúlunk a MyType objektumhoz, hanem a viselkedése miatt. Ezt a viselkedést az IMyInterface írja le, amelynek megvalósítása számos különböző típusban lehetséges. Ebben a példában a MyType hivatkozási típus és nem értéktípus kell hogy legyen. A MyType feladata ezúttal a viselkedése és nem az adattagjai körül forog.

Ebből az egyszerű kódocskából már kezdjük látni a különbséget: az értéktípusok adatokat tárolnak, a hivatkozási típusok viszont egyfajta viselkedést határoznak meg. Most pedig nézzünk meg kicsit közelebbről, hogyan tárolja a program ezeket a típusokat a memóriában, és milyen következményei vannak a tárolás módjának a teljesítményre.

Vegyük az alábbi osztályt:

```
public class C
{
  private MyType _a = new MyType( );
  private MyType _b = new MyType( );

  // a megvalósítás többi része törölve
}

C var = new C();
```

Hány objektumot hoz létre a program és mekkorákat? Az attól függ. Ha a `MyType` értéktípus, akkor egy foglalást hajtottunk végre. Ennek mérete a `MyType` méretének kétszerese. Ha azonban a `MyType` hivatkozási típus, akkor három foglalást hajtottunk végre: egyet a `C` objektumnak, ami 8 bájt (32 bites mutatókat feltételezve), és még kettőt, mindkét `MyType` objektum részére, amelyeket a `C` objektum tartalmaz. A különbség abból adódik, hogy az értéktípusok tárolása helyben, egy objektumban történik, míg a hivatkozási típusoknál ez nem így van. Minden hivatkozási típusú változó egy hivatkozást tárol, a tároláshoz pedig további memóriafoglalásra van szükség.

Hogy teljesen tisztán lássunk, nézzük meg ezt a memóriafoglalást:

```
MyType [] var = new MyType[ 100 ];
```

Ha a `MyType` értéktípus, akkor a program egy foglalást hajt végre, amivel a `MyType` objektum méretének 100-szorosával egyenlő méretű memóriablokkot foglal le. Ha viszont a `MyType` hivatkozási típus, akkor csak egyetlen foglalás történik, a tömb elemeinek értéke pedig `null` lesz. Mire a tömb összes elemének valamilyen kezdőértéket adunk, addigra összesen 101 foglalást hajtunk végre, és ugye 101 foglalás több időbe telik, mint 1 foglalás. Túl sok hivatkozási típus töredezetté teszi a halmot, és lelassítja a programot. Ha olyan típusokat akarunk létrehozni, amelyek adatokat fognak tárolni, akkor ezt mindenképpen értéktípusok segítségével tegyük.

Fontos döntést hozunk, amikor egy értéktípus vagy egy hivatkozási típus létrehozása mellett tesszük le a voksunkat. Az értéktípusok átalakítása osztálytípussá messzemenő következményekkel jár. Vegyük az alábbi típust:

```
public struct Employee
{
  private string  _name;
  private int     _ID;
  private decimal _salary;

  // a tulajdonságokat elhagytuk
```

```
    public void Pay( BankAccount b )
    {
      b.Balance += _salary;
    }
  }
```

Ez a meglehetősen egyszerű típus egyetlen tagfüggvényt tartalmaz, ami a dolgozók bérének kifizetésére szolgál. Telik-múlik az idő, és a rendszer egész szépen működik. Egy napon aztán úgy döntünk, hogy többféle csoportba soroljuk az alkalmazottainkat. Az értékesítők jutalékot kapnak, a vezetők meg jutalmat. Úgy döntünk hát, hogy az `Employee` típusból osztályt csinálunk:

```
public class Employee
{
  private string  _name;
  private int     _ID;
  private decimal _salary;

  // a tulajdonságokat elhagytuk

  public virtual void Pay( BankAccount b )
  {
    b.Balance += _salary;
  }
}
```

Ez bizony tönkreteszi annak a már létező kódnak a nagy részét, ami használta az alkalmazott struktúrát. Az érték szerinti visszatérési értékből hivatkozás szerinti visszatérési érték lesz. A paramétereket, amelyeket korábban értékként adtunk át, most hivatkozásként adjuk át. Az alábbi kódrészlet viselkedése gyökeresen megváltozik:

```
Employee e1 = Employees.Find( "CEO" );
e1.Salary += Bonus; // egyszeri jutalom hozzáadása
e1.Pay( CEOBankAccount );
```

Ami egykor egy egyszeri fizetéskiegészítés (jutalom) volt, abból most tartós fizetésemelés lett. Ahol egy érték másolása szerepelt, most egy hivatkozást találunk. A fordító vidáman elvégzi a módosítást. Ennél vidámabb már csak a cégvezető lesz. A gazdasági osztályon azonban valószínűleg kiszúrják a hibát. Egyszerűen nem lehet csak úgy utólag meggondolni magunkat az értéktípusokkal és a hivatkozási típusokkal kapcsolatban, mert ez befolyásolja a program működését.

A gond abból adódik, hogy az `Employee` típus már nem felel meg az értéktípusoktól elvártaknak. Az alkalmazottak adatainak tárolása mellett egy feladattal is elláttuk a típust, esetünkben az alkalmazott kifizetésével. A feladatok az osztálytípusok hatáskörébe tartoznak. Az osztályok egyszerűen képesek többalakú megvalósításokat létrehozni alapvető feladatokhoz. A struktúrák erre nem képesek, ezért használatukat korlátozzuk az adattárolásra.

A .NET dokumentációja azt javasolja, hogy a típus méretét vegyük figyelembe akkor, amikor egy értéktípus vagy egy hivatkozási típus közül választunk. A valóságban azonban a típus felhasználási módja sokkal fontosabb tényező. Azok a típusok, amelyek egyszerű struktúrák vagy adatokat hordoznak, kiválóan alkalmasak értéktípusnak. Fontos, hogy az értéktípusok hatékonyabbak a memóriakezelés szempontjából. Kevésbé lesz töredezett a halom, kevesebb a szemét, és kevesebb a közvetettség. De ami ennél is fontosabb, az értéktípusokat lemásolja a program, amikor egy tulajdonságtól vagy egy tagfüggvénytől visszakapjuk őket. Nincs az a veszély, hogy belső szerkezetekre mutató hivatkozásokat teszünk elérhetővé. Ennek árát azonban a kevesebb szolgáltatással fizetjük meg. Az értéktípusok csak nagyon korlátozott mértékben támogatják a gyakori objektumközpontú módszereket. Értéktípusokból nem tudunk objektumhierarchiákat építeni. Az értéktípusokra úgy kell tekintenünk, mintha egy zárt borítékban lennének. Létrehozhatunk felületeket megvalósító értéktípusokat, de ehhez „dobozolásra" van szükség, ami teljesítménycsökkenéshez vezet, ahogy azt a 17. tippben majd látni fogjuk. Az értéktípusokat csak tárolóhelyeknek tekintsük, és ne OO értelemben vett objektumoknak.

Hivatkozási típusból többet fogunk készíteni, mint értéktípusból. Ha az alábbi kérdésekre mind igennel felelünk, akkor készítsünk értéktípust. Vessük össze ezeket az előző `Employee` példával:

1. A típusnak az adattárolás lesz a fő feladata?
2. Kizárólag olyan tulajdonságokból áll a nyilvános felülete, amelyek csak az adattagok elérését és módosítását végzik?
3. Biztosan nem lesz ennek a típusnak a jövőben alosztálya?
4. Biztos, hogy a jövőben sosem kezelik majd ezt a típust többalakúként?

Az alacsonyszintű adattároló típusokat mindig értéktípusként vezessük be. Az alkalmazásunk viselkedésének felépítéséhez hivatkozási típusokat használjunk. Így biztonságos lesz az osztályok objektumaiból exportált adatok másolása. Kihasználhatjuk továbbá a verem alapú, illetve a belső memóriahasználattal kapcsolatos előnyöket, és a szokásos objektumközpontú módszerekkel építhetjük fel a program szerkezetét. Ha bizonytalanok vagyunk egy típus jövőbeli felhasználását illetően, akkor mindig használjunk hivatkozási típust.

7. tipp

Használjunk nem változó, elemi értéktípusokat

A nem változó típusok (immutable types) egyszerűek: miután létrehoztuk őket, állandók maradnak. Ha ellenőrizzük az objektum létrehozásához megadott paramétereket, akkor biztosak lehetünk benne, hogy onnantól kezdve az objektum érvényes adatokat fog tartalmazni. Az objektum belsejét ugyanis nem módosíthatjuk, tehát érvénytelenítésére sincs mód. Rengeteg, máskülönben szükséges hibaellenőrzéstől kímélhetjük meg magunkat azzal, hogy az objektum létrehozása után nem engedünk rajta változtatni. A nem változó típusok természetüknél fogva a különböző szálak hozzáférései ellen is védettek, vagyis egyszerre több szál is elérheti ugyanazt a tartalmat. Ha az objektum belső állapota nem változhat, akkor annak sem forog fenn a veszélye, hogy a különböző szálak következetlen képet kapjanak az adatokról. A nem változó típusokat biztonságosan exportálhatjuk az objektumainkból. A hívó függvény nem módosíthatja az objektum belső állapotát. A nem változó típusok a kivonat (hasítótábla, hash) alapú gyűjteményekkel is jobban használhatók. Az `Object.GetHashCode()` függvény által visszaadott értéknek a példányra nézve invariánsnak (lásd a 10. tippet) kell lennie, ami a nem változó típusokra mindig igaz.

Nem minden típus lehet változatlan. Ha így lenne, akkor a program állapotának módosításához mindig el kellene készítenünk az objektumok pontos másolatát. Ezért szól ez a tipp az elemi nem változó típusokról. Mindig daraboljuk szét a típusainkat olyan szerkezetekre, amelyek természetes egységet alkotnak. Ilyen például egy `Address` típus. A cím egyetlen dolog, amit több hozzá kapcsolódó mező alkot. Ha változik az egyik mező, akkor valószínűleg más mezők is változni fognak. A megrendelő viszont nem elemi típus. A megrendelő típusa valószínűleg sokféle információt tartalmaz: egy címet, egy nevet, és egy vagy több telefonszámot. Ezen független információk bármelyike megváltozhat. A megrendelő lecserélheti a telefonszámát anélkül, hogy elköltözne. Az is lehet, hogy elköltözik, de megtartja a régi számát. A megrendelő neve is változhat anélkül, hogy elköltözne vagy megváltoztatná a telefonszámát. A megrendelő objektum tehát nem elemi objektum. Sokféle nem változó típus összetételéből épül fel: egy címből, egy névből és esetleg a telefonszámok és telefonszámtípusok párjainak gyűjteményéből. Az elemi típusok egy egységet alkotnak: az elemi típusoknak természetükből fakadóan mindig a teljes tartalmát cseréljük. A kivételes eset az, ha csak egy alkotó mezőt módosítunk.

Álljon itt egy jellegzetes megvalósítása egy változó címnek:

```
// Változó címszerkezet
public struct Address
{
  private string  _line1;
  private string _line2;
  private string  _city;
```

```
  private string _state;
  private int    _zipCode;

  // Az alapértelmezett, rendszer által biztosított konstruktorra
  // hagyatkozunk

  public string Line1
  {
    get { return _line1; }
    set { _line1 = value; }
  }
  public string Line2
  {
    get { return _line2; }
    set { _line2 = value; }
  }
  public string City
  {
    get { return _city; }
    set { _city= value; }
  }
  public string State
  {
    get { return _state; }
    set
    {
      ValidateState(value);
      _state = value;
    }
  }
  public int ZipCode
  {
    get { return _zipCode; }
    set
    {
      ValidateZip( value );
      _zipCode = value;
    }
  }
  // az egyéb részleteket elhagytuk
}

// Példa a használatra:
Address a1 = new Address( );
a1.Line1 = "111 S. Main";
a1.City = "Anytown";
a1.State = "IL";
a1.ZipCode = 61111 ;
```

```
// Módosítás:
a1.City = "Ann Arbor"; // Az irányítószám és az állam most hibás
a1.ZipCode = 48103; // Az állam még mindig hibás
a1.State = "MI"; // Most már rendben van
```

A belső állapot módosításával megsérthetjük az objektum belső szabályait, legalábbis átmenetileg. A `City` mező értékének lecserélése után az `a1` érvénytelen állapotba került. A város megváltozott, és már nem illeszkedik az állam és az irányítószám mezők tartalmához. A kód elég ártalmatlannak tűnik, de tegyük fel, hogy ez a kódrészlet egy több szálat futtató program része. Egy környezetváltás a város megváltoztatása után, de még az állam megváltoztatása előtt, lehetőséget ad arra, hogy egy másik szál értelmetlen adatokat kapjon.

Jó, de mi van, ha a mi programunk nem több szálon fut? Ettől még bajba kerülhetünk. Képzeljük el, hogy az irányítószám még hibás, és az értékadás kivált egy kivételt. Ilyenkor még nem fejeztük be az összes szándékolt módosítást, és a rendszert érvénytelen állapotban hagyjuk. Hogy ezt elkerüljük, jelentős mennyiségű belső ellenőrzőkódot kellene hozzáadnunk a cím szerkezethez. Az ellenőrző kód csak növelné a program méretét, ami sokkal bonyolultabb is lenne. Ha teljes mértékben védekezni akarnánk a kivételek ellen, akkor minden olyan kódblokk köré biztonsági másolatokat kellene tennünk, amelyekben egynél több mező értékét módosítjuk. A szálak biztonságos működéséhez jelentős, a szálak összehangolásáért felelős ellenőrzésekkel kellene ellátnunk minden tulajdonságelérőt, beleértve az értékek kiolvasását és az értékek beállítását végző elérőket is. Mindent egybevetve igen nagy fába vágnánk a fejszénket, és ez a fa egyre csak nő, amint újabb és újabb szolgáltatásokkal látjuk el a programunkat.

Ehelyett legyen az `Address` szerkezet egy nem változó típus. Kezdjük azzal, hogy a példány minden mezőjét csak olvashatóra állítjuk:

```
public struct Address
{
  private readonly string  _line1;
  private readonly string  _line2;
  private readonly string  _city;
  private readonly string  _state;
  private readonly int    _zipCode;

  // további részletek elhagyva
}
```

Ezután távolítsuk el az összes tulajdonság `set` elérőjét:

```
public struct Address
{
  // ...
```

```
  public string Line1
  {
    get { return _line1; }
  }
  public string Line2
  {
    get { return _line2; }
  }
  public string City
  {
    get { return _city; }
  }
  public string State
  {
    get { return _state; }
  }
  public int ZipCode
  {
    get { return _zipCode; }
  }
}
```

Most már készen is van a nem változó típusunk. Hogy valami hasznát is vegyük, adjuk hozzá a szükséges konstruktorokat, hogy elvégezhessük a teljes `Address` szerkezet kezdőértékekkel való ellátását. Az `Address` szerkezetnek egyetlen további konstruktorra van szüksége, amiben minden mezőnek értéket adunk. Másoló konstruktorra nincs szükség, mert az értékadó művelet éppen olyan jól megteszi. Ne feledjük, hogy az alapértelmezett konstruktor továbbra is elérhető. Van egy alapértelmezett cím, ahol az összes karakterlánc értéke `null`, az irányítószám pedig `0`:

```
public struct Address
{
  private readonly string  _line1;
  private readonly string  _line2;
  private readonly string  _city;
  private readonly string  _state;
  private readonly int    _zipCode;

  public Address( string line1,
    string line2,
    string city,
    string state,
    int zipCode)
  {
    _line1 = line1;
    _line2 = line2;
    _city = city;
```

```
      _state = state;
      _zipCode = zipCode;
      ValidateState( state );
      ValidateZip( zipCode );
    }

    // stb.
  }
```

A nem változó típus használatakor kissé más hívási sorrendet kell alkalmaznunk ahhoz, hogy megváltoztassuk az állapotát. Új objektumot hozunk létre, ahelyett, hogy a már létező objektumot módosítanánk:

```
// Létrehozunk egy címet:
Address a1 = new Address( "111 S. Main",
  "", "Anytown", "IL", 61111 );

// Ha módosítani akarunk valamit, akkor kezdjük elölről:
a1 = new Address( a1.Line1,
  a1.Line2, "Ann Arbor", "MI", 48103 );
```

Az `a1` értéknek kétféle állapota van: vagy az eredeti `Anytown`, vagy a frissített `Ann Arbor`-i cím. Az eredeti címet nem módosítjuk, így az előbbi példában látható érvénytelen állapotokat sem hozhatjuk létre. Ezek a köztes állapotok csak az `Address` konstruktor végrehajtása közben léteznek, és ezek a konstruktoron kívül láthatatlanok. Amint elkészül egy `Address` objektum, annak értéke örökre változatlan marad, és a kivételek ellen is védett. Az `a1` értéke vagy az eredeti érték lesz, vagy az újonnan megadott érték. Ha kivétel következik be az új `Address` objektum létrehozása közben, akkor az `a1` eredeti értéke változatlan marad.

Egy nem változó típus létrehozásához gondoskodnunk kell róla, hogy ne maradjanak olyan rések rajta, amelyeken keresztül az ügyfelek módosítani tudják a belső állapotát. Az értéktípusok nem támogatják a származtatott típusokat, tehát attól nem kell tartanunk, hogy egy származtatott típus módosítja a mezőket. Azokra a nem változó mezőkre azonban figyelnünk kell, amelyek változó hivatkozási típusok. Amikor a konstruktor megvalósításán dolgozunk ezeknél a típusoknál, mindig készítenünk kell egy biztonsági másolatot a változó típusról. Az alábbi példák feltételezik, hogy a `Phone` egy nem változó értéktípus, mivel minket csak az értéktípusok változtathatatlansága érdekel:

```
// Majdnem nem változó, de van néhány rés, ami
// lehetővé tenné az állapotváltozásokat
public struct PhoneList
{
  private readonly Phone[] _phones;

  public PhoneList( Phone[] ph )
  {
```

```
      _phones = ph;
    }

    public IEnumerator Phones
    {
      get
      {
        return _phones.GetEnumerator();
      }
    }
  }

  Phone[] phones = new Phone[10];
  // a phones létrehozása
  PhoneList pl = new PhoneList( phones );

  // A PhoneList módosítása
  // kihat a (feltételezetten)
  // nem változó objektum belsejére
  phones[5] = Phone.GeneratePhoneNumber( );
```

A tömb osztály hivatkozási típus. Az a tömb, amire a `PhoneList` szerkezet belsejében hivatkozunk, ugyanarra a tömbtárolóhelyre (`phones`) mutat, amit az objektumon kívül foglaltunk le. Más fejlesztők módosítani tudnák a nem változó szerkezetünket egy másik változón keresztül, ami ugyanarra a tárhelyre mutat. Hogy megszüntessük ennek a lehetőségét, készítenünk kell a tömbről egy biztonsági másolatot. Az előbbi példa jól mutatja a nem változó gyűjtemények buktatóit. Még ennél is több bajjal jár, ha a `Phone` típus egy változó hivatkozási típus. Ebben az esetben az ügyfelek még akkor is módosítani tudnák a gyűjteményben található értékeket, ha a gyűjtemény minden módosítással szemben védett. A biztonsági másolatot minden olyan esetben el kell készítenünk a konstruktorban, amikor egy nem változó típus egy változó hivatkozási típust tartalmaz:

```
// Nem változó: létrehozásakor lemásoljuk
public struct PhoneList
{
  private readonly Phone[] _phones;

  public PhoneList( Phone[] ph )
  {
     _phones = new Phone[ ph.Length ];
     // Értékeket másol, mert a Phone értéktípus
     ph.CopyTo( _phones, 0 );
  }

  public IEnumerator Phones
  {
```

```
      get
      {
        return _phones.GetEnumerator();
      }
    }
  }

  Phone[] phones = new Phone[10];
  // a phones létrehozása
  PhoneList pl = new PhoneList( phones );

  // A PhoneList módosítása
  // a pl-ben található másolatot nem érinti
  phones[5] = Phone.GeneratePhoneNumber( );
```

Ugyanezeket a szabályokat kell követnünk, amikor egy változó hivatkozási értéket adunk vissza. Ha megadunk egy olyan tulajdonságot, amivel megkapjuk az egész tömböt a `PhoneList` struktúrából, akkor annak az elérőnek is el kell készítenie egy biztonsági másolatot. A részleteket lásd a 23. tippnél.

A típusok összetettsége határozza meg, hogy a három eljárás közül melyiket kell alkalmaznunk a nem változó típus létrehozásához. Az `Address` struktúra egy konstruktort határozott meg, amivel az ügyfelek létrehozhattak egy címet. Általában a konstruktorok megfelelő meghatározása adja a probléma legegyszerűbb megközelítését.

Beépített függvényeket is használhatunk a szerkezet létrehozásához. A beépített függvényekkel egyszerűbben hozhatunk létre gyakori értékeket. A .NET keretrendszer `Color` típusa ezt a módszert használja a rendszerszínek kezdőértékeinek megadásához. A statikus `Color.FromKnownColor()` és `Color.FromName()` tagfüggvények egy színérték másolatát adják vissza, ami az adott rendszerszín aktuális színértékének felel meg.

Harmadszor pedig létrehozhatunk egy kísérőosztályt azokhoz a példányokhoz, amelyeknél többlépéses műveletekre van szükség a nem változó típus teljes létrehozásához. A .NET karakterlánc-osztálya ezt a megközelítést alkalmazza a `System.Text.StringBuilder` osztály esetében. A `StringBuilder` osztállyal több lépésben hozhatunk létre egy karakterláncot. A karakterlánc felépítéséhez szükséges összes művelet végrehajtása után kiolvassuk a nem változó karakterláncot a `StringBuilder` objektumból.

A nem változó típusok megírása és karbantartása egyszerű. Ne készítsünk ész nélkül a típusunk minden tulajdonságához `get` és `set` elérőket. Az adatok tárolására szolgáló típusoknál a nem változó, elemi értéktípusok a nyerők. Ezekből az egységekből kiindulva már egyszerűen készíthetünk bonyolultabb szerkezeteket is.

8. tipp

Gondoskodjunk róla, hogy minden értéktípusnál érvényes legyen a 0 állapot

A .NET rendszer alapértelmezése szerint minden objektum kezdőértéke 0. Azt sem tudjuk megakadályozni, hogy egy másik programozó létrehozzon egy példányt egy értéktípusból, aminek kezdőértéke csupa 0 lesz. Legyen tehát ez a saját típusunk alapértelmezett értéke is.

Ennek a kérdésnek egyik speciális esetét a felsorolások adják. Soha ne hozzunk létre olyan felsorolást, ami nem tartalmazza a 0 értéket, mint érvényes választási lehetőséget. Minden felsorolás a System.ValueType osztályból származik. A felsorolások értéke a 0 értékkel indul, de ezt a viselkedést módosíthatjuk is:

```
public enum Planet
{
  // Megadjuk az értékeket, különben
  // a felsorolás az alapértelmezés szerint a 0 értéktől kezdődne
  Mercury = 1,
  Venus = 2,
  Earth = 3,
  Mars = 4,
  Jupiter = 5,
  Saturn = 6,
  Neptune = 7,
  Uranus = 8,
  Pluto = 9
}

Planet sphere = new Planet();
```

A sphere értéke 0, ami érvénytelen érték. Az a kód, ami arra a (józan) feltevésre épül, hogy a felsorolások a felsorolt értékek meghatározott halmazára szorítkoznak, nem fog működni. Ha saját értékeket adunk meg egy felsorolásban, akkor mindig gondoskodjunk róla, hogy a 0 is köztük legyen. Ha bitmaszkokból álló felsorolást készítünk, akkor a 0 mindig az összes tulajdonság hiányát jelezze. A dolgok jelenlegi állása szerint arra kényszerítjük az összes felhasználót, hogy megadjon egy tényleges kezdőértéket:

```
Planet sphere = Planet.Mars;
```

Ez megnehezíti más olyan típusok létrehozását, amelyek erre a típusra épülnek:

```
public struct ObservationData
{
  Planet   _whichPlanet; // Mit látunk?
  Double _magnitude; // látszólagos fényesség
}
```

Azok a felhasználók, akik létrehoznak egy új ObservationData objektumot, rögtön egy érvénytelen Planet mezőhöz jutnak:

```
ObservationData d = new ObservationData();
```

Az újonnan létrehozott ObservationData objektum fényessége (_magnitude) 0 lesz, ami teljesen rendben is van. A bolygó azonban érvénytelen lesz, ezért a 0 állapothoz egy érvényes értéket kell megadnunk. Ha lehetséges, akkor a legkézenfekvőbb alapértelmezéshez válasszuk a 0 értéket. A Planet felsorolásnak nincs nyilvánvaló alapértelmezése. Annak semmi értelme, hogy véletlenszerűen alapértelmezettnek válasszunk egy bolygót, amikor a felhasználó nem választ egyet sem. Ha ilyen helyzetbe ütközünk, akkor a 0 esetet egy olyan kezdőértékhez rendeljük, amit később frissíthetünk:

```
public enum Planet
{
  None = 0,
  Mercury = 1,
  Venus = 2,
  Earth = 3,
  Mars = 4,
  Jupiter = 5,
  Saturn = 6,
  Neptune = 7,
  Uranus = 8,
  Pluto = 9
}

Planet sphere = new Planet();
```

A sphere most a None értéket tartalmazza. Azzal, hogy hozzáadtuk ezt az alapértelmezést a Planet felsoroláshoz, a módosítás továbbterjed az ObservationData szerkezetre is. Az újonnan létrehozott ObservationData objektumok fényereje 0, célpontjuk pedig None lesz. Egy konstruktort megadva lehetővé tehetjük a felhasználó számára, hogy pontos kezdőértéket adjon a típusunk összes mezőjének:

```
public struct ObservationData
{
  Planet    _whichPlanet; // Mit látunk?
  Double  _magnitude; // látszólagos fényesség

  ObservationData( Planet target,
    Double mag )
  {
    _whichPlanet = target;
    _magnitude = mag;
  }
}
```

Ne feledjük azonban, hogy az alapértelmezett konstruktor továbbra is ott van, és része a szerkezetnek. A felhasználók továbbra is létrehozhatják a rendszer által előállított és kezdőértékekkel ellátott változatot, és mi semmit sem tehetünk ez ellen.

Mielőtt elbúcsúznánk a felsorolásoktól, és rátérnénk a többi értéktípusra, néhány szabállyal még meg kell ismerkednünk, amelyek a jelzőértékként használt felsorolásokra vonatkoznak. Azoknál a felsorolásoknál, amelyek a `Flags` jellemzőt használják, a `None` (egyik sem) mindig `0` értéket kell, hogy kapjon:

```
[Flags]
public enum Styles
{
  None = 0,
  Flat = 1,
  Sunken = 2,
  Raised = 4,
}
```

Sok fejlesztő használja a jelzőérték-felsorolásokat a bitenkénti és (AND) művelettel. A `0` értékek komoly gondokat okozhatnak a jelzőbiteknél. Az alábbi összehasonlítás eredménye sosem lesz igaz, ha a `Flat` értéke `0`:

```
if ( ( flag & Styles.Flat ) != 0 ) // Soha nem igaz, ha Flat == 0
  DoFlatThings( );
```

Ha a `Flags` jellemzőt használjuk, mindig gondoskodjunk róla, hogy a `0` érvényes érték legyen, és azt jelentse, hogy „minden jelzőbit kikapcsolva".

Egy másik gyakori hiba a kezdőérték megadásánál azokat az értéktípusokat érinti, amelyek hivatkozásokat tartalmaznak. A leggyakoribb példát a karakterláncok szolgáltatják:

```
public struct LogMessage
{
  private int _ErrLevel;
  private string _msg;
}

LogMessage MyMessage = new LogMessage( );
```

A `MyMessage _Msg` mezője egy `null` hivatkozást tartalmaz. Más kezdőérték kikényszerítésére nincs mód, de a tulajdonságok segítségével meghatározhatjuk a problémát. Létrehoztunk egy tulajdonságot, amivel elérhetővé tehetjük az `_Msg` értékét az ügyfelek számára.

Adjunk hát némi értelmet ennek a tulajdonságnak, hogy egy üres karakterláncot adjon vissza a `null` hivatkozás helyett:

```
public struct LogMessage
{
  private int _ErrLevel;
  private string _msg;

  public string Message
  {
    get
    {
      return (_msg != null ) ?
        _msg : string.Empty;
    }
    set
    {
      _msg = value;
    }
  }
}
```

Ezt a tulajdonságot a saját típusunkon belül kell használnunk. Ezzel egyetlen helyen megoldhatjuk a `null` hivatkozás ellenőrzését. A `Message` elérő is szinte biztosan bekerül a kódba, amikor a szerelvényünkön belülről hívja meg a program. Ez hatékonyabb kódot és kevesebb hibát eredményez.

A rendszer a típusértékek összes példányának `0` kezdőértéket ad. Nincs rá semmilyen mód, hogy megakadályozzuk, hogy a felhasználók ne csupa `0` kezdőértékkel hozzák létre az értéktípusok példányait. Ha lehet, akkor a csupa `0` érték a legtermészetesebb alapértelmezést jelölje. Speciális esetként, a jelzőértékként használt felsorolásoknál mindig biztosítsuk, hogy a `0` az összes jelzőérték kikapcsolt állapotát jelezze.

9. tipp

Értsük meg a ReferenceEquals(), a statikus Equals(), a példány Equals() és az operator==() közti összefüggéseket

Amikor saját típusokat készítünk (legyen az egy osztály vagy egy struktúra), akkor meghatározzuk, hogy az egyenlőség mit jelent az adott típusnál. A C# nyelvben négy különböző függvényt találunk, amelyek eldöntik, hogy két különböző objektum „egyenlő"-e:

```
public static bool ReferenceEquals
  ( object left, object right );
public static bool Equals
  ( object left, object right );
public virtual bool Equals( object right);
public static bool operator==( MyClass left, MyClass right );
```

A nyelv lehetővé teszi, hogy mind a négy fenti tagfüggvényből elkészítsük a saját változatunkat. De az, hogy erre lehetőségünk van, még nem ok arra, hogy meg is tegyük. Az első két statikus függvényt soha ne írjuk újra. Az `Equals()` tagfüggvényből gyakran elkészítünk egy változatot a saját példányunk számára a típus jelentésének meghatározásához, illetve alkalmanként felülbíráljuk az `operator==()` tagfüggvényt, de csak az értéktípusoknál és csak a teljesítmény növelése miatt. A négy függvény összefügg, tehát ha módosítjuk az egyiket, azzal befolyásolhatjuk a többi viselkedését is. Bizony elég bonyolult, hogy négy függvényre is szükség van az egyenlőség ellenőrzéséhez. De ne aggódjunk, egyszerűsíthetjük a dolgot.

A C# nyelv sok más bonyolult eleméhez hasonlóan itt is az okozza a gondot, hogy a C# nyelvben értéktípusokat és hivatkozási típusokat is létrehozhatunk. Két hivatkozási típusú változó akkor egyenlő, ha mindkettő ugyanarra az objektumra hivatkozik. Ezt nevezzük az objektumok azonosságának. Két értéktípusú változó akkor egyenlő, ha típusuk és tartalmuk is megegyezik. Ezért van az, hogy az egyenlőség ellenőrzéséhez ennyi különböző tagfüggvényre van szükség.

Kezdjük azzal a két függvénnyel, amit soha nem szabad módosítanunk. Az `Object.ReferenceEquals()` igaz (`true`) értéket ad vissza, ha két változó ugyanarra az objektumra hivatkozik, vagyis a két objektum azonossága megegyezik. Ez a tagfüggvény mindig az objektumok azonosságát és nem a tartalmukat veti össze, attól függetlenül, hogy az összehasonlított típusok hivatkozási típusok vagy értéktípusok. Igen, ez azt jelenti, hogy a `ReferenceEquals()` mindig hamis (`false`) értéket ad vissza, ha két értéktípus összehasonlítására használjuk.

Ennek a „dobozolás" az oka, amiről a 16. tippben lesz szó.

```
int i = 5;
int j = 5;
if ( Object.ReferenceEquals( i, j ))
  Console.WriteLine( "Never happens." );
else
  Console.WriteLine( "Always happens." );

if ( Object.ReferenceEquals( i, i ))
  Console.WriteLine( "Never happens." );
else
  Console.WriteLine( "Always happens." );
```

Az `Object.ReferenceEquals()` tagfüggvényt sosem írjuk át, mert pontosan azt teszi, amit tennie kell: összehasonlítja két változó objektumazonosságát.

A másik függvény, amit sosem írunk át, a statikus `Object.Equals()`. Ez a tagfüggvény azt ellenőrzi, hogy két változó egyenlő-e, ha nem ismerjük a két paraméter futásidejű típusát. Emlékezzünk vissza rá, hogy a C# nyelvben minden osztály végeredményben a `System.Object` osztály leszármazottja. Bármely két változót hasonlítunk is össze, azok a `System.Object` példányai lesznek. De hogyan tudja eldönteni a függvény, hogy két változó egyenlő-e, ha egyszer nem ismeri a típusukat, az egyenlőség jelentése pedig típusonként változik? A válasz egyszerű. Ez a tagfüggvény továbbadja a feladatot az egyik kérdéses típusnak. A statikus `Object.Equals()` tagfüggvény megvalósítása valahogy így néz ki:

```
public static bool Equals( object left, object right )
{
  // Objektumok azonosságának ellenőrzése
  if (left == right )
    return true;
  // ha mindkettő null hivatkozás, azt az előbb kezeltük
  if ((left == null) || (right == null))
    return false;
  return left.Equals (right);
}
```

Ebben a példában bemutatkozott az a két tagfüggvény is, amiről még nem beszéltünk: az `operator==()` és a példány `Equals()` tagfüggvénye. Mindkettőt részletesen elmagyarázzuk majd, de még nem végeztünk a statikus `Equals()` tagfüggvény tárgyalásával. Egyelőre azt szeretnénk elérni, hogy megértsük, hogy a statikus `Equals()` tagfüggvény a bal oldali paraméter példányának `Equals()` tagfüggvényét használja annak eldöntéséhez, hogy két objektum egyenlő-e.

A `ReferenceEquals()` tagfüggvényhez hasonlóan a statikus `Object.Equals()` tagfüggvényt sem fogjuk soha felülírni, mert éppen azt teszi, amit tennie kell: eldönti két objektumról, hogy azonosak-e, ha nem ismerjük a futásidejű típusokat. A statikus `Equals()` tagfüggvény továbbadja a feladatot a bal oldali paraméter példány `Equals()` tagfüggvényének, ezért annak a típusnak a szabályai szerint működik.

Most már értjük, miért nem kell soha felülírnunk a statikus `ReferenceEquals()` és a statikus `Equals()` tagfüggvényeket. Térjünk most rá azokra a tagfüggvényekre, amelyeket felülbírálunk. De előbb még nézzük meg gyorsan az ekvivalencia relációk matematikai tulajdonságait. Fontos, hogy az általunk megadott reláció megvalósítása megfeleljen a többi programozó elvárásainak. Ez azt jelenti, hogy nem szabad megfeledkeznünk az egyenlőség (azonosság, egyenértékűség, ekvivalencia) matematikai tulajdonságairól. Az egyenlőség **reflexív**, **szimmetrikus** és **tranzitív**. A reflexivitás azt jelenti, hogy minden objektum egyenlő önmagával. Bármilyen típusról legyen is szó, az `a == a` kifejezésnek mindig igaznak kell lennie. A szimmetrikusság azt jelenti, hogy a sorrend nem számít: ha `a == b` igaz, akkor `b == a` is igaz. Ha `a == b` hamis, akkor `b == a` is hamis. Az egyenlőség utolsó tulajdonsága, hogy ha `a == b` igaz és `b == c` is igaz, akkor `a == c` is igaz kell, hogy legyen. Ez a tulajdonság a tranzitivitás.

Elérkezett az idő, hogy megbeszéljük, hogy mikor és hogyan kell használnunk és felülbírálnunk a példányok `Object.Equals()` tagfüggvényét. Akkor készítjük el az `Equals()` függvény egy változatát a saját példányunkhoz, ha az alapértelmezett viselkedés nem megfelelő a típusunkhoz. Az `Object.Equals()` tagfüggvény az objektumok azonosságát használja annak eldöntéséhez, hogy két változó egyenlő-e. Az alapértelmezett `Object.Equals()` függvény pontosan úgy működik, mint az `Object.ReferenceEquals()` tagfüggvény. De álljunk meg egy szóra, hiszen az értéktípusok eltérnek ettől. A `System.ValueType` bizony felülbírálja az `Object.Equals()` tagfüggvényt. Emlékezzünk vissza, hogy a `ValueType` osztály adja az összes általunk (a `struct` kulcsszóval) létrehozott értéktípus alaposztályát. Két értéktípusú változó akkor egyenlő, ha típusuk és tartalmuk is megegyezik. Ennek a viselkedésnek a megvalósítását tartalmazza a `ValueType.Equals()` függvény. Sajnos a `ValueType.Equals()` megvalósítása nem túl hatékony. A `ValueType.Equals()` az összes értéktípus alaposztálya. A helyes működés érdekében minden származtatott típusnál az összes tagváltozót össze kell hasonlítania, anélkül, hogy az objektum futásidejű típusát ismerné. A C# nyelvben ezt visszatekintéssel (reflection) lehet elérni. Ahogy azt a 44. tippnél látni fogjuk, a visszatekintésnek számos hátránya van, különösen, ha a minél jobb teljesítmény elérése az egyik célunk. Az egyenlőség azon alapvető dolgok egyike, amit sokszor használunk a programokban, vagyis érdemes odafigyelni a teljesítményre. Szinte bármilyen körülmények között képesek vagyunk egy gyorsabb változattal felülbírálni az `Equals()` függvényt, bármely értéktípusról legyen is szó. Az értéktípusokra vonatkozó tanács egyszerű: ha értéktípust készítünk, mindig bíráljuk felül a `ValueType.Equals()` tagfüggvényt.

A hivatkozási típusoknál csak akkor bíráljuk felül a példányok `Equals()` függvényét, ha meg akarjuk változtatni a típus jelentését. A .NET keretrendszer osztálykönyvtárának néhány osztálya a hivatkozások helyett értékjelentést használ az egyenlőségnél. Két karakterlánc objektum akkor egyenlő, ha a tartalmuk megegyezik. Két `DataRowView` objektum akkor egyenlő, ha ugyanarra a `DataRow` objektumra hivatkoznak. A lényeg az, hogy ha a típusunknak értékjelentést kell használnia (tehát tartalmakat hasonlítunk össze) hivatkozási jelentés (objektumok azonosságát hasonlítjuk össze) helyett, akkor mindig írjuk meg a példány `Object.Equals()` függvényének saját felülbírált változatát.

Most, hogy már tudjuk, mikor kell felülbírálnunk az `Object.Equals()` tagfüggvényt, ismerkedjünk meg ennek megvalósításával is. Az értéktípusok egyenlőségének számos következménye van a dobozolásra, és a 17. tippnél erről még beszélni fogunk. A hivatkozási típusoknál a példány tagfüggvényének előre meghatározott viselkedés szerint kell működnie, nehogy az osztály felhasználóinak különös meglepetésekben legyen része. Az általános minta így néz ki:

```
public class Foo
{
  public override bool Equals( object right )
  {
    // null ellenőrzése:
    // a this mutató soha nem null a C# tagfüggvényekben
    if (right == null)
      return false;

    if (object.ReferenceEquals( this, right ))
      return true;

    // a magyarázatot lásd lejjebb
    if (this.GetType() != right.GetType())
      return false;

    // Itt hasonlítjuk össze a this típus tartalmát:
    return CompareFooMembers(
      this, right as Foo );
  }
}
```

Először is, az `Equals()` soha nem válthat ki kivételeket, hiszen ennek nem sok értelme lenne. Két változó vagy egyenlő, vagy nem. Más hibalehetőség nem nagyon van. Bármilyen hiba esetén adjunk vissza hamis (`false`) értéket, például ha `null` hivatkozással találkozunk, vagy a paraméterek nem megfelelő típusúak. Most pedig haladjunk végig a tagfüggvényen, hogy lássuk, melyik ellenőrzés miért van ott, ahol van, és hogy mely ellenőrzéseket lehet elhagyni. Elsőként azt ellenőrizzük, hogy a jobb oldali objektum `null`-e. A `this` hi-

vatkozást nem ellenőrizzük, mert a C# nyelvben az sosem lehet `null`. A CLR kivételt vált ki, mielőtt meghívná egy példány tagfüggvényét egy `null` hivatkozáson keresztül. A következő vizsgálat meghatározza, hogy a két objektumhivatkozás azonos-e, vagyis ellenőrzi az objektumok azonosságát. Ez az összehasonlítás nagyon hatékony, és az egyenlő objektumazonosság garantálja az azonos tartalmat.

A következő vizsgálattal eldöntjük, hogy az összehasonlított objektumok típusa megegyezik-e. Ennek pontos formája rendkívül fontos. Először is figyeljük meg, hogy nem csak úgy feltételezzük, hogy a `this` `Foo` típusú, hanem meghívjuk a `this.GetType()` tagfüggvényt, hiszen a tényleges típus a `Foo` típusból származtatott osztály is lehet. Másodszor, a kód az összehasonlított objektumok pontos típusát ellenőrzi. Az nem elegendő, hogy a jobb oldali paramétert át lehessen alakítani az aktuális típusra. Egy ilyen vizsgálat két alig észrevehető hibát is eredményezhet. Vegyük az alábbi példát, ami egy kis öröklődési rendszerre épül:

```
public class B
{
  public override bool Equals( object right )
  {
    // null ellenőrzése:
    if (right == null)
      return false;

    // Hivatkozás azonosságának ellenőrzése:
    if (object.ReferenceEquals( this, right ))
      return true;

    // Itt gondok vannak, a magyarázatot lásd lejjebb
    B rightAsB = right as B;
    if (rightAsB == null)
      return false;

    return CompareBMembers( this, rightAsB );
  }
}

public class D : B
{
  // stb.
  public override bool Equals( object right )
  {
    // null ellenőrzése:
    if (right == null)
      return false;

    if (object.ReferenceEquals( this, right ))
      return true;
```

```
        // Itt gondok vannak
        D rightAsD = right as D;
        if (rightAsD == null)
          return false;

        if (base.Equals( rightAsD ) == false)
          return false;

        return CompareDMembers( this, rightAsD );
      }

    }

    // A vizsgálat:
    B baseObject = new B();
    D derivedObject = new D();

    // 1. összehasonlítás
    if (baseObject.Equals(derivedObject))
      Console.WriteLine( "Equals" );
    else
      Console.WriteLine( "Not Equal" );

    // 2. összehasonlítás
    if (derivedObject.Equals(baseObject))
      Console.WriteLine( "Equals" );
    else
      Console.WriteLine( "Not Equal" );
```

Bármilyen lehetséges körülmény között arra számítanánk, hogy az `Equal` vagy a `Not Equal` üzenetet kétszer írja ki a program. A fenti kódban lévő hibák miatt azonban nem ez a helyzet. A második összehasonlítás soha nem ad vissza igaz értéket. A `B` típusú alapobjektumot soha nem lehet `D` típusúra alakítani. Az első összehasonlítás eredménye viszont lehet igaz. A származtatott `D` típusú objektum burkoltan átalakítható `B` típusúvá. Ha a jobb oldali paraméter `B` tagjai passzolnak a bal oldali paraméter `B` tagjaihoz, akkor a `B.Equals()` egyenlőnek tekinti az objektumokat. A tagfüggvényünk annak ellenére egyenlőnek vette a két objektumot, hogy azok különböző típusúak. Ezzel megsértettük az `Equals` függvény szimmetrikus tulajdonságát. A szerkezet az objektumhierarchiáknál alkalmazott automatikus típusátalakítások miatt nem működik.

Ha ezt írjuk, azzal a `D` objektumot egyértelműen `B` típusúra alakítjuk:

```
baseObject.Equals( derived )
```

Ha a `baseObject.Equals()` azt találja, hogy a típusában megadott mezők stimmelnek, akkor a két objektum egyenlő lesz. Ha viszont ezt írjuk, akkor a `B` objektumot nem lehet `D` objektummá alakítani:

```
derivedObject.Equals( base )
```

A `B` objektumot nem tudjuk `D` objektummá alakítani. A `derivedObject.Equals()` tagfüggvény mindig hamis értéket ad vissza. Ha nem ellenőrizzük az objektumok pontos típusát, akkor könnyen ebbe a helyzetbe kerülhetünk, amikor is az összehasonlítás sorrendje is számít.

Van még egy másik dolog is, amihez tartanunk kell magunkat, amikor felülbíráljuk az `Equals()` tagfüggvényt. Az alaposztályt csak akkor szabad meghívnunk, ha az alapváltozatot nem a `System.Object` vagy a `System.ValueType` adja. Egy példa erre az előbbi kód. A `D` osztály az alaposztályában, a `B` osztályban meghatározott `Equals()` tagfüggvényt hívja meg. A `B` osztály viszont nem a `baseObject.Equals()` függvényt hívja meg, hanem a `System.Object` osztályban található változatot, ami csak akkor ad vissza igaz értéket, amikor a két paraméter ugyanarra az objektumra hivatkozik. Ez nem az, amit mi szeretnénk, hiszen ha az lenne, akkor eleve nem írtuk volna meg a saját tagfüggvényünket.

A szabály tehát az, hogy értéktípusok létrehozásakor mindig felülbíráljuk az `Equals()` függvényt, a hivatkozási típusoknál pedig csak akkor, ha nem akarjuk, hogy a hivatkozási típus a `System.Object` osztály által meghatározott hivatkozási jelentést kövesse. Ha saját `Equals()` függvényt írunk, akkor mindig kövessük a fent bemutatott megvalósítást. Ha felülbíráljuk az `Equals()` tagfüggvényt, akkor ezt meg kell tennünk a `GetHashCode()` függvénnyel is. Ennek részleteit lásd a 10. tippnél.

Hárommal megvolnánk, jöhet a negyedik: az `operator==()`. Ha egy értéktípust hozunk létre, akkor mindig írjuk át az `operator==()` függvényt. Ennek magyarázata éppen az, mint az `Equals()` függvény esetében. Az alapértelmezett változat visszatekintést használ a két értéktípus tartalmának összehasonlításához. Ez sokkal kevésbé hatékony, mint bármilyen általunk elkészített megvalósítás, tehát mindig írjuk meg a saját változatunkat. Fogadjuk meg a 17. tipp tanácsait a dobozolás elkerülése érdekében, ha értéktípusokat hasonlítunk össze.

Remélhetőleg észrevettük, hogy nem azt írtuk, hogy mindig írjuk át az `operator==()` függvényt, amikor felülbíráljuk a példányok `Equals()` tagfüggvényét, hanem azt, hogy írjuk át az `operator==()` függvényt, amikor értéktípusokat hozunk létre. Ritka az az eset, amikor egy hivatkozási típus létrehozásakor át kell írnunk az `operator==()` függvényt. A .NET keretrendszer osztályai elvárják, hogy az `operator==()` a hivatkozási jelentést kövesse minden hivatkozási típusnál.

A C# nyelv négy módszert kínál az egyenlőség vizsgálatához, de ezek közül csak kettő esetében kell elgondolkoznunk azon, hogy elkészítsük-e belőlük saját változatainkat. A statikus `Object.ReferenceEquals()` és a statikus `Object.Equals()` tagfüggvényeket soha ne bíráljuk felül, mert azok a futásidejű típusoktól függetlenül mindig a helyes összehasonlítást végzik el. Az értéktípusoknál a jobb teljesítmény érdekében mindig bíráljuk felül a példányok `Equals()` tagfüggvényét és az `operator==()` függvényt. A hivatkozási típusoknál akkor bíráljuk felül a példányok `Equals()` tagfüggvényét, ha azt szeretnénk, hogy az egyenlőség valami mást jelentsen, mint az objektumok azonosságát. Ugye milyen egyszerű?

10. tipp

Ismerjük meg a GetHashCode() buktatóit

Ez az egyetlen olyan tipp a könyvben, amit teljes egészében egy olyan függvénynek szentelünk, amit sosem szabad megírnunk. A `GetHashCode()` függvény csak egyetlen helyen használatos, mégpedig a kivonat alapú gyűjtemények kulcsaihoz tartozó hasítóértékékek meghatározásához. Általában ilyenek a `Hashtable` és a `Dictionary` tárolók. Ez nagy szerencse, mert van néhány gond a `GetHashCode()` alaposztálybeli megvalósításával. A hivatkozási típusoknál működik, de nem túl hatékony, az értéktípusoknál azonban az alaposztály változata gyakran téved. És ez még nem minden: nagyon is valószínű, hogy nem is lehet úgy megírni a `GetHashCode()` függvényt, hogy hatékony is legyen, meg helyesen is működjön. Nincs még egy olyan függvény, ami annyi vitát és zűrzavart okozott volna, mint a `GetHashCode()`. Ha minket is megzavart már, akkor feltétlenül olvassunk tovább.

Amennyiben egy olyan típust határozunk meg, amit soha nem fogunk egy tároló kulcsaként használni, akkor ez az egész nem is fontos. Az ablakos és webes vezérlőelemeket, illetve az adatbázis-kapcsolatokat megtestesítő típusokat valószínűleg soha nem fogjuk egy gyűjtemény kulcsaként használni. Ezekben az esetekben semmit nem kell tennünk. Minden hivatkozási típusnak helyes hasítókódja lesz, ha nem is túl hatékony. Ha az értéktípusaink a tanácsnak megfelelően nem változóak (lásd a 7. tippet), akkor ebben az esetben az alapértelmezett megvalósítás mindig működik, bár szintén nem túl hatékonyan. A legtöbb típusnál, amit létrehozunk, a legjobb, ha tudomást sem veszünk a `GetHashCode()` létezéséről.

Eljön azonban a nap, amikor egy olyan típust készítünk, amit egy hasítótábla kulcsának szánunk, s így meg kell írnunk a `GetHashCode()` függvény saját magunk készítette megvalósítását, tehát ne hagyjuk abba az olvasást. A kivonat (hasítótábla) alapú tárolók hasítókódokat használnak a keresések optimalizálásához. Minden objektum előállít egy egész számot, amit hasítókódnak hívunk. Az objektumok a hasítókódjuk alapján különböző tárolómezőkbe (buckets) kerülnek. Ha meg akarunk találni egy objektumot, akkor bekérjük a kulcsát, és csak a hozzá tartozó tárolómezőben keresünk. A .NET keretrendszerben minden objektumnak van egy hasítókódja, amit a `System.Object.GetHashCode()` függvény ad meg.

A GetHashCode() bármely túlterhelésének meg kell felelnie ennek a három szabálynak:

1. Ha két objektum egyenlő (az operator== meghatározása szerint), akkor ugyanazt a hasítókódot kell előállítaniuk, különben a hasítókódok segítségével nem lehetne megtalálni a tárolókban az objektumokat.
2. Bármely A objektumra, az A.GetHashCode() függvénynek invariánsnak kell lennie a példányra nézve. Mindegy, hogy milyen tagfüggvényeket hívunk meg az A-n, az A.GetHashCode() tagfüggvénynek mindig ugyanazt az értéket kell visszaadnia. Ez biztosítja, hogy egy adott tárolómezőbe helyezett objektum mindig a helyes tárolómezőben lesz.
3. A hasítófüggvény az egész értékek véletlenszerű eloszlását állítsa elő az összes bemenetet figyelembe véve. Ettől lesznek hatékonyak a kivonat alapú tárolók.

Egy helyesen működő és hatékony hasítófüggvény megírásához részletesen ismernünk kell az adott típust, hogy biztosíthassuk a 3. szabály betartását. A System.Object és System.ValueType osztályokban meghatározott változatok erre nem képesek. Ezeknek a változatoknak a legjobb alapértelmezett viselkedéssel kell rendelkezniük úgy, hogy közben szinte semmilyen információval nem rendelkeznek a típusunkról. Az Object.GetHashCode() egy belső mezőt használ a System.Object osztályban a hasítóérték előállításához. Minden létrehozott objektum egy egyedi objektum kulcsot kap, amit a program az objektum létrehozásakor egy egészként raktároz el. Ezek a kulcsok 1-től kezdődnek, és az érték eggyel nő minden egyes alkalommal, amikor létrehozunk egy bármilyen típusú objektumot. Az objektum azonosító mezőjét a System.Object konstruktorban állítja be a program, és ezt később már nem módosíthatjuk. Az Object.GetHashCode() ezt az értéket adja vissza hasítókódként az adott objektumhoz.

Vizsgáljuk most meg az Object.GetHashCode() függvényt a három szabály ismeretében. Ha két objektum egyenlő, akkor az Object.GetHashCode() ugyanazt a hasítóértéket adja vissza, hacsak át nem írtuk az operator== meghatározását. A System.Object osztály operator==() változata az objektumok azonosságát vizsgálja. A GetHashCode() a belső objektumazonosító mezőt adja vissza. A dolog működik. Ha viszont megadtunk egy saját operator== változatot, akkor a GetHashCode() saját változatát is el kell készítenünk, hogy gondoskodjunk az első szabály betartásáról. (Lásd a 9. tippet az egyenlőséggel kapcsolatban.)

A második szabály betartásával nincs gond: ha egyszer létrehoztunk egy objektumot, annak hasítókódja már soha többet nem változik.

A harmadik szabály, vagyis az egészek véletlenszerű eloszlása az összes bemenetet figyelembe véve, már nem teljesül. Egy számsorozat nem véletlenszerű eloszlása az egész számoknak, hacsak nem hatalmas számú objektumot hozunk létre. Az Object.GetHashCode() által előállított hasítókódok az egész számok alsó tartományában összpontosulnak.

Ez azt jelenti, hogy az Object.GetHashCode() működése helyes, de nem hatékony. Ha létrehozunk egy hasítótáblát egy általunk meghatározott hivatkozási típushoz, akkor a System.Object alapértelmezett viselkedésének eredménye egy működő, de lassú hasítótábla lesz. Ha olyan hivatkozási típust hozunk létre, amit hasítókulcsnak szánunk, akkor bíráljuk felül a GetHashCode() függvényt, hogy az egészek jobb eloszlását kapjuk hasítóértékekként az adott típusunkhoz.

Mielőtt rátérnénk a GetHashCode felülbírálatának mikéntjére, az alábbi részben a fenti három szabályt figyelembe véve megvizsgáljuk a ValueType.GetHashCode() függvényt. A System.ValueType osztály felülbírálja a GetHashCode() függvényt, megadva ezzel egy alapértelmezett viselkedést az értéktípusoknak. Ez a változat a típusban elsőként meghatározott mező alapján adja vissza a hasítókódot. Nézzük meg ezt a példát:

```
public struct MyStruct
{
  private string   _msg;
  private int      _id;
  private DateTime _epoch;
}
```

A MyStruct objektumok azt a hasítókódot adják vissza, amit a _msg mező állít elő. Az alábbi kódrészlet mindig igaz értéket ad vissza:

```
MyStruct s = new MyStruct( );
return s.GetHashCode( ) == s._msg.GetHashCode( );
```

Az első szabály szerint, ha két objektum egyenlő (az operator== meghatározása szerint), akkor hasítókódjuk is ugyanaz lesz. Az értéktípusok a legtöbb esetben megfelelnek ennek a szabálynak, de a szabályt meg is sérthetjük, éppen úgy, ahogy a hivatkozási típusoknál. A ValueType.operator==() függvény a struktúra első mezőjét vizsgálja, a többi mezővel együtt. Ez eleget tesz az 1. szabálynak. Amíg az operator== felülbírálatát úgy írjuk meg, hogy az az első mezőt használja, a dolog működni fog. De minden olyan struktúra, amelynek első mezője nem vesz részt az egyenlőség meghatározásában, megsérti a szabályt, és elrontja a GetHashCode() működését.

A második szabály szerint a hasítókódnak invariánsnak kell lennie a példányra nézve. Ennek a szabálynak csak akkor felel meg a típus, ha az első mező egy nem változó értéktípus. Ha az első mező értéke megváltozhat, akkor a hasítókód értéke is megváltozhat. Ezzel megszegtük a szabályt. Bizony, a GetHashCode() egyik olyan struktúránál sem fog működni, amelyiknél az első mező értéke megváltozhat a típus élettartama alatt. Ez még egy indok arra, hogy miért érdemes nem változó értéktípusokat használnunk (lásd a 7. tippet).

A harmadik szabály az első mező típusától és annak használatától függ. Ha az első mező az egész számokon vett véletlen eloszlást állít elő, és az első mező értékei megoszlanak a struktúra értékei között, akkor a struktúra is egyenletes eloszlást állít elő. Ha viszont az első mező sokszor ugyanazt az értéket kapja, akkor megszegjük a szabályt. Gondoljunk bele, hogy mi lesz az eredménye ennek a korábbi struktúrán elvégzett apró változtatásnak:

```
public struct MyStruct
{
  private DateTime _epoch;
  private string   _msg;
  private int      _id;
}
```

Ha az `_epoch` az aktuális dátum értékét kapja (az időpont nélkül), akkor az összes adott napon létrehozott `MyStruct` objektumnak ugyanaz lesz a hasítókódja. Ez nem teszi lehetővé az összes hasítókód egyenletes eloszlását.

Összefoglalva az alapértelmezett viselkedést, az `Object.GetHashCode()` a hivatkozási típusoknál helyesen működik, bár nem feltétlenül hatékony eloszlását állítja elő a hasítókódoknak. (Ha felülbíráljuk az `Object.operator==()` függvényt, akkor elronthatjuk a `GetHashCode()` működését.) A `ValueType.GetHashCode()` csak akkor működik, ha a struktúra első mezője csak olvasható. A `ValueType.GetHashCode()` csak akkor állít elő hatékony hasítókódokat, ha a struktúra első mezőjének értékei jól megoszlanak a bemenetek között.

Ha jobb hasítókódokat akarunk, akkor bizonyos megszorításokat kell alkalmaznunk a típusunkra. Vizsgáljuk meg ismét a három szabályt, de ezúttal azzal a céllal, hogy elkészítsük a `GetHashCode()` működő megvalósítását.

Először is, ha két objektum egyenlő az `operator==()` meghatározása alapján, akkor ugyanazt a hasítókódot kell visszaadniuk. Minden olyan tulajdonságnak és adatnak, amit felhasználunk a hasítókód előállításához, részt kell vennie az egyenlőség megállapításában. Ez nyilván azt jelenti, hogy ugyanazokat a tulajdonságokat fogjuk használni az egyenlőség eldöntéséhez és a hasítókód előállításához is. Arra persze lehetőségünk van, hogy az egyenlőség eldöntéséhez olyan más tulajdonságokat is használjunk, amelyeket a hasítókód kiszámításához nem. Az alapértelmezett viselkedés pontosan ezt teszi, de ez gyakran a 3. szabály megsértéséhez vezet. Mindkét számításhoz használjuk tehát ugyanazokat az adatelemeket.

A második szabály az, hogy a GetHashCode() visszatérési értéke invariáns legyen a példányra nézve. Képzeljük el, hogy meghatároztunk egy Customer elnevezésű hivatkozási típust:

```
public class Customer
{
  private string _name;
  private decimal _revenue;

  public Customer( string name )
  {
    _name = name;
  }

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

  public override int GetHashCode()
  {
    return _name.GetHashCode();
  }
}
```

Tegyük fel, hogy végrehajtjuk az alábbi kódrészletet:

```
Customer c1 = new Customer( "Acme Products" );
myHashMap.Add( c1, orders );
// Hoppá, hibás a név:
c1.Name = "Acme Software";
```

A c1 elveszett a hasítótérképben. Amikor a c1 bekerült a térképbe, a hasítókódot az "Acme Products" karakterláncból állította elő a program. Miután a megrendelő nevét "Acme Software"-re módosítottuk, a hasítókód megváltozott, hiszen az immár az "Acme Software" névből áll elő. A c1 az "Acme Products" által kijelölt tárolómezőben van, pedig az "Acme Software" által kijelölt tárolómezőben kellene, hogy legyen. Elvesztettünk egy megrendelőt a saját gyűjteményünkön belül. Azért veszett el, mert a hasítókód nem invariáns a példányra nézve. Az objektum tárolása után módosítottuk a tárolómezőt.

A fenti helyzet csak akkor fordulhat elő, ha a Customer hivatkozási típus. Az értéktípusok másképpen rosszalkodnak, és velük is gondunk akadhat. Ha a megrendelő értéktípus, akkor a hasítótérképbe a c1 másolata kerül. Az utolsó sorban végrehajtott módosítás a meg-

rendelő nevén nem érinti a hasítótérképben tárolt másolatot. Mivel a bedobozolás és kicsomagolás során szintén másolatok készülnek, elég valószínűtlen, hogy meg tudjuk változtatni egy értéktípus tagjait, miután az objektumot hozzáadtuk egy gyűjteményhez.

Az egyetlen mód a 2. szabály betartására, ha úgy határozzuk meg a hasítófüggvényt, hogy az az objektum valamely invariáns tulajdonsága(i) alapján állítsa elő a hasítókód értékét. A `System.Object` úgy alkalmazkodik ehhez a szabályhoz, hogy az objektum azonosítóját használja, ami nem változik. A `System.ValueType` abban bízik, hogy a típusunk első mezőjének értéke nem változik. Anélkül, hogy nem változóvá tennénk a típust, mi is csak bízhatunk ebben. Ha olyan értéktípust határozunk meg, amit egy kivonat alapú tároló kulcsaként szeretnénk használni, akkor az mindig nem változó típusú legyen. Ha nem fogadjuk meg ezt a tanácsot, akkor a típusunk felhasználói szinte bizonyosan tönkreteszik azokat a hasítótáblákat, amelyek a mi típusunkra épülnek. Térjünk vissza a `Customer` osztályhoz, és változtassuk meg úgy, hogy a megrendelő neve ne változhasson:

```
public class Customer
{
  private readonly string _name;
  private decimal _revenue;

  public Customer( string name ) :
    this ( name, 0 )
  {
  }

  public Customer( string name, decimal revenue )
  {
    _name = name;
    _revenue = revenue;
  }

  public string Name
  {
    get { return _name; }
  }

  // A név módosítása új objektum visszaadásával:
  public Customer ChangeName( string newName )
  {
    return new Customer( newName, _revenue );
  }

  public override int GetHashCode()
  {
    return _name.GetHashCode();
  }
}
```

A név nem változóvá tétele miatt ezután másképp kell megváltoztatnunk a megrendelők nevét az objektumokban:

```
Customer c1 = new Customer( "Acme Products" );
myHashMap.Add( c1,orders );
// Hoppá, hibás a név:
Customer c2 = c1.ChangeName( "Acme Software" );
Order o = myHashMap[ c1 ] as Order;
myHashMap.Remove( c1 );
myHashMap.Add( c2, o );
```

Törölnünk kell az eredeti megrendelőt, majd megváltoztatjuk a nevet, és az új megrendelő objektumot hozzáadjuk a hasítótáblához. Ez kissé nehézkesebb, mint az első változat, de működik. Az előző változat lehetővé tette, hogy a programozó hibás kódot írjon. Azzal, hogy kikényszerítjük a hasítókód előállításához használt tulajdonságok állandóságát, egyben biztosítjuk a helyes viselkedést is. A típusunk felhasználói nem tudnak hibázni. Való igaz, ez a változat kicsit több munkát igényel. Arra kényszerítjük a fejlesztőket, hogy több kódot írjanak meg, de pusztán azért, mert ez az egyetlen módja a helyes kód előállításának. Mindig gondoskodjunk róla, hogy a hasítókód előállításához használt adattagok állandók legyenek.

A harmadik szabály szerint a `GetHashCode()` függvénynek az egész értékek véletlenszerű eloszlását kell előállítania az összes bemenetet figyelembe véve. Ennek a követelménynek a kielégítése a típusunk részletein múlik. Ha létezne egy varázslatos képlet, akkor annak megvalósítása ott lenne a `System.Object` osztályban és erre a tippre nem lenne szükség. Egy gyakran sikerrel alkalmazott algoritmus a következő. A `GetHashCode()` által visszaadott érték és a típus minden mezőjének értéke között végezzünk el `XOR` műveleteket. Ha a típusban változó értékű mezők is vannak, akkor azokat ne vegyük figyelembe az érték kiszámításánál.

A `GetHashCode()` függvény használata különleges követelményeket támaszt. Az egyenlő objektumoknak egyenlő hasítókódot kell előállítaniuk, a hasítókódoknak invariánsnak kell lenniük az adott objektumra nézve, és egyenletes eloszlással kell rendelkezniük a hatékonyság érdekében. Mindhárom követelménynek csak a nem változó típusok képesek megfelelni. Más típusoknál támaszkodjunk az alapértelmezett viselkedésre, de legyünk tisztában annak buktatóival.

11. tipp

Használjunk foreach ciklusokat

A C# nyelv `foreach` ciklusa nem csak egyszerűen egy változata a `do`, a `while` és a `for` ciklusoknak. A legjobb bejáró (iterációs) kód a `foreach` ciklusokból áll elő, bármilyen gyűjteményünk is legyen. Az utasítás meghatározása a .NET keretrendszer gyűjteményeinek felületéhez kapcsolódik, ezért a C# fordító mindig az adott típusú gyűjteménynek legjobban megfelelő kódot állítja elő. A gyűjtemények bejárásához használjunk `foreach` ciklusokat a többi ciklusvezérlő utasítás helyett. Vizsgáljuk meg ezt a három ciklust:

```
int [] foo = new int[100];
// 1. ciklus:
foreach ( int i in foo)
  Console.WriteLine( i.ToString( ));

// 2. ciklus:
for ( int index = 0;
  index < foo.Length;
  index++ )
  Console.WriteLine( foo[index].ToString( ));

// 3. ciklus:
int len = foo.Length;
for ( int index = 0;
  index < len;
  index++ )
  Console.WriteLine( foo[index].ToString( ));
```

A jelenlegi és a jövőbeni C# fordítók (az 1.1 és az újabb változatok) esetében az 1. ciklus a legjobb. Még gépelnünk sem kell annyit, ami a termelékenységünket is megnöveli. (A C# 1.0 fordító sokkal lassabb kódot készített az 1. ciklusból, tehát ennél a változatnál a 2. ciklus a legjobb.) A 3. ciklus, amit a legtöbb C és C++ programozó a legjobbnak tartana, valójában a legrosszabb választás. Azzal, hogy a ciklus elé tesszük a `Length` változót, olyan módosítást hajtunk végre, ami csökkenti az esélyét annak, hogy a JIT fordító eltávolítsa a tartományellenőrzést a ciklus belsejéből.

A C# kód biztonságosan kezelt környezetben fut. Minden memóriahely ellenőrzés alatt áll, még a tömbindexek is. Kicsit szabadosan, a 3. ciklus tényleges kódja valahogy így néz ki:

```
// A 3. ciklus a fordítás után:
int len = foo.Length;
for ( int index = 0;
  index < len;
  index++ )
{
```

```
    if ( index < foo.Length )
      Console.WriteLine( foo[index].ToString( ));
    else
      throw new IndexOutOfRangeException( );
}
```

A JIT C# fordítónak egyszerűen nincs ínyére, hogy így próbálunk meg segíteni neki. Azzal, hogy kivittük a ciklus elé a `Length` tulajdonság elérését, csupán azt értük el, hogy a JIT fordítónak még több dolga lett, és lassabb kódot állított elő. A CLR biztosítékai közül az egyik az, hogy nem írhatunk olyan kódot, ami túlszaladna azon a memóriaterületen, ami a változóink birtokában van. Futásidőben a program a tényleges tömb korlátait vizsgálja (nem a `len` változóét), mielőtt hozzáférne egy adott tömbelemhez. Vagyis egy korlátellenőrzést kapunk kettő áráért.

Persze a ciklus minden végrehajtása során fizetnünk kell a tömbindex ellenőrzéséért, és ezt kétszer kell megtennünk. Az 1. és a 2. ciklus azért gyorsabb, mert a C# fordító és a JIT fordító ellenőrizni tudják, hogy a ciklus korlátai garantáltan biztonságosak-e. Ha a ciklusváltozó hossza eltér a tömb hosszától, akkor a ciklus minden végrehajtásánál megtörténik a korlátok ellenőrzése.

Az, hogy a `foreach` és a tömbök használata nagyon lassú kódot eredményezett az eredeti C# fordítónál, a dobozolással (boxing) van összefüggésben, amiről részletesen szó lesz a 17. tippnél. A tömbök típusellenőrzöttek. A `foreach` immár más IL kódot állít elő a tömbökhöz, mint más gyűjteményekhez. A tömbös változata nem használja az `IEnumerator` felületet, amihez bedobozolásra és kicsomagolásra lenne szükség:

```
IEnumerator it = foo.GetEnumerator( );
while( it.MoveNext( ))
{
  int i = (int) it.Current; // itt bedobozolunk és kicsomagolunk
  Console.WriteLine( i.ToString( ) );
}
```

A `foreach` utasítás ehelyett az alábbi szerkezetet állítja elő a tömböknél:

```
for ( int index = 0;
  index < foo.Length;
  index++ )
  Console.WriteLine( foo[index].ToString( ));
```

A `foreach` mindig a legjobb kódot állítja elő. Nem kell megjegyeznünk, hogy melyik szerkezet adja a legjobb kódot: használjuk a `foreach`-et, és a többit elvégzi helyettünk a fordító.

Ha a hatékonyság nem elég, akkor gondoljunk bele a nyelvek közötti együttműködés lehetőségébe. Néhány földi lelket (bizony, többségük más programnyelvet használ) átjár az az erős hit, hogy a sorszámok 1-től és nem 0-tól kezdődnek. Hiába is próbálkozunk, lehetetlen lebeszélni őket erről, a .NET fejlesztőcsapata mégis megpróbálta. Ahhoz, hogy a C# nyelvben egy olyan tömböt hozzunk létre, aminek sorszámozása egy 0-tól eltérő értékkel indul, valami ilyesmit kell írnunk:

```
// Létrehozunk egy egydimenziós tömböt
// A korlátai [1..5] lesznek
Array test = Array.CreateInstance( typeof( int ),
new int[ ]{ 5 }, new int[ ]{ 1 });
```

A fenti kódnak önmagában is elég ijesztőnek kellene lennie ahhoz, hogy mindenki 0-tól kezdje a tömbök elemeinek sorszámozását. De vannak rendkívül makacs emberek. Csináljunk bármit, ők akkor is 1-től kezdik a számozást. Ez szerencsére azok közé a gondok közé tartozik, amiket lepasszolhatunk a fordítónak. A tömböt a `foreach` utasítás segítségével járjuk be:

```
foreach( int j in test )
  Console.WriteLine ( j );
```

A `foreach` utasítás tudja, hogyan kell ellenőrizni a tömb felső és alsó korlátait, tehát nekünk ezzel nem kell foglalkoznunk. Ráadásul éppen olyan gyors, mint egy „kézzel" megírt `for` ciklus, attól függetlenül, hogy milyen alsó korlátot használ valaki.

A `foreach` használatának más előnyei is vannak. A `loop` változó csak olvasható: a gyűjteményben található objektumokat nem tudjuk lecserélni a `foreach` segítségével. Ehhez jön még a mindig a megfelelő típusra alakítás. Ha a gyűjtemény nem a megfelelő típusú objektumokból áll, akkor a bejárás kivételt vált ki.

A `foreach` a többdimenziós tömbök esetében is hasonló előnyöket kínál. Tegyük fel, hogy egy sakktáblát készítünk. Ekkor az alábbi két kódrészletet írnánk meg:

```
private Square[,] _theBoard = new Square[ 8, 8 ];

// a kód egy másik részén:
for ( int i = 0; i < _theBoard.GetLength( 0 ); i++ )
  for( int j = 0; j < _theBoard.GetLength( 1 ); j++ )
    _theBoard[ i, j ].PaintSquare( );
```

A tábla kifestését egyszerűbben is elvégezhetjük:

```
foreach( Square sq in _theBoard )
  sq.PaintSquare( );
```

A foreach utasítás előállítja a megfelelő kódot, ami a tömb összes dimenzióját bejárja. Ha később egy háromdimenziós sakktáblát készítünk, a foreach utasítás ugyanúgy működni fog. A másik cikluson viszont változtatnunk kell:

```
for ( int i = 0; i < _theBoard.GetLength( 0 ); i++ )
  for( int j = 0; j < _theBoard.GetLength( 1 ); j++ )
    for( int k = 0; k < _theBoard.GetLength( 2 ); k++ )
      _theBoard[ i, j, k ].PaintSquare( );
```

Sőt, a foreach ciklus még egy olyan többdimenziós tömbbel is működne, amelyiknek minden irányban más alsó korlátja van. Ilyen kódot persze még példaként sem vagyok hajlandó leírni. De ha valaki más megír egy ilyen gyűjteményt, a foreach azzal is elbánik.

