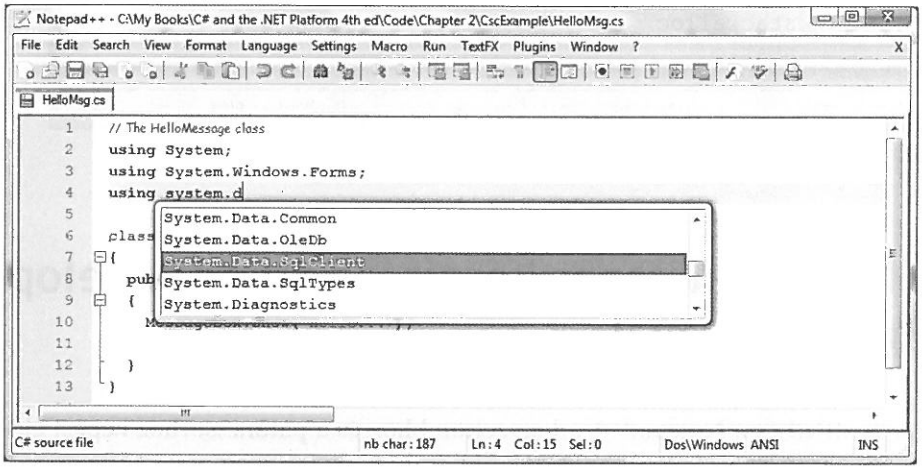
- a szöveg nagyításának és kicsinyítésének lehetősége a Ctrl + egérgörgő segítségével;
- a konfigurálható automatikus kiegészítés sok C#-kulcsszóhoz és .NET-névtérhez.

Az utolsó pont kapcsán megjegyzendő, hogy a Ctrl + szóköz billentyűkombináció aktiválja a C# automatikus kiegészítést (lásd 2.10. ábra).

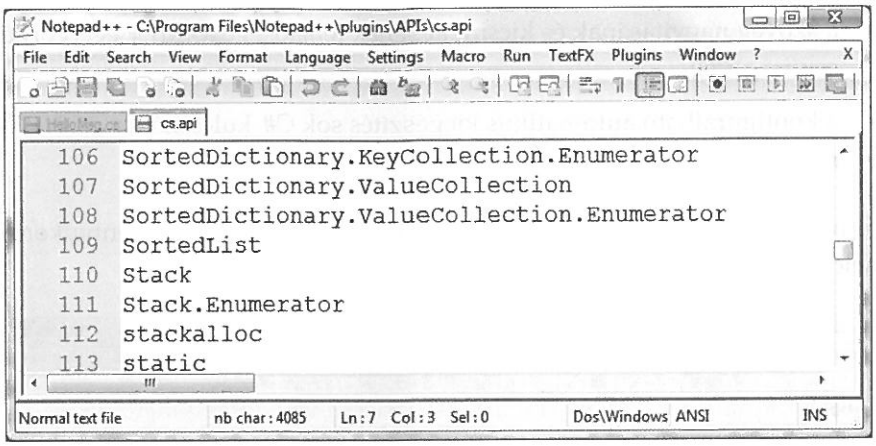


2.10. ábra: *A Notepad++ és az automatikus kiegészítés*

Az automatikus kiegészítési lista testreszabása

Az automatikuskiegészítés-ablakban látható opciók listája módosítható és bővíthető. Egyszerűen nyissuk meg a C:\Program Files\Notepad++\plugins\APIs\cs.api fájlt szerkesztésre, és írjuk be a további bejegyzéseket. Mint a 2.11. ábrán látható, minden bejegyzés külön sorban található.

A Notepad++ működése nagyon hasonlít a TextPad működéséhez. Ha több információra lenne szükség ezzel kapcsolatban, válaszuk a ? > Online Help menüpontot.



2.11. ábra: *A Notepad++ automatikus kiegészítési listájának módosítása*

.NET-alkalmazás készítése SharpDevelop használatával

A TextPad és a Notepad++ a Jegyzettömbhöz és a parancssorhoz képest előrelépést jelent a C#-kódok írásában. Ám ezek az eszközök nem biztosítanak megfelelő IntelliSense-szolgáltatásokat a C#-kódokhoz, tervezőeszközöket a grafikus felhasználói felület készítéséhez, továbbá projektsablonokat vagy adatbáziskezelő eszközöket. Az ilyen igények kielégítésére szolgál a következő .NET-fejlesztőeszköz: a SharpDevelop (vagy más néven *#Develop*).

A SharpDevelop nyílt forráskódú, sokoldalú IDE, amelynek segítségével .NET-szerelvények építhetők C# vagy VB, valamint CIL és a Boo nevű Python-inspirált .NET-nyelven. Ez az IDE teljesen ingyenes, és teljes egészében C#-ban íródott. Választhatunk, hogy a `*.csc` fájlokat letöltjük, és manuálisan fordítjuk, vagy a `setup.exe` program lefuttatásával telepítjük a SharpDevelop eszközt a fejlesztőgépünkre. Mindkét disztribúció beszerezhető a http://www.sharpdevelop.com oldalról.

A SharpDevelop számos termelékenységnövelő eszközt biztosít, és sok esetben éppoly szolgáltatásgazdag, mint a Visual Studio .NET 2008 Standard Edition. Íme a legfőbb erényeinek listája:

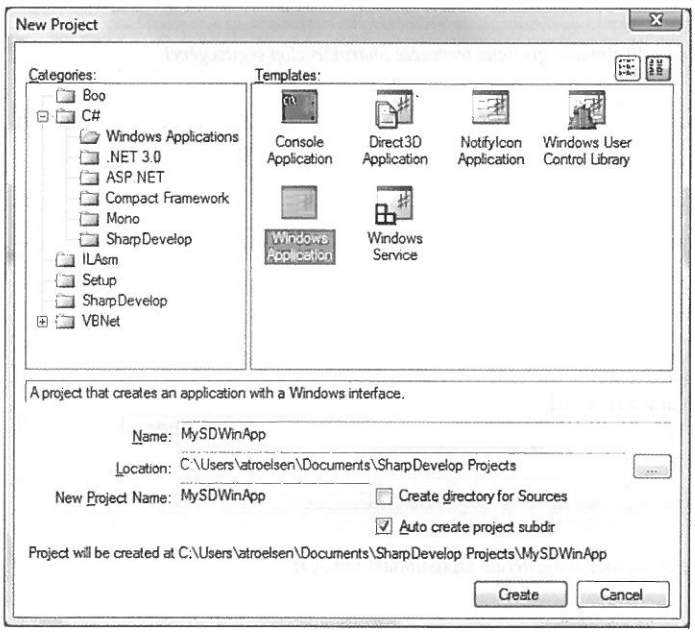
- a Microsoft és Mono C#-fordítók támogatása;
- IntelliSense kódkiegészítés, és kódrészlet-lehetőségek;

- Add Reference párbeszédpanel a külső, többek között a globális szerelvénytárba (GAC) telepített szerelvényekre való hivatkozásokhoz;
- vizuális Windows Forms-szerkesztő;
- integrált objektumkereső és kóddefiníciós szolgáltatások;
- vizuális adatbázis-tervező eszközök;
- C#-ról VB-re (és vissza) konvertáló segédprogram;
- NUnit (.NET-unit tesztelő segédprogram) és NAnt (.NET-fordító segédprogram) integrálási lehetőség;
- NET Framework 3.5 SDK dokumentációval való integrálhatóság.

Igazán szép teljesítmény egy ingyenes IDE-től. A részletek helyett nézzünk néhány érdekességet.

Egy egyszerű tesztprojekt létrehozása

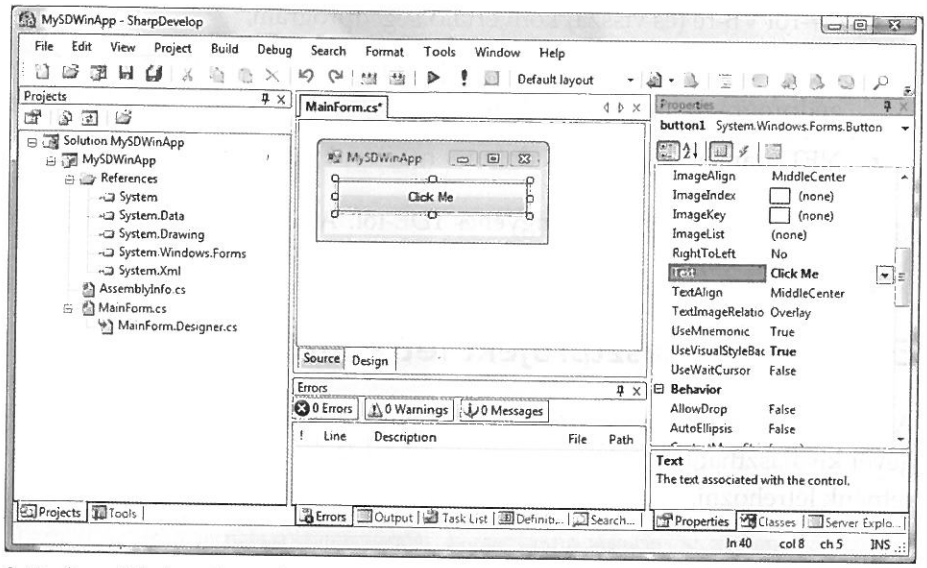
A SharpDevelop telepítése után a File > New > Solution menüpont segítségével kiválaszthatjuk, milyen típusú projektet (és melyik .NET-nyelven) szeretnénk létrehozni.



2.12. ábra: *A SharpDevelop New Project párbeszédablaka*

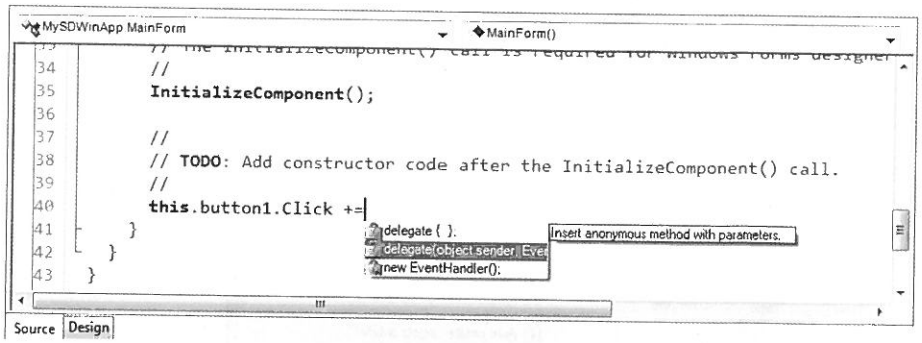
Tegyük fel például, hogy készítünk egy MySDWinApp nevű C# Windows-alkalmazást (lásd 2.12. ábra).

A Visual Studióhoz hasonlóan van grafikusfelület-tervező, eszköztár (vezérlők áthúzásához a tervezőre) és egy Properties ablak az egyes UI-elemek beállításához. A 2.13. ábra bemutatja a Button típus konfigurálását integrált fejlesztői környezet segítségével.



2.13. ábra: *Windows Forms-alkalmazás grafikus tervezése SharpDevelop segítségével*

Ha rákattintanánk az űrlaptervező aljára helyezett Source gombra, megtalálnánk a várt IntelliSense, kódkiegészítés és integrált súgó funkciókat (2.14. ábra).



2.14. ábra: *A SharpDevelop számos kódgeneráló szolgáltatást támogat*

A SharpDevelop kialakítása szerint hozza létre a Microsoft a .NET IDE legtöbb szolgáltatását (ezeket a következőkben mutatjuk be). Így a nyílt forráskódú .NET IDE funkcióit nem részletezzük. Ha további információra lenne szükség, egyszerűen használjuk a program Help menüjét.

Megjegyzés A II. kötetben lévő B függelékben találjuk a platformok közötti .NET-alkalmazások létrehozását nyílt forráskódú Mono .NET-disztribúció használatával. Ne felejtsük el, hogy a SharpDevelop Mono-tudatos.

.NET-alkalmazás készítése Visual C# 2008 Express használatával

2004 nyarán a Microsoft az integrált fejlesztői környezetek teljesen új irányzatával jelent meg, amely az „Express" termékcsalád elnevezést kapta (http://msdn.microsoft.com/express). Az Express családnak mára már nagyon sok tagja van (amelyek mindegyike *teljesen ingyenes*, a Microsoft pedig támogatja és karbantartja), többek között:

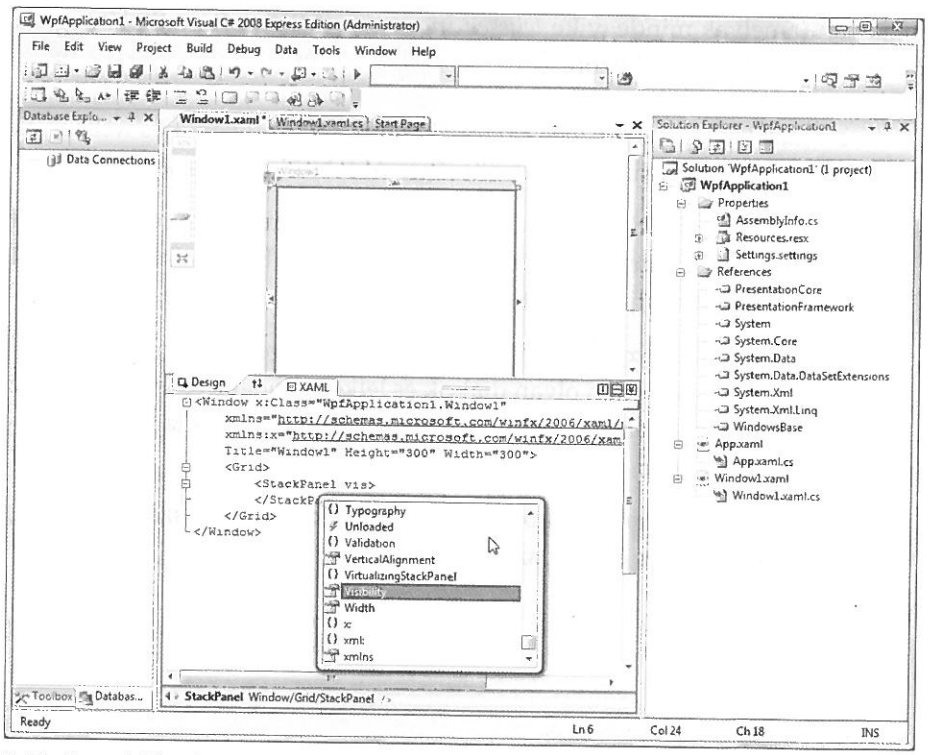
- a *Visual Web Developer 2008 Express:* könnyű eszköz dinamikus webhelyek és XML-webszolgáltatások készítéséhez ASP.NET alkalmazásával;
- a *Visual Basic 2008 Express:* áramvonalas programfejlesztő eszköz, ideális kezdő programozók számára, akik a Visual Basic felhasználóbarát szintaxisával akarják elsajátítani az alkalmazások építésének módját;
- *Visual C# 2008 Express és Visual* C++ *2008 Express:* integrált fejlesztői környezetek célzottan olyan diákok és lelkes rajongók számára, akik az általuk választott szintaxis segítségével kívánják elsajátítani a szoftverfejlesztés alapjait;
- a *SQL Server Express:* alapszintű adatbáziskezelő rendszer amatőrök, lelkes rajongók és diákok számára.