A foreach további rugalmasságot ad a kezünkbe azzal, hogy a kód nagy részéhez nem kell hozzányúlnunk, ha egyszer rájövünk, hogy meg kell változtatnunk egy már létező tömb mögött meghúzódó adatszerkezetet. A téma tárgyalását egy egyszerű tömbbel kezdtük:

```
int [] foo = new int[100];
```

Tegyük fel, hogy egy későbbi időpontban rájövünk, hogy olyan adottságokra van szükségünk, amelyeket a tömbosztály segítségével nem túl egyszerű megvalósítani. Ekkor a tömböt egyszerűen átalakíthatjuk egy ArrayList gyűjteménnyé:

```
// Kezdeti méret megadása:
ArrayList foo = new ArrayList( 100 );
```

A saját for ciklusok ezután működésképtelenek lesznek:

```
int sum = 0;
for ( int index = 0;
  // nem lehet lefordítani:
  // az ArrayList a Count és nem a Length változót használja
  index < foo.Length;
  index++ )
  // nem lehet lefordítani:
  // a foo[index] típusa objektum, nem int
  sum += foo[ index ];
```

A foreach ciklus fordítása során mindig különböző kódot kapunk, mégpedig úgy, hogy minden automatikusan a megfelelő típusú lesz. Semmin sem kell változtatnunk. Ez nem csak a szabványos gyűjteményosztályokra igaz; bármilyen gyűjteménnyel használhatjuk a foreach utasítást.

A típusaink felhasználói bejárhatják a tagokat, ha támogatjuk a .NET környezet gyűjteményekre vonatkozó szabályait. A `foreach` akkor tekint valamit gyűjteménynek, ha az adott osztály az alábbi néhány tulajdonságból legalább az egyikkel rendelkezik.

A `GetEnumerator()` tagfüggvény jelenléte gyűjteményt csinál egy osztályból. Gyűjtemény típust úgy is készíthetünk, ha megírjuk az `IEnumerable` felület megvalósítását. Az `IEnumerator` felület megvalósításának megírásával is egy gyűjteménytípushoz jutunk. A `foreach` mindegyik változattal működik.

A `foreach` használatának az erőforrás-kezelést illetően is van egy további előnye. Az `IEnumerable` felület egy tagfüggvényt tartalmaz: ez a `GetEnumerator()`. A `foreach` utasítás egy felsorolható típushoz az alábbit állítja elő egy kis optimalizálással:

```
IEnumerator it = foo.GetEnumerator( ) as IEnumerator;
using ( IDisposable disp = it as IDisposable )
{
  while ( it.MoveNext( ))
  {
    int elem = ( int ) it.Current;
    sum += elem;
  }
}
```

A fordító automatikusan optimalizálja a `finally` blokkban található kódot, ha biztosan el tudja dönteni, hogy a felsorolás megvalósítja az `IDisposable` felületet. Számunkra viszont sokkal fontosabb, hogy lássuk, bármi is történjék, a `foreach` mindig helyes kódot eredményez.

A `foreach` nagyon sokrétű utasítás. Helyes kódot állít elő a tömbök alsó és felső korlátaihoz, képes többdimenziós tömbök bejárására, mindent a megfelelő típusúra alakít (és teszi ezt a leghatékonyabb szerkezettel), sőt a leghatékonyabb ciklikus szerkezeteket is vele készíthetjük el. A legjobb módja a gyűjtemények bejárásának. A segítségével olyan kódot hozhatunk létre, ami valószínűleg hosszabb ideig használható marad, és megírása már eleve egyszerűbb. Ez csak egy kis hatékonyságnövelés, de idővel a sok kicsi sokra megy.

2

Erőforrás-kezelés a .NET-ben

Annak az egyszerű ténynek, hogy a .NET programok kezelt környezetben futnak, komoly hatása van arra, hogy milyen programfelépítés vezet a C# nyelv hatékony használatához. Ha a lehető legjobban ki szeretnénk használni a környezet nyújtotta lehetőségeket, akkor a gondolkodásmódunkat teljesen át kell állítani a hagyományos környezetről a .NET CLR környezetre. Ez azt jelenti, hogy meg kell ismerkednünk a szemétgyűjtő (Garbage Collector) működésével. Ahhoz, hogy megérthessük a fejezet tippjeit, előbb nagy vonalakban ismernünk kell a .NET memóriakezelő környezetét. Kezdjük hát rögtön az áttekintéssel.

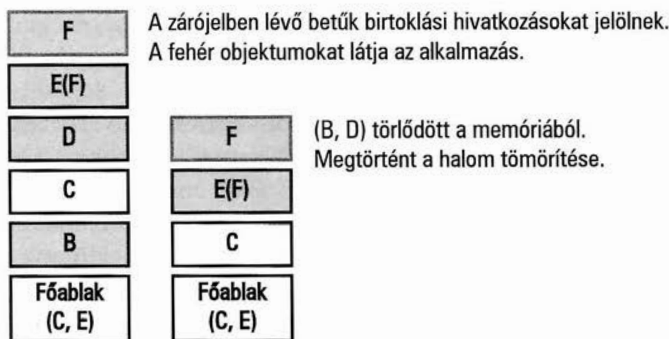
A Garbage Collector (GC) szabályozza helyettünk a kezelt memóriát. A hagyományos programozási környezetektől eltérően itt nem mi felelünk a memóriaszivárgásokért, a „lebegő" mutatókért, a kezdőérték nélküli mutatókért, és a többi memóriakezeléssel kapcsolatos problémáért. A szemétgyűjtő azonban nem csodafegyver: saját magunk után nekünk is ki kell takarítanunk. A kezeletlen erőforrásokért továbbra is mi felelünk. Ilyenek például az állományazonosítók, az adatbázis-kapcsolatok, a GDI+ objektumok, a COM objektumok, és más rendszerobjektumok.

Jó hír, hogy mivel a GC kezeli a memóriát, bizonyos programszerkezetek megvalósítása sokkal egyszerűbb, például körkörös hivatkozásokat – egyszerű kapcsolatokat vagy objektumok bonyolult hálózatát – sokkal könnyebb kialakítani. A GC Mark and Compact (jelölés és tömörítés) algoritmusa hatékonyan feltárja ezeket a kapcsolatokat, és az elérhetetlen objektumhálózatokat teljes egészében törli. A GC úgy dönti el, hogy mely objektumok elérhetetlenek, hogy az alkalmazás gyökerétől kezdve végighalad az objektumfán, nem pedig úgy, mint a COM, ami rákényszeríti az objektumokat a rájuk mutató hivatkozások nyilvántartására. A `DataSet` (adathalmaz) osztály jól példázza azt, hogy ez az algoritmus hogyan egyszerűsíti le az objektumok hovatartozásával kapcsolatos kérdéseket. A `DataSet` objektum `DataTable` (adattábla) objektumok gyűjteménye, a `DataTable` objektumok `DataRow` (adatsor) objektumoké, a `DataRow` objektumok pedig `DataItem` (adatelem) objektumoké. Minden `DataTable` tartalmaz egy `DataColumn` (adatoszlop) objektumok-

ból álló gyűjteményt is. A DataColumn objektumok határozzák meg az adatoszlopok típusait. Más hivatkozások is vannak a DataItem objektumoktól a megfelelő oszlopra. Minden DataItem objektum tartalmaz egy hivatkozást az őt tartalmazó DataRow objektumra is. A DataRow objektumok tartalmaznak egy hivatkozást a DataTable objektumukra, és végül minden elem tartalmaz egy hivatkozást az őt tartalmazó DataSet objektumra.

Ha ez nem lenne elég bonyolult, akkor gondoljuk meg, hogy készíthetünk DataView (adatnézet) objektumokat is, amelyekkel az adattáblák szűrt és rendezett változataihoz férhetünk hozzá. Ezeket egy DataViewManager objektum kezeli. A DataSet objektumot felépítő objektumháló telis-tele van hivatkozásokkal. A memória felszabadítása a GC feladata. Mivel a .NET keretrendszer tervezőinek nem kellett felszabadítaniuk ezeket az objektumokat, az objektumhivatkozások bonyolult hálózata sem okozott nekik gondot. Nem nekik kellett eldönteni, hogy milyen sorrendben történjen az objektumok hálójának felszabadítása, hiszen ez a GC dolga. A GC felépítése úgy egyszerűsíti le a problémát, hogy az objektumhálót teljes egészében szemétnek veszi. Miután az alkalmazás elereszti a DataSet objektumra mutató hivatkozását, az alá tartozó objektumokat már nem lehet elérni. Az nem számít, hogy a DataSet, DataTable és a háló más objektumai között még oda-vissza mutató hivatkozások vannak. Mivel ezeket az objektumokat nem lehet elérni az alkalmazásból, a GC szemében szemétnek számítanak.

A szemétgyűjtő saját szálán fut, miközben törli a programunk által nem használt memória tartalmát. Minden alkalommal, amikor lefut, egyben össze is tömöríti a halmot. A halom tömörítésével minden élő objektumot úgy helyez át, hogy az üres memóriahelyek egy folytonos blokkban helyezkedjenek el. A 2.1. ábrán egy pillanatfelvételt láthatunk a halomról a szemétgyűjtés előtt és után. A GC művelet végrehajtása után az összes szabad memória egy folytonos blokkba kerül.



2.1. ábra

A szemétgyűjtő nem csak törli a nem használt memóriát, de a memóriában található objektumokat is áthelyezi. Ezzel összetömöríti a felhasznált memóriát, és a lehető legnagyobb méretűvé teszi a szabad helyet.

Ahogy azt az imént megtanultuk, a memóriakezelés teljesen a szemétgyűjtő hatáskörébe tartozik. A többi rendszererőforrás kezelése viszont a mi feladatunk. Az egyéb rendszererőforrások felszabadításáról úgy gondoskodhatunk, hogy a típusunkban megadunk egy megsemmisítőt (finalizer). A rendszer mindig meghívja a szemétté vált objektumok megsemmisítőjét, mielőtt törölné őket a memóriából. Ezeket a tagfüggvényeket használhatjuk, illetve kell használnunk az objektumhoz tartozó kezeletlen erőforrások felszabadításához. Az objektum megsemmisítőjének meghívására valamikor az objektum szemétté válása után, de még azelőtt kerül sor, hogy a rendszer visszavenné tőle a hozzá tartozó memóriát. Ez a fajta nem előre meghatározott megsemmisítés azt jelenti, hogy nem befolyásolhatjuk az objektum használatának befejezése és megsemmisítőjének futása közti kapcsolatot. Ez nagy változás a C++ nyelvhez képest, és fontos következményei vannak a programok felépítését illetően. A tapasztalt C++ programozók rendszeresen írnak olyan osztályokat, amelyek a konstruktorukban lefoglalnak egy nélkülözhetetlen erőforrást, majd az osztály destruktorában szabadítják fel azt:

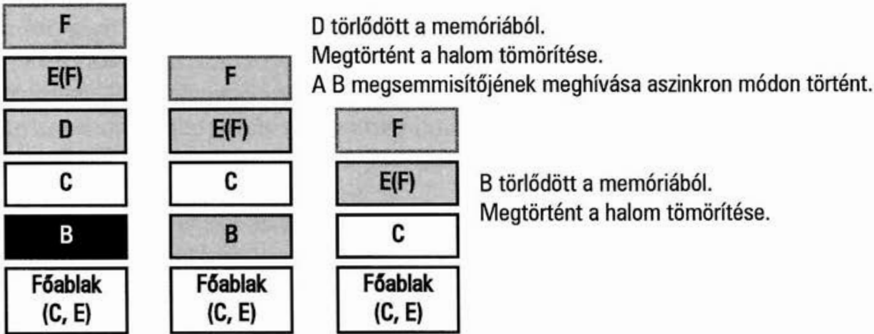
```
// A C++-ban helyes, a C#-ben helytelen:
class CriticalSection
{
public:
  // A konstruktor megszerzi a rendszererőforrást
  CriticalSection( )
  {
    EnterCriticalSection( );
  }

  // A destruktor felszabadítja a rendszererőforrást
  ~CriticalSection( )
  {
    ExitCriticalSection( );
  }
};

// Használat:
void Func( )
{
  // Az s élettartama határozza meg
  // a rendszererőforrás elérhetőségét
  CriticalSection s;
  // Elvégezzük a dolgunk.

  //...

  // A fordító előállít egy destruktorhívást
  // A program kilép a kritikus részből
}
```

Ez a jellegzetes C++ megoldás biztosítja, hogy az erőforrások felszabadítása során ne legyenek kivételek. A C# nyelvben azonban ez nem működik, de legalábbis pontosan így biztosan nem. Az előre meghatározott (determinisztikus) megsemmisítés sem a .NET környezetnek, sem a C# nyelvnek nem része. Nem érünk el jó eredményt, ha ezt a C++ nyelvre jellemző előre meghatározott megsemmisítést próbáljuk ráerőltetni a C# nyelvre. A C# nyelvben a megsemmisítő előbb-utóbb lefut, de futását nem köthetjük egy pontos időponthoz. Az előző példában a kód végül kilép a kritikus részből. A C# nyelvben viszont nem lép ki onnan, amikor a függvény futása befejeződik. Erre csak egy előre ismeretlen, későbbi időpontban kerül sor. Azt hogy mikor, nem tudjuk, és nem is tudhatjuk.

A megsemmisítők használatának teljesítménycsökkentő hatása is van. A megsemmisítést igénylő objektumok megnövelik a szemétgyűjtő teljesítményigényét. Ha a GC azt találja, hogy egy objektum szemét ugyan, de még megsemmisítést igényel, akkor azt az objektumot egyelőre nem törölheti a memóriából; először meg kell hívnia a megsemmisítőt. A megsemmisítők nem ugyanazon a szálon futnak, mint amelyiken a szemétgyűjtés folyik. A GC ehelyett a megsemmisítésre készen álló objektumokat egy várakozási sorba teszi, létrehoz egy másik szálat a megsemmisítők végrehajtásához, majd folytatja a dolgát a többi, memóriában található szemét eltávolításával, és a következő GC ciklusban törli a memóriából a megsemmisített objektumokat. A 2.2. ábrán három különböző GC műveletet láthatunk a memória aktuális állapotával együtt. Figyeljük meg, hogy a megsemmisítést igénylő objektumok több cikluson át megmaradnak a memóriában.

Ebből arra a következtetésre juthatunk, hogy a megsemmisítést igénylő objektumok eggyel több ciklusnyi idővel maradnak tovább a memóriában, mint kellene, ez azonban leegyszerűsítés; a GC egy másik működési elvéből adódóan ennél bonyolultabb a helyzet. A .NET szemétgyűjtője **generációkat** határoz meg munkájának optimalizálásához. A generációk segítségével a GC gyorsabban találja meg azokat az objektumokat, amelyekből a legvalószínűbben lesz szemét. A legutóbbi szemétgyűjtés óta létrehozott objektumok lesznek a 0. generációs objektumok. Az 1. generációs objektumok azokból az objektumokból lesznek, amelyek túlélték egy GC műveletet. A kettő vagy több GC műveletet megélt objektumok a 2. generációs objektumok. A generációk lényege a helyi változók és az alkalmazás egésze során létező objektumok szétválasztása. A 0. generációs objektumok általában helyi változók. A tagváltozókból és a globális változókból gyorsan 1., majd később 2. generációs változók lesznek.

A GC úgy optimalizálja a munkáját, hogy behatárolja, hogy milyen gyakran vizsgálja az első és második generációs objektumokat. A 0. generációs objektumokat minden GC ciklus megvizsgálja. Nagyjából 10-ből 1 GC ciklus vizsgálja meg a 0. és az 1. generációs objektumokat is. Nagyjából 100-ból 1 GC ciklus vizsgálja meg az összes objektumot. Gondoljuk át még egyszer a megsemmisítést és az érte fizetett árat: egy megsemmisítést igénylő objektum akár kilenc GC ciklussal is tovább maradhat a memóriában, mintha nem kellene megsemmisíteni. Ha még ekkor sem történik meg a megsemmisítése, akkor 2. generációs lesz. A 2. generációban az objektumok további 100 GC ciklust élnek meg, mielőtt elérné őket a következő 2. generációs gyűjtés.



2.2. ábra

Ez a sorozat bemutatja a megsemmisítők hatását a szemétgyűjtő tevékenységére. Az objektumok tovább maradnak a memóriában, és a szemétgyűjtő futásához egy új szálat is létre kell hozni.

Zárásul ne feledkezzünk meg róla, hogy a kezelt környezetnek, ahol a szemétgyűjtő felel a memóriakezelésért, számos előnye van. A memóriaszivárgás és sok más mutatókkal kapcsolatos hibalehetőség már nem a mi gondunk. A nem memóriajellegű erőforrások használata esetén megsemmisítőket kell használnunk, hogy gondoskodjunk a felszabadításukról. A megsemmisítők komoly terhelést jelenthetnek a programunk teljesítményére, de meg kell írnunk őket, hogy elkerüljük az elérhető erőforrások számának csökkenését. Az `IDisposable` felület megvalósításával és használatával elkerülhetjük a megsemmisítők által okozott lassulást a szemétgyűjtő működésében. A következő részekben olyan tippeket találunk, amelyek segítenek az olyan programok elkészítésében, amelyek hatékonyabban használják a környezetüket.

12. tipp

Értékadó utasítások helyett használjunk beállító utasításokat

Az osztályoknak gyakran egynél több konstruktoruk van. Idővel a tagváltozók és a konstruktorok összhangja megszűnhet. A legjobb mód arra, hogy ennek elejét vegyük, ha már ott beállítjuk a változók kezdőértékét, ahol bevezetjük őket, ahelyett, hogy ezt minden egyes konstruktor törzsében külön tennénk meg. A beállítás szokásos formáját a statikus és a példányváltozóknál is érdemes használni:

```
public class MyClass
{
  // gyűjtemény bevezetése, kezdőérték beállítása
  private ArrayList _coll = new ArrayList( );
}
```

Innentől kezdve akárhány konstruktort is adjunk a MyClass típushoz, a _coll mindig megfelelő kezdőértékkel fog rendelkezni. A fordító minden konstruktor elejére olyan kódot állít elő, ami a példány összes általunk megadott tagváltozójának beállítását (inicializálását) elvégzi. Ha egy új konstruktort adunk meg, a _coll beállítása akkor is megtörténik. Hasonlóképpen, ha új tagváltozót vezetünk be, nem kell minden konstruktor törzsében elvégeznünk a beállítását, hiszen elég, ha ezt ott tesszük meg, ahol bevezetjük. Ugyanilyen fontos az is, hogy a beállító utasítások a fordító által létrehozott alapértelmezett konstruktorba is bekerülnek. A C# nyelv mindig létrehoz egy alapértelmezett konstruktort, ha mi nem adunk meg egyet sem a típusunkban.

A beállító utasítások nem csak egyszerűen a konstruktorok törzsében lévő utasítások kényelmesebb kiváltását jelentik. A beállító utasítások a tárgykódban a konstruktorok törzse elé kerülnek. A beállító utasítások végrehajtása az alaposztály konstruktorának végrehajtása előtt történik, abban a sorrendben, ahogy a változókat bevezettük a típusban.

A beállító utasítások használata a legegyszerűbb módja annak, hogy elkerüljük a beállítatlan változókat a típusainkban. A módszer azonban nem tökéletes. Van három olyan eset, amikor nem célszerű beállító utasításokat használni. Először is, amikor az objektum kezdőértéke 0 vagy null lesz. A rendszer alapértelmezés szerint mindennek 0 kezdőértéket ad, még azelőtt, hogy a kódunk lefutna. A rendszer által előállított 0 kezdőérték beállítása alacsony szinten történik. A CPU utasításait használva a rendszer egy teljes memóriablokk értékét 0-ra állítja. Ha ezután mi még meg is adjuk a 0 kezdőértékeket, azzal feleslegesen pazaroljuk az energiánkat. A C# fordító persze kötelességtudóan elkészíti a további utasításokat, amelyekkel ismételten lenullázza a memória tartalmát. Ez hibát ugyan nem okoz, de nem is túl hatékony – sőt ha értéktípusokról van szó, akkor nagyon nem az:

```
MyValType _MyVal1;  // kezdőérték beállítva 0 értékre
MyValType _MyVal2 = new MyValType(); // szintén 0 lesz
```

Mindkét utasítás csupa 0 értékre állítja a változót. Az első úgy, hogy a _MyVal1 változót tartalmazó memóriának 0 értéket ad. A második az IL nyelv initobj utasítását használja, ami egy be- és egy kicsomagolást is elvégez a _MyVal2 változón. Ez bizony jó sok időbe telik (lásd a 17. tippet).

A második rossz hatékonyságú megoldás az, amikor ugyanahhoz az objektumhoz többféle kezdőérték-beállítást is készítünk. A beállító utasításokat csak olyan változóknál érdemes használni, amelyeken minden konstruktorban ugyanazt a kezdőbeállítást végeznénk el. A MyClass alábbi változatában van egy olyan eset, amikor a konstruktor végrehajtásakor két külön ArrayList objektumot hozunk létre:

```
public class MyClass
{
  // gyűjtemény bevezetése, kezdőérték beállítása
  private ArrayList _coll = new ArrayList( );
```

```
  MyClass( )
  {
  }

  MyClass( int size )
  {
    _coll = new ArrayList( size );
  }
}
```

Ha a gyűjtemény méretének megadásával hozunk létre egy új MyClass objektumot, akkor egyből két tömblistát hozunk létre, és az egyikből azonnal szemét lesz. A változót beállító utasítás minden konstruktor előtt lefut, a konstruktor törzse pedig létrehozza a második tömblistát is. A fordító ezt a MyClass-változatot hozza létre, amit persze magunk így soha nem írnánk meg. (A 14. tippben láthatjuk, hogyan kell helyesen kezelni az ilyen helyzeteket.)

```
public class MyClass
{
  // gyűjtemény bevezetése, kezdőérték beállítása
  private ArrayList _coll;

  MyClass( )
  {
    _coll = new ArrayList( );
  }

  MyClass( int size )
  {
    _coll = new ArrayList( );
    _coll = new ArrayList( size );
  }
}
```

Az utolsó ok arra, hogy a változók beállítását a konstruktor belsejébe helyezzük, ha elő szeretnénk segíteni a kivételek kezelését. A beállító utasításokat nem tehetjük egy try blokk belsejébe. Ha a tagváltozók létrehozása során bekövetkezne egy kivétel, annak kezelésére mindenképpen az objektumon kívül kerülhet csak sor. Az osztályon belül ez nem lehetséges. Ilyenkor a beállító kódot tegyük a konstruktorok törzsébe, ahol megírhatjuk a helyreigazító kódot a típus létrehozásához és a kivétel elegáns kezeléséhez (45. tipp).

A beállító utasítások használata a legegyszerűbb módja annak, hogy gondoskodjunk arról, hogy a típusok tagváltozóinak beállítása megtörténjen, függetlenül attól, hogy melyik konstruktort hívjuk meg. A beállító utasítások végrehajtása a típus konstruktorainak futása előtt történik. Ezt a módszert követve megelőzhetjük, hogy megfeledkezzünk a megfelelő beállításokról, amikor a programunk egy új változatához létrehozunk egy új konstruktort. Mindig beállító utasításokat használjunk, ha a konstruktorok ugyanúgy hozzák létre a tagváltozókat. Programunk karbantartása és értelmezése egyaránt egyszerűbb lesz.

13. tipp

Az osztály statikus tagjainak beállításához használjunk statikus konstruktorokat

Mint ismeretes, a típusok statikus tagváltozóinak kezdőértéket kell adnunk, mielőtt létrehozhatnánk a típus példányait. A C# nyelvben erre a célra statikus beállító utasításokat és statikus konstruktorokat használhatunk. A statikus konstruktorok olyan különleges függvények, amelyek végrehajtása még azelőtt bekövetkezik, hogy az osztályban bevezetett többi tagfüggvényhez, változóhoz és tulajdonságokhoz hozzáférhetnénk. Ezt a függvényt a statikus változók beállításához, az egyke (singleton) minta alkalmazásához, illetve bármilyen más olyan tevékenységhez használhatjuk, amit el kell végezni ahhoz, hogy az osztályunk használható állapotba kerüljön. A statikus változók beállításához soha ne a példány konstruktorát használjuk, vagy valamilyen különleges privát függvényt, vagy valami más barkácsmegoldást.

A példányok kezdeti beállításához hasonlóan a statikus konstruktorok helyett használhatunk beállító utasításokat is. Ha csak egy statikus tag memóriájának lefoglalása a cél, akkor használjunk egy beállító utasítást. Ha ennél bonyolultabb módon kell kezdőértéket adnunk a tagváltozóknak, készítsünk statikus konstruktort.

A statikus konstruktorokat a C# nyelvben leggyakrabban az egyke minta megvalósításához használják. A példány konstruktora legyen privát, és adjunk hozzá egy beállító utasítást:

```
public class MySingleton
{
  private static readonly MySingleton _theOneAndOnly =
    new MySingleton( );

  public static MySingleton TheOnly
  {
    get
    {
      return _theOneAndOnly;
    }
  }

  private MySingleton( )
  {
  }

  // a többit most elhagyjuk
}
```

Az egyke mintát akkor is éppen ilyen egyszerűen megírhatjuk, ha az egyke beállításához bonyolultabb logikát kell követnünk:

```
public class MySingleton
{
  private static readonly MySingleton _theOneAndOnly;

  static MySingleton( )
  {
    _theOneAndOnly = new MySingleton( );
  }

  public static MySingleton TheOnly
  {
    get
    {
      return _theOneAndOnly;
    }
  }

  private MySingleton( )
  {
  }

  // a többit most elhagyjuk
}
```

A példányok beállító utasításaihoz hasonlóan a statikus beállító utasítások hívása még azelőtt megtörténik, hogy bármely statikus konstruktort meghívnánk. És igen, a statikus beállító utasítások végrehajtása még az alaposztály statikus konstruktorát is megelőzi.

A CLR automatikusan meghívja a statikus konstruktort, amikor a típus először betöltődik az alkalmazástérbe. Csak egy statikus konstruktort adhatunk meg, és az egyetlen paramétert sem kaphat. Mivel a statikus konstruktorokat a CLR hívja meg, vigyáznunk kell a belsejükben előálló kivételekkel. Ha egy kivétel kiszökik a statikus konstruktorból, akkor a CLR leállítja a programot. A kivételek kezelése a leggyakoribb oka annak, hogy a statikus beállító utasítások helyett statikus konstruktorokat használunk. Statikus beállító utasításokkal nem foghatjuk el a kivételeket, statikus konstruktorokkal viszont igen (lásd a 45. tippet):

```
static MySingleton( )
{
  try {
    _theOneAndOnly = new MySingleton( );
  } catch
  {
    // itt megpróbálhatjuk helyrehozni a hibát
  }
}
```

A statikus beállító utasítások és a statikus konstruktorok adják a legtisztább, legvilágosabb módját az osztályok statikus tagjainak beállítására. Könnyen követhetők, és nehezen lehet hibásan megírni őket. Kimondottan azzal a céllal kerültek a nyelvbe, hogy elkerülhessük a más nyelvekben tapasztalható bonyodalmakat a statikus tagok beállításával kapcsolatban.

14. tipp

Használjuk ki a konstruktorláncolás lehetőségét

A konstruktorok megírása sokszor ismétlésekkel jár. Sok fejlesztő megírja az első konstruktort, aztán átmásolja a kódot a többi konstruktorba, eleget téve ezzel az osztály felületében meghatározott felülbírálatok által támasztott követelményeknek. Remélem, az olvasó nem tartozik ezen fejlesztők közé. Ha igen, akkor ideje, hogy felhagyjon ezzel a szokásával. A veterán C++ programozók a közös algoritmusokat egy privát segédfüggvénybe szokták tenni. Ezt is felejtsük el. Ha úgy adódik, hogy több konstruktor szerkezete is megegyezik, akkor inkább hozzunk létre egy olyan konstruktort, ami tartalmazza a közös részt. Ezzel számos előnyhöz jutunk. Elkerülhetjük a kódismétlést, és a konstruktorokból sokkal hatékonyabb tárgykód készül. A C# fordító felismeri a konstruktorokat, és eltávolítja a változók többszöri beállítását, illetve az alaposztály konstruktorának többszöri meghívását. Ennek eredményeképpen a végső objektum minimális mennyiségű kód végrehajtásával végzi el az objektum beállítását. A kód mennyiségét az is csökkenti, hogy a közös kódot csak egyetlen közös konstruktorban írjuk meg.

A konstruktorok kezdeti beállítása során egy konstruktorban meghívhatunk egy másik konstruktort. Az alábbi példa ennek egy egyszerű felhasználását mutatja be:

```
public class MyClass
{
  // adatok gyűteménye
  private ArrayList _coll;
  // A példány neve:
  private string  _name;

  public MyClass() :
    this( 0, "" )
  {
  }

  public MyClass( int initialCount ) :
    this( initialCount, "" )
  {
  }

  public MyClass( int initialCount, string name )
  {
```

```
    _coll = ( initialCount > 0 ) ?
      new ArrayList( initialCount ) :
      new ArrayList();
    _name = name;
  }
}
```

A C# nem támogatja az alapértelmezett paramétereket, ami a C++ nyelvben a legjobb megoldást kínálná erre a problémára. Minden általunk támogatott konstruktort külön függvényként kell megírnunk. Konstruktorokról lévén szó, ez sok kódismétléssel járhat. Egy közös részeket tartalmazó segédeljárás használata helyett használjuk ki a konstruktorok láncolhatóságát. A konstruktorok közös szerkezeti elemeinek alábbi kiszűrése számos teljesítményromboló hatással rendelkezik:

```
public class MyClass
{
  // adatok gyűteménye
  private ArrayList _coll;
  // A példány neve:
  private string  _name;

  public MyClass()
  {
    commonConstructor( 0, "" );
  }

  public MyClass( int initialCount )
  {
    commonConstructor( initialCount, "" );
  }

  public MyClass( int initialCount, string Name )
  {
    commonConstructor( initialCount, Name );
  }

  private void commonConstructor( int count,
    string name )
  {
    _coll = (count > 0 ) ?
      new ArrayList(count) :
      new ArrayList();
    _name = name;
  }
}
```

Ez a változat ugyanolyannak tűnik, de sokkal kevésbé hatékony tárgykód lesz belőle. A fordító sok olyan kódot ad a konstruktorokhoz, amivel helyettünk elvégez bizonyos feladatokat. Például létrehozza a beállító utasításokat a változókhoz (lásd a 12. tippet), és meghívja az alaposztály konstruktorát. Ha saját közös segédeljárást írunk, akkor a fordító nem tudja megszüntetni ennek a kódnak a felesleges megismétlését. A második változathoz tartozó IL olyan lesz, mintha ezt írtuk volna:

```
// Helytelen, csak az előálló IL kód szemléltetésére szolgál:
public MyClass()
{
  private ArrayList _coll;
  private string  _name;

  public MyClass( )
  {
    // Ide jönnének a példány beállító utasításai
    object(); // Helytelen, csak a szemléltetést szolgálja
    commonConstructor( 0, "" );
  }

  public MyClass (int initialCount)
  {
    // Ide jönnének a példány beállító utasításai
    object(); // Helytelen, csak a szemléltetést szolgálja
    commonConstructor( initialCount, "" );
  }

  public MyClass( int initialCount, string Name )
  {
    // Ide jönnének a példány beállító utasításai
    object(); // Helytelen, csak a szemléltetést szolgálja
    commonConstructor( initialCount, Name );
  }

  private void commonConstructor( int count,
    string name )
  {
    _coll = (count > 0 ) ?
      new ArrayList(count) :
      new ArrayList();
    _name = name;
  }
}
```

Ha el tudnánk készíteni a létrehozási kódot, ahogyan azt a fordító látja az első változat esetében, akkor ezt kapnánk:

```
// Helytelen, csak az előálló IL kód szemléltetésére szolgál:
public MyClass()
{
  private ArrayList _coll;
  private string  _name;

  public MyClass( )
  {
    // Itt nincsenek változókat beállító utasítások
    // Meghívjuk az alábbi, harmadik konstruktort
    this( 0, "" ); // Helytelen, csak a szemléltetést szolgálja
  }

  public MyClass (int initialCount)
  {
    // Itt nincsenek változókat beállító utasítások
    // Meghívjuk az alábbi, harmadik konstruktort
    this( initialCount, "" );
  }

  public MyClass( int initialCount, string Name )
  {
    // Ide jönnének a változókat beállító utasítások
    object(); // Helytelen, csak a szemléltetést szolgálja
    _counter = initialCount;
    _name = Name;
  }
}
```

A különbség az, hogy a fordító által előállított kód csak egyszer hívja meg az alaposztály konstruktorát, és nem másolja át a változók beállító utasításait minden konstruktor törzsébe. Annak is jelentősége van, hogy az alaposztály konstruktorának meghívása csak az utolsó konstruktorban következik be. Egy konstruktor meghatározása nem tartalmazhat egynél több konstruktorbeállító utasítást. Vagy átadjuk a feladatot az osztály egy másik konstruktorának a `this()` segítségével, vagy a `base()` függvénnyel meghívjuk az alaposztály konstruktorát. Mindkettő egyszerre nem megy.

Még ezek után sem kezdünk hinni a konstruktorbeállítókban? Akkor gondoljunk a csak olvasható állandókra. Az alábbi példában az objektum nevének nem szabad megváltoznia annak élettartama alatt. Ez azt jelenti, hogy csak olvashatóra kell állítanunk.

A közös segédfüggvény használata ekkor fordítói hibákat eredményez:

```
public class MyClass
{
  // adatok gyűteménye
  private ArrayList _coll;
  // A példány száma:
  private int        _counter;
  // A példány neve:
  private readonly string  _name;

  public MyClass()
  {
    commonConstructor( 0, "" );
  }

  public MyClass( int initialCount )
  {
    commonConstructor( initialCount, "" );
  }

  public MyClass( int initialCount, string Name )
  {
    commonConstructor( initialCount, Name );
  }

  private void commonConstructor( int count,
    string name )
  {
    _coll = (count > 0 ) ?
      new ArrayList(count) :
      new ArrayList();
    // HIBA, a név módosítása a konstruktoron kívül
    _name = name;
  }
}
```

A C++ programozók vagy beletörődnek ebbe, és a _name értékét beállítják minden konstruktorban, vagy az állandót félreteszik a segédfüggvénybe. A C# konstruktorbeállítói jobb megoldást kínálnak erre. A legegyszerűbb eseteket leszámítva minden osztályban több konstruktort találunk. Ezek feladata, hogy beállítsák az objektumok összes tagjának kezdőértékét. Természetüknél fogva ezeknek a függvényeknek hasonló, ideális esetben pedig jórészt azonos a szerkezetük. Használjunk konstruktorbeállítókat, hogy a közös részeket kiszűrhessük, és csak egyszer kelljen megírnunk és végrehajtanunk azokat.

Ez a tipp az utolsó, ami az objektumok kezdőértékeinek beállításával foglalkozott. Ez jó alkalmat ad arra, hogy áttekintsük a típusok példányainak létrehozásával kapcsolatos események teljes sorozatát. Ismernünk kell a műveletek sorrendjét, és az objektumok alapértelmezett beállítási folyamatát is. Arra kell törekednünk, hogy minden tagváltozó beállítását pontosan egyszer végezzük el az objektum létrehozása során. Ezt úgy érhetjük el a legegyszerűbben, ha a kezdőértékeket a lehető leghamarabb beállítjuk. A műveletek sorrendje a típusok első példányának létrehozásánál a következő:

1. A statikus változók tárhelyének feltöltése 0 értékekkel.
2. Statikus változókat beállító utasítások végrehajtása.
3. Az alaposztály statikus konstruktorainak végrehajtása.
4. A statikus konstruktor végrehajtása.
5. A példány változóihoz tartozó tárhely feltöltése 0 értékekkel.
6. A példány változóit beállító utasítások végrehajtása.
7. A megfelelő alaposztály példány konstruktorának végrehajtása.
8. A példány konstruktorának végrehajtása.

A típus további példányainak létrehozása az 5. lépésnél kezdődik, mivel az osztályt beállító utasítások végrehajtása csak egyszer történik meg. A 6. és 7. lépések optimalizáltak, tehát a konstruktorbeállítók hatására a fordító eltávolítja a többször szereplő utasításokat.

A C# nyelv fordítója biztosítja, hogy minden kezdőértéket kap, amikor létrehozunk egy objektumot. Ez legalábbis azt jelenti, hogy egy példány létrehozásakor az objektum által felhasznált összes memóriahely értéke 0 lesz. Ez egyaránt igaz a statikus és a példány tagokra. A mi célunk az, hogy minden a kívánt kezdőértéket kapja, és minden beállító utasítást csak egyszer hajtsunk végre. Az egyszerűbb erőforrásokhoz használjunk beállító utasításokat. A bonyolultabb szerkezetű tagokat konstruktorokkal hozzuk létre, az ismétlések elkerülése végett pedig szűrjük ki a többi konstruktorra irányuló hívásokat.

15. tipp

Az erőforrások felszabadításához használjuk a using és a try/finally kulcsszavakat

Azokat a típusokat, amelyek nem kezelt rendszererőforrásokat használnak, az `IDisposable` felület `Dispose()` tagfüggvényével kell felszabadítanunk. A .NET környezet törvényei szerint ez a típust felhasználó kód feladata, tehát nem a típusé vagy a rendszeré. Ezért ha olyan típusokat használunk, amelyeknek van `Dispose()` tagfüggvénye, akkor a mi feladatunk, hogy a `Dispose()` meghívásával felszabadítsuk ezeket az erőforrásokat. A legjobb módszer arra, hogy bebiztosítsuk a `Dispose()` meghívását, ha kihasználjuk a `using` utasítás vagy a `try/finally` blokkok nyújtotta előnyöket.

Minden olyan típus tartalmazza az `IDisposable` felület megvalósítását, aminek nem kezelt erőforrásai is vannak. Ezen kívül elővigyázatosságból létrehoznak egy megsemmisítőt is, arra az esetre, ha megfeledkeznénk az erőforrások helyes felszabadításáról. Amennyiben valóban elfelejtjük eltávolítani ezeket az elemeket, a nem memóriajellegű erőforrásokat később szabadítja fel a program, amikor a megsemmisítők esélyt kapnak arra, hogy lefussanak. Ezek az objektumok tehát ennyivel tovább maradnak a memóriában, az alkalmazásunkból meg egy lassú erőforrásfaló lesz.

Szerencsénkre a C# nyelv tervezői tudták, hogy az erőforrások felszabadítására gyakran szükség lesz. Hogy leegyszerűsítsék ezt a feladatot, különféle kulcsszavakkal látták el a nyelvet.

Tegyük fel, hogy az alábbi kódot írtuk:

```
public void ExecuteCommand( string connString,
  string commandString )
{
  SqlConnection myConnection = new SqlConnection( connString );
  SqlCommand mySqlCommand = new SqlCommand( commandString,
    myConnection );

  myConnection.Open();
  mySqlCommand.ExecuteNonQuery();
}
```

Ebben a példában két eldobható objektumtól nem szabadultunk meg a megfelelő módon: ezek az `SqlConnection` és az `SqlCommand`. Mindkét objektum a memóriában marad, amíg a program meg nem hívja a megsemmisítőiket. (Mindkét osztály a `System.ComponentModel.Component` osztálytól örökli a megsemmisítőjét.)

A problémát úgy oldhatjuk meg, ha meghívjuk a `Dispose` függvényt, miután végeztünk a paranccsal, és bezártuk a kapcsolatot.

Ez rendben is van, hacsak nem kapunk egy kivételt az SQL-parancs végrehajtása közben. Ebben az esetben a `Dispose()` hívásaink soha nem kerülnek sorra. A `using` utasítás gondoskodik róla, hogy a `Dispose()` meghívása megtörténjen. Elég, ha egy `using` utasításon belül foglalunk az objektumoknak memóriát, és a fordító minden objektum köré előállít egy `try/finally` blokkot:

```
public void ExecuteCommand( string connString,
  string commandString )
{
  using ( SqlConnection myConnection = new
    SqlConnection( connString ))
  {
```

```
      using ( SqlCommand mySqlCommand = new
        SqlCommand( commandString,
        myConnection ))
      {
        myConnection.Open();
        mySqlCommand.ExecuteNonQuery();
      }
    }
  }
```

Ha egy `Disposable` típusú objektumot használunk egy függvényen belül, akkor mindig a `using` utasítás használata a legegyszerűbb módszer arra, hogy gondoskodjunk az objektum által elfoglalt memória megfelelő felszabadításáról. A `using` utasítás hatására a program előállít egy `try/finally` blokkot a memóriát lefoglaló objektum köré. Az alábbi két kódrészletből pontosan ugyanaz az IL kód áll elő:

```
SqlConnection myConnection = null;

// Példa a using utasítással:
using ( myConnection = new SqlConnection( connString ))
{
  myConnection.Open();
}

// Példa a try/finally blokkal:
try {
  myConnection = new SqlConnection( connString );
  myConnection.Open();
}
finally {
  myConnection.Dispose( );
}
```

Ha a `using` utasítást egy olyan típusú változóval használjuk, amelyik nem támogatja az `IDisposable` felületet, akkor a C# fordító hibát jelez:

```
// Nem lehet lefordítani:
// a karakterlánc zárt, és nem támogatja az IDisposable felületet
using( string msg = "This is a message" )
  Console.WriteLine( msg );
```

A `using` utasítás csak akkor működik, ha a fordítási idejű típus támogatja az `IDisposable` felületet. Akármilyen típusú objektumokkal nem használhatjuk:

```
// Nem lehet lefordítani:
// az objektum nem támogatja az IDisposable felületet
using ( object obj = Factory.CreateResource( ))
  Console.WriteLine( obj.ToString( ));
```

Mindössze egy óvatos és gyors `as` utasításra van szükségünk ahhoz, hogy biztonságosan megszabaduljunk azoktól az objektumoktól, amelyek vagy tartalmazzák az `IDisposable` felület megvalósítását, vagy nem:

```
// A helyes megoldás
// Az objektum vagy támogatja az IDisposable felületet, vagy nem
object obj = Factory.CreateResource( );
using ( obj as IDisposable )
  Console.WriteLine( obj.ToString( ));
```

Ha az `obj` tartalmazza az `IDisposable` megvalósítását, akkor a `using` utasítás előállítja a felszabadító kódot. Ha nem, akkor a `using` utasítás visszafejlődik egy `using(null)` utasítássá, ami biztonságos, de nem csinál semmit. Ha nem vagyunk biztosak benne, hogy be kellene csomagolnunk egy `using` blokkba egy objektumot, akkor a biztonságot tartsuk szem előtt. Vegyük úgy, hogy így kell tennünk, és csomagoljuk be egy `using` blokkba a korábban bemutatott módon.

Ezzel az egyszerű eseten túl is vagyunk. Ha egy tagfüggvényen belül egy darab eldobható (disposable) helyi objektumot használunk, akkor csomagoljuk be ezt az objektumot egy `using` blokkba. Az első példában két különböző objektumtól kell megszabadulnunk: a kapcsolattól és a parancstól. A bemutatott példa két különböző `using` utasítást használ, mindegyik megsemmisítendő objektumhoz egyet-egyet. A két `using` utasítás más `try/finally` blokkot állít elő. Tulajdonképpen az alábbi szerkezetet írtuk meg:

```
public void ExecuteCommand( string connString,
  string commandString )
{
  SqlConnection myConnection = null;
  SqlCommand mySqlCommand = null;
  try
  {
    myConnection = new SqlConnection( connString );
    try
    {
      mySqlCommand = new SqlCommand( commandString,
      myConnection );
```

```
      myConnection.Open();
      mySqlCommand.ExecuteNonQuery();
    }
    finally
    {
      if ( mySqlCommand != null )
        mySqlCommand.Dispose( );
    }
  }
  finally
  {
    if ( myConnection != null )
      myConnection.Dispose( );
  }
}
```

Mindegyik `using` utasítás egy új beágyazott `try/finally` blokkot hoz létre. Nekem nem túlzottan tetszik ez a szerkezet, ezért ha több olyan objektum számára is foglalok helyet, amelyek tartalmazzák az `IDisposable` megvalósítását, akkor jobban szeretem saját magam elkészíteni a `try/finally` blokkokat:

```
public void ExecuteCommand( string connString,
  string commandString )
{
  SqlConnection myConnection = null;
  SqlCommand mySqlCommand = null;
  try {
    myConnection = new SqlConnection( connString );
    mySqlCommand = new SqlCommand( commandString,
      myConnection );

    myConnection.Open();
    mySqlCommand.ExecuteNonQuery();
  }
  finally
  {
    if ( mySqlCommand != null )
      mySqlCommand.Dispose();
    if ( myConnection != null )
      myConnection.Dispose();
  }
}
```

Azért ne próbáljuk túlszépíteni a dolgot azzal, hogy megpróbáljuk ezt a `using` utasítással és `as` utasításokkal megírni:

```
public void ExecuteCommand( string connString,
  string commandString )
{
  // Rossz ötlet. Erőforrás-szivárgás fenyeget!
  SqlConnection myConnection =
    new SqlConnection( connString );
  SqlCommand mySqlCommand = new SqlCommand( commandString,
      myConnection );
      using ( myConnection as IDisposable )
      using (mySqlCommand as IDisposable )
      {
        myConnection.Open();
        mySqlCommand.ExecuteNonQuery();
      }

}
```

Ez tisztábbnak tűnik, de van benne egy apró hiba. Az `SqlConnection` objektumot soha nem szabadítjuk fel, ha az `SqlCommand()` konstruktor kivált egy kivételt. Mindig gondoskodnunk kell róla, hogy az `IDisposable` felületet megvalósító objektumok lefoglalása egy `using` vagy egy `try` blokk belsejében történjen, különben szépen elcsordogálhatnak az erőforrások.

Ezidáig a két legegyszerűbb eset kezelésével foglalkoztunk. Ha egy tagfüggvényben foglalunk memóriát egy darab eldobható objektumnak, akkor a `using` utasítás használatával biztosíthatjuk a legegyszerűbben, hogy a lefoglalt erőforrások felszabadítására minden osztálynál sor kerüljön. Ha egyazon tagfüggvényben több objektumnak is foglalunk memóriát, akkor több `using` blokkot kell létrehoznunk, vagy meg kell írnunk a saját `try/finally` blokkunkat.

Van még egy apróság, amit az eldobható objektumok felszabadításáról tudnunk kell. Vannak olyan típusok, amelyek az erőforrások felszabadításához a `Dispose` és a `Close` tagfüggvények használatát is támogatják. Az `SqlConnection` éppen ilyen típus.
Az `SqlConnection` bezárását így kell elvégeznünk:

```
public void ExecuteCommand( string connString,
  string commandString )
{
  SqlConnection myConnection = null;
  try {
    myConnection = new SqlConnection( connString );
    SqlCommand mySqlCommand = new SqlCommand( commandString,
      myConnection );
```

```
            myConnection.Open();
            mySqlCommand.ExecuteNonQuery();
        }
        finally
        {
            if ( myConnection != null )
                myConnection.Close();
        }
    }
```

Ez a változat bezárja ugyan a kapcsolatot, de ez nem ugyanaz, mintha felszabadítottuk volna a helyét. A `Dispose` tagfüggvény túlmegy azon, hogy felszabadítja az erőforrásokat. A szemétgyűjtőt is értesíti, hogy már nincs szükség az objektum megsemmisítésére. A `Dispose` meghívja a `GC.SuppressFinalize()` függvényt. A `Close` ezt általában nem teszi meg. Ennek eredményeképp az objektum bent marad a megsemmisítésre várakozó objektumok sorában, bár már nincs szükség a megsemmisítésére. Ha van választási lehetőségünk, akkor a `Dispose()` mindig jobb választás, mint a `Close()`. A 18. tippnél minden további részletre fény derül.

A `Dispose()` nem távolítja el az objektumokat a memóriából. Csak egy horog, aminek a segítségével az objektumok elereszthetik a nem kezelt erőforrásokat. Ez azt jelenti, hogy bajba juthatunk, ha olyan objektumtól akarunk megszabadulni, ami még használatban van. Soha ne dobjunk ki olyan objektumokat, amelyekre a programunk egy másik részében még hivatkozunk.

Bizonyos szempontból az erőforrások kezelése a C# nyelvben nehezebb lehet, mint a C++-ban. Nem támaszkodhatunk az előre meghatározott megsemmisítésre, ami felszabadítja az összes általunk használt objektumot, de a szemétgyűjtővel ellátott környezet valójában nagyon leegyszerűsíti az életünket. A ténylegesen használt típusok döntő többsége nem tartalmazza az `IDisposable` megvalósítását. A .NET keretrendszer 1500 típusa közül kevesebb, mint 100 osztályban találjuk meg az `IDisposable` megvalósítását. Ha viszont azokat használjuk, amelyekben ott van az `IDisposable` felület, akkor minden esetben szabadítsuk fel a memóriát. Az objektumokat `using` vagy `try/finally` blokkokkal vegyük körül. Bármelyik módszert is használjuk, ezzel mindig gondoskodunk róla, hogy az objektumok felszabadítása mindig és mindenkor a megfelelő módon történjék.

16. tipp

Ne szemeteljünk!

A szemétgyűjtő kitűnően elvégzi helyettünk a memóriakezeléssel kapcsolatos feladatokat, hatékonyan eltávolítva a memóriából a nem használt objektumokat. De bárhonnan is nézzük a dolgot, egy objektum memóriahelyének lefoglalása és felszabadítása a halmon több időbe telik, mintha a területet nem kellene lefoglalnunk és felszabadítanunk. Komoly érvágást jelenthet a programunk teljesítményére nézve, ha túl sok helyi hivatkozási objektumot hozunk létre a tagfüggvényeinken belül.

Ezért ha lehet, ne terheljük le túlzottan a szemétgyűjtőt. Néhány egyszerű eljárást követve minimálisra csökkenthetjük azt a munkát, amit a szemétgyűjtőnek kell elvégeznie a programunk helyett. Minden hivatkozási típus, még a helyi változók is, a halomra kerülnek. Minden hivatkozási típusú helyi változóból abban a pillanatban szemét lesz, amint kilépünk az adott függvényből. Nagyon gyakori hiba például, amikor GDI objektumoknak foglalunk memóriát egy windowsos ablakrajzoló eseménykezelőben:

```
protected override void OnPaint( PaintEventArgs e )
{
  // Hiba. Minden újrarajzolásnál létrehozzuk ugyanazt a betűtípust.
  using ( Font MyFont = new Font( "Arial", 10.0f ))
  {
    e.Graphics.DrawString( DateTime.Now.ToString(),
      MyFont, Brushes.Black, new PointF( 0,0 ));
  }
  base.OnPaint( e );
}
```

Az `OnPaint()` függvényt sokszor meghívja a program. Ilyenkor minden alkalommal egy új `Font` objektumot hozunk létre, mindig ugyanazokkal a beállításokkal, a szemétgyűjtőnek pedig minden alkalommal el kell takarítania ezeket. Ez rendkívül rossz hatékonyságú megoldás. Ehelyett csináljunk a helyi `Font` objektumból egy tagváltozót, és mindig használjuk azt, amikor újra kell rajzolnunk az ablakot:

```
private readonly Font _myFont =
  new Font( "Arial", 10.0f );

protected override void OnPaint( PaintEventArgs e )
{
  e.Graphics.DrawString( DateTime.Now.ToString( ),
    _myFont, Brushes.Black, new PointF( 0,0 ));
  base.OnPaint( e );
}
```

A programunk így már nem termel szemetet minden újrarajzolási eseménynél, tehát a szemétgyűjtőnek kevesebbet kell dolgoznia, a programunk pedig kissé gyorsabban fog futni. Ha tagváltozóvá léptetünk elő egy olyan helyi változót, például egy betűtípus-objektumot, ami tartalmazza az `IDisposable` megvalósítását, akkor az osztályban el kell készítenünk az `IDisposable` megvalósítását. A 18. tippben pontosan arról lesz szó, hogy ezt hogyan tehetjük meg helyesen.

A helyi változókból akkor célszerű tagváltozókat csinálni, ha azok hivatkozási típusúak (az értéktípusoknál ez nem érdekes), és olyan eljárásokban használjuk őket, amelyeket gyakran hív meg a program. Az újrarajzoló eljárás betűtípus-objektuma nagyszerű példa erre. A fent leírtak azonban csak a gyakran meghívott eljárások helyi változóira igazak, a ritkán meghívott eljárásokra nem. A cél az, hogy ne kelljen újra és újra létrehoznunk ugyanazokat az objektumokat, és nem az, hogy minden helyi változóból tagváltozót csináljunk.

A korábban használt `Brushes.Black` statikus tulajdonság egy másik olyan megoldás alkalmazását szemlélteti, amellyel elkerülhetjük az egymáshoz hasonló objektumok memóriahelyének ismételt lefoglalását. A szükséges hivatkozási típusok gyakran használt példányaihoz készítsünk statikus tagváltozókat. Példaként vegyük a korábban használt fekete ecsetet. Minden alkalommal, ha valamit feketével akarunk berajzolni az ablakunkba, szükségünk lesz egy fekete ecsetre. Ha mindig új ecsetet hozunk létre, amikor valamit rajzolunk, akkor óriási mennyiségű helyet foglalunk le és szabadítunk fel a program futása közben. Az első megközelítés – ha minden típusunkhoz készítünk egy fekete ecsetet – ugyan működik, de nem ez a leghatékonyabb megoldás. A programok tucatnyi ablakot és vezérlőelemet hozhatnak létre, ami tucatnyi fekete ecset létrehozásához vezethet. A .NET keretrendszer tervezői számítottak erre, és ezért létrehoztak egyetlen fekete ecsetet, amit újra és újra felhasználhatunk, ha szükségünk van rá. A `Brushes` osztály számos statikus `Brush` (Ecset) objektumot tartalmaz, amelyek egy-egy gyakori színt képviselnek. A `Brushes` osztály belsőleg egy lusta kiértékelő algoritmust használ, ami csak azokat az ecseteket hozza létre, amelyekre igényt tartunk. Egyszerűsített megvalósítása így néz ki:

```
private static Brush _blackBrush;
public static Brush Black
{
  get
  {
    if ( _blackBrush == null )
      _blackBrush = new SolidBrush( Color.Black );
      return _blackBrush;
  }
}
```

Amikor első alkalommal kérünk egy fekete ecsetet, a `Brushes` osztály létrehozza azt. Az osztály az egyetlen fekete ecsetre való hivatkozást adja vissza, amikor ismét kérjük tőle. Végeredményben ez azt jelenti, hogy létrehozunk egy fekete ecsetet, és azt használjuk

az idők végezetéig. Ráadásul ha az alkalmazásunknak egy bizonyos erőforrásra, mondjuk egy világos citromzöld ecsetre nincs szüksége, akkor azt nem hozza létre. A keretrendszer módot ad arra, hogy éppen csak annyi objektumot kelljen létrehoznunk, amennyire tényleg szükségünk van a céljaink eléréséhez. Kövessük mi is ezt a módszert a programjainkban.

Két megoldást is megismertünk arra, hogy miként csökkenthetjük minimálisra a memóriafoglalást, miközben a programunk teszi a dolgát. A gyakran használt helyi változókat tagváltozókká léptethetjük elő, illetve létrehozhatunk egy osztályt, ami egyke objektumokat tárol, amelyek egy adott típus gyakran használt példányait testesítik meg. Az utolsó módszer a nem változó típusok végső értékének felépítésével kapcsolatos. A `System.String` osztály nem változó típusú. Miután létrehozunk egy karakterláncot, annak tartalmát már nem módosíthatjuk. Ha olyan kódot írunk, ami látszólag módosítja a karakterlánc tartalmát, akkor valójában egy új karakterlánc objektumot hozunk létre, a régiből pedig szemét lesz. Az alábbi látszólag ártalmatlan gyakorlat:

```
string msg = "Hello, ";
msg += thisUser.Name;
msg += ". Today is ";
msg += System.DateTime.Now.ToString();
```

De pont annyira nem hatásos, mintha ezt írtuk volna:

```
string msg = "Hello, ";
// Helytelen, csak a szemléltetést szolgálja:
string tmp1 = new String( msg + thisUser.Name );
string msg = tmp1; // A "Hello " szemét lett
string tmp2 = new String( msg + ". Today is " );
msg = tmp2; // A "Hello <user>" szemét lett
string tmp3 = new String( msg + DateTime.Now.ToString( ) );
msg = tmp3; // A "Hello <user>. Today is " szemét lett
```

A `tmp1`, `tmp2`, `tmp3` és az eredetileg létrehozott `msg ("Hello")` karakterláncokból mind szemét lesz. A karakterlánc osztályon alkalmazott += tagfüggvény létrehoz egy új karakterlánc objektumot, és azt a karakterláncot adja vissza. Nem az eredeti karakterláncot módosítja úgy, hogy hozzáfűzi az új karaktereket. Az előbbihez hasonló egyszerű szerkezetekhez célszerű a `string.Format()` tagfüggvényt használni:

```
string msg = string.Format ( "Hello, {0}. Today is {1}",
  thisUser.Name, DateTime.Now.ToString( ));
```

A bonyolultabb műveleteket a `StringBuilder` osztály segítségével végezhetjük el:

```
StringBuilder msg = new StringBuilder( "Hello, " );
msg.Append( thisUser.Name );
msg.Append( ". Today is " );
msg.Append( DateTime.Now.ToString());
string finalMsg = msg.ToString();
```

A `StringBuilder` egy változó karakterlánc osztály, amivel nem változó karakterlánc objektumokat építhetünk fel. Olyan szolgáltatásokat kínál a változó karakterláncokhoz, amelyekkel módosíthatjuk a szöveges adatokat, mielőtt létrehoznánk egy nem változó karakterlánc objektumot. A karakterlánc objektumok végső változatának létrehozásához használjuk a `StringBuilder` osztályt. Ami ennél is fontosabb, lessük el a programtervezési fogást. Ha a programunkban nem változó típusokra van szükség (lásd a 7. tippet), akkor mindig fontoljuk meg egy építő objektum használatát, ami elősegíti a végső objektum létrehozását. Ez lehetőséget ad az osztályunk felhasználóinak arra, hogy lépésenként építsék fel az objektumaikat, és mellette megőrizzék a típus nem változó mivoltát.

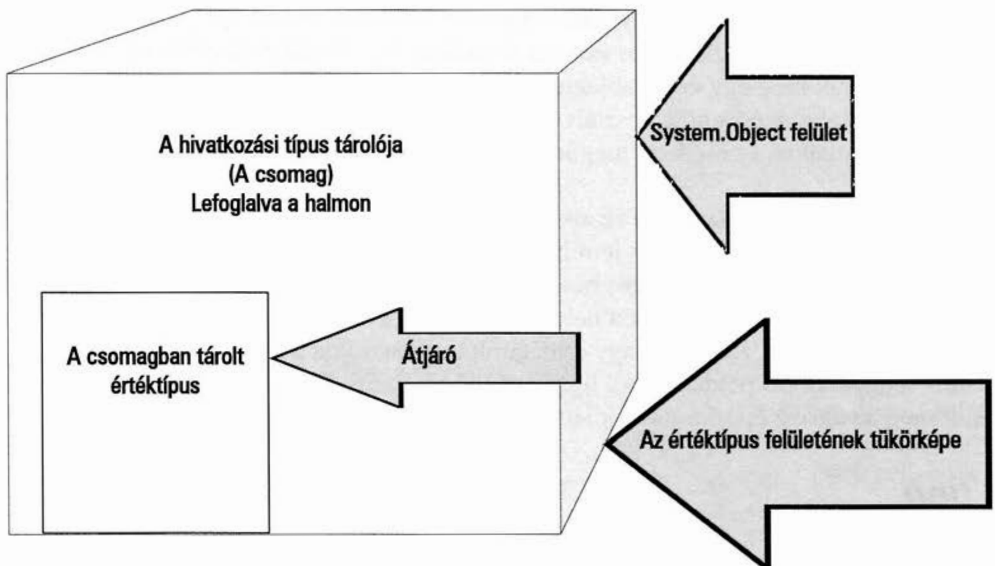
A szemétgyűjtő kitűnően elvégzi az alkalmazásunk által használt memória kezelését. De ne feledjük, hogy az objektumok létrehozása és megsemmisítése a halmon mégiscsak időt vesz igénybe. Kerüljük a felesleges objektumok létrehozását. Amire nincs szükségünk, azt ne hozzuk létre. Szintén kerüljük a helyi függvényekben a hivatkozási típusú objektumok tömeges létrehozását. Helyette vagy csináljunk tagváltozókat a helyi változókból, vagy a típusunk leggyakoribb példányaiból hozzunk létre statikus objektumokat. Végezetül, fontoljuk meg a változó építőosztályok létrehozását a nem változó típusokhoz.

17. tipp

Ne csomagoljunk feleslegesen!

Az értéktípusok adatokat tárolnak, és nem többalakúak. Másrészt viszont a .NET keretrendszer csak egyetlen hivatkozási típust tartalmaz: a `System.Object` osztályt, ami a teljes objektumhierarchia gyökerénél helyezkedik el. A két megállapítás között ellentmondás van. A .NET keretrendszer **becsomagolást** és **kicsomagolást** használ a köztük lévő szakadék áthidalására. A becsomagolás során az értéktípus egy típus nélküli hivatkozási objektumba kerül, hogy olyan helyen is használhassuk, ahol egy hivatkozási típust vár a program. A kicsomagolás során a program kimásolja a csomagból az értéktípust. A be- és kicsomagolásra azért van szükségünk, hogy ott is használhassunk értéktípusokat, ahol `System.Object` típust vár a program. A be- és kicsomagolás azonban mindig lelassítja a programot, néha pedig, amikor a be- és kicsomagolás során átmeneti másolat készül az objektumokról, a folyamat alig észrevehető hibákhoz vezethet a programokban. Ezért ha lehet, kerüljük a be- és kicsomagolást.

A becsomagolás során az értéktípusból hivatkozási típus lesz. Egy új hivatkozási objektum (a csomag) kerül a halomra, és a hivatkozási objektumba bekerül az értéktípus másolata. A 2.3. ábrán láthatjuk annak szemléltetését, hogy miként tárolja a program a becsomagolt objektumot, és hogy miként érhetjük el azt. A csomag tartalmazza az értéktípus objektum másolatát, és megkettőzi a becsomagolt értéktípusban megvalósított felületeket. Ha ki kell olvasnunk valamit a csomagból, akkor a program lemásolja az értéktípust, és a másolatot adja vissza. Ez a lényege a be- és kicsomagolásnak: az objektum másolata kerül a csomagba, és egy másik másolatot kapunk vissza, ha hozzányúlunk a csomag tartalmához.



2.3. ábra

Becsomagolt értéktípus. Egy értéktípus System.Object típusra alakításakor egy név nélküli hivatkozási típus jön létre. Az értéktípus beszúrva tárolódik a név nélküli hivatkozási típusban. Az értéktípushoz hozzáférni kívánó tagfüggvények a csomagon keresztül haladva érhetik el a tárolt értéktípust.

A legalattomosabb probléma a be- és kicsomagolással, hogy az egész automatikusan zajlik. A fordító mindig előállítja a be- és kicsomagoló utasításokat, amikor olyan helyen használunk egy értéktípust, ahol a program hivatkozási típust várna, mint amilyen a `System.Object`. A be- és kicsomagolási műveletek akkor is végrehajtódnak, ha egy értéktípust egy felületmutatón keresztül használunk.

Figyelmeztetés nincs, a csomagolás viszont bekövetkezik. Még az alábbi egyszerű utasítás is csomagoláshoz vezet:

```
Console.WriteLine("A few numbers:{0}, {1}, {2}",
  25, 32, 50);
```

A hivatkozott `Console.WriteLine` túlterhelt tagfüggvény `System.Object` objektumokra mutató hivatkozásokból álló tömböt vár paraméterként. Az egészek értéktípusok, amiket be kell csomagolni, hogy a program átadhassa őket a `WriteLine` túlterhelt tagfüggvényének. Az egyetlen módja annak, hogy `System.Object` alakúra formáljuk a három egész paramétert, ha becsomagoljuk őket. Ráadásul a `WriteLine` belsejében a kód benyúl a csomagba, hogy meghívja a csomagban lévő objektum `ToString()` tagfüggvényét. Bizonyos értelemben az alábbi szerkezetet állítottuk elő:

```
int i =25;
object o = i; // becsomagolás
Console.WriteLine(o.ToString());
```

A `WriteLine` belsejében az alábbi kód végrehajtása történik:

```
object o;
int i = ( int )o; // kicsomagolás
string output = i.ToString( );
```

Ilyen kódot magunktól soha nem írnánk. De azzal, hogy ráhagytuk a fordítóra, hogy automatikusan átalakítson egy adott értéktípust `System.Object` típusúra, hagytuk, hogy megtörténjen. A fordító csak segíteni akart. Azt szeretné, hogy örüljünk. Vidáman előállítja azokat az utasításokat, amelyek bármilyen értéktípusból előállítják a `System.Object` osztály egy példányát. Hogy elkerüljük ezt a csapást, mindig alakítsuk át a típusainkat karakterlánccá, mielőtt átadjuk azokat a `WriteLine` függvénynek:

```
Console.WriteLine("A few numbers:{0}, {1}, {2}",
 25.ToString(), 32.ToString(), 50.ToString());
```

Ez a kód az ismert egész típust használja, és a program az értéktípusokat (egészeket) nem alakítja át a háttérben `System.Object` típusúra. Ez a gyakori példa szemlélteti a csomagolás elkerülésének első számú szabályát: mindig figyeljünk oda, nehogy a program rejtve átalakítsa `System.Object` típusúra a változóinkat. Ha tehetjük, akkor értéktípusokat soha ne tegyünk a `System.Object` helyére.

Egy másik gyakori eset, amikor akaratlanul is egy értéktípus kerülhet egy `System.Object` helyére, ha egy értéktípust teszünk egy .NET 1.x gyűjteménybe. A .NET keretrendszeri gyűjteményeknek ez az alfaja `System.Object` példányokra mutató hivatkozásokat tárol. Ha egy értéktípust adunk a gyűjteményhez, az bekerül egy csomagba. Ha eltávolítunk egy objektumot a gyűjteményből, akkor azt a csomagból másolja ki a program. Ha kiveszünk egy objektumot a csomagból, az mindig az objektum lemásolásával jár. Ez alig észrevehető hibákat eredményezhet az alkalmazásunkban. A fordító nem képes segíteni ezeknek a hibáknak a felkutatásában, és mindez a csomagolás miatt van. Induljunk ki egy egyszerű szerkezetből, ami megengedi, hogy módosítsuk az egyik mezőjét, majd tegyünk néhány ilyen objektumot egy gyűjteménybe:

```
public struct Person
{
  private string _Name;

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

  public override string ToString( )
  {
    Return _Name;
  }
}

// A Person struktúra használata egy gyűjteményben:
ArrayList attendees = new ArrayList( );
Person p = new Person( "Old Name" );
attendees.Add( p );

// Megpróbáljuk módosítani a nevet:
// Működne, ha a Person hivatkozási típus lenne
Person p2 = (( Person )attendees[ 0 ] );
p2.Name = "New Name";

// A régi nevet ("Old Name") írja ki:
Console.WriteLine(
  attendees[ 0 ].ToString( ));
```

A Person értéktípus. A program becsomagolja, mielőtt beletenné az ArrayList gyűjteménybe, ami egy másolat létrehozását jelenti. Ezután egy másik másolat is készül, amikor eltávolítjuk a Person objektumot, hogy hozzáférjünk a módosításra váró Name tulajdonsághoz. Mindössze annyit értünk el, hogy módosítottuk a másolatot. Sőt egy harmadik másolat is készült ahhoz, hogy a program az attendees[0] objektumon keresztül meghívhassa a ToString() függvényt.

Ezért és még sok más miatt is célszerű nem változó értéktípusokat készítenünk (lásd a 7. tippet). Ha feltétlenül változó értéktípusra van szükségünk egy gyűjteményben, akkor használjuk a System.Array osztályt, ami típusbiztos.

Ha egy tömb nem a megfelelő gyűjtemény, akkor ezt a hibát a C# 1.x változataiban felületek használatával orvosolhatjuk. Ha a felületekhez készítjük a kódot és nem a típus nyilvános tagfüggvényeihez, akkor benyúlhatunk a csomag belsejébe, hogy módosítsuk az értékeket:

```
public interface IPersonName
{
  string Name
  {
    get; set;
  }
}

struct Person : IPersonName
{
  private string _Name;

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

  public override string ToString( )
  {
    return _Name;
  }
}
```

```
// A Person struktúra használata egy gyűjteményben:
ArrayList attendees = new ArrayList( );
Person p = new Person( "Old Name" );
attendees.Add( p ); // box

// Megpróbáljuk módosítani a nevet:
// A felületet használjuk, nem a típust
// Nincs szükség a kicsomagolásra
(( IPersonName )attendees[ 0 ] ).Name = "New Name";

// Az új nevet ("New Name") írja ki:
Console.WriteLine(
  attendees[ 0 ].ToString( )); // kicsomagolás
```

A csomag hivatkozási típusa tartalmazza az összes olyan felület megvalósítását, ami az eredeti objektumban is megtalálható. Ez azt jelenti, hogy nem készül másolat, hanem a csomagra meghívjuk az `IPersonName.Name` tagfüggvényt, ami továbbítja a kérést a csomagban lévő értéktípushoz. Az értéktípusokban létrehozott felületek lehetővé teszik, hogy belenyúljunk a csomagba, hogy módosíthassuk a gyűjteményben tárolt értéket. A felület megvalósítása nem ugyanaz, mintha többalakúan kezelnénk az értéktípust, ami ismét a csomagolással járó hátrányokhoz vezet (lásd a 20. tippet).

Ezek a korlátok a C# 2.0 megjelenésével érkező általános elemek (generics) hatására módosulnak. Az általános felületek és az általános gyűjtemények a gyűjteményekkel és a felületekkel kapcsolatos kérdésekre is választ adnak. Addig viszont kerüljük a csomagolást. Bizony az értéktípusok átalakulhatnak egy `System.Object` vagy valamilyen felületre mutató hivatkozássá. Ez az átalakítás rejtett módon zajlik le, ami megnehezíti a felkutatását. Ez része a nyelv és a környezet által diktált játékszabályoknak. A be- és kicsomagolási műveletek közben olyankor jöhetnek létre másolatok, amikor arra a legkevésbé számítanánk, ami hibákhoz vezet. Az értéktípusok többalakú kezeléséért teljesítménycsökkenéssel is fizetünk. Figyeljünk oda azokra a szerkezetekre, amelyek az értéktípusokat `System.Object` vagy felület típusra alakítják: értékek gyűjteménybe helyezésekor, a `System.Object` tagfüggvényeinek hívásakor, és a `System.Object` típusra alakításkor. Ha tehetjük, messze kerüljük el ezeket a helyzeteket.

18. tipp

Használjuk a szabványos felszabadító minta megvalósítását

A könyvben már szót ejtettünk a kezeletlen erőforrásokat tartalmazó objektumok megsemmisítésének fontosságáról. Eljött az idő, hogy bemutassuk, hogyan kell megírnunk a saját erőforrás-kezelő kódunkat, amikor olyan típusokat hozunk létre, amelyek a memórián kívül más erőforrásokat is használnak. A .NET keretrendszerben mindenhol egy szabványos mintát használnak a nem memóriajellegű erőforrások felszabadításához. A típusaink felhasználói arra számítanak, hogy mi is ezt a szabványos mintát követjük. A szabványos felszabadító eljárás az `IDisposable` felületet használva szabadítja fel a kezeletlen erőforrásokat, ha az ügyfélkód nem feledkezik meg a dolgáról, védekezésképpen pedig a megsemmisítőt használva teszi ezt meg, ha az ügyfélkód nem végzi el a szükséges teendőket. Mindez a szemétgyűjtő segítségével történik, hogy az objektumok megsemmisítésével járó teljesítményveszteség csak szükség esetén következzen be. Ez a helyes módja a kezeletlen erőforrások használatának, ezért érdemes tökéletesen tisztában lenni vele.

Az osztályhierarchia gyökerénél lévő alaposztálynak tartalmaznia kell az `IDisposable` felület megvalósítását az erőforrások felszabadításához. Ehhez a típushoz védekezésképpen tartoznia kell egy megsemmisítőnek is. Mindkét eljárás egy virtuális tagfüggvénynek adja át az erőforrások felszabadításával járó munka elvégzését, így a származtatott osztályok felülbírálhatják ezt a tagfüggvényt a saját erőforrás-kezelési szükségleteiknek megfelelően. A származtatott osztályoknak csak akkor kell felülbírálniuk a virtuális tagfüggvényt, ha saját erőforrásaikat kell felszabadítaniuk, de ilyenkor sem szabad megfeledkezni a függvény alaposztályhoz tartozó változatának meghívásáról.

Ha az osztályunk nem memóriajellegű erőforrásokat használ, akkor először is kell, hogy legyen egy megsemmisítő eljárása. Nem szabad arra számítanunk, hogy az ügyfelek majd biztosan meghívják a `Dispose()` tagfüggvényt. Ha elfelejtik, akkor máris megindul az erőforrások szivárgása. Az ugyan nem a mi hibánk, ha az ügyfél elfelejti meghívni a `Dispose` függvényt, de a bűnbak szerepe a miénk lesz. A megsemmisítő elkészítése az egyetlen módja annak, hogy a nem memóriajellegű erőforrások felszabadítását biztosítsuk. Készítsük el tehát a megsemmisítőt.

Amikor a szemétgyűjtő működésbe lép, azonnal eltávolítja a memóriából azokat a szemétté vált objektumokat, amelyeknek nincs megsemmisítőjük. Azok az objektumok, amelyeknek van megsemmisítőjük, a memóriában maradnak. Ezek az objektumok bekerülnek egy megsemmisítési sorba, a Garbage Collector pedig elindít egy új szálat, ami lefuttatja az objektumokhoz tartozó megsemmisítőket. Miután a megsemmisítő szál végzett a munkájával, a szemétté vált objektumokat már el lehet távolítani a memóriából. Azok az objektumok, amelyek megsemmisítésre szorulnak, sokkal tovább maradnak a memóriában, mint a megsemmisítő nélküli objektumok. De nincs más választásunk. Ha biztosra akarunk menni,

akkor mindig meg kell írnunk a megsemmisítőt, ha az objektum kezeletlen erőforrásokat használ. A teljesítménnyel egyelőre ne foglalkozzunk. A következő lépések gondoskodnak róla, hogy az ügyfeleknek egyszerűbb legyen elkerülni a megsemmisítéssel járó teljesítményveszteséget.

Az `IDisposable` felület megvalósítása a szabványos módja annak, hogy tudassuk a felhasználókkal és a futtató környezettel, hogy az objektum olyan erőforrásokat tartalmaz, amelyeket a megfelelő időben fel kell szabadítani. Az `IDisposable` felület mindössze egyetlen tagfüggvényt tartalmaz:

```
public interface IDisposable
{
  void Dispose( );
}
```

Az `IDisposable.Dispose()` megvalósítása az alábbi négy feladat elvégzéséért felelős:

1. Az összes kezeletlen erőforrás felszabadítása.
2. Az összes kezelt erőforrás felszabadítása (beleértve az események eleresztését is).
3. Egy jelzőérték beállítása, ami azt jelzi, hogy az objektum felszabadítása megtörtént. Ezt az állapotjelzést ellenőriznünk kell a nyilvános tagfüggvényekben, és `ObjectDisposed` kivételt kell kiváltanunk, ha az objektum helyének felszabadítása után következett be a meghívásuk.
4. A megsemmisítés letiltása. Ezt a `GC.SuppressFinalize(this)` tagfüggvény hívásával érjük el.

Az `IDisposable` megvalósításával két dolgot érünk el. Egyrészt egy olyan rendszert bocsátunk az ügyfelek rendelkezésére, amellyel a megfelelő időben felszabadíthatják az összes kezelt erőforrást, másrészt szabványos módszert adunk az ügyfelek kezébe az összes kezeletlen erőforrás felszabadításához. Ez bizony komoly fejlődés. Miután elkészítettük a típusban az `IDisposable` felület megvalósítását, az ügyfelek elkerülhetik a megsemmisítéssel járó hátrányokat. Az osztályunk immár illedelmes tagja a .NET közösségnek.

Az imént létrehozott rendszeren azonban még van néhány rés. Hogyan takaríthat ki maga után egy származtatott osztály úgy, hogy az alaposztály is megtehesse ezt? Ha a származtatott osztályok felülbírálják a `finalize` függvényt, vagy rendelkeznek az `IDisposable` saját megvalósításával, akkor ezeknek a tagfüggvényeknek meg kell hívniuk az alaposztályt, különben az alaposztály nem tud kitakarítani maga után. Ott van még az a gond is, hogy a `finalize` és a `Dispose` néhány feladata közös. Szinte bizonyos, hogy a `finalize` és a `Dispose` kódja ugyanolyan részeket is tartalmaz. Ahogy arról majd a 26. tippben szó lesz, a felületek függvényeinek felülbírálata nem úgy működik, mint ahogy várnánk. A felszabadító minta harmadik tagfüggvénye, ami egy védett virtuális segédfüggvény, kiszűri a közös fel-

adatokat, és a származtatott osztályokhoz egy horgot rendel, amivel felszabadíthatók az általuk lefoglalt erőforrások. Az alaposztály tartalmazza a felület lényegét képező kódot. A virtuális függvény adja a horgot a származtatott osztályoknak, hogy válaszul a `Dispose()` vagy a megsemmisítő eljárás meghívására felszabadíthassák az erőforrásokat:

```
protected virtual void Dispose( bool isDisposing );
```

Ez a túlterhelt függvény elvégzi a `finalize` és a `Dispose` függvények támogatásához szükséges munkát, és mivel virtuális függvényről van szó, belépési pontot kínál az összes származtatott osztály részére. A származtatott osztályok felülbírálhatják ezt a tagfüggvényt, megvalósítva ezzel az erőforrásaik felszabadítását, és így meghívhatják az alaposztályban található változatot is. A kezelt és kezeletlen erőforrásokat akkor szabadíthatjuk fel, amikor az `isDisposing` értéke igaz (`true`). Ha az `isDisposing` értéke hamis (`false`), akkor csak a kezeletlen erőforrásokat szabadíthatjuk fel. Az alaposztály `Dispose(bool)` függvényét mindkét esetben hívjuk meg, hogy az felszabadíthassa a saját erőforrásait.

Az alábbi egy rövid példa, ami bemutatja azt a kódrendszert, amit a minta használatakor meg kell írnunk. A `MyResourceHog` osztály az `IDisposable` megvalósításához, a megsemmisítéshez, és a virtuális `Dispose` tagfüggvény megírásához szükséges kódot is bemutatja:

```
public class MyResourceHog : IDisposable
{
  // Jelzőérték, ha már megtörtént a felszabadítás
  private bool _alreadyDisposed = false;

  // megsemmisítő:
  // Meghívjuk a virtuális Dispose tagfüggvényt
  ~MyResourceHog()
  {
    Dispose( false );
  }

  // Az IDisposable megvalósítása
  // A virtuális Dispose tagfüggvény meghívása
  // Letiltjuk a megsemmisítést
  public void Dispose()
  {
    Dispose( true );
    GC.SuppressFinalize( true );
  }

  // A virtuális Dispose tagfüggvény
  protected virtual void Dispose( bool isDisposing )
  {
```

```
    // Csak egyszer végezzük el a felszabadítást
    if ( _alreadyDisposed )
      return;
    if ( isDisposing )
    {
      // Teendő: itt szabadítsuk fel a kezelt erőforrásokat
    }
    // Teendő: itt szabadítsuk fel a kezeletlen erőforrásokat
    // Beállítjuk a felszabadítás megtörténtét jelző értéket:
    _alreadyDisposed = true;
  }
}
```

Ha egy származtatott osztálynak további tisztogatásra van szüksége, akkor elkészítheti a védett `Dispose` tagfüggvény megvalósítását:

```
public class DerivedResourceHog : MyResourceHog
{
  // Saját jelzőértéke van a felszabadítás megtörténtének jelzéséhez
  private bool _disposed = false;

  protected override void Dispose( bool isDisposing )
  {
    // Csak egyszer végezzük el a felszabadítást
    if ( _disposed )
      return;
    if ( isDisposing )
    {
      // Teendő: itt szabadítsuk fel a kezelt erőforrásokat
    }
    // Teendő: itt szabadítsuk fel a kezeletlen erőforrásokat

    // Hagyjuk, hogy az alaposztály felszabadítsa az erőforrásait
    // Az alaposztály felel a GC.SuppressFinalize()
    // meghívásáért
    base.Dispose( isDisposing );

    // Beállítjuk a származtatott osztályhoz tartozó
    // felszabadítás-jelző értéket:
    _disposed = true;
  }
}
```

Figyeljük meg, hogy az alaposztály és a származtatott osztály is tartalmaz egy jelzőértéket az objektum felszabadított állapotának jelzésére. Erre pusztán óvatosságból van szükség. Az állapotjelző megkétszerezésével kivédhetjük azokat az esetleges hibákat, amelyek az egyes típus és nem az objektumot felépítő összes típus felszabadításakor keletkezhetnek.

A `Dispose` és a `finalize` függvényeket úgy kell megírnunk, hogy minden eshetőséget kivédjünk. Az objektumok felszabadítása bármilyen sorrendben bekövetkezhet. Találkozni fogunk olyan esettel, amikor a típus egy tagobjektumát már azelőtt felszabadítja a program, hogy meghívtuk volna a `Dispose()` tagfüggvényt. Ez nem okoz gondot, mert a `Dispose()` tagfüggvényt többször is meg lehet hívni. Ha olyan objektumra hívjuk meg, aminek a felszabadítása már megtörtént, akkor nem csinál semmit. A megsemmisítőkre hasonló szabályok érvényesek. A hivatkozott objektumok még a memóriában vannak, így nem kell figyelnünk a null hivatkozásokra. Az viszont előfordulhat, hogy egy hivatkozott objektum felszabadítása már megtörtént. Az is lehet, hogy a program már meg is semmisítette.

Ezzel elérkeztünk a legfontosabb tanácshoz, amit az erőforrások felszabadítására szolgáló tagfüggvényekkel kapcsolatban adhatunk: mindig csak az erőforrások felszabadítására használjuk őket. Semmilyen más műveletet ne hajtsunk végre egy felszabadító tagfüggvényben. Súlyos bonyodalmakat idézhetünk elő az objektumok élettartamát illetően, ha egyéb műveleteket is végzünk a `Dispose` vagy a `finalize` tagfüggvényekben. Az objektumok akkor születnek, amikor meghívjuk a konstruktorukat, és akkor haláloznak el, amikor a szemétgyűjtő magához szólítja őket. Amikor a programunk már nem képes elérni őket, akkor úgy vehetjük, hogy egyfajta kómás állapotban vannak. Ha nem tudunk elérni egy objektumot, akkor nem tudjuk meghívni a tagfüggvényeit. Vagyis ami minket illet, az objektum eltávozott az élők sorából. A megsemmisítővel rendelkező objektumok azonban lehetőséget kapnak még egy utolsó leheletre, mielőtt kimúlnának. A megsemmisítőknek kizárólag a kezeletlen erőforrások felszabadításával szabad foglalkozniuk. Ha egy megsemmisítő valahogy ismét elérhetővé tesz egy objektumot, akkor az *feltámad*. Él és nem túlzottan virul, habár felébredt a kómából. Álljon itt egy kézenfekvő példa:

```
public class BadClass
{
  // Hivatkozás tárolása egy globális objektumra:
  private readonly ArrayList _finalizedList;
  private string _msg;

  public BadClass( ArrayList badList, string msg )
  {
    // A hivatkozás gyorstárba helyezése:
    _finalizedList = badList;
    _msg = (string)msg.Clone();
  }

  ~BadClass()
  {
    // Hozzáadjuk ezt az objektumot a listához
    // Az objektum elérhető, tehát
    // már nem szemét. Visszatért!
    _finalizedList.Add( this );
  }
}
```

Amikor a `BadClass` objektum végrehajtja a megsemmisítőjét, elhelyez saját magára egy hivatkozást egy globális listában, amivel elérhetővé teszi magát, és ismét életre kel. Ezzel annyi kalamajkát idézünk elő, hogy még rágondolni is rossz. Az objektumot megsemmisítettük, így a szemétgyűjtő most már úgy véli, hogy nem kell meghívnia többször a megsemmisítőjét. Ha tényleg meg szeretnénk semmisíteni egy feltámasztott objektumot, az sajnos nem fog menni. Másodszor pedig, néhány erőforrásunk elveszhet. A szemétgyűjtő nem távolítja el azokat az objektumokat a memóriából, amelyeket csak a megsemmisítési sorban lévő objektumok érhetnek el, de lehet, hogy már megsemmisítette őket. Ha így van, akkor már szinte bizonyosan használhatatlanok. Habár azok a tagok, amelynek a `BadClass` a tulajdonosa, még a memóriában vannak, valószínűleg már megtörtént a megsemmisítésük. A nyelvben semmilyen módszer nincs a megsemmisítés sorrendjének befolyásolására. Ezt a fajta szerkezetet nem tudjuk megbízhatóan működtetni. Ne is próbálkozzunk vele.

Érvelések alátámasztásán kívül persze még soha nem láttam olyan kódot, amiben ennyire nyilvánvalóan szerepeltek volna feltámasztott objektumok. De olyan kódot már láttam, amiben a megsemmisítő valamilyen tényleges feladatot próbált elvégezni, és végül feltámasztotta önmagát, amikor egy függvény, amit a megsemmisítő hívott meg, elraktározott egy hivatkozást az objektumra. A tanulság az, hogy mindig gondosan nézzük át a megsemmisítő kódját, és ennek kiegészítéseként mindkét `Dispose` tagfüggvény kódját is. Ha a kód valami mást is csinál az erőforrások felszabadításán kívül, akkor nézzük át még egyszer. Ezek a műveletek ugyanis valószínűleg programhibákhoz vezetnek majd a jövőben. Ezért szabaduljunk meg ezektől a műveletektől, és gondoskodjunk róla, hogy a megsemmisítők és a `Dispose()` tagfüggvények csak az erőforrások felszabadításával foglalkozzanak, és semmi mással.

Egy kezelt környezetben nem kell minden létrehozott típushoz megsemmisítő kódot írnunk. Csak azoknál a típusoknál van erre szükség, amelyek kezeletlen típusokat tárolnak, vagy amikor a típus olyan tagokat tartalmaz, amelyek megvalósítják az `IDisposable` felületet. Ha csak a `Disposable` felületre van szükségünk, és nem egy megsemmisítőre, akkor is mindig írjuk meg a teljes mintát. Ha nem így teszünk, azzal korlátozzuk a származtatott osztályokat, mivel sokkal bonyolultabb lesz megvalósítani velük a szabványos felszabadító eljárást. Ezért mindig kövessük az itt leírt szabványos erőforrás-felszabadító mintát. Ezzel megkönnyítjük a saját, az osztályunk felhasználóinak, és a típusunkból származtatott osztályokat írók életét is.

3

Tervezés C# alapokon

A C# nyelvben új nyelvi elemeket vezettek be a programtervek megvalósításához. Szándékainkat mindig az általunk választott megoldás árulja el azoknak a fejlesztőknek, akik használják, bővítik és karbantartják a programjainkat. A C# típusok egytől-egyig a .NET környezet részei. A környezet bizonyos dolgokat feltételez a típusok képességeiről. Ha megsértjük ezeket az elvárásokat, azzal megnöveljük annak a valószínűségét, hogy a típusaink helytelenül fognak működni.

A következő tippek nem programtervezési útmutatóként szolgálnak, hiszen a programtervezésről többkötetes műveket írtak már. A most következő tippekben inkább arra helyezzük a hangsúlyt, hogy a különféle C# nyelvi elemek melyik programtervezési minta megvalósítására a legalkalmasabbak. A C# nyelv megalkotói olyan szolgáltatásokat adtak a nyelvhez, amelyekkel világosabban fogalmazhatjuk meg a modern programtervezési mintákat. Egyes nyelvi elemek között igen finom különbségek vannak csak, és sokszor több választási lehetőségünk is van. A választás lehetősége elsőre a „legjobb" megoldásnak tűnhet, de a különbségek sokszor csak később derülnek ki, amikor rájövünk, hogy tovább kell fejlesztenünk egy létező programot. Figyelmesen olvassuk el ezeket a tippeket, amíg meg nem értjük őket, és a bennük lévő tanácsok alkalmazásakor a fél szemünket mindig tartsuk azokon a lehetséges bővítéseken, amiket a készülő rendszeren el kell majd végeznünk.

Némelyik változás a nyelvtanban olyan új fogalmak bevezetésével jár, amelyekkel a naponta használt eljárásokat írhatjuk le. Ilyenek például a tulajdonságok, az indexelők, az események és a képviselők. De ilyen az osztályok és felületek közti különbségek leírása is. Az osztályok típusokat határoznak meg. A felületek viselkedéseket vezetnek be. Az alaposztályok típusokat vezetnek be, és típusok egymással összefüggő halmazának közös viselkedését határozzák meg. Vannak olyan programtervezési minták, amelyek a szemétgyűjtés miatt változtak meg. Megint mások azért, mert a változók többsége hivatkozási típusú.

A fejezetben található javaslatok segítenek megtalálni a legtermészetesebb kifejezésmódot a programterveink megvalósításához. Ez lehetővé teszi, hogy olyan programokat írjunk, amelyeket könnyebb karbantartani, könnyebb bővíteni, és egyszerűbb használni.

19. tipp

Használjunk felületeket az öröklés helyett

Az elvont alaposztályok közös ősként szolgálnak egy osztályhierarchiában. A felületek valamilyen elemi szolgáltatást írnak le, amit egy típusban valósíthatunk meg. Mindegyiknek megvan a maga helye, de nem ugyanott. A felületekkel megállapodás kérdése a programtervezés: ha egy típus megvalósít egy felületet, akkor a megfelelő tagfüggvények megvalósítását is tartalmaznia kell. Az elvont alaposztályok egymással valamilyen kapcsolatban lévő típusok halmazának közös elvont ábrázolását adják. Közhely, de igaz, hogy az öröklés azt jelenti, hogy „valami egy valami" („is-a" kapcsolat), a felület pedig azt, hogy „úgy viselkedik, mint valami". Ezek a közhelyek azért maradtak fenn oly rég óta, mert jól megfogható velük a két szerkezet közti különbség. Az alaposztályok azt írják le, hogy egy objektum micsoda, a felületek pedig az objektum egy adott viselkedését adják meg.

A felületek bizonyos szolgáltatások egy halmazát írják le, amit felfoghatunk egy megállapodásként is. A felületekben bárminek hagyhatunk helyet, tagfüggvényeknek, tulajdonságoknak, indexelőknek, és eseményeknek is. Minden olyan típusnak, ami megvalósítja a felületet, tartalmaznia kell a felületben meghatározott összes elem konkrét megvalósítását. Meg kell adnunk az összes tagfüggvény megvalósítását, ha pedig vannak tulajdonságok, akkor mindegyikhez meg kell adnunk a megfelelő hozzáférőket és indexelőket, és meg kell határoznunk a felületben megadott összes eseményt is. Az újrahasznosítható viselkedési formákat azonosítjuk, majd felületekbe gyűjtjük. A felületeket paraméterként és visszatérési értékként használjuk. A kód ismételt felhasználására azért is több lehetőségünk lesz, mert egymástól teljesen eltérő típusok is tartalmazhatják egyazon felület megvalósítását. Sőt a többi fejlesztőnek is könnyebb megvalósítani egy felületet, mint származtatni valamit az általunk létrehozott alaposztályból.

Amit a felületekben nem tudunk megtenni, az az, hogy megírjuk ezeknek a tagoknak a megvalósítását. A felületek semmiféle megvalósítást nem tartalmaznak, és konkrét adattagok sem lehetnek bennük. Egy megállapodás alapjait fektetjük le, amit az összes olyan típusnak támogatnia kell, amelyik megvalósítja a felületet.

Az elvont alaposztályok képesek néhány dolog megvalósítására a származtatott osztályok számára, a közös viselkedés leírásán kívül. Megadhatunk adattagokat, konkrét tagfüggvényeket, virtuális tagfüggvények megvalósítását, tulajdonságokat, eseményeket és indexelőket. Egy alaposztályban megadhatjuk néhány tagfüggvény megvalósítását, amivel egységessé tehetjük a megvalósítások újrahasznosítását. Bármelyik elem lehet virtuális, elvont, és nem virtuális is. Az elvont alaposztályok bármely konkrét viselkedés megvalósítását megadhatják. A felületek nem.

A megvalósítások újrahasznosításának van még egy előnye. Ha hozzáadunk egy tagfüggvényt az alaposztályhoz, akkor az összes származtatott osztály automatikusan bővülni fog, minden további nélkül. Ebben az értelemben az alaposztályok hatékony módszert kínálnak arra, hogy az idő multával több típus viselkedését is bővíthessük. Ha hozzáadjuk egy szolgáltatás megvalósítását az alaposztályhoz, az összes származtatott osztálynak abban a pillanatban szintén része lesz az adott viselkedés. Ha egy felülethez adunk hozzá egy tagot, azzal elrontjuk az összes olyan osztályt, amelyik megvalósítja a felületet. Ezekben nem fog szerepelni az új tagfüggvény, és így nem lehet lefordítani őket. Minden osztály megvalósítását frissíteni kell, hogy azok is tartalmazzák az új tagfüggvényt.

Az elvont alaposztályok és a felületek közti választás azon múlik, hogy melyik felel meg jobban az elvont fogalmainknak a későbbiekben. A felületek rögzítettek, azokat bizonyos szolgáltatások egy csoportjára vonatkozó megállapodásként tesszük közzé, amelyet bármely típus megvalósíthat. Az alaposztályok később is bővíthetők, a bővítések minden származtatott osztály részévé válnak.

A két modellt keverhetjük is, vagyis az újrahasznosítható megvalósítások kódja mellett felületeket is támogathatunk. Ilyen például a `System.Collections.CollectionBase` osztály, ami egy olyan alaposztályt bocsát a rendelkezésünkre, amivel megvédhetjük az ügyfeleket a .NET környezet típusbiztonságának hiányától. Eközben számos felület megvalósítását végzi el helyettünk: ezek az `IList`, az `ICollection`, és az `IEnumerable`. Ezen kívül védett tagfüggvényei is vannak, amelyeket felülbírálhatunk, hogy viselkedésüket a különféle felhasználási módokhoz igazítsuk. Az `IList` felületben van egy `Insert()` tagfüggvény, amivel új objektumokat adhatunk egy gyűjteményhez. Ahelyett, hogy elkészítenénk a saját `Insert`-megvalósításunkat, az ilyen eseményeket a `CollectionBase` osztály `OnInsert()` vagy `OnInsertComplete()` virtuális tagfüggvényeinek felülbírálásával dolgozhatjuk fel.

```
public class IntList : System.Collections.CollectionBase
{
  protected override void OnInsert( int index, object value )
  {
    try
    {
      int newValue = System.Convert.ToInt32( value );
      Console.WriteLine( "Inserting {0} at position {1}",
        index.ToString(), value.ToString());
        Console.WriteLine( "List Contains {0} items",
        this.List.Count.ToString());
    }
    catch( FormatException e )
    {
      throw new ArgumentException(
       "Argument Type not an integer",
       "value", e );
    }
  }
```

```
    protected override void OnInsertComplete( int index,
      object value )
    {
      Console.WriteLine( "Inserted {0} at position {1}",
        index.ToString( ), value.ToString( ));
      Console.WriteLine( "List Contains {0} items",
        this.List.Count.ToString( ) );
    }
  }

  public class MainProgram
  {
    public static void Main()
    {
      IntList l = new IntList();
      IList il = l as IList;
      il.Insert( 0,3 );
      il.Insert( 0, "This is bad" );
    }
  }
```

A fenti kód egy egészekből álló tömblistát hoz létre, és az `IList` felületmutató segítségével két különböző értéket ad a gyűjteményhez. Az `IntList` osztály az `OnInsert()` tagfüggvény felülbírálásával ellenőrzi a beszúrt értéket, és ha az nem egész, akkor kivételt vált ki. Az alaposztály tartalmazza az alapértelmezett megvalósítást, és horgokat kínál a származtatott osztályok viselkedéseknek testreszabásához.

A `CollectionBase`, vagyis az alaposztály, ad egy megvalósítást, amit felhasználhatunk a saját osztályainkhoz. Közel sem kell olyan sokat kódolnunk, hiszen felhasználhatjuk a már létező közös megvalósítást. Az `IntList` nyilvános alkalmazás-programozási felületét (API) viszont a `CollectionBase` osztályban megvalósított felületek adják: ezek az `IEnumerable`, az `ICollection`, és az `IList` felületek. A `CollectionBase` egy általános megvalósítást ad a felületekhez, amit felhasználhatunk.

Ezzel elérkeztünk következő témánkhoz, a felületek paraméterként és visszatérési értékként való használatához. Egy felületet akárhány egymástól független típus megvalósíthat. Felületek írásával sokkal nagyobb rugalmasságot adhatunk a többi fejlesztőnek, mintha alaposztályokat írnánk. Ez a .NET környezetben kötelezően egységes öröklési hierarchia miatt fontos.

Ez a két tagfüggvény ugyanazt a feladatot végzi el:

```
public void PrintCollection( IEnumerable collection )
{
  foreach( object o in collection )
  Console.WriteLine( "Collection contains {0}",
    o.ToString( ) );
}

public void PrintCollection( CollectionBase collection )
{
  foreach( object o in collection )
  Console.WriteLine( "Collection contains {0}",
    o.ToString( ) );
}
```

A második tagfüggvény újrahasznosítása sokkal nehezebb, hiszen azt nem használhatjuk például `Array`, `ArrayList`, `DataTable`, `HashTable`, `ImageList` és sok más gyűjteményes osztállyal sem. Sokkal általánosabb és könnyebben újrahasznosítható megoldás, ha a tagfüggvény paramétereként egy felületet adunk meg.

Ha egy osztályhoz tartozó API megadásához felületeket használunk, azzal szintén nagyobb rugalmasságot érünk el. Sok alkalmazás például egy `DataSet` segítségével mozgatja az adatokat az alkalmazás különböző részei között. Könnyen beleeshetünk abba a hibába, hogy ezt a bevett szokást kötelezővé tesszük:

```
public DataSet TheCollection
{
  get { return _dataSetCollection; }
}
```

Ezzel kiszolgáltatottá válunk a jövőben bekövetkező problémákkal szemben. Előfordulhat, hogy egyszer csak egy `DataSet` helyett egy `DataTable`, egy `DataView` vagy éppen egy saját objektumot használunk majd. Bármelyikre is váltsunk, azzal elrontjuk a kódot. Persze módosíthatjuk a paraméter típusát, de azzal megváltoztatjuk az osztályunk nyilvános felületét is. Egy nagyobb rendszer esetében az osztály nyilvános felületének módosítása számos más módosítást von maga után, hiszen minden olyan helyen változásokat kell eszközölnünk, ahol hozzáfértünk a nyilvános tulajdonsághoz.

Van egy másik gond is, ami sokkal hamarabb és sokkal vészjóslóbban jelentkezik. A `DataSet` osztály számtalan tagfüggvényt kínál a benne lévő adatok módosításához. Az osztályunk felhasználói táblákat törölhetnek, oszlopokat módosíthatnak, de akár le is cserélhetik a `DataSet` összes objektumát. Ezt bizonyára nem szeretnénk. Szerencsére be-

határolhatjuk az osztály használóinak mozgásterét. Ahelyett, hogy egy DataSet típusra mutató hivatkozást adunk vissza, inkább adjuk vissza azt a felületet, amit az ügyfelek a szándékainknak megfelelően használhatnak. A DataSet támogatja az IListSource felületet, amit az adatkötéshez használ:

```
using System.ComponentModel;

public IListSource TheCollection
{
  get { return _dataSetCollection as IListSource; }
}
```

Az IListSource a GetList() tagfüggvényen keresztül engedi, hogy az ügyfelek megnézhessék az elemeket. Van egy ContainsListCollection tulajdonsága is, hogy a felhasználók módosíthassák a gyűjtemény általános szerkezetét. Az IListSource felületet használva a DataSet egyes elemei elérhetők, de a DataSet szerkezetét nem lehet módosítani. A hívónak arra sincs lehetősége, hogy a DataSet tagfüggvényeinek meghívásával megváltoztassa az elérhető adatműveleteket a megszorítások eltávolításával vagy új képességek megadásával.

Ha a típusunk osztálytípusként teszi elérhetővé a tulajdonságokat, akkor a teljes felületet is elérhetővé teszi az osztályok részére. Ha felületeket használunk, akkor eldönthetjük, hogy pontosan mely tagfüggvényeket és tulajdonságokat használhassák az ügyfelek. A felület megvalósításához használt osztály csak egy megvalósítással kapcsolatos részletkérdés, amit később módosíthatunk (lásd a 23. tippet).

Ráadásul egymástól teljesen független típusok is megvalósíthatják ugyanazt a felületet. Tegyük fel, hogy egy alkalmazottakat, vevőket és kereskedőket kezelő alkalmazást készítünk. Ezek egymástól függetlenek, legalábbis ami az osztályhierarchiát illeti. Van azonban néhány dolog, amit ugyanúgy csinálnak. Mindegyiknek van neve, és valószínűleg majd Windows vezérlőkben akarjuk majd megjeleníteni ezeket a neveket az alkalmazásunkban.

```
public class Employee
{
  public string Name
  {
    get
    {
      return string.Format( "{0}, {1}", _last, _first );
    }
  }

  // a többi részlet elhagyva
}
```

```
public class Customer
{
  public string Name
  {
    get
    {
      return _customerName;
    }
  }

  // a többi részlet elhagyva
}

public class Vendor
{
  public string Name
  {
    get
    {
      return _vendorName;
    }
  }
}
```

Nem szabadna, hogy az `Employee`, a `Customer` és a `Vendor` osztályoknak közös alaposztálya legyen. Van azonban néhány közös tulajdonságuk: a nevek (ahogy azt korábban már láttuk), a címek és a telefonszámok. Ezeket a közös tulajdonságokat kigyűjthetnénk egy felületbe:

```
public interface IContactInfo
{
  string Name { get; }
  PhoneNumber PrimaryContact { get; }
  PhoneNumber Fax { get; }
  Address PrimaryAddress { get; }
}

public class Employee : IContactInfo
{
  // megvalósítás törölve
}
```

Ez az új felület leegyszerűsítheti a további programozási feladatokat, hiszen így közös eljárásokat készíthetünk egymástól eltérő típusokhoz:

```
public void PrintMailingLabel( IContactInfo ic )
{
  // megvalósítás törölve
}
```

Ez az eljárás mindennel működik, ami megvalósítja az `IContactInfo` felületet. A `Customer`, az `Employee` és a `Vendor` mind ugyanazt az eljárást használhatják. Erre viszont csak azért van lehetőség, mert készítettünk hozzájuk egy közös felületet.

A felületek használata azt is jelenti, hogy időnként megtakaríthatunk egy kicsomagolásnyi időt a struktúrák esetében. Amikor becsomagolunk egy struktúrát, a csomag támogatni fogja az összes olyan felületet, amit a struktúra támogat. Amikor a felületmutatón keresztül férünk hozzá a struktúrához, akkor nem kell kicsomagolnunk a struktúrát, hogy elérhessük az objektumot. Példaként képzeljünk el egy struktúrát, ami egy kapcsolatot és annak leírását tartalmazza:

```
public struct URLInfo : IComparable
{
  private string URL;
  private string description;

  public int CompareTo( object o )
  {
    if (o is URLInfo)
    {
      URLInfo other = ( URLInfo ) o;
      return CompareTo( other );
    }
    else
      throw new ArgumentException(
        "Compared object is not URLInfo" );
  }

  public int CompareTo( URLInfo other )
  {
    return URL.CompareTo( other.URL );
  }
}
```

Az `URLInfo` objektumokból készíthetünk egy rendezett listát, mert az `URLInfo` megvalósítja az `IComparable` felületet. Az `URLInfo` struktúrákat becsomagolja a program, amikor hozzáadja őket a listához. A `Sort()` tagfüggvénynek viszont nem kell kicsomagolnia mindkét objektumot ahhoz, hogy meghívja a `CompareTo()` függvényt. A paramétert (`other`) ugyan ki kell csomagolnunk, de az összehasonlítás bal oldalát nem kell kicsomagolnunk ahhoz, hogy meghívhassuk az `IComparable.CompareTo()` tagfüggvényt.

Az alaposztályok egymással kapcsolatban lévő konkrét típusok közös viselkedését írják le, illetve valósítják meg. A felületek elemi szolgáltatásokat írnak le, amit egymástól független típusok is megvalósíthatnak. Mindkettőnek megvan a maga helye. Az osztályok az általunk létrehozott típusokat határozzák meg. A felületek ezeknek a típusoknak a viselkedését írják

le elemi szolgáltatások formájában. Ha értjük a különbséget a kettő között, akkor sokkal kifejezőbben tudunk programozni, és az eredmény rugalmasabban követi majd a későbbi változásokat. Az osztályhierarchiákat használjuk az egymással kapcsolatban álló típusok meghatározásához, a szolgáltatásokat pedig felületekkel vigyük rá ezekre a típusokra.

20. tipp

Ne keverjük össze a felületek megvalósítását a virtuális függvények felülbírálásával

Első pillantásra úgy tűnhet, hogy a felületek megvalósítása nem más, mint virtuális függvények felülbírálása. Meghatározunk egy olyan tagot, amit egy másik típusban vezettek be. Ez az első pillantás elég csalóka. A felületek megvalósítása teljesen eltér a virtuális függvények felülbírálásától. A felületekben bevezetett tagok nem virtuálisak, legalábbis alapértelmezés szerint nem azok. A származtatott osztályok nem tudják felülbírálni az alaposztályokban megvalósított felülettagokat. A felületeket konkrétan megvalósíthatjuk, ami elrejti őket az osztályok nyilvános felületétől. Ezek különböző fogalmak, és mindegyiknek megvan a maga felhasználási területe.

A felületeket azonban úgy is megvalósíthatjuk, hogy a származtatott osztályok módosíthassák a mi megvalósításunkat. Ehhez mindössze kapcsolódási pontokat, horgokat kell készítenünk a származtatott osztályok részére.

A különbségeket jól megfigyelhetjük, ha megvizsgálunk egy egyszerű felületet és annak megvalósítását egy osztályban:

```
interface IMsg
{
  void Message();
}

public class MyClass : IMsg
{
  public void Message()
  {
    Console.WriteLine( "MyClass" );
  }
}
```

A Message() tagfüggvény a MyClass osztály nyilvános felületének része. A Message az IMsg ponton keresztül is elérhető, ami a MyClass típus része. Tegyük most még bonyolultabbá a helyzetet, és hozzunk létre egy származtatott osztályt:

```
public class MyDerivedClass : MyClass
{
  public new void Message()
  {
    Console.WriteLine( "MyDerivedClass" );
  }
}
```

Figyeljük meg, hogy hozzá kellett adnunk a new kulcsszót az előbbi Message tagfüggvény meghatározásához (lásd a 29. tippet). A MyClass.Message() nem virtuális függvény. A származtatott osztályok nem tudják felülbírálni a Message függvényt. A MyDerived osztály egy új Message tagfüggvényt hoz létre, és nem a MyClass.Message függvényt bírálja felül; azt elrejti. A MyClass.Message továbbra is elérhető az IMsg hivatkozáson keresztül:

```
MyDerivedClass d = new MyDerivedClass( );
d.Message( ); // azt írja ki, hogy "MyDerivedClass"
IMsg m = d as IMsg;
m.Message( ); // azt írja ki, hogy "MyClass"
```

A felületek tagfüggvényei nem virtuálisak. Amikor megvalósítunk egy felületet, akkor a kérdéses típusban egy adott megállapodás konkrét megvalósítását vezetjük be.

Gyakran előfordul azonban, hogy felületeket akarunk létrehozni, aztán alaposztályokban akarjuk megvalósítani őket, majd a származtatott osztályokban módosítani akarjuk a viselkedést. Ezt is megtehetjük, méghozzá kétféleképpen. Ha nem tudunk hozzáférni az alaposztályhoz, akkor ismét elkészíthetjük a felület megvalósítását a származtatott osztályban:

```
public class MyDerivedClass : MyClass, IMsg
{
  public new void Message()
  {
    Console.WriteLine( "MyDerivedClass" );
  }
}
```

Az IMsg kulcsszó beszúrása úgy változtatja meg a leszármazott osztály viselkedését, hogy az IMsg.Message() most már a leszármazott osztályban található változat szerint fog működni:

```
MyDerivedClass d = new MyDerivedClass( );
d.Message( ); // azt írja ki, hogy "MyDerivedClass"
IMsg m = d as IMsg;
m.Message( ); // azt írja ki, hogy "MyDerivedClass"
```

A new kulcsszóra továbbra is szükség van a MyDerivedClass.Message() tagfüggvénynél. Innen kitalálhatjuk, hogy még mindig gondok vannak (lásd a 29. tippet). Az alaposztályban található változat továbbra is elérhető az alaposztályra mutató hivatkozáson keresztül:

```
MyDerivedClass d = new MyDerivedClass( );
d.Message( ); // azt írja ki, hogy "MyDerivedClass"
IMsg m = d as IMsg;
m.Message( ); // azt írja ki, hogy "MyDerivedClass"
MyClass b = d;
b.Message( ); // azt írja ki, hogy "MyClass"
```

Ezt a problémát csak úgy oldhatjuk meg, ha módosítjuk az alaposztályt, megadva, hogy a felület tagfüggvényei virtuálisak legyenek:

```
public class MyClass : IMsg
{
  public virtual void Message()
  {
    Console.WriteLine( "MyClass" );
  }
}

public class MyDerivedClass : MyClass
{
  public override void Message()
  {
    Console.WriteLine( "MyDerivedClass" );
  }
}
```

A MyDerivedClass, és a MyClass osztályból származtatott összes osztály bevezetheti a Message() függvény saját változatát. A felülbírált változat minden alkalommal meghívódik: a MyDerivedClass, az IMsg, és a MyClass hivatkozásokon keresztül is.

Ha fanyalogva fogadjuk a nem tiszta virtuális függvényeket, akkor csak egy apró változtatást hajtsunk végre a MyClass meghatározásán:

```
public abstract class MyClass, IMsg
{
  public abstract void Message();
}
```

Bizony, egy felületet anélkül is megvalósíthatunk, hogy ténylegesen megírnánk a benne lévő tagfüggvények megvalósítását. Azáltal, hogy a felület tagfüggvényeinek elvont változatát

vezetjük be, leszögezzük, hogy a típusunkból származtatott összes típusnak meg kell valósítania ezt a felületet. Az `IMsg` felület a `MyClass` osztály bevezetésének a részét képezi, de a tagfüggvények meghatározásának feladata az egyes leszármazott osztályokra hárul.

A felületek konkrét megvalósítása lehetővé teszi, hogy megvalósítsunk egy felületet, és mégis elrejtsük annak tagjait a típus nyilvános felülete elől. Ezzel néhány csavart teszünk a felületek megvalósítása és a virtuális függvények felülbírálása közti kapcsolatba. A felületek konkrét megvalósítását arra használjuk, hogy korlátozzuk az ügyfeleket abban, hogy a felület tagfüggvényeit használják, amikor egy alkalmasabb változat is elérhető. A 26. tippben az `IComparable` mintával kapcsolatban ezt részletesen bemutatjuk.

A felületek megvalósítása több lehetőséget nyújt, mint a virtuális függvények létrehozása és felülbírálása. Létrehozhatunk zárt megvalósításokat, virtuális megvalósításokat és elvont megállapodásokat is az osztályhierarchiáknál. Pontosan eldönthetjük, hogy a származtatott osztályok mikor és hogyan módosíthatják az osztályunkban található felületek tagjainak viselkedését. A felületek tagfüggvényei nem virtuális tagfüggvények, hanem egy külön megállapodás részei.

21. tipp

Használjunk képviselőket a visszahívások kifejezésére

Én: „Fiam, nyírd le a füvet! Addig én olvasok egy kicsit."
Lacika: „Apu, rendet raktam a kertben."
Lacika: „Apu, tettem benzint a fűnyíróba."
Lacika: „Apu, nem indul be a fűnyíró."
Én: „Majd én beindítom."
Lacika: „Apu, kész vagyok."

Ez a kis párbeszéd a visszahívásokat szemlélteti. Adtam a fiamnak egy feladatot, és ő (újra meg újra) félbeszakított, hogy közölje, hol tart. Nem kellett abbahagynom, amit csináltam, miközben arra vártam, hogy befejezze a feladat egyes részeit. Időnként félbe tudott szakítani, amikor valami fontos (vagy éppen kevésbé fontos) részhez ért, vagy szüksége volt a segítségemre. A visszahívások segítségével általában a kiszolgálók szoktak visszajelezni aszinkron módon az ügyfeleknek. Működésük többszálas is lehet, de lehet, hogy csak belépési pontként szolgálnak a szinkron módon történő frissítésekhez. A C# nyelvben a visszahívásokat képviselők segítségével valósíthatjuk meg.

A képviselők típusbiztos meghatározásokat nyújtanak a visszahívásokhoz. Habár a képviselőket leggyakrabban az eseményekkel használják, ez a nyelvi elem nem feltétlenül csak erre való. Amikor több osztály közötti kommunikációt kell létrehoznunk, és kevesebb kapcsolódási pont lenne kívánatos, mint amennyit a felületek használata jelentene, mindig

a képviselőkre essen a választásunk. A képviselőkkel futásidőben adhatjuk meg a célt, és több ügyfelet is értesíthetünk. A képviselők olyan objektumok, amelyek egy tagfüggvényre mutató hivatkozást tartalmaznak. Ez a tagfüggvény lehet egy statikus tagfüggvény vagy egy példány tagfüggvény is. A képviselőt használva egy vagy több futásidőben beállított ügyfélobjektummal is kapcsolatot tarthatunk fenn.

A csoportos képviselők (multicast delegate) egyetlen függvényhívásba fogják mindazokat a függvényeket, amelyeket hozzáadtunk az adott képviselőhöz. Ezzel a szerkezettel kapcsolatban két dologra kell felhívni a figyelmet: a kivételek ellen nem véd meg, és a visszatérési értéke az utoljára meghívott függvény visszatérési értéke lesz.

Egy csoportos képviselő hívásának belsejében minden cél hívása megtörténik, szépen egymás után. A képviselő semmilyen kivételt nem fog el. Ezért ha a cél kivált egy kivételt, az megszakítja a képviselőn belüli hívások láncolatát.

A visszatérési értékekkel kapcsolatban hasonló problémával találkozunk. Olyan képviselőket is meghatározhatunk, amelyeknek visszatérési értéke nem `void` lesz. Írhatunk például egy olyan visszahívást, ami azt ellenőrzi, hogy megszakított-e egy programot a felhasználó:

```
public delegate bool ContinueProcessing();

public void LengthyOperation( ContinueProcessing pred )
{
  foreach( ComplicatedClass cl in _container )
  {
    cl.DoLengthyOperation();
    // ellenőrizzük, hogy nem szakította-e meg a programot a felhasználó
    if (false == pred())
      return;
  }
}
```

Ez egy sima képviselő esetében működik, de csoportos használata már gondot jelenthet:

```
ContinueProcessing cp = new ContinueProcessing ( CheckWithUser );
cp += new ContinueProcessing( CheckWithSystem );
c.LengthyOperation( cp );
```

A képviselő meghívásakor visszakapott érték a csoportos hívásláncban utolsóként meghívott függvény visszatérési értéke lesz. A többi visszatérési értéket figyelmen kívül hagyja a program. A `CheckWithUser()` visszatérési értékét sem veszi figyelembe.

Mindkét gondot úgy oldhatjuk meg, ha minden képviselő célpontját mi magunk hívjuk meg. Minden általunk létrehozott képviselő tartalmaz egy képviselőlistát. A lánc vizsgálatához és az egyes elemek meghívásához magunknak kell bejárnunk a listát:

```
public delegate bool ContinueProcessing();

public void LengthyOperation( ContinueProcessing pred )
{
  bool bContinue = true;
  foreach( ComplicatedClass cl in _container )
  {
    cl.DoLengthyOperation();
    foreach( ContinueProcessing pr in
      pred.GetInvocationList( ))

      bContinue &= pr();

    if (false == bContinue)
      return;
  }
}
```

Ebben az esetben úgy határoztuk meg a program működését, hogy a bejárás csak akkor folytatódhasson, ha minden egyes képviselő igaz értékekkel rendelkezik.

A képviselők kínálják a legjobb módját a visszahívások alkalmazására futásidőben, egyszerűbb követelményeket támasztva az ügyfélosztályokkal szemben. A képviselők célpontjait futásidőben is beállíthatjuk. Lehetőség van egyszerre több ügyfélcélpont támogatására is. A .NET környezetben tehát a visszahívásokat mindig képviselők segítségével valósítsuk meg.

22. tipp

A kimenő felületeket eseményekkel határozzuk meg

A típusaink kimenő felületét az események határozzák meg. Az eseményeket képviselőkre építjük, hogy típusbiztos függvényaláírásokat biztosítsunk az eseménykezelők részére. Ha ehhez hozzávesszük, hogy a legtöbb programozási példa, ami képviselőket használ, egy eseményt ír le, akkor nem szabad csodálkoznunk azon, hogy a fejlesztők elkezdik azt hinni, hogy az események megegyeznek a képviselőkkel. A 21. tippben már láttunk egy példát arra, hogy mikor használhatunk képviselőket események meghatározása nélkül. Eseményeket akkor kell kiváltanunk, amikor a típusunknak több ügyféllel is kommunikálnia kell, értesítve őket a rendszerben végbemenő műveletekről.

Vegyünk egy egyszerű példát. Egy naplózó osztályt készítünk, ami az alkalmazás összes üzenetének elosztójaként működik. A programban található összes forrástól elfogad üzeneteket, és továbbadja azokat bárminek, ami figyel ezekre. Ezek a figyelők kapcsolódhatnak a konzolhoz, egy adatbázishoz, a rendszernaplóhoz vagy valamilyen más mechanizmushoz is.

Az osztályt az alábbiak szerint határozhatjuk meg, mindig kiváltva egy eseményt, amikor beérkezik egy üzenet:

```
public class LoggerEventArgs : EventArgs
{
  public readonly string Message;
  public readonly int Priority;

  public LoggerEventArgs ( int p, string m )
  {
    Priority = p;
    Message = m;
  }
}

// Meghatározzuk az aláírást az eseménykezelőhöz:
public delegate void AddMessageEventHandler( object sender,
  LoggerEventArgs msg );

public class Logger
{
  static Logger( )
  {
    _theOnly = new Logger( );
  }

  private Logger( )
  {
  }

  private static Logger _theOnly = null;
  public Logger Singleton
  {
    get
    {
      return _theOnly;
    }
  }

  // Meghatározzuk az eseményt:
  public event AddMessageEventHandler Log;
```

```
    // beszúrunk egy eseményt és naplózzuk
    public void AddMsg ( int priority, string msg )
    {
      // Erről a mintáról az alábbiakban lesz szó
      AddMessageEventHandler l = Log;
      if ( l != null )
        l ( null, new LoggerEventArgs( priority, msg ) );
    }
  }
```

Az AddMsg tagfüggvény az események kiváltásának helyes módját szemlélteti. A naplózási eseménykezelőre hivatkozó átmeneti változó fontos biztonsági lépés, amivel kivédhetjük a többszálú programoknál keletkező versenyhelyzeteket. A hivatkozás másolata nélkül az ügyfelek eseménykezelőket törölhetnénk az if utasítással végzett ellenőrzés és az eseménykezelő végrehajtása között. Ha lemásoljuk a hivatkozást, akkor ez nem fordulhat elő.

A LoggerEventArgs osztályt úgy határoztuk meg, hogy az tartalmazza egy esemény fontosságát és az üzenetet. A képviselő határozza meg az aláírást az eseménykezelő részére. A Logger osztály belsejében az eseménymező határozza meg az eseménykezelőt. A fordító látja a nyilvános eseménymező meghatározását, és létrehozza helyettünk az Add (Beszúrás) és a Remove (Törlés) műveleteket. Az előállított kód pontosan ugyanaz lesz, mintha ezt írtuk volna:

```
public class Logger
{
  private AddMessageEventHandler _Log;

  public event AddMessageEventHandler Log
  {
    add
    {
      _Log = _Log + value;
    }
    remove
    {
      _Log = _Log - value;
    }
  }

    public void AddMsg (int priority, string msg)
    {
      AddMessageEventHandler l = _Log;
      if (l != null)
        l (null, new LoggerEventArgs (priority, msg));
    }
  }
}
```

A C# fordító elkészíti az esemény add és remove hozzáférőit. Ez a nyilvános esemény-bevezetési stílus szerintem tömörebb, egyszerűbben olvasható és karbantartható, és helyesebb is. Ha eseményeket hozunk létre az osztályokban, akkor mindig nyilvános eseményeket vezessünk be, és hagyjuk, hogy a fordító hozza létre helyettünk az add és a remove tulajdonságokat. Ha más szabályokat is be szeretnénk tartatni, akkor megtehetjük, és jobb választás is, hogy mi magunk írjuk meg ezeket a kezelőfüggvényeket.

Az eseményeknek semmit nem kell tudniuk a lehetséges figyelőkről. Az alábbi osztály minden üzenetet automatikusan a Standard Error konzolra (a szabványos hibakonzolra) irányít:

```
class ConsoleLogger
{
  static ConsoleLogger()
  {
    logger.Log += new AddMessageEventHandler( Logger_Log );
  }

  private static void Logger_Log( object sender,
    LoggerEventArgs msg )
  {
    Console.Error.WriteLine( "{0}:\t{1}",
      msg.Priority.ToString(),
      msg.Message );
  }
}
```

Egy másik osztály a rendszernaplóba küldheti az üzeneteket:

```
class EventLogger
{
  private static string eventSource;
  private static EventLog logDest;

  static EventLogger()
  {
    logger.Log +=new AddMessageEventHandler( Event_Log );
  }

  public static string EventSource
  {
    get
    {
      return eventSource;
    }
```

```
      set
      {
        eventSource = value;
        if ( ! EventLog.SourceExists( eventSource ) )
          EventLog.CreateEventSource( eventSource,
            "ApplicationEventLogger" );

        if ( logDest != null )
          logDest.Dispose( );
        logDest = new EventLog( );
        logDest.Source = eventSource;
      }
    }

    private static void Event_Log( object sender,
      LoggerEventArgs msg )
    {
      if ( logDest != null )
        logDest.WriteEntry( msg.Message,
          EventLogEntryType.Information,
          msg.Priority );
    }
  }
```

Az események az összes érdekelt ügyfelet értesítik arról, hogy valami történt. A `Logger` osztálynak nem kell előre tudnia, hogy mely objektumokat érdekli az események naplózása.

A `Logger` osztály csak egyetlen eseményt tartalmazott. Vannak olyan osztályok (ez leginkább a windowsos vezérlőkre jellemző), amelyekben nagyon sok eseményt találunk. Ezekben az esetekben az eseményenként egy-egy mező használata elfogadhatatlan megközelítés lehet. Néhány esetben csak a meghatározott események kis része az, amit ténylegesen használunk egy adott alkalmazásban. Ha ilyen helyzettel találkozunk, akkor úgy módosíthatjuk a program szerkezetét, hogy az eseményobjektumokat csak szükség esetén, futásidőben hozzuk létre.

A keretrendszer magjában is találunk példát arra, hogy miként valósították meg ezt a Windows vezérlőkhöz tartozó alrendszerben. Hogy mi is lássuk, hogyan történik ez, adjunk alrendszereket a `Logger` osztályhoz. Hozzunk létre egy eseményt minden alrendszerhez. Az ügyfelek arra az eseményre figyelnek, amelyik az ő alrendszerükre vonatkozik.

A kibővített `Logger` osztálynak van egy `System.ComponentModel.EventHandlerList` tárolója, ami az összes olyan eseményobjektumot tárolja, amit egy adott rendszernél ki kell váltani. A módosított `AddMsg()` tagfüggvény immár egy karakterlánc paramétert kap, ami megadja a naplóüzenetet előállító alrendszert. Ha az alrendszernek vannak figyelői, akkor az esemény kiváltása megtörténik.

Ha pedig egy eseményfigyelő minden esemény figyelésére jelentkezett, akkor a hozzá tartozó esemény is kiváltódik:

```
public class Logger
{
  private static System.ComponentModel.EventHandlerList
    Handlers = new System.ComponentModel.EventHandlerList();

  static public void AddLogger(
    string system, AddMessageEventHandler ev )
  {
    Handlers[ system ] = ev;
  }

  static public void RemoveLogger( string system )
  {
    Handlers[ system ] = null;
  }

  static public void AddMsg ( string system,
    int priority,  string msg )
  {
    if ( ( system != null ) && ( system.Length > 0 ) )
    {
      AddMessageEventHandler l =
        Handlers[ system ] as AddMessageEventHandler;

      LoggerEventArgs args = new LoggerEventArgs(
        priority, msg );
      if ( l != null )
        l ( null, args );

      // Az üres karakterlánc az összes esemény fogadását jelenti:
      l = Handlers[ "" ] as AddMessageEventHandler;
      if ( l != null )
        l( null, args );
    }
  }
}
```

Az új példa az egyes eseménykezelőket az `EventHandlerList` gyűjteményben tárolja. Az ügyfélkód egy adott alrendszerhez csatlakozik, és létrejön egy új eseményobjektum. Az ugyanehhez az alrendszerhez érkező további kérelmek ugyanezt az eseményobjektumot hívják le. Ha kifejlesztünk egy olyan osztályt, amelynek a felülete sok eseményt tartalmaz, akkor érdemes fontolóra venni egy ilyen eseménykezelőkből álló gyűjtemény használatát. Az eseménytagokat akkor hozzuk létre, amikor az ügyfelek a kiválasztott eseménykezelőre

kapcsolódnak. A .NET keretrendszeren belül a `System.Windows.Forms.Control` osztály egy bonyolultabb változatát használja ennek a megvalósításnak, hogy elrejtse az eseménymezőinek összetettségét. Minden eseménymező belsőleg fér hozzá egy objektumgyűjteményhez, hogy beszúrja vagy törölje az adott eseménykezelőt. A C# nyelv leírásában még több helyen is találkozunk ezzel a mintával (lásd a 49. tippet).

Az osztályok kimenő felületét eseményekkel határozzuk meg. Akárhány ügyfél csatolhat kezelőket az eseményekhez, és fel is dolgozhatja azokat. Ezeket az ügyfeleket fordításkor nem kell feltétlenül ismernünk. Az eseményeknek nincs szükségük előfizetőkre ahhoz, hogy a rendszer működjön. A C# nyelvben az események használatakor elválasztjuk egymástól a küldőt és az üzenetek valószínűsíthető fogadóit. A küldőt a fogadóktól teljesen függetlenül is kifejleszthetjük. Az események segítségével szabványos módon terjeszthetünk információkat a típus végrehajtotta műveletekről.

23. tipp

Kerüljük a belső osztályobjektumokra mutató hivatkozások visszaadását

Az ember szeretné azt hinni, hogy a csak olvasható (read-only) tulajdonságok valóban csak olvashatók, és hogy a hívók nem képesek azokat módosítani. Sajnos ez nem mindig így működik. Ha egy olyan tulajdonságot hozunk létre, ami egy hivatkozási típust ad vissza, akkor a hívó fél az adott objektum bármely tagjához hozzáférhet, beleértve azokat is, amelyek módosíthatják a tulajdonság állapotát. Például:

```
public class MyBusinessObject
{
  // Csak olvasható tulajdonság, ami egy
  // privát adattaghoz biztosít hozzáférést:
  private DataSet _ds;
  public DataSet Data
  {
    get
    {
      return _ds;
    }
  }
}

// Az adathalmaz elérése:
DataSet ds = bizObj.Data;
// Nem így akartuk, de meg lehet tenni:
ds.Tables.Clear( ); // Az összes adattábla törlése
```

A MyBusinessObject bármelyik nyilvános ügyfele módosíthatja a belső adathalmazunkat. Azért hoztunk létre tulajdonságokat, hogy elrejtsük a belső adatszerkezeteket. Megadtunk tagfüggvényeket is, hogy az ügyfelek csak az ismert tagfüggvényeken keresztül dolgozhassanak az adatokkal, s így az osztályunk kezelhesse a belső állapotra vonatkozó változtatásokat. És ekkor egy csak olvasható tulajdonság tátongó rést nyit a szépen betokozott osztályon. Nem, nem egy írható-olvasható tulajdonságról van szó, ahol számításba vennénk ezt az eshetőséget, hanem egy csak olvasható tulajdonságról.

Üdvözöljük a hivatkozás alapú rendszerek csodálatos világában! Bármely tag, amely egy hivatkozási típust ad vissza, az adott objektumhoz tartozó leírót ad vissza. Átadtunk a hívónak egy leírót, ami a belső adatszerkezeteinkre mutat, tehát a hívónak már nem kell az objektumon keresztül haladnia, hogy módosítsa a tartalmazott hivatkozást.

Világos, hogy ezt a fajta viselkedést meg szeretnénk akadályozni. Készítettünk egy felületet az objektumunknak, és azt szeretnénk, hogy a felhasználók ezt használják. Nem akarjuk, hogy a felhasználók a tudtunkon kívül elérjék vagy módosítsák az objektumaink belső állapotát. Négy különböző eszköz közül választhatunk, amelyekkel megvédhetjük a belső adatszerkezeteinket a nem kívánt módosításoktól: ezek az értéktípusok, a nem változó típusok, a felületek és a burkolók.

Az értéktípusokról másolat készül, amikor az ügyfelek egy tulajdonságon keresztül érik el azokat. Az osztály ügyfelei által kiolvasott másolaton végrehajtott módosítások nem érintik az objektumunk belső állapotát. Az ügyfelek tetszés szerint módosíthatják a másolatot a céljaiknak megfelelően, de ez semmilyen hatással nincs az objektum belső állapotára.

A nem változó típusok, mint amilyen a System.String is, szintén biztonságosak. Karakterláncokat vagy bármilyen más nem változó típust is teljes biztonságban visszaadhatunk, tudva, hogy az osztály egyetlen ügyfele sem képes módosítani a karakterláncot. Az objektum belső állapota biztonságban van.

A harmadik lehetőség az, ha egy olyan felületet határozunk meg, amelyik csak a belső tagok működésének egy részéhez engedi hozzáférni az ügyfeleket (lásd a 19. tippet). Amikor saját osztályt készítünk, létrehozhatunk olyan felületegyütteseket, amelyek az osztály szolgáltatásainak csak bizonyos részhalmazát támogatják. Ha az osztály szolgáltatásait ilyen felületeken keresztül tesszük elérhetővé, azzal csökkentjük annak a valószínűségét, hogy a belső adatok valamilyen nem tervezett módon változzanak meg. Az ügyfelek az általunk megadott felületen keresztül érhetik el a belső objektumot, ami nem foglalja magában az objektum teljes működését. Az IListsource felület alkalmazása a DataSet osztálynál jó példa erre a megoldásra. Az igazán machiavellista programozók persze átverhetik a rendszert, ha kitalálják az objektum típusát, és végrehajtanak egy típusátalakítást. Azok a programozók viszont, akik nem sajnálnak ennyi időt és fáradságot egy programhiba előidézésére, csak azt kapják, amit megérdemelnek.

A `System.Dataset` osztály az utolsó eszközt használja: a burkoló objektumokat. A `DataViewManager` osztály alkalmat ad arra, hogy hozzáférjünk a `DataSet` objektumhoz, de megvédi a `DataSet` osztályon keresztül elérhető változtató tagfüggvényeket (mutator method):

```
public class MyBusinessObject
{
  // Csak olvasható tulajdonság, ami egy
  // privát adattaghoz biztosít hozzáférést:
  private DataSet _ds;
  public DataView this[ string tableName ]
  {
    get
    {
      return _ds.DefaultViewManager.
        CreateDataView( _ds.Tables[ tableName ] );
    }
  }
}

// Az adathalmaz elérése:
DataView list = bizObj[ "customers" ];
foreach ( DataRowView r in list )
  Console.WriteLine( r[ "name" ] );
```

A `DataViewManager` `DataView` objektumokat hoz létre, amivel elérhetők a `DataSet` objektum egyes adattáblái. Az osztályunk felhasználói a `DataViewManager` objektumon keresztül sehogy nem tudják módosítani a `DataSet` objektumban található táblákat. Minden egyes `DataView` objektumot be lehet állítani úgy, hogy az lehetővé tegye az egyes adatelemek módosítását. A táblákat és az adatoszlopokat azonban az ügyfél nem módosíthatja. Az alapértelmezés az olvasható–írható beállítás, tehát az ügyfelek továbbra is képesek beszúrni, módosítani, vagy törölni az egyes elemeket.

Mielőtt megbeszélnénk, hogyan lehet egy kizárólag olvasható nézetet készíteni az adatokhoz, vessünk egy gyors pillantást arra, hogy miként reagálhatunk az adatainkon végzett módosításokra, amikor megengedjük, hogy nyilvános ügyfelek megváltoztathassák azokat. Ez azért fontos, mert sokszor előfordul, hogy a felhasználói felület vezérlőiben akarjuk megjeleníteni a `DataView` objektumunkat, hogy a felhasználók módosíthassák az adatokat (lásd a 38. tippet). Minden bizonnyal volt már dolgunk a windowsos űrlapok adatkötésével, amivel lehetővé tettük a felhasználóknak, hogy módosítsák az objektumunk privát adatait. A `DataTable` osztály a `DataSet` osztályon belül eseményeket vált ki, amelyekkel könnyedén megvalósítható a Megfigyelő (observer) minta. Az osztályaink bármilyen módosításra válaszolhatnak, amit az osztály más ügyfelei hajtottak végre. A `DataSet` objektumon belül található `DataTable` objektumok eseményeket váltanak ki, amikor egy oszlop vagy egy sor megváltozik az adott táblában. A `ColumnChanging` és a `RowChanging` ese-

mények azelőtt következnek be, mielőtt a módosítások érvényre jutnának a DataTable objektumban. A ColumnChanged, illetve a RowChanged események a módosítások végrehajtása után következnek be.

Ezt a módszert minden olyan esetben használhatjuk, amikor azt szeretnénk elérni, hogy a belső adatelemeket nyilvános ügyfelek módosíthassák, de a módosításokat ellenőriznünk kell, és reagálnunk is kell rájuk. Az osztály feliratkozik azokra az eseményekre, amelyeket a belső adatszerkezet állít elő. Az eseménykezelők ellenőrzik a módosításokat, és más belső állapotok frissítésével reagálnak is rájuk.

Visszatérve az eredeti kérdéshez, most azt szeretnénk elérni, hogy az ügyfelek lássák az adatainkat, de ne módosíthassák azokat. Ha az adatainkat egy DataSet objektumban tároljuk, akkor ezt úgy érhetjük el, hogy létrehozunk egy DataView objektumot ahhoz a táblához, amelyik nem engedélyezi az adatok módosítását. A DataView osztályban olyan tulajdonságok vannak, amelyek lehetővé teszik, hogy testreszabjuk a beszúrási, törlési, módosító, de még a rendező műveleteket is a táblában. Egy indexelő létrehozásával visszaadhatunk egy DataView objektumot a kérdéses táblához az indexelő segítségével:

```
public class MyBusinessObject
{
  // Csak olvasható tulajdonság, ami egy
  // privát adattaghoz biztosít hozzáférést:
  private DataSet _ds;
  public IList this[ string tableName ]
  {
    get
    {
      DataView view =
        _ds.DefaultViewManager.CreateDataView
        ( _ds.Tables[ tableName ] );
      view.AllowNew = false;
      view.AllowDelete = false;
      view.AllowEdit = false;
      return view;
    }
  }
}

// Az adathalmaz elérése:
    IList dv = bizOjb[ "customers" ];
    foreach ( DataRowView r in dv )
      Console.WriteLine( r[ "name" ] );
```

Az osztálynak eme utolsó kivonata egy adott adattábla nézetét adja vissza az IList felületére mutató hivatkozáson keresztül. Az IList felületet bármely gyűjteménnyel használhatjuk, nemcsak a DataSet osztállyal. Nem elég azonban egyszerűen visszaadnunk

a DataView objektumot. A felhasználók könnyedén ismét engedélyezhetnék a módosításokat és a beszúrási, illetve törlési lehetőségeket. A visszaadott nézetet úgy állítottuk be, hogy az ne engedje meg a listában található objektumok módosítását. Az IList mutatójának visszaadásával azt érjük el, hogy az ügyfelek ne tudják módosítani azokat a jogosultságokat, amelyeket a DataView objektumhoz kaptak.

Ha az objektumunk nyilvános felületén hivatkozási típusokat teszünk elérhetővé, azzal lehetővé tesszük az objektum felhasználóinak, hogy anélkül módosítsák annak belsejét, hogy át kellene haladniuk az általunk biztosított tagfüggvényeken és tulajdonságokon. Ez teljesen logikátlan, ami egyben azt is jelenti, hogy gyakori hibáról van szó. Az osztály felületeinek módosításánál vegyük figyelembe, hogy hivatkozásokat adunk vissza és nem értékeket. Ha simán visszaadjuk a belső adatokat, akkor ezzel elérhetővé tesszük a belső tagokat. Az ügyfeleink a tagokban található bármely tagfüggvényt meghívhatják. Ezt a hozzáférési lehetőséget úgy korlátozhatjuk, ha a privát belső adatokat felületeken vagy burkoló objektumokon keresztül tesszük hozzáférhetővé. Amikor ténylegesen azt szeretnénk, hogy az ügyfelek módosíthassák a belső adatelemeket, akkor használjuk a Megfigyelő programtervezési mintát, hogy az objektumaink ellenőrizhessék a módosításokat és reagálhassanak rájuk.

24. tipp

Felszólító stílus helyett programozzunk kijelentő stílusban

A kijelentő stílusú programozás (deklaratív programozás) gyakran egyszerűbb, és tömörebb módszert kínál egy program viselkedésének leírására, mint a felszólító stílusú programozás (imperatív programozás). A **kijelentő** stílusú programozás azt jelenti, hogy a program viselkedését utasítások helyett bevezetésekkel (deklarációkkal) írjuk le. A C# nyelvben – sok más nyelvhez hasonlóan – a programozás jórész **felszólító** stílusú. Tagfüggvényeket írunk, amelyek meghatározzák a programunk viselkedését. Kijelentő stílusú programozást a C# nyelvben a jellemzők segítségével valósíthatunk meg. A jellemzőket osztályokhoz, tulajdonságokhoz, adattagokhoz vagy tagfüggvényekhez csatoljuk, a .NET futásidejű környezet pedig ehhez hozzáteszi a megfelelő viselkedéseket helyettünk. Ezt a fajta kijelentő stílusú megközelítést egyszerűbb megvalósítani, olvasni és karbantartani.

Elsőnek nézzünk meg egy kézenfekvő példát, amit már használtunk. Amikor megírtuk az első ASP.NET webszolgáltatásunkat, a varázsló ezt a kódot állította elő:

```
[WebMethod]
public string HelloWorld()
{
  return "Hello World";
}
```

A VS .NET Web Service varázsló hozzáadta a `[WebMethod]` jellemzőt a `HelloWorld()` tagfüggvényhez. Ezzel kijelentettük, hogy a `HelloWorld` egy webes tagfüggvény. Az ASP.NET futásidejű környezete ennek a jellemzőnek a megléte alapján előállít egy kódot. A futásidejű környezet létrehozta a Web Service Description Language (WSDL, webszolgáltatás-leíró nyelvi) dokumentumot, ami annak a SOAP dokumentumnak a leírását tartalmazza, amelyik meghívja a `HelloWorld` tagfüggvényt. Az ASP.NET futáskörnyezete azzal is támogatja a programot, hogy a SOAP kérelmeket a `HelloWorld` tagfüggvényünkhöz irányítja. Az ASP.NET futásidejű környezete ezen kívül dinamikusan HTML lapokat is létrehoz, amelyekkel kipróbálhatjuk az új webszolgáltatásunkat az Internet Explorerben. Ez mind-mind a `WebMethod` jellemző jelenlétének köszönhető. A jellemzővel kijelentettük, hogy mik a szándékaink, a futásidejű környezet pedig biztosította a megfelelő támogatást. A jellemzők használata sokkal kevesebb időt vesz igénybe, és ellenállóbbá teszi a programot a hibákkal szemben.

Varázslatról szó sincs. Az ASP.NET futásidejű környezet visszatekintéssel határozza meg, hogy az osztályunk mely tagfüggvényei webes tagfüggvények. Ha talál ilyeneket, akkor az ASP.NET futásidejű környezete bármely függvényből képes webes tagfüggvényt készíteni úgy, hogy hozzáadja a szükséges keretrendszeri kódot.

A `[WebMethod]` jellemző csak egy a .NET könyvtárban meghatározott számtalan jellemző közül, amelyeknek a segítségével gyorsabban készíthetünk helyesen működő programokat. Vannak olyan jellemzők, amelyek a sorosítható (rendezhető) típusok elkészítésében segítenek minket (lásd a 25. tippet). Ahogy azt a 4. tippnél láttuk, a feltételes fordítás menetét is jellemzők irányítják. Ezekben és más esetekben is, a kijelentő stílusú programozással gyorsabban készíthetjük el a szükséges kódot, és annak is kisebb lesz az esélye, hogy hibázunk. A szándékainkat mindig a .NET Framework jellemzőivel írjuk le, ahelyett, hogy saját kódot használnánk. Kevesebb időbe telik, könnyebb is, és a fordító sem hibázik.

Ha az előre meghatározott jellemzők nem felelnek meg az igényeinknek, akkor egyéni jellemzők és visszatekintés segítségével mi magunk is elkészíthetünk kijelentő stílusú programszerkezeteket. Létrehozhatunk például egy olyan jellemzőt és a hozzá tartozó kódot, ami lehetővé teszi a felhasználók számára, hogy olyan típusokat hozzanak létre, amelyek a jellemző segítségével meghatároznak egy alapértelmezett rendezési sorrendet. Mintaként bemutatjuk, hogy egy ilyen jellemző használatával miként lehet meghatározni, hogy milyen szempont szerint rendezzünk sorba egy vevőkből álló gyűjteményt:

```
[DefaultSort( "Name" )]
public class Customer
{
  public string Name
  {
    get { return _name; }
    set { _name = value; }
  }
```

```
  public decimal CurrentBalance
  {
    get { return _balance; }
  }

  public decimal AccountValue
  {
    get
    {
      return calculateValueOfAccount();
    }
  }
}
```

A `DefaultSort` jellemzőnél a `Name` tulajdonságot látjuk. Ebből az következik, hogy minden `Customer` objektumokat tartalmazó gyűjteményt a vevők neve szerint kell sorba rendezni. A `DefaultSort` jellemző nem része a .NET keretrendszernek. Megvalósításához létre kell hoznunk a `DefaultSortAttribute` osztályt:

```
[AttributeUsage( AttributeTargets.Class |
  AttributeTargets.Struct )]
public class DefaultSortAttribute : System.Attribute
{
  private string _name;
  public string Name
  {
    get { return _name; }
    set { _name = value; }
  }

  public DefaultSortAttribute( string name )
  {
    _name = name;
  }
}
```

Ezen kívül meg kell írnunk még azt a kódot is, ami sorba rendezi az objektumokat egy gyűjteményben a `DefaultSort` jellemző jelenlététől függően. A megfelelő tulajdonságot visszatekintéssel találjuk meg, majd ennek a tulajdonságnak az értékét hasonlítjuk össze két objektumnál. A jó hír az, hogy ezt a kódot csak egyszer kell megírnunk.