A Visual C# Express néhány jellegzetessége

Az Express termékek nagyjából a Visual Studio 2008 megfelelőinek lebutított verziói, amelyek elsősorban lelkes amatőrök és diákok számára készültek. A SharpDevelophoz hasonlóan a Visual C# 2008 Express számos objektumkereső eszközt, Windows Forms tervezőt, Add References párbeszédpanelt, IntelliSense szolgáltatásokat és kódkiegészítő sablonokat tartalmaz.

Viszont a Visual C# 2008 Express kínál néhány olyan (fontos) lehetőséget, amely jelenleg a SharpDevelopban nem érhető el, például:

- a Windows Presentation Foundation (WPF) XAML-alkalmazások támogatása;
- IntelliSense az új C# 2008 szintaktikai konstrukciókhoz, mint például a lambda-kifejezések és a LINQ-lekérdezések;
- Xbox 360 és PC-s videojátékok programozásának lehetősége az ingyen elérhető Microsoft XNA Game Studio segítségével.



2.15. ábra: *A Visual C# Express integrált módon támogatja a NET 3.0 és .NET 3.5 API-kat*

A 2.15 ábra a Visual C# Express használatát mutatja be egy WPF-projekt XAML-címkézésének előállítására.

Mivel a Visual C# 2008 Express használata nagyon hasonló a Visual Studio 2008 (és egy bizonyos mértékig a SharpDevelop) használatához, így nem vizsgáljuk meg részletesen ezt az integrált fejlesztői környezetet. Bővebb információ a termékről online, az „An Introduction to Programming Using Microsoft Visual C# 2005 Express Edition" című cikkemben olvasható a http://msdn.microsoft.com oldalon. Ez a cikk a Visual C# 2005 Express alapján íródott, ám a témák többsége változatlanul érvényes.

.NET-alkalmazás készítése Visual Studio 2008 használatával

A professzionális .NET-szoftverfejlesztők elég jó eséllyel a Microsoft vezető integrált fejlesztői környezetét, a Visual Studio 2008-at vásárolják meg a fejlesztési munkákhoz (http://msdn.microsoft.com/vstudio). Ez az eszköz közel s távol a legsokoldalúbb és a vállalati alkalmazásra leginkább kész integrált fejlesztői környezet. Természetesen ennek a minőségnek ára van, ez pedig a megvásárolt Visual Studio 2008 verziójától függ. Minden verziónak van valamilyen sajátossága.

Megjegyzés A Visual Studio 2008 családnak rengeteg tagja van. A könyv további részeiben a Visual Studio 2008 Professional verziót tekintjük alapnak integrált fejlesztői környezetként.

Az, ha nem rendelkezünk a Visual Studio 2008 Professional verziójával, nem okoz problémát, ugyanis ez a verzió *nem feltétlenül szükséges* könyvünk tanulmányozásához. A legrosszabb esetben is csupán az történhet, hogy egy olyan opciót vizsgálunk, amely a saját integrált fejlesztői környezetünkben nem áll a rendelkezésünkre. A könyvben szereplő példák tökéletesen lefordíthatók bármely más termékkel is.

Megjegyzés Ha letöltöttük a könyvhöz tartozó forráskódokat az Apress weboldal Source Code/Downloads területéről (http://www.apress.com), az aktuális példát betölthetjük a Visual Studio 2008-ba (vagy a C# 2008 Expressbe), ha kétszer rákattintunk a példa `*.sln` fájlja. Ha nem a Visual Studio 2008/C# 2008 Express programot választottuk, manuálisan kell beilleszteni az adott `*.cs` fájlt az integrált fejlesztői környezet munkaterületére.

A Visual Studio 2008 néhány jellegzetessége

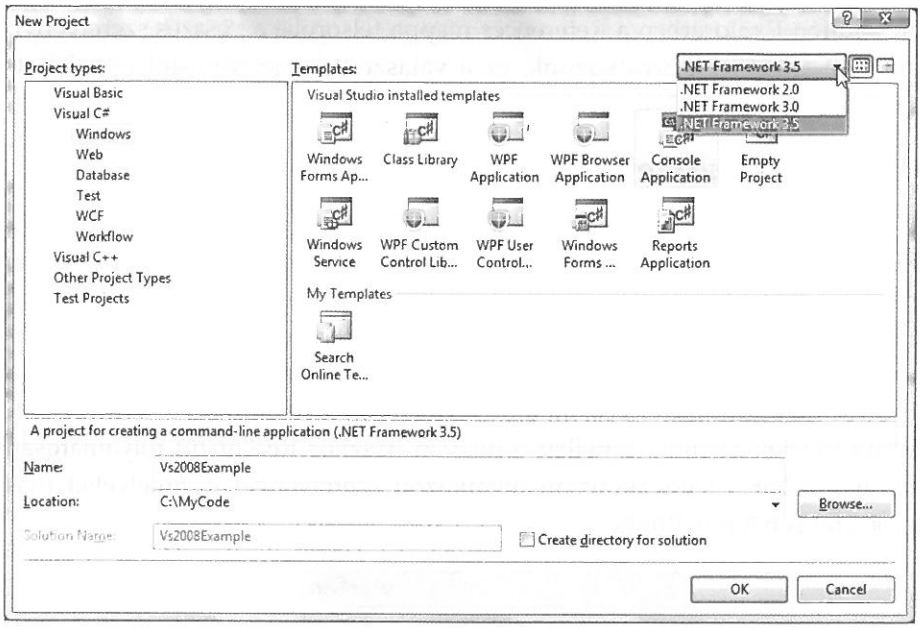
A Visual Studio 2008 az megszokott GUI-szerkesztővel, kódrészlet-támogatással, adatbáziskezelő eszközökkel, objektum- és projektböngésző szolgáltatásokkal és integrált súgórendszerrel kerül forgalomba. A megvizsgált integrált fejlesztői környezetek többségétől eltérően a Visual Studio 2008 számos kiegészítést biztosít. Íme néhány példa:

- Visual XML-szerkesztők/tervezők;
- mobileszközre (pl. Smartphone-ok és Pocket PC) történő fejlesztés támogatása;
- Microsoft Office-hoz történő fejlesztés támogatása;
- Grafikus tervezőfelület Windows Workflow Foundation projektekhez;
- kódújratervezés integrált támogatása;
- vizuális osztálytervezési eszközök;
- Object Test Bench ablak, amely lehetővé teszi objektumok létrehozását és tagjai meghívását közvetlenül az integrált fejlesztői környezetben.

A Visual Studio 2008 olyan sok szolgáltatást biztosít, hogy egy teljes kötetre (mégpedig elég nagyra) lenne szükség az integrált fejlesztői környezet minden elemének bemutatásához. *Ám ez a könyv nem az a kötet!* A következőkben csak a legjelentősebb szolgáltatásokat mutatom be, valamint később még néhány információt megtudhatunk a Visual Studio 2008 integrált fejlesztői környezetről.

A .NET-keretrendszer kiválasztása a New Project párbeszédablak segítségével

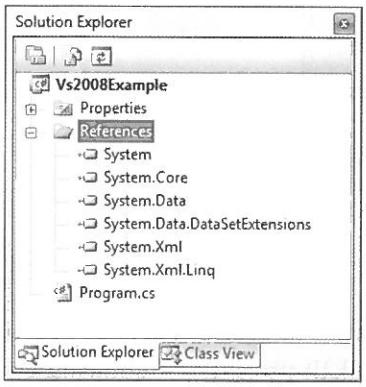
A következőkben egy új (Vs2008Example nevű) C# parancssori alkalmazást hozunk létre a File > New > Project menüelem segítségével. Ahogy a 2.16. ábrán látható, a Visual Studio 2008 (*végre*) támogatja azt a lehetőséget, hogy a New Project párbeszédablak jobb felső sarkában található legördülő listában kiválaszthassuk, mely .NET-keretrendszer verzióra (2.0, 3.0 vagy 3.5) szeretnénk építeni. A könyvben előforduló példákra meghagyhatjuk az alapértelmezett .NET-keretrendszer 3.5 verziót.



2.16. ábra: *A Visual Studio 2008 segítségével kiválaszthatjuk a .NET-keretrendszer konkrét verzióját*

A Solution Explorer használata

A (View menüből elérhető) Solution Explorer segítségével megtekinthetjük az aktuális projektben használt összes fájlt és hivatkozott szerelvényt (2.17. ábra)

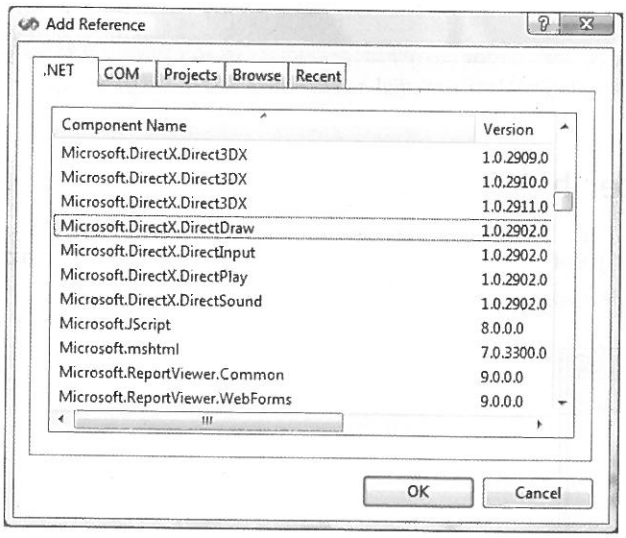


2.17. ábra: *A Solution Explorer*

A Solution Explorerben a References mappa felsorolja az összes szerelvényt, amelyre aktuálisan hivatkozunk, ez a választott projekttípustól és a keretrendszer verziójától függően változik.

Hivatkozás külső szerelvényekre

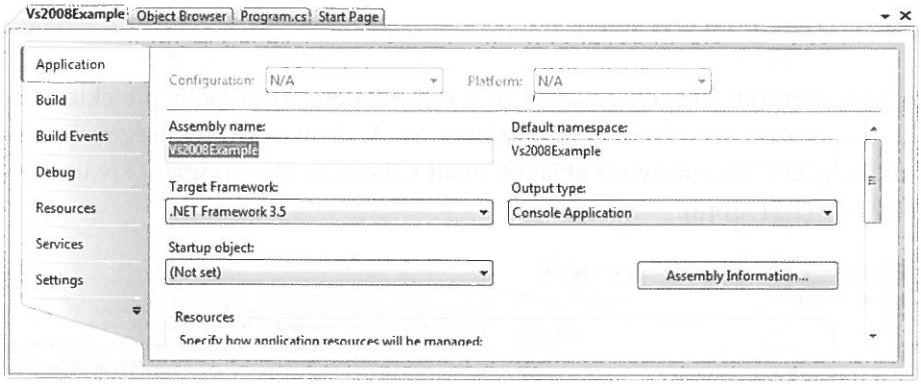
Ha további szerelvényekre kellene hivatkoznunk, jobb gombbal kattintsunk a References mappára, és válasszuk az Add Reference elemet. A megjelenő párbeszédablakban kiválaszthatjuk a szerelvényünket (lényegében a Visual Studio így biztosítja a parancssoros fordító /reference opcióját). A .NET-fülön (lásd 2.18. ábra) sok általánosan használt .NET-szerelvény jelenik meg; a Browse fül segítségével viszont a merevlemezen lévő bármelyik .NET-szerelvényt megkereshetjük. Emellett a nagyon hasznos Recent fül folyamatosan tárolja azokat a leggyakrabban hivatkozott szerelvényeket, amelyeket más projektekben használtunk.



2.18. ábra: *Az Add Reference párbeszédpanel*

Projekttulajdonságok megtekintése

Végül figyeljük meg a Solution Explorer Properties ikonját. Ha kétszer rákattintunk erre az elemre, megjelenik egy szofisztikált projektkonfigurációszerkesztő (lásd 2.19. ábra).

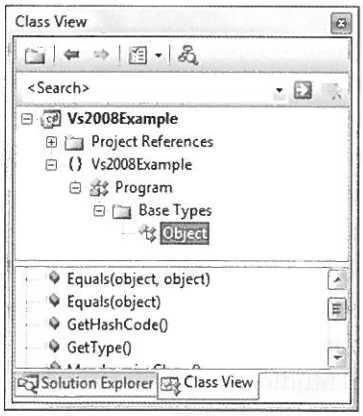


2.19. ábra: *A Project Properties ablak*

A későbbiekben több oldalról megvizsgáljuk a Project Properties ablakot. Kis utánjárással azt tapasztalhatjuk vele kapcsolatban, hogy megadhatunk számos biztonsági beállítást benne, erős nevet rendelhetünk a szerelvényünkhöz, telepíthetjük alkalmazásunkat, beilleszthetünk alkalmazás-erőforrásokat, valamint konfigurálhatunk fordítás előtti és utáni eseményeket.

A Class View

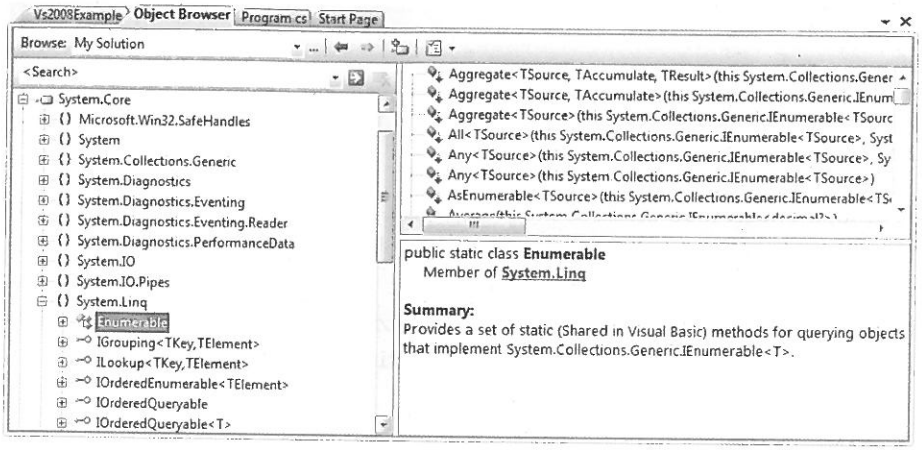
A következő vizsgált eszköz a Class View, amelyet a View menüből tölthetük be. Ennek az eszközben az a rendeltetése, hogy az aktuális projektben részt vevő típusokat (a Solution Explorer fájlalapú nézete helyett) obejektumorientált nézőpontból mutassa be. A felső panel mutatja a névterek halmazát és azok típusait, míg az alsó panel mutatja az aktuálisan kiválasztott típus tagjait (lásd 2.20. ábra).