A következő lépésben létrehozunk egy osztályt, ami megvalósítja az `IComparer` felületet. (Erről a 26. tippben részletesen is szó lesz.) Az `IComparer` felületnek van egy `CompareTo()` függvénye, ami két adott típusú objektumot hasonlít össze, miközben megengedi, hogy a célosztály, vagyis amelyik megvalósítja az `IComparable` felületet, határozza meg a rendezési

sorrendet. Az általános összehasonlító az összehasonlított típusok alapján találja meg az alapértelmezett rendezési tulajdonság leíróját. A `Compare` tagfüggvény két valamilyen típusú objektumot rendez sorba az alapértelmezett rendezési tulajdonság alapján:

```
internal class GenericComparer : IComparer
{
  // Információ az alapértelmezett tulajdonságról:
  private readonly PropertyDescriptor _sortProp;

  // Növekvő vagy csökkenő sorrend
  private readonly bool _reverse = false;

  // Szerkezet egy típushoz
  public GenericComparer( Type t ) :
    this( t, false )
  {
  }

  // Szerkezet egy típushoz
  // és egy irányhoz
  public GenericComparer( Type t, bool reverse )
  {
    _reverse = reverse;
    // megkeressük a jellemzőt,
    // és a rendezési tulajdonság nevét:

    // Beolvassuk a típus alapértelmezett rendezési jellemzőit:
    object [] a = t.GetCustomAttributes(
      typeof( DefaultSortAttribute ),
      false );

    // Beolvassuk a tulajdonsághoz tartozó PropertyDescriptor-t:
    if ( a.Length > 0 )
    {
      DefaultSortAttribute sortName = a[ 0 ] as
        DefaultSortAttribute;
      string name = sortName.Name;

      // Beállítjuk a rendezési tulajdonságot:
      PropertyDescriptorCollection props =
        TypeDescriptor.GetProperties( t );
      if ( props.Count > 0 )
      {
        foreach ( PropertyDescriptor p in props )
        {
          if ( p.Name == name )
          {
            // Megvan az alapértelmezett rendezési tulajdonság:
```

```
              _sortProp = p;
              break;
            }
          }
        }
      }
    }

    // A Compare tagfüggvény
    int IComparer.Compare( object left,
      object right )
    {
      // a null kisebb, mint bármelyik valódi objektum:
      if (( left == null ) && ( right == null ))
        return 0;
      if ( left == null )
        return -1;
      if ( right == null )
        return 1;

      if ( _sortProp == null )
      {
        return 0;
      }

      // Beolvassuk mindegyik objektum rendezési tulajdonságát:
      IComparable lField =
        _sortProp.GetValue( left ) as IComparable;
      IComparable rField =
        _sortProp.GetValue( right ) as IComparable;
      int rVal = 0;
      if ( lField == null )
        if ( rField == null )
          return 0;
        else
          return -1;
      rVal = lField.CompareTo( rField );
      return ( _reverse ) ? -rVal : rVal;
    }
  }
```

A `Generic` összehasonlító bármely `Customer` objektumokból álló gyűjteményt a `DefaultSort` jellemzőben megadott tulajdonság alapján rendez:

```
CustomerList.Sort( new GenericComparer(
  typeof( Customer )));
```

A `GenericComparer` megvalósítását végző kód olyan fejlett eljárásokat használ, mint amilyen a visszatekintés (lásd a 43. tippet) – de csak egyszer kell megírnunk. Onnantól kezdve már csak a jellemzőt kell hozzáadnunk egy osztályhoz, és az általános összehason-

lító segítségével máris sorba rendezhetünk egy ilyen objektumokból álló gyűjteményt. Ha módosítjuk a `DefaultSort` jellemző paraméterét, azzal megváltoztathatjuk az osztály viselkedését. A többi kódon és algoritmuson már sehol nem kell változtatnunk.

A kijelentő stílusú minta segítségével elkerülhetjük a kódismétléseket, amikor egy egyszerű kijelentéssel megadhatjuk a szándékunkat. Nézzük meg ismét a `GenericComparer` osztályt. Megtehetnénk, hogy egy másfajta (és kissé egyszerűbb) rendező algoritmust készítünk minden egyes típusunkhoz. A kijelentő stílusú programozás előnye az, hogy csak egyszer kell megírnunk egy általános osztályt, aztán egy egyszerű bevezetéssel megadhatjuk minden típus viselkedését. A dolog kulcsa, hogy az osztály viselkedése egyetlen bevezetésen múlik, és nem az algoritmus megváltoztatásán. A `GenericComparer` minden olyan típussal működik, amit feldíszítünk a `DefaultSort` jellemzővel. Ha csak egyszer-kétszer van szükségünk rendezésre az alkalmazásunkban, akkor az egyszerűbb eljárásokat írjuk meg. Ha viszont tucatnyi különböző típusnál lesz szükségünk ugyanarra a viselkedésre a programban, akkor hosszú távon az általános algoritmus és a kijelentő stílus alkalmazásával sok időt és energiát takaríthatunk meg. A `WebMethod` jellemző hatására előállított összes kódot magunktól soha nem írnánk meg. Erre a módszerre a saját algoritmusainknál is építhetünk. A 42. tippnél egy példát is láthatunk erre, ahol arról lesz szó, hogy miként készíthetünk parancskezelő bővítményeket. A lehetséges további példák sorában egyébként szinte bármit megtalálhatunk a bővítőcsomagok meghatározásától kezdve a weblapok felhasználói felületét alkotó dinamikus elemekig.

A kijelentő stílusú programozás hatékony eszköz. Amikor jellemzőket használva is kifejezhetjük a szándékainkat, olyankor rengeteg logikai hibát kerülhetünk el, amelyek az egymáshoz hasonló, saját kezűleg megírt algoritmusokban jelentkeznének. A kijelentő stílusú programozás olvashatóbb, tisztább kódot eredményez. Ez kevesebb hibát jelent a program írásakor és a jövőben is. Ha lehetőségünk van egy jellemző használatára a .NET keretrendszerben, akkor éljünk vele. Ha ez a lehetőség nem adott, akkor fontoljuk meg egy saját jellemző elkészítését, amivel később már könnyen megvalósíthatjuk az adott viselkedést.

25. tipp

Használjunk sorosítható típusokat

Az állandóság a típusaink alapvető tulajdonsága. Azon dolgok közé tartozik, amiket senki nem vesz észre egészen addig, amíg meg nem feledkezünk róla. Ha a típusunk nem támogatja megfelelően a sorosíthatóságot (rendezhetőséget), akkor rengeteg munkát hárítunk azokra, akik tagként vagy alaposztályként akarják használni a típusainkat. Ha a típusaink nem támogatják a sorosíthatóságot, akkor a felhasználókat kerülőútra kényszerítjük, hiszen ilyenkor nekik kell megvalósítani egy szabványos szolgáltatást. Nem valószínű, hogy az ügyfelek helyesen meg tudják valósítani a típusunk sorosítását anélkül, hogy hozzáférnének annak privát részleteihez. Ha nem támogatjuk a sorosíthatóságot, akkor a felhasználó nehéz vagy éppen lehetetlen feladattal találja magát szemben, ha mégis szeretné sorosíthatóvá tenni a típust.

Ezt elkerülendő, mindig adjuk hozzá a sorosíthatóságot a típusainkhoz, amikor ez célszerűnek tűnik. Márpedig ez a felhasználói felületek elemeit, vezérlőket és ablakokat ábrázoló típusokon kívül minden esetben ajánlatos. Ez alól az sem ment fel minket, hogy ez látszólag többletmunkával jár. A .NET Serialization támogatás olyan egyszerű, hogy semmilyen ésszerű magyarázatot nem adhatunk arra, hogy miért ne használjuk. Sok esetben elég, ha az osztályunkhoz hozzáadjuk a `Serializable` jellemzőt:

```
[Serializable]
public class MyType
{
  private string _label;
  private int _value;
}
```

A `Serializable` jellemző megadása azért működik, mert ennek a típusnak az összes tagja sorosítható. A karakterláncok (`string`) és az egészek (`int`) is támogatják a .NET sorosítást. Az, hogy miért fontos a sorosíthatóság támogatása, egyből világossá válik, ha egy egyéni típusú újabb mezőt adunk az osztályhoz:

```
[Serializable]
public class MyType
{
  private string       _label;
  private int          _value;
  private OtherClass   _object;
}
```

A `Serializable` jellemző itt már csak akkor működik, ha az `OtherClass` típus támogatja a .NET sorosítást. Ha az `OtherClass` nem sorosítható, akkor futásidejű hibát kapunk, tehát meg kell írnunk azt a kódot, ami sorosíthatóvá teszi a `MyType` osztályt és a benne lévő `OtherClass` objektumot. Ez egyszerűen nem lehetséges anélkül, hogy részletesen ismernénk az `OtherClass` osztály belsejét.

A .NET sorosítás az objektumunk összes tagját a kimeneti adatfolyamba menti. A .NET sorosításért felelős kód ezen felül tetszőleges objektumgráfokat is támogat. A `serialize`, illetve a `deserialize` tagfüggvények akkor is csak egyszer mentenek, illetve fejtenek vissza minden objektumot, ha az objektumok körkörös hivatkozásokat tartalmaznak. A .NET Serialization Framework (.NET sorosító keretrendszer) a hivatkozások hálózatát akkor is képes újra létrehozni, amikor az objektumok hálózatát visszafejtjük. Bármilyen általunk készített objektumhálózat visszaállítása helyesen történik meg az objektumgráf visszafejtésekor. Egy utolsó fontos megjegyzés, hogy a `Serializable` jellemző a bináris és a SOAP sorosítást is támogatja. Az ebben a tippben tárgyalt megoldások mindkét sorosítási formátumot támogatják, de ne felejtsük el, hogy mindez csak akkor működik, ha az objektumgráf összes típusa támogatja a sorosítást. Ezért fontos, hogy minden típusunk-

nál támogassuk a sorosítást. Amint kihagyunk egy osztályt, rést hozunk létre az objektumgráfon, ami az osztályunkat használók számára megnehezíti a sorosítás egyszerű támogatását. Nem telik bele sok idő, és mindenki szépen újraírhatja a saját sorosító kódját.

A `Serializable` jellemző megadása a legegyszerűbb módszer az objektumok sorosíthatóságának támogatására – de nem mindig a legegyszerűbb megoldás a helyes. Van úgy, hogy nem akarjuk sorosítani az objektum összes tagját, hiszen lehet, hogy néhány tagot csak egy hosszabb művelet eredményeinek átmeneti tárolására hoztunk létre. Lehetnek olyan tagok is, amelyek futásidőben olyan erőforrásokat tárolnak, amelyeket kizárólag a memóriában végzett műveletekhez használunk. Ezeket az eshetőségeket is kezelhetjük jellemzőkkel. Annyit kell tennünk, hogy hozzácsatoljuk a `[NonSerialized]` jellemzőt azokhoz az adattagokhoz, amelyeket nem kell menteni az objektum állapotának részeként. Ezzel nem sorosítható jellemzőkké válnak:

```
[Serializable]
public class MyType
{
  private string _label;

  [NonSerialized]
  private int _cachedValue;

  private OtherClass  _object;
}
```

A nem sorosított tagok kicsit több munkát jelentenek számunkra, vagyis az osztály tervezőjének számára. A sorosító API-k nem állítják be a nem sorosított tagokat a visszafejtési folyamat során. A típusaink konstruktorait nem hívja meg a program, így a tagok beállítófüggvényeit sem hajtja végre. Amikor a sorosíthatóságot jelző jellemzőket használjuk, akkor a nem sorosított tagok az alapértelmezett kezdőértéket kapják a rendszertől, ami `0` vagy `null` lesz. Ha a nullás kezdőérték nem helyes, akkor el kell készítenünk az `IDeserializationCallback` felület megvalósítását, ezen nem sorosított tagok kezdőértékeinek beállításához. Az `IDeserializationCallback` felület egyetlen tagfüggvényt tartalmaz: ez az `OnDeserialization`. A keretrendszer a teljes objektumgráf visszafejtése után hívja meg ezt a tagfüggvényt. Ezt a tagfüggvényt az objektum nem sorosított tagjainak a kezdőértékének a beállításához hívjuk meg, ha vannak ilyenek. Mivel a teljes objektumgráfot beolvastuk, biztosak lehetünk benne, hogy bármilyen függvényt meghívhatunk az objektumon és annak nem sorosított tagjain. Sajnos azonban ez sem „bolondbiztos" megoldás. Miután beolvasta a teljes objektumgráfot, a keretrendszer a gráfban található összes olyan objektumra meghívja az `OnDeserialization` tagfüggvényt, amely támogatja az `IDeserializationCallback` felületet. Az `OnDeserialization` végrehajtása közben az objektumgráfban található bármely másik objektum is meghívhatja az objektumunk nyilvános tagjait. Ha ezek hajtódnak végre elsőként, akkor az objektum nem sorosított tag-

jai null vagy 0 értéket kapnak. A sorrend nem garantált, tehát gondoskodnunk kell róla, hogy az összes nyilvános tagfüggvényünk kezelje azt az esetet, amikor a nem sorosított tagok kezdőértékeinek beállítása még nem történt meg.

Eddig arról volt szó, hogy miért érdemes minden típushoz hozzáadni a sorosíthatóságot. A nem sorosított típusokkal több munka van, amikor olyan típusokban használjuk őket, amelyeknek sorosítottnak kellene lenniük. Megismerkedtünk a jellemzők használatával elérhető legegyszerűbb sorosítási módszerrel, és a nem sorosított tagok kezdőértékeinek beállításával is.

A sorosított adatok tovább élnek a programunk következő változataiban is. A sorosítás megadása azt jelenti, hogy eljön a nap, amikor egy régebbi változatot is be kell olvasnia a programnak. A Serializable jellemző által előállított kód kivételt vált ki, amikor olyan mezőket talál, amelyeket hozzáadtak vagy eltávolítottak az objektumgráfból. Ha úgy érezzük, hogy készen állunk az eltérő változatok támogatására, és szeretnénk még inkább átvenni az irányítást a sorosítási folyamat felett, akkor használjuk az ISerializable felületet. Ez a felület határozza meg azokat a horgokat, amelyekkel testreszabhatjuk a típusaink sorosítását. Az ISerializable felület által használt tagfüggvények és tárolók összhangban vannak azokkal a tagfüggvényekkel és tárolókkal, amelyeket az alapértelmezett sorosító tagfüggvények használnak. Ez azt jelenti, hogy egy osztály létrehozásakor használhatjuk a sorosító jellemzőket. Ha egyszer szükségünk lenne a saját bővítményeink megadására, akkor adjuk hozzá az ISerializable felület támogatását a programhoz.

Példaként vegyük a MyType 2-es változatának támogatását, miután hozzáadunk egy újabb mezőt a típushoz. Azzal az egyszerű módosítással, hogy hozzáadunk a típushoz egy új mezőt, máris egy olyan új formátumot hoztunk létre, ami nem csereszabatos a lemezen tárolt korábbi változatokkal:

```
[Serializable]
public class MyType
{
  private string _label;

  [NonSerialized]
  private int _value;

  private OtherClass  _object;

  // A 2-es változatban adtuk az osztályhoz
  // A futásidejű környezet egy kivételt vált ki,
  // mivel hiányolja ezt a mezőt az 1.0-s fájlokból
  private int _value2;
}
```

Ezt a viselkedést az `ISerializable` felület támogatásával kezelhetjük. Az `ISerializable` felület egyetlen tagfüggvényt határoz meg, de nekünk kettőt kell megvalósítanunk. Az `ISerializable` meghatározza a `GetObjectData()` tagfüggvényt, ami adatokat ír egy adatfolyamba. Ehhez még meg kell adnunk egy sorosító konstruktort, ami beállítja az adatfolyamból származó objektumot:

```
private MyType( SerializationInfo info,
  StreamingContext cntxt );
```

Az alábbi osztályban található sorosító konstruktor azt szemlélteti, hogy miként kell következetesen beolvasni a típus korábbi változatát és a jelenlegi változatát úgy, hogy az alapértelmezett megvalósítást a `Serializable` jellemző megadása állítja elő:

```
using System.Runtime.Serialization;
using System.Security.Permissions;

[Serializable]
public sealed class MyType : ISerializable
{
  private string _label;

  [NonSerialized]
  private int _value;

  private OtherClass  _object;

  private const int DEFAULT_VALUE = 5;
  private int  _value2;

  // nyilvános konstruktorok elhagyva

  // Privát konstruktor, amit csak a Serialization
  // keretrendszer használ
  private MyType( SerializationInfo info,
    StreamingContext cntxt )
  {
    _label = info.GetString( "_label" );
    _object = ( OtherClass )info.GetValue( "_object", typeof
      ( OtherClass ));
    try {
      _value2 = info.GetInt32( "_value2" );
    } catch ( SerializationException e )
    {
      // Megvan az 1. változat
      _value2 = DEFAULT_VALUE;
    }
  }
```

```
    [SecurityPermissionAttribute(SecurityAction.Demand,
      SerializationFormatter =true)]
    void ISerializable.GetObjectData (SerializationInfo inf,
      StreamingContext cxt)
    {
      inf.AddValue( "_label", _label );
      inf.AddValue( "_object", _object );
      inf.AddValue( "_value2", _value2 );
    }
}
```

A sorosító adatfolyam minden elemet kulcs–érték párként tárol. A jellemzők segítségével előállított kód a változók neveit használja kulcsként minden értékhez. Amikor megadjuk az `ISerializable` felületet, akkor igazodnunk kell a kulcsnevekhez és a változók sorrendjéhez. A sorrend megegyezik az osztályban lévő bevezetési sorrenddel. (Jut eszembe, ez egyben azt is jelenti, hogy ha átrendezzük a változók sorrendjét egy osztályban vagy változókat nevezünk át, azzal tönkretesszük a már létező fájloknak való megfelelőséget.)

Megköveteltük továbbá a `SerializationFormatter` jogosultságot. A `GetObjectData` biztonsági rést jelenthet, amelyen keresztül be lehet jutni az osztályba, ha nem biztosítjuk a megfelelő védelmet. Egy rosszindulatú kód létrehozhat egy `StreamingContext` objektumot, a `GetObjectData` segítségével kiolvashatja az objektum értékeit, átrendezheti a módosított változatot egy másik `SerializationInfo` objektumba, majd újból összerakhatná az immár módosított objektumot. Egy rosszindulatú támadó számára lehetővé válna, hogy megkaparintsa az objektumunk belső állapotát, módosítsa azt egy adatfolyamban, majd a módosításokat visszaküldje hozzánk. A `SerializationFormatter` jogosultság megkövetelésével betömjük ezt a fenyegető rést. Ezzel biztosítjuk, hogy csak megbízható forrásból származó kód használhassa ezt a tagfüggvényt az objektum belső állapotához való hozzáféréshez (lásd a 47. tippet).

Van azonban egy árnyoldala is az `ISerializable` felület megvalósításának. Láttuk, hogyan zártuk le korábban a `MyType` típust. Ennek következményeképpen levélosztály lesz belőle. Az `ISerializable` felület megvalósítása egy alaposztályban bonyolultabbá teszi a sorosítást az összes származtatott osztálynál. Az `ISerializable` megvalósítása azt jelenti, hogy minden leszármazott osztályban létre kell hozni a védett konstruktort a visszafejtéshez. A nyitott osztályok támogatásához pedig horgokat kell létrehoznunk a `GetObjectData` tagfüggvényben a leszármazott osztályokhoz, hogy azok is hozzáadhassák a saját adataikat az adatfolyamhoz. A fordító egyik hibát sem csípi el. A megfelelő konstruktor hiányának hatására a futásidejű környezet kivételt vált ki, amikor egy származtatott objektumot olvas be az adatfolyamból. A `GetObjectData()` részére készült horog hiánya azt jelenti, hogy az objektum származtatott részében található adatokat a program nem menti a fájlba. Hibát nem kapunk. A tanács lehetne az is, hogy „a levélosztályokban valósítsuk meg a `Serializable` felületet". De azért nem ezt javasoltam, mert ez nem működne. Ahhoz, hogy a származtatott osztály sorosítható legyen, az alaposztálynak is annak

kell lennie. Ha úgy akarjuk módosítani a `MyType` típust, hogy abból sorosítható alaposztály legyen, a sorosítható konstruktort `protected` típusúra kell változtatnunk, és létre kell hoznunk egy virtuális tagfüggvényt, amit a származtatott osztályok felülbírálhatnak az adataik mentéséhez:

```
using System.Runtime.Serialization;
using System.Security.Permissions;

[Serializable]
public class MyType : ISerializable
{
  private string _label;

  [NonSerialized]
  private int _value;

  private OtherClass  _object;

  private const int DEFAULT_VALUE = 5;
  private int  _value2;

  // nyilvános konstruktorok elhagyva

  // Védett konstruktor, amit csak a Serialization keretrendszer használ
  protected MyType( SerializationInfo info,
    StreamingContext cntxt )
  {
    _label = info.GetString( "_label" );
    _object = ( OtherClass )info.GetValue( "_object", typeof
      ( OtherClass ));
    try {
      _value2 = info.GetInt32( "_value2" );
    } catch ( SerializationException e )
    {
      // Megvan az 1. változat
      _value2 = DEFAULT_VALUE;
    }
  }

  [ SecurityPermissionAttribute( SecurityAction.Demand,
    SerializationFormatter =true ) ]
  void ISerializable.GetObjectData(
    SerializationInfo inf,
    StreamingContext cxt )
  {
    inf.AddValue( "_label", _label );
    inf.AddValue( "_object", _object );
    inf.AddValue( "_value2", _value2 );
```

```
    WriteObjectData( inf, cxt );
  }

  // A származtatott osztályokban felülbíráljuk, hogy
  // kiírhassuk az osztály származtatott részének adatait
  protected virtual void
    WriteObjectData(
    SerializationInfo inf,
    StreamingContext cxt )
  {
  }
}
```

A származtatott osztályokban saját sorosító konstruktorokat adunk meg, és felülbíráljuk a `WriteObjectData` tagfüggvényt:

```
public class DerivedType : MyType
{
  private int _DerivedVal;

  private DerivedType ( SerializationInfo info,
    StreamingContext cntxt ) :
      base( info, cntxt )
  {
      _DerivedVal = info.GetInt32( "_DerivedVal" );
  }

  protected override void WriteObjectData(
    SerializationInfo inf,
    StreamingContext cxt )
  {
    inf.AddValue( "_DerivedVal", _DerivedVal );
  }

}
```

Az értékek írásának és olvasásának sorrendje következetes kell hogy legyen az adatfolyamon. Az előbb azt választottuk, hogy előbb az alaposztály értékeit olvassuk és írjuk, mert így egyszerűbb. Ha az írást és olvasást végző kód nem ugyanabban a sorrendben sorosítja a teljes hierarchiát, akkor a sorosító kódunk nem fog működni.

A .NET Framework egy egyszerű, szabványos algoritmust kínál az objektumaink sorosításához. Ha a típus állandósága fontos, akkor kövessük a szabványos megvalósítást. Ha a típusainkban nem támogatjuk a sorosítást, akkor a típusainkat használó osztályok sem tudják azt támogatni. Amennyire lehet, mindig könnyítsük meg az osztályaink ügyfeleinek dolgát. Ha tehetjük, mindig használjuk az alapértelmezett tagfüggvényeket. Ha az alapértelmezett jellemzők kevésnek bizonyulnak, készítsük el az `ISerializable` felület megvalósítását.

26. tipp

A rendező relációkat az IComparable és az IComparer segítségével valósítsuk meg

A típusoknak rendező relációkra van szükségük annak leírásához, hogy a gyűjteményekben hogyan rendezzük és keressük őket. A :NET keretrendszer két felületet határoz meg, amelyek a típusok rendező relációit írják le: ezek az `IComparable` és az `IComparer`. Az `IComparable` a típusaink természetes sorrendjét adja meg. Az `IComparer` felületet csak akkor valósítja meg egy típus, ha a természetestől eltérő rendezési sorrendre van szüksége. Az összehasonlító műveletek (<, >, <=, >=) saját megvalósítását is elkészíthetjük a típusfüggő összehasonlításokhoz, elkerülve ezzel a felületek megvalósításából adódó futásidejű teljesítménycsökkenést. Ebben a tippben arról lesz szó, hogy miként kell megvalósítani a rendező relációkat úgy, hogy a .NET keretrendszer magja a meghatározott felületeken keresztül rendezze a típusokat, és hogy a többi felhasználó a lehető legjobb teljesítményt kapja ezektől a műveletektől.

Az `IComparable` felület egyetlen tagfüggvényt tartalmaz: a `CompareTo()` függvényt. Ez a tagfüggvény a C nyelv `strcmp` könyvtári függvénye által megteremtett régi hagyományt követi. Visszatérési értéke kisebb, mint 0, ha az adott objektum kisebb, mint az objektum, amivel összehasonlítottuk. Ha egyenlők, akkor a visszatérési érték 0. Ha az objektum nagyobb, mint az objektum, amivel összehasonlítottuk, akkor a visszatérési érték nagyobb lesz, mint 0. A `CompareTo()` tagfüggvény `System.Object` típusú paramétereket vár. Az átadott paraméter típusát futásidőben ellenőriznünk kell. Minden egyes összehasonlítás alkalmával ismételten értelmeznünk kell a paraméter típusát:

```
public struct Customer : IComparable
{
  private readonly string _name;

  public Customer( string name )
  {
    _name = name;
  }

  #region IComparable Members
  public int CompareTo( object right )
  {
    if ( ! ( right is Customer ) )
      throw new ArgumentException( "Argument not a customer",
        "right" );
    Customer rightCustomer = ( Customer )right;
    return _name.CompareTo( rightCustomer._name );
  }
  #endregion
}
```

Sok ellenszenves dolog van az `IComparable` felülettel összhangban lévő összehasonlítások megvalósításával kapcsolatban. Ellenőriznünk kell a paraméter futásidejű típusát. Egy helytelen kódrészlet akár szabályosan is meghívhatja ezt a függvényt bármilyen paramétert megadva a `CompareTo` tagfüggvénynek. Annál is inkább, mert a helyes paramétereket be- és ki kell csomagolni a tényleges összehasonlításhoz. Ez futásidőben többletmunkával jár minden egyes összehasonlításnál. Egy gyűjtemény rendezése átlagosan N x log(n) összehasonlítást végez az objektumon az `IComparable` felületen keresztül. Ezek mindegyike három be- és három kicsomagolási művelettel jár. Egy 1000 pontból álló tömbnél ez átlagosan több mint 20 000 be- és kicsomagolási műveletet jelent: az N x log(n) itt majdnem 7000, és összehasonlításonként 3 be- és kicsomagolási művelettel kell számolnunk. Ennél jobb megoldást kell találnunk. Az `IComparable.CompareTo()` meghatározásán nem változtathatunk. Ez azonban nem jelenti azt, hogy rá vagyunk kényszerítve arra, hogy elviseljük a gyengén típusos megvalósításból eredő, a felhasználókat terhelő teljesítményveszteséget. Elkészíthetjük a `CompareTo` tagfüggvény saját felülbírált változatát, ami már egy `Customer` típusú objektumot vár paraméterként:

```
public struct Customer : IComparable
{
  private string _name;

  public Customer( string name )
  {
    _name = name;
  }

  #region IComparable Members
  // IComparable.CompareTo()
  // Ez nem típusbiztos. A jobb oldali paraméter
  // futásidejű típusát ellenőrizni kell.
  int IComparable.CompareTo( object right )
  {
    if ( ! ( right is Customer ) )
      throw new ArgumentException( "Argument not a customer",
        "right" );
    Customer rightCustomer = ( Customer )right;
    return CompareTo( rightCustomer );
  }

  // típusbiztos CompareTo
  // A jobb oldal Customer típusú vagy annak leszármazottja
  public int CompareTo( Customer right )
  {
    return _name.CompareTo( right._name );
  }

  #endregion
}
```

Az `IComparable.CompareTo()` immár egy egyértelmű felületmegvalósítás, meghívása csak egy `IComparable` hivatkozáson keresztül lehetséges. A `Customer` struktúránk felhasználói a típusbiztos összehasonlítást kapják, a nem biztonságos összehasonlítás pedig nem lesz elérhető. Az alábbi ártatlan hibát már nem lehet lefordítani:

```
Customer c1;
Employee e1;
if ( c1.CompareTo( e1 ) > 0 )
  Console.WriteLine( "Customer one is greater" );
```

A fordítás azért nem lehetséges, mert a nyilvános `Customer.CompareTo(Customer right)` tagfüggvény paramétere hibás. Az `IComparable.CompareTo(object right)` tagfüggvény nem elérhető. Az `IComparable` tagfüggvényét csak a hivatkozás egyértelmű átalakításával érhetjük el:

```
Customer c1;
Employee e1;
if ( ( c1 as IComparable ).CompareTo( e1 ) > 0 )
  Console.WriteLine( "Customer one is greater" );
```

Ha megvalósítjuk az `IComparable` felületet, akkor mindig használjunk egyértelmű felületmegvalósításokat, és erősen típusos nyilvános túlterhelést adjunk meg. Az erősen típusos túlterhelés javítja a teljesítményt és csökkenti annak az esélyét, hogy valaki nem arra használja a `CompareTo` függvényt, amire való. A .NET környezet `Sort` függvényénél nem fogjuk tapasztalni mindezeket az előnyöket, mert az továbbra is eléri a `CompareTo()` tagfüggvényt a felületmutatón keresztül (lásd a 19. tippet), de a kód, ami mindkét összehasonlított objektum típusát ismeri, jobb teljesítményhez vezet.

Még egy utolsó apró változtatást elvégzünk a `Customer` struktúrán. A C# nyelv lehetővé teszi, hogy túlterheljük a szabványos összehasonlító műveleteket. Úgy jó, ha ezek is a típusbiztos `CompareTo()` tagfüggvényt használják:

```
public struct Customer : IComparable
{
  private string _name;

  public Customer( string name )
  {
    _name = name;
  }

  #region IComparable Members
  // IComparable.CompareTo()
  // Ez nem típusbiztos. A jobb oldali paraméter
  // futásidejű típusát ellenőrizni kell.
```

```
    int IComparable.CompareTo( object right )
    {
      if ( ! ( right is Customer ) )
        throw new ArgumentException( "Argument not a customer",
          "right");
      Customer rightCustomer = ( Customer )right;
      return CompareTo( rightCustomer );
    }

    // típusbiztos CompareTo
    // A jobb oldal Customer típusú vagy annak leszármazottja
    public int CompareTo( Customer right )
    {
      return _name.CompareTo( right._name );
    }

    // Összehasonlító műveletek
    public static bool operator < ( Customer left,
      Customer right )
    {
      return left.CompareTo( right ) < 0;
    }
    public static bool operator <=( Customer left,
      Customer right )
    {
      return left.CompareTo( right ) <= 0;
    }
    public static bool operator >( Customer left,
      Customer right )
    {
      return left.CompareTo( right ) > 0;
    }
    public static bool operator >=( Customer left,
      Customer right )
    {
      return left.CompareTo( right ) >= 0;
    }
    #endregion
  }
```

Ez minden, ami a vevők szabványos sorrendjét illeti, név szerint. Mert később majd készítenünk kell egy jelentést, amiben jövedelem szerint kell rendeznünk a vevőket. A `Customer` struktúra által meghatározott szokásos összehasonlítási műveletekre továbbra is szükségünk lesz, vagyis a név szerinti rendezés is kell. Az újabb rendezési követelménynek úgy felelhetünk meg, ha létrehozunk egy osztályt, ami megvalósítja az `IComparer` felületet. Az `IComparer` szabványos módszert kínál egy adott típus többféle szempont szerinti rendezéséhez. A .NET FCL összes olyan tagfüggvénye, ami `IComparable` típusokkal dolgozik, olyan túlterheléseket kínál, amelyek az `IComparer` felületen keresztül rendezik

sorba az objektumokat. Mivel mi vagyunk a `Customer` struktúra szerzői, ezt az új `RevenueComparer` nevű osztályt privát beágyazott osztályként hozhatjuk létre a `Customer` struktúrán belül. A külvilág számára a `Customer` struktúra egy statikus tulajdonságán keresztül tesszük elérhetővé:

```
public struct Customer : IComparable
{
  private string _name;
  private double _revenue;

  // a fenti példa kódját elhagytuk

  private static RevenueComparer _revComp = null;

  // visszaadunk egy IComparer felületet megvalósító objektumot
  // lusta kiértékeléssel hozunk létre egy ilyet
  public static IComparer RevenueCompare
  {
    get
    {
      if ( _revComp == null )
        _revComp = new RevenueComparer();
      return _revComp;
    }
  }

  // A vevőket jövedelmük alapján összehasonlító osztály
  // Ezt mindig a felületmutatón keresztül használjuk,
  // így csak a felület felülbírálását kell megadnunk
  private class RevenueComparer : IComparer
  {
    #region IComparer Members
    int IComparer.Compare( object left, object right )
    {
      if ( ! ( left is Customer ) )
        throw new ArgumentException(
          "Argument is not a Customer",
          "left");
      if (! ( right is Customer) )
        throw new ArgumentException(
          "Argument is not a Customer",
          "right");
      Customer leftCustomer = ( Customer ) left;
      Customer rightCustomer = ( Customer ) right;

      return leftCustomer._revenue.CompareTo(
        rightCustomer._revenue);
    }
    #endregion
  }
}
```

A `Customer` struktúra utolsó változata, amelyikben már ott van a `RevenueComparer`, lehetővé teszi, hogy egy vevőkből álló gyűjteményt név szerint rendezzünk, ami egyben a természetes sorrend, illetve egy másik rendezési sorrendet is kínál egy olyan osztály elérhetővé tételével, amelyik megvalósítja az `IComparer` felületet a vevők jövedelem szerinti rendezéséhez. Ha nem elérhető számunkra a `Customer` osztály forráskódja, akkor arra is lehetőségünk van, hogy egy olyan `IComparer` felületet adjunk meg, ami az egyik nyilvános tulajdonságának alapján rendezi sorba a vevőket. Ezt a mintát csak akkor kövessük, ha tényleg nem érhető el az osztály forráskódja, mint például abban az esetben, amikor a .NET Framework egy osztályánál van szükségünk az alapértelmezettől eltérő rendezési elvre.

Ebben a tippben sehol nem ejtettem szót sem az `Equals()`, sem a == műveletről (lásd a 9. tippet). A rendező relációk és az egyenlőség két külön művelethez tartoznak. Egy rendező relációhoz nem kell megvalósítanunk az egyenlőség összehasonlítását. Ez olyannyira igaz, hogy a hivatkozási típusok többnyire az objektumok tartalmára támaszkodva valósítják meg a rendezéseket, az egyenlőség megállapításához viszont az objektumazonosságot használják. A `CompareTo()` 0 értéket adhat vissza akkor is, amikor az `Equals()` hamis értékkel tér vissza. Ez teljesen szabályos. Az egyenlőség és a rendező relációk nem feltétlenül esnek egybe.

Az `IComparable` és az `IComparer` felületek szabványos eszközt kínálnak a típusaink rendező relációinak megadásához. Az `IComparable` felületet a legtermészetesebb sorrend megadására használjuk. Ha megvalósítjuk az `IComparable` felületet, akkor mindig terheljük túl az összehasonlító műveleteket (<, >, <=, >=) az `IComparable` rendezésnek megfelelően. Az `IComparable.CompareTo()` tagfüggvény `System.Object` paramétereket használ, ezért mindig adjuk meg a `CompareTo()` tagfüggvény típusfüggő túlterhelt változatát. Az `IComparer` felület további rendezési sorrendek megadására használható, illetve olyankor, amikor egy típus nem biztosítja számunkra a rendezési lehetőséget.

27. tipp

Kerüljük az ICloneable felületet

Az `ICloneable` felület elsőre nem is hangzik rossz ötletnek. Ezt a felületet olyan osztályokban valósítjuk meg, amelyek támogatják a másolatok készítését. Ha nem akarjuk támogatni a másolatokat, akkor ne valósítsuk meg. A típusunk azonban nem légüres térben létezik. Az a döntésünk, hogy támogatjuk-e az `ICloneable` felületet, kihat a származtatott osztályokra is. Ha egy típus támogatja az `ICloneable` felületet, akkor a belőle származtatott típusoknak is meg kell ezt tenniük. Minden tagtípusának is támogatnia kell az `ICloneable` felületet, vagy rendelkeznie kell valamilyen más megoldással a másolatok létrehozásához. Végül, amikor objektumhálózatokat tartalmazó programszerkezetekkel dolgozunk, a mély másolatok elkészítése elég körülményes. Az `ICloneable` ezt a problémát a hivatalos meghatározásában cselesen kikerüli: vagy mély vagy sekély másolatokat támogat. A sekély másolatnál új objektum készül, ami az összes tagváltozó másolatát tartal-

mazza. Ha a tagváltozók hivatkozási típusok, az új objektum ugyanarra az objektumra *hivatkozik*, mint az eredeti. A mély másolatnál egy olyan új objektum jön létre, amelyik az összes tagváltozót lemásolja. Az összes hivatkozási típus önhívással előállított pontos másolata (klónja) benne lesz a másolatban. A beépített típusoknál a mély, illetve a sekély másolatok készítése egyforma eredményhez vezet. Melyiknél van típustámogatás? Ez az adott típustól függ. Ha viszont egy objektumon belül keverjük a sekély és a mély másolatokat, az elég sok következetlenséghez vezet. Ha rátévedünk az `ICloneable` ingoványos területeire, akkor nehezen ússzuk meg szárazon. Legtöbbször úgy jutunk egyszerűbb osztályokhoz, ha teljes egészében elkerüljük az `ICloneable` használatát. Az eredmény egy egyszerűbben használható és könnyebben megvalósítható osztály lesz.

Azoknak az értéktípusoknak felesleges támogatniuk az `ICloneable` felületet, amelyeknek tagjai kizárólag beépített típusok. Egy egyszerű értékadás sokkal hatékonyabban másolja le a struktúra értékeit, mint a `Clone()` tagfüggvény. A `Clone()` függvénynek be kell csomagolnia a visszatérési értékét, hogy belegyömöszölhesse egy `System.Object` hivatkozásba. A hívónak egy újabb típusátalakítást kell elvégeznie, hogy kiolvashassa a csomagban lévő értéket. Biztos van ennél jobb dolgunk is, ezért ne írjunk olyan `Clone()` függvényeket, amelyek nem tesznek egyebet, mint amire az értékadás való.

És mi a helyzet azokkal az értéktípusokkal, amelyekben hivatkozási típusok is vannak? Erre a legkézenfekvőbb példa egy olyan értéktípus, amiben egy karakterlánc is van:

```
public struct ErrorMessage
{
  private int errCode;
  private int details;
  private string msg;

 // részletek elhagyva
}
```

A `string` típus különleges eset, mert nem változó osztály. Ha hozzárendelünk egy értéket egy hibaüzenet objektumhoz, akkor mindkét hibaüzenet objektum ugyanarra a karakterláncra fog hivatkozni. Ez egyetlen olyan problémához sem vezet, mint egy általános hivatkozási típus esetében. Ha bármelyik hivatkozáson keresztül módosítjuk az `msg` változót, akkor egy új `string` objektumot hozunk létre (lásd a 7. tippet).

Az általános eset, amikor olyan struktúrát hozunk létre, amelyben tetszőleges hivatkozási típus van, már sokkal bonyolultabb, de sokkal ritkábban is fordul elő. A struktúra beépített értékadása során sekély másolat készül, aminek következtében mindkét struktúra ugyanarra az objektumra fog hivatkozni. Egy mély másolat létrehozásához pontos másolatot (klónt) kell készítenünk a struktúrában lévő hivatkozási típusról, és biztosnak kell lennünk benne, hogy a hivatkozási típus támogatja a mély másolat készítését a `Clone()` tagfüggvényével. Bárhogy is legyen, az értéktípusoknál ne támogassuk az `ICloneable` felületet. Az értékadó művelet bármely értéktípusból új másolatot készít.

Az értéktípusokkal ezzel meg is volnánk. Nincs olyan indok, ami alapján valaha is támogatnunk kellene az ICloneable felületet az értéktípusokban. Menjünk most tovább a hivatkozási típusokra. A hivatkozási típusoknak támogatniuk kell az ICloneable felületet, jelezve ezzel, hogy a sekély vagy a mély másolatokat támogatják. Az ICloneable-támogatással azonban körültekintően kell eljárnunk, mivel ha megtesszük, akkor a típusból származtatott összes osztályban is kötelezővé válik az ICloneable támogatása. Vegyük az alábbi egyszerű hierarchiát:

```
class BaseType : ICloneable
{
  private string _label = "class name";
  private int [] _values = new int [ 10 ];

  public object Clone()
  {
    BaseType rVal = new BaseType( );
    rVal._label = _label;
    for( int i = 0; i < _values.Length; i++ )
      rVal._values[ i ] = _values[ i ];
    return rVal;
  }
}

class Derived : BaseType
{
  private double [] _dValues = new double[ 10 ];

  static void Main( string[] args )
  {
    Derived d = new Derived();
    Derived d2 = d.Clone() as Derived;

    if ( d2 == null )
      Console.WriteLine( "null" );
  }
}
```

Ha futtatjuk ezt a programot, akkor azt tapasztaljuk, hogy a d2 értéke null lesz.
A Derived osztály ugyan örökli az ICloneable.Clone() tagfüggvényt a BaseType típustól, de annak megvalósítása a származtatott osztály számára már nem megfelelő, hiszen csak az alaposztályt klónozza. A BaseType.Clone() BaseType és nem Derived típusú objektumot hoz létre. Ezért lesz a d2 értéke null a tesztprogramban, hiszen az nem Derived objektum. Még ha meg is tudnánk oldani ezt a gondot, a BaseType.Clone() akkor sem tudná helyesen lemásolni a _dValues tömböt, amit a Derived osztályban vezettünk be. Amikor megvalósítjuk az ICloneable felületet egy osztályban, akkor az

összes származtatott osztályban is meg kell valósítanunk azt. Sőt egy horogként szolgáló függvényt is készítenünk kell, hogy a származtatott osztályok használhassák a mi megvalósításunkat (lásd a 21. tippet). A klónozás támogatásának feltétele, hogy a származtatott osztályokba csak olyan tagváltozók kerülhetnek be, amelyek vagy értéktípusok, vagy olyan hivatkozási típusok, amelyek megvalósítják az `ICloneable` felületet. Ez elég szigorú követelmény az összes származtatott osztállyal szemben. Az `ICloneable` támogatása az alaposztályokban általában olyan komoly terhet ró a származtatott típusokra, hogy célszerű kerülni az `ICloneable` megvalósítását a nem zárt osztályoknál.

Amikor egy teljes hierarchiában kell megvalósítani az `ICloneable` felületet, létrehozhatunk egy elvont `Clone()` tagfüggvényt, ezzel arra kényszerítve a származtatott osztályokat, hogy azok is megvalósítsák azt. Ezekben az esetekben meg kell határoznunk annak módját, hogy a származtatott osztályok hogyan készíthetnek másolatot az alaposztály tagjairól. Ezt egy védett másoló konstruktor létrehozásával érhetjük el:

```
class BaseType
{
  private string _label;
  private int [] _values;

  protected BaseType( )
  {
    _label = "class name";
    _values = new int [ 10 ];
  }

  // A származtatott értékek használják a klónozáshoz
  protected BaseType( BaseType right )
  {
    _label = right._label;
    _values = right._values.Clone( ) as int[ ] ;
  }
}

sealed class Derived : BaseType, ICloneable
{
  private double [] _dValues = new double[ 10 ];

  public Derived ( )
  {
    _dValues = new double [ 10 ];
  }

  // Létrehozunk egy másolatot az alaposztály másoló konstruktorával
  private Derived ( Derived right ) :
    base ( right )
  {
```

```
      _dValues = right._dValues.Clone( )
        as double[ ];
    }

    static void Main( string[] args )
    {
      Derived d = new Derived();
      Derived d2 = d.Clone() as Derived;
      if ( d2 == null )
        Console.WriteLine( "null" );
    }

    public object Clone()
    {
      Derived rVal = new Derived( this );
      return rVal;
    }
  }
```

Az alaposztályok nem valósítják meg az `ICloneable` felületet, hanem egy védett másoló konstruktorral adnak lehetőséget a származtatott osztályoknak arra, hogy lemásolják az alaposztályhoz tartozó részeket. A levélosztályok, amelyeknek zártnak kell lenniük, csak akkor valósítják meg az `ICloneable` felületet, ha szükséges. Az alaposztály nem kényszeríti ki, hogy az összes származtatott osztály megvalósítsa az `ICloneable` felületet, de az összes olyan származtatott osztálynak a rendelkezésére bocsátja a szükséges tagfüggvényeket, amely támogatni szeretné az `ICloneable` felületet.

Az `ICloneable` felületnek megvan a maga haszna, de ezzel inkább a kivételt erősíti, mint a szabályt. Az értéktípusoknál soha ne támogassuk az `ICloneable` felületet, inkább használjuk az értékadó műveletet. A levélosztályoknál csak akkor támogassuk az `ICloneable` felületet, ha a másolat készítéséhez erre feltétlenül szükség van az adott típusnál. Azoknál az alaposztályoknál, amelyeket valószínűleg olyan helyen fogunk használni, ahol az `ICloneable` támogatott lesz, hozzunk létre védett másoló konstruktort. Minden más esetben kerüljük az `ICloneable` felület használatát.

28. tipp

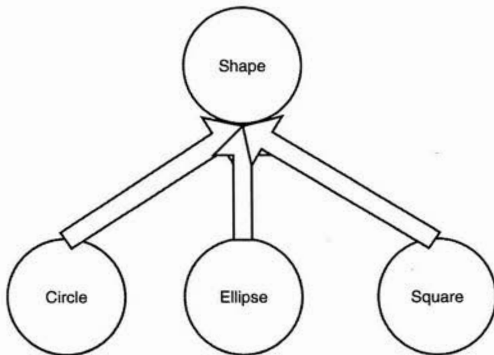
Kerüljük az átalakító műveleteket

Az átalakító műveletekkel egyfajta **helyettesíthetőséget** vezetünk be az osztályok között. A helyettesíthetőség azt jelenti, hogy az egyik osztály helyére egy másikat tehetünk. Ez előnyös is lehet. Egy alaposztály helyén használhatjuk a belőle származtatott osztály példányát, mint ahogyan a klasszikus alakzathierarchiát bemutató példában is. Létrehozunk egy `Shape` alaposztályt, majd különféle alakzatokat származtatunk belőle: `Rectangle`, `Ellipse`, `Circle` és így tovább. A `Circle` objektumokat bárhol használ-

hatjuk, ahol a program Shape objektumot vár. Ezzel a behelyettesítéssel a többalakúságot (polimorfizmus) használjuk ki. Erre azért van lehetőségünk, mert a kör nem más, mint egyfajta alakzat. Amikor létrehozunk egy osztályt, bizonyos átalakításokra automatikusan lehetőséget kapunk. Bármely objektumot behelyettesíthetünk a System.Object egy példányának helyére, ami egyben a .NET osztályhierarchia gyökere. Hasonlóképpen, bármely általunk létrehozott osztályt automatikusan behelyettesíthetjük az osztályban megvalósított felület helyére, az alapfelületek helyére, továbbá az alaposztályainak helyére. A nyelv különféle számszerű átalakításokat is támogat.

Amikor egy átalakító műveletet határozunk meg a típusunkhoz, akkor azzal azt mondjuk a fordítónak, hogy a típusunkat be lehet helyettesíteni a céltípus helyére. Ezek a behelyettesítések azonban gyakran alig észrevehető hibákhoz vezetnek, mivel a típusunk valószínűleg nem tudja tökéletesen helyettesíteni a céltípust. A különféle mellékhatásoknak, amelyek módosítják a céltípus állapotát, valószínűleg egészen más lesz a hatása a mi típusunkra. Ennél is rosszabb, ha az átalakító művelet egy átmeneti objektumot ad vissza, akkor a mellékhatások az átmeneti objektumot módosítják, és így örökre elnyeli őket a szemétgyűjtő. Végül, az átalakító műveletek meghívása az objektum fordítási idejű és nem annak futásidejű típusán alapszik. A típusunk felhasználóinak akár több típusátalakítást is végre kell hajtaniuk az átalakító műveletek felélesztéséhez. Ez a gyakorlat viszont karbantarthatatlan kódhoz vezet.

Ha egy másik típust szeretnénk a mi típusunkra alakítani, használjunk konstruktort. Ez világosabban tükrözi az új objektum létrehozásának műveletét. Az átalakító műveletek nehezen azonosítható hibákat eredményezhetnek a kódunkban. Tegyük fel, hogy a 3.1. ábrán látható könyvtár kódját örököltük. A Circle és az Ellipse osztály is a Shape osztályból származik. Úgy döntünk, hogy békén hagyjuk ezt a hierarchiát, mert bár a Circle és az Ellipse kapcsolatban van egymással, nem szeretnénk nem elvont levélosztályokat a hierarchiánkban, és számos gond jelentkezik, ha megpróbáljuk az Ellipse osztályt a Circle osztályból származtatni. Észrevesszük azonban, hogy minden kör lehetne ellipszis is. Ezen kívül néhány ellipszist behelyettesíthetnénk a körök helyére.



3.1. ábra
Egyszerű alakzathierarchia.

Ebből adódóan megadunk két átalakító műveletet. Minden kör ellipszis, tehát megadunk egy rejtett (beleértett) átalakítást, amivel egy körből ellipszist készíthetünk. A rejtett átalakító műveletet hívja meg a program, amikor az egyik típust át kell alakítani a másik típusra. A nyílt átalakításra ezzel szemben akkor kerül sor, amikor a programozó egy típusátalakító műveletet helyez el a forráskódban.

```
public class Circle : Shape
{
  private PointF _center;
  private float _radius;

  public Circle() :
    this ( PointF.Empty, 0 )
  {
  }

  public Circle( PointF c, float r )
  {
    _center = c;
    _radius = r;
  }

  public override void Draw()
  {
    //...
  }

  static public implicit operator Ellipse( Circle c )
  {
    return new Ellipse( c._center, c._center,
      c._radius, c._radius );
  }
}
```

Most, hogy már megvan a rejtett átalakító műveletünk, bárhol használhatunk `Circle` objektumot, ahol `Ellipse` objektumot vár a program. Az átalakítás ráadásul automatikusan történik:

```
public double ComputeArea( Ellipse e )
{
  // visszaadjuk az ellipszis területét
}

// meghívjuk:
Circle c = new Circle( new PointF( 3.0f, 0 ), 5.0f );
ComputeArea( c );
```

A példa jól mutatja, hogy mit értünk behelyettesíthetőségen. Az ellipszis helyére egy kör került. A ComputeArea függvény a behelyettesítés után is működik. Mázlink volt. Most viszont vizsgáljuk meg ezt a függvényt:

```
public void Flatten( Ellipse e )
{
  e.R1 /= 2;
  e.R2 *= 2;
}

// meghívjuk egy körrel:
Circle c = new Circle( new PointF ( 3.0f, 0 ), 5.0f );
Flatten( c );
```

Ez így nem megy. A Flatten() tagfüggvény ellipszist vár paraméterként. A fordítónak valahogy át kell alakítania a kört ellipszissé. Létrehoztunk egy rejtett átalakítást, ami éppen ezt csinálja. Az átalakítást meghívja a program, és a Flatten() függvény paraméterként megkapja a rejtett átalakításunk során keletkezett ellipszist. Ezt az átmeneti objektumot előbb módosítja a Flatten(), majd rögtön szemét lesz belőle. A Flatten() függvény mellékhatásai bekövetkeznek ugyan, de csak az átmeneti objektumot érintik. A végeredmény az, hogy a c körrel semmi nem történik.

Ha az átalakítást rejtettről nyíltra módosítjuk, azzal csak azt érjük el, hogy a felhasználóknak egy típusátalakítást is meg kell adniuk a függvény hívásakor:

```
Circle c = new Circle( new PointF( 3.0f, 0 ), 5.0f );
Flatten( ( Ellipse ) c );
```

Az eredeti probléma megmaradt. Csak azt értük el, hogy a felhasználóknak már egy típusátalakítást is meg kell adniuk a probléma előidézéséhez. Továbbra is egy átmeneti objektumot hozunk létre, az átmeneti objektumot lapítjuk össze, majd szépen el is dobjuk. A c körnek eközben egy haja szála se görbül. Ha ehelyett létrehozunk egy konstruktort, amivel ellipszist csinálunk a körből, a művelet sokkal világosabb lesz:

```
Circle c = new Circle( new PointF( 3.0f, 0 ), 5.0f );
Flatten ( new Ellipse( c ));
```

A programozók többsége, amikor ránéz a fenti két sorra, azonnal észreveszi, hogy a Flatten() függvénynek átadott ellipszisen végrehajtott módosítások elvesznek. Ők ezt a problémát az új objektum követésével oldanák meg:

```
Circle c = new Circle( new PointF( 3.0f, 0 ), 5.0f );
// Dolgozunk a körrel
// ...
```

```
// Átalakítjuk ellipszissé
Ellipse e = new Ellipse( c );
Flatten( e );
```

Az összelapított ellipszis az e változóban van. Azzal, hogy az átalakító művelet helyett konstruktort használunk, semmit sem veszítettünk a használhatóságból. Mindössze világosabbá tettük, hogy mikor jönnek létre új objektumok. (Felhívom a figyelmét a veterán C++ programozóknak, hogy a C# nem hív meg konstruktorokat a rejtett és nyílt átalakításoknál. Új objektumokat csak akkor hozunk létre, amikor kimondottan a new műveletet használjuk, más esetben soha. Az `explicit` kulcsszóra nincs szükség a C# konstruktorainál.)

Azok az átalakító műveletek, amelyek mezőket adnak vissza az objektumon belül, nem mutatnak ilyen viselkedést. Azokkal más gondok vannak, mivel tátongó rést nyitunk az osztályunk bezártságán. Azzal, hogy a típusunkat egy másik objektumra alakítjuk, az osztályunk felhasználói hozzáférhetnek a belső változókhoz. Ezt a legjobb elkerülni a 23. tippben tárgyalt okok miatt.

Az átalakító műveletek által bevezetett helyettesíthetőség gondokat okozhat a kódunkban. Arra utalunk ugyanis, hogy a felhasználók minden esetben joggal elvárhatják, hogy egy másik osztályt használhatnak az általunk létrehozott osztály helyett. Ilyenkor átmeneti objektumokat módosítunk, majd a változtatások eredményét eldobjuk. Ezeket a finom kis hibákat nehezen lehet megtalálni, mert a fordító az előállított kódban átalakítja ezeket az objektumokat. Kerüljük tehát az átalakító műveletek használatát.

29. tipp

Csak akkor használjuk a new módosítót, ha az alaposztály frissítése miatt ez feltétlenül szükséges

A new módosítót akkor használjuk az osztály egy tagján, ha át akarjuk fogalmazni az alaposztálytól örökölt nem virtuális tag meghatározását. De az, hogy erre lehetőségünk van, nem jelenti azt, hogy feltétlenül élnünk is kell ezzel a lehetőséggel. A nem virtuális tagfüggvények ismételt meghatározása kétértelmű viselkedéshez vezet. A fejlesztők többsége rápillantva az alábbi két kódblokkra feltételezné, hogy ugyanúgy működnek, ha a két osztály öröklés útján kapcsolatban áll egymással:

```
object c = MakeObject( );

// Hívás a MyClass hivatkozáson keresztül:
MyClass cl = c as MyClass;
cl.MagicMethod( );
```

```
// Hívás a MyOtherClass hivatkozáson keresztül:
MyOtherClass cl2 = c as MyOtherClass;
cl2.MagicMethod( );

public class MyClass
{
  public void MagicMethod( )
  {
    // részletek elhagyva
  }
}

public class MyOtherClass : MyClass
{
  // Az osztály MagicMethod tagfüggvényének ismételt meghatározása
  public new void MagicMethod( )
  {
    // részletek elhagyva
  }
}
```

Ez a fajta gyakorlat komoly zavarokat okozhat a fejlesztés során. Ha meghívjuk ugyanazt a függvényt ugyanarra az objektumra, akkor elvárjuk, hogy ugyanaz a kód fusson le. Érezzük, hogy valami nem stimmel azzal a ténnyel, hogy módosítva a hivatkozást, a címkét, amit használunk a függvény hívásához, más viselkedéshez jutunk. Ez így nem következetes. A `MyOtherClass` objektum másképpen viselkedik attól függően, hogy hogyan hivatkozunk rá. A `new` módosító nem csinál virtuális tagfüggvényt a nem virtuális tagfüggvényekből. Csak azt teszi lehetővé, hogy egy másik tagfüggvény kerüljön az osztály névadási hatókörébe.

A nem virtuális függvények kötése statikus. Bármilyen és bárhol lévő forráskód, ami hivatkozik a `MyClass.MagicMethod()` tagfüggvényre, azt és pontosan azt a függvényt fogja meghívni. A futásidejű környezetben semmi nem kutat egy másik, származtatott osztályban található változat után. A virtuális függvények kötése azonban dinamikus. A futásidejű környezet az objektum futásidejű típusa alapján hívja meg a megfelelő függvényt.

Nem szabad azonban úgy értelmeznünk a `new` módosító nem virtuális függvények ismételt meghatározásához való használatát ellenző tanácsot, hogy akkor az alaposztályokban minden tagfüggvényből csináljunk virtuális tagfüggvényt. Ezzel arra utalunk ugyanis, hogy az összes származtatott osztályban illik elkészíteni a virtuális tagfüggvények megvalósítását. A virtuális tagfüggvények halmaza azokat a viselkedéseket fogja össze, amelyeket a származtatott osztályok várhatóan módosítani fognak. Az „alapértelmezés szerint virtuális" stílus azt fejezi ki, hogy a származtatott osztályok módosíthatják az osztály teljes viselkedését. Valójában azonban azt jelenti, hogy nem igazán gondoltuk végig annak következményeit, hogy mely viselkedéseket módosíthatják a származtatott osztályok. Jobban tesszük, ha ehelyett átgondoljuk, hogy mely tagfüggvényeket és tulajdonságokat szánjuk

többalakúnak. Ezek – de csak ezek – legyenek virtuálisak. Ne úgy gondoljunk erre, mintha korlátoznánk az osztályunk felhasználóit. Gondoljunk inkább arra, hogy ezzel útmutatást adunk a típusaink viselkedésének testreszabásához szükséges belépési pontokat illetően.

Van egy alkalom, de tényleg csak egy, amikor érdemes használni a `new` módosítót. Ez akkor fordul elő, ha egy olyan alaposztály új változatát szeretnénk felhasználni, amelynek egyik tagfüggvénye pont olyan néven szerepel, amit korábban már mi is használtunk. Már írtunk olyan kódot, ami az osztályunkban található tagfüggvény nevétől függ. Az is lehet, hogy korábban kiadtunk különböző szerelvényeket is, amelyek ezt a tagfüggvényt használják. Létrehoztuk az alábbi osztályt a könyvtárunkban, a `BaseWidget` osztályt használva, aminek a meghatározása egy másik könyvtárban található:

```
public class MyWidget : BaseWidget
{
  public void DoWidgetThings( )
  {
    // részletek elhagyva
  }
}
```

Befejeztük a vezérlőt, a megrendelőink pedig használni kezdik. Aztán megtudjuk, hogy a `BaseWidget` cége kiadott egy új változatot. Kiéhezve az újabb szolgáltatásokra azonnal megvásároljuk, és megpróbáljuk felépíteni a `MyWidget` osztályt. De nem megy, mert a fiúk a `BaseWidget` cégnél hozzáadták az osztályhoz a saját `DoWidgetThings()` tagfüggvényüket:

```
public class BaseWidget
{
  public void DoWidgetThings()
  {
    // részletek elhagyva
  }
}
```

Ez baj. Az alaposztály becsempészett egy tagfüggvényt az osztály névterébe. Két megoldás kínálkozik a problémára. Módosíthatnánk a `DoWidgetThings` tagfüggvényünk nevét:

```
public class MyWidget : BaseWidget
{
  public void DoMyWidgetThings( )
  {
    // részletek elhagyva
  }
}
```

Vagy használhatjuk a new módosítót:

```
public class MyWidget : BaseWidget
{
  public new void DoWidgetThings( )
  {
    // részletek elhagyva
  }
}
```

Ha a MyWidget osztály összes ügyfelének forráskódja elérhető, akkor hosszabb távon egyszerűbb, ha módosítjuk a tagfüggvény nevét. Ha azonban a nagyvilág szétkapkodta a MyWidget osztályunkat, akkor minden felhasználónknak számos módosítást kellene végrehajtania. Ekkor jön jól a new módosító. Az ügyfeleink továbbra is a DoWidgetThing tagfüggvényünket fogják használni. Egyik sem fogja meghívni a BaseWidget.DoWidgetThings() tagfüggvényt, hiszen az nem létezett. A new módosító kezeli azt az esetet, amikor az alaposztály frissítése során bekövetkezett módosítások ütköznek az osztályunk egy korábban bevezetett tagjával.

Később persze előfordulhat, hogy a felhasználók használni akarják majd a BaseWidget.DoWidgetThings() tagfüggvényt. Ha így van, akkor visszaérkeztünk az eredeti problémához: két tagfüggvény, ami ugyanúgy néz ki, de mégis más. Gondoljuk át a new módosító használatának hosszú távú következményeit. Néha jobban járunk, ha a rövid távon kellemetlenebb utat választjuk, és módosítjuk a tagfüggvényünk nevét.

A new módosítót körültekintően kell használnunk. Ha ész nélkül alkalmazzuk, félreérthető függvényhívásokat hozunk létre az objektumainkban. Arra a különleges esetre tartogassuk, amikor az alaposztály frissítése ütközik a saját osztályunkkal. Még ilyenkor is jól gondoljuk meg, hogy tényleg használni akarjuk-e. De ami ennél is fontosabb, hogy más körülmények között soha ne használjuk.

4

Bináris összetevők létrehozása

A bináris összetevők létrehozása sokban hasonlít az osztályok létrehozásához. Mindkét esetben a különböző szolgáltatásokat próbáljuk meg egymástól elkülöníteni és felosztani a program részei között. A különbség az, hogy a bináris összetevők segítségével az így leválasztott szolgáltatásokat önálló egységekben tehetjük közzé. Az összetevő-szerelvényeket azzal a céllal hozzuk létre, hogy leegyszerűsítsük a közös logikán alapuló viselkedés megosztását, hogy elősegítsük a különböző nyelveken íródott programok együttműködését, és hogy leegyszerűsítsük a programok és azok frissítéseinek telepítését.

A szerelvények a .NET összetevőcsomagjai. Minden szerelvényt önállóan lehet szállítani és frissíteni. Az, hogy egy korábban kiadott alkalmazás szerelvényenkénti frissítése mennyire lesz egyszerű, attól függ, hogy milyen hatékonyan tudjuk megszüntetni a szerelvények közti átfedéseket. Az átfedések minimalizálása nem csak a fordítási függőségek megszüntetését jelenti. Ennél többről van szó. Olyan szerelvényeket kell építenünk, amelyeket könnyen frissíthetünk a terepen az új változatokkal. Ez a fejezet arról szól, hogy miként lehet egyszerűen kezelhető, egyszerűen használható és egyszerűen frissíthető szerelvényeket készíteni.

A .NET környezetet olyanra tervezték, hogy képes legyen támogatni a több bináris összetevőből álló alkalmazásokat. Ezeket az összetevőket egymástól függetlenül frissíthetjük, ha eljár felettük az idő, méghozzá úgy, hogy elég, ha a több szerelvényből álló alkalmazásoknál csak a frissítésre szoruló szerelvényt frissítjük és telepítjük. Ennek a szolgáltatásnak a kihasználásához viszont legalább egy kicsit meg kell ismerkednünk azzal, hogy a CLR miként találja meg és tölti be a szerelvényeket. Ezen kívül az általunk készített összetevőknek meg kell felelniük bizonyos elvárásoknak, ha a legtöbbet szeretnénk kihozni a bináris frissítések adta lehetőségekből. A bevezető további részében a legfontosabb fogalmakkal ismerkedünk meg.

A CLR betöltő a program indításakor nem tölti be az összes szerelvényt, amire a programunkban hivatkozunk. A betöltő ehelyett csak akkor oldja fel a szerelvényhivatkozást, ha a futásidejű környezetnek szüksége van egy olyan tagra, ami az adott szerelvényben található. A hivatkozás lehet egy tagfüggvény hívása vagy adathozzáférés is. A betöltő megkeresi a hivatkozott szerelvényt, betölti, majd a JIT fordító létrehozza a szükséges IL kódot.

Amikor a CLR-nek be kell töltenie egy szerelvényt, első lépésként meg kell határoznia, hogy melyik fájlt kell betöltenie. A szerelvényben található metaadatok között találunk egy bejegyzést minden olyan szerelvényhez, amelyre a szerelvény hivatkozik. Ez a bejegyzés különbözik az erős és a gyenge névvel rendelkező szerelvények esetében. Az erős nevek négy részből állnak: ezek a szerelvény szöveges megnevezése, a szerelvény változatszáma, az országot leíró jellemző és a nyilvános kulcs elem. Ha a kért szerelvény neve nem erős név, akkor a bejegyzés csak a szerelvény nevét tartalmazza. Ha erős neveket használunk, csökkenthetjük annak a valószínűségét, hogy a szerelvényünk helyére rosszindulatú kód férkőzik be. Az erős nevek alkalmazásával lehetőségünk nyílik beállítófájlok használatára is, amelyekkel a kért változatot leképezhetjük az összetevő újabb, fejlettebb változatára.

A megfelelő szerelvény nevének és változatának megállapítása után a CLR megnézi, hogy be van-e már töltve a szerelvény az adott alkalmazáskörnyezetbe. Ha igen, akkor használni kezdi a szerelvényt. Ha nem, akkor a CLR folytatja a szerelvény keresését. Ha a kért szerelvénynek erős neve van, akkor a CLR előbb átnézi a Global Assembly Cache-t (GAC, globális szerelvénygyorstárt). Ha itt sem találja a szerelvényt, akkor a betöltő megkeresi az alap programkönyvtárat (codebase directory) a beállítófájlban. Ha van ilyen könyvtár, akkor csak ebben keresi a kért szerelvényt, és ha nem találja, a betöltés sikertelen lesz. Amennyiben a betöltő nem talál utalást programkönyvtárra, néhány előre meghatározott könyvtárat néz végig. Ezek a következők:

- Az alkalmazáskönyvtár. Ez megegyezik azzal a hellyel, ahol a fő alkalmazásszerelvény található.
- Az országkönyvtár. Ez az alkönyvtár az alkalmazáskönyvtárban található. Az alkönyvtár neve megegyezik az aktuális ország nevével.
- A szerelvényalkönyvtár. Az alkönyvtár neve megegyezik a kért szerelvény nevével. A kettő kombinációjára is lehetőség van *[ország]/[szerelvénynév]* alakban.
- A privát bináris útvonal (binpath). Ez egy privát könyvtár, melynek meghatározása az alkalmazás beállítóállományában található. Ezt szintén kombinálni lehet az ország- és a szerelvényútvonalakkal: vagy *[bináris útvonal]/[szerelvénynév]*, vagy *[bináris útvonal]/[ország]*, de még a *[bináris útvonal]/[ország]/[szerelvénynév]* is lehetséges.

Három dolgot kell megjegyeznünk az eddigiekből. Először is, csak erős névvel ellátott szerelvényeket lehet a GAC gyorstárban tárolni. Másodszor, beállítóállományok segítségével módosíthatjuk az alapértelmezett viselkedést az alkalmazásban található, erős névvel ellátott szerelvények frissítése érdekében. Harmadszor, az erős névvel rendelkező szerelvények nagyobb biztonságot jelentenek a rosszindulatú támadásokkal szemben.

Remélhetőleg a CLR szerelvénybetöltési módszeréről szóló bevezető elindította már a fantáziánkat a terepen frissíthető összetevők készítését illetően. Elsőként mindig gondoljunk arra, hogy a szerelvényeket erős nevekkel lássuk el, vagyis töltsük ki az összes metaadatbejegyzést. Amikor létrehozunk egy projektet a VS .NET környezetben, akkor célszerű kitölteni az `assemblyInfo.cs` állományban létrehozott összes jellemző értékét, beleértve a teljes változatszámot is. Ezzel megkönnyítjük a szerelvény későbbi frissítését a terepen. A VS .NET három különböző részt hoz létre az `assemblyInfo.cs` állományban. Az elsőnek leginkább tájékoztató célja van:

```
[assembly: AssemblyTitle("My Assembly")]
[assembly: AssemblyDescription
  ("This is the sample assembly")]
#if DEBUG
[assembly: AssemblyConfiguration("Debug")]
#else
[assembly: AssemblyConfiguration("Release")]
#endif
[assembly: AssemblyCompany("My company")]
[assembly: AssemblyProduct("It is part of a product")]
[assembly: AssemblyCopyright("Insert legal text here.")]
[assembly: AssemblyTrademark("More legal text")]
[assembly: AssemblyCulture("en-US")]
```

Az utolsó elemet csak a szerelvény helyi változataiban kell kitölteni. Ha a szerelvényünk nem tartalmaz helyi erőforrásokat, akkor hagyjuk üresen. Az országot leíró karakterláncnak (`AssemblyCulture`) az RFC 1766 szabványt kell követnie.

A következő rész a változatszámot tartalmazza. A VS .NET megfogalmazásában ez így hangzik:

```
[assembly: AssemblyVersion("1.0.*")]
```

Az `AssemblyVersion` négy részből áll: `Major.Minor.Build.Revision` (főváltozat.alváltozat.változat.javítás). A csillag arra utasítja a fordítót, hogy a változat és a javítás helyére az aktuális dátumot és időt írja be. A változatba a 2000. január 1-től eltelt napok száma, míg a javítás helyére a helyi idő szerint éjfél óta eltelt másodpercek 2-vel osztott száma kerül. Ez az algoritmus garantálja, hogy a változat és a javítás száma mindig nő. Minden újabb változathoz nagyobb szám tartozik, mint az előzőhöz.

A jó hír ezzel az algoritmussal kapcsolatban az, hogy nem lesz két olyan változat, amelynek pontosan megegyezik a változatszáma. A rossz hír az, hogy a változat és a javítás számát utólag kell feljegyeznünk azoknál a változatoknál, amelyek ténylegesen megjelennek. Én szeretem, ha a fordító hozza létre helyettem a változat és a javítás számát. Az adott változat kiadásakor pontosan feljegyzem az előállított változatszámot, így mindig tudom

a legutolsó változatszámot. A változatszámot mindig módosítom, amikor kiadok egy új szerelvényt, bár vannak ez alól kivételek is. A COM összetevők minden alkalommal bejegyeztetik magukat, amikor felépítjük őket. Ha megengedjük, hogy a fordító automatikusan előállítsa a változat számát, akkor minden felépítésnél létrejön egy bejegyzés a rendszerleíró adatbázisban, ami így hamarosan megtelik haszontalan információkkal.

Az utolsó rész az erős nevekkel kapcsolatos információkat tartalmazza:

```
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("")]
[assembly: AssemblyKeyName("")]
```

Érdemes megfontolni, hogy minden szerelvényünkhöz előállítsunk egy erős nevet. Az erős névvel ellátott szerelvények ellenállóbbak a támadásokkal szemben, és támogatják az alkalmazás szerelvényeinek egymástól független frissítését. Az ASP.NET alkalmazásoknál azonban kerüljük az erős neveket, mert a helyben telepített erős névvel ellátott szerelvények helytelenül töltődnek be. Az erős névvel ellátott szerelvényeket ezen kívül fel kell díszíteni az `AllowPartiallyTrustedCallers` jellemzővel, különben az erős névvel nem rendelkező szerelvények nem tudják elérni őket. Ahhoz, hogy frissíthessünk egy összetevőt kint a terepen, a felület nyilvános és védett részeinek összeegyeztethetőnek kell lenniük az IL kód szintjén. Ez azt jelenti, hogy nem törölhetünk tagfüggvényeket, nem módosíthatjuk a paramétereket és a visszatérési értékeket sem. A lényeg röviden az, hogy ne kelljen újrafordítani azokat az összetevőket, amelyek hivatkoznak az összetevőnkre.

A hivatkozott szerelvény módosítását a beállításokon keresztül módosíthatjuk. A beállításokat három különböző helyen tárolhatják, attól függően, hogy milyen összetevőt akarunk frissíteni. Egy adott alkalmazás beállításához egy alkalmazásbeállító állományt kell létrehoznunk, amit abba a könyvtárba teszünk, ahol az alkalmazás található. Ha az összes olyan alkalmazást be szeretnénk állítani, amelyik az adott összetevőt használja, akkor a GAC gyorstárban hozunk létre egy kiadói házirendállományt (publisher policy file). Végül pedig, ha globális beállításmódosítást akarunk elvégezni, akkor a `machine.config` állományt módosítjuk, amit a .NET futásidejű környezet `Config` könyvtárában találunk (lásd a 37. tippet).

A gyakorlatban a `machine.config` állományt soha nem fogjuk módosítani a szerelvényeink frissítésekor. Ez a fájl a teljes gépre vonatkozó beállításokat tartalmazza. Egy alkalmazás frissítésekor az alkalmazás beállítóállományát használjuk, amikor pedig több alkalmazás osztozik egy összetevőn, akkor a kiadói házirendet módosítjuk.

A beállítófájlok olyan XML kódot tartalmaznak, ami leírja a már létező és a frissített változatot is:

```
<dependentAssembly>
  <assemblyIdentity name="MyAssembly"
```

```
    publicKeyToken="a0231341ddcfe32b" culture="neutral" />
  <bindingRedirect oldVersion="1.0.1444.20531"
    newVersion="1.1.1455.20221" />
</dependentAssembly>
```

A beállítófájlt a szerelvény régi és új változatának azonosításához használjuk. Amikor egy frissített szerelvényt telepítünk, akkor frissítjük vagy elkészítjük a megfelelő beállítófájlt, és onnantól kezdve az alkalmazások az új változatot használják.

Gondoljunk úgy a programjainkra, mint szerelvények gyűjteményére, amelyeket egymástól függetlenül akarunk frissíteni. Ahhoz, hogy szerelvények önálló frissítésével tudjuk frissíteni az alkalmazásunkat, előbb egy kis munkát is el kell végeznünk, hogy az első telepítés tartalmazza a frissítések támogatásához szükséges információkat.