2.20. ábra: *A Class View*

Az Object Browser

A Visual Studio 2008 rendelkezik még egy ablakkal az aktuális projekten belül hivatkozott szerelvények vizsgálatára is. A View menü segítségével nyissuk meg az Object Browser ablakot, majd válasszuk ki a vizsgálni kívánt szerelvényt (2.21. ábra).



2.21. ábra: *Az Object Browser*

Megjegyzés Ha a Solution Explorerben a referenciák könyvtárban kétszer rákattintunk egy szerelvény ikonjára, az Object Browser automatikusan megnyílik, és a kiválasztott szerelvényt kiemelve mutatja.

Integrált támogatás a kódújratervezéshez

A Visual Studio 2008 egyik legjelentősebb beépített tulajdonsága a meglévő kódok „újratervezésének" támogatása. Egyszerűen fogalmazva: az *újratervezés* olyan formális és mechanikus eljárás, amelynek segítségével a meglévő kód tökéletesíthető. A hőskorban az újratervezés tipikusan hatalmas mennyiségű manuális munkát jelentett. Szerencsére a Visual Studio 2008 nagyon sokat tesz az újratervezési eljárás automatizálására.

A Refactor menü (amely csak akkor érhető el, ha egy kódfájl aktív), a kapcsolódó gyorsbillentyűk, intelligens címkék és/vagy környezetfüggő egérkattintások segítségével minimális erőfeszítéssel hihetetlen mértékben átalakíthatók a kódok. A 2.2. táblázat definiálja a Visual Studio 2008 néhány gyakori kódújratervezését.

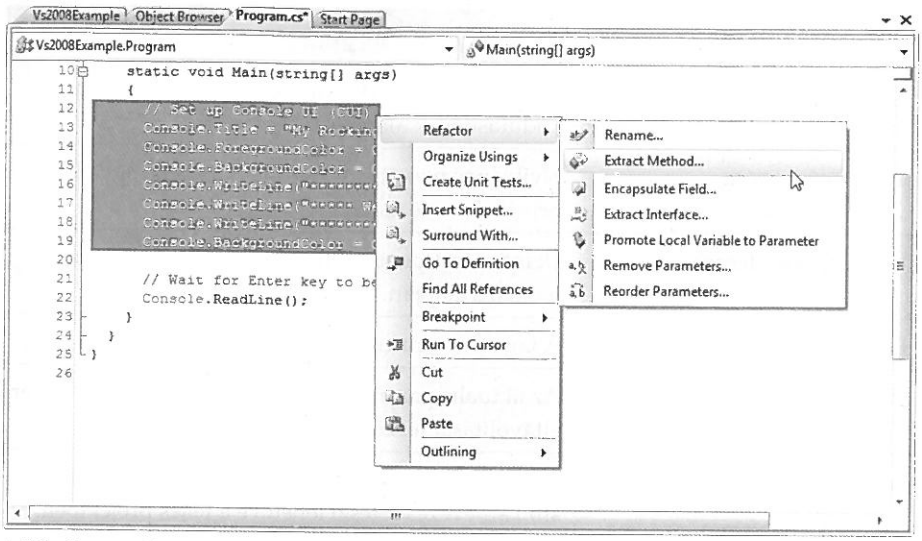
Újratervezési technika	Valós jelentése
Kiemelés metódusba	Új metódus definiálását teszi lehetővé kiválasztott kódutasítások alapján.
Mező beágyazása	Nyilvános mezőt egy C#-tulajdonságba ágyazott privát mezővé alakít.
Kiemelés interfészre	Új interfésztípust definiál a meglévő típustagok halmaza alapján.
Paraméterek átrendezése	A tagargumentumok átrendezésére biztosít módot.
Paraméterek eltávolítása	Az aktuális paraméterlistából a megadott paraméter eltávolítását teszi lehetővé.
Átnevezés	Egy kódtoken (metódusnév, mező, lokális változó stb.) átnevezését teszi lehetővé a teljes projektben.
Lokális változó előléptetése paraméterré	Egy lokális változót a definiáló metódus paraméterkészletébe helyez át.

2.2. táblázat: *Visual Studio 2008 újratervezések*

Az újratervezés működésének illusztrálására egészítsük ki a `Main()` metódusunkat a következő kóddal:

```
static void Main(string[] args)
{
  // A konzol UI (CUI) beállítása.
  Console.Title = "My Rocking App";
  Console.ForegroundColor = ConsoleColor.Yellow;
  Console.BackgroundColor = ConsoleColor.Blue;
  Console.WriteLine("**************************************");
  Console.WriteLine("***** Welcome to My Rocking App *****");
  Console.WriteLine("**************************************");
  Console.BackgroundColor = ConsoleColor.Black;

  // Várakozás az Enter enter billentyű megnyomására.
  Console.ReadLine();
}
```

Mit kell tenni akkor, ha ezt az üdvözlő üzenetet a program több különböző részén szeretnénk megjeleníteni. Ahelyett, hogy begépelnénk ugyanezt a konzolos felhasználóifelület-logikát, jobban járnánk egy ennek az elvégzésére meghívható segédfüggvény. Így a kiemelés metódusba újratervezési módszert fogjuk alkalmazni a meglévő kódunkra.



2.22. ábra: *A kódújratervezés aktiválása*

Először a szerkesztővel válasszuk ki az összes utasítást (a `Console.ReadLine()`hívását kivéve) a `Main()` metódusban. Jobb gombbal kattintsunk a kiválasztott szövegre, és válasszuk az Extract Method opciót a Refactor környezetfüggő menüből (lásd 2.22. ábra).

Az új metódusnak adjuk a `ConfigureCUI` nevet a megjelenő párbeszédablakban. Ha végeztünk, a `Main()` metódus meghívja az újonnan létrehozott `ConfigureCUI()` metódust, amely most már a korábban kijelölt kódot tartalmazza:

```
class Program
{
  static void Main(string[] args)
  {
    ConfigureCUI();
    // A bezáráshoz billentyű megnyomására vár.
    Console.ReadLine();
  }

  private static void ConfigureCUI()
  {
    // A konzol UI (CUI) beállítása.
    Console.Title = "My Rocking App";
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.BackgroundColor = ConsoleColor.Blue;
    Console.WriteLine("**************************************");
    Console.WriteLine("***** Welcome to My Rocking App *****");
    Console.WriteLine("**************************************");
    Console.BackgroundColor = ConsoleColor.Black;
  }
}
```

Ez egy egyszerű példa a Visual Studio 2008 beépített újratervezésének használatára, és további példákat is láthatunk majd erre a későbbiekben. Ha érdekel bennünket az újratervezési eljárás, és bővebb információt szeretnénk a Visual Studio 2008 által támogatott egyes újratervezésekről, olvassuk el a „Refactoring C# Code Using Visual Studio 2005" című cikket online a http://msdn.microsoft.com oldalon (ez a cikk ugyancsak a Visual Studio 2005-höz íródott, ám a Visual Studio 2008 ugyanolyan újratervezés-támogatással rendelkezik).

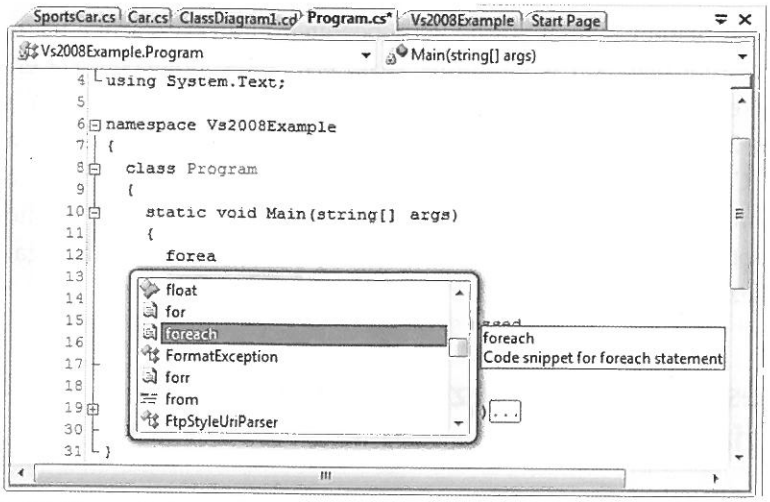
Kódbővítések és a környezetbe foglalás technológiája

A Visual Studio 2008 (és a Visual C# 2008 Express) biztosítja az előre gyártott C#-kódblokkok beszúrásának lehetőségét menükiválasztások, környezetfüggő egérkattintások és/vagy gyorsbillentyűk segítségével. Az elérhető kódbővítések száma igencsak nagy, és két nagy csoportra osztható:

1. *Kódrészletek:* ezek a sablonok gyakori kódblokkokat szúrnak be az egérkurzor helyére.
2. *Környezetbe foglalás:* ezek a sablonok a kijelölt utasításblokkokat csomagolják a releváns hatókörbe.

Ennek a funkciónak közvetlen megismeréséhez tegyük fel, hogy iterálni szeretnénk a `Main()` metódus bemeneti paraméterein egy `foreach` szerkezet használatával. Ahelyett, hogy a kódot manuálisan begépelnénk, használhatjuk a `foreach` kódrészletet. Amikor ez megtörtént, az integrált fejlesztői környezet az egérkurzor aktuális helyére beilleszti a `foreach` ciklushoz tartozó kódsablont.

Ennek kipróbálásához helyezzük az egérkurzort a `Main()` nyitó kapcsos zárójele mögé. A kódrészlet aktiválásának egyik módja, hogy jobb egérgombbal kattintunk, és kiválasztjuk az Insert Snippet (vagy Surround With) menüopciót. Itt megtaláljuk az ebbe a kategóriába tartozó összes kódrészletet (az előugró menü eltüntetéséhez nyomjuk meg az Esc billentyűt). Egyszerűsítésként mindig begépelhetjük egyszerűen a kódrészlet nevét, ez ebben az esetben „foreach". A 2.23. ábrán megfigyelhetjük, hogy a kódrészlet ikonja egy szakadt papírdarabra emlékeztet.



2.23. ábra: *A kódrészlet aktiválása*

Ha megtaláltuk azt a kódrészletet, amelyet aktiválni szeretnénk, nyomjuk meg kétszer a Tab billentyűt. Ezáltal a teljes kódrészlet automatikusan kiegészül, és egy sor *helyőrző* jelenik meg, amelyeket ki kell töltenünk a kódrészlet befejezéséhez. Ha megnyomjuk a Tab billentyűt, ciklikusan váltogathatunk az egyes helyőrzők között, és kitölthetjük az üres helyeket (a kódrészletszerkesztő módból való kilépéshez nyomjuk meg az Esc billentyűt).

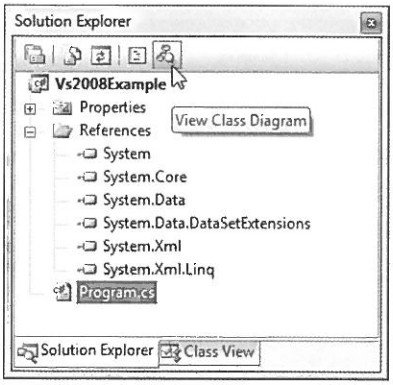
Ha a jobb gombbal való kattintás után a Surround With menüelemet választottuk volna, hasonlóan egy opciólista jelent volna meg. A Surround With kódrészletek használatakor először tipikusan kódutasításblokkot jelölünk ki, majd döntünk, hogy mibe foglaljuk azt (`try/catch` blokk stb.). Mindenképpen szánjunk időt ezeknek a kódkiegészítő sablonoknak a tanulmányozására, ugyanis nagyon nagy mértékben meg tudják gyorsítani a fejlesztési folyamatot.

Megjegyzés Minden kódkiegészítő sablon az integrált fejlesztői környezetben generálandó kód XML-alapú leírása. A Visual Studio 2008 (és a Visual C# 2008 Express) segítségével létrehozhatjuk saját egyedi kódsablonjainkat. Ennek módja elolvasható az „Investigating Code Snippet Technology" című cikkben a http://msdn. microsoft.com oldalon.

A Visual Class Designer

A Visual Studio 2008 lehetőséget biztosít az osztályok vizuális tervezésére (ez a szolgáltatás nem eleme a Visual C# 2008 Expressnek). A Class Designer lehetővé teszi a projekten belüli típusok (osztályok, interfészek, struktúrák, felsorolt típusok és metódusreferenciák) közötti kapcsolatok megtekintését és módosítását. Ezzel az eszközzel lehetőségünk van vizuálisan hozzáadni (vagy eltávolítani) tagokat a típusokhoz (vagy típusoktól), és a módosításokat érvényre juttatni a megfelelő C#-fájlokban. Ahogyan módosítjuk az adott C#-fájlt, a változások megjelennek a diagramon is.

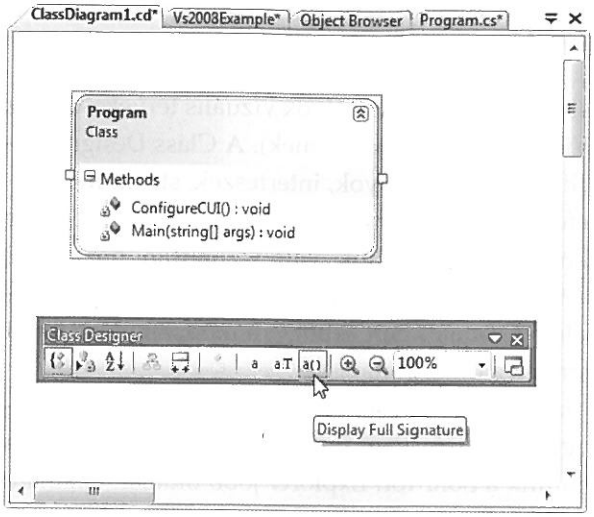
Ha a Visual Studio 2008 ezen szolgáltatását akarjuk használni, az első lépés az, hogy illesszünk be egy új osztálydiagramfájlt. Ezt többféleképpen is megtehetjük, például rákattintunk a Solution Explorer jobb oldalán található View Class Diagram gombra (2.24. ábra).



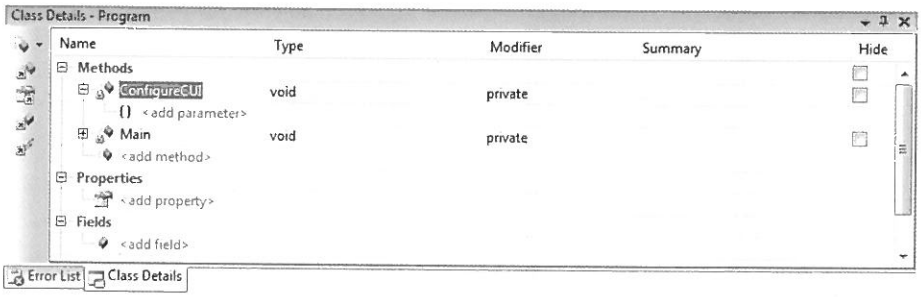
2.24. ábra: *Osztálydiagramfájl beszúrása*

Ha ez megtörtént, osztálydiagramokat látunk, amelyek az aktuális projektünkben található osztályokat jelölik. Ha egy adott típus nyíl ikonjára kattintunk, megjeleníthetjük vagy elrejthetjük a típus tagjait. A Class Designer eszköztár segítségével beállíthatjuk a tervezőfelület megjelenítési opcióit (lásd 2.25. ábra).

Ez az eszköz a Visual Studio 2008 két másik szolgáltatásával működik együtt: a Class Details ablakkal (amely a View > Other Windows menüből aktiválható) és a Class Designer eszköztárral (amely a View > Toolbox menüpontból aktiválható). A Class Details ablak nemcsak a diagramon aktuálisan kiválasztott elem részleteit mutatja, hanem segítségével módosíthatók a meglévő tagok, és menet közben új tagok is beszúrhatók (lásd 2.26. ábra).



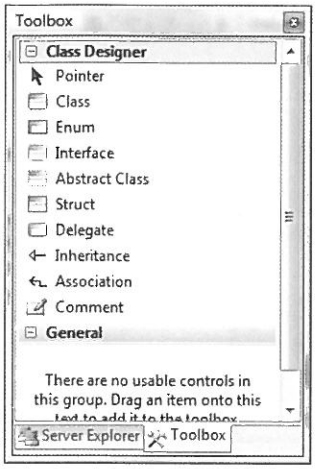
2.25. ábra: *A Class Diagram nézegető*



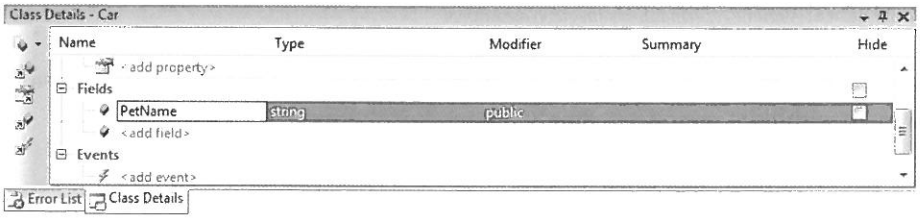
2.26. ábra: *A Class Details ablak*

A Class Designer eszköztár (lásd 2.27. ábra) lehetővé teszi új típusok vizuálisan történő beszúrását a projektünkbe (és a típusok közötti kapcsolatok megadását is). (Ne feledjük, hogy az eszköztár megtekintéséhez egy osztálydiagramnak kell az aktív ablakban lennie.) Ekkor az integrált fejlesztői környezet automatikusan létrehozza az új C#-típusdefiníciót a háttérben.

A példa kedvéért húzzunk át egy új osztályt a Class Designer eszköztárból a Class Designer ablakba. A megjelenő párbeszédablakban adjuk az osztálynak a `Car` nevet. Most a Class Details ablak segítségével adjunk hozzá egy `PetName` nevű publikus sztringmezőt (lásd 2.28. ábra).



2.27. ábra: *A Class Designer eszköztár*

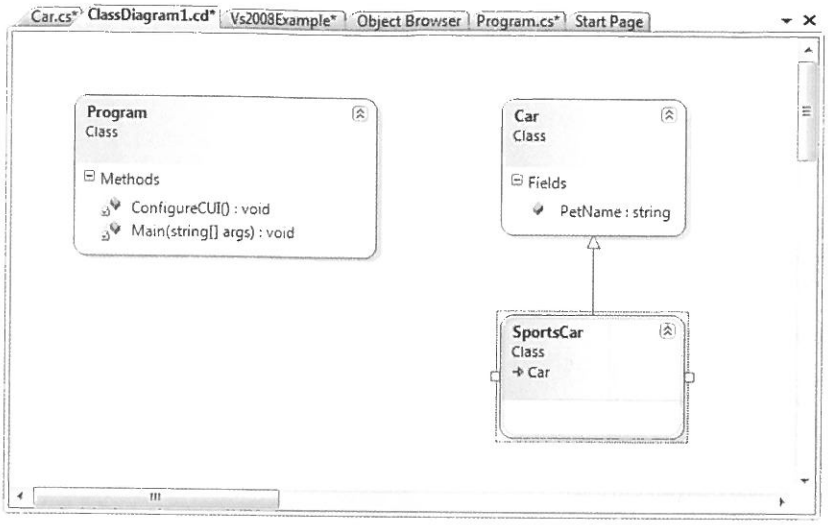


2.28. ábra: *Mező hozzáadása a Class Details ablak használatával*

Ha megnézzük a `Car` osztály C#-definícióját, láthatjuk, hogy a következőre módosult (kivéve a hozzáadott magyarázatokat):

```
public class Car
{
  // A publikus adatok használata rendszerint kerülendő;
  // de most egyszerűbbé teszik a példát.
  public string PetName;
}
```

Húzzunk egy másik új osztályt a tervezőbe, a `SportsCart`. Majd válasszuk az Inheritance ikont a Class Designer eszköztárban, és kattintsunk a `SportsCar` ikon tetejére. A bal egérgomb felengedése nélkül mozgassuk az egeret a `Car` osztályikonra, majd engedjük el az egérgombot. Ha megfelelően hajtottuk végre ezeket a lépéseket, akkor ezzel származtattuk a `SportsCar` osztályt a `Car` osztályból (lásd 2.29. ábra).



2.29. ábra: *Vizuális származtatás meglévő osztályból*

A példa befejezéséhez adjunk a generált `SportsCar` osztályhoz egy `GetPetName()` publikus metódust, amely a következőképpen néz ki:

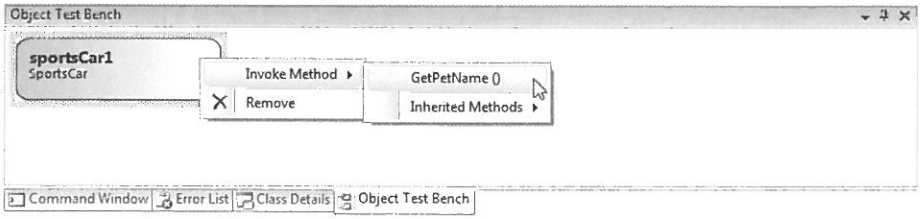
```
public class SportsCar : Car
{
  public string GetPetName()
  {
    PetName = "Fred";
    return PetName;
  }
}
```

Object Test Bench

A Visual Studio 2008 másik hasznos vizuális eszköze az Object Test Bench (OTB). Az integrált fejlesztői környezet ezen szolgáltatásának segítségével gyorsan hozhatunk létre osztálypéldányokat, valamint hívhatjuk meg tagjaikat a teljes alkalmazás futtatása nélkül. Ez kifejezetten hasznos lehet akkor, ha egy adott konkrét metódust szeretnénk tesztelni, és ehhez nem kívánunk többtucatnyi kódsoron végigmenni. Akkor is nagyon hasznos, amikor .NET-kódkönyvtárat építünk, és a működésének ellenőrzéséhez nem szeretnénk kliensalkalmazást létrehozni.

Megjegyzés Az OTB egyik korlátja az, hogy nem tudja az adott objektumok által küldött bejövő eseményeket kezelni. Ha tesztelnünk kell a bemeneti eseményeket, akkor ehhez egy különálló szerelvényt kell építenünk.

Az OTB használatához a jobb oldali gombbal kattintsunk a Class Designerrel példányosítani kívánt típusra, például a `SportsCarra`, majd a megjelenő környezetfüggő menüben válasszuk a `Create Instance > SportsCar()` elemet. Ekkor megjelenik egy párbeszédablak, amelyben elnevezhetjük az ideiglenes objektumváltozónkat (és szükség esetén megadhatjuk az esetleges konstruktorargumentumokat). Ha az eljárás befejeződött, az objektumot a rendszer az IDE-n belül hosztolja. Jobb gombbal kattintsunk az objektum ikonjára, és hívjuk meg a `GetPetName()` metódust (lásd 2.30. ábra).



2.30. ábra: *A Visual Studio 2008 Object Test Bench*

Ha megtörtént, a `Fred` érték fog megjelenni a Method Call Result párbeszédablakban. Ezt az eredményt új objektumként elmenthetjük (`System.String` típussal) az OTB-n.

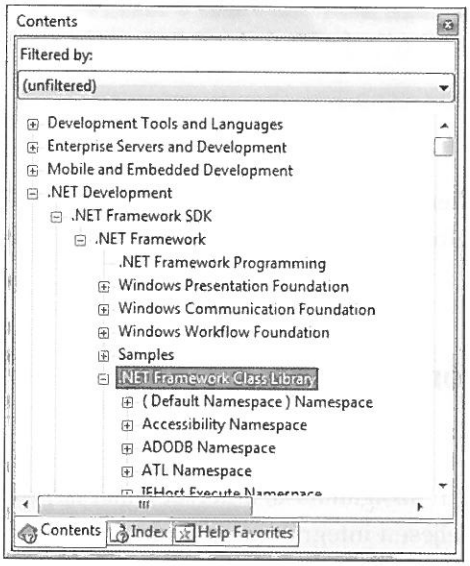
Az integrált .NET Framework 3.5 dokumentációs rendszer

Végül a Visual Studio 2008 utolsó olyan szolgáltatása, amelyet *mindenképpen* biztonsággal kell tudnunk kezelni, a teljesen integrált súgórendszer. A .NET Framework 3.5 SDK dokumentációja kifejezetten jó, nagyon olvasmányos, és telis-tele van hasznos információkkal. Az előre definiált .NET-típusok nagy száma (ez több ezerre is tehető) igencsak feladja a leckét, és le kell ásnunk az elérhető dokumentáció mélyére. Ha ezt elmulasztjuk, jelentősen megnehezítjük magunknak a .NET-fejlesztést.

A Visual Studio 2008 dinamikus súgóablakkal is rendelkezik (a Help menüből érhető el), amely az aktuálisan kiválasztott elemnek (ablak, menü, forráskód-kulcsszó stb.) megfelelően változtatja a tartalmát. Ha például a kurzort a `Console` osztály fölé helyezzük, a dinamikus súgóablakban a `System.Console` típusra vonatkozó hivatkozások csoportja jelenik meg.

Megjegyzés Egy másik kiváló gyorsbillentyű-kombináció, ha egyszer rákattintunk egy C# kulcsszóra vagy .NET-típusra (de nem jelöljük ki), majd megnyomjuk az F1 gombot. Ekkor automatikusan megnyílik a villogó szövegkurzort tartalmazó elemre vonatkozó dokumentáció.

A .NET Framework 3.5 SDK dokumentáció egyik rendkívül fontos alkönyvtárát is ismernünk kell. A dokumentáció .NET Development > .NET Framework SDK > .NET Framework> .NET Framework Class Library Reference csomópontja alatt található a .NET-alaposztálykönyvtárban olvasható minden egyes névtér teljes dokumentációja (lásd 2.31. ábra).



2.31. ábra: *A .NET-alaposztálykönyvtár hivatkozása*

A fa minden egyes csomópontja az adott névtér egy típuscsoportját, az adott típus tagjait és az adott tag paramétereit definiálja. Ezen túl, amikor egy adott típus súgóoldalait olvassuk, megtudhatjuk a kérdéses típust tartalmazó szerelvény és névtér nevét (ez az említett oldal tetején látható). A továbbiakban érdemes elmélyülten tanulmányozni ezt a *nagyon-nagyon* fontos csomópontot és a vizsgálat alatt álló entitás további tulajdonságait.

Megjegyzés Nem győzöm elégszer hangsúlyozni, mennyire fontos elsajátítani a .NET Framework 3.5 SDK dokumentáció használatát. Mindegy, milyen vastag, ám egyetlen könyv sem képes a .NET-platform minden vonatkozását teljes mértékben magába foglalni. Mindenképpen szánjunk elég időt arra, hogy el tudjunk igazodni a súgórendszerben.

További .NET-fejlesztőeszközök

A fejezet lezárásaként bemutatunk néhány olyan .NET-fejlesztőeszközt, amely kiegészíti az általunk választott integrált fejlesztői környezet szolgáltatáskészletét. Az itt bemutatott eszközök közül sok nyílt forráskódú, és mindegyikük ingyenes. Bár nem tudjuk ezeket a segédprogramokat részletesen bemutatni, a 2.3. táblázatban megtalálható néhány igen hasznos eszköz, valamint az az URL-cím, ahonnan a rájuk vonatkozó bővebb információ beszerezhető.

Eszköz	Valós jelentése	URL
FxCop	Erre mindenkinek szüksége van, akit érdekel a .NET legjobb gyakorlata. A FxCop bármilyen .NET-szerelvényt ellenőriz a hivatalos Microsoft .NET legjobb gyakorlat kódolási útmutató szerint.	`http://www.gotdotnet.com/team/fxcop`
Lutz Roeder Reflector	Ez a fejlett .NET visszafordító/objektumböngésző sok nyelven lehetővé teszi bármilyen .NET-típusú .NET-megvalósítás megtekintését.	`http://www.aisto.com/roeder/dotnet`
NAnt	A NAnt az Ant, a népszerű automatikus Java-fordítóeszköz .NET-megfelelője. Az NAnt lehetővé teszi részletes forítószkriptek definiálását és végrehajtását XML-alapú szintaxis használatával.	`http://sourceforge.net/projects/nant`
NDoc	Az NDoc olyan eszköz, amely a C#-kódokhoz (vagy lefordított .NET-szerelvényekhez) dokumentációfájlokat generál több népszerű formátumban (MSDN `*.chm`, XML, HTML, Javadoc és LaTeX).	`http://sourceforge.net/projects/ndoc`

Eszköz	Valós jelentése	URL
NUnit	Az NUnit a Java-centrikus JUnit-tesztelő eszköz .NET-megfelelője. Az NUnit segítségével tesztelhető a felügyelt kódunk.	`http://www.nunit.org`

2.3. táblázat: *.NET-fejlesztőeszközök kiválasztása*

Összefoglalás

Látható, hogy rengeteg új „játék" áll a rendelkezésünkre. A fejezet célja az volt, hogy áttekintse azokat a legfontosabb programozóeszközöket, amelyekkel egy C#-programozó a fejlesztés során találkozhat. A kalauz először azt mutatta be, hogy miként állíthatunk elő .NET-szerelvényeket csupán az ingyenes C#-fordító és a Jegyzettömb segítségével. Ezután megismerkedtünk a TextPad és a Notepad++ alkalmazásokkal, valamint ezek használatával `*.cs` kódfájlok szerkesztésére és fordítására.