30. tipp

Használjunk CLS-megfelelő szerelvényeket

A .NET környezet érzéketlen a nyelvek közti különbségekre. A fejlesztők elméletileg korlátok nélkül használhatják a különböző .NET nyelveken íródott összetevőket. Ez a gyakorlatban is majdnem így van. Ahhoz, hogy a más nyelveken programozó fejlesztők is használni tudják a szerelvényeinket, azoknak meg kell felelniük a Common Language Subsystem (CLS, közös nyelvi alrendszer) előírásainak.

A CLS előírásai új fordulatot jelentenek a nyelvek közti együttműködés terén. A CLS leírása a különféle műveleteknek egy olyan részhalmazát tartalmazza, amit minden nyelvnek támogatnia kell. A CLS előírásainak megfelelő szerelvényt úgy hozhatunk létre, ha a szerelvényünk nyilvános felületét a CLS leírásában meghatározott szolgáltatásokra korlátozzuk. Ilyenkor az összes olyan nyelvnek fel kell tudnia használni az összetevőt, amelyik megfelel a CLS előírásainak. Ez persze nem jelenti azt, hogy a teljes programozási eszköztárunkat a C# nyelv CLS-nek megfelelő részeire kell korlátoznunk.

Egy CLS-megfelelő szerelvény létrehozásához két szabályt kell betartanunk. Először is, a nyilvános, illetve védett tagoktól származó visszatérési értékek típusának meg kell felelnie a CLS előírásainak. Másodszor, a CLS előírásainak nem megfelelő minden nyilvános vagy védett tagnak rendelkeznie kell egy CLS-nek megfelelő változattal.

Az első szabályt nem nehéz betartani. A betartatására felkérhetjük a fordítót is, ha a szerelvényhez hozzáadjuk a `CLSCompliant` jellemzőt:

```
[ assembly: CLSCompliant( true ) ]
```

A fordító ezután megköveteli, hogy a szerelvény végig megfeleljen a CLS előírásainak. Ha olyan nyilvános tagfüggvényt vagy tulajdonságot írunk, ami a CLS előírásainak ellentmondó szerkezetet használ, akkor hibát jelez. Ez azért jó, mert a CLS előírások betartásának célja így egyszerűen követhetővé válik. A CLS-megfelelés figyelésének bekapcsolása után az alábbi két meghatározást nem lehet lefordítani, mert az előjel nélküli egészek nem felelnek meg a CLS-nek:

```
// Nem felel meg a CLS-nek, mert a visszatérési értéke
// előjel nélküli egész
public UInt32 Foo( )
{
  return _foo;
}

// Nem felel meg a CLS-nek, mert a paraméterének értéke
// előjel nélküli egész
public void Foo2( UInt32 parm )
{
}
```

Emlékezzünk vissza, hogy a CLS-megfelelő szerelvény létrehozása csak az adott szerelvényen kívülről látható elemeket érinti. A `Foo` és a `Foo2` akkor vált ki CLS-megfelelési hibákat, ha nyilvánosként vagy védettként vezetjük be őket. Ha a `Foo` és a `Foo2` belső vagy privát tag lenne, akkor bekerülhetnének egy CLS-megfelelő szerelvénybe. CLS-megfelelő felületekkel csak azoknak az elemeknek kell rendelkezniük, amelyek a szerelvényen kívülről láthatók.

De mi a helyzet az alábbi tulajdonsággal? Ez megfelel a CLS előírásainak?

```
public MyClass TheProperty
{
  get { return _myClassVar; }
  set { _myClassVar = value; }
}
```

Az attól függ. Ha a `MyClass` megfelel a CLS előírásainak és ezt jeleztük is, akkor a tulajdonság is megfelel a CLS-nek. Másrészt viszont, ha semmi nem jelzi, hogy a `MyClass` megfelel a CLS előírásainak, akkor a tulajdonság sem felel meg a CLS-nek. Ez azt jelenti, hogy az előbbi `TheProperty` tulajdonság csak akkor felel meg a CLS-nek, ha a `MyClass` egy olyan szerelvényben található, ami megfelel a CLS-nek.

Úgy nem építhetünk fel egy CLS-megfelelő szerelvényt, hogy közben a nyilvános és védett felületébe olyan típusokat teszünk, amelyek nem felelnek meg a CLS-nek. Ha összetevő-fejlesztőként nem jelöljük meg a CLS-megfelelő szerelvényeket, azzal megnehezítjük az össze-

tevőink felhasználóinak a CLS-nek megfelelő összetevők létrehozását, hiszen ilyenkor el kell rejteniük a típusainkat, majd egy CLS-megfelelő burkolóba kell helyezniük az összetevő szolgáltatásait. Igen, így is meg lehet oldani. És nem, nem illik így bánni azokkal a programozókkal, akik fel akarják használni az összetevőinket. Az a legjobb, ha a munkánk során mindig CLS-megfelelő összetevők írására törekszünk. Az ügyfelek számára így a legegyszerűbb a CLS-nek megfelelő szerelvények létrehozása a mi munkánk felhasználásával.

A második szabály betartása rajtunk áll vagy bukik. Gondoskodnunk kell róla, hogy az összes nyilvános és védett műveletet nyelvfüggetlen módon valósítsuk meg. Arra is vigyáznunk kell, nehogy a többalakúságot kihasználva egy nem megfelelő objektum átcsússzon a felületen.

A műveletek túlterhelése olyan szolgáltatás, amit néhányan imádnak, mások viszont ki nem állhatnak. Mint ilyen, nem minden nyelv támogatja a műveletek (operátorok) túlterhelését. A CLS sem mellette, sem ellene nem foglal állást. Ehelyett minden művelethez meghatároz egy függvénynevet: az `op_equals` lesz annak a függvénynek a neve, amit az `operator =` függvény megírásakor kapunk. Az `op_add` a túlterhelt összeadás művelet neve lesz. Amikor túlterhelt műveletet írunk, az ezeket támogató nyelvekben a műveleti jelekre vonatkozó nyelvtant használhatjuk. Azoknak a fejlesztőknek, akik olyan nyelvet használnak, ami nem támogatja a műveletek túlterhelését, az `op_` alakú függvénynevet kell használniuk. Ha számítunk rá, hogy ilyen programozók is használni fogják a CLS-megfelelő szerelvényeinket, adjuk meg a lehetőséget a kényelmesebb forma használatára. A javaslatom tehát a következő: minden esetben, amikor túlterhelünk egy műveletet, hozzunk létre egy vele jelentésben megegyező függvényt:

```
// Túlterhelt összeadás művelet, szokásos C# forma:
public static Foo operator+( Foo left, Foo right)
{
  // Ugyanazt a megvalósítást használjuk, mint az Add tagfüggvény:
  return Foo.Add( left, right );
}

// Statikus függvény, néhány nyelvnél kívánatos:
public static Foo Add( Foo left, Foo right)
{
  return new Foo ( left.Bar + right.Bar );
}
```

Végezetül, figyeljünk oda a CLS-nek nem megfelelő típusokra, nehogy az egyik felületen keresztül átjusson egy ilyen, amikor többalakú paramétereket használunk. Eseményparaméatereknél ez könnyen megeshet. Létrehozunk egy típust, ami nem felel meg a CLS-nek, aztán egy olyan helyen használjuk, ahol a várt alaposztály egyébként megfelelne a CLS előírásainak.

Tegyük fel, hogy létrehozzuk ezt az osztályt, amit az EventArgs osztályból származtatunk:

```
internal class BadEventArgs : EventArgs
{
  internal UInt32 ErrorCode;
}
```

A BadEventArgs típus nem felel meg a CLS-nek. Más nyelven íródott eseménykezelőkkel nem szabad használnunk. A többalakúság azonban ezt lehetővé teszi. Az esemény típusát megadhatjuk az alaposztály, vagyis az EventArgs osztály segítségével is:

```
// Elrejtjük a CLS-nek nem megfelelő eseményparamétert:
public delegate void MyEventHandler(
  object sender, EventArgs args );

public event MyEventHandler OnStuffHappens;


// A kód, ami kiváltja az eseményt:
BadEventArgs arg = new BadEventArgs( );
arg.ErrorCode = 24;

// A felület szabályos, a futásidejű típus viszont nem az:
OnStuffHappens( this, arg );
```

A felület bevezetése, ami egy EventArgs paramétert használ, megfelel a CLS-nek. A tényleges paraméter viszont, amit a helyére írtunk, már nem. A végeredmény egy olyan típus, amit néhány nyelv képtelen használni.

A CLS-megfelelés tárgyalását annak bemutatásával zárjuk, hogy a CLS-nek megfelelő osztályok miként valósíthatnak meg a CLS előírásainak megfelelő, illetve nem megfelelő felületeket. Ez elég bonyolult dolog, de mi leegyszerűsítjük. A felületek CLS-megfelelésének megértése hozzásegít minket a CLS előírásainak való megfelelés tökéletes megértéséhez, és hogy lássuk, hogyan értelmezi a környezet a megfelelést.

Az alábbi felület megfelel a CLS előírásainak, ha egy CLS-megfelelő szerelvényben vezetjük be:

```
[ assembly:CLSCompliant( true ) ]
public interface IFoo
{
  void DoStuff( Int32 arg1, string arg2 );
}
```

Ezt a felületet bármely CLS-megfelelő osztályban megvalósíthatjuk. Ha viszont egy olyan szerelvényben valósítjuk meg, amit nem jelöltünk meg a CLS-megfelelést jelző jellemzővel, akkor az `IFoo` felület nem fog megfelelni a CLS-nek. Vagyis egy felület csak akkor felel meg a CLS-nek, ha egy CLS-megfelelő szerelvényben határozzuk meg. Az, hogy a felület maga megfelel a CLS-nek, önmagában nem elegendő. Ennek oka a fordító teljesítményében keresendő. A fordítók csak akkor vizsgálják a típusok CLS-megfelelését, ha az adott szerelvényben ott a jelzés, hogy az megfelel a CLS-nek. Hasonlóképpen, a fordítók feltételezik, hogy a CLS-nek nem megfelelő szerelvények típusai maguk sem felelnek meg a CLS előírásainak. Ennek a felületnek a tagjai azonban CLS-megfelelő aláírással rendelkeznek. Bár az `IFoo` felületnél nem jeleztük, hogy az megfelel a CLS-nek, attól még az `IFoo` felületet megvalósíthatjuk egy CLS-megfelelő osztályban. Ennek az osztálynak az ügyfelei aztán elérhetnék a `DoStuff` tagfüggvényt az osztályra mutató hivatkozáson keresztül, de az `IFoo` felületre mutató hivatkozáson keresztül nem.

Vegyük most az alábbi apró módosítást:

```
public interface IFoo2
{
  // Nem felel meg a CLS előírásainak, előjel nélküli egész
  void DoStuff( UInt32 arg1, string arg2 );
}
```

Azok az osztályok, amelyek nyilvánosan valósítják meg az `IFoo2` felületet, nem felelnek meg a CLS előírásainak. Ahhoz, hogy egy olyan CLS-megfelelő osztályt készítsünk, ami megvalósítja az `IFoo2` felületet, konkrétan meg kell valósítanunk a felületet:

```
public class MyClass: IFoo2
{
  // konkrét felületmegvalósítás
  // A DoStuff() nem része a MyClass osztály nyilvános felületének
  void IFoo2.DoStuff( UInt32 arg1, string arg2 )
  {
    // tartalom elhagyva
  }
}
```

A `MyClass` osztály nyilvános felülete megfelel a CLS-nek; az `IFoo2` felületet váró ügyfelek a CLS-nek nem megfelelő `IFoo2` mutatón keresztül érhetik el.

Bonyolult? Nem, nem igazán. A CLS-megfelelő típusok létrehozása megköveteli, hogy a nyilvános és a védett felületek csak olyan típusokat tartalmazzanak, amelyek megfelelnek a CLS előírásainak. Ez azt jelenti, hogy az alaposztálynak meg kell felelnie a CLS-nek. Az összes olyan felületnek, amit nyilvánosan valósítunk meg, meg kell felelnie a CLS-nek.

Ha olyan felületet valósítunk meg, amelyik nem felel meg a CLS-nek, akkor azt egy konkrét felületmegvalósítással el kell rejtenünk a nyilvános felület elől.

A CLS-megfeleléshez nincs szükség arra, hogy a program szerkezetében és megvalósításában végig közös nevezőre jussunk más nyelvekkel. Csak a szerelvényünk nyilvánosan megvalósított részeire kell nagyon odafigyelnünk. Minden nyilvános és védett osztálynak, és a bennük lévő összes típusnak meg kell felelnie a CLS előírásainak:

- az alaposztályoknak,
- a nyilvános és védett tagfüggvények visszatérési értékeinek,
- a nyilvános és védett tagfüggvények és indexelők paramétereinek,
- a futásidejű eseményparamétereknek,
- a nyilvánosan bevezetett vagy megvalósított felületeknek.

A fordító megkísérli betartatni a CLS-megfelelést. Ez megkönnyíti számunkra, hogy minimális szintű CLS-támogatást nyújtsunk. Egy kis odafigyeléssel olyan szerelvényeket készíthetünk, amit bárki használni tud, bármilyen nyelven programozzon is. A CLS leírása kísérletet tesz a nyelvek közötti együttműködés lehetőségének megteremtésére anélkül, hogy ehhez fel kellene áldoznunk a kedvenc nyelvünkben megszokott szerkezetek használatát. Ehhez csak különböző változatokat kell megadnunk a felületünkön.

A CLS-megfelelés arra kényszerít minket, hogy egy kicsit elgondolkozzunk a nyilvános felületeken, a többi nyelv szemszögéből nézve őket. Az összes kódunkat nem kell a CLS-nek megfelelő szerkezetekre korlátoznunk, csak a nyilvános felületeken kell kerülnünk a nem megfelelő elemeket. A cserébe kapott együttműködési lehetőség más nyelvekkel megéri a fáradságot.

31. tipp

Használjunk rövid, egyszerű függvényeket

Gyakorlott programozók lévén, bármilyen nyelven műveltük is a programozás művészetét a C# előtt, sokféle trükköt sajátítottunk el a hatékonyabb fejlesztés érdekében. Előfordul, hogy ami korábban működött, az a .NET környezetben éppen visszafogja a programunkat. Különösen igaz ez, amikor saját kezűleg próbáljuk meg optimalizálni a kódunkat a C# fordító helyett. Ezzel gyakran éppen a JIT fordítót akadályozzuk meg abban, hogy hatékony kódot állítson elő. A többletteljesítmény oltárán feláldozott munkánk eredménye lassabb kód lesz. Jobban járunk, ha a legvilágosabb kódot írjuk meg, aztán hagyjuk, hogy a többit a JIT fordító végezze el helyettünk. Az egyik leggyakoribb példa a korai optimalizálásra, amikor bonyolult függvényeket írunk annak reményében, hogy ezzel elkerülhetünk néhány függvényhívást. A .NET alkalmazásaink teljesítményét erősen visszavetheti az a gyakorlat, amikor egy ciklus belsejében valósítunk meg egy olyan gondolatmenetet, ami megérdemelne egy külön függvényt. Nézzük meg részletesen, miről is van szó.

Ennek a fejezetnek a bevezetőjében már kaptunk egy rövid és leegyszerűsített bevezetőt a JIT fordító működési elvébe. A .NET futásidejű környezet elindítja a JIT fordítót, ami lefordítja a C# fordító által létrehozott IL kódot gépi kódra. Ez a feladat azonban részekre bomlik a programunk végrehajtási ideje alatt. Ahelyett, hogy a program indításakor bekövetkezne a teljes program JIT fordítása, a CLR függvényenként hívja meg a JIT fordítót. Ezzel elfogadható szintre csökkennek az indítással járó feladatok, az alkalmazás pedig továbbra is gyorsan reagál, ha további kód JIT fordítására van szükség. Azok a függvények pedig, amelyeket soha nem hív meg a programunk, egyáltalán nem kerülnek a JIT fordítóhoz. Ha a kódot kevés hosszabb függvény helyett sok kisebb függvény között osztjuk el, azzal csökkenthetjük a feleslegesen JIT-fordítandó kód mennyiségét. Vegyük például az alábbi meglehetősen mesterkélt példát:

```
public string BuildMsg( bool takeFirstPath )
{
  StringBuilder msg = new StringBuilder( );
  if ( takeFirstPath )
  {
    msg.Append( "A problem occurred." );
    msg.Append( "\nThis is a problem." );
    msg.Append( "imagine much more text" );
  } else
  {
    msg.Append( "This path is not so bad." );
    msg.Append( "\nIt is only a minor inconvenience." );
    msg.Append( "Add more detailed diagnostics here." );
  }
  return msg.ToString( );
}
```

Amikor először meghívjuk a `BuildMsg` függvényt, akkor mindkét részét lefordítja a JIT fordító, pedig csak az egyikre lenne szükség. De tegyük fel, hogy átírjuk a függvényt a következőképpen:

```
public string BuildMsg( bool takeFirstPath )
{
  if ( takeFirstPath )
  {
    return FirstPath( );
  } else
  {
    return SecondPath( );
  }
}
```

Mivel mindkét elágazás belsejét áttettük egy-egy külön függvénybe, ezeket a függvényeket már igény szerint fordíthatja le a JIT fordító, ahelyett, hogy mindent rögtön először le kellene fordítania. Ez a példa valóban elég mesterkélt, és ebben az esetben nem sokat nyerünk a dolgon. De gondoljunk bele, hogy milyen gyakran írunk ennél jóval hosszabb függvényeket. Képzeljünk el például egy olyan `if` utasítást, aminek mindkét ágában 20 vagy annál is több utasítás van. Mindkét ág JIT fordításával fizetünk, amikor először meghívjuk a függvényt. Ha az egyik ág egy valószínűtlen hiba miatt van csak ott, akkor olyan költségeket idézünk elő, amiket könnyen elkerülhetnénk. A rövid függvényekkel a JIT fordítónak csak a program végrehajtásához szükséges kódot kell lefordítania, nem pedig terjedelmes kódtömeget, amire az adott pillanatban nem is lenne szükség. A JIT fordításon megtakarítható idő hatványozódik a hosszú `switch` elágazásoknál, ha az egyes eseteket nem a `case` utasítások belsejében, hanem külön függvényekben kezeljük.

A rövid és egyszerű függvények megkönnyítik a JIT fordítónak a változók regiszterezését. A **regiszterezés** (enregistration) az a folyamat, amikor a fordító kiválasztja, hogy mely helyi változókat lehet a regiszterekben tárolni a verem helyett. Ha kevesebb helyi változóval dolgozunk, azzal megkönnyítjük a JIT fordítónak, hogy megtalálja a legmegfelelőbb változókat a regiszterezéshez. A programvezérlés egyszerűsége szintén hatással van arra, hogy a JIT fordító milyen hatékonysággal végzi a változók regiszterezését. Ha egy függvény egyetlen ciklusból áll, akkor az adott ciklusváltozót a JIT valószínűleg regiszterezni fogja. Ha viszont a függvényben több ciklus is található, akkor a JIT fordítót nehéz döntés elé állítjuk, és megnehezítjük annak a ciklusváltozónak a kiválasztását, amit a fordító végül regiszterezni fog. Minél egyszerűbb a függvény, annál jobb. Minél rövidebb egy függvény, annál valószínűbb, hogy kevesebb helyi változót használ, s így a JIT fordító könnyebben optimalizálhatja a regiszterek használatát. A JIT fordító a helyben kifejtéssel kapcsolatban is különféle döntéseket hoz. A **helyben kifejtés** (behelyettesítés, inlining) a függvény törzsének beírását jelenti a függvényhívás helyére. Vegyük az alábbi példát:

```
// csak olvasható név tulajdonság:
private string _name;
public string Name
{
  get
  {
    return _name;
  }
}

// hozzáférés a tulajdonsághoz:
string val = Obj.Name;
```

A tulajdonság hozzáférőjének törzse kevesebb utasításból áll, mint amennyi a függvény meghívásához szükséges: a regiszterek állapotának mentése, a tagfüggvény elő- és zárszó-

kódjának végrehajtása, majd a függvény visszatérési értékének mentése. Még ennél is több munkával járna, ha a függvény paramétereit menteni kellene a verembe. Sokkal kevesebb gépi kóddal járna, ha ezt írnánk:

```
string val = Obj._name;
```

Persze ilyet soha nem tennénk, hiszen annál több eszünk van, mintsem hogy nyilvános adattagokat hozzunk létre (lásd az 1. tippet). A JIT fordító megértő az eleganciára és a hatékonyságra vonatkozó igényeinkkel szemben, így behelyettesíti a tulajdonság-hozzáférőt. A JIT fordító akkor fejti ki helyben a tagfüggvényeket, amikor a sebességnövekedés vagy a méretcsökkenés indokolja a függvényhívás lecserélését a hívott függvény törzsére. A szabvány nem határozza meg pontosan a helyben kifejtés szabályait, így a jövőbeni megvalósítások eltérőek lehetnek, de a behelyettesítés nem is a mi feladatunk. A C# nyelvben nincs is olyan kulcsszó, amivel jelezhetnénk a fordítónak, hogy az adott tagfüggvényt célszerű helyben kifejteni. A C# fordító még a JIT fordítónak sem jelzi, hogy hol kell helyben kifejtést végeznie. Mindössze annyit tehetünk, hogy gondoskodunk róla, hogy a kódunk a lehető legvilágosabb legyen, megkönnyítve ezzel a JIT fordítónak a legjobb döntések meghozatalát. A jótanács talán már kezd ismerőssé válni: minél rövidebb egy tagfüggvény, annál jobb esély van a helyben kifejtésére. De ne feledjük, hogy bármennyire rövid legyen is egy függvény, ha virtuális, vagy ha `try/catch` blokkok vannak benne, akkor már nem lehet helyben kifejteni.

A helyben kifejtés módosítja azt az elvet, miszerint a kód JIT fordítása annak végrehajtása előtt történik meg. Vegyük még egyszer a `name` tulajdonság elérésének példáját:

```
string val = "Default Name";
if ( Obj != null )
  val = Obj.Name;
```

Ha a JIT fordító behelyettesíti a tulajdonság-hozzáférőt, akkor le kell fordítania a kódját, amikor az azt tartalmazó tagfüggvényt meghívjuk.

Nem a mi feladatunk meghatározni az algoritmusaink leghatékonyabb ábrázolását a gépi kód szintjén. Ezt a C# és a JIT fordító együttesen végzi el helyettünk. A C# fordító minden tagfüggvényhez előállítja a megfelelő IL kódot, a JIT fordító pedig ebből az IL kódból hozza létre a gépi kódot a célgépen. Felesleges túl sokat törnünk a fejünket azon, hogy a JIT pontosan milyen szabályok alapján végzi a dolgát, hiszen ezek úgyis fejlődnek az idő múltával, ahogy egyre fejlettebb algoritmusokat fejlesztenek ki. Ehelyett összpontosítsunk arra, hogy úgy fejezzük ki az algoritmusainkat, hogy a környezet eszközei a leghatékonyabban tudják kezelni a kódunkat. Szerencsére az ehhez szükséges szabályok megegyeznek a programfejlesztés általában követendő gyakorlataival. Tehát még egyszer: használjunk rövidebb és egyszerűbb függvényeket.

Ne feledjük, hogy a C# kódunk gépi kóddá alakítása két lépésben zajlik. A C# fordító IL kódot állít elő, amit szerelvények formájában szállítunk. A JIT fordító az igényeknek megfelelően fordítja le gépi kódra az egyes tagfüggvényeket (vagy azok csoportjait, ha helyben kifejtés történt). A rövid függvények megkönnyítik a JIT fordítónak, hogy a fordítással járó költségeket a lehető legkisebbre csökkentse. A rövidebb függvények helyben kifejtésére is nagyobb eséllyel kerül sor. Nem csak a rövidség, hanem a vezérlés egyszerűsége is számít. Minél kevesebb ág van a függvényben, annál könnyebben tudja regiszterezni a változókat a JIT fordító. A világos programozás nem csak követendő gyakorlat, de a hatékony futásidejű kód előállításának alapját is ez képezi.

32. tipp

Készítsünk kisebb, összetartó szerelvényeket

Ennek a tippnek inkább azt a címet kellett volna adni, hogy „Építsünk olyan szerelvényeket, amelyek pont megfelelő méretűek, és kevés nyilvános típust tartalmaznak". Ez azonban egy kissé terjengős lett volna, ezért ehelyett azt a hibát vettem alapul, amivel a leggyakrabban találkozom: ez pedig nem más, mint amikor a fejlesztők Mari néni tehenét is beleteszik a szerelvényekbe. Ezzel megnehezítik az összetevők újrahasznosítását, és a rendszer részenkénti frissítését. Sok kisebb szerelvénnyel az osztályaink könnyebben használhatók bináris összetevőként.

A cím az összetartás fontosságát is kiemeli. Az **összetartás** (kohézió, cohesion) annál nagyobb egy összetevőben, minél inkább egy egységet alkotnak a benne megvalósított feladatok. Az összetartó összetevőket általában egy egyszerű mondat segítségével le lehet írni. Számos példát láthatunk erre a .NET FCL szerelvényeinél: *„A `System.Collections` szerelvény adatszerkezeteket kínál az egymással kapcsolatban lévő objektumok tárolásához"*, illetve *„A `System.Windows.Forms` szerelvény Windows-vezérlőket modellező osztályokat tartalmaz."* A webes és a windowsos vezérlők külön szerelvénybe kerültek, mivel nincsenek szoros kapcsolatban egymással. A saját szerelvényeinket is le kell tudnunk írni egy a fentiekhez hasonló egyszerű mondattal. Csalni nem ér: *„A `MyApplication` szerelvényben minden szükséges dolog benne van."* Igen, ez is egy mondat, de elég semmirekellő darab, hiszen valószínűleg nem lesz szükségünk az összes dologra a `My2ndApplication` programban (bár egy részére viszont igen, és ezt a részt kell egy saját szerelvénybe költöztetnünk).

Ez persze nem azt jelenti, hogy egyetlen nyilvános osztályból álló szerelvényeket kell készítenünk. Mindig meg kell találnunk az arany középutat. Ha átesünk a ló túloldalára, és túl sok szerelvényt készítünk, azzal elveszítjük a betokozással járó előnyöket. Le kell mondanunk például a belső típusok előnyeiről, ha a kapcsolódó nyilvános osztályokat nem csomagoljuk be egyazon szerelvénybe (lásd a 33. tippet). A JIT fordító sokkal hatékonyabban tud helyben kifejtést végezni a szerelvényeken belül, mint több szerelvény között. Ez azt jelenti, hogy előnyünkre válik, ha az egymással kapcsolatos típusokat egy szerel-

vénybe tesszük. A cél az, hogy a legjobb méretű csomagban szállítsuk az összetevő által képviselt szolgáltatást. Ezt a célt összetartó összetevőkkel könnyebb elérni. Minden összetevő csak egy feladattal rendelkezzen.

Bizonyos értelemben a szerelvények az osztályok bináris megfelelői. Az osztályokat algoritmusok és adatok betokozására (egységbe zárására) használjuk. Kizárólag a nyilvános felület része a hivatalos megállapodásnak, tehát csak a nyilvános felületet látják a felhasználók. Ehhez hasonlóan a szerelvények egymással kapcsolatban lévő osztályok számára szolgálnak egy egységes bináris csomagként. A szerelvényen kívülről csak a nyilvános és a védett osztályok látszanak. A segédosztályok a szerelvények magánügyét képezhetik. Ezek valóban láthatóbbak, mint a privát beágyazott osztályok, de van egy módszerünk, amivel a szerelvényen belül megoszthatunk egy közös megvalósítást, anélkül, hogy ezt a megvalósítást az osztályaink összes felhasználója elérhetné. Az alkalmazásunkat több szerelvényre osztva betokozhatjuk az egymással kapcsolatos típusokat.

Másodszor, több szerelvényt használva a különböző telepítési műveletek is leegyszerűsödnek. Képzeljünk el egy három pillérre épülő alkalmazást, amelynek egyik része egy intelligens ügyfélprogram, egy másik része pedig egy kiszolgálón fut. Megadunk néhány adatellenőrző szabályt az ügyfélnél, hogy a felhasználók kapjanak valamilyen visszajelzést, amikor beviszik az adataikat. Ezeket a szabályokat a kiszolgálón lemásoljuk, és más szabályokkal kombináljuk őket a még pontosabb ellenőrzés elérése végett. Az üzleti szabályok összessége megtalálható a kiszolgálón, és csak egy részük van jelen minden ügyfélnél.

Persze megtehetnénk, hogy a forráskódot újrahasznosítva különböző szerelvényeket hozunk létre a kiszolgáló és az ügyfelek részére, de ezzel nehézkessé válna a terjesztési folyamat. Ekkor ugyanis két változatot kellene frissíteni, majd két külön telepítést is igénybe venne a szabályok módosítása. Jobb megoldás, ha elválasztjuk egymástól az ügyféloldali érvényesítést és az erősebb kiszolgálóoldali érvényesítést úgy, hogy külön szerelvényekbe helyezzük őket. Így szerelvényekbe csomagolt bináris objektumokat használunk fel újra,, ahelyett, hogy tárgykódot vagy forráskódot használnánk több helyen azzal, hogy több szerelvénybe is belefordítjuk ezeket.

A szerelvényeknek egymással összefüggő szolgáltatások gyűjteményét kell tartalmazniuk. Ezt így leírni persze könnyebb, mint a gyakorlatban megvalósítani. A valóság sokszor az, hogy nem tudjuk előre, mely osztályokra lesz majd szükség a kiszolgálón és az ügyfeleken is egy elosztott alkalmazásnál. Ennél is valószínűbb, hogy a kiszolgáló-, illetve ügyféloldali szolgáltatások közötti határ elmosódhat, miközben egyik helyről a másikra tolunk át különféle szolgáltatásokat. Ha kisebb méretű szerelvényekkel dolgozunk, valószínűleg könnyebben tudjuk megoldani a szerelvények újraelosztását az ügyfelek és a kiszolgáló között. A szerelvények az alkalmazásunk bináris építőkövei. Segítségükkel könnyebben adhatunk egy új összetevőt egy már működő alkalmazáshoz. Ha már egyszer hibázunk, akkor inkább az legyen a gondunk, hogy túl sok apró szerelvényt készítettünk, mintsem a túl kevés óriási szerelvényen bosszankodjunk.

Gyakran szoktam a Lego kockákhoz hasonlítani a szerelvényeket és a bináris összetevőket. Az apró kockákat könnyen kivehetjük az építményünkből, és más kockákat tehetünk a helyükre. Ezt a szerelvényeknél is meg kell tudnunk tenni. Kivesszük az egyiket, és máris betehetünk a helyére egy másikat, amelyiknek a felülete megegyezik az előbbiével. Az alkalmazás többi része továbbra is úgy működik, mintha mi sem történt volna. Vigyük tovább egy kicsit a Lego-hasonlatunkat. Ha a paraméterek és a visszatérési értékek mind felületek, akkor bármelyik szerelvényt le lehet cserélni egy másik szerelvényre, amelyik megvalósítja ugyanazokat a felületeket (lásd a 19. tippet).

A kis szerelvényekkel az alkalmazás indításakor jelentkező költségeket is csökkenthetjük. Minél nagyobb egy szerelvény, annál többet dolgozik a processzor, miközben betölti a szerelvényt, és a szükséges IL kódból előállítja a gépi kódot. Bár csak az indításhoz szükséges eljárásokat fordítja le a JIT fordító, a teljes szerelvény betöltődik, és a CLR a szerelvény összes tagfüggvényéhez elkészít egy beillesztő eljárást (csonkot, stub).

Itt az idő, hogy lassítsunk egy kicsit, nehogy átessünk a ló túloldalára. Ennek a tippnek az a fő üzenete, hogy a programjaink ne egyetlen irdatlan nagy darabból álljanak, hanem újrahasznosítható bináris összetevőkből építsük fel őket. Figyelnünk kell azonban, hogy a tanács hatására ne essünk túlzásokba. A túl sok apró szerelvényre épülő programoknak is megvan a maguk hátránya. Amikor a programvezérlés átszeli a szerelvények határait, az mindig teljesítménycsökkenéssel jár. A CLR betöltőnek kicsit több munkájába kerül a sok szerelvény betöltése és az IL kód gépi kóddá alakítása, különösen a függvénycímek feloldásánál.

A szerelvények határainak átlépésekor további biztonsági ellenőrzéseket is végrehajt a környezet. Egyazon szerelvényből származó összes kód ugyanazt a bizalmat élvezi (ez nem feltétlenül jelent azonos jogosultságokat, csak azonos biztonsági szintet). Ha a program futása közben átlépjük a szerelvények között húzódó határvonalakat, akkor a CLR mindig végrehajt néhány biztonsági ellenőrzést. Minél kevesebbszer halad át a vezérlés a szerelvények közti határokon, annál hatékonyabb lesz a programunk.

Nem szabad hagynunk azonban, hogy ezek a teljesítménnyel kapcsolatos aggodalmak eltántorítsanak minket attól a szándékunktól, hogy a túlzottan nagy szerelvényeket apróbb darabokra tördeljük. Az ezzel járó teljesítményveszteség elhanyagolható. A C# nyelvet és a .NET környezetet az összetevőkre gondolva alkották meg, és a nagyobb rugalmasság általában kifizetődőbb.

Most akkor hogyan döntsük el, hogy mennyi kód vagy hány osztály kerüljön egy szerelvénybe? De ami ennél is fontosabb, hogyan döntsük el, hogy mely kódrészletek kerüljenek egyazon szerelvénybe? Ez nagyban az adott alkalmazástól függ, ezért nincs rá egyszerű válasz. A tanácsom a következő: mindig kezdjük el végignézni az összes nyilvános osztályunkat. A nyilvános osztályokat a közös alaposztályokkal kombinálva tegyük szerelvényekbe. Ezután a nyilvános osztályok működését szolgáló segédosztályokat tegyük ugyanebbe a szerelvénybe. A kapcsolódó nyilvános felületeket csomagoljuk a saját szerelvé-

nyükbe. Utolsó lépésként válogassuk ki azokat az osztályokat, amelyeket az alkalmazásunk egésze használ. Ezekből alkothatunk egy külön segédszerelvényt, ami az alkalmazásunk segédfüggvényeit tartalmazó könyvtárnak felel meg.

A végeredmény az lesz, hogy létrehozunk összetevőket, amelyek egymással összefüggő nyilvános osztályokból és azokat támogató segédosztályokból állnak. Ezzel a létrehozott szerelvény elég kicsi lesz ahhoz, hogy általa kihasználhassuk a könnyű frissítés és az újrahasznosíthatóság előnyeit, miközben minimálisra csökkentjük a sok szerelvény használatából származó teljesítménycsökkenést. A jól átgondolt, összetartó összetevőket egyetlen egyszerű mondattal le lehet írni. Az a meghatározás például, hogy *„A* `Common.Storage.dll` *a nem valósidejű (offline) adatgyorstárat és a felhasználói beállítások összességét kezeli."* egy alacsony összetartású összetevőre utal. Ehelyett két összetevőt kell készítenünk: *„A* `Common.Data.dll` *a nem valósidejű gyorstárat kezeli. A* `Common.Settings.dll` *a felhasználói beállításokat kezeli."* A feldarabolásnál rájöhetünk, hogy szükségünk lesz egy harmadik összetevőre is: *„A* `Common.EncryptedStorage.dll` *a titkosított helyi tárolók fájlrendszerével kapcsolatos IO műveleteket kezeli."* Ezután a három összetevő bármelyikét a többitől függetlenül frissíthetjük.

Hogy mi számít *kicsinek*, az attól függ. Az `mscorlib.dll` durván 2 MB, a `System.Web.RegularExpressions.dll` viszont csupán 56 KB. Ennek ellenére mindkettő megfelel annak a követelménynek, hogy minél kisebb, újrahasznosítható szerelvényeket hozzunk létre. Ez azért van, mert mindegyik egymással szorosan összefüggő osztályokból és felületekből áll. Az abszolút méretbeli különbség a megvalósított szolgáltatásban keresendő. Az `mscorlib.dll` az összes olyan alacsonyszintű osztályt tartalmazza, amire minden alkalmazásnak szüksége van.
A `System.Web.RegularExpressions.dll` kimondottan szűk feladatkört lát el. Csak a webes vezérlőkben használt szabályos kifejezések támogatásához szükséges osztályokat tartalmazza. Mindkét fajta összetevővel lesz majd dolgunk. Készíteni fogunk kicsi, egy feladat megvalósítására kihegyezett összetevőt, és nagyobb, gyakran használt szolgáltatásokat megvalósító összetevőket is. Mindkét esetben mindig törekedjünk arra, hogy a kész összetevő mérete az ésszerűség határain belül a lehető legkisebb legyen.

33. tipp

Korlátozzuk a típusok láthatóságát

Nem kell mindenkinek mindent látnia. Nem kell mindegyik típusunknak nyilvánosnak lennie. Minden típusnak csak annyi láthatóságot kell adnunk, amennyi feltétlenül szükséges a céljaink eléréséhez. Ez sokszor sokkal kisebb láthatóságot jelent, mint gondolnánk. A belső vagy privát osztályok megvalósíthatnak nyilvános felületeket. A privát típusban meghatározott nyilvános felületekben található szolgáltatásokhoz az összes ügyfél hozzáférhet.

Kezdjük mindjárt ennek alapvető előidézőjével, ami nem más, mint a hatékony eszközök és a lusta fejlesztők keveréke. A VS .NET nagyszerű termelőeszköz. Vagy ezt, vagy a C# Builder környezetet használom minden munkámhoz, egyszerűen azért, mert többet tudok elvégezni rövidebb idő alatt. Az egyik teljesítménynövelő bővítés segítségével két gombnyomással létre lehet hozni egy új osztályt. Hát még ha pontosan az is jönne létre, amit szerettem volna. A VS .NET által létrehozott osztály így néz ki:

```
public class Class2
{
  public Class2()
  {
    //
    // TODO: Add constructor logic here
    //
  }
}
```

Ez egy nyilvános osztály. Minden olyan kód látni fogja, ami felhasználja azt a szerelvényt, amin éppen dolgozom. Ez általában túl nagy láthatóság. Az általunk készített önálló osztályok közül soknak belső osztálynak kellene lennie. A láthatóságot úgy is korlátozhatjuk, hogy az eredeti osztály belsejébe védett vagy privát osztályokat ágyazunk be. Minél kisebb a láthatóság, annál kevésbé kell változtatni a teljes rendszeren egy későbbi frissítés során. Minél kevesebb helyről lehet hozzáférni egy kódrészlethez, annál kevesebb helyen kell módosítanunk a programunkat a frissítéskor.

Csak azt tegyük elérhetővé, aminek elérhetőnek kell lenni. Próbáljuk meg kevésbé látható osztályokkal megvalósítani a nyilvános felületeket. Példákat láthatunk erre a .NET keretrendszer különböző részeiben visszaköszönő `Enumerator` (Felsoroló) minta megvalósításában. A `System.ArrayList` például egy privát `ArrayListEnumerator` nevű osztályt tartalmaz, ami az `IEnumerator` felületet valósítja meg:

```
// Példa, nem teljes forráskód
public class ArrayList: IEnumerable
{
  private class ArraylistEnumerator : IEnumerator
  {
    // Ez tartalmazza a MoveNext(), a Reset() és a Current
    // konkrét megvalósítását
  }

  public IEnumerator GetEnumerator()
  {
    return new ArrayListEnumerator( this );
  }

// egyéb ArrayList tagok
}
```

Az ügyfélkódnak, amit mi írunk meg, nem kell tudnia az ArrayListEnumerator osztály létezéséről. Mindössze annyit kell tudnunk, hogy kapunk egy osztályt, ami megvalósítja az IEnumerator felületet, amikor meghívjuk a GetEnumerator függvényt egy ArrayList objektumra. A tényleges típus csak a megvalósítással kapcsolatos részletkérdés. A .NET keretrendszer tervezői ugyanezt a mintát követték a többi gyűjteményosztálynál is. A Hashtable osztályban van egy HashtableEnumerator, a Queue osztályban egy QueueEnumerator, és így tovább. Számos előny származik abból, hogy a felsoroló osztály privát. Először is, az ArrayList osztály teljes egészében helyettesítheti az IEnumerator felületet megvalósító típust, és erről nekünk mit sem kell tudnunk. Minden működik, ahogy kell. A felsoroló osztálynak még CLS-megfelelőnek sem kell lennie, mivel nem nyilvános (lásd a 30. tippet), nyilvános felülete pedig megfelel a CLS-nek. A felsorolást anélkül használhatjuk, hogy részletesen ismernénk az őt megvalósító osztályt.

A belső osztályok készítése a típusok hatókörének korlátozására gyakran méltatlanul mellőzött módszer. A legtöbb programozó nyilvános osztályokat hoz létre anélkül, hogy átgondolná az egyéb lehetőségeket. Ez a VS .NET varázsló átka. Ahelyett hogy gondolkodás nélkül elfogadnánk az alapbeállításokat, mindig célszerű pontosan átgondolni, hogy az új típust hol fogjuk használni. Minden ügyfélnek szüksége lesz rá, vagy elsősorban az adott szerelvény belsejében lesz rá szükség?

Ha a szolgáltatásokat felületekkel tesszük elérhetővé a külvilág számára, akkor lehetővé válik a belső osztályok egyszerűbb létrehozása anélkül, hogy ezzel veszítenének a szerelvényen kívüli használhatóságukból (lásd a 19. tippet). Szükség van-e arra, hogy a típus nyilvános legyen, vagy felületek együttesével jobban leírható a működése? Belső osztályokat használva lecserélhetünk egy osztályt egy újabb változattal, ha az új változat is ugyanazokat a felületeket valósítja meg, mint a régi. Példaként vegyünk egy telefonszámokat ellenőrző osztályt:

```
public class PhoneValidator
{
  public bool ValidateNumber( PhoneNumber ph )
  {
    // ellenőrzés végrehajtása
    // Érvényes körzetszám, előhívószám ellenőrzése
    return true;
  }
}
```

Telnek a hónapok, és az osztály nagyszerűen működik. Aztán befut egy igény, hogy ezentúl a nemzetközi számokat is kezelni tudjuk. A PhoneValidator előző változata már nem jó, mert azt csak belföldi hívások kezelésére terveztük. A belföldi hívásokat ellenőrző változatra továbbra is szükség van, de egy helyen a nemzetközi számok ellenőrzése is

szükséges. Ahelyett, hogy ezt a további szolgáltatást begyömöszölnénk ebbe az osztályba, jobban járunk, ha csökkentjük az átfedést a különböző elemek között. Létrehozunk egy felületet, amely bármely telefonszám ellenőrzésére képes:

```
public interface IPhoneValidator
{
  bool ValidateNumber( PhoneNumber ph );
}
```

Ezután módosítjuk a létező telefonszám-ellenőrző osztályunkat, hogy az megvalósítsa a felületet, és belső osztály legyen:

```
internal class USPhoneValidator : IPhoneValidator
{
  public bool ValidateNumber( PhoneNumber ph )
  {
    // ellenőrzés végrehajtása
    // Érvényes körzetszám, előhívószám ellenőrzése
    return true;
  }
}
```

Utolsó lépésként pedig létrehozhatunk egy osztályt a nemzetközi telefonszámok érvényességének vizsgálatához:

```
internal class InternationalPhoneValidator : IPhoneValidator
{
  public bool ValidateNumber( PhoneNumber ph )
  {
    // ellenőrzés végrehajtása
    // nemzetközi hívószám ellenőrzése
    // konkrét hívószámszabályok ellenőrzése
    return true;
  }
}
```

A megvalósítás befejezéséhez létre kell hoznunk a megfelelő osztályt a telefonszám típusa alapján. Erre a célra használhatjuk a Gyár mintát (Factory pattern). A szerelvényen kívülről csak a felület látszik. Az osztályok, amelyek a világ különböző régióiban máshogy néznek ki, csak az osztályon belülről láthatók. A különböző régiókhoz különböző ellenőrző osztályokat készíthetünk, anélkül, hogy a rendszerben lévő többi szerelvényt megzavarnánk. Az osztályok láthatóságának korlátozásával csökkentettük annak a kódnak a mennyiségét is, amit a rendszer frissítéséhez, illetve bővítéséhez módosítanunk kell.

Azt is megtehetnénk, hogy egy nyilvános elvont alaposztályt készítünk a `PhoneValidator` feladatainak ellátására, ami tartalmazhatná a közös megvalósítás algoritmusait. A fogyasztók aztán a nyilvános szolgáltatásokat a hozzáférhető alaposztályon keresztül érhetnék el. Ebben

a példában személy szerint a nyilvános felületek használatát tartom jobb megoldásnak, mivel itt nem sok megosztott szolgáltatásról beszélhetünk. Más esetekben jobb szolgálatot tehetnek a nyilvános elvont alaposztályok. Bármelyik megvalósítást válasszuk is, mindkét esetben kisebb lesz a nyilvánosan elérhető osztályok száma.

A külvilág számára elérhetővé tett osztályok és felületek egy megállapodás részei, aminek meg kell felelnünk. Minél bonyolultabb ez a felület, annál jobban beszűkülnek a jövőbeni lehetőségeink. Minél kevesebb nyilvános típust teszünk elérhetővé, annál több lehetőségünk marad az egyes megvalósítások módosítására, illetve bővítésére.

34. tipp

Ne aprózzuk el a webes API-kat

Egy kommunikációs protokoll alkalmazásának költségei és kényelmi színvonala határozzák meg azt, hogy miként használjuk az adott csatornát. Másképpen kommunikálunk telefonon, faxon, postán és elektronikus leveleken keresztül. Emlékezzünk vissza arra, amikor utoljára rendeltünk egy katalógusból. Ha telefonon rendelünk, akkor egyfajta kérdezz-felelek játékba bonyolódunk az eladóval:

„Melyik áru legyen az első?"
„Az 123-456 számú."
„Hány darabot küldhetünk belőle?"
„Hármat."

Ez a párbeszéd egészen addig folytatódik, amíg az eladó fel nem vette a teljes rendelésünket, a számlázási címünket, a hitelkártyánk adatait, a szállítási címet, meg a többi információt, ami a tranzakció sikeres lebonyolításához szükséges. Ez az oda-vissza párbeszéd megnyugtató a telefonon keresztül. Soha nem kell hosszú monológokat előadnunk anélkül, hogy kapnánk valamilyen visszajelzést. Nehezen viseljük, ha sokáig csend van a vonal másik oldalán, és azon kell tűnődnünk, hogy vajon ott van-e még az eladó.

Vessük ezt össze azzal, amikor faxon rendelünk. Kitöltjük a teljes dokumentumot, majd ha kész, elfaxoljuk a cégnek. Egy dokumentum, egy ügylet. Nem azt csináljuk, hogy kitöltjük az egyik termékre vonatkozó igényünket, elfaxoljuk, beírjuk a címünket, megint elfaxoljuk, beírjuk a hitelkártyánk számát, aztán megint faxolunk.

Ez a példa jól szemlélteti a silányan tervezett webes tagfüggvényfelületek buktatóit. Mindegy, hogy egy webszolgáltatást vagy a .NET Remoting (távelérési) rendszert használjuk, soha nem szabad megfeledkeznünk arról, hogy a művelet legköltségesebb része mindig az, amikor objektumokat kell átvinnünk két egymástól távol lévő gép között. Fel kell hagynunk azzal a szokással, hogy a távoli eléréshez olyan API-kat használunk, amelyek csupán a helyben használt felületeink újracsomagolt változatai. Így is működik a dolog, de lerí ró-

la, hogy mennyire nem hatékony. Olyan ez, mintha a telefonos módszert követve rendelnénk faxon. Az alkalmazás minden alkalommal a hálózatra várakozik, amikor üzenetváltással átküldjük az új információdarabkát a vezetéken. Minél jobban elaprózzuk az API-t, annál több idejét tölti az alkalmazásunk arra várva, hogy választ kapjon a kiszolgálótól.

Készítsünk inkább olyan web alapú felületeket, amelyek a dokumentumok vagy objektumhalmazok rendezettségén alapulnak. A távolsági kommunikációnak úgy kell működnie, mint amikor faxon küldjük el a rendelésünket a csomagküldő cégnek. Az a jó, ha az ügyfélgép hosszú időn keresztül képes végezni a dolgát anélkül, hogy kapcsolatba kellene lépnie a kiszolgálóval. Aztán amikor minden információ megvan ahhoz, hogy létrejöhessen a tranzakció, az ügyfél a teljes dokumentumot elküldheti a kiszolgálónak. A kiszolgáló válaszának is hasonló elvet kell követnie. Amikor a kiszolgáló adatokat küld az ügyfélnek, az ügyfélnek egyben meg kell kapnia az összes olyan információt, amire szüksége van a feladatainak ellátásához.

A rendelési hasonlatnál maradva, tervezzünk egy megrendelés-feldolgozó rendszert, ami egy központi kiszolgálóból, és asztali ügyfelekből áll, amelyek webszolgáltatásokon keresztül kapják meg a szükséges információkat. A rendszer egyik osztálya a megrendelő osztály lesz. Ha a szállítással kapcsolatos kérdéseket most mellőzzük, akkor a megrendelő osztály valahogy így fog kinézni, megengedve, hogy az ügyfélkód kiolvassa és módosítsa a nevet, a szállítási címet, és az ügyfélszámlát:

```
public class Customer
{
  public Customer( )
  {
  }

  // A megrendelő mezőinek elérésére és módosítására
  // szolgáló tulajdonságok:
  public string Name
  {
    // a get és a set részleteit elhagytuk
  }

  public Address shippingAddr
  {
    // a get és a set részleteit elhagytuk
  }

  public Account creditCardInfo
  {
    // a get és a set részleteit elhagytuk
  }
}
```

A megrendelő osztály API-ja egyáltalán nem olyan, mint amit a távoli hívásokhoz használnunk kellene. Egy távoli megrendelő meghívása óriási forgalmat eredményez a kiszolgáló és az ügyfél között:

```
// a megrendelő létrehozása a kiszolgálón
Customer c = new Server.Customer( );
// üzenetváltás a név beállításához
c.Name = dlg.Name.Text;
// üzenetváltás a cím beállításához
c.shippingAddr = dlg.Addr;
// üzenetváltás a hitelkártya beállításához
c.creditCardInfo = dlg.credit;
```

Ehelyett célszerűbb elkészíteni helyben egy `Customer` objektumot, majd az összes mező beállítása után elküldeni a kész objektumot a kiszolgálónak:

```
// megrendelő létrehozása az ügyfélen
Customer c = new Customer( );
// helyi másolat beállítása
c.Name = dlg.Name.Text;
// a helyi cím beállítása
c.shippingAddr = dlg.Addr;
// a helyi hitelkártya beállítása
c.creditCardInfo = dlg.credit;
// a kész objektum elküldése a kiszolgálónak (egy menetben)
Server.AddCustomer( c );
```

A megrendelős példa egy kézenfekvő és egyszerű tanmese, amelynek az a tanulsága, hogy teljes objektumokat érdemes küldenünk a kiszolgáló és az ügyfelek között. A hatékony programok írásához azonban tovább kell vinnünk ezt a gondolatmenetet, és az egymással összefüggő objektumokra kell alkalmaznunk. Rendkívül elaprózzuk a dolgokat, ha távoli hívásokat kezdeményezünk csupán azért, hogy beállítsuk egy objektum egyetlen tulajdonságát. De az is előfordulhat, hogy még egy teljes megrendelő is túl apró egység a kiszolgáló és az ügyfél közti adatcseréhez.

Ahhoz, hogy a példánkat kiterjeszthessük olyan életszerűbb feladatokra, amelyekkel a programozás során találkozunk, fogalmazzunk meg néhány további feltevést a rendszerünkkel kapcsolatban. Tegyük fel, hogy ez a programrendszer egy nagy, 1 milliós ügyfélkörrel rendelkező internetes csomagküldő cég munkáját segíti. Képzeljük el, hogy egy nagy cégről van szó, és minden vásárló átlagosan 15 megrendelést adott le a tavalyi év során. Minden telefonos ügyfélszolgálati munkatárs egy gépet használ a munkaideje alatt, és minden alkalommal megrendelői rekordokat kell kikeresnie vagy módosítania, amikor fogad egy telefonhívást. A feladatunk egy olyan szerkezet megalkotása, amelyben a leghatékonyabban lehet az objektumokat ide-oda küldeni az ügyfélgépek és a kiszolgáló között.

Kezdésképpen kizárhatunk néhány nyilvánvalóan rossz döntést. Arról mindenképpen le kell mondanunk, hogy minden egyes megrendelő és megrendelés adatait lehívjuk minden telefonhívás alkalmával. 1 millió megrendelő és 15 millió megrendelés adatai túl nagy terhet jelentenének, ha mindig mindet át akarnánk vinni az ügyfélgépekre. Ezzel csak azt érjük el, hogy a szűk keresztmetszetet egyik helyről áttesszük egy másikra. Ahelyett, hogy állandóan bombáznánk a kiszolgálót minden egyes apró változtatással, csak oda jutunk, hogy több mint 15 millió objektumot kérünk tőle. Igaz, így csak egy tranzakcióról van szó, de az aztán egyáltalán nem nevezhető hatékonynak.

Ehelyett próbáljuk meg lehívni az objektumoknak azt a halmazát, amelyik a legjobb megközelítését adja annak az adathalmaznak, amelyre a képzeletbeli ügyfélszolgálatosunknak a következő néhány percben szüksége lesz. A kolléga felveszi a telefont, és egy adott ügyféllel fog társalogni. A telefonhívás ideje alatt megrendeléseket töröl vagy vesz fel, módosítja azokat, esetleg módosítja az ügyfélszámla adatait. A kézenfekvő választás tehát, ha lehívjuk az adott vásárló adatait az összes általa leadott rendeléssel együtt. A kiszolgálón található tagfüggvény valahogy így nézne ki:

```
public OrderData FindOrders( string customerName )
{
  // Kikeressük a megrendelőt a neve alapján
  // Kikeressük a megrendelő összes rendelését
}
```

De vajon tényleg jó ez így? Azokra a megrendelésekre nagy valószínűséggel nem lesz szükség az ügyfélgépen, amelyeket már elküldtek, és a megrendelő át is vett. Jobb döntésnek tűnik, ha csak a megrendelő le nem zárt megrendeléseit hívjuk le. Ekkor a kiszolgálón található tagfüggvény valami ilyenre változna:

```
public OrderData FindOpenOrders( string customerName )
{
  // Kikeressük a megrendelőt a neve alapján
  // Kikeressük a megrendelő összes rendelését
  // Kiszűrjük azokat, amelyeket
  // már átvett
}
```

Ezzel még mindig arra kényszerítjük az ügyfélgépet, hogy minden egyes telefonhívásnál új kéréssel álljon elő. Van-e valami más lehetőségünk is arra, hogy optimalizáljuk a kommunikációs csatorna kihasználását azon túl, hogy a rendeléseket is hozzávesszük a megrendelőhöz? Bővítsük ki még néhány feltevéssel az üzleti folyamatot, hátha ettől támad még néhány ötletünk. Tegyük fel, hogy a központban a munkát úgy osztották fel, hogy minden csapat csak egy adott körzetből fogad hívásokat. Így már módosíthatjuk a program szerkezetét, hogy a kommunikációs folyamat sokkal hatékonyabb legyen.

A műszak elején minden ügyfélszolgálatos kolléga lehívja az adott körzethez tartozó friss megrendelői és rendelési adatokat. Az ügyfélprogram minden telefonhívás után elküldi a módosított adatokat a kiszolgálónak, mire a kiszolgáló visszaküldi az összes olyan módosítást, ami azóta következett be, hogy az adott ügyfélgép utoljára a kiszolgálóhoz fordult. Ennek az lesz az eredménye, hogy a kolléga minden hívás után elküldi az elvégzett módosításokat, és cserébe megkapja az összes módosítást, amit az adott csapat többi tagja végzett el az adatokon. Ez a programszerkezet azt jelenti, hogy telefonhívásonként csak egy tranzakcióra van szükség, és minden ügyfélszolgálatos számára rendelkezésre állnak a szükséges adatok a telefonhívások fogadásához. A kiszolgálón immár két tagfüggvény van, amelyek valahogy így néznek ki:

```
public CustomerSet RetrieveCustomerData(
  AreaCode theAreaCode )
{
  // Kikeressük az összes, az adott körzethez tartozó megrendelőt
  // Az adott körzet összes megrendelőjéhez:
   // Kikeressük a megrendelő összes rendelését
   // Kiszűrjük azokat, amelyeket
   // már átvett
  // Visszaadjuk az eredményt
}

public CustomerSet UpdateCustomer( CustomerData
  updates, DataTime lastUpdate, AreaCode theAreaCode )
{
  // Először mentjük a frissítéseket, minden frissítést megjelölve az
  // aktuális idővel

  // Ezután beolvassuk a frissítéseket:
  // Kikeressük az összes, az adott körzethez tartozó megrendelőt
  // Az adott körzet összes megrendelőjéhez:
   // Kikeressük a megrendelő összes olyan rendelését, amelyet az utolsó
   // mentés óta frissítettünk. Ezeket hozzáadjuk az eredményhalmazhoz
  // Visszaadjuk az eredményt
}
```

Előfordulhat, hogy még így is pazarlóan bánunk a sávszélességgel. Az előző minta akkor működik a legjobban, ha minden ismert megrendelő minden nap telefonál. Ez valószínűleg nem így van. Ha igen, akkor olyan gondok vannak a cég terméktámogatásával, hogy az már kívül esik a hatékony programtervezés hatókörén.

Hogyan lehetne tovább csökkenteni a tranzakciók során átadott adatmennyiséget anélkül, hogy növelnénk a tranzakciók számát, vagy a módosítás az ügyfélszolgálat reakcióidejének rovására menne? Például bizonyos feltevéseket fogalmazhatunk meg azzal kapcsolatban, hogy mely megrendelők fognak a legvalószínűbben telefonálni. Megnézünk néhány statisztikát, és látjuk, hogy ha egy vásárló hat hónapja nem adott le megrendelést, akkor

valószínűleg már nem is fog, ezért ezeknek a megrendelőknek az adatait már nem kell lehívnunk a műszak elején. Ezzel csökkentettük az első tranzakció méretét. Rájövünk arra is, hogy azok a vásárlók, akik röviddel egy megrendelés leadása után telefonálnak, általában a legutóbbi rendelésükkel kapcsolatban érdeklődnek. Tehát úgy módosítjuk a megrendelési listát, amit a kiszolgáló elküld az ügyfeleknek, hogy az csak a megrendelők legutolsó és nem az összes rendelését tartalmazza. Ez nem változtat a kiszolgáló tagfüggvényeinek aláírásán, de csökkenthetjük vele az ügyfélnek visszaküldött csomagok méretét.

Ennek az eszmefuttatásnak az volt a célja, hogy elgondolkozzunk a távoli gépek közti kommunikáció folyamatán. A cél mindig az, hogy csökkentsük a gépek közti tranzakciók gyakoriságát és az elküldött adatmennyiséget is. Ez két homlokegyenest ellenkező cél, így valamiféle kompromisszumot kell kötnünk. Valahol a két szélsőség között félúton kell megállapodnunk, de ha muszáj, akkor inkább a kisebb számú, de több adatot mozgató tranzakciót használó felépítés felé csússzunk el.

5

A keretrendszer használata

Martin Shoemaker kollégám, aki egyben jó barátom is, egy nagyszerű fórumot működtet „Biztos, hogy újra meg kell írnunk a .NET kódot?" címmel. A válasz remélhetőleg az, hogy nem. Mindig használjuk fel az összes rendelkezésünkre álló eszközt ahelyett, hogy újra megírnánk azt a már létező kódot, ami csak arra vár, hogy használatba vegyük.

A .NET keretrendszer igen gazdag osztálykönyvtár. Minél jobban megismerjük a keretrendszert, annál kevesebb kódot kell megírnunk magunknak. Ebben a fejezetben olyan gyakran alkalmazott megoldásokat mutatunk be, amelyekkel a legtöbbet hozhatjuk ki a .NET keretrendszerből. Lesznek itt olyan tippek is, amelyek segítenek dönteni, ha a keretrendszer egyszerre több lehetőséget is nyújt egy adott probléma megoldására. Az osztályainkat és az algoritmusainkat úgy is megírhatjuk, hogy felhasználjuk azt, ami már ott van a keretrendszerben, ahelyett, hogy az árral szemben úsznánk. Ennek a fejezetnek a tippjeiben látni fogjuk azokat az algoritmusokat és osztályokat, amelyeket a fejlesztők önfejűen újra és újra megírnak, pedig könnyebb lenne a dolguk, ha kihasználnák a .NET keretrendszer adottságait. Előfordul, hogy ezt azért teszik, mert a keretrendszer nem pont azt teszi, amit szeretnének. Meg fogom mutatni, hogy ezekben az esetekben hogyan bővíthetjük ki a rendszer alapjait. Máskor azért tesznek ilyet, mert túlságosan el vannak foglalva a teljesítménnyel.

Hiába van ott a sok-sok eszköz a keretrendszerben, a fejlesztők szeretik feltalálni a spanyolviaszt. Egy jótanács: soha ne írjunk olyan kódot, amit valaki már megírt helyettünk!

35. tipp

Eseménykezelők helyett használjunk felülbírált tagfüggvényeket

Sok olyan .NET osztály van, amelyik kétféleképpen tudja kezelni a rendszer eseményeit. Megadhatunk egy eseménykezelőt, vagy felülbírálhatunk egy virtuális függvényt az alaposztályban. De mi a fittyfenének kell kétféle módszer ugyanarra? Hát azért, mert a különböző

böző helyzetek egymástól eltérő bánásmódot kívánnak. A származtatott osztályokon belül mindig a virtuális tagfüggvényt bíráljuk felül. Az eseménykezelők használatát korlátozzuk az egymástól független objektumok eseményeinek kezelésére.

Tegyük fel, hogy írunk egy pofás kis Windows alkalmazást, aminek reagálnia kell az egér gombjának lenyomására. Az ablak (form) osztályban dönthetünk úgy, hogy felülbíráljuk az `OnMouseDown()` tagfüggvényt:

```
public class MyForm : Form
{

  // A többi kód elhagyva

  protected override void OnMouseDown(
    MouseEventArgs e )
  {
    try {
      HandleMouseDown( e );
    } catch ( Exception e1 )
    {
      // ide jön a tényleges hibakezelő kód
    }
    // *majdnem mindig* meghívjuk az alaposztályt, hogy
    // a többi eseménykezelő feldolgozhassa az üzenetet
    // Az osztályunk felhasználói számítanak erre
    base.OnMouseDown( e );
  }
}
```

Vagy éppen megadhatunk egy eseménykezelőt is:

```
public class MyForm : Form
{

  // A többi kód elhagyva

  public MyForm( )
  {
    this.MouseDown += new
      MouseEventHandler( this.MouseDownHandler );
  }

  private void MouseDownHandler( object sender,
    MouseEventArgs e )
  {
    try {
      HandleMouseDown( e );
```

```
      } catch ( Exception e1 )
      {
        // ide jön a tényleges hibakezelő kód.
      }
    }
}
```

Az első megoldás célravezetőbb. Ha egy eseménykezelő kivált egy kivételt, akkor a többi eseménykezelőt az adott láncolatban már nem hívja meg a program (lásd a 21. tippet). Egy másik, hibásan megírt kód megakadályozza a rendszert abban, hogy meghívja az eseménykezelőnket. A virtuális tagfüggvényt felülbírálva előbb a mi eseménykezelőnk fut le. A virtuális függvény alaposztályban található változata felel azért, hogy a program meghívja az eseményhez tartozó többi eseménykezelőt. Ez azt jelenti, hogy ha szeretnénk (márpedig általában szeretnénk), hogy a program meghívja az eseménykezelőket, akkor meg kell hívnunk az alaposztályt. Néhány ritka esetben előfordulhat, hogy meg akarjuk változtatni az alapértelmezett viselkedést, és nem hívjuk meg az alaposztályban található változatot, mert nem szeretnénk, hogy lefussanak az eseménykezelők. Arra ugyan semmilyen biztosíték nincs, hogy az összes eseménykezelő le fog futni, hiszen egy hibásan megírt eseménykezelő kiválthat egy kivételt, de abban biztosak lehetünk, hogy a származtatott osztályunk helyesen fog működni.

A felülbírált függvény használata sokkal hatékonyabb, mint egy eseménykezelő megadása. A 22. tippben már bemutattam, hogy a `System.Windows.Forms.Control` osztály hogyan tárolja és képezi le az eseménykezelőket egy adott eseményre, egy kifinomult gyűjteményeket használó módszerrel. Az eseménykezelőt használó megoldás több munkát jelent a processzor számára, mert meg kell vizsgálnia, hogy az eseményhez tartozik-e eseménykezelő. Ha igen, akkor végig kell mennie a teljes híváslistán. Az eseményhez tartozó lista minden egyes tagfüggvényét meg kell hívni. Annak eldöntése, hogy vannak-e megfelelő eseménykezelők, majd végigmenni az összes függvényen, sokkal tovább tart, mint meghívni egyetlen virtuális függvényt.

Ha ez még mindig nem győzött volna meg minket, akkor nézzük meg még egyszer a tipp első kódpéldáját. Melyik változat a világosabb? Ha módosítanunk kell az ablakosztályt, akkor a virtuális függvény felülbírálásánál csak egyetlen függvényt kell megvizsgálnunk. Az eseménykezelős megoldásnál két pontra is oda kell figyelnünk: az eseménykezelő függvényre, és arra a kódra is, amelyik továbbküldi az eseményt. Mindkettő okozhat gondokat. Ha csak egy függvény van, az sokkal egyszerűbb.

Rendben, szóval sorolom itt a különféle okokat, hogy miért használjuk inkább a felülbírált függvényeket és ne az eseménykezelőket. De ugye a .NET keretrendszer tervezői biztosan nem potyára hozták létre az eseménykezelőket? Persze, hogy nem. Hozzánk hasonlóan, nekik is van jobb dolguk annál, mintsem hogy olyan kódok írásával töltsék az idejüket, amiket aztán senki sem használ. A felülbírált függvények a származtatott osztályoknak va-

lók. A többi osztálynak az eseménykezelő rendszert kell használnia. Gyakran megadunk például gombnyomáskezelőket az ablakokban. Az eseményt a gomb állítja elő, de az ablakobjektum kezeli. Meghatározhatnánk egy egyedi gombot, és felülírhatnánk ebben az osztályban a kattintás kezelőjét, de ez túl sok munka egyetlen esemény kezelésének megoldásához. Ezzel amúgy is csak a saját osztályunkba helyeznénk át a problémát. A testreszabott gombunknak valahogy tudatnia kell az ablakkal, hogy rákattintottak. Ezt a legkönnyebben egy esemény létrehozásával tehetjük meg. Végeredményben tehát létrehozunk egy új osztályt, ami egy eseményt küld az ablakosztálynak. Akkor már egyszerűbb, ha rögtön az ablak eseménykezelőjét rendeljük az ablak eseményéhez, hiszen pont ezért tették az ablakosztályokba az eseményeket a keretrendszer tervezői.

Egy másik ok, amiért az eseménykezelőket létrehozták, hogy az események futásidőben kapcsolódnak a kezelőkhöz. Az eseményeket tehát rugalmasabban használhatjuk: különböző eseménykezelőket csatolhatunk az eseményekhez az adott körülményektől függően. Tegyük fel, hogy egy rajzolóprogramot írunk. Attól függően, hogy milyen üzemmódban van a program, az egér lenyomásával rajzolhatunk egy vonalat, vagy kiválaszthatunk egy objektumot. Amikor a felhasználó üzemmódot vált, a program is átválthat egy másik eseménykezelőre. Különböző eseménykezelőkkel rendelkező különböző osztályok egymástól eltérően kezelhetik az eseményt az alkalmazás állapotától függően.

Végezetül, az eseményekkel egyszerre több kezelőt is hozzákapcsolhatunk egy adott eseményhez. Vegyük ismét az előbbi rajzolóprogramot. A `MouseDown` eseményhez egyszerre több eseménykezelő is csatlakozhat. Az első elvégzi a konkrét műveletet, a másik pedig frissíti az állapotsort vagy az egyes parancsok elérhetőségét. Egy esemény így egyszerre több művelet végrehajtását is eredményezheti.

Ha egy függvényünk van, ami egy adott eseményt kezel egy származtatott osztályban, akkor a függvény felülbírálása a gyümölcsözőbb megoldás. Könnyebb karbantartani, nagyobb valószínűséggel marad hibátlan a működése a jövőben, és hatékonyabb is. Az eseménykezelőket az egyéb esetekre kell tartogatnunk. Tehát ha tehetjük, az eseménykezelők használata helyett inkább bíráljuk felül az alaposztályban található virtuális függvény megvalósítását.

36. tipp

Használjuk ki a .NET futásidejű hibakeresési lehetőségeit

Hibák márpedig előfordulnak. Sajnos nem mindig a fejlesztő gépén, ahol könnyen megtalálhatná őket az ember. Azok a gondok, amelyekre képtelenek vagyunk megoldást találni, általában mindig a terepen következnek be egy felhasználó gépén, ahol nincs hibakereső környezet, és esélyünk sincs rájönni a hiba okára. A tapasztaltabb fejlesztők megtanulták beépíteni azt a képességet a programjaikba, hogy azok a lehető legtöbb információt szol-

gáltassák az éles rendszereken is. A .NET keretrendszerben az osztályoknak van egy olyan csoportja, amely segít előállítani a hibakereséshez szükséges információkat. Ezeknek a beállítását fordítási és futásidőben is elvégezhetjük. Ha hasznosítjuk a keretrendszer nyújtotta lehetőségeket, gyorsabban találhatjuk meg azoknak a hibáknak az okát, amelyek csak a terepen fordulnak elő. A keretrendszerben már meglévő kódot használva hibakeresési üzeneteket írhatunk egy fájlba, a rendszernaplóba, vagy egy hibakereső terminálra. Ráadásul még a programunk által hibakeresés közben előállított kimenet szintjét is beállíthatjuk. Ezeket a szolgáltatásokat már a fejlesztési folyamat elejétől érdemes használnunk, hogy biztosan elő tudjuk állítani azt a kimenetet, ami a terepen jelentkező váratlan hibák kijavításához szükséges. Addig viszont semmi esetre se álljunk neki megírni a saját hibakereső könyvtárunkat, amíg meg nem ismerkedünk azzal, ami már adott.

A `System.Diagnostics.Debug`, a `System.Diagnostics.Trace` és a `System.Diagnostics.EventLog` osztályok minden olyan eszközt a rendelkezésünkre bocsátanak, amire szükségünk lehet ahhoz, hogy a hibakereséshez szükséges információkat kinyerhessük egy futó programból. Az első két osztály szinte azonos képességekkel rendelkezik. A különbség annyi, hogy a `Trace` osztály tagfüggvényeit a `TRACE`, míg a `Debug` osztály tagfüggvényeit a `DEBUG` előfeldolgozó-szimbólum vezérli. Amikor létrehozunk egy projektet a VS .NET-ben, a `TRACE` szimbólum meghatározása elérhető lesz a végleges és a hibakereséshez használt változatoknál is, míg a `DEBUG` szimbólumot csak a hibakereső változatban használjuk. A végleges változatokban a hibakeresést a `Trace` osztály segítségével végezzük. Az `EventLog` osztály belépési pontokat kínál a programunk számára ahhoz, hogy az alkalmazás írni tudjon a rendszernaplóba. Az `EventLog` osztály nem támogatja a futásidejű beállítást, de beburkolhatjuk, hogy összhangban legyen ugyanazzal a felülettel, amit mindjárt be fogok mutatni.