Megvizsgáltunk három sokoldalú IDE-eszközt; először a nyílt forráskódú SharpDevelop, majd a Microsoft Visual C# 2008 Express és a Visual Studio 2008 Professional következett. Bár a fejezet csupán az egyes eszközök szolgáltatásainak felszínét érintette, jó alapot kínált ahhoz, hogy szabadidőnkben elmélyedjünk az általunk választott IDE tanulmányozásában (továbbá a könyv további fejezeteiben elő-előbukkannak majd a Visual Studio 2008 további szolgáltatásai). A fejezet számos nyílt forráskódú .NET-fejlesztőeszköz vizsgálatával zárult, amelyekkel bővíthető az általunk választott integrált fejlesztői környezet szolgáltatáskészlete.

2. rész

A C# alapvető építőelemei

HARMADIK FEJEZET

A C# alapvető építőelemei, I. rész

Ebben a fejezetben megkezdjük a C# programozási nyelv formális vizsgálatát. Bemutatunk számos kisebb különálló témát, amelyekre szükség van a .NET-keretrendszer tanulmányozásához. Tisztázni fogjuk, hogy hogyan építhető fel a programunk *alkalmazásobjektuma*, és hogy hogyan működik a végrehajtható fájl belépési pontja, a `Main()` metódus. Ezután megvizsgáljuk a beépített C#-adattípusokat (és a `System` névtérbeli megfelelőiket), többek között a `System.String` és `System.Text.StringBuilder` osztálytípusokat.

Miután megismertük az alapvető .NET-adattípusokat, megvizsgálunk több adattípus-konverziós technikát, többek között szűkítő és bővítő műveleteket, valamint az `unchecked` kulcsszó használatát. A fejezet lezárásaként megvizsgáljuk az érvényes C#-utasítások építésében központi szerepet játszó operátorokat, valamint az iterációs és döntési konstrukciókat.

Egy egyszerű C#-program anatómiája

A C# megköveteli, hogy minden programozási logika egy típusdefiníción belül szerepeljen (emlékezzünk vissza az 1. fejezetre: a *típus* egy olyan általános fogalom, amely a halmaz [osztály, interfész, struktúra, felsorolt típus, metódusreferencia] egy tagjára utal). Sok más nyelvtől eltérően a C#-ban nem lehet globális függvényeket vagy globális adatelemeket létrehozni. Minden adattagnak és metódusnak egy típusdefiníción belül kell szerepelnie. Kiindulásként hozzunk létre egy új C# parancssori alkalmazás projektet `SimpleCSharpApp` névvel. A kiinduló kód nem túl érdekes:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;
```

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

Ezek után módosítsuk a `Main()` metódust a következő kódutasításokkal:

```
class Program
{
  static void Main(string[] args)
  {
    // Egyszerű üzenet megjelenítése a felhasználó számára.
    Console.WriteLine("***** My First C# App *****");
    Console.WriteLine("Hello World!");
    Console.WriteLine();

    // Várakozás az Enter billentyű megnyomására a leállítás előtt.
    Console.ReadLine();
  }
}
```

Így egy olyan osztálytípus-definíciónk van, amely egyetlen `Main()` nevű metódust tartalmaz. Az alapértelmezés szerint a Visual Studio 2008 a `Main()` metódust tartalmazó osztályt „`Program`"-nak nevezi, de, ha akarjuk, tetszés szerint megváltoztathatjuk ezt a nevet. Minden végrehajtható C#-alkalmazásnak (konzolalkalmazás, Windows asztali alkalmazás vagy Windows-szolgáltatás) tartalmaznia kell egy olyan osztályt, amely a `Main()` metódust definiálja, amely tulajdonképpen az alkalmazás belépési pontját jelöli.

Hivatalosan a `Main()` metódust definiáló osztályt *alkalmazásobjektum*nak nevezzük. Bár elvileg lehetséges, hogy egyetlen végrehajtható fájl több alkalmazásobjektumot is tartalmazzon (ez egységtesztek végrehajtása során nagyon hasznos lehet), a parancssoros fordító /main kapcsolóján keresztül meg kell adni a fordítónak, hogy melyik `Main()` metódust használja belépési pontként.

Figyeljük meg, hogy a `Main()` szignatúrájában a `static` kulcsszó szerepel, amelyet részletesen az 5. fejezetben vizsgálunk majd meg. Egyelőre jegyezzünk meg annyit erről, hogy a statikus tagok hatóköre osztályszintű (nem pedig objektumszintű), és így meghívhatók anélkül, hogy előbb egy új osztálypéldányt kellene létrehoznunk.

Megjegyzés A C# kis- és nagybetűérzékeny programozási nyelv. Éppen ezért a „Main" nem ugyanaz, mint a „main", és a „Readline" sem ugyanaz, mint a „ReadLine". Jegyezzük meg tehát, hogy az összes C#-kulcsszó kisbetűs (`public`, `lock`, `class`, `global` stb.), míg a névterek, a típusok és a tagok nevei (konvenció szerint) nagy kezdőbetűvel írandók, és minden egyes beágyazott szó első betűje nagybetűs (pl. `Console.WriteLine`, `System.Windows.Forms.MessageBox`, `System.Data.SqlClient` stb.). Ökölszabály, hogy ha a fordítótól „definiálatlan szimbólumok" hibaüzenetet kapunk, ellenőrizni kell a helyesírást.

A `static` kulcsszón kívül a `Main()` metódusnak egyetlen paramétere van, és ez történetesen éppen egy sztringtömb (`string [] args`). Egyelőre ne foglalkozzunk még ennek a tömbnek a feldolgozásával. Ez a paraméter tetszőleges számú bejövő parancssori paramétert tartalmazhat (a későbbiekben látni fogjuk, hogyan érjük el őket). Végül a `Main()` metódus `void` visszatérési értékkel rendelkezik, ez azt jelenti, hogy nem határozunk meg explicit módon visszatérési értéket a `return` kulcsszóval a metódus hatóköréből való kilépés előtt.

A `Program` logikáját maga a `Main()` tartalmazza. Ebben használhatjuk a `Console` osztályt, amelyet a `System` névtér definiál. A tagjai között megtalálható a statikus `WriteLine()`, amely egy szövegsztringet és kocsivissza jelet küld az alapértelmezett kimenetre. A `Console.ReadLine()` meghívásával biztosítjuk, hogy a Visual Studio 2008 integrált fejlesztői környezet által elindított parancssor látható maradjon a hibakeresési munkamenetben az Enter billentyű lenyomásáig.

Variációk Main() metódusra

Az alapértelmezés szerint a Visual Studio 2008 void visszatérési értékkel, egyetlen sztringtömbtípusú bemeneti paraméterrel rendelkező `Main()` metódust generál. De nem csak ilyen típusú `Main()` létezhet. Az alkalmazásunk belépési pontját a következő szignatúrák bármelyikével létrehozhatjuk (feltéve, hogy egy C#-osztály vagy -struktúra definícióján belül van):

```
// int visszatérési érték, a paraméter egy sztringtömb.
static int Main(string[] args)
{
}

// Nincs visszatérési érték, nincs paraméter.
static void Main()
{
}
```

```
// int visszatérési érték, nincs paraméter.
static int Main()
{
}
```

Megjegyzés A `Main()` metódust a specifikus hozzáférés-módosító megadása nélkül feltételezett privát helyett nyilvánosként is definiálhatjuk. A Visual Studio 2008 implicit módon automatikusan privátként definiálja a program `Main()` metódusát. Ezzel biztosítható, hogy egy alkalmazás ne tudja közvetlenül meghívni más alkalmazások belépési pontját.

A `Main()` létrehozásának módja két dologtól függ. Egyrészt attól, hogy a `Main()` végrehajtása és a program lefutása után vissza akarunk-e valamilyen értéket adni. Ha igen, akkor a `void` helyett `int` típusú visszatérési értéket kell alkalmaznunk. Másrészt pedig attól, hogy fel kell-e a használók által megadott parancssori paramétereket dolgoznunk. Ha igen, akkor ezeket sztringtömb tárolja. A következőkben vizsgáljuk meg az összes lehetőségünket.

Alkalmazás-hibakód megadása

Bár a `Main()` metódusok túlnyomó többségének a visszatérési értéke `void` lesz, a `Main()` `int` típusú visszatérési értékének lehetőségével a C# konzisztens marad az egyéb C-alapú nyelvekkel. Konvencionálisan a 0 visszatérési érték azt jelzi, hogy a program sikeresen lefutott, míg az ettől eltérő értékek (pl. -1) hiba feltételre utalnak (a 0 értéket a rendszer automatikusan visszaadja akkor is, ha a `Main()` metódust `void` prototípussal hoztuk létre).

A Windows operációs rendszerben egy alkalmazás visszatérési értékét az `%ERRORLEVEL%` környezeti változó tárolja. Ha olyan alkalmazást szeretnénk készíteni, amely programozottan elindít egy másik végrehajtható fájlt (ezt a témát a 17. fejezet tárgyalja), a `%ERRORLEVEL%` értékét a `System.Diagnostics.Process.ExitCode` tulajdonság segítségével kérdezhetjük le..

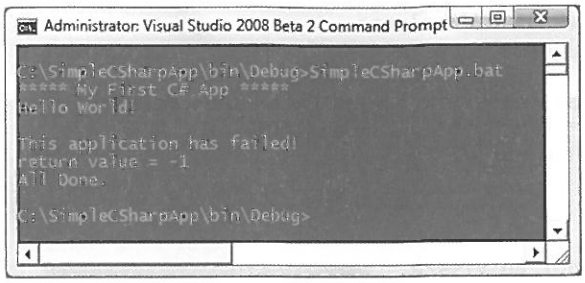
Mivel az alkalmazás visszatérési értéke akkor jut el a rendszerhez, amikor az alkalmazás már lefutott, teljesen nyilvánvaló, hogy futás közben egyetlen alkalmazás sem képes a végső hibakódját lekérni és megjeleníteni. Hogy lássuk, miként lehet ezt a hibakódot a program lefutásakor megnézni, kezdjünk hozzá a `Main()` metódusunk módosításához a következőképpen:

```
// int visszatérési érték a void helyett.
static int Main(string[] args)
{
  // Üzenet megjelenítése, és várakozás az Enter megnyomására.
  Console.WriteLine("***** My First C# App *****");
  Console.WriteLine("Hello World!");
  Console.WriteLine();
  Console.ReadLine();

  // Tetszőleges hibakódot ad vissza.
  return -1;
}
```

A `Main()` visszatérési értékét egy parancsköteg segítségével kapjuk el. A Windows Intézővel navigáljunk a lefordított alkalmazásunkat tartalmazó mappához (pl. C:\SimpleCSharpApp\bin\Debug). A Debug mappában hozzunk létre olyan új szövegfájlt (`SimpleCSharpApp.bat` néven), amely a következő utasításokat tartalmazza (eddig még nem írtunk `*.bat` fájlokat, egyelőre ne is foglalkozzunk a részletekkel; ez csupán egy teszt):

```
@echo off

rem Egy parancsköteg a SimpleCSharpApp.exe-hez,
rem ami elkapja az alkalmazás visszatérési értékét.

SimpleCSharpApp
@if "%ERRORLEVEL%" == "0" goto success

:fail
  echo Ez az alkalmazás hibás!
  echo return value = %ERRORLEVEL%
  goto end
:success
  echo Ez az alkalmazás sikeresen lefutott!
  echo return value = %ERRORLEVEL%
  goto end
:end
  echo Vége.
```

Itt nyissunk egy parancssort, és navigáljunk a végrehajtható fájlt és az új `*.bat` fájlt tartalmazó mappához (pl. megint: C:\SimpleCSharpApp\bin\Debug). Futtassuk a parancsköteget a neve begépelésével és az Enter billentyű megnyomásával. Mivel a `Main()` visszatérési értéke -1, a következő kimenetet kell látnunk (lásd 3.1. ábra). Ha a `Main()` 0-val tért volna vissza, az „Ez az alkalmazás sikeresen lefutott!" üzenet jelent volna meg a konzolon.



3.1. ábra: *Egy alkalmazás visszatérési értékének elkapása parancsköteg segítségével*

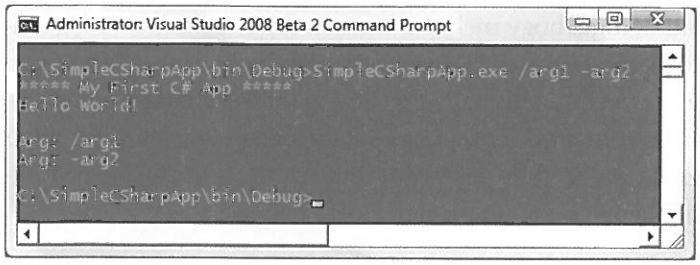
A C#-alkalmazásaink túlnyomó többségénél (ha nem is mindegyikénél) a `Main()` visszatérési értéke `void` típusú lesz, amely implicit módon 0 hibakódot ad vissza. Éppen ezért a szövegben használt `Main()` metódusok `void` visszatérési értékűek (és a többi projektnek nyilvánvalóan nem lesz szüksége parancskötegre a visszatérési kódok elkapásához).

A parancssori argumentumok feldolgozása

Most, hogy már ismerjük a `Main()` metódus visszatérési értékét, vizsgáljuk meg a sztringadatok bemeneti tömbjét. Tegyük fel, hogy az alkalmazásunkat úgy akarjuk módosítani, hogy feldolgozza a lehetséges parancssori paramétereket. Ennek egyik módja, hogy C# `for` ciklust alkalmazunk (a C# iterációs szerkezeteit a fejezet végén még megvizsgáljuk részletesebben):

```
static int Main(string[] args)
{
...
  // Bejövő paraméterek feldolgozása.
  for(int i = 0; i < args.Length; i++)
  Console.WriteLine("Arg: {0}", args[i]);

  Console.ReadLine();
  return -1;
}
```

Ekkor a `System.Array Length` tulajdonságának segítségével azt ellenőrizzük, hogy a sztringtömb mennyi elemet tartalmaz. Ahogy erről a 4. fejezetben szó lesz még, az összes C#-tömb tulajdonképpen a `System.Array` álneve, és így közös tagkészlettel rendelkeznek. Amint a ciklussal végighaladunk egy tömb összes elemén, az értéke kiíródik a konzolablakra. Az argumentumok parancssori megadása is éppen ilyen egyszerű (lásd a 3.2. ábrán).



3.2. ábra: *Argumentumok megadása parancssorban*

A `for` ciklus alternatívájaként a C# `foreach` kulcsszóval is iterálhatunk a bemeneti sztringtömb felett. Például:

```
// A "foreach" használatával nem kell ellenőrizni
// a tömb méretét.
static int Main(string[] args)
{
...
  // Bejövő paraméterek feldolgozása a foreach kulcsszóval.
  foreach(string arg in args)
    Console.WriteLine("Arg: {0}", arg);

  Console.ReadLine();
  return -1;
}
```

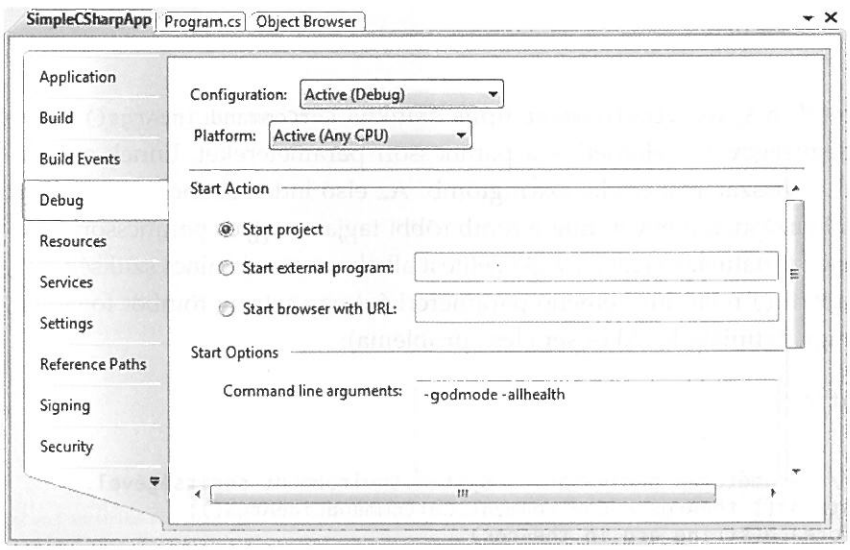
Végezetül: a `System.Environment` típus statikus `GetCommandLineArgs()` metódusa segítségével is elérhetjük a parancssori paramétereket. Ennek a metódusnak a visszatérési értéke sztringtömb. Az első index azonosítja magának az alkalmazásnak a nevét, míg a tömb többi tagja az egyes parancssori paramétereket tartalmazza (ezt a megközelítést alkalmazva már nincs szükség arra, hogy a `Main()` metódus bemenő paraméterként egy sztring tömböt fogadjon, bár ha így definiáljuk, akkor sem lesz probléma):

```
static int Main(string[] args)
{
...
  // Paraméterek beolvasása a System.Environment segítségével.
  string[] theArgs = Environment.GetCommandLineArgs();
  foreach(string arg in theArgs)
    Console.WriteLine("Arg: {0}", arg);

  Console.ReadLine();
  return -1;
}
```

Természetesen az tőlünk függ, hogy mely parancssori argumentumokra fog reagálni a programunk (ha fog egyáltalán), és hogy azokat hogyan kell formázni (például - vagy / prefixszel). Tegyük fel, hogy egy új videojátékot készítünk, és az alkalmazásunkat úgy programozzuk, hogy feldolgozzon egy `-godmode` nevű opciót. Ha a felhasználó ezzel a kapcsolóval indítja az alkalmazást, akkor nyilvánvaló, hogy tulajdonképpen *csaló*, és megtehetjük a megfelelő lépéseket.

Parancssori paraméterek megadása Visual Studio 2008-ban

A gyakorlatban a végfelhasználó adja meg az adott alkalmazás által használt parancssori argumentumokat a program elindításakor. Ugyanakkor a fejlesztési ciklusban előfordulhat, hogy tesztelési célra szeretnénk megadni lehetséges parancssori kapcsolókat. A Visual Studio 2008-ban ehhez kétszer kattintsunk a Solution Explorer Properties ikonjára, és válasszuk a bal oldalon a Debug fület. Itt a Command line arguments szövegdobozban adjuk meg az értékeket (lásd 3.3. ábra).



3.3. ábra: *Parancsori paraméterek beállítása a Visual Studio 2008-ban*

Érdekes kitérő: a System.Environment osztály néhány további tagja

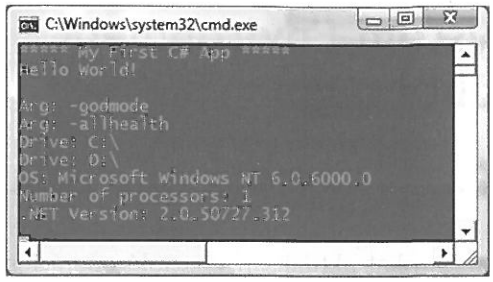
Az `Environment` típus a `GetCommandLineArgs()` mellett számos hasznos metódussal rendelkezik. Az `Environment` a különböző statikus tagok segítségével lehetővé teszi, hogy a .NET-alkalmazásunkat aktuálisan hosztoló operációs rendszer sok részletét megismerhessük. Hasznosságának bizonyítására módosítsuk a `Main()` metódusunkat úgy, hogy a `ShowEnvironmentDetails()` nevű segédmetódust hívja meg:

```
static int Main(string[] args)
{
...
  // Segédmetódus a Program osztályban.
  ShowEnvironmentDetails();

  Console.ReadLine();
  return -1;
}
```

Ezt a metódust valósítsuk meg a `Program` osztályban az `Environment` típus különböző tagjainak hívására, például:

```
static void ShowEnvironmentDetails()
{
  // A számítógép meghajtóinak és egyéb érdekes
  // adatainak kiírása.
  foreach (string drive in Environment.GetLogicalDrives())
    Console.WriteLine("Drive: {0}", drive);

  Console.WriteLine("OS: {0}", Environment.OSVersion);
  Console.WriteLine("Number of processors: {0}",
    Environment.ProcessorCount);
  Console.WriteLine(".NET Version: {0}",
    Environment.Version);
}
```

A 3.4. ábra a metódus meghívásának egy lehetséges tesztfutását mutatja (ha nem adunk meg parancssori argumentumokat a Visual Studio 2008 Debug fülén, akkor azok nem jelennek meg a konzolon).



3.4. ábra: *A rendszer környezeti változóinak megjelenítése*

Az `Environment` típus az előbbi példában bemutatottakon kívül más tagokat is definiál. A 3.1. táblázat néhány további figyelemre méltó tulajdonságot mutat be; a teljes leírás a .NET Framework 3.5 SDK dokumentációban olvasható.

Tulajdonság	Valós jelentése
`ExitCode`	A visszatérési kódot kérdezi le vagy állítja be az alkalmazáson belül bárhol.
`MachineName`	Az aktuális gép nevét kérdezi le.
`NewLine`	Az aktuális környezet új sor szimbólumát kérdezi le.
`StackTrace`	Az alkalmazás aktuális stack trace információját kérdezi le.
`SystemDirectory`	A rendszerkönyvtár teljes elérési útvonalát adja vissza.
`UserName`	Az alkalmazást elindító felhasználó nevét adja vissza.

3.1. táblázat: *A System.Environment néhány tulajdonsága*

Forráskód A SimpleCSharpApp projekt a 3. fejezet alkönyvtárában található.

A System.Console osztály

A könyv első néhány fejezetében található példaprogramok szinte kivétel nélkül erősen támaszkodnak a `System.Console` osztályra. Bár igaz, hogy a konzolos felhasználói felület (CUI) nem annyira tetszetős, mint a grafikus (GUI) vagy a webalapú front end, a korábbi példák parancssori alkalmazásokra történő korlátozása lehetővé teszi, hogy a C# szintaxisára és a .NET-platform központi vonásaira koncentráljunk a GUI-k létrehozásának bonyolultsága helyett.

Mint a neve is sugallja, a `Console` osztály konzolalapú alkalmazások bemeneti, kimeneti és hibastreamkezelését foglalja magában. A 3.2. táblázat bemutat néhány (de természetesen nem minden) érdekes tagot.

Tag	Valós jelentése
`Beep()`	Ezzel a metódussal adott gyakoriságú és hosszúságú sípoló hangjelzést ad ki a konzol.
`BackgroundColor` `ForegroundColor`	Ezek a tulajdonságok az aktuális kimenet hátterének/előterének színét állítják be. A `ConsoleColor` felsorolt típus bármelyik tagját értékül kaphatják.
`BufferHeight` `BufferWidth`	Ezek a tulajdonságok a konzol pufferterületének magasságát/szélességét szabályozzák.
`Title`	Ez a tulajdonság az aktuális konzol címét adja meg.

`WindowHeight`
`WindowWidth`
`WindowTop`
`WindowsLeft`
	Ezek a tulajdonságok a konzol méreteit szabályozzák a megadott pufferhez viszonyítva.

`Clear ()`	Ez a metódus törli a megadott puffer és a konzol megjelenítési területének tartalmát.

3.2. táblázat: *A System.Console néhány tagja*

A Console osztály alapvető bemenete és kimenete

A 3.2. táblázatban bemutatott tagokon kívül a `Console` típus a bemenet és kimenet elkapására is definiál egy olyan metóduskészletet, amelyben mindegyik metódus statikus, így elnevezésükben az osztály neve (`Console`) jelenik meg prefixumként a metódus neve előtt. Ahogy már láttuk, a `WriteLine()` egy szövegsztringet küld a kimeneti adatfolyamra (kocsivissza jellel együtt). A `Write()` metódus kocsivissza jel nélkül küldi a szöveget a kimeneti adatfolyamra. A `ReadLine()` a bemeneti adatfolyamról olvas be információt az Enter billentyű megnyomásáig, míg a `Read()` egyetlen karaktert fogad a bemeneti adatfolyamról.

A `Console` osztály alapvető I/O műveleteinek illusztrálására hozzunk létre egy új parancssorialkalmazás-projektet `BasicConsoleIO` néven, és módosítsuk a `Main()` metódusunkat úgy, hogy a `GetUserData()` nevű segédfüggvényt hívja meg:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Basic Console I/O *****");
    GetUserData();
    Console.ReadLine();
  }
}
```

Valósítsuk meg a metódust a `Program` osztályban olyan logikával, amely néhány információt kér a felhasználótól, és minden elemet megismétel a kimeneti folyamra. Megkérdezhetjük a felhasználótól például a nevét és korát (ezt az egyszerűség kedvéért szöveges értékként fogjuk kezelni az elvárható számérték helyett) a következőképpen:

```
static void GetUserData()
{
  // A név és az életkor beolvasása.
  Console.Write("Please enter your name: ");
  string userName = Console.ReadLine();
  Console.Write("Please enter your age: ");
  string userAge = Console.ReadLine();

  // A kiírás színének módosítása, pusztán szórakozásból.
  ConsoleColor prevColor = Console.ForegroundColor;
  Console.ForegroundColor = ConsoleColor.Yellow;

  // Kiírás a konzolra.
  Console.WriteLine("Hello {0}! You are {1} years old.",
  userName, userAge);

  // Az előző szín visszaállítása.
  Console.ForegroundColor = prevColor;
}
```

Az alkalmazás futtatásakor a bevitt adatok megjelennek a konzolon (egyedi színekkel).

A konzolkimenet formázása

Az első néhány fejezetben megfigyelhettük a {0}, {1} és hasonló jelölések előfordulását a különböző sztringliterálokban. A .NET-platform által használt sztringformázás módja némiképp hasonlít a C `printf()` utasítására. Egyszerűen, amikor olyan sztringliterált definiálunk, amelynek adatszegmens-

értékei csak futásidőben derülnek ki, a literálon belül kapcsos zárójeles szintaxissal megadhatunk egy helyőrzőt. Futásidőben a `Console.WriteLine()`-nak átadott paraméter vagy paraméterek váltják fel az egyes helyőrzőket.

A `WriteLine()` első paramétere olyan sztringliterált jelöl, amely {0}, {1}, {2} stb. jelzésű opcionális helyőrzőket tartalmaz. Ne feledjük, hogy a kapcsos zárójeles helyőrző első sorszáma mindig 0. A `WriteLine()` többi paramétere egyszerűen a megfelelő helyőrző helyére beillesztendő érték.

Megjegyzés Ha nem egyezik meg az egyedi számozású kapcsos zárójeles helyőrzők és a kitöltési argumentumok száma, futásidőben FormatException kivételt fogunk kapni.

Egy adott helyőrző az adott sztringben ismétlődhet is. Ha például Beatles-rajongók vagyunk és a „9, Number 9, Number 9" sztringet szeretnénk előállítani, ezt írhatjuk:

```
// John azt mondja...
Console.WriteLine("{0}, Number {0}, Number {0}", 9);
```

Azt is el kell még mondani, hogy az egyes helyőrzők a sztringliterálon belül bárhova kerülhetnek, és nem kell növekvő sorrendben állniuk. Például nézzük a következő kódrészletet:

```
// A következőket írja ki: 20, 10, 30.
Console.WriteLine("{1}, {0}, {2}", 10, 20, 30);
```

Numerikus adatok formázása

Ha a numerikus adatok kifinomultabb formázására van szükség, minden egyes helyőrző tartalmazhat opcionálisan különböző formázási karaktereket. A 3.3. táblázatban láthatók a legáltalánosabb formázási lehetőségek.

Formázási karakter sztring	Valós jelentése
`C` vagy `c`	Pénznem formázására használatos. Alapértelmezés szerint a jelző a helyi kultúrában használt szimbólum jelenik meg (dollárjel [$] USA angol esetén).
`D` vagy `d`	Tizedes törtek formázására használatos. Ez a jelző azt is megadhatja, hogy az értéket legkevesebb hány számjeggyel kell kitölteni.