A hibakeresés kimenetét szintén befolyásolhatjuk futásidőben. A .NET keretrendszer egy alkalmazásbeállító állományt használ a különféle futásidejű beállítások megadásához. Ez a fájl egy XML dokumentum, ami ugyanabban a könyvtárban található, mint a főprogram futtatható állománya. A fájl neve megegyezik a futtatható fájléval, de `.config` végződéssel. A `MyApplication.exe` beállítását például a `MyApplication.exe.config` XML dokumentum végzi. Az összes beállítást egy beállítási csomópont alatt találjuk:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

</configuration>
```

A .NET keretrendszer előre meghatározott kulcsokat használ a keretrendszerben található osztályok viselkedésének beállításához. Ezeken kívül mi magunk is meghatározhatunk saját kulcsokat és értékeket.

Az alkalmazásunk által előállított kimenet részletességét a `Trace.WriteLineIf()` tagfüggvény és a `TraceSwitch` kapcsolók kombinációjával szabályozhatjuk. Alapértelmezésben a kimenet ki van kapcsolva, hogy az alkalmazás a lehető legjobb teljesítményt nyújtsa. Amikor felmerül egy probléma, megemelhetjük a hibakeresés kimeneti szintjét, és kijavíthatjuk a hibákat. A `WriteLineIf()` csak akkor állít elő kimenetet, ha egy kifejezés értéke igaz lesz:

```
bool _printDiagnostics = true;
Trace.WriteLineIf( _printDiagnostics,
  "Printing Diagnostics Today", "MySubSystem" );
```

A kimenet részletességét a `TraceSwitch` kapcsolók létrehozásával szabályozhatjuk. A `TraceSwitch` olyan változó, amelynek az értékét az alkalmazás beállítóállományának segítségével állíthatjuk be öt különböző érték egyikére. Értéke lehet `Off`, `Error`, `Warning`, `Info` vagy `Verbose`. Ezek az állapotok egy felsorolás részei, és 0 és 4 közötti értékeket képviselnek. Minden alrendszerhez létrehozhatunk egy kapcsolót az üzeneteinek beállításához. A kapcsoló létrehozásához be kell vezetni egy `TraceSwitch` típusú változót, majd meg kell hívni a konstruktorát:

```
static private TraceSwitch librarySwitch = new
  TraceSwitch( "MyAssembly",
  "The switch for this assembly" );
```

Az első paraméter a kapcsoló megjelenítendő neve, a második paraméter pedig a leírása. A kapcsoló értékét futásidőben az alkalmazás beállítóállományának segítségével állíthatjuk be. A következő kódrészlet a `librarySwitch` kapcsolónak az `Info` értéket adja:

```
<system.diagnostics>
  <switches>
    <add name="MyAssembly" value="3" />
  </switches>
</system.diagnostics>
```

Ha módosítjuk a kapcsoló értékét a beállítófájlban, azzal megváltoztatjuk a kapcsoló által vezérelt összes utasítás kimenetét.

Van még egy feladat: be kell állítanunk, hogy hová kerüljön a nyomkövetés kimenete. Alapértelmezés szerint egy figyelő, pontosabban egy `DefaultTraceListener` objektum tartozik a `Trace` osztályhoz. A `DefaultTraceListener` üzeneteket küld a hibakeresőnek, és a `Fail` tagfüggvénye (ami egy kérelem sikertelensége esetén hívódik meg) tájékoztató üzeneteket ír ki, és leállítja a programot. Éles környezetben nem látjuk ezeket

az üzeneteket, de ilyenkor beállíthatunk egy másik figyelőt. A figyelőket az alkalmazás beállítófájljában adhatjuk meg. Az alábbi kódrészlet egy `TextWriterTraceListener` figyelőt ad az alkalmazáshoz:

```
<system.diagnostics>
  <trace autoflush="true" indentsize="0">
    <listeners>
      <add name="MyListener"
        type="System.Diagnostics.TextWriterTraceListener"
        initializeData="MyListener.log"/>
    </listeners>
  </trace>
</system.diagnostics>
```

Ez a `TextWriterTraceListener` az összes nyomkövetési információt a `MyListener.log` állományba írja. A name jellemző adja meg a figyelő nevét, a típus pedig a figyelőként szolgáló objektum típusát, aminek a `System.Diagnostics.TraceListener` leszármazottjának kell lennie. Abban a ritka esetben, ha a .NET keretrendszerben levő figyelőosztályok nem felelnének meg nekünk, létrehozhatjuk a saját figyelőosztályunkat is. Az `initalizeData` érték egy karakterlánc, amit az objektum konstruktora kap meg paraméterként. A `TextWriter-TraceListener` figyelők ezt az értéket használják a fájl nevéhez.

Ezeket az alapokat kicsit kibővíthetjük, hogy leegyszerűsítsük a hibakeresést az alkalmazásunk összes szerelvényében. Minden szerelvényhez adjunk hozzá egy osztályt, ami figyelemmel kíséri a szerelvény által előállított nyomkövetési információkat:

```
internal class MyAssemblyDiagnostics
{
  static private TraceSwitch myAssemblySwitch =
    new TraceSwitch( "MyAssembly",
    "The switch for this assembly" );

  internal static void Msg( TraceLevel l, object o )
  {
    Trace.WriteLineIf( myAssemblySwitch.Level >= l,
      o, "MyAssembly" );
  }

  internal static void Msg( TraceLevel l, string s )
  {
    Trace.WriteLineIf( myAssemblySwitch.Level >= l,
      s, "MyAssembly" );
  }

  // Ide jönnek a megfelelő további kimeneti tagfüggvények
}
```

A `MyAssemblyDiagnostics` osztály hibakeresési üzeneteket állít elő a szerelvényben, az adott szerelvényhez tartozó kapcsoló állapotának megfelelően. Az üzenetek előállításához hívjuk meg valamelyik túlterhelt `Msg` eljárást:

```
public void Method1( )
{
  MyAssemblyDiagnostics.Msg( TraceLevel.Info,
    "Entering Method1." );

  bool rVal = DoMoreWork( );

  if( rVal == false )
  {
    MyAssemblyDiagnostics.Msg( TraceLevel.Warning,
      "DoMoreWork Failed in Method1" );
  }

  MyAssemblyDiagnostics.Msg( TraceLevel.Info,
    "Exiting Method1." );
}
```

A szerelvényhez tartozó kapcsolót kombinálhatjuk egy globális kapcsolóval is, így az egész alkalmazás kimenetét szabályozhatjuk:

```
internal static void Msg( TraceLevel l, object o )
{
  Trace.WriteLineIf ( librarySwitch.Level >= l ||
    globalSwitch.Level >= l,
    o, "MyLibrary" );
}

internal static void Msg( TraceLevel l, string s )
{
  Trace.WriteLineIf( librarySwitch.Level >= l ||
    globalSwitch.Level >= l,
    s, "MyLibrary" );
}
```

Ez lehetővé teszi, hogy a teljes alkalmazás nyomkövetését szabályozva finomabban vezérelhessük az egyes könyvtárak kimenetét. Az alkalmazásszintű nyomkövetést `Error` szintre állíthatjuk, megtalálva így az alkalmazás bármely részében jelentkező hibákat. Ha sikerült elszigetelni a problémát, akkor a vétkes szerelvény kimenetét magasabb szintre állíthatjuk, és így azonosíthatjuk a hiba pontos helyét.

A nyomkövető könyvtárakra a már kiadott, vagyis a terepen futó programok karbantartásához és azok esetleges hibáinak felkutatásához van szükség. Ne írjunk saját nyomkövető

könyvtárat. A .NET FCL könyvtárban már minden ott van, amire szükségünk lehet. Használjuk ki a legteljesebb mértékben, és ha kell, bővítsük a saját igényeinknek megfelelően. Ha így teszünk, akkor minden hibát fülön csíphetünk, még éles környezetben is.

37. tipp

Használjuk a szabványos programbeállítási rendszert

Számtalan utat bejártunk már, hogy megtaláljuk azt a megoldást, amivel elkerülhető az alkalmazás beállításainak behuzalozása a programkódba. A helyes út után kutatva sokszor tökéletesítettük és váltogattuk az erre vonatkozó nézeteinket. INI fájlok? Cikis Win 3.1-es megoldás. A beállítások szerkezete korlátozott volt, és versenyezni kellett a fájlnevekért, nehogy két alkalmazás ütközzön egymással. És a rendszerleíró adatbázis? Nos, ezzel egy lépéssel közelebb kerültünk, de ennek is megvan a maga baja. A rosszindulatú programok komoly kárt tehetnek azokban a gépekben, amelyeknek a rendszerleíró adatbázisába be tudnak férkőzni néhány gonosz kis bejegyzéssel. A rendszerleíró adatbázis sebezhetősége miatt annak bizonyos részeibe írni szándékozó programoknak rendszergazdai jogosultsággal kell rendelkezniük. De tényleg minden felhasználó rendszergazdai jogosultságokkal használja a programokat, hogy azok beleírhassanak a rendszerleíró adatbázisba? Remélhetőleg nem. Ha a rendszerleíró adatbázist használjuk, azoknál a felhasználóknál, akik nem rendelkeznek rendszergazdai jogokkal, a program kivételeket és hibákat fog produkálni, amikor a felhasználó menteni vagy olvasni akarja a beállításokat.

Szerencsére sokkal jobb módszerek is vannak a beállítások tárolására, hogy a programunk a felhasználók igényeihez, a telepítési beállításokhoz, a gép beállításaihoz és még ki tudja mihez igazíthassa a viselkedését. A .NET keretrendszer szabványos helyeket biztosít az alkalmazásunk beállításainak tárolásához. Ezek a helyek az adott alkalmazáshoz tartoznak, és akkor is működnek, ha a felhasználó csak korlátozott jogosultságokkal rendelkezik azon a gépen, amelyiken a program fut.

A csak olvasható információknak az alkalmazásbeállításokat tartalmazó fájlokban a helyük, amelyek olyan XML állományok, amelyek az alkalmazás különböző viselkedéstípusait szabályozzák. Meghatározott sémák adják meg az összes elemet és jellemzőt, amit a .NET FCL kiszűr a beállítófájlokból. Ezek az elemek szabályozzák az olyan beállításokat, mint például hogy a keretrendszer melyik változatát használjuk, a nyomkövetés szintjét (lásd a 36. tippet), illetve a szerelvények keresési útvonalát. Az egyik rész, amivel feltétlenül meg kell ismerkednünk, az `appSettings` szakasz, ami a webes és az asztali alkalmazásokra egyaránt vonatkozik. A futásidejű környezet az alkalmazás indításakor mindig beolvassa ezt a részt. Az összes kulcsot és értéket egy `NameValueCollection` gyűjteménybe teszi, ami az alkalmazásunk tulajdona. Ez a szakasz a miénk. Minden olyan értéket, ami befolyásolja az alkalmazásunk működését, itt kell megadnunk. Ha módosítjuk a beállítófájlt, azzal módosítjuk az alkalmazás viselkedését is.

Az ASP.NET alkalmazásoknak a beállítófájlokat illetően kicsit több mozgásterük van, mint az asztali alkalmazásoknak. Minden virtuális könyvtárnak lehet saját beállítóállománya. A rendszer sorban beolvassa minden olyan virtuális könyvtár beállítóállományát, amely része az URL-nek. A leginkább helyi lesz a nyertes. A http://localhost/MyApplication/SubDir1/SubDir2/file.aspx URL-t például akár négy különböző beállítófájl is irányíthatja. Először a machine.config állományt olvassa be a rendszer. Másodszorra jön a MyApplication könyvtárban található web.config állomány beolvasása. Ezután következik a SubDir1 és a SubDir2 könyvtárak web.config állományainak beolvasása, ebben a sorrendben. Ezek mindegyike módosíthatja a korábban beolvasott értékeket, vagy hozzáteheti a beállításokhoz a saját kulcs–érték párjait. Ezt a fajta beállításöröklődést arra is használhatjuk, hogy a globális beállítások mellett olyan beállításokat is megadjunk, amelyek korlátozzák a privát erőforrásokhoz való hozzáférést. A webes alkalmazásoknál a virtuális könyvtárakhoz más-más beállítások tartozhatnak.

Az asztali rendszereknél csak egyetlen alkalmazásbeállításokat tartalmazó fájl tartozik minden alkalmazástartományhoz. A .NET futásidejű környezet minden általa betöltött végrehajtható fájlhoz létrehoz egy alapértelmezett alkalmazástartományt, majd beolvas egy előre meghatározott beállítófájlt ebbe a tartományba. Ez az alapértelmezett beállítófájl ugyanabban a könyvtárban található, mint a végrehajtható fájl, és a neve <alkalmazasneve>.<kit>.config. A MyApp.exe fájlhoz tartozó beállítófájl neve például az lenne, hogy MyApp.exe.config. Ennek appsettings szakaszába tehetjük a saját kulcs–érték párjainkat az alkalmazásunk számára.

A beállítóállományok kitűnően megfelelnek azoknak az információknak a tárolására, amelyekkel futásidőben szabályozhatjuk az alkalmazásunk viselkedését. De hamar rá fogunk jönni, hogy nincsenek olyan programozói felületek (API-k), amelyekkel az alkalmazás beállításait kiírhatnánk a beállítófájlokba. A felhasználói beállítások tárolásához tehát a beállítóállományok nem felelnek meg. Azért ne rohanjuk mindjárt a rendszerleíró adatbázishoz, de a sajátunkat se írjuk meg teljesen elölről: a .NET asztali alkalmazásaihoz van ennél jobb megoldás is.

Meg kell határoznunk a beállítások formátumát, és ezt az információt a megfelelő helyre kell tennünk. Ezeket a beállításokat egyszerűen menthetjük és beolvashatjuk, ha meghatározzuk a beállítások szerkezetét, és nyilvános írható-olvasható tulajdonságokat adunk a globális beállításokhoz:

```
[ Serializable( ) ]
public struct GlobalSettings
{
  // Ide jönnek a menteni kívánt tulajdonságok
}
```

A beállításokat az XML sorosító (XML rendező) segítségével menthetjük:

```
XmlSerializer ser = new XmlSerializer(
  typeof( GlobalSettings ));
TextWriter wr = new StreamWriter( "data.xml" );
ser.Serialize( wr, myGlobalSettings );
wr.Close( );
```

Az XML formátum használata azt jelenti, hogy a beállítások olvasása, elemzése és nyomkövetése is egyszerű lesz. Titkosított tárolókat is használhatunk a felhasználói beállítások mentéséhez, ha az alkalmazásnál erre van szükség. Ez a példa a következetesség miatt az XML sorosítót és nem az objektumsorosítót használja (lásd a 25. tippet). Az XML sorosító dokumentumokat tárol, és nem teljes objektumfákat. A beállítások általában nem objektumok hálóiból állnak, az XML fájlformátum pedig egyszerűbb is.

Már csak arra kell választ kapnunk, hogy hová kell mentenünk az információkat. A beállításokat három különböző helyre tehetjük. Hogy éppen melyiket választjuk, attól függ, hogy mikor kell az adott információt felhasználni: globálisan, felhasználónként, vagy éppen gépenként és felhasználónként. Mindhárom helyet a `System.Environment.GetFolderPath()` tagfüggvény különböző hívásaival kaphatjuk meg. Az egyes alkalmazásokhoz kapcsolódó könyvtárakat a `GetFolderPath` által visszaadott útvonal végéhez kell hozzáillesztenünk. Legyünk nagyon óvatosak, ha az összes felhasználóra vagy a teljes gépre vonatkozó könyvtárakba írunk. Ehhez a célgépen magasabb jogosultsági szintre lesz szükség.

Az `Environment.SpecialFolders.CommonApplicationData` az összes gép összes felhasználója által közösen használt beállításokat tartalmazó könyvtár címét adja vissza. Egy alapértelmezésekkel telepített gépen a `GetFolderPath(SpecialFolders.CommonApplicationData)` a `C:\Documents and Settings\All Users\Application Data` útvonalat adja vissza. Ide kerülnek azok a beállítások, amelyek minden gépre és felhasználóra egyaránt vonatkoznak. Ha olyan információt hozunk létre, amelynek ide kell kerülnie, akkor a telepítővel írjuk be ide, vagy egy felügyeleti modult használjunk. A felhasználói programokból soha ne írjunk ide adatokat. Valószínű ugyanis, hogy a felhasználók gépén az alkalmazás nem fog rendelkezni a megfelelő jogosultságokkal.

Az `Environment.SpecialFolders.ApplicationData` az adott felhasználó könyvtárát adja vissza, amin a hálózat összes gépe osztozik. Egy alapértelmezésekkel telepített gépen a `GetFolderPath(SpecialFolders.ApplicationData)` a `C:\Documents and Settings\<felhasználónév>\Application Data` útvonalat adja vissza. Minden felhasználónak van saját alkalmazásadatokat tároló könyvtára. Amikor a felhasználó bejelentkezik egy tartományba, ezt a felsorolást használva megkapjuk azt a megosztott hálózati mappát, amelyik a felhasználó globális beállításait tartalmazza. Az adott felhasználó az itt tárolt beállításokat használja, attól függetlenül, hogy a hálózat melyik gépéről jelentkezett be.

Az Environment.SpecialFolders.LocalApplicationData azt a könyvtárat adja vissza, amelyik kimondottan az adott felhasználóhoz tartozó információkat tartalmazza, de csak akkor, ha a felhasználó az adott gépen jelentkezik be. A GetFolderPath(SpecialFolders.LocalApplicationData) jellemző visszatérési értéke lehet például a C:\Documents and Settings\<felhasználónév>\Local Settings\Application Data útvonal.

Ezen a három különböző helyen tárolhatjuk azokat a beállításokat, amelyek mindenkire, az adott felhasználóra vagy az adott gépre és felhasználóra vonatkoznak. Az, hogy ezek közül pontosan melyiket használjuk, az adott alkalmazástól függ. De nézzünk is mindjárt meg néhány kézenfekvő példát. Az adatbáziskapcsolat egy globális beállítás, amit a Common Application Data (közös alkalmazásadatok) könyvtárban kell tárolni. Egy adott felhasználó munkakörnyezetét az Application Data könyvtárban kell tárolni, mert az kizárólag a felhasználótól függ. Az ablakbeállításoknak a Local Application Data (helyi alkalmazásadatok) könyvtárban a helyük, mert azok az adott felhasználótól és az adott géptől is függnek (a különböző gépek képernyőfelbontása eltérő lehet).

Ezek a különleges mappák írják le a legmagasabb szintű könyvtárszerkezetet az alkalmazások által mentett felhasználói beállítások tárolásához. Ezekben a legfelső szinten lévő mappákban minden esetben alkönyvtárakat kell létrehoznunk. A .NET keretrendszer System.Windows.Application osztálya meghatároz olyan tulajdonságokat, amelyek segítenek felépíteni a gyakran használt beállítási útvonalakat. Az Application.LocalAppDataPath tulajdonság a GetFolderPath(SpecialFolders.CommonApplicationData)+"\\CompanyName\\ProductName\\ProductVersion" útvonalat adja vissza. Hasonlóképpen, az Application.UserDataPath és az Application.LocalUserDataPath a felhasználói adatok, illetve a helyi adatok könyvtárainak útvonalát adja meg az adott cég adott alkalmazásának adott verziószámához. Ha kombináljuk ezeket a helyeket, akkor a cégünk összes alkalmazásához létrehozhatunk beállítóinformációkat, valamint az adott alkalmazás összes változatához, illetve csak az adott változatra vonatkozó információkat is előállíthatunk.

Vegyük észre, hogy a könyvtárak között egyszer sem említettem az alkalmazás könyvtárát a Program Files mappában. A Program Files mappába és a Windows mappába soha nem szabad írnunk. Ezek a helyek magasabb szintű jogosultságokat igényelnek, és nem szabad arra számítanunk, hogy a felhasználók majd rendelkezni fognak az íráshoz szükséges jogosultságokkal.

Ahogy a vállalati felhasználóktól kezdve az otthoni felhasználókig mindenki egyre inkább odafigyel a biztonságra, egyre fontosabbá válik az a kérdés, hogy hová kerüljenek az alkalmazás beállításai. Ha a megfelelő helyre tesszük ezeket az információkat, azzal úgy könnyítjük meg a felhasználóink számára az alkalmazásunk használatát, hogy eközben nem veszélyeztetjük a gép biztonságát. Ettől még személyessé tehetjük a felhasználók

munkakörnyezetét. Ehhez csak a .NET sorosító környezetét kell kombinálnunk a megfelelő helyekkel, és így könnyen egyedivé tehetjük az alkalmazás megjelenését minden felhasználó számára anélkül, hogy kockára tennénk a biztonságot.

38. tipp

Használjuk és támogassuk az adatkötést

A gyakorlott Windows-programozók tisztában vannak azzal, hogy milyen kóddal lehet adatokat helyezni a vezérlőkbe, majd kiolvasni a bennük lévő értékeket:

```
public Form1 : Form
{
  private MyType myDataValue;
  private TextBox textBoxName;

  private void InitializeComponent( )
  {
    textBoxName.Text = myDataValue.Name;
    this.textBoxName.Leave += new
      System.EventHandler( this.OnLeave );
  }

  private void OnLeave( object sender, System.EventArgs e )
  {
    myDataValue.Name = textBoxName.Text;
  }
}
```

Egyszerű, de monoton ismétlődő kód ez, amit mindnyájan utálunk megírni, mert biztosak vagyunk benne, hogy léteznie kell jobb módszernek. És létezik is. A .NET keretrendszer támogatja az **adatkötést**, ami az objektumok tulajdonságait a vezérlők tulajdonságaira képezi le:

```
textBoxName.DataBindings.Add ( "Text",
myDataValue, "Name" );
```

Az előbbi kód a `textBoxName` vezérlő „`Text`" tulajdonságát köti a `myDataValue` ojektum „`Name`" tulajdonságához. A program belsejében két objektum, a `BindingManager` és a `CurrencyManager` valósítja meg a vezérlő és az adatforrás közti adatátvitelt. Ezzel a szerkezettel már bizonyára számtalan példában találkoztunk, különösen a `DataSet` és a `DataGrid` objektumoknál. Minden bizonnyal szövegmezőkkel is végeztünk már egyszerű adatkötéseket. Az is valószínű, hogy ezzel éppen csak belekóstoltunk az adatkötés nyújtotta lehetőségekbe. Az adatkötés hatékonyabb használatával ugyanis elkerülhetjük a monoton kódolást.

Az adatkötés teljes bemutatása biztosan megtöltene egy-két könyvet, hiszen a windowsos és a webes alkalmazások is támogatják az adatkötést. Ahelyett, hogy a témát teljesen kimerítő értekezést írnék, inkább azt próbálom meg elérni, hogy megjegyezzük az adatkötés kulcsfontosságú előnyeit. Először is, az adatkötés használata sokkal egyszerűbb, mint saját kódot írni. Másodszor, a szöveges elemeken kívül rengeteg más felhasználási területe van, hiszen más megjelenítendő tulajdonságok kötésére is lehetőség van. Harmadszor, a windowsos ablakokban az adatkötés felel azon vezérlők összehangolásáért, amelyek egymással kapcsolatban lévő adatforrásokat használnak.

Tegyük fel például, hogy felmerül egy olyan igény, hogy egy szöveget piros színnel írjunk ki, amikor az adat értéke érvénytelen. Ezt az alábbi kódrészlettel írhatnánk meg:

```
if ( src.TextIsInvalid )
{
  textBox1.ForeColor = Color.Red;
} else
{
  textBox1.ForeColor = Color.Black;
}
```

Ez mind szép és jó, de minden alkalommal meg kell hívnunk ezt a kódrészletet, amikor az adatforrásban található szöveg megváltozik. Rengeteg eseményt kell kezelni, és előfordulhat, hogy megfeledkezünk néhányról. Használjunk inkább adatkötést. Adjunk hozzá a `src` objektumhoz egy tulajdonságot, ami visszaadja a megfelelő betűszínt, egy másik helyen pedig állítsuk be ennek a változónak az értékét a megfelelő színre a szöveges üzenet állapotától függően:

```
private Color _clr = Color.Black;
public Color ForegroundColor
{
  get
  {
    return _clr;
  }
}

private string _txtToDisplay;
public string Text
{
  get
  {
    return _txtToDisplay;
  }
  set
  {
    _txtToDisplay = value;
```

```
    UpdateDisplayColor( IsTextValid( ) );
  }
}

private void UpdateDisplayColor( bool bValid )
{
  _clr = ( bValid ) ? Color.Black : Color.Red;
}
```

Ezután már csak a kötést kell hozzáadnunk a szövegmezőhöz:

```
textBox1.DataBindings.Add ("ForeColor",
src, "ForegroundColor");
```

Miután megtörtént az adatkötés beállítása, a `textBox1` mindig a helyes színnel fogja kirajzolni a szövegét, a forrásobjektum belső értéke alapján. Ahelyett, hogy egy csomó eseménykezelőnk lenne, és egy csomó olyan hely, ahol változna a megjelenítendő szín, most csak két ilyen hely van. Az adatforrás-objektum figyeli azokat a tulajdonságokat, amelyek hatással vannak a megfelelő megjelenítésre, az ablak pedig szépen irányítja az adatkötést.

Habár a bemutatott példák windowsos ablakokra épültek, ugyanez az elv érvényesül a webes alkalmazásoknál is. A webes vezérlőknél is hozzáköthetjük az adatforrás tulajdonságait a webes vezérlők tulajdonságaihoz:

```
<asp:TextBox id=TextBox1 runat="server"
  Text="<%# src.Text %>"
  ForeColor="<%# src.ForegroundColor %>">
```

Ez azt jelenti, hogy amikor olyan típusokat hozunk létre, amelyeket az alkalmazásunk a felhasználói felületen jelenít meg, akkor mindig célszerű megadni azokat a tulajdonságokat, amelyek segítségével létrehozhatjuk és frissíthetjük a felhasználói felület megjelenését a felhasználói igényeknek megfelelően.

De mi van akkor, ha egy objektum nem támogatja azokat a tulajdonságokat, amelyekre szükségünk lenne? Ilyenkor összecsomagoljuk azt, amink van, és amire szükségünk lenne. Nézzük meg ezt az adatszerkezetet:

```
public struct FinancialResults
{
  public decimal Revenue
  {
    get { return _revenue; }
  }
```

```
  public int NumberOfSales
  {
    get { return _numSales; }
  }

  public decimal Costs
  {
    get { return _cost;}
  }

  public decimal Profit
  {
    get { return _revenue - _cost; }
  }
}
```

Felmerül az igény, hogy megjelenítsük ezeket egy ablakban valamilyen különleges formátumban. Ha a nyereség értéke negatív, akkor pirossal kell kiírni. Ha az eladások száma 100 alá csökken, akkor félkövéren kell megjelenítenünk az értéket. Ha a költség, több mint 10 000, akkor szintén legyen félkövér. Az a fejlesztő, aki létrehozta a `FinancialResults` szerkezetet, nem látta el azt a felhasználói felületet támogató képességekkel. Valószínűleg ez volt a helyes döntés. A `FinancialResults` szerkezetnek csupán az a feladata, hogy tárolja a tényleges értékeket. Most létrehozhatunk egy új típust, amiben a `FinancialResults` eredeti, adatokat tároló tulajdonságai mellett már benne lesznek a felhasználói felület formátumára vonatkozó tulajdonságok is:

```
public struct FinancialDisplayResults
{
  private FinancialResults _results;
  public FinancialResults Results
  {
    get { return _results; }
  }

  public Color ProfitForegroundColor
  {
    get
    {
      return ( _results.Profit >= 0 ) ?
        Color.Black : Color.Red;
    }
  }

  // a többi formázási beállítás elhagyva
}
```

Ezzel létrehoztunk egy egységes adatszerkezetet, hogy elősegítsük a benne lévő szerkezet adatkötését:

```
// Ugyanazt az adatforrást használjuk
// Ez egyetlen Binding Manager (adatkötés-kezelő) létrehozásával jár
textBox1.DataBindings.Add ("Text",
  src, "Results.Profit");
textBox1.DataBindings.Add ("ForeColor",
  src, "ProfitForegroundColor");
```

Egyetlen, csak olvasható tulajdonságot hoztunk létre, ami biztosítja a hozzáférést a belső pénzügyi szerkezethez. Ez a szerkezet persze nem megfelelő, ha az adatok írását és olvasását is lehetővé akarjuk tenni. A `FinancialResults` szerkezet egy értéktípus, ami azt jelenti, hogy a `get` hozzáférő nem nyújt lehetőséget a létező tár elérésére, csak egy másolatot ad vissza. Vagyis ez a megoldás vidáman visszaad egy másolatot, amit az adatkötéssel nem tudunk módosítani. Ha viszont módosításokat is szeretnénk végezni, akkor a `FinancialResults` típus osztály lenne, és nem struktúra (lásd a 6. tippet). Hivatkozási típusról lévén szó, ekkor a `get` hozzáférő egy hivatkozást ad vissza a belső tárolóra, s így már támogatja a felhasználói adatmódosításokat. A belső szerkezetnek valahogy reagálnia kellene a belső tárolón végrehajtott módosításokra. A `FinancialResults` események kiváltásával értesíthetné a kód többi részét az állapotváltozás bekövetkeztéről.

Fontos, hogy az adatforrást minden kapcsolódó vezérlővel használjuk az adott ablakban. A `DataMember` tulajdonsággal tehetünk különbséget az egyes vezérlőkben megjelenítendő tulajdonságok között. A kötést végző kódot így is megírhattuk volna:

```
// Rossz szokás, két adatkötés-kezelőt hoz létre
textBox1.DataBindings.Add ("Text",
src.Results, "Profit");
textBox1.DataBindings.Add ("ForeColor",
src, "ProfitForegroundColor");
```

Ezzel két adatkötés-kezelőt hoznánk létre, egyet az `src` objektumhoz, egy másikat meg az `src.Results` objektumhoz. Mindegyik adatforrást más-más adatkötés-kezelő kezelné. Ha azt szeretnénk, hogy az adatkötés-kezelő az összes tulajdonságot módosítsa, amikor megváltozik az adatforrás, akkor gondoskodnunk kell róla, hogy az adatforrások passzoljanak.

Az adatkötést a windowsos és webes vezérlők szinte minden tulajdonságával használhatjuk. A vezérlőben megjelenő értékekhez, a betűtípushoz, a csak olvasható állapot beállításához, sőt még a vezérlő helye is lehet az adatkötés célpontja. A tanácsom tehát az, hogy az osztályokat és a struktúrákat úgy készítsük el, hogy bennük legyenek azok az értékek, amelyekre szükség van ahhoz, hogy a felhasználók igényeinek megfelelően jeleníthessük meg az adatokat. A vezérlők frissítését ezután az adatkötés segítségével oldjuk meg.

Az egyszerű vezérlőkön túl az adatkötést gyakran használjuk a DataSet és a DataGrid objektumokkal. Ez igen hatékony megoldás. A DataGrid objektumot hozzákötjük a DataSet objektumhoz, és az megjeleníti a DataSet összes értékét. Ha a DataSet több táblát is tartalmaz, akkor még a táblák között is ugrálhatunk. Hát nem imádnivaló?

Gond csak akkor van, ha az adathalmazban nincsenek benne azok a mezők, amelyeket meg akarunk jeleníteni. Ilyenkor be kell szúrnunk egy oszlopot a DataSet objektumba, ami kiszámolja azokat az értékeket, amelyeket meg akarunk jeleníteni a felhasználó számára. Ha az értéket meg lehet határozni egy SQL lekérdezéssel, akkor a DataSet ki tudja számolni az értéket helyettünk. Az alábbi kód beszúr egy oszlopot az Employees adattáblába, ami a név formázott változatát jeleníti meg:

```
DataTable dt = data.Tables[ "Employees" ];
dt.Columns.Add( "EmployeeName",
  typeof( string ),
   "lastname + ', ' + firstname");
```

Oszlopokat adva a DataSet objektumhoz, oszlopokat adhatunk a DataGrid objektumhoz is. Objektumokból álló rétegeket építünk a tárolt adatobjektumokra, és így jelenítjük meg az adatokat a felhasználónak szánt formátumban.

Az eddig bemutatott példák egytől-egyig karakterlánctípusokra épültek. A keretrendszer viszont képes átalakítani a karakterláncokat számokká úgy, hogy megkísérli átalakítani a felhasználó által bevitt adatokat a megfelelő típusúra. Ha ez nem megy, akkor visszaállítja az eredeti értéket. Ez működik ugyan, de a felhasználó semmilyen visszajelzést nem kap, a bevitt adatokat szép csendben figyelmen kívül hagyja a program. A hiányzó visszajelzést úgy pótolhatjuk, ha feldolgozzuk az adatkötő környezet Parse eseményét. Ez az esemény akkor következik be, amikor az adatkötés-kezelő a vezérlőből származó értékkel frissíti az adatforrásban található értéket. A ParseEventArgs megadja a felhasználó által begépelt szöveget, és azt hogy milyen típusra kellene átalakítani ezt a szöveget. Ezt az eseményt elfoghatjuk és végrehajthatjuk a saját értesítő műveletünket, sőt ezzel egyetemben akár módosíthatjuk az értéket, és a saját értékünkkel frissíthetjük a szöveget:

```
private void Form1_Parse( object sender, ConvertEventArgs e )
{
  try {
    Convert.ToInt32 ( e.Value );
  } catch
  {
    MessageBox.Show (
     string.Format( "{0} is not an integer",
       e.Value.ToString( ) ) );
    e.Value = 0;
  }
}
```

Előfordulhat, hogy a `Format` eseményt is érdemes kezelnünk. Ez az a kapocs, aminek a segítségével formázhatjuk az adatforrásból a vezérlőbe kerülő adatokat. A megjelenítendő karakterláncot a `ConvertEventArgs Value` mezőjének módosításával formázhatjuk meg.

A .NET keretrendszer általános keretet nyújt az adatkötések támogatásához. A mi feladatunk az, hogy megírjuk az alkalmazásunkhoz és az adatainkhoz a konkrét eseménykezelőket. A Windows Forms és a Web Forms alrendszerek mindegyike gazdag adatkötési képességekkel rendelkezik. A könyvtárban ott van minden eszköz, amire szükségünk lehet, ezért a felhasználói felületet vezérlő kódunknak csak a megjelenítendő adatforrások és tulajdonságok leírásával és azoknak a szabályoknak a megadásával kell foglalkoznia, amelyek az elemekben található értékek visszaírását segítik az adatforrásokba. Nekünk csak a megjelenítési paramétereket leíró adattípusokra kell összpontosítanunk, a Winforms és a Webforms adatkötés pedig elvégzi a többit helyettünk. Elkerülhetetlen, hogy megírjuk azt a kódot, ami a felhasználói felület vezérlői és az adatforrás között mozgatja az adatokat. Az adatoknak valahogy el kell jutniuk az adatokat tároló objektumoktól azokhoz a vezérlőkig, amelyek segítségével a felhasználók párbeszédet folytathatnak a programmal. De ha a típusok egymásra épülő rétegeit és az adatkötés nyújtotta lehetőségeket is kihasználjuk, akkor sokkal kevesebb kódot kell megírnunk; a keretrendszer elvégzi helyettünk az adatok mozgatását a windowsos és a webes alkalmazásokban is.

39. tipp

Használjuk a .NET adatérvényesítést

A felhasználóktól kapott adatok számtalan helyről érkezhetnek. A fájlokon és az interaktív vezérlőkön keresztül bevitt adatok helyességét is ellenőriznünk kell. A felhasználók által bevitt adatok érvényesítése rendkívül aprólékos munkát igényel, ami ugyan számos hibalehetőséget hordoz magában, de mindenképpen szükség van rá. Ha feltétel nélkül megbízunk a felhasználó által bevitt adatokban, az szerencsésebb esetben csak egy kivételhez vezethet, rosszabb esetben azonban akár egy SQL-befecskendező támadás (SQL injection attack) áldozatává válhat a programunk. Egyik sem kellemes. Eleget láttunk már ahhoz, hogy egészséges gyanakvással fogadjuk a felhasználótól érkező adatokat. Helyes. Szerencsére így vannak ezzel mások is. A .NET keretrendszer ezért széleskörű adatérvényesítési képességekkel rendelkezik, amit felhasználva minimálisra csökkenthetjük az általunk megírt kód mennyiségét, és mégis minden egyes felhasználótól érkező adat érvényességét ellenőrizhetjük.

A .NET keretrendszer más-más módszert kínál a felhasználó által bevitt adatok érvényesítéséhez a webes, illetve a Windows alapú alkalmazásoknál. A webes alkalmazásoknak a böngészőben kell érvényesíteniük az adatokat JavaScript kód segítségével. Az érvényesítő vezérlők a HTML lap részeként állítják elő a szükséges JavaScript kódot. Ez a leghatékonyabb megoldás a felhasználók szemszögéből, hiszen nem kell mindig tiszteletköröket tenniük a kiszolgálóhoz, amikor módosítanak egy adatot. A webes vezérlők szabályos ki-

fejezéseket használnak a felhasználó által bevitt adatok előzetes ellenőrzéséhez, mielőtt a lapot visszaküldenék a kiszolgálónak. A kiszolgálón persze további ellenőrzést kell végeznünk, hogy megelőzzük a rosszindulatú programok támadásait. A windowsos alkalmazások másik modellt használnak. A felhasználó által bevitt adatokat C# kóddal lehet érvényesíteni, ami ugyanabban a környezetben fut, mint maga az alkalmazás. A Windows vezérlők teljes palettáját használhatjuk arra, hogy értesítsük a felhasználót, ha érvénytelen adatokat adott meg. Az általános modell a tulajdonságok hozzáférőiben kivételekkel jelzi, ha érvénytelen adatokkal találkozik. A felhasználói felület elemei elfogják ezeket a kivételeket, és értesítik a hibáról a felhasználót.

Az ASP.NET alkalmazásokban öt webes vezérlővel kezelhetjük a legtöbb adatérvényesítő feladatot. Mind az öt működését olyan tulajdonságok vezérlik, amelyek megadják, hogy mely mezőket kell ellenőrizni, és milyen szabályoknak kell megfelelnie a helyesen bevitt adatoknak. A `RequiredFieldValidator` arra kényszeríti a felhasználót, hogy mindenképpen beírjon valamilyen értéket az adott mezőbe. A `RangeValidator` megköveteli, hogy az adott mező értéke a megadott korlátok közé essen. A korlátok jelölhetik egy számérték nagyságát vagy egy karakterlánc hosszát. A `CompareValidator` segítségével olyan érvényességi szabályokat adhatunk meg, amelyek összekapcsolják egy weblap két mezőjét. Ez a három elég egyszerűen kezelhető, az utolsó kettővel viszont gyakorlatilag minden felhasználói adatot érvényesíthetünk. A `RegularExpression` érvényesítő egy szabályos kifejezés segítségével érvényesíti a felhasználó által megadott adatokat. Ha az összehasonlítás megfelelő eredményt ad, akkor a felhasználó által bevitt adatok helyesek. A szabályos kifejezések rendkívül hatékony nyelvi eszközök. Gyakorlatilag bármilyen helyzethez megalkothatjuk a megfelelő szabályos kifejezést. A Visual Studio .NET rendszerben találunk olyan érvényesítő mintakifejezéseket, amelyek segítségével elindulhatunk. Rengeteg forrás áll rendelkezésre, amiből megtanulhatjuk a szabályos kifejezések használatát, és javaslom, hogy éljünk a lehetőséggel. Persze nem hagyhatom ennyiben a témát anélkül, hogy be ne mutatnék néhányat a szabályos kifejezésekben használt leggyakoribb szerkezetek közül. Az 5.1. táblázatban azokat az elemeket láthatjuk, amelyeket a leggyakrabban fogunk használni az alkalmazásainkban a beérkező adatok érvényesítéséhez.

5.1. táblázat Szabályos kifejezések gyakran használt elemei

Elem	Jelentés
`[a-z]`	Bármely kisbetű. A szögletes zárójelek között lévő karakterhalmaz egy karaktere.
`\d`	Bármely számjegy.
`^, $`	A ^ a sor elejét, a $ a sor végét jelzi.
`\w`	Bármely alfanumerikus karakter. Az `[A-Za-z0-9]` rövidített változata.

Elem	Jelentés
`(?NamedGroup\d{4,16})`	Két különböző gyakori elemet szemléltet. A `?NamedGroup` a mintára illeszkedő találatra hivatkozó változót határoz meg. A `{4,16}` azt adja meg, hogy az őt megelőző szerkezetnek legalább 4-szer kell, de legfeljebb 16-szor szabad előfordulnia. Ez a minta egy legalább 4, de legfeljebb 16 számjegyből álló karakterláncra illeszkedik. Ha talál ilyet a program, akkor az eredményre később `NamedGroup` néven lehet hivatkozni.
`(a|b|c)`	Az `a`, a `b` vagy a `c` illeszkedik rá. Az egymástól függőleges vonalakkal elválasztott lehetséges értékeket egy logikai vagy (`OR`) kapcsolja össze. A bevitt karakterláncban bármelyik szerepelhet.
`(?(NamedGroup)a|b)`	Választás. A C# nyelv háromtagú műveletének felel meg. Azt jelenti, hogy „Ha a `NamedGroup` létezik, akkor az a lesz az illesztési minta, különben pedig a `b`".

Meg fogjuk látni, hogy ezeket a szerkezeteket és szabályos kifejezéseket használva szinte bármit érvényesíthetünk, amit ránk zúdítanak a felhasználók. Ha pedig a szabályos kifejezések mégis kevésnek bizonyulnának, akkor készíthetünk egy saját érvényesítőt is, egy új osztályt származtatva a `CustomValidator` osztályból. Ez nem kevés munkával jár, s ha tehetem, akkor inkább kerülni szoktam. A kiszolgálóoldali ellenőrzőfüggvényeket C# nyelven írjuk meg, majd ECMAscript kóddal elkészítjük az ügyféloldali ellenőrzőfüggvényeket is. Utálok kétszer megírni valamit. Kerülöm az ECMAscript használatát is, inkább szeretek megmaradni a szabályos kifejezéseknél.

Példaként nézzünk meg egy szabályos kifejezést, ami USA-beli telefonszámok érvényesítéséhez használható. A körzetszámokat elfogadja zárójelek között vagy nélkülük. A körzetszám, a központszám és a vonalszám között bármennyi elválasztó karakter, vagyis üres köz állhat. A körzetszám és a központszám közé tetszés szerint tehetünk egy kötőjelet, de ez nem kötelező:

```
((\(\s*\d{3}\s*\))|(\d{3}))-?\s*\d{3}\s*-\s*\d{4}
```

Az egyes kifejezéscsoportokat megvizsgálva világosan látható a gondolatmenet:

```
((\(\s*\d{3}\s*\))|(\d{3}))-?
```

Ez a rész a körzetszám illesztését végzi. Megengedi az (xxx) és az xxx formátumot, ahol az xxx három számjegyet jelöl. A számjegyek körül bármennyi üres köz lehet. Az utolsó két karakter, a - és a ? megengedi, de nem teszi kötelezővé a kötőjelet.

A kifejezés többi része a telefonszám xxx-xxxx részére illeszkedik. Az \s bármennyi üres közre illeszkedik. A \d{3} 3 számjegyet jelöl. Az \s*-\s* egy kötőjelre és az azt körülvevő bármennyi üres közre illeszkedik. Végezetül, a \d{4} pontosan 4 számjegyet jelöl.

A windowsos adatérvényesítés kicsit másképpen működik. Nincsenek előregyártott érvényesítők, amelyek kielemeznénk helyettünk a bevitt adatokat. Ehelyett egy eseménykezelőt kell írnunk a System.Windows.Forms.Control.Validating eseményhez. Vagy, ha egyedi vezérlőt készítünk, felül kell bírálnunk az OnValidating tagfüggvényt (lásd a 35. tippet). Az érvényesítő esemény egy szokásos formáját az alábbiakban közöljük:

```
private void textBoxName_Validating( object sender,
  System.ComponentModel.CancelEventArgs e )
{
  string error = null;
  // Ellenőrzés végrehajtása
  if ( textBoxName.Text.Length == 0 )
  {
    // Ha az eredmény nem megfelelő, beállítjuk a hibaüzenet tartalmazó
    // karakterláncot, majd töröljük az érvényesítő eseményt
    error = "Please enter a name";
    e.Cancel = true;
  }
  // Frissítjük a hibaszolgáltató állapotát
  // a megfelelő hibaüzenettel. Ha nincs hiba,
  // az értéke null lesz.
  this.errorProviderAll.SetError( textBoxName, error );
}
```

Van még néhány kisebb feladat, amit el kell végeznünk ahhoz, hogy biztosak lehessünk benne, hogy az ellenőrzésen nem csúszhat át érvénytelen adat. Minden vezérlőnek van egy CausesValidation tulajdonsága. Ez a tulajdonság határozza meg, hogy a vezérlő részt vesz-e az érvényesítésben. Általában célszerű igaz értéken hagyni az összes vezérlőnél, de kivétel ez alól a Cancel gomb. Ha elfelejtjük levenni az igaz értéket, akkor a felhasználónak érvényes adatokat kell megadnia ahhoz is, hogy eltüntesse a párbeszédablakot. A második egyszerű feladat az, hogy írjunk egy kezelőt az OK gombhoz, ami utasítja a programot az összes többi vezérlő adatainak érvényesítésére. Érvényesítés ugyanis csak akkor következik be, amikor a felhasználó egy vezérlőre lép, vagy ha elhagyja a vezérlőt. Ha a felhasználó az ablak megnyitása után azonnal megnyomja az OK gombot, akkor az érvényesítő kódból semmi nem fut le. Hogy ezt orvosoljuk, meg kell adnunk egy kezelőt az OK gombhoz, ami végiglépked az összes vezérlőn, és rákényszeríti azokat az adatok

érvényesítésére. Az alábbi két eljárás megmutatja, hogyan lehet ezt helyesen megtenni. Az önhívó (rekurzív) eljárások azokról a vezérlőkről gondoskodnak, amelyek maguk is más vezérlőket tartalmaznak: ilyenek a fülekkel rendelkező többlapos párbeszédablakok (tab page), a csoportmezők (group box) és a táblák (panel):

```
private void buttonOK_Click( object sender,
  System.EventArgs e )
{
  // Mindenkit ellenőrzünk:
  // A this.DialogResult értéke itt
  // DialogResult.OK lesz
  ValidateAllChildren( this );
}

private void ValidateAllChildren( Control parent )
{
  // Ha az érvényesítés már kudarcot vallott,
  // ne folytassuk az ellenőrzést
  if( this.DialogResult == DialogResult.None )
    return;

  // Minden vezérlőnél
  foreach( Control c in parent.Controls )
  {
    // átadjuk a fókuszt
    c.Focus( );

    // megkíséreljük az érvényesítést
    if (!this.Validate( ))
    {
      // ha nem sikerül, akkor nem hagyjuk, hogy
      // a párbeszédablakot bezárják
      this.DialogResult = DialogResult.None;
      return;
    }
    // A gyermekek adatainak érvényesítése
    ValidateAllChildren( c );
  }
}
```

Ez a kód a legtöbb előforduló eset kezelésére alkalmas. A `DataGrid–DataSet` pároshoz tartozik még egy trükk. Az `ErrorProvider` osztály `DataSource` és `DataMember` tulajdonságait vagy a tervezéskor kell megadnunk:

```
ErrProvider.DataSource = myDataSet;
ErrProvider.DataMember = "Table1";
```

Vagy futásidőben meg kell hívnunk a `BindToDataErrors` tagfüggvényt, amivel mindkettőt egy művelet segítségével állíthatjuk be:

```
ErrProvider.BindToDataAndErrors(
  myDataSet, "Table1" );
```

A hibák megjelenítése úgy történik, hogy beállítjuk a `DataRow.RowError` tulajdonságot, majd a `DataRow.SetColumnError` tagfüggvény meghívásával megjelenítjük a konkrét hibákat. Az `ErrorProvider` kirajzol egy piros felkiáltójelet ábrázoló ikont az érintett sornál és a `DataGrid` konkrét cellájánál.

A keretrendszerben található vezérlők érvényesítését bemutató szélsebes túránk kellő alapot nyújt ahhoz, hogy hatékonyan segítse a munkánkat számos alkalmazásban, ahol ellenőriznünk kell a felhasználó által bevitt adatokat. A felhasználó adataiban soha ne bízzunk meg. A felhasználók hibáznak, és időnként rosszindulatú felhasználók megpróbálhatnak betörni a rendszerünkbe. Az ennek kivédése érdekében megírt kód mennyiségét minimálisra csökkenthetjük, ha a lehető legjobban kihasználjuk a .NET keretrendszerben elérhető szolgáltatásokat. A felhasználóktól érkező összes adatot ellenőrizzük, de tegyük ezt a már meglévő eszközök használatával.

40. tipp

Igazítsuk a gyűjteményeket az igényeinkhez

Arra a kérdésre, hogy „Melyik a legjobb gyűjtemény?" a válaszom az, hogy „Attól függ." A különféle gyűjteményeknek különböző teljesítményjellemzőik vannak, és más-más feladatok ellátására optimalizálták őket. A .NET keretrendszer sok gyakran használt gyűjteményt támogat: listákat, tömböket, sorokat, vermeket és egyebeket. A C# támogatja a többdimenziós tömböket, amelyeknek a teljesítménymutatói eltérnek az egydimenziós, illetve a fogazott tömbökétől (jagged array, tömböket tartalmazó tömb). A keretrendszerben számos különleges gyűjteményt is találunk, amelyeket érdemes mindig megnézni, mielőtt hozzáfognánk egy saját gyűjtemény elkészítéséhez. A gyűjteményeket könnyű felismerni arról, hogy mindegyik megvalósítja az `ICollection` felületet. Az `ICollection` dokumentációja felsorolja az összes olyan osztályt, amelyik megvalósítja ezt a felületet. Így összesen húszegynéhány osztály áll a rendelkezésünkre.

Ahhoz, hogy a szándékainknak legmegfelelőbb gyűjteményt választhassuk, át kell gondolnunk, hogy milyen műveleteket fogunk a leggyakrabban elvégezni az adott gyűjteményen. Rugalmas programot úgy készíthetünk, ha az osztályokon belül megvalósított felületekre támaszkodunk. Ilyenkor lecserélhetjük a gyűjteményt egy másikra, ha rájövünk, hogy a feltevéseink a gyűjtemény használatát illetően tévesnek bizonyultak (lásd a 19. tippet).

A .NET keretrendszerben három különböző típusú tömb van: léteznek tömbök, tömbszerű gyűjtemények és kivonat (hasítótábla) alapú tárolók. A tömbök a legegyszerűbbek és általában véve a leggyorsabbak, ezért kezdjük velük. Ezt a gyűjteménytípust fogjuk a legtöbbet használni.

Az elsőszámú választásunk általában a `System.Array` osztály, helyesebben egy típusos tömbosztály lesz. A legfontosabb ok, amiért a tömböket választjuk az, hogy a tömbök típusbiztosak. Minden más gyűjtemény `System.Object` hivatkozásokat tárol, egész addig, amíg a C# 2.0 változatával meg nem érkeznek az általános elemek (lásd a 49. tippet). Amikor bevezetünk egy tömböt, a fordító létrehoz egy konkrét `System.Array` származékot az általunk megadott típushoz. Ez a kifejezés például egy egészekből álló tömböt vezet be:

```
private int [] _numbers = new int[100];
```

A tömb egészeket tárol, nem `System.Object` hivatkozásokat. Ez azért fontos, mert így elkerüljük a be- és kicsomagolással járó teljesítménycsökkenést, amikor értéktípusokat szúrunk be, érünk el, vagy törlünk a tömbben (lásd a 17. tippet). Az iménti kezdeti értékadással egy egydimenziós tömböt hoztunk létre, amiben 100 darab egészet tárolunk. A tömb által elfoglalt memóriaterületen csupa 0 érték található. Az értéktípusokat tartalmazó tömböknél csupa 0, a hivatkozási típusokat tároló tömböknél csupa null érték lesz a memóriában. Minden egyes elemet a hozzá tartozó index segítségével érhetünk el:

```
int j = _numbers[ 50 ];
```

A tömb elérésén kívül be is járhatjuk azt a `foreach` vagy egy felsorolás segítségével (lásd a 11. tippet):

```
foreach ( int i in _numbers )
  Console.WriteLine( i.ToString( ) );

// vagy:
IEnumerator it = _numbers.GetEnumerator( );
while( it.MoveNext( ))
{
  int i = (int) it.Current;
  Console.WriteLine( i.ToString( ) );
}
```

Ha egy sor egymáshoz hasonló objektumot tárolunk, akkor ehhez mindig tömböt használjunk. Gyakran előfordul azonban, hogy az adatszerkezet bonyolultabb gyűjtemény. Ilyen-

kor könnyű engedni a csábításnak, és visszanyúlni a C stílusú fogazott, vagyis tömböket tartalmazó tömbökhöz. Néha pontosan erre is van szükség. A külső gyűjtemény minden eleme egy belső tömb:

```
public class MyClass
{
  // Fogazott tömb bevezetése:
  private int[] [] _jagged;

  public MyClass()
  {
    // Külső tömb létrehozása:
    _jagged = new int[5][];

    // A belső tömbök létrehozása:
    _jagged[0] = new int[5];
    _jagged[1] = new int[10];
    _jagged[2] = new int[12];
    _jagged[3] = new int[7];
    _jagged[4] = new int[23];
  }
}
```

A belső egydimenziós tömbök más-más méretűek lehetnek, mint a külső tömb. A fogazott tömböket akkor használjuk, ha különböző méretű tömbökből álló tömbre van szükségünk. A fogazott tömbök hátránya, hogy az oszlop szerinti bejárásuk nem elég hatékony. Ha meg szeretnénk vizsgálni a harmadik oszlopot egy fogazott tömb minden sorában, akkor minden egyes elérésnél két kikeresésre van szükségünk. A 0. sor 3. oszlopában található elem elhelyezkedésének semmi köze az 1. sor 3. oszlopában található elem helyéhez. Az oszlop szerinti bejárást hatékonyan csak többdimenziós tömbökkel lehet elvégezni. A régi motoros C és C++ programozók úgy hozták létre a saját két- vagy többdimenziós tömbjeiket, hogy leképezték azokat egy egydimenziós tömbbe. A hajdani C és C++ programozóknak ismerős ez a jelölési forma:

```
double num = MyArray[ i * rowLength + j ];
```

A világ másik fele pedig ezt kedveli jobban:

```
double num = MyArray[ i, j ];
```

A C és a C++ nem támogatta a többdimenziós tömböket. A C# viszont igen. Használjuk a többdimenziós tömbök jelölésrendszerét, ami világosabb lesz a számunkra és a fordító számára is, amikor ténylegesen többdimenziós szerkezetet hozunk létre. Többdimenziós tömböket az egydimenziós tömböknél használt szokásos jelölés kiterjesztésével hozhatunk létre:

```
private int[ , ] _multi = new int[ 10, 10 ];
```

Ez a kifejezés egy kétdimenziós tömböt vezet be. A tömb 10x10, vagyis 100 elemből áll. A többdimenziós tömböknél az egyes dimenziók mérete állandó. A fordító ezt a tulajdonságot kihasználva hatékonyabb értékadó kódot képes létrehozni. A fogazott tömbök kezdőbeállításához több beállító utasításra van szükség. A korábban bemutatott egyszerű példában öt ilyen utasítás kell. A nagyobb vagy még több dimenzióból álló tömböknél még több kódra van szükség a tömb kezdeti létrehozásához, és ezt a kódot magunknak kell megírnunk. A többdimenziós tömböknél viszont csak egy helyen, a beállító utasításban kell megadnunk az újabb dimenziókat. A többdimenziós tömbök ráadásul az elemek kezdőbeállítását is hatékonyabban végzik el. Az értéktípusokat tartalmazó tömböknél minden érvényes indexnél lesz egy érték, ami csupa 0 értéket fog tartalmazni. A hivatkozási típusokat tároló tömböknél minden elemnél null fog szerepelni. A tömbökből álló tömbökben ezzel szemben null belső tömbök lesznek.

A többdimenziós tömbök bejárása szinte mindig gyorsabb, mint a fogazott tömbök bejárása, különösen az oszloponkénti vagy az átlós bejárásnál. A fordító mutatóműveleteket használhat a tömb bármely dimenzióján. A fogazott tömböknél minden egydimenziós tömbnél meg kell találni a helyes értéket.

A többdimenziós tömböket sok szempontból ugyanúgy használhatjuk, mint a többi gyűjteményt. Tegyük fel, hogy egy olyan játékprogramot írunk, amit egy sakktáblán játszanak. Ehhez egy 64 négyzetből álló négyzetrácsos táblát hoznánk létre:

```
private Square[ , ] _theBoard = new Square[ 8, 8 ];
```

Ez a beállítás hozza létre a `Squares` (Négyzetek) tömb elemeihez szükséges tárhelyet. Feltéve, hogy a `Square` hivatkozási típus, maguk a négyzetek még nem jönnek létre, így a tömb minden eleme null lesz. Az elemek beállításához végig kell mennünk a tömb mindkét dimenzióján:

```
for ( int i = 0; i < _theBoard.GetLength( 0 ); i++ )
  for( int j = 0; j < _theBoard.GetLength( 1 ); j++ )
    _theBoard[ i, j ] = new Square( );
```

A többdimenziós tömbök bejárása rugalmas. Egy adott elemhez hozzáférhetünk a tömbindexek segítségével:

```
Square sq = _theBoard[ 4, 4 ];
```

Ha a teljes gyűjteményt be kell járnunk, akkor használhatunk egy bejárót (iterator):

```
foreach( Square sq in _theBoard )
  sq.PaintSquare( );
```

Hasonlítsuk ezt össze azzal, amit egy fogazott tömbnél kellene megírnunk:

```
foreach( Square[] row in _theBoard )
  foreach( Square sq in row )
    sq.PaintSquare( );
```

A fogazott tömböknél minden újabb dimenzió egy újabb `foreach` utasítást jelent. A többdimenziós tömböknél viszont egyetlen `foreach` utasítás elegendő ahhoz, hogy a program ellenőrizze az egyes dimenziók méretét, és végigjárja a tömb összes elemét. A `foreach` utasítás konkrét kódot hoz létre a tömb bejárásához a tömb dimenzióinak felhasználásával. A `foreach` ciklus ugyanazt a kódot állítja elő, mintha ezt írtuk volna:

```
for ( int i = _theBoard.GetLowerBound( 0 );
  i <= _theBoard.GetUpperBound( 0 ); i++ )
  for( int j = _theBoard.GetLowerBound( 1 );
    j <= _theBoard.GetUpperBound( 1 ); j++ )
    _theBoard[ i, j ].PaintSquare( );
```

Ez nem tűnik túl hatékonynak, ha a `GetLowerBound` és a `GetUpperBound` hívásokra gondolunk a cikluson belül, de valójában ez a leghatékonyabb szerkezet. A JIT fordító elég jól ismeri a tömb osztályt ahhoz, hogy a gyorstárába helyezze a tömb korlátait, és felismerje, hogy a belső korlátellenőrzések elhagyhatók (lásd a 11. tippet).

A tömb osztály használatának van két hátránya, aminek következtében érdemes megvizsgálnunk a .NET keretrendszer többi gyűjteményes osztályát is. Az első hátrány a tömbök méretezhetősége, illetve annak hiánya. A tömböket nem lehet dinamikusan átméretezni. Ha módosítanunk kell a tömb egy dimenziójának méretét, akkor új tömböt kell létrehoznunk, és át kell másolnunk oda a már meglévő elemeket. Az átméretezés időt vesz igénybe. Le kell foglalni a helyet az új tömbnek, és az összes meglévő elemet át kell másolni a régi tömbből az újba. Bár ez a másolás és áthelyezés nem olyan költséges a kezelt halmon, mint annak idején a C és C++ nyelveknél, azért így is eltart egy darabig. De ami ennél is fontosabb, hogy ez elavult adatok használatához vezethet. Vizsgáljuk meg az alábbi kódrészletet:

```
private string [] _cities = new string[ 100 ];

public void SetDataSources( )
{
  myListBox.DataSource = _cities;
}

public void AddCity( string CityName )
{
  String[] tmp = new string[ _cities.Length + 1 ];
```

```
    _cities.CopyTo( tmp, 0 );
    tmp[ _cities.Length ] = CityName;

    _cities = tmp; // tárcsere
}
```

A listamező még azután is a régi `_cities` tömböt használja, miután meghívtuk az `AddCity` függvényt. Az új város soha nem jelenik meg a listában.

Az `ArrayList` osztály a tömbre épülő magasabb szintű elvont fogalmat jelöl. Az `ArrayList` gyűjteményben az egydimenziós tömb és a láncolt lista fogalma keveredik. Az `ArrayList` gyűjteményekbe elemeket szúrhatunk be, és az `ArrayList` objektumokat át is méretezhetjük. Az `ArrayList` szinte minden feladatát a benne lévő tömbre hárítja, ami azt jelenti, hogy az `ArrayList` osztály teljesítménye nagyon hasonlít az `Array` osztály teljesítményére. Az `ArrayList` legfőbb előnye az `Array` osztállyal szemben, hogy az `ArrayList` egyszerűbben használható, amikor nem tudjuk előre, hogy mekkora lesz a gyűjteményünk. Az `ArrayList` mérete idővel nőhet vagy csökkenhet is. Az elemek másolásával és áthelyezésével járó költségek itt is jelentkeznek ugyan, de az ehhez szükséges algoritmusok kódját már megírták, és kis is próbálták. Mivel a tömb belső tárolója be van zárva az `ArrayList` objektumba, az elavult adatok használatának problémájával nem kell foglalkoznunk.
Az ügyfelek az `ArrayList` objektumra hivatkoznak a belső tömb helyett. Az `ArrayList` gyűjtemény a C++ Standard Library könyvtárában található `vector` osztály .NET keretrendszerbeli megfelelője.

A `Queue` (Sor) és a `Stack` (Verem) osztályok a `System.Array` osztályra építve kínálnak felületeket. Az osztályok felületei egyedi felületeket alkotnak az „elsőként be, elsőként ki" típusú sorhoz, illetve az „utolsóként be, elsőként ki" veremhez. Soha ne feledkezzünk meg róla, hogy ezeknek a gyűjteményeknek a belső tárolói egydimenziós tömbök, tehát átméretezésükkor az ismert költségekkel kell számolnunk.

A .NET gyűjtemények között nem találunk láncolt listát. A szemétgyűjtő hatékonysága minimálisra csökkenti azoknak a helyzeteknek a számát, amikor valóban egy listaszerkezet lenne a legjobb választás. Ha tényleg láncolt listára van szükségünk, két választási lehetőségünk van. Ha azért használunk listát, mert arra számítunk, hogy gyakran kell elemeket beszúrnunk és törölnünk, akkor használhatjuk a szótár (dictionary) osztályokat null értékekkel. Egyszerűen tároljuk a kulcsokat. Használhatjuk a `ListDictionary` osztályt is, ami egyetlen láncolt, kulcs–érték párokból álló listát valósít meg, vagy a `HybridDictionary` osztályt, ami kisebb gyűjteményekhez a `ListDictionary` osztályt használja, nagyobb gyűjteményeknél pedig átvált egy `Hashtable` osztályra. Ezeket a gyűjteményeket sok más gyűjteménnyel egyetemben a `System.Collection.Specialized` névtérben találjuk.
Ha viszont egy felhasználó által szabályozott sorrendhez használunk listát, akkor használhatjuk az `ArrayList` gyűjteményt. Az `ArrayList` osztályban bárhova beszúrhatunk elemeket, annak ellenére, hogy az belsőleg egy tömbben tárolja az elemeket.

Két másik osztály támogatja a szótár alapú gyűjteményeket: ezek a `SortedList` és a `Hashtable`. Mindkettő kulcs–érték párokból áll. A `SortedList` rendezi a kulcsokat, a `Hashtable` viszont nem. A `Hashtable` gyorsabban keres egy adott kulcs alapján, a `SortedList` viszont biztosítja az elemek kulcs szerinti rendezett bejárását. A `Hashtable` a kulcsokat a kulcsobjektumok hasítóértékei alapján keresi ki. A keresési művelet állandó, `O(1)` idejű, ha a hasítókulcs elég hatékony. A rendezett lista egy bináris keresőalgoritmussal keresi ki a kulcsokat. Ez egy logaritmikus, `O(ln n)` idejű művelet.

Végezetül van még egy `BitArray` osztály is. Ahogy a neve is utal rá, bitértékeket tárol. A `BitArray` tárolója egy egészekből álló tömb. Mindegyik egész 32 bináris értéket tárol. Ettől a `BitArray` osztály kellően tömör lesz, de ez egyben a teljesítmény romlásához is vezet. A `BitArray` osztályban minden egyes `get` és `set` műveletnek bitműveleteket kell végrehajtania azon az egész értéken, amely a keresett értéken kívül még 31 másik bitet is tartalmaz. A `BitArray` osztályban olyan tagfüggvények vannak, amelyek egyszerre több értéken végeznek bitenkénti `OR`, `XOR`, `AND` és `NOT` műveleteket. Ezek a tagfüggvények egy `BitArray` paramétert várnak, és a segítségükkel gyorsan maszkolható a `BitArray` több bitje is. A `BitArray` a bitműveletekre optimalizált tároló. Akkor használjuk, amikor olyan jelzőbiteket tárolunk, amelyeket gyakran kell maszkolnunk. Soha ne használjuk az általánosabb logikai (`Boolean`) értékeket tartalmazó tömbök helyett!

Az `Array` osztály kivételével egyik gyűjtemény sem erősen típusos a C# nyelv 1.x változataiban. Mindegyik `System.Object` hivatkozásokat tárol. A C# általános gyűjteményei között megtalálhatjuk ezeknek a szerkezeteknek az újabb változatait, amelyek mind az új általános elemekre (generics) fognak épülni. Ez lesz majd a legalkalmasabb módszer a típusbiztos gyűjtemények létrehozására. Addig is a jelenlegi `System.Collections` névtérben ott vannak azok az elvont osztályok, amelyek segítségével létrehozhatjuk a saját típusbiztos felületeinket a nem típusos gyűjteményekre építve. A `CollectionBase` és a `ReadOnlyCollectionBase` alaposztályokat lista- és vektorszerkezetek készítéséhez használhatjuk. A `DictionaryBase` a kulcs–érték párokhoz kínál alaposztályt. A `DictionaryBase` egy `Hashtable`-megvalósításra épül, így teljesítményjellemzői is ahhoz hasonlóak.

Ha az osztályainkban gyűjtemények vannak, akkor általában valószínűleg elérhetővé kívánjuk tenni azokat az osztály felhasználói számára. Ezt kétféleképpen tehetjük meg: indexelőkkel és az `IEnumerable` felülettel. Emlékezzünk vissza, hogy a tipp korábbi részében bemutattam, hogy a tömb elemeit közvetlenül elérhetjük a `[]` jelölés segítségével, a tömb elemein pedig a `foreach` utasítás segítségével mehetünk végig.

Az osztályainkhoz készíthetünk többdimenziós indexelőket a C++ túlterhelt [] műveletéhez hasonló módon. A tömbökhöz hasonlóan a C# nyelvben többdimenziós indexelőket is készíthetünk:

```
public int this [ int x, int y ]
{
  get
  {
    return ComputeValue( x, y );
  }
}
```

Az indexelők támogatása általában azt jelzi, hogy a típusunkban van egy gyűjtemény. Ez azt jelenti, hogy illik támogatni az `IEnumerable` felületet. Az `IEnumerable` szabványos módszert biztosít a gyűjteményben lévő elemek bejárásához:

```
public interface IEnumerable
{
  IEnumerator GetEnumerator( );
}
```

A `GetEnumerator` tagfüggvény egy olyan objektumot ad vissza, amelyik támogatja az `IEnumerator` felületet. Az `IEnumerator` felület a gyűjtemény bejárását támogatja:

```
public interface IEnumerator
{
  object Current
  { get; }

  bool MoveNext( );

  void Reset( );
}
```

Az `IEnumerable` felületen kívül érdemes fontolóra vennünk az `IList` és az `ICollection` felületeket is, ha a típusunk egy tömböt modellez. Ha a típusunk szótár modellt követ, gondolkodjunk el az `IDictionary` felület megvalósításán. Ezeknek a hatalmas felületeknek a megvalósítását mi is elkészíthetnénk, és oldalakon keresztül magyarázhatnám még, hogy miként lehetne ezt megoldani, de van egyszerűbb megoldás. Származtassuk az osztályunkat a `CollectionBase` vagy a `DictionaryBase` osztályból, amikor egy saját, speciális gyűjteményt hozunk létre.

Nézzük át, hogy mit is tanultunk. A legjobb gyűjtemény kiválasztása mindig attól függ, hogy milyen műveleteket szeretnénk végrehajtani rajta, illetve függ még az alkalmazás méretével és sebességével kapcsolatos általános megfontolásoktól is. A legtöbb helyzetben az `Array`

osztály lesz a leghatékonyabb tároló. Az, hogy a C# nyelvben már vannak többdimenziós tömbök is, azt jelenti, hogy egyszerűbben lehet világosan modellezni a többdimenziós adatszerkezeteket anélkül, hogy ez a teljesítmény rovására menne. Amikor a programnak nagyobb rugalmasságra van szüksége, például elemek beszúrására és törlésére, használjunk bonyolultabb gyűjteményt. Végezetül pedig, mindig valósítsunk meg indexelőket és az `IEnumerable` felületet, amikor egy gyűjteményt modellező osztályt készítünk.

41. tipp

Használjunk DataSet objektumokat az egyedi struktúrák helyett

A `DataSet` objektumokról sok rosszat elmondtak már, aminek főként két oka van. Először is, az XML-sorosított `DataSet` objektumok nem működnek túl jól a nem .NET-es programokkal. Ha egy webszolgáltatás felületeként `DataSet` objektumokat használunk, az megnehezíti az együttműködést az olyan rendszerekkel, amelyek nem használják a .NET keretrendszert. Másodszor, a `DataSet` osztályok nagyon általános tárolóosztályok. A `DataSet` objektumokat helytelenül is használhatjuk, ha megkerüljük a .NET Framework típusbiztonságát. A `DataSet` ennek ellenére a modern rendszereken belül fellépő számos gyakori igényt képes kielégíteni. Ha megismerjük az erősségeit, és elkerüljük a gyenge pontjait, akkor széles körben használhatjuk ezt a típust.

A `DataSet` osztályt arra tervezték, hogy egy relációs adatbázis adatait tárolja egyfajta kapcsolat nélküli gyorstárként. Azt már tudjuk, hogy `DataTable` objektumokat tárol, amelyekben viszont sorokba és oszlopokba rendezett adatokat találunk, az adatbázis felépítésének megfelelően. Tudjuk, hogy a `DataSet` és annak tagjai támogatják az adatkötést. Talán már arra is láttunk példát, hogy miként támogatja a `DataSet` a benne lévő `DataTable` objektumok közti kapcsolatokat. Még az is lehet, hogy olyan megszorításokra is láttunk már példát, amelyekkel ellenőrizni lehet, hogy csak érvényes adatok kerüljenek a `DataSet` objektumba.

Ezzel azonban még nincs vége. A `DataSet` objektumok a tranzakciókat is támogatják az `AcceptChanges` és a `RejectChanges` tagfüggvények segítségével, és olyan `DiffGram` elemként is tárolhatók, amelyekben megtalálhatók az adatokon elvégzett korábbi módosítások. A `DataSet` objektumokat egy közös tárolóba vonhatjuk össze. A `DataSet` támogatja a nézeteket is, amelyek segítségével az adatoknak azt a részét vehetjük szemügyre, amelyek eleget tesznek egy adott keresési feltételnek. Olyan nézeteket is létrehozhatunk, amelyek több táblán is átívelnek.

Mégis vannak köztünk olyanok, akik saját tárolószerkezetet akarnak fejleszteni, ahelyett, hogy a `DataSet` osztályt használnák. A `DataSet` általános tároló, a teljesítményt pedig kicsit visszafogja ez az általánosság. A `DataSet` nem erősen típusos tároló. A `DataTable`

objektumok gyűjteménye egy szótár, a táblában található oszlopok gyűjteményeké szintén. Az elemeket System.Object hivatkozásként tároljuk. Ezzel oda jutunk, hogy ilyen szerkezeteket írunk:

```
int val = ( int )MyDataSet.Tables[ "table1" ].
  Rows[ 0 ][ "total" ];
```

Az erősen típusos C# gondolkodásmóddal nem igazán fér össze ez a szerkezet. Ha elgépeljük a table1 vagy a total kifejezést, futásidejű hibát kapunk. Az adatelemhez csak típusátalakítás után férünk hozzá. Ha belegondolunk, és ezeket a hibalehetőségeket megszorozzuk azzal, hogy hányszor kell hozzáférnünk egy DataSet elemeihez, akkor valószínűleg hevesen keresni kezdünk egy erősen típusos megoldást. Ezért kipróbálunk egy típusos DataSet objektumot. Úgy tűnhet, hogy pont erre van szükségünk:

```
int val = MyDataSet.table1.
  Rows[ 0 ].total;
```

Tökéletes. Egészen addig, amíg bele nem nézünk abba a C# által előállított kódba, ami a típusos DataSet objektumot alkotja. Beburkolja a létező DataSet objektumot, majd típusos hozzáférést biztosít az adatokhoz, a DataSet gyengén típusos elérhetősége mellett. Az ügyfelek továbbra is hozzáférhetnek a gyengén típusos API-hoz. Ez bizony nem optimális megoldás.

Azt javaslom, hogy fogadjuk el, hogy ez így van. Hogy rávilágítsak arra, hogy mit is kell feláldoznunk, egy saját gyűjtemény példáján keresztül bemutatom, hogyan valósítanak meg a DataSet osztályon belül néhány szolgáltatást. Most azt gondoljuk, hogy nem is olyan nagy ügy. Az jár a fejünkben, hogy nincs is szükség a DataSet összes szolgáltatására, tehát biztos nem kerül annyi munkába az egész. Na jó, felőlem kezdhetjük.

Képzeljük el, hogy létre kell hoznunk egy címeket tároló gyűjteményt. Az egyes elemeknek támogatniuk kell az adatkötést, tehát létrehozunk egy struktúrát nyilvános tulajdonságokkal:

```
public struct AddressRecord
{
  private string _street;
  public string Street
  {
    get { return _street; }
    set { _street = value; }
  }

  private string _city;
  public string City
  {
```

```
    get { return _city; }
    set { _city = value; }
  }

  private string _state;
  public string State
  {
    get { return _state; }
    set { _state = value; }
  }

  private string _zip;
  public string Zip
  {
    get { return _zip; }
    set { _zip = value; }
  }
}
```

Ezután létre kell hoznunk a gyűjteményt. Típusbiztos gyűjteményt szeretnénk, ezért a `CollectionBase` osztályból származtatunk egyet:

```
public class AddressList : CollectionBase
{
}
```

A `CollectionBase` támogatja az `IList` felületet, ezért használhatjuk forrásként az adatkötéshez. És ekkor felfedezzük az első problémát: az összes adatkötési művelet sikertelen lesz, ha a címlistánk üres. A `DataSet` esetében nem így volt. Az adatkötés visszatekintésen (reflection) alapuló, késői kötésű kódot használ. A vezérlő visszatekintéssel tölti be a lista első elemét, majd visszatekintéssel határozza meg a típusát és azokat a tulajdonságokat, amelyek az adott típus tagjai. Innen tudja a `DataGrid`, hogy milyen oszlopokat kell beszúrnia. Megkeresi a gyűjtemény első elemének összes nyilvános tulajdonságát, majd megjeleníti azokat. Ha a gyűjtemény üres, akkor ez nem működik. Erre a problémára két megoldás létezik. Az első az egyszerűbb, de csúnyább megoldás: ne engedjük meg az üres listákat. A második elegánsabb, de időigényesebb: valósítsuk meg az `ITypedList` felületet. Az `ITypedList` felületnek van két tagfüggvénye, amelyek leírják a gyűjteményben található típusokat. A `GetListName` egy emberi olvasásra szánt karakterláncot ad vissza, ami leírja a listát. A `GetItemProperties` visszatérési értéke egy olyan `PropertyDescriptor` elemekből álló lista, amelyek leírják az összes olyan tulajdonságot, amiből oszlop lesz az adatrácsban:

```
public class AddressList : CollectionBase
{
  public string GetListName(
```

```
    PropertyDescriptor[ ] listAccessors )
  {
    return "AddressList";
  }

  public PropertyDescriptorCollection
    GetItemProperties(
    PropertyDescriptor[ ] listAccessors)
  {
    Type t = typeof( AddressRecord );
    return TypeDescriptor.GetProperties( t );
  }
}
```

Egyre jobb. Most már van egy gyűjteményünk, ami támogatja az egyszerű adatkötést. Viszont számtalan más szolgáltatás hiányzik még. A következő szükséges szolgáltatás a tranzakciók támogatása. Ha `DataSet` objektumokat használtunk volna, akkor a felhasználók az Esc billentyű lenyomásával elvethetnék a `DataGrid` egy adott során végrehajtott módosításaikat. A felhasználó például elgépelhetné a várost, azután az Esc leütésével visszaállíthatná az eredeti értéket. A `DataGrid` a hibaüzeneteket is támogatja. Bármely oszlophoz hozzákapcsolhatunk egy `ColumnChanged` eseménykezelőt, ami elvégezhetné az adott oszlop adatainak érvényesítését. Az ország kódjának például mindig egy kétbetűs rövidítésnek kell lennie. A `DataSet` keretrendszert használva ennek így néz ki a kódja:

```
ds.Tables[ "Addresses" ].ColumnChanged +=new
  DataColumnChangeEventHandler( ds_ColumnChanged );

private void ds_ColumnChanged( object sender,
  DataColumnChangeEventArgs e )
{
  if ( e.Column.ColumnName == "State" )
  {
    string newVal = e.ProposedValue.ToString( );
    if ( newVal.Length != 2 )
    {
      e.Row.SetColumnError( e.Column,
        "State abbreviation must be two letters" );
      e.Row.RowError = "Error on State";
    }
    else
    {
      e.Row.SetColumnError( e.Column,
        "" );
      e.Row.RowError = "";
    }
  }
}
```

A saját gyártmányú gyűjteményünk esetében még komoly munka vár ránk, mire mindkét szolgáltatást meg tudjuk valósítani. Módosítanunk kell az `AddressRecord` struktúrát, hogy az támogasson két felületet, az `IEditableObject` és az `IDataErrorInfo` felületeket. Az `IEditableObject` nyújtja az objektumnak a tranzakciók támogatását, az `IDataErrorInfo` pedig a hibakezelő eljárásokat. A tranzakciók támogatásához módosítanunk kell az adattároló szerkezetünket, hogy beépítsük a visszavonás lehetőségét. Több oszlopban is lehetnek hibák, ezért a tárolónknak tartalmaznia kell egy hibagyűjteményt minden egyes oszlophoz. Itt van tehát az `AddressRecord` módosított kódja:

```
public class AddressRecord : IEditableObject, IDataErrorInfo
{
    private struct AddressRecordData
    {
      public string street;
      public string city;
      public string state;
      public string zip;
    }

    private AddressRecordData permanentRecord;
    private AddressRecordData tempRecord;

    private bool _inEdit = false;
    private IList _container;

    private Hashtable errors = new Hashtable();

    public AddressRecord( AddressList container )
    {
      _container = container;
    }

    public string Street
    {
      get
      {
        return ( _inEdit ) ? tempRecord.street :
          permanentRecord.street;
      }
      set
      {
        if ( value.Length == 0 )
          errors[ "Street" ] = "Street cannot be empty";
        else
        {
          errors.Remove( "Street" );
        }
```

```
      if ( _inEdit )
        tempRecord.street = value;
      else
      {
        permanentRecord.street = value;
        int index = _container.IndexOf( this );
        _container[ index ] = this;
      }
    }
  }

  public string City
  {
    get
    {
      return ( _inEdit ) ? tempRecord.city :
        permanentRecord.city;
    }
    set
    {
      if ( value.Length == 0 )
        errors[ "City" ] = "City cannot be empty";
      else
      {
        errors.Remove( "City" );
      }
      if ( _inEdit )
        tempRecord.city = value;
      else
      {
        permanentRecord.city = value;
        int index = _container.IndexOf( this );
        _container[ index ] = this;
      }
    }
  }

  public string State
  {
    get
    {
      return ( _inEdit ) ? tempRecord.state :
        permanentRecord.state;
    }
    set
    {
      if ( value.Length == 0 )
        errors[ "State" ] = "City cannot be empty";
      else
      {
```

```
        errors.Remove( "State" );
      }
      if ( _inEdit )
        tempRecord.state = value;
      else
      {
        permanentRecord.state = value;
        int index = _container.IndexOf( this );
        _container[ index ] = this;
      }
    }
  }

  public string Zip
  {
    get
    {
      return ( _inEdit ) ? tempRecord.zip :
        permanentRecord.zip;
    }
    set
    {
      if ( value.Length == 0 )
        errors["Zip"] = "Zip cannot be empty";
      else
      {
        errors.Remove ( "Zip" );
      }
      if ( _inEdit )
        tempRecord.zip = value;
      else
      {
        permanentRecord.zip = value;
        int index = _container.IndexOf( this );
        _container[ index ] = this;
      }
    }
  }
  public void BeginEdit( )
  {
    if ( ( ! _inEdit ) && ( errors.Count == 0 ) )
      tempRecord = permanentRecord;
    _inEdit = true;
  }

  public void EndEdit( )
  {
    // A módosítást nem lehet befejezni, amíg hibák vannak
    if ( errors.Count > 0 )
      return;
```

```
    if ( _inEdit )
      permanentRecord = tempRecord;
    _inEdit = false;
  }

  public void CancelEdit( )
  {
    errors.Clear( );
    _inEdit = false;
  }

  public string this[string columnName]
  {
    get
    {
      string val = errors[ columnName ] as string;
      if ( val != null )
        return val;
      else
        return null;
    }
  }

  public string Error
  {
    get
    {
      if ( errors.Count > 0 )
      {
        System.Text.StringBuilder errString = new
          System.Text.StringBuilder();
        foreach ( string s in errors.Keys )
        {
          errString.Append( s );
          errString.Append( ", " );
        }
        errString.Append( "Have errors" );
        return errString.ToString( );
      }
      else
        return "";
    }
  }
}
```

Ez a kód jópár oldal hosszú, és mindez csak azért, hogy olyan szolgáltatásokat támogassunk, amelyeket a `DataSet` osztályban már megvalósítottak. Sőt ezzel még nincs is meg a `DataSet` összes szolgáltatása. Az új rekordok interaktív hozzáadása a gyűjteményhez, il-

letve a tranzakciók támogatása további mutatványokkal jár a `BeginEdit`, `CancelEdit` és `EndEdit` függvényeknél. Meg kell vizsgálnunk, hogy a `CancelEdit` hívása mikor történik egy új objektumon és mikor egy módosított objektumon. A `CancelEdit` függvénynek törölnie kell az új objektumot a tárolóból, ha az objektumot az utolsó `BeginEdit` óta hoztuk létre. Ehhez tovább kell módosítanunk az `AddressRecord` struktúrát, és néhány eseménykezelőt is hozzá kell adnunk az `AddressList` osztályhoz.

Végül pedig ott van még az `IBindingList` felület. Ebben a felületben 20-nál is több tagfüggvény és tulajdonság van, amelyeket a vezérlők lekérdeznek, hogy megtudják a lista képességeit. Az `IBindingList` felületet meg kell valósítanunk a csak olvasható listákhoz, az interaktív rendezésekhez, illetve a keresések támogatásához. Ezt még azelőtt, hogy a navigáció és a hierarchiák egyáltalán szóba jöhetnének. Mindezt a kódot már tényleg nem vagyok hajlandó egy példával szemléltetni.

Átjutva ezen a néhány oldalon tegyük fel magunknak ismét a kérdést: még mindig akarunk saját gyűjteményt? Vagy inkább használjunk a `DataSet` osztályt? Hacsak nem az alkalmazásnak a teljesítmény szempontjából létfontosságú részét képezi a gyűjteményünk, vagy hordozható formátumra van szükségünk, használjuk a `DataSet` osztályt, különösen a típusos változatát. Rengeteg időt takaríthatunk meg ezzel. Lehet vitatkozni, hogy a `DataSet` nem a legszebb példája az objektumközpontú programozásnak, és hogy a típusos `DataSet` osztályok még több szabályt szegnek meg. De ez azon esetek egyike, amikor a termelékenység sokkal fontosabb, mint egy saját kezűleg szépen megírt kód.

42. tipp

Használjunk jellemzőket a visszatekintés egyszerűsítéséhez

Amikor visszatekintésre támaszkodó rendszereket építünk, érdemes egyedi jellemzőket meghatározni a felhasználni kívánt típusokhoz, tagfüggvényekhez és tulajdonságokhoz, hogy azok könnyebben elérhetők legyenek. Az egyedi jellemzők jelzik, hogy milyen felhasználásra szántuk a tagfüggvényeket futásidőben. A jellemzők ellenőrizhetik a cél néhány tulajdonságát. Ezeknek a tulajdonságoknak az ellenőrzése minimálisra csökkenti az esélyt arra, hogy elgépelünk valamit, ami a visszatekintésnél előfordulhat.

Tegyük fel, hogy egy olyan rendszert építünk, amivel menüpontokat és parancskezelőket lehet hozzáadni egy futó programrendszerhez. A követelmények elég egyszerűek. Dobjunk egy szerelvényt egy könyvtárba, amit a program majd megtalál, és ennek alapján új menüpontokat ad az új parancshoz. Ez azon feladatok egyike, amelyeket a legjobb visszatekintéssel megoldani. A főprogramnak olyan szerelvényekkel kell együttműködnie, amelyeket még nem írtunk meg, az új bővítmények pedig olyan szolgáltatásokat tartalmaznak, amiket nem lehet egyszerűen kódolni egy felület segítségével.

Kezdjük azzal a kóddal, amire a bővítmény-keretrendszer megalkotásához lesz szükségünk. Be kell töltenünk egy szerelvényt az `Assembly.LoadFrom()` függvény segítségével. Meg kell keresnünk azokat a típusokat, amelyek valószínűleg képesek menüket kezelni. Létre kell hoznunk egy megfelelő típusú objektumot.

Erre a `Type.GetConstructor()` és a `ConstructorInfo.Invoke()` a megfelelő eszközök. Keresnünk kell egy olyan tagfüggvényt, ami passzol a menü parancskezelőjének aláírásához. Ha mindezzel végeztünk, akkor még azt is ki kell találnunk, hogy a menüben hová kerüljön az új szöveg, és hogy mi legyen az.

A jellemzőkkel egyszerűbben lehet elvégezni ezeket a feladatokat. Egyedi jellemzőkkel felcímkézve a különböző osztályokat és eseménykezelőket, lényegesen egyszerűbben találhatjuk meg és telepíthetjük a lehetséges parancskezelőket. A jellemzők és a visszatekintés együttes használatával minimálisra csökkenthetjük a 43. tippben bemutatott kockázatokat.

Az első feladat annak a kódnak a megírása, amelyik megkeresi és betölti a bővítményként szolgáló szerelvényeket. Tegyük fel, hogy a bővítmények egy alkönyvtárban vannak a főprogram könyvtára alatt. A szerelvények keresését és betöltését végző kód egyszerű:

```
// Megkeressük az összes szerelvényt az Add-ins (Bővítmények)
// könyvtárban
string AddInsDir = string.Format( "{0}/Addins",
  Application.StartupPath );
string[] assemblies = Directory.GetFiles( AddInsDir,
  "*.dll" );
foreach ( string assemblyFile in assemblies )
{
  Assembly asm = Assembly.LoadFrom( assemblyFile );
  // Megkeressük és telepítjük a szerelvényben lévő parancskezelőket
}
```

Ezután le kell cserélnünk az utolsó megjegyzést arra a kódra, amelyik megkeresi és telepíti a parancskezelőket megvalósító osztályokat. A szerelvény betöltése után a visszatekintés segítségével megtalálhatjuk az összes exportált típust a szerelvényben. A jellemzők segítségével határozzuk meg, hogy mely típusokban vannak a parancskezelők, illetve hogy mely tagfüggvények azok. Egy jellemzőosztály jelöli azokat a típusokat, amelyekben parancskezelők vannak:

```
// Meghatározzuk a parancskezelőt jelző egyedi jellemzőt:
[AttributeUsage( AttributeTargets.Class )]
public class CommandHandlerAttribute : Attribute
{
  public CommandHandlerAttribute( )
  {
  }
}
```

Ez a jellemző az összes kód, amit meg kell írnunk az egyes parancsok megjelöléséhez. A jellemzőosztályokat mindig jelöljük meg az `AttributeUsage` jellemzővel. Ez árulja el a többi programozónak és a fordítónak, hogy hol használható a jellemző. Az előző példában azt látjuk, hogy a `CommandHandlerAttribute` jellemzőt osztályoknál lehet alkalmazni, más nyelvi elemeknél nem.

A `GetCustomAttributes` függvény hívásával határozhatjuk meg, hogy egy adott típusnak van-e `CommandHandlerAttribute` jellemzője. Csak az ilyen típusok lehetnek bővítmények:

```
// Megkeressük az összes szerelvényt az Add-ins (Bővítmények)
// könyvtárban
string AddInsDir = string.Format( "{0}/Addins",
Application.StartupPath);
string[] assemblies = Directory.GetFiles( AddInsDir,
  "*.dll" );
foreach ( string assemblyFile in assemblies )
{
  Assembly asm = Assembly.LoadFrom( assemblyFile );
  // Megkeressük és telepítjük a szerelvényben lévő parancskezelőket
  foreach( System.Type t in asm.GetExportedTypes( ))
  {
    if (t.GetCustomAttributes(
      typeof( CommandHandlerAttribute ), false ).Length > 0 )
    {
      // Ennél a típusnál megtaláltuk a parancskezelő jellemzőt,
      // vagyis a típus megvalósít egy parancskezelőt
      // Beállítjuk és beszúrjuk
    }
    // Különben nem parancskezelővel van dolgunk. Átugorjuk.
  }
}
```

Most pedig készítsünk egy újabb jellemzőt a parancskezelők kereséséhez. Könnyen előfordulhat, hogy egy típus több parancskezelőt is megvalósít, ezért létrehozunk egy új jellemzőt, amit a bővítmények készítői az egyes parancskezelőkhöz csatolhatnak. Ennek a jellemzőnek paraméterei is lesznek, amelyek meghatározzák, hogy hová kerüljenek az új parancshoz tartozó menüpontok. Minden eseménykezelő csak egy adott parancsot kezel, ami a menü egy adott pontján található. A parancskezelők felcímkézéséhez meghatározunk egy olyan jellemzőt, ami egy tulajdonságot parancskezelőként jelöl meg, és megadja a menüpont szövegét, illetve a szülő menüpont szövegét. A `DynamicCommand` jellemzőt két paraméterrel készítjük el: az egyik a parancs (menüpont) szövege, a másik a szülő menü szövege. A jellemzőosztálynak lesz egy konstruktora, ami beállítja a két karakterláncot a két menüponthoz.

Ezek a karakterláncok írható-olvasható tulajdonságként is elérhetők:

```
[AttributeUsage( AttributeTargets.Property ) ]
public class DynamicMenuAttribute : System.Attribute
{
  private string _menuText;
  private string _parentText;

  public DynamicMenuAttribute( string CommandText,
    string ParentText )
  {
    _menuText = CommandText;
    _parentText = ParentText;
  }

  public string MenuText
  {
    get { return _menuText; }
    set { _menuText = value; }
  }

  public string ParentText
  {
    get { return _parentText; }
    set { _parentText = value; }
  }
}
```

Ezt a jellemzőosztályt megcímkéztük, hogy jelezzük, hogy csak tulajdonságoknál lehet alkalmazni. A parancskezelőt egy tulajdonságként kell elérhetővé tenni abban az osztályban, amelyik hozzáférést biztosít a parancskezelőhöz. Ezzel a módszerrel egyszerűbb megtalálni a parancskezelő kódját és induláskor a programhoz csatolni.

Most készítünk egy ilyen típusú objektumot, megkeressük a parancskezelőket, majd hozzácsatoljuk azokat az új menüpontokhoz. Biztos kitaláltuk, hogy a parancskezelő tulajdonságok kikereséséhez és felhasználásához a jellemzők és a visszatekintés kombinációját fogjuk használni:

```
// Az első kódpéldát bővítjük:
// Megkeressük a típusokat a szerelvényben
foreach( Type t in asm.GetExportedTypes( ) )
{
  if (t.GetCustomAttributes(
    typeof( CommandHandlerAttribute ), false).Length > 0 )
  {
    // Találtunk egy parancskezelő típust:
    ConstructorInfo ci =
```

```
          t.GetConstructor( new Type[0] );
        if ( ci == null ) // No default ctor
          continue;
        object obj = ci.Invoke( null );
        PropertyInfo [] pi = t.GetProperties( );

        // Megkeressük azokat a tulajdonságokat, amelyek parancskezelők
        foreach( PropertyInfo p in pi )
        {
          string menuTxt = "";
          string parentTxt = "";
          object [] attrs = p.GetCustomAttributes(
            typeof ( DynamicMenuAttribute ), false );
          foreach ( Attribute at in attrs )
          {
            DynamicMenuAttribute dym = at as
              DynamicMenuAttribute;
            if ( dym != null )
            {
              // Ez egy parancskezelő
              menuTxt = dym.MenuText;
              parentTxt = dym.ParentText;
              MethodInfo mi = p.GetGetMethod();
              EventHandler h = mi.Invoke( obj, null )
                as EventHandler;
              UpdateMenu( parentTxt, menuTxt, h );
            }
          }
        }
      }
    }

private void UpdateMenu( string parentTxt, string txt,
  EventHandler cmdHandler )
{
  MenuItem menuItemDynamic = new MenuItem();
  menuItemDynamic.Index = 0;
  menuItemDynamic.Text = txt;
  menuItemDynamic.Click += cmdHandler;

  // Megkeressük a szülő menüpontot
  foreach ( MenuItem parent in mainMenu.MenuItems )
  {
    if ( parent.Text == parentTxt )
    {
      parent.MenuItems.Add( menuItemDynamic );
      return;
    }
  }
```

```
    // Nem létezik szülő:
    MenuItem newDropDown = new MenuItem();
    newDropDown.Text = parentTxt;
    mainMenu.MenuItems.Add( newDropDown );
    newDropDown.MenuItems.Add( menuItemDynamic );
  }
```

Most felépítünk egy egyszerű parancskezelőt. Először felcímkézzük a típust a `CommandHandler` jellemzővel. Ahogy itt is látható, általában az a szokás, hogy az `Attribute` kimarad a névből, amikor a jellemzőt egy elemhez csatoljuk:

```
[ CommandHandler ]
public class CmdHandler
{
  // Nemsokára itt lesz a megvalósítás
}
```

A `CmdHandler` osztály belsejében megadunk egy tulajdonságot, ami visszaadja a parancskezelőt. Ezt a tulajdonságot kell megjelölnünk a `DynamicMenu` jellemzővel:

```
[DynamicMenu( "Test Command", "Parent Menu" )]
public EventHandler CmdFunc
{
  get
  {
    if ( theCmdHandler == null )
      theCmdHandler = new System.EventHandler
        (this.DynamicCommandHandler);
    return theCmdHandler;
  }
}

private void DynamicCommandHandler(
  object sender, EventArgs args )
{
  // Tartalom elhagyva
}
```

Ennyi. Ebből a példából jól látható, hogyan lehet felhasználni a jellemzőket a visszatekintést használó programszerkezetek leegyszerűsítéséhez. Minden olyan típust megjelöltünk egy jellemzővel, amiben volt egy dinamikus parancskezelő. Ezáltal egyszerűbben lehetett megtalálni a parancskezelőket a szerelvény dinamikus betöltésekor. Az `AttributeTargets` alkalmazásával (ami maga is egy jellemző) korlátozni tudtuk, hogy hol legyen alkalmazható a dinamikus parancs jellemző. Ezzel leegyszerűsödik az az egyébként nehézkes feladat,

hogy megtaláljuk a keresett típusokat a dinamikusan betöltött szerelvényben. Nagyban csökkentettük annak a veszélyét is, hogy nem a megfelelő típust használjuk. A kód így sem túl egyszerű, de kissé könnyebben emészthető, mint jellemzők nélkül.

A jellemzők mutatják meg a futásidőre vonatkozó szándékainkat. Egy jellemzővel megcímkézve egy elemet, jelezhetjük a felhasználási módját és egyszerűbben meg tudjuk találni azt futásidőben. A jellemzők nélkül meg kellene határoznunk egyfajta névadási szabályt, hogy futásidőben meg tudjuk találni a felhasználni kívánt típusokat és egyéb elemeket, a névadási szabályok viszont az emberi hiba forrásai. Ha a szándékainkat jellemzők segítségével jelezzük, azzal a felelősséget átháríthatjuk a fejlesztőről a fordítóra. A jellemzőket mindig csak adott típusú nyelvi elemekkel lehet használni. A jellemzők ezen kívül nyelvtani és jelentéstani információkat is hordoznak.

A visszatekintéssel olyan dinamikus kódot készíthetünk, amit át lehet állítani kint a terepen. A futásidejű hibák lehetőségét is csökkentjük azzal, ha olyan jellemzőosztályokat tervezünk és valósítunk meg, amelyek olyan típusok, tagfüggvények és tulajdonságok bevezetésére kényszerítik a fejlesztőket, amelyeket dinamikusan lehet használni. Ez pedig növeli annak az esélyét, hogy olyan programokat írunk, amivel elégedettek lesznek a felhasználóik.

43. tipp

Ne vigyük túlzásba a visszatekintés használatát

A bináris összetevők építése néha azzal jár, hogy késői kötést és visszatekintést kell alkalmaznunk, hogy megtaláljuk a számunkra megfelelő működést nyújtó kódot. A visszatekintés igen hatékony eszköz, ami lehetővé teszi, hogy sokkal dinamikusabb programokat írjunk. Visszatekintést használva úgy lehet frissíteni egy alkalmazást, hogy olyan összetevőket adunk hozzá, amelyek még nem léteztek az alkalmazás első kiadásakor. Ez a dolog kellemes oldala.

A nagyobb rugalmassággal azonban együtt jár a bonyolultság is, a bonyolultsággal pedig jönnek a hibalehetőségek. Amikor visszatekintést használunk, megkerüljük a C# típusbiztonságát. Az `Invoke` tagok ugyanis `System.Object` típusú paramétereket és visszatérési értékeket használnak. Nekünk kell gondoskodnunk róla, hogy a megfelelő típust használjuk futásidőben. Rövidre fogva a szót, visszatekintéssel sokkal könnyebb dinamikus programokat írni, de sajnos sokkal könnyebb hibásakat is. Gyakran azonban egy kis odafigyeléssel csökkenthetjük, vagy teljesen meg is szüntethetjük a visszatekintéseket, ha létrehozunk olyan felületmeghatározásokat, amelyek jól kifejezik az adott típussal kapcsolatos feltevéseinket.

A visszatekintés lehetőséget ad objektumpéldányok létrehozására, az objektumok tagjainak meghívására és az objektumok adattagjainak elérésére. Ezek mindennapos programozási feladatoknak tűnnek. Azok is. Semmi varázslatos nincs a visszatekintésben. Az pusz-

tán egy eszköz ahhoz, hogy dinamikusan kapcsolatba lépjünk más bináris összetevőkkel. A legtöbb esetben nincs is szükségünk a visszatekintés által kínált rugalmasságra, mert más, a karbantarthatóság szempontjából jobb lehetőségek közül is választhatunk.

Kezdjük azzal, hogy létrehozzuk egy adott típus példányait. Ugyanazt az eredményt gyakran egy osztálygyárral (class factory) is elérhetjük. Vegyük ezt a kódrészletet, ami létrehozza a `MyType` egy példányát úgy, hogy visszatekintéssel meghívja az alapértelmezett konstruktort:

```
// Használat: Új objektum létrehozása visszatekintéssel:
Type t = typeof( MyType );
MyType obj = NewInstance( t ) as MyType;


// Gyári függvény példa, a visszatekintés alapján:
object NewInstance( Type t )
{
  // Megkeressük az alapértelmezett konstruktort:
  ConstructorInfo ci = t.GetConstructor( new Type[ 0 ] );
  if ( ci != null )
    // Meghívjuk az alapértelmezett konstruktort, és visszaadjuk
    // az új objektumot
    return ci.Invoke( null );

  // Ha nem sikerült, akkor null értéket adunk vissza
  return null;
}
```

A kód visszatekintéssel vizsgálja meg a típust, majd az alapértelmezett konstruktor meghívásával hozza létre az új objektumot. Ha úgy kell létrehoznunk egy típust futásidőben, hogy semmilyen előzetes információnk nincs az adott típusról, akkor ez az egyetlen lehetőségünk. Ez törékeny kód, ami az alapértelmezett konstruktor meglétén áll vagy bukik. Akkor is le lehet fordítani, ha eltávolítjuk a `MyType` típusból annak alapértelmezett konstruktorát. Futásidőben kell ellenőrzéseket végrehajtanunk, hogy elcsípjük az esetlegesen felmerülő problémákat. Ugyanezt a műveletet végző osztálygyártó függvényt nem lehetne lefordítani, ha előtte eltávolítjuk az alapértelmezett konstruktort:

```
public MyType NewInstance( )
{
  return new MyType();
}
```

Az objektumpéldányok létrehozásához statikus gyártófüggvényeket célszerű létrehoznunk, ahelyett hogy a visszatekintésre támaszkodnánk. Ha késői kötéssel kell osztálypéldányokat készítenünk, akkor hozzunk létre gyártófüggvényeket, és jelöljük meg azokat ilyenként jellemzők segítségével (lásd a 42. tippet).

A visszatekintés egy másik lehetséges felhasználási területe egy adott típus tagjainak elérése. Futásidőben a tag nevét és a típust használva meghívhatunk egy adott függvényt:

```
// Példa a használatra:
Dispatcher.InvokeMethod( AnObject, "MyHelperFunc" );

// az elosztó tagfüggvény hívója
public void InvokeMethod ( object o, string name )
{
  // Megkeressük az adott nevű tagfüggvényeket
  MemberInfo[] myMembers = o.GetType( ).GetMember( name );
  foreach( MethodInfo m in myMembers )
  {
    // Meggyőződünk róla, hogy stimmel-e a paraméterlista:
    if ( m.GetParameters( ).Length == 0 )
      // Meghívjuk:
     m.Invoke( o, null );
  }
}
```

A fenti kódban futásidejű hibák leselkednek ránk. Ha elgépeljük a nevét, akkor a tagfüggvényt nem fogja megtalálni a program. Az alkalmazás ilyenkor semmilyen tagfüggvényt nem hív meg.

És ez csak egy egyszerű példa volt. Az `InvokeMethod` ellenállóbb változatának össze kellene hasonlítania az összes javasolt paraméter típusát a `GetParameters()` tagfüggvény által visszaadott paraméterlistával. Ez a kód olyan hosszú és csúnya lenne, hogy nem is pazaroltam rá a helyet csak azért, hogy láthassuk. Higgyük el, hogy tényleg nagyon rémes.

A visszatekintés harmadik felhasználási területe az adattagok elérése. A kód hasonlít a tagfüggvények eléréséhez:

```
// Példa a használatra:
object field = Dispatcher.RetrieveField ( AnObject,
  "MyField" );

// az elosztó osztály egy másik részében:
public object RetrieveField ( object o, string name )
{
  // Megkeressük a mezőt
  FieldInfo myField = o.GetType( ).GetField( name );
```

```
    if ( myField != null )
      return myField.GetValue( o );
    else
      return null;
  }
```

A tagfüggvény hívásához hasonlóan, ahhoz, hogy visszatekintéssel kiolvashassuk egy adattag értékét, le kell kérdeznünk a típustól a mezőt egy olyan név segítségével, ami megegyezik a kért mező nevével. Ha van ilyen, akkor a `FieldInfo` szerkezet segítségével ki lehet olvasni az adott értéket. Ez a szerkezet gyakran előfordul a keretrendszerben. A `DataBinding` (adatkötés) visszatekintéssel találja meg azokat a tulajdonságokat, amelyek a kötési művelet célpontjai. Ezekben az esetekben az adatkötés dinamikus természete nagyobb súllyal esik latba, mint az esetleges járulékos költségek.

Tehát, ha a visszatekintés ilyen fájdalmas, akkor egyszerűbb és jobb módszereket kell találnunk. Három lehetőségünk van. Az első a felületek használata. Felületekkel bármilyen megállapodást létrehozhatunk, aminek a megvalósítását elvárjuk az osztályoktól és a struktúráktól (lásd a 19. tippet). Ezzel kiválthatjuk az összes visszatekintéses kódot néhány messze világosabb kódsorral:

```
IMyInterface foo = obj as IMyInterface;
if ( foo != null)
{
  foo.DoWork( );
  foo.Msg = "work is done.";
}
```

Ha egy jellemzővel ellátott gyártófüggvénnyel kombináljuk a felületeket, akkor szinte minden olyan rendszer sokkal egyszerűbbé válik, amiről korábban azt hittük, hogy csak visszatekintéssel lehet megoldani:

```
public class MyType : IMyInterface
{
  [FactoryFunction]
  public static IMyInterface
    CreateInstance( )
  {
    return new MyType( );
  }

  #region IMyInterface
  public string Msg
  {
    get
    {
```

```
        return _msg;
      }
      set
      {
        _msg = value;
      }
    }
    public void DoWork( )
    {
      // részletek elhagyva.
    }
    #endregion
  }
```

Vessük össze ezt a kódot a korábban bemutatott, visszatekintésre épülő megoldással. Még ezekből az egyszerű példákból is kiviláglik néhány olyan, a gyenge típusosságot érintő probléma, ami a visszatekintést használó API-k sajátja. A visszatérési értékek mindig objektum típusúak. Ha meg akarjuk kapni a helyes típust, akkor el kell végeznünk egy típusátalakítást. Az ilyen műveletek kudarcot vallhatnak, és veszélyeket hordoznak magukban. A fordító által biztosított erős típusellenőrzés, amit a felületek készítésénél kapunk, sokkal világosabb és karbantarthatóbb.

Visszatekintést csak akkor használjunk, ha a hívás célpontját nem tudjuk világosan kifejezni felületek segítségével. A .NET adatkötése a típusok bármely nyilvános tulajdonságával működik. Ha felületmeghatározásokra korlátoznánk, azzal jelentősen lecsökkentenénk a hatókörét. A menükezelő példa bármely függvényt (példány- és statikus függvényeket egyaránt) megenged a parancskezelő megvalósításához. Egy felületet használva ezt a működést példány tagfüggvényekre korlátoznánk. Az `FxCopy` és az `NUnit` is (lásd a 48. tippet) sokszor használ visszatekintést. Azért teszik ezt, mert az általuk kezelt feladatok természetéből következően azokat visszatekintéssel lehet a legjobban kezelni. Az `FxCopy` megvizsgálja a teljes kódot, hogy azt összevesse szabályok egy előre megadott halmazával. Ehhez visszatekintésre van szükség. Az `NUnit`-nak meg kell hívnia az általunk készített tesztkódot. Visszatekintéssel határozza meg, hogy milyen kódot írtunk a programunk egységeinek teszteléséhez. Egy felület nem képes kifejezni az összes tagfüggvényt, amit az általunk írt bármely kódunk teszteléséhez kell használni. Az `NUnit` jellemzők segítségével találja meg a teszteket és a teszteseteket a munkája megkönnyítése érdekében (lásd a 42. tippet).

Ha felületekbe tudjuk gyűjteni azokat a tagfüggvényeket vagy tulajdonságokat, amelyeket meg akarunk hívni, akkor világosabb, karbantarthatóbb rendszerhez jutunk. A visszatekintés nagy teljesítményű, késői kötést támogató rendszer; a .NET keretrendszer a webes és windowsos vezérlők adatkötésének megvalósításához használja. Számos kevésbé általános esetben azonban az osztálygyárakat, képviselőket és felületeket használó kóddal karbantarthatóbb rendszerekhez jutunk.

44. tipp

Készítsünk teljes, alkalmazásfüggő kivételosztályokat

A kivételek segítségével úgy lehet jelezni a hibákat, hogy azokat egészen távol is kezelni lehet attól a helytől, ahol bekövetkeztek. A hiba okaival kapcsolatos információknak részletesen szerepelniük kell a kivételobjektumban. Előfordulhat, hogy menet közben át szeretnénk alakítani egy alacsonyszintű hibát egy alkalmazásfüggő hibává, méghozzá úgy, hogy közben ne veszítsünk el semmilyen információt az eredeti hibával kapcsolatban.
A C# alkalmazások készítésénél nagyon körültekintően kell eljárnunk, amikor saját kivételosztályokat készítünk.

Első lépésként meg kell ismerkednünk azzal, hogy mikor és miért érdemes új kivételosztályokat készíteni, és hogy miként lehet beszédes hierarchiákat építeni a kivételekből. Amikor a könyvtárainkat használó fejlesztők `catch` blokkokat írnak, az egyes végrehajtandó műveleteket a kivételek futásidejű típusa szerint különböztetik meg egymástól. Minden egyes kivételosztályhoz más-más műveletsor tartozhat:

```
try {
  Foo( );
  Bar( );
} catch( MyFirstApplicationException e1 )
{
  FixProblem( e1 );
} catch( AnotherApplicationException e2 )
{
  ReportErrorAndContinue( e2 );
} catch( YetAnotherApplicationException e3 )
{
  ReportErrorAndShutdown( e3 );
} catch( Exception e )
{
  ReportGenericError( e );
}
finally
{
  CleanupResources( );
}
```

Különböző `catch` blokkok létezhetnek a futásidőben egymástól eltérő típusú kivételekhez. Nekünk, mint a könyvtárak szerzőjének, különböző kivételosztályokat kell létrehoznunk, ha az egyes `catch` blokkok más-más műveletekből állhatnak. Ha nem így teszünk, a felhasználókat jól benne hagyjuk a pácban. Megtehetjük, hogy amikor bekövetkezik egy hiba, és kivált egy kivételt, akkor egyszerűen megvonjuk a vállunkat, és kilépünk az alkal-

mazásból. Ez minden bizonnyal kevesebb erőfeszítést igényel, de ne várjunk érte önfeledt tapsvihart a felhasználóktól. Az ügyfelek megpróbálhatnak belenyúlni a kivételbe, hogy eldöntsék, kijavítható-e a hiba:

```
try {
  Foo( );
  Bar( );
} catch( Exception e )
{
  switch( e.TargetSite.Name )
  {
    case "Foo":
      FixProblem( e );
      break;
    case "Bar":
      ReportErrorAndContinue( e );
      break;
    // valamilyen eljárás, amit a Foo vagy a Bar hív meg
    default:
      ReportErrorAndShutdown( e );
      break;
  }
} finally
{
  CleanupResources( );
}
```

Ez sokkal kevésbé tetszetős, mintha több `catch` blokkot használnánk; a kód elég sérülékeny. Ha módosítjuk az eljárás nevét, azzal már tönkre is tettük. Ha áthelyezzük a hibát előállító hívásokat egy közös segédfüggvénybe, azzal is tönkretesszük. Minél mélyebben következik be egy kivétel a hívásveremben, annál érzékenyebb lesz ez a szerkezet.

Mielőtt jobban elmélyednénk ebben a témában, hadd figyelmeztessek két dologra. Először is, a kivételek nem alkalmasak minden olyan hibalehetőség kezelésére, amivel összeakadhatunk. Habár nincsenek kőbe vésett szabályok, én magam olyan hibajelenségeknél szeretek kiváltani egy kivételt, amelyek komoly, messzire nyúló gondokat jelenthetnek, ha nem jelezzük és kezeljük őket azonnal. Egy adatbázisban fellépő adatépségi hibánál például érdemes létrehozni egy kivételt. Ha figyelmen kívül hagynánk, azzal csak súlyosabbá tennénk a helyzetet. Ha nem sikerül menteni egy felhasználó által használt ablak elhelyezkedésére vonatkozó beállítást, annak nem valószínű, hogy komoly következményei lennének. A hibát jelző kód visszaadása ebben az esetben elegendő.

Másodszor, a `throw` utasítás használata még nem jelenti azt, hogy rögtön létre kell hoznunk egy új kivételosztályt. A tanácsom, miszerint inkább több kivételosztályt írjunk, mint keveset, az emberi természeten alapul. A programozók hajlamosak minden alkalommal

a System.Exception osztályra támaszkodni, amikor kiváltanak egy kivételt. A hívó kód számára ez adja a legkevésbé hasznos információkat. Okosabb, ha inkább átgondoljuk és létrehozzuk a megfelelő kivételosztályt, aminek a segítségével a hívó kód könnyebben megértheti a hiba okát és így nagyobb eséllyel képes azt helyrehozni.

Még egyszer leírom: a különböző kivételosztályok létrehozását kizárólag az indokolhatja, hogy könnyebb legyen különböző műveleteket végrehajtani, amikor a felhasználók catch blokkokkal kezelik a kivételeket. Válasszuk ki azokat a hibákat, amelyeket jó eséllyel helyre lehet hozni valamilyen helyreállító művelet segítségével, és ezekhez készítsünk külön kivételosztályokat. Orvosolni tudja-e az alkalmazás, ha nem talál egy fájlt vagy egy könyvtárat? Tudja-e kezelni a nem megfelelő hozzáférési jogosultságokat? És mi a helyzet, ha egy hálózati erőforrást nem talál? Mindig hozzunk létre külön kivételosztályt, ha olyan hibalehetőségekkel találkozunk, amelyeket a megfelelő műveletek segítségével helyre lehet hozni.

Nekifogunk tehát a saját kivételosztályunk elkészítésének. Egy új kivételosztály létrehozásánál pontosan meghatározott feladataink vannak. A saját kivételosztályainkat mindig a System.ApplicationException osztályból és nem a System.Exception osztályból kell származtatnunk. Ehhez az alaposztályhoz ritkán kell újabb képességeket adnunk. A különböző kivételosztályok létrehozásával az a célunk, hogy a catch blokkokban meg lehessen különböztetni a hibák okait.

De azért el se vegyünk semmit azokból a kivételosztályokból, amelyeket létrehozunk. Az ApplicationException osztálynak négy konstruktora van:

```
// Alapértelmezett konstruktor
public ApplicationException( );

// Egy üzenettel együtt hozzuk létre
public ApplicationException( string );

// Egy üzenettel és egy belső kivétellel hozzuk létre
public ApplicationException( string, Exception );

// Egy bemeneti adatfolyamból hozzuk létre
protected ApplicationException(
  SerializationInfo, StreamingContext );
```

Amikor létrehozunk egy új kivételosztályt, készítsük el mind a négy konstruktort. A különböző helyzetekben a kivételek létrehozásának különböző módjaira van szükség. A munkát áthárítjuk az alaposztálybeli megvalósításra:

```
public class MyAssemblyException :
  ApplicationException
{
```

```
    public MyAssemblyException( ) :
      base( )
    {
    }

    public MyAssemblyException( string s ) :
      base( s )
    {
    }

    public MyAssemblyException( string s,
      Exception e) :
      base( s, e )
    {
    }

    protected MyAssemblyException(
      SerializationInfo info, StreamingContext cxt ) :
      base( info, cxt )
    {
    }
  }
```

Azok a konstruktorok, amelyek egy kivételparamétert várnak, egy kicsit részletesebb bemutatást érdemelnek. Előfordul néha, hogy az egyik könyvtár, amit használunk, kivételt vált ki. Ha csak egyszerűen továbbítjuk a segédfüggvényektől érkező kivételeket, akkor a kód, amelyik meghívta a könyvtárunkat, minimális információt kap a hiba helyreállításához:

```
public double DoSomeWork( )
{
  // Ez itt kiválthat egy olyan kivételt, amit
  // egy külső könyvtárban határoztak meg:
  return ThirdPartyLibrary.ImportantRoutine( );
}
```

Amikor előállítunk egy kivételt, meg kell adnunk a saját könyvtárunk adatait is. Váltsunk ki egy saját kivételt, és tegyük bele az eredeti kivételt az `InnerException` tulajdonságába. Így annyi többletinformációt adhatunk, amennyit csak tudunk:

```
public double DoSomeWork( )
{
  try {
    // Ez itt kiválthat egy olyan kivételt, amit
    // egy külső könyvtárban határoztak meg:
    return ThirdPartyLibrary.ImportantRoutine( );
```

```
    } catch( Exception e )
      {
        string msg =
          string.Format("Problem with {0} using library",
            this.ToString( ));
        throw new DoingSomeWorkException( msg, e );
      }
    }
}
```

Ez az újabb változat több információval szolgál arról a pontról, ahol a hiba keletkezett. Ha létrehoztuk a megfelelő `ToString()` tagfüggvényt (lásd az 5. tippet), akkor olyan kivételt sikerült készítenünk, ami leírja annak az objektumnak a teljes állapotát, ami a gondot okozta. Sőt mi több, a belső kivétel a hiba okának gyökereiről is árulkodik, ami valahol a felhasznált külső könyvtárban van.

Ezt a módszert kivételfordításnak (exception translation) nevezik. Egy alacsonyszintű kivételt egy magasabb szintű kivételre fordítunk át, ami több információt ad a hibáról. Minél több információt vagyunk képesek előállítani a hibával kapcsolatban, annál könnyebben tudják a felhasználók felismerni, illetve lehetőség szerint helyrehozni a hibát. Saját kivételtípusok létrehozásával az általános, alacsonyszintű problémákból konkrét kivételeket állíthatunk elő, amelyek tartalmazzák az alkalmazásra vonatkozó összes olyan információt, amire szükség lehet a hiba tökéletes felismeréséhez és esetleges kijavításához.

Az alkalmazásaink kivételeket váltanak ki. Remélhetőleg nem túl gyakran, de biztosan lesz olyan, amikor ez bekövetkezik. Ha semmilyen konkrét lépést sem teszünk, akkor az alkalmazás a .NET keretrendszer alapértelmezett kivételeit fogja kiváltani, ha hiba csúszik a keretrendszer magján meghívott tagfüggvényekbe. Ha több információt adunk a hibákról, azzal nagy lépést tehetünk afelé, hogy mi magunk és a felhasználóink is könnyebben felismerjék, és így esetleg kijavíthassák azokat. Különböző kivételosztályokat csakis akkor készítsünk, ha a hibák kijavításához különböző műveletekre van szükség. Teljes kivételosztályokat úgy hozhatunk létre, ha minden olyan konstruktort megadunk, amit az alaposztály is támogat. Az alacsonyabb szintről származó hibákkal kapcsolatos információkat az `InnerException` tulajdonság segítségével adhatjuk tovább.

6

Egyebek

Van néhány olyan tipp, amit egyik kategóriába sem lehet igazán besorolni, ettől viszont még éppen olyan fontosak, mint a többi. A kódhozzáférési jogosultságok megértése mindenki számára lényeges, éppen úgy, mint a kivételkezelési stratégiák megismerése. Vannak olyan tanácsok is, amelyek állandóan változnak, hiszen a C# élő nyelv, amit egy tevékeny csapat formál szabvánnyá. Megéri hát felkészülten várni a változásokat, mert azok hatással lesznek a jövőbeni munkánkra.

45. tipp

Részesítsük előnyben az erős kivételgaranciát

Egy kivétel kiváltásával olyan eseményt idézünk elő, ami megzavarja az alkalmazás normális működését: a folyamatvezérlés óraművébe porszem kerül. Olyan műveletek végrehajtása marad el, amelyekre számít a program. De ami ennél is rosszabb, a takarítást arra a programozóra hagyjuk, aki a kivételt elfogó kódot írja. A kivételek elfogásakor rendelkezésre álló cselekvési lehetőségek közvetlen kapcsolatban állnak azzal, hogy mennyire kezeljük jól a program állapotát a kivétel kiváltásakor. Szerencsére a C# fejlesztőknek már nem kell kidolgozniuk saját stratégiákat a kivételek kezelésére, mert a C++ programozók már elvégezték helyettünk a munka oroszlánrészét. Tom Cargill *Exception Handling: A False Sense of Security* (Kivételkezelés: Csalóka biztonságérzet) című cikkétől kezdve, Herb Sutter, Scott Meyers, Matt Austern, Greg Colvin és Dave Abrahams írásain át a C++ fejlesztők közössége kifejlesztett egy sor követendő gyakorlatot, amit fel tudunk használni a C# alkalmazások írásánál. A kivételkezelés körüli viták 6 éven át, 1994 és 2000 között folytak. A szerzők beszélgettek, vitatkoztak, és minden oldalról megvizsgálták a kérdést. Nekünk csak az a dolgunk, hogy ennek a kemény munkának a gyümölcsét learassuk a C# alkalmazásokban.

Dave Abrahams három kivételgaranciát határoz meg: az alapgaranciát, az erős garanciát és a „nincs kiváltás" („no-throw") garanciát. Herb Sutter ezeket a garanciákat *Exceptional C++* (Kivételes C++, Addison-Wesley, 2000) című könyvében mutatja be részletesen. Az alapgarancia szerint semmilyen erőforrás nem szivároghat, és minden objektumnak érvényes állapotban kell lennie, miután az alkalmazás kivált egy kivételt. Az erős garancia az alapgaranciára épül, továbbá kiköti, hogy a kivétel bekövetkezte után a program állapotának változatlannak kell maradnia. A „nincs kiváltás" garancia szerint a műveletek soha nem lehetnek sikertelenek, amiből az következik, hogy a tagfüggvények soha nem váltanak ki kivételeket. Az erős kivételgarancia jelenti a legjobb kompromisszumot a kivételkezelés egyszerűsítése és az alkalmazásnak a kivételekből való felépülése között.

Az alapgarancia szinte alapértelmezett a .NET környezetben és a C# nyelvben. A kivételek bekövetkezésekor csak úgy tudunk erőforrásokat szivárogtatni, ha egy `IDisposable` felületet megvalósító erőforrás birtokában váltunk ki kivételt. A 18. tipp bemutatja, hogyan kerülhető el az erőforrások szivárgása kivételek esetén.

Erős kivételgarancia esetén, ha egy művelet megszakad egy kivétel miatt, akkor a program állapota változatlan marad. A művelet vagy sikeresen befejeződik, vagy nem módosítja a program állapotát. Más választása nincs. Az erős garancia előnye, hogy könnyebben lehet folytatni a program végrehajtását egy kivétel elfogása után, ha betartjuk az erős garancia előírásait. Amikor elfogunk egy kivételt, akkor bármilyen műveletet kíséreltünk is meg, az biztosan nem következett be: nem kezdődött el, és semmilyen módosítást nem hajtott végre. A program állapota olyan, mintha el sem indítottuk volna a műveletet.

A korábban adott tanácsaim közül sok segíteni fog bennünket az erős kivételgarancia megvalósításában. A program által használt adatelemeket mindig nem változó értéktípusokban tároljuk (lásd a 6. és a 7. tippet). Ha az említett két tippet kombináljuk, akkor a program állapotán végrehajtott módosítások könnyedén bekövetkezhetnek azután, hogy végrehajtottunk egy olyan műveletet, ami kiválthat egy kivételt. Általánosságban elmondható, hogy az adatok bármilyen módosítását az alábbi módon célszerű végrehajtani:

1. Készítsünk biztonsági másolatot a módosítandó adatokról.
2. A módosításokat ezen a biztonsági másolaton hajtsuk végre, beleértve azokat a műveleteket is, amelyek kivételeket válthatnak ki.
3. Cseréljük vissza az átmeneti másolatot az eredetire. Ez a művelet nem válthat ki semmilyen kivételt.

A következő kód például egy másolat segítségével frissíti egy alkalmazott besorolását és fizetését:

```
public void PhysicalMove( string title, decimal newPay )
{
  // A bérlista egy elemi szerkezet:
  // a ctor kivételt vált ki, ha érvénytelenek a mezők
```

```
    PayrollData d = new PayrollData( title, newPay,
      this.payrollData.DateOfHire );

    // ha a d létrehozása sikeres volt, akkor jöhet a csere:
    this.payrollData = d;
  }
```

Van úgy, hogy az erős garancia nem elég hatékony ahhoz, hogy támogassuk, és az is előfordul, hogy csak úgy lehet támogatni, hogy közben finom kis hibák csúsznak a programba. Az első és legegyszerűbb eset a ciklikus szerkezeteké. Amikor egy ciklus belsejében lévő kód módosítja a program állapotát, és eközben kivételt válthat ki, nehéz döntés elé kerülünk. Vagy készítünk egy biztonsági másolatot a cikluson belül használt összes objektumról, vagy alacsonyabbra tesszük a mércét, és csak az alapgaranciát támogatjuk. Bár nincsenek kőbe vésett szabályok ez ügyben, a halmon tárolt objektumok másolása kevésbé költséges egy kezelt környezetben, mint az eredeti (natív) környezetekben. A tervezők sok időt szenteltek a .NET környezet memóriakezelésének optimalizálására. Ha tehetem, én mindig támogatom az erős garanciát, még akkor is, ha ehhez egy nagyobb tárolót kell is átmásolnom. A hibákból való felépülés képessége többet nyom a latban, mint az a csekély teljesítménynövekedés, amit abból nyerünk, hogy nem kell elkészítenünk a másolatokat. Vannak olyan különleges esetek, amikor nincs értelme elkészíteni a másolatot. Ha egy kivétel azzal jár, hogy megszakad a program futása, akkor felesleges az erős garanciával foglalkozni. Sokkal nagyobb aggodalomra ad okot, hogy a hivatkozási típusok cseréje programhibákhoz vezethet. Vegyük az alábbi példát:

```
private DataSet _data;
public IListSource MyCollection
{
  get
  {
    return _data;
  }
}

public void UpdateData( )
{
  // elkészítjük a biztonsági másolatot:
  DataSet tmp = _data.Clone( ) as DataSet;

  using ( SqlConnection myConnection =
    new SqlConnection( connString ))
  {
    myConnection.Open();

    SqlDataAdapter ad = new SqlDataAdapter( commandString,
      myConnection );
```

```
      // Adatokat tárolunk a másolatban
      ad.Fill( tmp );

      // sikerült, elvégezzük a cserét:
      _data = tmp;
    }
  }
```

A biztonsági másolat készítésének eme felhasználási módja nagyszerűnek tűnik. Létrehoztuk a `DataSet` objektum egy másolatát. Ezután kiolvassuk az új adatokat az adatbázisból, majd betöltjük azokat az átmeneti `DataSet` objektumba, végül visszatesszük az átmeneti tároló tartalmát az eredetibe. Csodálatosnak tűnik. Ha bármilyen hiba csúszik az adatok kiolvasásába, akkor nem módosítottunk semmit.

Csak egy apró gond van, mégpedig az, hogy ez így nem működik. A `MyCollection` tulajdonság egy hivatkozást ad vissza a `_data` objektumra (lásd a 23. tippet). Ennek az osztálynak az összes ügyfele továbbra is hivatkozásokat fog tartalmazni az eredeti `DataSet` objektumra, miután meghívjuk az `UpdateData` függvényt. Ezek a hivatkozások a régi adatokra mutatnak. A cserélgetős trükk nem működik a hivatkozási típusoknál, csak az értéktípusokkal lehet megcsinálni. Gyakori műveletről lévén szó, a `DataSet` objektumok esetében van egy megoldás. Használjuk a `Merge` tagfüggvényt:

```
private DataSet _data;
public IListSource MyCollection
{
  get
  {
    return _data;
  }
}

public void UpdateData( )
{
  // elkészítjük a biztonsági másolatot:
  DataSet tmp = new DataSet( );

  using ( SqlConnection myConnection =
    new SqlConnection( connString ))
  {
    myConnection.Open();

    SqlDataAdapter ad = new SqlDataAdapter( commandString,
      myConnection);

    ad.Fill( tmp );
```

```
        // sikerült, jöhet az egybeolvasztás:
        _data.Merge( tmp );
    }
}
```

A módosítások beolvasztása az aktuális DataSet objektumba lehetővé teszi az ügyfelek számára, hogy érvényesek maradjanak a hivatkozásaik, miközben a DataSet belső tartalmát frissítjük.

Az általános esetben azonban nem lehet megoldani azt, hogy úgy cserélgessük a hivatkozási típusokat, hogy közben az összes ügyfélnél meglegyen az objektum aktuális másolata. A csere csak az értéktípusoknál működik. Ha megfogadjuk a 6. tipp tanácsát, akkor ennek éppen elegendőnek kell lennie.

Utolsóként vegyük a legszigorúbb, „nincs kiváltás" garanciát. A „nincs kiváltás" garancia nagyjából azt jelenti, amire a neve utal: egy tagfüggvény akkor tesz eleget ennek a garanciának, ha mindig végig lefut, és nem engedi meg, hogy kivétel váltódjon ki. A nagyobb programoknál ez egyszerűen megvalósíthatatlan az összes eljárásnál. Néhány helyen azonban a tagfüggvényeknek támogatniuk kell a „nincs kiváltás" garanciát. A véglegesítők és a Dispose tagfüggvények sem válthatnak ki kivételeket. A kivételek kiváltása mindkét esetben komolyabb gondot okozna, mint bármely más választási lehetőség. A véglegesítők esetében egy kivétel kiváltása megszakítja a program futását, és leállítja a további tisztogató műveleteket.

Ha a Dispose tagfüggvény vált ki egy kivételt, akkor elképzelhető, hogy a rendszeren két kivétel fog végigfutni. A .NET környezet elveszti az első kivételt, és létrehoz egy újat. Az eredeti kivételt a programunkon belül sehol nem tudjuk elfogni, azt már „megette" a rendszer. Ez nagyon bonyolulttá teszi a hibakezelést. Hogyan lehet kimászni egy olyan hibából, amit nem is látunk?

A „nincs kiváltás" garancia utolsó lehetséges helyét a képviselőcélpontok jelentik. Amikor egy képviselőcélpont kivételt vált ki, a többi képviselőcélpont egyike sem hívódik meg ugyanabból a csoportos képviselőből (multicast delegate). Az egyetlen mód ennek elkerülésére, ha gondoskodunk róla, hogy a képviselőcélpontok soha ne váltsanak ki kivételeket. Vegyük át ezt még egyszer. A képviselők célpontjainak (beleértve az eseménykezelőket is) nem szabad kivételeket kiváltaniuk. Ha mégis így teszünk, az azt jelenti, hogy az eseményt kiváltó kód nem lehet része erős garanciát nyújtó kódnak. Itt azonban módosítani fogom ezt a tanácsot. A 21. tippben láttuk, hogy meg lehet hívni a képviselőket úgy is, hogy felépüljön a program a kivételekből. Csakhogy nem mindenki csinálja így, ezért a képviselőkezelő függvényekben kerüljük a kivételek kiváltását. Az, hogy mi nem váltunk ki kivételeket a képviselőkben, még nem jelenti azt, hogy mások is tartják magukat ehhez.

Ne bízzunk a „nincs kiváltás" garanciában a saját képviselőhívásainknál. Ez a védekező programozás lényege: mindig a lehető legjobb megközelítést alkalmazzuk, mert lehet, hogy mások a lehető legrosszabb utat választják.

A kivételek komoly változásokat eredményeznek egy alkalmazás vezérlésében. A legrosszabb esetben bármi történhetett vagy éppen semmi sem történt. Az egyetlen módja annak, hogy biztosan tudhassuk, mi változott, ha betartjuk az erős kivételgarancia szabályait. Ilyenkor egy művelet vagy sikeresen befejeződik, vagy semmilyen módosítást nem hajt végre. A véglegesítők, a `Dispose()`, és a képviselőcélpontok különleges eseteknek számítanak. Ezeknek úgy kell lefutniuk, hogy közben egyetlen kivétel sem hagyhatja el őket. Zárszóként azt tanácsolom, hogy legyünk nagyon óvatosak a hivatkozási típusok cseréjénél, mert ez rengeteg apró hibát okozhat.

46. tipp

Bánjunk csínján az együttműködési képességekkel

Az egyik legbölcsebb dolog, amire a Microsoft rájött a .NET tervezésekor, hogy senki nem fogja használni ezt az új környezetet, ha nem tudja majd felhasználni a régi kódgyűjteményét. A Microsoft tudta, hogy ha nem hagy lehetőséget a már meglévő kód felhasználására, akkor az új környezet elfogadása nagyon lassan fog menni. A rendszerek közti együttműködés ettől azonban még nem lett se egyszerű, se hatékony. Az együttműködésre van lehetőség, de ezzel nagyjából minden jót el is mondtunk róla. Minden együttműködési stratégia megkövetel valamilyen irányítást (marshalling), amikor a vezérlés átlépi az eredeti (natív) kód és a kezelt környezet határait. Az együttműködési stratégiák arra is rákényszerítik a fejlesztőket, hogy a tagfüggvények paramétereit saját kezűleg vezessék be. Végezetül, a CLR nem képes optimalizálni az együttműködési határokon átívelő kódot. Mi sem lenne egyszerűbb a fejlesztő szemszögéből, mint az, hogy figyelmen kívül hagyja az eredeti kódba és a COM objektumokba fektetett energiát. A valóságban azonban sokszor más a helyzet. A legtöbbünknek sokszor az a feladata, hogy létező alkalmazásokhoz adjon új elemeket, javítgassa és frissítse a meglévő eszközöket, vagy valamilyen más módon rávegye az új, kezelt alkalmazásokat, hogy azok együttműködjenek a ránk maradt régi alkalmazásokkal. Sokszor a rendszerek közti együttműködés valamilyen fokának biztosítása az egyetlen módja annak, hogy fokozatosan lecseréljük a régi rendszereket. Ezért fontos, hogy megértsük a különböző együttműködési stratégiákhoz kapcsolódó költségeket. Ezek a költségek a fejlesztéssel járó munkamennyiségnél és a futásidejű teljesítménynél egyaránt jelentkeznek. Néha az a legegyszerűbb, ha teljesen újraírjuk az elavult alkalmazást, máskor pedig ki kell választanunk a legmegfelelőbb együttműködési stratégiát.

Mielőtt rátérnék az elérhető együttműködési stratégiák bemutatására, előbb egy bekezdést még a „kukába a régivel" stratégiának kell szentelnem. Az 5. fejezetben (*A keretrendszer használata*) bemutattam néhány olyan osztályt és trükköt, amit a .NET keretrendszerrel

együtt megkapunk. A programunk fő szolgáltatásait leíró osztályokat és algoritmusokat szinte mindig megtalálhatjuk, és csak át kell tennünk ezeket a C# nyelvbe, a többi, már meglévő kódot pedig lecserélhetjük a .NET nyújtotta lehetőségekre. Ez persze nem mindig és mindenhol működik, de komolyan érdemes megfontolni, mint egy eshetőséget a váltáshoz. Az 5. fejezetre teljes egészében tekinthetünk úgy, mint egy ajánlásra, ami a „kukába a régivel" stratégia követésére buzdít. Ez a tipp viszont az együttműködésről szól, ami néha kínkeserves dolog.

A továbbiakban feltesszük, hogy már eldöntöttük, hogy a teljes újraírás nem megoldható. A .NET kódból számos módon elérhetjük az eredeti (natív) kódot. Tisztában kell lennünk viszont a kezelt és a kezeletlen kód közti határ átlépésével járó költségekkel és hatékonyságcsökkenéssel. Az első költségtétel akkor jelentkezik, amikor oda-vissza irányítjuk, terelgetjük az adatokat a kezelt és az eredeti halom között. A második költségtétel a kezelt és kezeletlen kód közti közlekedés közben fellépő címfordítási (thunking) költség. Ezek a tételek minket és a felhasználóinkat egyaránt terhelik. A harmadik tétel csak minket érint. Ez nem más, mint az a munkamennyiség, amire a vegyes környezet kezeléséhez van szükség. Ez a harmadik tétel a legnagyobb, így a programtervezéssel kapcsolatos döntéseinknél mindig figyeljük oda, hogy ezt a lehető legkisebb szinten tartsuk.

Először nézzük meg az együttműködés során fellépő, a teljesítménynél jelentkező költségeket, és azt, hogy miként csökkenthetjük ezeket. Itt az irányítás a legeslegfontosabb tényező. Ahogy a webszolgáltatások és a távoli elérések esetében is, itt is „tagbaszakadt", kevésbé „szószátyár" API-k létrehozására kell törekednünk. A kezeletlen kóddal való együttműködésnél azonban más módszerhez kell folyamodnunk. A tagbaszakadt együttműködési API-kat úgy készíthetjük el, hogy a meglévő kezeletlen API-hoz hozzáadunk egy új, az együttműködést jobban támogató API-t. Gyakori COM-os megoldás, hogy sok olyan tulajdonságot adunk meg, amit az ügyfelek beállíthatnak, megváltoztatva ezzel az objektum belső állapotát és viselkedését. A tulajdonságok beállításánál minden egyes alkalommal át kell irányítani az adatokat a határon. (A határ átlépésekor egyben címfordítás is történik.) Ez nem túl hatékony. Sajnos előfordulhat, hogy a COM objektum vagy kezeletlen könyvtár nem a mi irányításunk alatt áll. Ha ez a helyzet, kicsit több munkára van szükség. Ebben az esetben létrehozhatunk egy sovány kezeletlen C++ könyvtárat, ami egy olyan tagbaszakadt API segítségével teszi elérhetővé a típus képességeit a külvilág felé, amilyenre szükségünk van. Ezzel persze nő a fejlesztésre szánt idő (az a fránya harmadik tétel).

Amikor beburkolunk egy COM objektumot, mindig módosítsuk az adattípusokat, hogy a kezelt és kezeletlen kódrészek közti irányítás zökkenőmentesebb legyen. Néhány típus irányítása sokkal hatékonyabban kivitelezhető, mint másoké. Törekedjünk arra, hogy a kezelt és kezeletlen kódrétegek között bitblokkosítható típusok segítségével vigyük át az adatokat. A **bitblokkosítható típusok** (blittable types) azok, ahol a típus kezelt és kezeletlen ábrázolása megegyezik. A típus tartalmát ilyenkor az objektum belső szerkezetétől függetlenül másolhatjuk. Néhány esetben a kezeletlen kód használhatja a kezelt memóriát.

A bitblokkosítható típusokat az alábbi felsorolás tartalmazza:

```
System.Byte
System.SByte
System.Int16
System.UInt16
System.Int32
System.UInt32
System.Int64
System.UInt64
System.UIntPtr
```

Ezen kívül minden bitblokkosítható elemekből álló egydimenziós tömb maga is bitblokkosítható lesz. Végezetül, azok a formázott típusok is bitblokkosíthatók, amelyek kizárólag bitblokkosítható típusokat tartalmaznak. A **formázott típus** (formatted type) olyan szerkezet, aminek a megjelenését a `StructLayoutAttribute` jellemző adja meg:

```
[ StructLayout( LayoutKind.Sequential ) ]
public struct Point3D
{
  public int X;
  public int Y;
  public int Z;
}
```

Ha kizárólag bittblokkosítható típusokat használunk a kódunk kezelt és kezeletlen rétegei közötti adatátvitelhez, azzal minimálisra csökkenthetjük a másolandó információmennyiséget. A másolási műveletek így szintén optimálisak lesznek.

Amennyiben nem tudjuk bitblokkosítható típusokra korlátozni a típusainkat, akkor az `InAttribute` és az `OutAttribute` jellemzők segítségével szabályozhatjuk, hogy mikor készüljenek másolatok. A COM-hoz hasonlóan ezek a jellemzők szabályozzák, hogy melyik irányban másoljuk az adatokat. Míg az `In/Out` paramétereket mindkét irányban lemásoljuk, a csak `In` vagy `Out` jellemzőkkel ellátott paramétereket csak az egyik irányban. Mindig a legszűkebb `In/Out` kombinációt használjuk, hogy elkerüljük a felesleges másolást.

Végezetül, a teljesítményt azzal is javíthatjuk, ha megadjuk, hogyan történjen az adatok irányítása. Erre a leggyakrabban a karakterláncoknál kerül sor. A karakterláncok irányítása alapértelmezés szerint BSTR-eket használ. Ez biztonságos megközelítés, de egyben ez a legkevésbé hatékony is. A felesleges másolást megspórolhatjuk, ha módosítjuk az alapértelmezett irányítási módszert a `MarshalAs` jellemző alkalmazásával. Az alábbi bevezetés `LPWStr` vagy `wchar*` típusként irányítja a karakterláncot:

```
public void SetMsg(
  [ MarshalAs( UnmanagedType.LPWStr ) ] string msg );
```

Ez lenne hát az adatok kezelt és kezeletlen rétegek közti mozgatásának rövid története. A program lemásolja az adatokat, és esetleg a típusukat is átfordítja a kezelt és a kezeletlen típusok között. A másolási műveletek számát háromféleképpen csökkenthetjük. Először is azzal, ha a paraméterek, illetve a visszatérési értékek típusát bitblokkosítható típusokra korlátozzuk. Ez a legjobb módszer. Ha ez nem lehetséges, akkor az `In` és az `Out` jellemzők alkalmazásával csökkenthetjük a szükséges másolási és szállítási műveletek számát. Végül pedig, néhány típus irányítása többféleképpen is megoldható, tehát ilyenkor kiválaszthatjuk a számunkra legmegfelelőbb megoldást.

Most pedig menjünk tovább arra a kérdésre, hogy miként adható át a programvezérlés a kezelt és kezeletlen összetevők között. Három választási lehetőségünk is van: a COM együttműködés, a Platform Invoke (P/Invoke), és a kezelt C++ használata. Mindegyiknek megvan a maga előnye és hátránya.

A COM együttműködés a legegyszerűbb módja annak, hogy a már használatban lévő COM összetevőinket felhasználhassuk. A COM együttműködés azonban a legkevésbé hatékony módja az eredeti (natív) kód elérésének a .NET környezetben. Csak akkor kövessük ezt az utat, ha korábban már jelentős munkát fektettünk COM összetevők készítésébe. Egyébként rá se nézzünk, és felejtsük el, úgy ahogy van. Ha nincsenek COM összetevőink, akkor a COM együttműködés használatához meg kell ismerkednünk a COM és az együttműködési szabályokkal is, az `IUnknown` megértésére pedig nem ez a megfelelő időpont. Azon keveseknek, akiknek ez sikerült, a mai napig gondot okoz, hogy valahogy megpróbálják elfelejteni ezt a szörnyűséget. A COM együttműködés használatával egyben kitesszük magunkat a COM alrendszer használatával járó futásidejű költségeknek. Azt is vegyük figyelembe, hogy milyen különbségek vannak a CLR, illetve a COM objektumok életciklusainak kezelése között. Ha a CLR-re hagyjuk ezt, akkor minden átvett COM objektumnak lesz egy véglegesítője, ami meghívja a `Release()` függvényt az adott COM felületen. Azt is megtehetjük, hogy konkrétan felszabadítjuk a COM objektumot a `ReleaseCOMObject` meghívásával. Az első megközelítés futásidőben csökkenti a programunk hatékonyságát (lásd a 15. tippet), a másodikkal pedig a programozóinknak okozunk komoly fejfájást. A `ReleaseCOMObject` használatával olyan memóriakezelési problémákat vetünk fel, amit a CLR COM együttműködési rétegében már régen megoldottak. Átvesszük az irányítást, és azt gondoljuk, hogy mi vagyunk a legokosabbak. A CLR persze másképp gondolja, és felszabadítja a COM objektumokat, hacsak meg nem mondjuk neki, hogy mi ezt már megtettük helyette. Ez finoman fogalmazva elég cseles lehet, mert a COM azt várja, hogy a programozó minden felületen meghívja a `Release()` függvényt, és a kezelt kód intézi el az objektumokat. Rövidre fogva a szót, tudnunk kell, hogy mely felületek kerültek az `AddRef` segítségével egy objektumra, és csak ezeket szabad felszabadítanunk. Engedjük, hogy a CLR kezelje a COM életciklusait, és viseljük el a futásidejű teljesítménycsökkenés költségeit. Elfoglalt fejlesztők vagyunk. Az, hogy feleslegesen kitanuljuk a COM erőforráskezelést a .NET környezetben, tényleg több, mint amit be kell vállalnunk (és itt megint megjelenik a harmadik költségtétel).

A másik választási lehetőségünk a P/Invoke használata. Ez a leghatékonyabb módja a Win32 API-k hívásának, mert így elkerülhetjük a COM-költségeket. A rossz hír az, hogy a P/Invoke segítségével meghívott tagfüggvényekhez magunknak kell megírnunk a felületet. Minél több tagfüggvényt hívunk meg, annál több tagfüggvény-bevezetést kell megírnunk. Ez a P/Invoke bevezetés mutatja meg a CLR számára, hogy miként érheti el az eredeti (natív) tagfüggvényt. Ez a munkamennyiség a magyarázata annak, hogy miért használja minden P/Invoke példa (beleértve a következőt is) a `MessageBox`-ot:

```
public class PInvokeMsgBox
{
   [ DllImport( "user32.dll" ) ]
   public static extern int MessageBoxA(
      int h, string m, string c, int type );

   public static int Main()
   {
      return MessageBoxA( 0,
        "P/InvokeTest",
        "It is using Interop", 0 );
   }
}
```

A P/Invoke másik nagy hátránya, hogy nem objektumközpontú nyelvekhez tervezték. Ha át kell vennünk egy C++ könyvtárat, akkor meg kell adnunk a kitüntetett neveket (decorated names) az átvétel bevezetésénél. Tegyük fel, hogy a Win32 MessageBox API helyett a két `AfxMessageBox` egyikét akarjuk elérni az MFC C++ DLL könyvtárban. Ehhez az alábbi két tagfüggvény közül az egyikhez létre kell hoznunk egy P/Invoke bevezetést:

```
?AfxMessageBox@@YGHIII@Z
?AfxMessageBox@@YGHPBDII@Z
```

Ez a két kitüntetett név ennek a két tagfüggvénynek felel meg:

```
int AfxMessageBox( LPCTSTR lpszText,
  UINT nType, UINT nIDHelp );
int AFXAPI AfxMessageBox( UINT nIDPrompt,
  UINT nType, UINT nIDHelp);
```

Pár túlterhelt tagfüggvény után gyorsan rájövünk, hogy ez nem a leghatékonyabb módszer az együttműködés megvalósítására. Röviden csak annyit, hogy a P/Invoke csak a C stílusú Win32 tagfüggvények eléréséhez való (további költségek a fejlesztési idő szempontjából).

Az utolsó választási lehetőségünk az, hogy a Microsoft C++ fordító `/CLR` kapcsolóját használva keverjük a kezelt és a kezeletlen kódot. Ha a teljes kódunkat a `/CLR` segítségével fordítjuk le, akkor egy olyan MSIL alapú könyvtárat készítünk, ami a natív halmot használja az adatok tárolására. Ez azt jelenti, hogy ezt a könyvtárat közvetlenül nem lehet meghívni a C# nyelvből. Készítenünk kell egy kezelt C++ könyvtárat a régi kódunkhoz, hogy áthidaljuk a kezelt és kezeletlen típusok közti rést, támogatva a kezelt és kezeletlen halmok közti irányítási folyamatot. Ebben a kezelt C++ könyvtárban kezelt osztályok lesznek, amelyeknek az adattagjai a kezelt halomra kerülnek. Az osztályokban a natív objektumokra mutató hivatkozások is előfordulnak:

```
// A kezelt osztály bevezetése:
public __gc class ManagedWrapper : public IDisposable
{
private:
  NativeType* _pMyClass;

public:
  ManagedWrapper( ) :
    _pMyClass( new NativeType( ) )
  {
  }

  // Eldobjuk:
  virtual void Dispose( )
  {
    delete _pMyClass;
    _pMyClass = NULL;
    GC::SuppressFinalize( this );
  }

  ~ManagedWrapper( )
  {
    delete _pMyClass;
  }

  // példa egy tulajdonságra:
  __property System::String* get_Name( )
  {
    return _pMyClass->Name( );
  }
  __property void set_Name( System::String* value )
  {
    char* tmp  = new char [ value->Length + 1 ];
    for (int i = 0 ; i < value->Length; i++ )
      tmp[ i ] = ( char )value->Chars[ i ];
    tmp[ i ] = 0;
    _pMyClass->Name( tmp );
```

```
      delete [] tmp;
    }

    // példa egy tagfüggvényre:
    void DoStuff( )
    {
      _pMyClass->DoStuff( );
    }

    // egyéb tagfüggvények elhagyva...
  }
```

Ez ismét csak nem a leghatékonyabb programozási eszköz, amit valaha használtunk. Ez egy ismétlődő kód, aminek csupán az a célja, hogy elbánjon a kezelt és a kezeletlen adatok irányításával és címfordításával. Az előnye abban rejlik, hogy teljes egészében mi szabályozzuk, miként tesszük elérhetővé a natív kód tagfüggvényeit és tulajdonságait a külvilág felé. A hátránya az, hogy ezt a sok kódot úgy kell megírnunk, hogy az agyunk egyik féltekéje .NET kódot ír, a másik meg C++ kódot. Amikor átváltunk az egyikről a másikra, könnyen becsúszhat egy-egy hiba. Nem szabad megfeledkeznünk a kezeletlen objektumok törléséről, a kezelt objektumok törlése azonban nem a mi feladatunk. Jelentősen lassítja a fejlesztést, hogy állandóan figyelnünk kell, hogy éppen melyik esettel van dolgunk.

A `/CLR` kapcsoló használata úgy hangzik, mint egyfajta csodaszer, de ezt a csodafegyvert nem használhatjuk minden együttműködési helyzetben. A sablonok és a kivételek kezelése eléggé eltér egymástól a C++ és a C# nyelvekben. A jól megírt, hatékony C++ kódból nem feltétlenül lesz a legjobb MSIL szerkezet. Ami ennél is fontosabb, a `/CLR` kapcsoló segítségével fordított C++ kód nem ellenőrizhető. Ahogy korábban már említettem, ez a fajta kód a natív halmot, vagyis a natív memóriát használja; a CLR nem képes meggyőződni a kód biztonságosságáról. Azoknak a programoknak, amelyek meghívják ezt a kódot, rendelkezniük kell a nem biztonságos kód eléréséhez szükséges jogosultságokkal. De ennek ellenére is, a `/CLR` kapcsolós megközelítés a legjobb módja a létező C++ kód (de nem a COM objektumok) felhasználásának a .NET környezetben. A programunk nem idéz elő címfordítási költségeket, mert a C++ könyvtáraink immár MSIL kódot és nem CPU utasításokat fognak tartalmazni.

A rendszerek közti együttműködés megvalósítása kínkeserves feladat. Komolyan fontoljuk meg az eredeti alkalmazás újraírásának lehetőségét, mielőtt az együttműködést számba vesszük. Sokszor ez az egyszerűbb és gyorsabb megoldás. Sajnos sok fejlesztő nem tudja megkerülni az együttműködést. Ha már vannak bármilyen nyelven megírt COM objektumaink, akkor használjuk a COM együttműködési képességeket. Ha már van kész C++ kódunk, akkor a `/CLR` kapcsoló és a kezelt C++ kínálják a legjobb módszert a régi kódok eléréséhez az újonnan készülő C# alkalmazásból. Mindig azt a módszert válasszuk, amelyik a legkevesebb időt veszi igénybe. Akkor is, ha ez éppen a „kukába a régivel" stratégia lesz.

47. tipp

Írjunk biztonságos kódot

A .NET futásidejű környezetet olyanra tervezték, hogy a rosszindulatú programok ne tudjanak beférkőzni egy távoli gépre, majd ott lefutni. Van azonban néhány olyan elosztott rendszer, amelynek működése távoli gépekről letöltött és futtatott kódokon alapszik. Ha előfordulhat, hogy a programjainkat az Interneten vagy egy intraneten keresztül juttatjuk el a felhasználókhoz, vagy éppen közvetlenül a Világhálón futtatjuk, akkor meg kell ismerkednünk azokkal a megszorításokkal, amiket a CLR kényszerít a szerelvényeinkre. Ha a CLR nem bízik meg teljes egészében egy szerelvényben, akkor korlátozza annak lépéseit. Ezt nevezik kódhozzáférési biztonságnak (code access security, CAS). Egy másik oldalról a CLR betartatja a szerep alapú biztonságot (role-based security), ami azt jelenti, hogy egy kód futásának engedélyezése az őt futtató felhasználó jogosultságaitól függ.

A biztonsági előírások megsértése futásidőben következik be, így a fordító nem képes betartatni azokat. Ráadásul ez sokkal kisebb eséllyel fordul elő azon a gépen, amelyiken a programot írjuk; a lefordítandó kód a merevlemezünkről töltődik be, ezért a rendszer jobban megbízik benne. A .NET Security (biztonsági) modell következményeinek bemutatása ugyan megtöltene akár több kötetet is, de azért még messze a józan ész határain belül tehetünk néhány lépést annak érdekében, hogy a szerelvényeink könnyebben együtt tudjanak működni a .NET biztonsági modellel. A most következő tanácsok csak arra az esetre vonatkoznak, amikor olyan könyvtárösszetevőket vagy összetevőket és programokat készítünk, amelyeket a Világhálón keresztül érhetnek el a felhasználók.

A következő sorok olvasása során végig gondoljunk arra, hogy a .NET kezelt környezet. A környezet már önmagában is garantál némi biztonságot. A .NET keretrendszeri könyvtár telepítésekor annak legnagyobb része teljes bizalmat élvez a .NET házirend teljes egészében. Ellenőrizhetően biztonságos, ami azt jelenti, hogy a CLR megvizsgálhatja az IL kódot, hogy az biztosan ne hajtson végre semmilyen veszélyes műveletet, mint amilyen mondjuk a nyers memóriaterület (raw memory) elérése. A könyvtár soha nem követel olyan jogosultságokat, amelyek a helyi erőforrások eléréséhez szükségesek. Mi is mindig kövessük ezt a példát. Ha a kódunknak nincs szüksége semmilyen különleges jogosultságra, akkor ha lehet, ne használjuk a CAS API-kat arra, hogy megállapítsuk a jogosultságainkat, hiszen ezzel csak a program teljesítményén rontunk.

A CAS API-kat arra használjuk, hogy hozzáférjünk néhány olyan védett erőforráshoz, amelyek emelt szintű jogosultságokat igényelnek. A leggyakoribb védett erőforrás a fájlrendszer és a kezeletlen memória. Védett erőforrások lehetnek még az adatbázisok, a hálózati kapuk (portok), a Windows rendszerleíró adatbázisa, illetve a nyomtatási alrendszer. Ha ezekhez az erőforrásokhoz a megfelelő jogosultságok nélkül próbálunk hozzáférni, akkor minden alkalommal kivételt kapunk. Ha megpróbálunk hozzáférni ezekhez az erőforrásokhoz, ak-

kor előfordulhat, hogy a futásidejű környezet elvégez egy biztonsági verembejárást, hogy meggyőződjön arról, hogy a hívásveremben található összes szerelvény rendelkezik a megfelelő jogosultságokkal. Nézzük meg a memóriát és a fájlrendszert, és beszéljük meg azokat a követendő gyakorlatokat, amelyekkel biztonságos programokat írhatunk.

A kezeletlen memóriához való hozzáférést úgy kerülhetjük el, hogy ha tehetjük, mindig ellenőrizhetően biztonságos szerelvényeket hozunk létre. A biztonságos szerelvények nem használnak mutatókat sem a kezelt, sem a kezeletlen halom eléréséhez. Ha tudtuk, ha nem, szinte az összes C# kód, amit megírunk, biztonságos lesz. Ha nem kapcsoljuk be az `/unsafe` C# fordítói beállítást, akkor bizony ellenőrizhetően biztonságos kódot írunk. Az `/unsafe` megengedi a mutatók használatát, amit a CLR nem tud ellenőrizni.

Nem biztonságos kód használatára kevés okunk lehet. Leggyakrabban a jobb teljesítmény érdekében vetemedünk erre. A nyers memóriára irányuló mutatók gyorsabbak a hivatkozások ellenőrzésénél; egy átlagos tömb esetében akár 10-szer gyorsabbak is lehetnek. De amikor nem biztonságos szerkezeteket használunk, akkor tisztában kell lennünk azzal, hogy ha ilyen kód van a szerelvény bármely pontján, akkor az az egész szerelvényre hatással van. Amikor nem biztonságos kódrészeket írunk, érdemes fontolóra vennünk azt a lehetőséget, hogy külön szerelvénybe tegyük ezeket a blokkokat (lásd a 32. tippet). Ezzel csökkenthetjük a nem biztonságos kód hatásait a teljes alkalmazásra. Ha elkülönítjük az ilyen kódot, akkor csak azokat a hívókat érintik ezek a hatások, akiknek szükségük van az adott szolgáltatásra, a többi szolgáltatást pedig továbbra is használhatjuk egy szigorúbban ellenőrzött környezetben is. Nem biztonságos kódra ahhoz is szükségünk lehet, hogy boldogulhassunk a P/Invoke és a COM felületekkel, amelyekhez nyers mutatókra van szükség. A tanácsom itt is érvényes: különítsük el ezt a kódot. A nem biztonságos kódnak csak a saját kis szerelvényének működését szabad befolyásolnia, semmi mást.

A memória elérésével kapcsolatos tanács egyszerű. Ha lehet, mindig kerüljük a kezeletlen memóriához való hozzáférést.

A következő leggyakoribb biztonsági kockázat a fájlrendszernél jelentkezik. A programok adatokat tárolnak, és ezt gyakran fájlokban teszik. Az Internetről letöltött kód általában nem férhet hozzá a helyi fájlrendszer legtöbb részéhez. Ez bizony hatalmas biztonsági rést jelentene. Ha viszont teljesen lemondanánk a fájlrendszer eléréséről, akkor nehezen tudnánk használható programokat írni. Ezt a problémát az információk elkülönített tárolásával lehet megoldani. Az elkülönített tárhelyre gondolhatunk úgy, mint egy virtuális könyvtárra, amit a szerelvény, az alkalmazástartomány, illetve a felhasználó alapján különítünk el. Tetszés szerint választhatunk egy általánosabb elkülönített tárhelyet, illetve virtuális könyvtárat is, ami csak a szerelvénytől és az adott felhasználótól függ.

A részleges jogosultságokkal rendelkező szerelvények a saját elkülönített tárterületükhöz hozzáférhetnek, de a fájlrendszer többi részéhez nem. Az elkülönített tárhely könyvtárát a többi szerelvény és felhasználó nem látja. Az elkülönített tárhelyeket a `System.IO.Isolated-`

Storage névtér osztályain keresztül használhatjuk. Az IsolatedStorageFile osztály tagfüggvényei nagyon hasonlítanak a System.IO.File osztály függvényeire, ami nem meglepő, hiszen egyenesen a System.IO.File osztályból származik. Egy elkülönített tárhelyre szinte ugyanazzal a kóddal írhatunk, mintha egy sima fájlba írnánk:

```
IsolatedStorageFile iso =
  IsolatedStorageFile.GetUserStoreForDomain( );

IsolatedStorageFileStream myStream = new
  IsolatedStorageFileStream( "SavedStuff.txt",
  FileMode.Create, iso );
StreamWriter wr = new StreamWriter( myStream );
// néhány wr.Write utasítás elhagyva
wr.Close();
```

Az adatok olvasása szintén ismerős lesz mindazok számára, akik használtak már fájlműveleteket:

```
IsolatedStorageFile isoStore =
  IsolatedStorageFile.GetUserStoreForDomain( );

string[] files = isoStore.GetFileNames( "SavedStuff.txt" );
if ( files.Length > 0 )
{
  StreamReader reader = new StreamReader( new
    IsolatedStorageFileStream( "SavedStuff.txt",
    FileMode.Open,isoStore ) );

  // Néhány reader.ReadLines() hívás elhagyva

  reader.Close();
}
```

Az elkülönített tárhelyek segítségével olyan elfogadható méretű adatelemeket tárolhatunk, amelyek lehetővé teszik a részleges jogosultságokkal rendelkező kód számára, hogy az adatokat mentsen, illetve töltsön be a helyi lemez körültekintően kiosztott területén. A .NET környezet korlátozza az egyes alkalmazások számára elérhető elkülönített tár területét. Ez megakadályozza, hogy egy rosszindulatú kód túl sok lemezterületet kebelezzen be, aminek következtében használhatatlanná válhatna a megtámadott rendszer. Az elkülönített tárhely rejtve marad a többi felhasználó és a többi program elől, ezért nem szabad olyan beállításokat tárolnunk itt, amiket a rendszergazdának esetleg meg kell változtatnia. Bármennyire is rejtett az elkülönített tárhely, a kezeletlen kódnak és a megfelelő jogosultságokkal rendelkező felhasználóknak azért ki van szolgáltatva. Soha ne használjuk az elkülönített tárhelyet titkos anyagok tárolására, hacsak nem alkalmazunk egy megfelelően biztonságos titkosító eljárást.

Ahhoz, hogy a szerelvényünk meg tudjon élni az esetleges biztonsági korlátozások mellett is, mindig különítsük el a tárolni kívánt adatfolyamaink létrehozását. Ha a szerelvény a Világhálón keresztül is futtatható, vagy hozzáférhet egy az Interneten futó kód, akkor érdemes meggondolnunk az elkülönített tárhely használatát.

Egyéb védett erőforrásokra is szükségünk lehet. Ha szükség van arra, hogy a programunk hozzáférjen ezekhez az erőforrásokhoz, az általában azt jelzi, hogy a rendszernek tökéletesen meg kell bíznia a programban. Ezen kívül csak azt tehetjük, hogy következetesen kerüljük a védett erőforrás használatát. Vegyük például a Windows rendszerleíró adatbázisát. Ha a programunknak hozzá kell nyúlni a rendszerleíró adatbázishoz, akkor úgy kell telepítenünk a programot a felhasználó gépére, hogy az rendelkezzen a szükséges jogosultságokkal az adatbázis eléréséhez. Az Interneten futó alkalmazásoknál egyszerűen lehetetlen biztonságosan megoldani a rendszerleíró adatbázis módosítását – és ez így is van rendjén.

A .NET biztonsági modell azt jelenti, hogy a programunk által végrehajtott műveleteket a környezet összeveti a program jogosultságaival. Figyeljünk oda, hogy milyen jogosultságokra van szüksége a programnak, és próbáljuk meg a legalacsonyabb szintűre csökkenteni őket. Ne akarjunk olyan jogosultságokat szerezni, amelyekre nincs is szükségünk. Minél kevesebb védett erőforrásra van szüksége egy szerelvénynek, annál kisebb az esélye annak, hogy biztonsági kivételt fog kiváltani. Kerüljük a védett erőforrások használatát, és ha tehetjük, igyekezzünk más megoldás után nézni. Amikor tényleg magasabb szintű jogosultságokra van szükségünk egy algoritmusnál, különítsük el az adott kódrészletet egy saját szerelvénybe.

48. tipp

Ismerjük meg az elérhető segédeszközöket és forrásokat

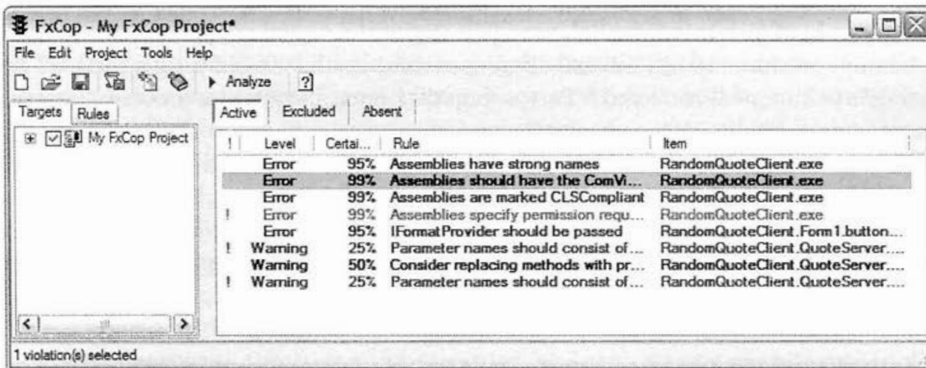
Izgalmas időket élünk mi, C# és .NET fejlesztők. Az eszközök még elég újak, így a teljes fejlesztőtábor még azon igyekszik, hogy megtanulja a legjobban kihasználni őket. Számos forrást találhatunk azonban, amelyekből táplálkozva fejleszthetjük a tudásunkat, és bővíthetjük a .NET és a C# társadalom számára rendelkezésre álló tudásbázist. Jómagam is ezeket az eszközöket használom nap mint nap, és ezt szoktam javasolni a többi C# fejlesztőnek is. A C# programozás követendő gyakorlatainak gyűjteménye még csak most íródik. Kövessük nyomon a fejleményeket, és vegyük ki mi is a részünket az alkotófolyamatból.

Az első eszköz, aminek szerepelnie kell minden C# fejlesztő eszköztárában, a `www.nunit.org` internetcímen elérhető **NUnit**. Az NUnit automatizált egységtesztelő eszköz, ami a szolgáltatásait tekintve a JUnit programhoz hasonlít. A legtöbb fejlesztőhöz hasonlóan én is utálok teszteket írni és a kód tesztelésével tölteni az időmet. Az NUnit olyan hatékonnyá teszi ezt a folyamatot, hogy a fejlesztők rendszeresen rászoknak arra, hogy az összes C# osztályukat leteszteljék. Ha egy osztálykönyvtárprojektet készítek, mindig hozzáadok egy NUnit tesztprojektet is, és a tesztek futtatását beleveszem a program automatizált felépítésébe. Olyan új beállításokat szoktam létrehozni, amelyek végrehajtják

a program felépítését és a teszteket is minden egyes fordításkor. Ilyenkor az aktív beállítások módosításával szabályozni tudom, hogy lefussanak-e a tesztek a program felépítésekor. Alapbeállításként futtatni szoktam a teszteket. Ha olyan tesztet futtatok, amihez szükség van a felhasználói felületre, akkor átváltok egy másik beállításra.

Azon kívül, hogy az NUnit program remekül használható, sok érdekes megoldást leshetünk el, ha megvizsgáljuk a forráskódját. Az NUnit néhány igen fejlett visszatekintési mintát használ a szerelvények betöltéséhez és teszteléséhez. Jellemzők segítségével találja meg a tesztcsomagokat, az egyes teszteseteket, illetve az egyes teszteseteknél várt eredményeket (lásd a 42. tippet). Nagyszerű példa ez arra, hogyan lehet ezeket a megoldásokat egy olyan eszköz kifejlesztésére használni, ami dinamikusan állítja be önmagát, és amit így széles körben fel lehet használni.

A következő eszköz az **FXCop**, egy ingyenes eszköz, amit a GotDotNet oldalról tölthetünk le (`www.gotdotnet.com`). Az FXCop összeveti a szerelvényünk IL kódját egy előre meghatározott szabály-, illetve ajánláshalmazzal, majd jelenti, ha valamilyen összeférhetetlenséget talál. Minden szabályhoz tartozik egy megbízhatósági mérce és egy indoklás.
A könyvben található összes tipphez hasonlóan, a szabály dokumentációjához hozzá tartozik a tanács rövid magyarázata, így eldönthetjük, hogy a tanács illik-e az adott helyzetre. Azt is beállíthatjuk, hogy minden egyes szabályt alkalmazni kell-e a projektünk esetében. Én személy szerint nem egészen értek egyet az FXCop minden szabályával, és akkor még finoman fogalmaztam. Ennek ellenére az FXCop az NUnit programhoz hasonlóan a szokásos programépítési folyamat részévé válhat. Minden fordítás után következhet még egy lépés, amiben az FXCop a kiválasztott szabályok alapján elemzi a kódunkat. A 6.1. ábrán az FXCop kimenetére láthatunk példát. Bár néhány javaslat nem igazán van ínyemre (például az, hogy minden szerelvényt lásson a COM), maga az eszköz nagyon hasznos, mert arra kényszerít, hogy átgondoljunk néhány olyan döntést, amit esetleg egy alapbeállítás meghagyásával hoztunk meg.



6.1. ábra

Az FXCop egy projekt elemzése közben

Az IldAsm egy IL visszafordító (disassembler). A könyv különböző részein előfordult, hogy bemutattam azt az IL kódot, amit a fordító a különféle C# kódrészletekből állított elő. Bár kétlem, hogy bárki is IL kóddal akarna programozni egy magasszintű nyelv helyett, azért nem árt, ha megismerkedünk vele. A különböző C# szerkezetekből előállított IL kódot megismerve jobb fejlesztő válhat belőlünk. A .NET Framework SDK részét képező IldAsm segítségével megvizsgálhatjuk a saját és a .NET keretrendszer szerelvényeinek IL kódját is. Az IL minden fejlesztő számára elérhető. Az IldAsm segítségével megismerhetjük a szerelvényeink köztes nyelvezetét, de még ennél is jobb módja annak, hogy jobban megismerjük a .NET keretrendszer szerelvényeit, ha beszerezzük a forráskódot.

Ezek azok az eszközök, amelyeknek szerepelniük kell egy szokásos eszköztárban. Az eszközök beszerzése azonban csak az egyik módja az ismereteink bővítésének. Az Interneten számos forrást találunk, amelyeket használhatunk, illetve tevékenyen részt vehetünk a közösségek munkájában, ezzel is tanulva, és tovább gyarapítva ismereteinket a C# nyelvről és a .NET keretrendszerről. Először is ott van a legfontosabb, a GotDotNet oldal (`www.gotdotnet.com`), ami a .NET fejlesztőcsapat hivatalos oldala. A C# csapatnak az MSDN webhelyen van egy oldala, aminek jelenlegi címe `msdn.microsoft.com/vcsharp/` (időnként más helyre költöznek, ha éppen átszervezik az MSDN oldalait). Ha alapvetően web alapú alkalmazásokkal foglalkozunk, akkor tegyünk egy próbát a `www.asp.net` oldallal, ami az ASP.NET csapat oldala. Ha főleg Windows Forms alapokra épülnek a programjaink, akkor a `www.windowsforms.net` oldalon próbálkozzunk, ami a Windows Forms csapat hivatalos oldala. Ezeken az oldalakon programfelépítési mintákat találhatunk, amelyek kitűnően használhatók a saját alkalmazásainkban. Mindegyikhez ott van a forráskód is, amit megvizsgálhatunk, vagy akár az igényeinknek megfelelően módosíthatunk is. Az utolsó, de egyben a legfontosabb hely, amit feltétlenül ismerni kell, az MS Patterns & Practices oldal. Ez az oldal jelenleg a `www.microsoft.com/resources/practices/` címen található. Itt is programtervezési mintákat és a hozzájuk tartozó kódvázakat találunk, amelyek segítségével követhetjük a bevált módszereket. Ezt a területet folyamatosan frissítik újabb és újabb példaprogramokkal és könyvtárakkal, amelyek segítenek a gyakori problémák megoldásában. A könyv írásakor 10 különböző alkalmazásblokkot használhatunk a gyakori programozási igények megvalósításához. Biztos vagyok benne, hogy mire az olvasó e sorokat olvassa, már sokkal több lesz ezekből.

Javaslom, hogy keressük fel a C# csapat gyakran ismételt kérdéseinek listáját is a `http://blogs.msdn.com/csharpfaq` címen. Ezen kívül a C# fejlesztők csapatának számos tagja saját webnaplót (blogot) vezet, ahol megtárgyalják a C# nyelvvel kapcsolatos problémákat.

A naplók frissített listáját a `http://msdn.microsoft.com/vcsharp/team/blogs/` címen találjuk meg.

Ha ennél is többre vágyunk, és még mélyebben szeretnénk elmerülni a nyelv és a környezet rejtelmeiben, akkor nézzük meg a megosztott CLI forrást (amelynek kódneve „rotor"). Ebben benne van a .NET keretrendszer magja és egy C# fordító. A forráskód áttanulmányozásával mélyebb betekintést kaphatunk a C# nyelvbe és a .NET keretrendszerbe. A közzétett változatban nem találjuk meg a kereskedelmi forgalomban kapható .NET keretrendszer minden szerelvényét; a Windowszal kapcsolatos kódrészletek például nem szerepelnek a megosztott forráskódban. De ami benne van, az is nagyon gazdag forrás, aminek a segítségével sokat megtudhatunk a CLR és a C# nyelv belső működési folyamatairól.

A megosztott CLI forráshoz tartozó C# fordítót C++ nyelven írták, ahogyan az alacsonyszintű CLR kód részeit is. Ahhoz, hogy teljes egészében megértsük, komoly C++ programozói tapasztalattal kell rendelkeznünk, és tisztában kell lennünk a fordítók felépítésével is. A modern nyelvekhez készült fordítók ugyan elég bonyolult programok, de a CLR forráskódja értékes eszköz a .NET keretrendszer alapszolgáltatásainak megvalósítását megismerni kívánók számára.

Szándékosan fogtam rövidre a fenti listát. Éppen csak érintettük azt a rengeteg forrást, amit a Microsoft, egyéb webhelyek és könyvek kínálnak a számunkra. Minél többet használjuk ezeket az eszközöket, annál komolyabb szakértőkké válhatunk. A teljes C# és .NET közösség folyamatosan halad előre. Mivel a fejlődés gyors, a források listájának összetétele is állandóan változik. Tanuljunk, és váljunk mi is a fejlődés részeseivé.

49. tipp

Készüljünk fel a C# 2.0 érkezésére

A C# 2.0 néhány fontos újdonsággal bővíti a C# nyelvet. A ma még követendőnek tartott módszerek közül néhány változni fog az új kiadásban megjelenő új eszközök hatására. Bár ezeket a szolgáltatásokat most még nem használjuk, érdemes felkészülni az érkezésükre.

A Visual Studio .NET 2005 kiadásakor egy új, frissített C# nyelvet kapunk. A nyelv újdonságai minden bizonnyal segíteni fognak abban, hogy még hatékonyabb programozókká válhassunk. Kevesebb sorban írhatunk majd könnyebben újrahasznosítható kódot és magasabb szintű szerkezeteket. Mindent egybevéve több munkát végezhetünk el rövidebb idő alatt.

A C# 2.0 négy fő újdonságot kínál: az általános elemeket (generics), a bejárókat (iterators), a névtelen tagfüggvényeket (anonymus methods) és a részleges típusokat (partial types). Ezek az újdonságok azt a célt szolgálják, hogy hatékonyabb legyen a fejlesztés a C# nyelven. Ebben a tippben hármat mutatunk be ezek közül, és azt, hogy miként készülhetünk fel az érkezésükre.

Az **általános elemek** (generics) minden más újdonságnál nagyobb hatást fognak gyakorolni programozási módszereinkre. Az általános elemek nem csak a C# nyelvben találhatók meg. A C# általános elemeinek megvalósításához a Microsoft kibővíti a CLR környezetet, a Microsoft Intermediate Language (MSIL) nyelvet és magát a C# nyelvet is. A C#, a Managed C++ (kezelt C++) és a VB .NET is használható lesz általános elemek előállítására, a J# pedig képes lesz felhasználni őket.

Az általános elemek „paraméteres többalakúságot" kínálnak, ami virágnyelven annyit tesz, hogy egyetlen forrásból kiindulva egy sor hasonló osztályt készíthetünk. A fordító eltérő változatokat állít elő, ha megadjuk az általános elem paraméterének típusát. Az általános elemeket olyan algoritmusok kifejlesztéséhez használhatjuk, amelyeknek a paraméterek segítségével megadhatjuk, hogy milyen típuson végezzék el a teendőiket. A .NET `Collections` névtérben sok kitűnő jelöltet találunk ehhez. A `Hashtable`, az `ArrayList`, a `Queue` és a `Stack` osztályok is képesek különböző típusok tárolására, anélkül, hogy változtatni kellene a megvalósításukon. Ezek a gyűjtemények annyira kínálják magukat, hogy a .NET Framework 2.0 kiadásában lesz egy `System.Collections.Generic` névtér, amiben minden jelenlegi gyűjteménynek meglesz az általános változata. A C# 1.0 egy hivatkozást tárol a `System.Object` típusra. Bár a jelenlegi megoldást minden típushoz újra felhasználhatjuk, sok hátránya van, és nem is típusbiztos. Vegyük például az alábbi kódot:

```
ArrayList myIntList = new ArrayList( );
myIntList.Add(32 );
myIntList.Add(98.6 );
myIntList.Add("Bill Wagner" );
```

Ezt le lehet ugyan fordítani, de valószínűleg nem szándékozzuk megtenni. Tényleg olyan programszerkezetet akarunk, amihez egy egymástól teljesen eltérő elemeket tartalmazó tárolóra van szükség? Vagy csak így próbáltuk megkerülni a nyelv korlátait? Ez a gyakorlat azzal jár, hogy amikor elemeket törlünk a gyűjteményből, akkor előbb újabb kódot kell írnunk csak azért, hogy előbb eldönthessük, hogy egyáltalán milyen típusú elemeket tettünk a gyűjteménybe. Minden egyes elemnél át kell alakítanunk a `System.Object` hivatkozást a listába tett objektum típusára.

És ez még nem minden. Az értéktípusoknál további költségekkel kell számolnunk, amikor ilyen 1.0 stílusú gyűjteményekhez adjuk őket. Amikor egy értéktípust teszünk egy gyűjteménybe, minden esetben be kell azt csomagolnunk. Ha hozzá szeretnénk férni a gyűjtemény egy eleméhez, újabb költségekkel kell számolnunk, hiszen az értéket ki kell csomagolnunk. Ezek csekély költségek, de elemek ezreit tároló gyűjteményeknél a sok kicsiből hamar sok lesz. Az általános elemeknél nem jelentkezik ez a többletköltség, mert minden értéktípushoz konkrét tárgykód készül.

Akik számára ismerősek a C++ sablonok, azok könnyen boldogulnak majd a C# általános elemeivel is, mert a nyelvtan nagyon hasonló. Az általános elemek belső működése azonban egészen más. Nézzünk meg egy egyszerű példát, hogy lássuk, hogyan működnek az általános elemek, és hogyan valósítják meg őket. Vegyük az alábbi részletet egy listaosztályból:

```
public class List
{
  internal class Node
  {
    internal object val;
    internal Node next;
  }
  private Node first;


 public void AddHead( object t )
  {
    // ...
  }

  public object Head()
  {
    return first.val;
  }

}
```

Ez a kód `System.Object` hivatkozásokat tárol a gyűjteményben. Amikor használjuk, mindig típusátalakítást kell végeznünk azokon az elemeken, amelyekhez hozzá szeretnénk férni a gyűjteményben. A C# általános elemeivel azonban így határozzuk meg ugyanezt az osztályt:

```
public class List < ItemType >
{
  private class Node < ItemType >
  {
    internal ItemType val;
    internal Node < ItemType > next;
  }

  private Node < ItemType > first;

  public void AddHead( ItemType t )
  {
    // ...
  }
```

```
    public ItemType Head( )
    {
      return first.val;
    }
}
```

Az `object` helyére az `ItemType` kerül, ami az osztály meghatározásában szereplő paramétertípus. A C# fordító az `ItemType` helyére a megfelelő típust teszi, amikor létrehozzuk a `list` egy példányát. Nézzük meg például ezt a kódot:

```
List < int > intList = new List < int >();
```

A létrehozott MSIL megadja, hogy az `intList` egészeket, és csakis egészeket tárol. Az általános elemeknek számos előnye van a ma használatos megvalósításokkal szemben. Kezdjük azzal, hogy a C# fordító hibát jelez a fordításkor, ha az egészeken kívül valami mást próbálunk hozzáadni a gyűjteményhez. Ma még csak úgy csíphetjük fülön ezeket a hibákat, ha futás közben teszteljük a kódot.

A C# 1.0 változatban egy be- és egy kicsomagolással kell fizetnünk azért, ha értéktípust teszünk egy olyan gyűjteménybe, ami `System.Object` hivatkozásokat tárol. Az általános elemeknél a JIT fordító létrehoz egy konkrét gyűjteménypéldányt az adott értéktípusok tárolására, s így nem kell be- és kicsomagolni az elemeket. És ez még nem minden. A C# tervezői el akarják kerülni azt a kódtömeget, ami az embernek a C++ sablonokról az eszébe jut. Helytakarékossági okból a JIT fordító csak a típus egy változatát készíti el az összes hivatkozási típushoz. Ezzel egy olyan méret–sebesség kompromisszum jön létre, ahol az értéktípusok kapnak egy saját változatot minden típushoz (a csomagolás elkerülése végett), a hivatkozási típusok pedig egyetlen `System.Object` hivatkozásokat tároló futásidejű változaton osztozkodnak (elkerülve a kódtömeget). A fordító ennek ellenére ezeknél a gyűjteményeknél is jelzi, ha nem a megfelelő hivatkozási típust használjuk.

Az általános elemek megvalósításához a CLR és az MSIL nyelv is átesett néhány változáson. Amikor lefordítunk egy általános osztályt, akkor az MSIL minden paraméterezett típusnál egy helyőrzőt használ. Nézzük meg ezt a két MSIL tagfüggvény-bevezetést:

```
.method public AddHead (!0 t) {
 }

.method public !0 Head () {
}
```

A `!0` kifejezés egy helyőrző, aminek a helyére egy típus kerül, amikor egy konkrét példányt vezetünk be és hozunk létre.

Egy lehetséges kitöltött változat a következő:

```
.method public AddHead (System.Int32 t) {
 }

.method public System.Int32 Head () {
}
```

Ehhez hasonlóan a változók példányai egy konkrét típust tartalmaznak. A korábbi egészekből álló lista bevezetéséből ez lesz:

```
.locals (class List<int>)
newobj void List<int>::.ctor ()
```

Ez szépen szemlélteti azt, ahogyan a C# fordító és a JIT fordító együttműködik az általános elemek használatakor. A C# fordító olyan MSIL kódot hoz létre, amiben helyőrzők vannak minden egyes típusparaméter helyén. A JIT fordító konkrét típusokra cseréli ezeket a helyőrzőket. A hivatkozási típusoknál `System.Object` típusokra, az értéktípusoknál pedig mindig az adott értéktípusra. Minden egyes alkalommal, amikor létrehozunk egy példányt egy általános típusból, akkor ott lesz a típusinformáció is, így a C# fordító kikényszerítheti a típusbiztonságot.

Az általános elemekhez kapcsolódó megszorítások meghatározása szintén komoly hatással van arra, hogy miként fogjuk használni az általános elemeket. Ne feledjük, hogy egy általános futásidejű osztály egy konkrét példányának létrehozása egészen addig nem következik be, amíg a CLR futásidőben be nem tölti, és el nem készíti az adott példányt. Ahhoz, hogy előállítsa az MSIL kódot egy általános osztály összes lehetséges példányához, a fordítónak ismernie kell a paraméterezett típus képességeit a létrehozott általános osztályoknál. A C# ezt a problémát a megszorításokkal oldja meg. A megszorítások a paraméterezett típus várható képességeit adják meg. Vegyük például egy bináris fa általános megvalósítását. A bináris fák sorba rendezve tárolják a bennük lévő objektumokat, ezért csak olyan típusokat tárolhatnak, amelyek megvalósítják az `IComparable` felületet. Ezt a követelményt a megszorítások segítségével adhatjuk meg:

```
public class BinaryTree < ValType > where
  ValType : IComparable < ValType >
{
}
```

Ezt a meghatározást használva nem lehet lefordítani a programot, ha olyan osztállyal akarunk létrehozni egy `BinaryTree` példányt, ami nem támogatja az `IComparable` felületet. Több megszorítást is megadhatunk. Tegyük fel, hogy csak olyan objektumokból álló

`BinaryTree` fákat engedélyezünk, amelyek támogatják az `ISerializable` felületet. Ehhez csak a további megszorítást kell megadnunk. Figyeljük meg, hogy a felületek és maguk a megszorítások is lehetnek általános típusok:

```
public class BinaryTree < ValType > where
        ValType : IComparable < ValType > ,
  ValType : ISerializable
{
}
```

Minden paraméterezett típushoz megadhatunk egy alaposztályt és egy akárhány felületből álló megszorításhalmazt. Továbbá azt is kiköthetjük, hogy a típus konstruktorának paraméter nélkülinek kell lennie.

A megszorításoknak van még egy előnyük. A fordító feltételezi, hogy az általános osztályunkban lévő objektumok támogatják az összes felületet (vagy alaposztály-tagfüggvényt), amit a megszorítások között felsoroltunk. Megszorítások hiányában a fordító csak a `System.Object` osztály tagfüggvényeivel számol. Más tagfüggvények használatához típusátalakításra lesz szükségünk. Ha tehát olyan tagfüggvényt használunk, aminek a meghatározása nem szerepel a `System.Object` osztályban, akkor ezt a követelményt mindig adjuk meg a megszorításoknál.

A megszorítások eggyel több okot adnak arra, hogy miért érdemes bátran használnunk a felületeket (lásd a 19. tippet). Ha felületekkel határoztuk meg a programunk működését, akkor viszonylag egyszerűbben adhatjuk meg a megszorításokat is.

A **bejárók** nyújtotta új nyelvtan segítségével sokkal kevesebb kód megírásával készíthetünk mintát. Képzeljük el, hogy létrehozunk egy különleges új tárolóosztályt. A felhasználók támogatásához olyan tagfüggvényeket kell létrehoznunk, amelyek támogatják ennek a gyűjteménynek a bejárását és a gyűjtemény egyes objektumainak visszaadását.

Ma még úgy csinálnánk mindezt, hogy létrehoznánk egy osztályt, ami támogatja az `IEnumerator` felületet. Az `IEnumerator` két tagfüggvényt, a `Reset` és a `MoveNext` függvényeket, és egy `Current` nevű tulajdonságot tartalmaz. Ezen kívül az `IEnumerable` felületet hozzáadnánk a gyűjteményen megvalósított felületek listájához, és annak `GetEnumerator` tagfüggvénye visszaadna egy `IEnumerator` objektumot a gyűjteményhez. Mire mindezzel megvagyunk, addigra megírtunk egy újabb osztályt legalább három függvénnyel, és némi állapotkezeléssel, meg egy másik tagfüggvénnyel a fő osztályban. Szemléltetésképpen nézzük meg ezt a teljes oldalnyi kódot, amit ma még meg kell írnunk ahhoz, hogy kezelhessük egy lista bejárását:

```
public class List : IEnumerable
{
```

```
  internal class ListEnumerator : IEnumerator
  {
    List theList;
    int pos = -1;

    internal ListEnumerator( List l )
    {
      theList = l;
    }

    public object Current
    {
      get
      {
        return theList [ pos ];
      }
    }

    public bool MoveNext( )
    {
      pos++;
      return pos < theList.Length;
    }

    public void Reset( )
    {
      pos = -1;
    }
  }

  public IEnumerator GetEnumerator()
  {
    return new ListEnumerator( this );
  }

  // Egyéb tagfüggvények törölve
}
```

A C# 2.0 új nyelvtant vezet be a `yield` kulcsszó révén, aminek a segítségével sokkal tömörebben írhatunk meg egy ilyen bejárást. Az előző kód C# 2.0 változata tehát így néz ki:

```
public class List
{
  public object iterate()
  {
    int i=0;
    while ( i < theList.Length ( ) )
```

```
        yield theList [ i++ ];
    }

    // Egyéb tagfüggvények törölve
}
```

A `yield` utasítás segítségével a durván 30 sort mindössze 6 sorral váltottuk ki. Ez kevesebb hibalehetőséget, rövidebb fejlesztési időt és kevesebb karbantartandó kódot jelent. Csupa jó dolog.

A színfalak mögött a fordító állítja elő azt az MSIL kódot, ami megfelel a ma használt változat 30 sorának. Vagyis a fordító elvégzi ezt helyettünk, hogy nekünk ne kelljen ezzel foglalkoznunk. A fordító előállít egy osztályt, ami megvalósítja az `IEnumerator` felületet, és hozzáadja azt a támogatott felületek listájához.

Az utolsó fontos újdonság a **részleges típusok** megjelenése. A részleges típusok segítségével egy C# osztály megvalósítását darabolhatjuk fel úgy, hogy az egyes részek más-más forrásfájlba kerülnek. A napi munkánk során aligha fogjuk magunktól használni ezt a szolgáltatást. A Microsoft a fejlesztőkörnyezetek és kódgenerátorok támogatásához javasolta ezt a módosítást a C# nyelven. Ma mindig van egy rész az ablakosztályainknál, amiben ott van a VS .NET tervező által előállított összes kód. A jövőben ezek az eszközök részleges osztályokat fognak létrehozni, és ezt a kódot egy külön fájlba fogják tenni.

Ezt a szolgáltatást úgy használhatjuk, ha hozzáadjuk a `partial` kulcsszót az osztály bevezetéséhez:

```
public partial class Form1
{
  // Varázsló által előállított kód:
  private void InitializeComponent()
  {
    // szintén a varázsló által előállított kód...
  }
}

// Egy másik fájlban:
public partial class Form1
{
  public void Method ()
  {
    // stb...
  }
}
```

A részleges típusokra vonatkozik néhány korlátozás. Csak a forrás szintjén működik a szolgáltatás, vagyis semmilyen különbség nincs az egyetlen, illetve a több fájl alapján elkészített MSIL között. Ezen kívül a teljes típushoz szükséges fájlokat egyetlen szerelvénybe kell fordítani, és nincs olyan automatizmus, ami ellenőrizné, hogy tényleg megadtuk-e a teljes típus felépítéséhez szükséges összes forrásfájlt. Számtalan gondot okozhatunk azzal, ha az osztályaink meghatározásait feldaraboljuk és külön fájlokba tesszük, ezért azt javaslom, hogy csak akkor használjuk ezt a szolgáltatást, ha a fejlesztőkörnyezet részleges típusokat használ a forrás előállításánál. Ahogy már bemutattam, ez az ablakoknál előfordulhat. A VS .NET a típusos `DataSet` objektumokhoz (lásd a 41. tippet) és a webszolgáltatási proxykhoz is részleges típusokat állíthat elő, hogy hozzáadhassuk a saját tagjainkat ezekhez az osztályokhoz.

Néhány C# 2.0 újdonságról nem szóltam, mert nem fogják lényegesen megváltoztatni azt, ahogy ma is programozunk. A típusainkat könnyebben használhatjuk általános elemekkel, ha felületeket határozunk meg a program viselkedésének leírásához. Ezeket a felületeket megszorításként használhatjuk. Az új bejárónyelvtan segítségével hatékonyabban valósíthatjuk meg egy gyűjtemény elemeinek sorra vételét. Az egymásba ágyazott felsorolókat könnyen és gyorsan lecserélhetjük ennek az új nyelvi elemnek a segítségével. A testreszabott külső osztályokat azonban nem lesz egyszerű lecserélni. Már most figyeljünk oda, hogy mire fogjuk használni ezeket az újdonságokat, és ennek fényében írjuk meg az alkalmazásainkat. Ha így teszünk, sokkal könnyebb lesz minimális munka árán frissíteni a meglévő kódunkat a C# 2.0 újsütetű elemeinek segítségével.

50. tipp

Ismerkedjünk meg az ECMA szabvánnyal

Az ECMA szabvány képviseli a hivatalos álláspontot arról, hogy a C# nyelv elemeinek hogyan kell viselkednie. Az ECMA-334 határozza meg a C# 1.0 szabványt. A C# 2.0 változatra vonatkozó javaslatokról Anders Hejlsberg, Scott Wiltamuth és Peter Golde *The C# Programming Language* (A C# programozási nyelv, Addison-Wesley, 2003) című könyvében olvashatunk. Ez inkább amolyan kézikönyv, nem tankönyv. Aprólékos részletességgel írja le, hogyan működnek a nyelv egyes elemei. Minden nyelvi elemhez magyarázatokat is találunk, így jobban megérthetjük, hogy az egyes elemekre miért van szükség. Mialatt ezt a könyvet írtam, az említett kézikönyv végig itt volt előttem kinyitva az asztalon.

Ha komolyan foglalkozunk C# programozással, akkor meg kell értenünk a nyelvet, beleértve az egyes nyelvi elemek fontosságát is. Egyszerűbb lesz a dolgunk, ha tudjuk, hogy mikor melyik elemre van szükség a saját munkánk során. Ezáltal jobban átlátjuk a finom különbségeket a különböző nyelvi kifejezési formák között.

A C# nyelven kívül tökéletesen tisztában kell lennünk a Common Language Runtime (CLR) környezettel is. A CLR és a Common Language Infrastructure (CLI) szabványokat az ECMA-335 dokumentum, vagyis a CLR szabvány tartalmazza. A C# nyelvhez hasonlóan ez is 1.0-s változatú szabvány. A CLI 2.0 változatát James Miller és Susann Ragsdale magyarázza el a *The Common Language Infrastructure Annotated Standard* (A közös nyelvi infrastruktúra szabványa jegyzetekkel, Addison-Wesley, 2003) című kiadványában. Ebben a kézikönyvben megtaláljuk a Common Language Subsystem (CLS) leírását is, ami hozzásegíthet minket a CLS megfelelés tökéletesebb megértéséhez. Ezen kívül jobban megérthetjük belőle az ECMA szabványnak a .NET futásidejű környezetére és infrastruktúrájára vonatkozó részeit.

A C# és a CLR szabvánnyal foglalkozó bizottságok is folyamatosan közzéteszik a C# nyelv és a CLR 2.0 változatáról szóló munkadokumentumokat. Ezekből érthetjük meg igazán, miként fog fejlődni és változni a C# a jövőben.

A jelenlegi szabvány tökéletesebb megértése, illetve a javasolt fejlesztések megismerése ahhoz is hozzásegít minket, hogy olyan kódot írjunk, ami kiállja az idő próbáját. Ha megismerkedünk azokkal az elemekkel, amelyeket várhatóan hozzá fognak adni a nyelvhez és a környezethez, könnyebben írhatunk olyan programokat, amelyeket még sokáig lehet majd használni a jövőben, és előre láthatjuk azt is, hogy milyen módosításokat kell majd esetleg elvégeznünk rajtuk.

Az idő előrehaladtával a programok folyamatosan változnak. A C# nyelv is fejlődni és változni fog, valószínűleg a 2.0-s változatát követően még sok időt és kiadást meg fog érni. Ezt az eszközt használjuk nap nap után. Ismerjük meg hát a hivatalos szabványokat, és folyamatosan kövessük a fejlődésüket.

Tárgymutató

#endif 23
#if 23, 28
.isinst 23
.NET 1.x gyűjtemény 98
.NET biztonsági modell 255, 258
.NET FCL 30, 146
.NET keretrendszer 2, 187
.NET Remoting 181
.NET Security 255
.property utasítás 10
/CLR kapcsoló 253
/unsafe 256
[] jelölés 216
[NonSerialized] 137
== 148
0 állapot 48
0 kezdőérték 76
0. generációs objektumok 74
1. generációs objektumok 74
2. generációs objektumok 74

A, Á

A.GetHashCode() 60
ablakbeállítások 198
Abrahams 244
AcceptChanges 218
adatbázis-kapcsolatok 71
adatelem 71
adatérvényesítés 205
adathalmaz 71
adatkapcsolás 2
adatkötés 199, 221, 235
adatkötés-kezelő 203
adatoszlop 71
adatsor 71
adattábla 71
adattagok elérése 234
adattárolás 40
add 123
AddressRecord 222
AfxMessageBox 252
alacsonyszintű adattároló típusok 40
alakzatok 152
alapértelmezett alkalmazástartomány 196
alapértelmezett beállítófájl 196
alapértelmezett kezdőérték 137
alapértelmezett konstruktor 76
alapértelmezett paraméterek 81
alapértelmezett sorosító tagfüggvények 138
alapgarancia 244
alaposztály 102, 114, 170
alaposztály frissítése 156
alapvető típusok 12
alkalmazás indítása 176

alkalmazásbeállító állomány 164
alkalmazásfüggő kivételosztályok 237
alkalmazáskönyvtár 162
alkalmazástartomány 196
alkalmazásszintű nyomkövetés 194
állandók 11
állapotjelzés 102
állományazonosítók 71
AllowPartiallyTrustedCallers 164
alsó korlát 69
általános elemek 100, 261
általános felületek 100
általános formátum 33
általános gyűjtemények 100
AND 50
AND kulcsszó 28
anonymus method 261
Application 13
Application Data 198
Application.LocalAppDataPath tulajdonság 198
Application.LocalUserDataPath 198
Application.UserDataPath 198
ApplicationException osztály 239
appSettings 195
ArrayList 69, 99, 215
ArrayList osztály 215
ArrayListEnumerator 178
as 16, 20, 88
ASP.NET 260
ASP.NET alkalmazások 196, 206
ASP.NET futásidejű környezet 131
Assembly.LoadFrom() 227
AssemblyCulture 163
assemblyInfo.cs 163
AssemblyVersion 163
átalakító műveletek 152
átmeneti objektumok 156
AttributeTargets 231
AttributeUsage 228
azonosság 23, 52, 54

B

BadCastException 22
BadClass 106
base() 83
BaseWidget 158
beállítások 164
beállítások tárolása 195
beállító utasítások 75, 77
beállítófájlok 162, 164, 195
becsomagolás 95
BeginEdit 226
behelyettesítés 172
beillesztő eljárás 176
bejárás 68
bejáró 213, 261, 266
belső adatok 127
belső kivétel 241
belső osztály 178, 179
belső osztályobjektumokra mutató hivatkozások 126
betokozás 174
betöltő 162
bevezetés 130
bináris fa 265
bináris megfelelőség 8
bináris összetevők 10, 161
BinaryTree 265
BindingManager 199
BindToDataErrors 210
binpath 162
BitArray osztály 216
bitblokkosítható típusok 249
bitenkénti és 50
bitértékek 216
bitműveletek 216
biztonsági ellenőrzés 176
biztonságos kód 255
biztonságos szerelvények 256
bolygó 49
bővítőcsomagok meghatározása 135

boxing 67
BSTR 250
buckets 59
BuildMsg 171
burkolók 127

C, Cs

C stílusú Win32 tagfüggvények elérése 252
C# 2.0 100, 261
C# Builder 178
C++ 36
C++ sablonok 263
Cancel gomb 208
CancelEdit 226
Cargill 243
CAS 255
CAS API 255
catch 17, 237
CausesValidation tulajdonság 208
CheckState() 27
ciklikus szerkezetek 245
ciklus 66, 245
ciklusváltozó hossza 67
ciklusvezérlő utasítás 66
címeket tároló gyűjtemény 219
címfordítás 249
class 35
CLI forrás 261
Clone 37
Close 90
CLR 1, 71, 79, 251, 255
CLR betöltő 162
CLR szabvány 270
CLS 165
CLSCompliant 165
CLS-megfelelés 170
CLS-megfelelő aláírás 169
CLS-megfelelő burkoló 167
CLS-megfelelő felületek 166
CLS-megfelelő szerelvény 165
code access security 255
CollectionBase 216, 220
Color 47
Color.FromKnownColor() 47
Color.FromName() 47
ColumnChanged 129
ColumnChanged eseménykezelő 221
ColumnChanging 128
COM együttműködés 251
COM objektumok 71
COM összetevők 164
CommandHandlerAttribute 228
Common Application Data 198
Common Language Runtime 1
Common Language Subsystem 165
CompareTo() 132, 143
CompareValidator 206
Conditional 23, 25, 28
Console.WriteLine 97
const 4
const kulcsszó 11, 16
ConstructorInfo.Invoke() 227
ContainsListCollection 112
ConvertEventArgs 205
CurrencyManager 199
Current 266
Customer 9, 30, 183
CustomValidator osztály 207
csak olvasható 126
csak olvasható állandó 12, 83
csak olvasható listák 226
csak olvasható tulajdonság 127
csomag 96
csomagolás 97
csonk 176
csoportos képviselő 119, 247
csupa 0 érték 51

D

DataColumn 71
DataGrid 204
DataItem 71
DataMember tulajdonság 203
DataRow 71
DataRow.RowError tulajdonság 210
DataRow.SetColumnError 210
DataSet 71, 111, 204
DataSet objektumok 218
DataTable 71
DataTable objektumok 218
DataView 72
DataViewManager 72, 128
DateTime 12
DEBUG 27, 28, 191
Debug.Assert 26
DefaultSort 135
DefaultTraceListener objektum 192
deklaráció 130
deklaratív programozás 130
derivedObject.Equals() 58
deserialize 136
Dictionary 59, 215
DictionaryBase 216
DiffGram 218
dinamikus elemek 135
dinamikus parancskezelő 231
disassembler 260
Dispose 90, 102
Dispose tagfüggvények 247
Dispose() 85, 101
Dispose(bool) 103
do 66
„dobozolás" 53, 67
DynamicCommand 228

E, É

ECMA szabvány 269
ECMA-334 269
ECMA-335 270
ECMAscript 207
egész típus 97
egészek 97
egészek véletlenszerű eloszlása 60
egydimenziós tömb 215
egyedi jellemzők 226
egyenértékűség 54
egyenlőség 52, 54, 148
egyke 78
egységbe zárás 175
egyszerű függvények 172
együttműködés 248
együttműködési API 249
együttműködési stratégiák 248
ekvivalencia 54
ekvivalencia relációk 54
eldobható objektumok 90
elemi nem változó típusok 41
elemi típus 41
elérésmódosítók 6
elkülönített tárhely 256
ellenőrzőfüggvények 207
ellipszis 155
előjel nélküli egészek 166
előre meghatározott megsemmisítés 74
elvont alaposztályok 108
elvont Clone() 151
EndEdit 226
EndValue 14
enregistration 172
Enumerator 178
Environment.SpecialFolders.-
ApplicationData 197
Environment.SpecialFolders.-
CommonApplicationData 197
Environment.SpecialFolders.-
LocalApplicationData 198
építő objektum 95
Equals() 52, 53, 148
erőforrás-kezelés 70, 71
erőforrások felszabadítása 85, 101
erőforrások szivárgása 244

erős garancia 244
erős kivételgarancia 244
erős nevek 162
erősen típusos nyilvános túlterhelés 145
ErrorProvider 209
érték szerinti átadás 36
értékadó utasítások 75
értékjelentés 55
értéktípusok 20, 22, 35, 37, 40, 52, 63, 93, 95, 127, 150, 247
érvényességi szabályok 206
érvénytelen adatok 206
események 120
események naplózása 124
eseménykezelő 187
eseményobjektum 124
Eseményparaméterek 167
EventHandlerList 125
EventLog osztály 191
explicit 156

F

Factory pattern 180
Fail 192
fájlrendszer 255
fax 181
FCL könyvtár 195
fekete ecset 93
feladatok 40
felhasználó által meghatározott típusátalakítások 17
felhasználó globális beállításai 197
felhasználói beállítások 196
felhasználók által bevitt adatok 205
felsorolások 12, 48
felsoroló osztály 179
felszabadítás 87
felszabadító kód 88
felszabadító minta 101
felszólító stílusú programozás 130
feltámasztott objektum 106
feltételes fordítás 23, 131
feltételes tagfüggvények 29
felülbírált tagfüggvények 187
felületek 99, 100, 108, 114, 127, 235
felületek megvalósítása 115
felületek paraméterként 110
felületek tagfüggvényei 116
FieldInfo 235
figyelő 192
finalize 102
finalizer 73
FinancialResults 203
Flags 50
Flatten() 155
fogadó 126
fogazott tömbök 212
for 66
fordítási idejű állandók 11, 12
fordítási idejű típus 20
foreach 66, 67, 69, 211, 214
foreach ciklus 21
Format esemény 205
formátum 33
formátumleíró 31
formátumszolgáltató 31
formázási információk 31
Framework Class Library 2
futásidejű állandók 11, 12
futásidejű átalakítás 20
futásidejű beállítások 191
futásidejű eseményparaméterek 170
futásidejű hibakeresés 190
futásidejű megfelelőség 13
FXCop 259
FxCopy 236

G, Gy

„G” formátum 33
GAC 162
Garbage Collector 71
GC.SuppressFinalize() 91
GC.SuppressFinalize(this) 102

GDI objektumok 92
GDI+ objektumok 71
generációk 74
GenericComparer 135
generics 100, 261
gépi kód 173
get 2
get hozzáférő 203
GetCustomAttributes 228
GetEnumerator 179, 217
GetEnumerator() 70
GetFolderPath 197
GetFormat() 33
GetHashCode() 58, 62, 65
GetItemProperties 220
GetList() 112
GetObjectData 140
GetParameters() 234
GetType() 22
Global Assembly Cache 162
globális beállításmódosítás 164
globális szerelvénygyorstár 162
globális változók 74
gombnyomáskezelők 190
GotDotNet 259, 260
Gyár minta 180
gyenge név 162
gyenge típusosság 236
gyűjtemények 70, 100, 210, 262

H

hagyományos típusátalakítás 20
háromdimenziós sakktábla 69
hash 41
Hashtable 59, 179
Hashtable osztály 215
HashtableEnumerator 179
hasítóértékékek 59
hasítófüggvény 60
hasítókódok 59
hasítókulcs 61
hasítótábla 41, 59
hasítótérkép 64
házirend 164
helyben kifejtés 172
helyettesíthetőség 152
helyi alkalmazásadatok 198
helyi erőforrások 163
helyi változó 74, 93, 172
helyőrző 264
hibakeresés 24, 29, 191
hibakereső utasítások 25
hidebysig 10
hívási sorrend 45
hivatkozás alapú rendszerek 127
hivatkozási jelentés 55
hivatkozási objektum 92
hivatkozási típus 22, 35, 37, 52, 63, 93, 95, 126, 130, 148, 150, 245, 247
horgok 115
hosszú switch elágazások 172
hozzáférhető adattagok 1
HybridDictionary osztály 215

I

IBindingList felület 226
ICloneable 148
ICollection 109, 217
ICollection felület 210
IComparable 114, 143, 265
IComparer 132, 143
ICustomFormatter 33
ICustomFormatter.Format() 34
IDataErrorInfo felület 222
IDeserializationCallback 137
IDictionary 217
IDisposable 70, 75, 85, 101, 102
IDisposable.Dispose() 102
IEditableObject 222
IEnumerable 70, 109
IEnumerable felület 216
IEnumerator 67, 70, 178, 266
IEnumerator.Current 22
IFormatProvider 33

IFormattable felület 30
IFormattable.ToString() 29, 31, 33
IL kód 8, 174, 260
IL visszafordító 260
IldAsm 260
IList 109, 129, 217
IList felület 220
IListSource 112
immutable types 41
imperatív programozás 130
IMsg 116
IMyInterface 37
InAttribute 250
indexelők 6, 216
Infrastructure 13
INI fájlok 195
initalizeData érték 193
initobj 76
inlining 172
InnerException tulajdonság 240
Insert() 109
interaktív rendezés 226
internetes csomagküldő 183
IntList 110
invariáns 41
Invoke 232
InvokeMethod 234
irányítás 249
is 16, 21
is-a 108
isDisposing 103
ISerializable 138, 266
IsolatedStorageFile 257
ItemType 264
iterator 213, 261
ITypedList felület 220
IUnknown 251

J

Java 36
JavaScript 205
jellemzők 130
jellemzőosztályok 228
jelölés és tömörítés 71
jelzőbitek 50
jelzőérték 102
jelzőértékként használt felsorolások 50
JIT fordítás 24, 28
JIT fordító 66, 171
jogosultságok 255
JUnit 258

K

kapcsolódási pontok 115
karakterláncok 12, 50, 94
karakterláncok irányítása 250
képviselőcélpont 247
képviselők 118, 247
kérelem sikertelensége 192
keresés 226
keretrendszer 187
késői kötés 232
kezdőérték 12, 48, 75
kezdőérték nélküli mutatók 71
kezeletlen erőforrások 101
kezeletlen kód 249
kezeletlen memória 255
kezelt C++ könyvtár 253
kezelt halom 253
kezelt környezet 255
kiadói házirendállomány 164
kicsomagolás 95
kijelentő stílusú programozás 130
kimenet részletessége 192
kimeneti adatfolyam 136
kimenő felületek 120
kis szerelvények 176
kísérőosztály 47
kivétel 237, 243
kivételek elfogása 243
kivételek kezelése 77, 79
kivételfordítás 241
kivonat alapú gyűjtemények 59
kivonat alapú tárolók 211

klón 149
kódhozzáférési biztonság 255
kommunikációs protokoll 181
konkrét felületmegvalósítás 170
konstruktor 156
konstruktorbeállítók 83
konstruktorláncolás 80
konstruktorok 75
konstruktorok kezdeti beállítása 80
korai optimalizálás 170
kör 155
körkörös hivatkozások 71
környezeti változók 28
közös algoritmusok 80
közös alkalmazásadatok 198
közös felület 114
közös nyelvi alrendszer 165
kulcs–érték párok 216
kulcsszavak 86
küldő 126

L

láncolt lista 215
láthatóság 178
len 67
Length 66
lenyíló listák 30
levélosztály 140
listamezők 30
ListDictionary osztály 215
Local Application Data 198
Logger 124
long 17
loop 68
lusta kiértékelő algoritmus 93

M

machine.config 164, 196
Mark and Compact 71
MarshalAs jellemző 250
másolás 250
másolat 96, 148, 244
Megfigyelő 128
megrendelés-feldolgozó rendszer 182
megrendelő 30
megsemmisítés 74
megsemmisítő 73, 101
megszorítások 255, 265
megvalósítások újrahasznosítása 109
mély másolatok 148
memória felszabadítása 72
memóriafoglalás 38
memóriakezelés 40, 92
memóriakezelő környezet 71
memóriaszivárgás 71
menü 227
Merge tagfüggvény 246
Message 51
MessageBox 252
metaadat 163
MFC 252
MouseDown esemény 190
MoveNext 266
MS Patterns & Practices 260
mscorlib.dll 177
MSDN webhely 260
Msg 194
MSIL 264
MSIL alapú könyvtár 253
MSIL kód 8
multicast delegate 119, 247
munkakörnyezet 198
mutator method 128
műveletek túlterhelése 167
MyAssemblyDiagnostics 194
MyResourceHog 103

N, Ny

name tulajdonság 9
NamedGroup 207
NameValueCollection 195
naplózó osztály 121
nem biztonságos kód 256
nem biztonságos összehasonlítás 145

nem memóriajellegű erőforrások 75, 101
nem sorosítható jellemzők 137
nem tiszta virtuális függvények 117
nem változó értéktípus 45, 99, 244
nem változó gyűjtemények 46
nem változó karakterlánc objektumok 95
nem változó típusok 41, 94, 127
nem virtuális tagfüggvények 156
névadási szabályok 232
névtelen tagfüggvények 261
new függvény 12
new kulcsszó 116
new módosító 156
„nincs kiváltás" garancia 244, 247
nonconst 4
no-throw 244
null 17
null hivatkozás 51, 56
NUnit 236, 258
nyelvek közti együttműködés 165
nyílt átalakítás 154
nyilvános elvont alaposztályok 181
nyilvános és védett tagfüggvények visszatérési értékei 170
nyilvános eseménybevezetés 123
nyilvános események 123
nyilvános felület 111, 175
nyilvános felületek 170
nyilvános osztály 176, 178
nyilvános változók 1
nyilvánosan bevezetett vagy megvalósított felületek 170

O, Ö

Object.Equals() 53, 54, 59
Object.GetHashCode() 41, 60
Object.operator==() 62
Object.ReferenceEquals() 52, 54, 59
Object.ToString() 30, 35
ObjectDisposed 102
objektum felszeletelése 36
objektum másolata 96
objektumazonosság 148
objektumgráf 136
objektumháló 72
objektumok azonossága 56
objektumok kezdőértéke 85
objektumok tulajdonságai 199
ObservationData 49
OK gomb 208
olvasható nézet 128
OnDeserialization 137
OnInsert() 109
OnInsertComplete() 109
OnMouseDown() 188
OnPaint() 92
OnValidating 208
op_ 167
op_add 167
op_equals 167
operator = 167
operator==() 52, 58
OR kulcsszó 28
országkönyvtár 162
országot leíró karakterlánc 163
oszlop szerinti bejárás 212
osztálygyár 233
osztályok 35, 114
osztályok kimenő felülete 126
osztálytípusok 40
OutAttribute jellemző 250
öröklés 108
összehasonlító műveletek 143
összetartás 174
összetartó összetevők 174
összetevő-szerelvények 161

P

P/Invoke 252
paraméterek 36
paraméteres többalakúság 262
paraméterezett típus 266

paraméterrel rendelkező
tulajdonságok 6
parancskezelők 228
Parse esemény 204
ParseEventArgs 204
partial kulcsszó 268
partial type 261
példány Equals() 53
példányállandók 12
Person 99
PhoneValidator 179
Planet 49
Platform Invoke 251
polimorfizmus 153
pontos típus 23
privát bináris útvonal 162
privát segédfüggvény 80
Program Files mappa 198
PropertyDescriptor 220
publisher policy file 164

Q

Queue 179, 215
QueueEnumerator 179

R

rajzolóprogram 190
RangeValidator 206
readonly állandók 12, 16
readonly kulcsszó 11
ReadOnlyCollectionBase 216
ReferenceEquals() 52
reflection 2, 54
reflexivitás 54
regiszterezés 172
RegularExpression 206
RejectChanges 218
rejtett átalakító művelet 154
Release() függvény 251
ReleaseCOMObject 251
remove 123
rendezési sorrend 131, 143
rendező relációk 143
rendszerek közti együttműködés 248
rendszererőforrások 73
rendszergazdai jogok 195
rendszerleíró adatbázis 195, 258
rendszernapló 123
rendszerobjektumok 71
rendszerszínek 47
RequiredFieldValidator 206
Reset 266
részleges másolás 36
részleges típusok 261, 268
RevenueComparer 148
role-based security 255
rosszindulatú programok 206
rotor 261
RowChanged 129
RowChanging 128
rövid függvények 172

S, Sz

saját kivételosztály 239
sakktábla 68, 213
segédeszközök 258
segédszerelvény 177
sekély másolatok 148
Serializable 136
Serialization 136
SerializationFormatter 140
serialize 136
set 2
short 17
sima fordítás 28
singleton 78
SOAP dokumentum 131
sor 215
sorosítható típusok 135
sorosító konstruktor 139
sorozatok 11
sorszámok 68
Sort 145
SortedList 216

specialname 10
SQL-befecskendező támadás 205
SqlConnection 90
Stack 215
StackTrace 26
StartValue 14
statikus állandók 12
statikus beállító utasítások 78
statikus Equals() 53
statikus konstruktorok 78
statikus tagváltozók 93
strcmp 143
string 149
string.Format() 34, 94
StringBuilder 47, 95
struct 35
StructLayoutAttribute jellemző 250
struktúrák 35, 40, 114
stub 176
System.ApplicationException 239
System.Array 99, 211, 215
System.ArrayList 178
System.Collection.Specialized 215
System.Collections 174
System.Collections.CollectionBase 109
System.Collections.Generic névtér 262
System.ComponentModel.EventHandlerList 124
System.Console.WriteLine() 30
System.Dataset 128
System.Diagnostics.Debug 26, 191
System.Diagnostics.EventLog 191
System.Diagnostics.Trace 26, 191
System.dll 28
System.Environment.GetFolderPath() 197
System.Exception 239
System.IO.File 257
System.IO.IsolatedStorage 257
System.Object 16, 53, 95, 153
System.Object hivatkozás 211, 216, 219, 263
System.Object típusú paraméterek 232
System.Object.GetHashCode() 59
System.Object.ToString() 29
System.String 94, 127
System.String.Format() 30
System.Text.StringBuilder 47
System.ValueType 48, 54
System.Web.RegularExpressions.dll 177
System.Windows.Application 198
System.Windows.Forms 174
System.Windows.Forms.Control 126, 189
System.Windows.Forms.Control.Validating 208
szabályos kifejezések 205
szabványos felszabadító eljárás 101
szabványos hibakonzol 123
szabványos összehasonlító műveletek 145
szabványos programbeállítás 195
számszerű átalakítás 153
származtatott objektum 36
származtatott osztály 102, 142
szemétgyűjtő 71, 92, 101, 215
szerelvényalkönyvtár 162
szerelvények 161
szerelvények határainak átlépése 176
szerep alapú biztonság 255
szimbólumok 29
szimmetrikusság 54
szótár 11, 215
szótár alapú gyűjtemények 216
szöveges ábrázolás 29
szövegmezők 30

T, Ty

tagjelölés 8
tagváltozó 74, 92
tárolás 38
tárolómezők 59
távelérés 181

távoli gépek közti kommunikáció 186
telefon 181
telefonszámok érvényesítése 207
telefonszámokat ellenőrző osztály 179
telepítési műveletek 175
teljes gépre vonatkozó beállítások 164
teljes objektumok 183
terepen frissíthető összetevők 163
természetes sorrend 143
this kulcsszó 7, 56
this() 83
this.GetType() 56
throw 238
thunking 249
típus 16, 108
típus nélküli hivatkozási objektum 95
típusátalakítás 16
típusbiztonság 232
típusbiztos gyűjtemények 216
típusbiztos összehasonlítás 145
típusműveletek 16
típusok hatókörének korlátozása 179
típusok láthatósága 177
típusos DataSet objektum 219
típusos tömbosztály 211
ToString() 29
többalakú paraméterek 167
többalakúság 153
többdimenziós indexelők 7, 217
többdimenziós tömbök 68, 210, 212
többszálas működés 3
többszálú programok 122
tömb hossza 67
tömbindex ellenőrzése 67
tömbindexek 213
tömbök 67, 211
tömbök méretezhetősége 214
tömböket tartalmazó tömbök 212
tömbszerű gyűjtemények 211
TRACE 28, 191
Trace.WriteLine 26
Trace.WriteLineIf() 192
TraceSwitch kapcsoló 192
tranzakciók 185, 218
tranzakciók támogatása 221
tranzitivitás 54
try 17
try/finally 85, 89
tulajdonságelérők 6
tulajdonság-hozzáférő 173
tulajdonságok 1, 112
túlterhelt összeadás 167
Type.GetConstructor() 227

U, Ü

újrahasznosítható bináris összetevők 176
újrarajzoló eljárás 93
URLInfo 114
using 85, 89
using(null) 88
üres formátumleíró 33
üres listák 220
üres tagfüggvény 24

V

változatszám 163
változó értéktípus 99
változó hivatkozási típus 46
változó karakterláncok 95
változók beállítása 77
változók regiszterezése 172
változtató tagfüggvény 128
ValueType 54
ValueType.Equals() 54
ValueType.GetHashCode() 61, 62
várakozási sor 74
védekező programozás 248
védett erőforrás 255
védett másoló konstruktor 151
véglegesítők 247
vegyes környezet 249
véletlenszerű eloszlás 60
verem 215
versenyhelyzet 122
vezérlők 30

vezérlők tulajdonságai 199
virtuális függvény 115, 187
virtuális függvények felülbírálása 115
virtuális függvények kötése 157
Visual Studio .NET 2005 261
visszahívás 118
visszatekintés 2, 54, 131, 220, 226, 232
visszatérési érték 28, 36, 119, 236
void 28
VS .NET 178
VS .NET Web Service 131

W

web alapú felületek 182
Web Forms 205
web.config 196
webes alkalmazások 201, 205
webes tagfüggvény 131
webes tagfüggvényfelületek 181
webes vezérlőkben használt szabályos kifejezések 177
WebMethod 131
webszolgáltatás 182
while 66
Windows Forms 205, 260
Windows mappa 198
Windows vezérlők 124
windowsos ablakok 128, 201
windowsos adatérvényesítés 208
WriteLine 97
WriteLineIf() 192
WSDL 131

X

XML állományok 195
XML formátum 197
XML rendező 197
XML sorosító 197
XOR 65

Y

yield kulcsszó 267