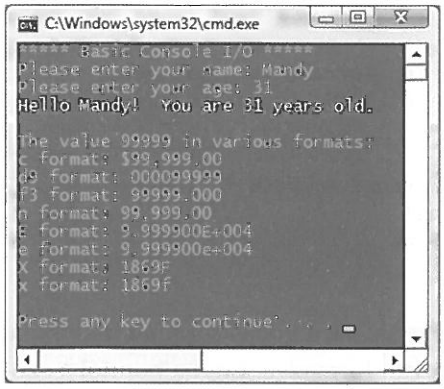
Formázási karakter sztring	Valós jelentése
E vagy e	Exponenciális ábrázolásra használatos. A megadott karaktertől függ, hogy az exponenciális konstans nagybetűs (E) vagy kisbetűs (e).
F vagy f	Fixpontos ábrázolásra használatos. Ez a jelző azt is megadhatja, hogy az értéket legkevesebb hány számjeggyel kell kitölteni.
G vagy g	Az általános esetet jelöli. Ezt a karaktert sok fixpontos vagy exponenciális formátum esetén használhatjuk.
N vagy n	Alapvető számformázásra használatos (vesszőkkel).
X vagy x	Hexadecimális formázásra használatos. Ha nagybetűs X-et használunk, a hexadecimális formátum is nagybetűs karaktereket fog tartalmazni.

3.3. táblázat: *.NET numerikus formázókarakterek*

Ezek a formázókarakterek egy adott helyőrző utótagjaként használhatók, kettőspontkapcsolóval (pl. {0:C},{1:d},{2:X} stb.). A példa kedvéért módosítsuk a `Main()` metódusunkat úgy, hogy a `FormatNumericalData()` nevű új segédfüggvényt hívja meg. Valósítsuk meg a metódust úgy, hogy egy fix értéket többféleképpen formázzon meg, például:

```
// Használjunk néhány formázási karaktert.
static void FormatNumericalData()
{
  Console.WriteLine("The value 99999 in various formats:");
  Console.WriteLine("c format: {0:c}", 99999);
  Console.WriteLine("d9 format: {0:d9}", 99999);
  Console.WriteLine("f3 format: {0:f3}", 99999);
  Console.WriteLine("n format: {0:n}", 99999);

  // A hex kis- és nagybetűs írásmódja határozza meg, hogy
  // a betűk kis- vagy nagybetűk lesznek.
  Console.WriteLine("E format: {0:E}", 99999);
  Console.WriteLine("e format: {0:e}", 99999);
  Console.WriteLine("X format: {0:X}", 99999);
  Console.WriteLine("x format: {0:x}", 99999);
}
```

A 3.5. ábrán látható az aktuális alkalmazásunk kimenete.



3.5. ábra: *Alapvető konzol I/O (.NET-sztringformázással)*

A numerikus adatok formázásának szabályozása mellett a .NET-platform további sztringliterálon belül használható tokeneket kínál a térköz és a tartalom elhelyezésének szabályozására. Ráadásul a numerikus adatokra alkalmazott tokenek más adattípusok (például felsorolást- vagy `DateTime` típus) esetén is használhatók az adatformátum szabályozására. Ne feledjük azt sem, hogy építhetünk egyedi osztályokat (vagy struktúrákat), amelyek egyedi formázási sémát definiálnak az `ICustomFormatter` interfész megvalósításával.

A későbbiekben lesznek még további formázási példák; ha azonban jobban el szeretnénk mélyedni a .NET-sztringformázás rejtelmeiben, olvassuk el a .NET Framework 3.5 SDK dokumentáció „Formatting Types" témakörét.

Forráskód A BasicConsoleIO projekt megtalálható a 3. fejezet alkönyvtárában.

Numerikus adatok formázása a parancssori alkalmazásokon túl

Mindenképpen meg kell még említeni, hogy a .NET sztringformázó karakterek használata nem csupán a parancssori alkalmazásokra korlátozódik. Ugyanez a formázási szintaxis használható a statikus `string.Format()` metódus meghívásakor is. Ez nagyon hasznosnak bizonyul akkor, amikor szöveges adatokat kell futásidőben létrehoznunk valamely alkalmazástípus (asztali grafikus alkalmazás, ASP.NET-webalkalmazás, XML-webszolgáltatás stb.) számára.

Gyors példaként tegyük fel, hogy egy grafikus asztali alkalmazást készítünk, és egy sztringet kell az üzenetdobozban való megjelenítéshez megformáznunk:

```
void DisplayMessage()
{
  // Sztringliterál formázása a string.Format() segítségével.
  string userMessage = string.Format("100000 in hex is {0:x}",
    100000);

  // Nem kell hivatkozni a System.Windows.Forms.dll szerelvényre
  // a következő kódsor fordításához!
  System.Windows.Forms.MessageBox.Show(userMessage);
}
```

Figyeljük meg, hogy a `string.Format()` egy új sztringobjektummal tér vissza, amely a megadott jelzőknek megfelelően van formázva. Ettől a ponttól a szöveges adatokat úgy használhatjuk, ahogy azt megfelelőnek látjuk.

A System-adattípusok és gyorsazonosítóik a C#-ban

Mint minden programozási nyelv, a C# is definiál egy belső adattípuskészletet, amelyeket a lokális változók, tagváltozók, visszatérési értékek és bemeneti paraméterek jelölésére használhatunk. Más programozási nyelvektől eltérően azonban ezek a kulcsszavak sokkal többek, mint csupán a fordító által felismert jelzések. A C#-adattípus kulcsszavai sokkal inkább a `System` névtér teljes értékű típusainak gyorsazonosítói. A 3.4. táblázat felsorolja az egyes rendszeradattípusokat, tartományaikat, a hozzájuk tartozó C#-kulcsszavakat és a típus CLS-kompatibilitását.

Megjegyzés Az 1. fejezetben már szó volt arról, hogy a CLS-kompatibilis .NET-kód bármely felügyelt programozási nyelvvel használható. Ha a programjainkból a CLS-nek nem megfelelő adatokat adunk át, előfordulhat, hogy más nyelvek nem fogják tudni használni.

C#-gyorsazonosító	CLS-kompatibilis?	Rendszer/Típus	Tartomány	Valós jelentés
`bool`	Igen	`System.Boolean`	Igaz vagy hamis	Igaz vagy hamis tulajdonságot jelöl
`sbyte`	Nem	`System.SByte`	-128 és 127 között	Előjeles 8 bites szám

C#-gyors-azonosító	CLS-kompatibilis?	Rendszer/ Típus	Tartomány	Valós jelentés
byte	Igen	System. Byte	0 és 255 között	Előjel nélküli 8 bites szám
short	Igen	System. Int16	-32768 és 32767 között	Előjeles 16 bites szám
ushort	Nem	System. UInt16	0 és 65535 között	Előjel nélküli 16 bites szám
int	Igen	System. Int32	-2147483648 és 2147483647 között	Előjeles 32 bites szám
uint	Nem	System. UInt32	0 és 4294967295 között	Előjel nélküli 32 bites szám
long	Igen	System. Int64	-9223372036854775808 és 9223372036854775807 között	Előjeles 64 bites szám
ulong	Nem	System. UInt64	0 és 18446744073709551615 között	Előjel nélküli 64 bites szám
char	Igen	System. Char	U+0000 és U+ffff között	Egyetlen 16 bites Unicode-karakter
float	Igen	System. Single	±1,5 X 10-45 és ±3,4X1038 között	32 bites lebegőpontos szám
double	Igen	System. Double	±5,0 X 10-324 és ±1,7 X 10308 között	64 bites lebegőpontos szám
decimal	Igen	System. Decimal	±1,0 X 10e-28 és ±7,9 X 10e28 között	Előjeles 96 bites szám
string	Igen	System. String	Korlátja a rendszermemória	Unicode-karakterek halmazát jelöli
object	Igen	System. Object	Objektumváltozóban bármilyen típust tárolhat	Az összes típus ősosztálya a .NET-univerzum

3.4. táblázat: *A C# belső adattípusai*

Megjegyzés Alapértelmezés szerint az értékadó operátor jobb oldalán található valós (`real`) számliterál `double`-ként kezelendő. Éppen ezért a `float` változó inicializálásához az `f` vagy `F` utótagot kell használni (például `5.3F`).

A numerikus típusok mindegyike (`short`, `int` stb.) adott megfelelő *struktúrára* képeződik le a `System` névtérben. A struktúrák röviden a veremben kijelölt „értéktípusok". Ugyanakkor a sztringek és objektumok „referenciatípusok", ez pedig azt jelenti, hogy a változók a felügyelt heapen vannak lefoglalva. Az érték és referenciatípusokat a 4. fejezetben fogjuk majd részletesen megvizsgálni; egyelőre annyit érdemes megjegyezni, hogy az értéktípusokat gyorsan le lehet a memóriában foglalni, valamint élettartamuk nagyon kötött és kiszámítható.

Változódeklarálás és inicializálás

Ha egy adattípust lokális változóként deklarálunk (pl. taghatókörön belüli változók), akkor ehhez megadjuk az adattípust, amelyet a változó neve követ. (Erre majd láthatunk néhány példát.) Hozzunk létre egy `BasicDataTypes` nevű parancssorialkalmazás-projektet. Módosítsuk a `Program` osztályt a következő segédmetódussal, amelyet a `Main()` metódusból hívunk meg:

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  // A lokális változókat a következőképpen deklaráljuk:
  // adatTípus változóNév;
  int myInt;
  string myString;
  Console.WriteLine();
}
```

Tudnunk kell, hogy ha egy lokális változót a kezdőérték megadása előtt használunk, akkor az *fordítási hibát* eredményez. Éppen ezért célszerű a deklarációval együtt megadni lokális adataink kezdőértékét is. Ezt megtehetjük egy sorban vagy a deklarációt és értékadást két kódutasításra bontva:

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  // A lokális változókat a következőképpen deklaráljuk és
  // inicializáljuk: adatTípus változóNév = kezdőÉrték;
  int myInt = 0;
```

```
    // A deklarálást és a hozzárendelést két sorba is írhatjuk.
    string myString;
    myString = "This is my character data";
    Console.WriteLine();
}
```

Az is megfelelő, ha ugyanabból a típusból több változót deklarálunk egyetlen kódsorban:

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  int myInt = 0;
  string myString;
  myString = "This is my character data";

  // Három logikai változó deklarációja egyetlen sorban.
  bool b1 = true, b2 = false, b3 = b1;
  Console.WriteLine();
}
```

Mivel a `bool` C#-kulcsszó is egyszerűen a `System.Boolean` struktúra rövidítése, teljes nevének használatával is lefoglalhatunk bármilyen adattípust (természetesen ugyanez igaz bármely C#-adattípuskulcsszóra is). Nézzük meg a `LocalVarDeclarations()` végleges megvalósítását:

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  // A lokális változókat a következőképpen deklaráljuk és
  // inicializáljuk: adatTípus változóNév = kezdőÉrték;
  int myInt = 0;

  string myString;
  myString = "This is my character data";

  // 3 logikai változó deklarálása egy sorban.
  bool b1 = true, b2 = false, b3 = b1;

  // A logikai változó deklarálásának roppant bőbeszédű módja.
  System.Boolean b4 = false;

  Console.WriteLine("Az adatok: {0}, {1}, {2}, {3}, {4}, {5}",
      myInt, myString, b1, b2, b3, b4);
  Console.WriteLine();
}
```

A beépített adattípusok példányosítása

Minden belső adattípus támogatja azt, amit *alapértelmezett konstruktor*nak nevezünk (lásd 5. fejezet). Röviden ez annyit jelent, hogy a `new` kulcsszó segítségével tudunk új változót létrehozni, ez pedig automatikusan az alapértelmezett értékére állítja a változót:

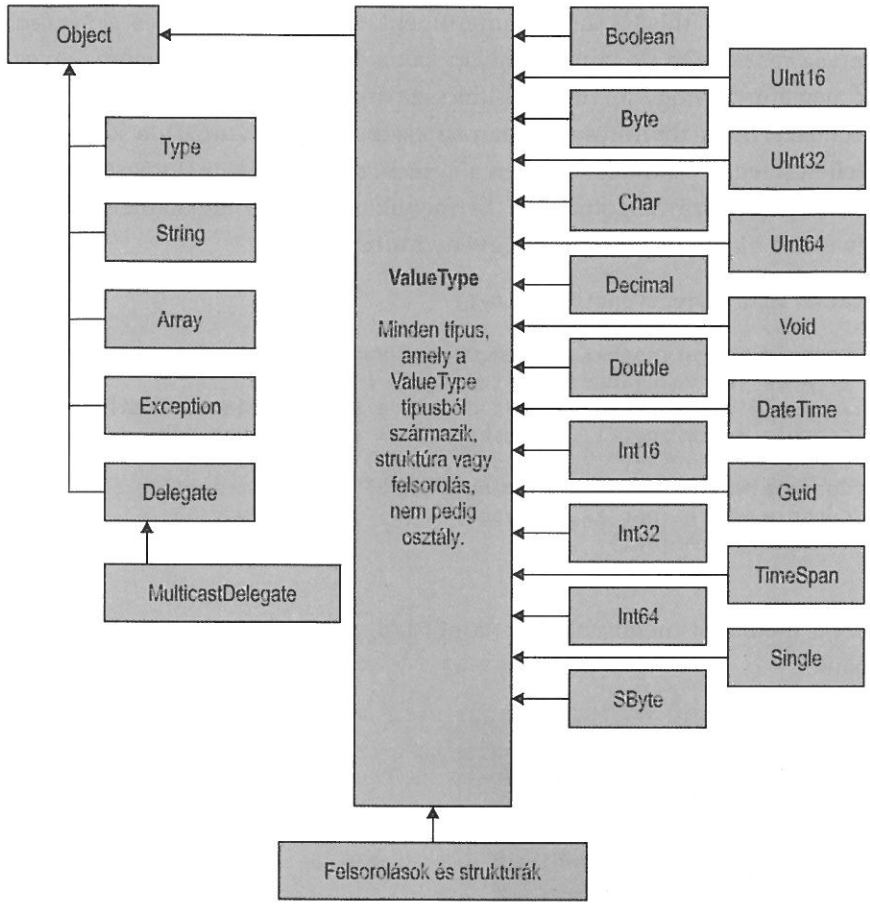
- A `bool` típusokat `false` értékre állítja.
- A numerikus adatokat 0 (vagy lebegő pontos adattípus esetén 0.0) értékre állítja.
- A `char` típusokat egyetlen üres karakterre állítja.
- A `DateTime` típusokat 1/1/0001 12:00:00 AM értékre állítja.
- Az objektumhivatkozásokat (a sztringeket is beleértve) nullra állítja.

Bár sokkal nehézkesebb a `new` kulcsszóval alaptípusú változókat létrehozni, az alábbiakban egy szintaktikailag jól formázott C#-kódot látunk:

```
static void NewingDataTypes()
{
  Console.WriteLine("=> Using new to create intrinsic data types:");
  bool b = new bool();              // Értéke: hamis.
  int i = new int();                // Értéke: 0.
  double d = new double();          // Értéke: 0.
  DateTime dt = new DateTime();     // Értéke: 1/1/0001 12:00:00 AM
  Console.WriteLine("{0}, {1}, {2}, {3}", b, i, d, dt);
  Console.WriteLine();
}
```

Az adattípusok hierarchiája

Még a primitív .NET-adattípusok is „osztályhierarchiába" rendeződnek. Ha nem ismerjük a származtatás világát, a részleteket a 6. fejezetből megtudhatjuk. Addig is röviden a lényeg: az osztályhierarchia élén szereplő típusok átadnak néhány alapvető viselkedésmintát a belőlük származtatott típusoknak. Az alapvető rendszertípusok közötti kapcsolat a 3.6. ábrán látható.



3.6. ábra: *A rendszertípusok osztályhierarchiája*

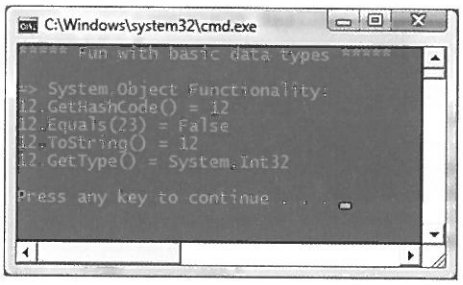
Az összes típus tulajdonképpen a `System.Object` leszármazottja, amely egy olyan metóduskészletet definiál (`ToString()`, `Equals()`, `GetHashCode()` stb.), amely a .NET-alaposztálykönyvtár minden típusában megtalálható (ezeket a metódusokat a 6. fejezet tárgyalja részletesen).

Továbbá sok numerikus adattípus a `System.ValueType` leszármazottja. A `ValueType` leszármazottai automatikusan a veremben foglalódnak le, így életciklusuk nagyon is jól kiszámítható, és meglehetősen hatékonyak. Másrészről azok a típusok, amelyek származtatási láncában nem szerepel a `System.ValueType` (például `System.Type`, `System.String`, `System.Array`, `System.Exception` és `System.Delegate`) nem a veremben foglalódnak le, hanem a szemétgyűjtött heapen.

Anélkül, hogy túlságosan belemerülnénk a `System.Object` és a `System.ValueType` részleteibe (továbbiak ugyancsak a 4. fejezetben), egyelőre jegyezzünk meg annyit, hogy mivel a C#-kulcsszavak (pl. `int`) egyszerűen a megfelelő rendszertípus rövidítései (ebben az esetben `System.Int32`), a következő egy teljesen legális szintaxis, hiszen a `System.Int32` (a C# `int`) végső soron a `System.Object` leszármazottja, így bármelyik nyilvános tagját meghívhatja, ahogy ezt az alábbi újabb segédfüggvény mutatja:

```
static void ObjectFunctionality()
{
  Console.WriteLine("=> System.Object Functionality:");
  // A C# int valójában a System.Int32 rövidítése,
  // amely a következő tagokat örökli a System.Object osztályból.
  Console.WriteLine("12.GetHashCode() = {0}", 12.GetHashCode());
  Console.WriteLine("12.Equals(23) = {0}", 12.Equals(23));
  Console.WriteLine("12.ToString() = {0}", 12.ToString());
  Console.WriteLine("12.GetType() = {0}", 12.GetType());
  Console.WriteLine();
}
```

Ha ezt a metódust meghívnánk a `Main()`-ből, a 3.7. ábrán látható kimenetet kapnánk.



3.7. ábra: *Minden típus (még a numerikus adatok is) a System.Object kiterjesztése*

A numerikus adattípusok tagjai

A C#-adattípusaival való kísérletezgetés folytatásához tudnunk kell, hogy a .NET numerikus típusai támogatják azokat a `MaxValue` és `MinValue` tulajdonságokat, amelyek az adott típus által tárolni képes tartományra vonatkozóan adnak információt. A `MinValue/MaxValue` tulajdonságokon kívül egy adott numerikusrendszer-típus definiálhat még további hasznos tagokat is. A `System.Double` típus például lehetővé teszi, hogy lekérjük az epszilon és a végtelen értékét (s ez érdekes lehet a matematikai vénával megáldottak számára). Ennek illusztrálására vizsgáljuk meg a következő segédfüggvényt:

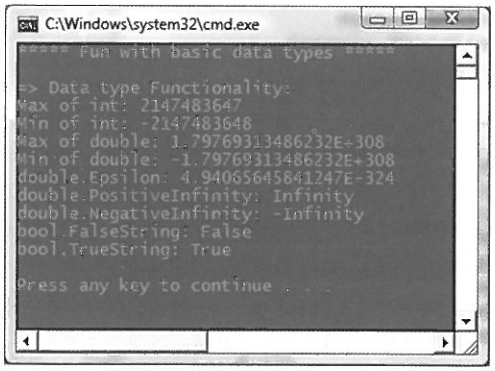
```
static void DataTypeFunctionality()
{
  Console.WriteLine("=> Data type Functionality:");
  Console.WriteLine("Max of int: {0}", int.MaxValue);
  Console.WriteLine("Min of int: {0}", int.MinValue);
  Console.WriteLine("Max of double: {0}", double.MaxValue);
  Console.WriteLine("Min of double: {0}", double.MinValue);
  Console.WriteLine("double.Epsilon: {0}", double.Epsilon);
  Console.WriteLine("double.PositiveInfinity: {0}",
    double.PositiveInfinity);
  Console.WriteLine("double.NegativeInfinity: {0}",
    double.NegativeInfinity);
  Console.WriteLine();
}
```

A System.Boolean tagjai

Következőnek nézzük meg a `System.Boolean` adattípust. Kizárólag a `{true | false}` halmaz valamelyik eleme lehet a C# `bool` érvényes értéke. Ennek tükrében teljesen nyilvánvaló, hogy a `System.Boolean` nem támogatja a `MinValue/MaxValue` tulajdonsághalmazt, hanem helyette a `TrueString/FalseString` tulajdonságokat (amely rendre a „`True`" és „`False`" értékeket adja a sztringnek). Írjuk be a következő kódutasítást a `DataTypeFunctionality()` segédmetódusba:

```
Console.WriteLine("bool.FalseString: {0}", bool.FalseString);
Console.WriteLine("bool.TrueString: {0}", bool.TrueString);s
```

A 3.8. ábra mutatja a `Main()` metódusból meghívott `DataTypeFunctionality()` kimenetét.



3.8. ábra: *Különböző adattípusok működési tartománya*

A System.Char tagjai

A C# szöveges adatokat a belső `string` és a `char` kulcsszavak jelölik, amelyek a `System.String` és a `System.Char` rövidítései, ezek pedig lényegében Unicode-karaktereket jelentenek. A sztring karakterek sorozatát jelöli (pl. „Hello"), míg a `char` egy cellát jelölhet a sztringtípusban (pl. 'H').

A `System.Char` típus többféle szolgáltatást nyújt azon túl, hogy egyetlen karakternyi adat tárolására képes. A `System.Char` statikus metódusaival eldönthetjük, hogy egy adott karakter numerikus, alfabetikus, írásjel vagy valami más egyéb. Vizsgáljuk meg a következő metódust:

```
static void CharFunctionality()
{
  Console.WriteLine("=> char type Functionality:");
  char myChar = 'a';
  Console.WriteLine("char.IsDigit('a'): {0}", char.IsDigit(myChar));
  Console.WriteLine("char.IsLetter('a'): {0}",
    char.IsLetter(myChar));
  Console.WriteLine("char.IsWhiteSpace('Hello There', 5): {0}",
    char.IsWhiteSpace("Hello There", 5));
  Console.WriteLine("char.IsWhiteSpace('Hello There', 6): {0}",
    char.IsWhiteSpace("Hello There", 6));
  Console.WriteLine("char.IsPunctuation('?'): {0}",
    char.IsPunctuation('?'));
  Console.WriteLine();
}
```

Mint a fenti kódrészletből látható, a `System.Char` tagjai konvencionálisan kétféleképpen hívhatók meg: egyedüli karakterre vagy a vizsgálandó karakter helyét jelző numerikus indexű sztringre.

Értékek értelmezése sztringadatokból

A .NET-adattípusok lehetőséget biztosítanak arra, hogy egy értéket annak sztringreprezentációjából olvassunk be. Ezt nevezzük értelmezésnek. Ez a technika akkor lehet nagyon hasznos, amikor egy felhasználói bemenő adatelemet (például grafikus felületalapú legördülő listamezőben történő választást) akarunk numerikus értékké konvertálni. Vizsgáljuk meg a következő elemzési logikát a `ParseFromStrings()` nevű metóduson belül:

```
static void ParseFromStrings()
{
  Console.WriteLine("=> Data type parsing:");
  bool b = bool.Parse("True");
  Console.WriteLine("Value of b: {0}", b);
  double d = double.Parse("99.884");
  Console.WriteLine("Value of d: {0}", d);
  int i = int.Parse("8");
  Console.WriteLine("Value of i: {0}", i);
  char c = Char.Parse("w");
  Console.WriteLine("Value of c: {0}", c);
  Console.WriteLine();
}
```

Forráskód A BasicDataTypes projekt megtalálható a 3. fejezet alkönyvtárában.

A System.String típus

A `System.String` számos, egy ilyen osztálytól elvárt metódust kínál, többek között olyanokat, amelyek a tárolt karakterek számátadják vissza, az aktuális sztring részsztringjeit keresik meg, kisbetűsről nagybetűsre és vissza váltanak, stb. A 3.5. ábra bemutat néhány (de közel sem minden) érdeklődésre számot tartó tagot.

Sztringtag	Valós jelentése
`Length`	Ez a tulajdonság az aktuális sztring hosszát adja vissza.
`Compare()`	Ez a metódus két sztringet hasonlít össze.
`Contains()`	Ez a metódus eldönti, hogy a sztring tartalmaz-e egy megadott részsztringet.
`Equals()`	Ez a metódus ellenőrzi, hogy két sztringobjektum azonos karaktereket tartalmaz-e.
`Format()`	Ez a metódus más primitívek (pl. numerikus adatok, más sztringek) és az előzőekben vizsgált {0} azonosítók segítségével formázza a sztringeket.
`Insert()`	Ez a metódus sztringet illeszt be egy adott sztringbe.
`PadLeft()` `PadRight()`	Ezeket a metódusokat arra használjuk, hogy kibéleljünk egy sztringet néhány karakterrel.

Sztringtag	Valós jelentése
`Remove()` `Replace()`	Ezekkel a metódusokkal egy sztring másolatát készíthetjük el, bizonyos módosításokkal (egyes karakterek eltávolításával vagy helyettesítésével).
`Split()`	Ez a metódus egy `String` tömböt ad vissza, amely az adott példány azon részsztringjeit tartalmazza, amelyet a megadott karakter vagy sztringtömb elemei határolnak.
`Trim()`	Ez a metódus eltávolítja a megadott karakterek minden előfordulását az aktuális sztring elejéről és végéről.
`ToUpper()` `ToLower()`	Ezek a metódusok rendre nagybetűs vagy kisbetűs formátumban készítenek másolatot az aktuális sztringről.

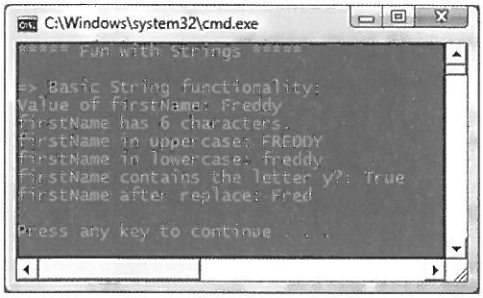
3.5. táblázat: *A System.String néhány tagja*

Egyszerű sztringkezelés

A `System.String` tagjait az elvártaknak megfelelően kell kezelni. Egyszerűen létrehozunk egy `string` adattípust és a pont operátor segítségével meghívjuk az általuk biztosított funkciókat. A `System.String` néhány tagja statikus tag, és így osztály- (és nem objektum-) szinten hívható. Tegyük fel, hogy létrehoztunk egy `FunWithStrings` nevű új parancssorialkalmazás-projektet. Valósítsuk meg a következő metódust, amelyet a `Main()`-ből hívunk meg:

```
static void BasicStringFunctionality()
{
  Console.WriteLine("=> Basic String functionality:");
  string firstName = "Freddy";
  Console.WriteLine("Value of firstName: {0}", firstName);
  Console.WriteLine("firstName has {0} characters.",
    firstName.Length);
  Console.WriteLine("firstName in uppercase: {0}",
    firstName.ToUpper());
  Console.WriteLine("firstName in lowercase: {0}",
    firstName.ToLower());
  Console.WriteLine("firstName contains the letter y?: {0}",
    firstName.Contains("y"));
  Console.WriteLine("firstName after replace: {0}",
    firstName.Replace("dy", ""));
  Console.WriteLine();
}
```

Ez a metódus egy helyi sztringváltozón hív meg különböző tagokat (`ToUpper()`, `Contains()` stb.) különböző formázások és transzformációk végrehajtására. A 3.9. ábrán látható a kezdeti kimenet.



3.9. ábra: *Egyszerű sztringkezelés*

Sztringösszefűzés

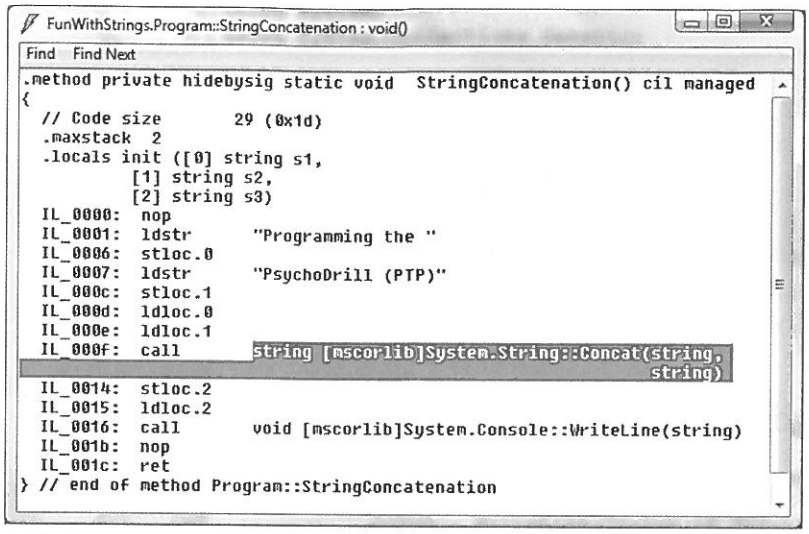
A sztringváltozók a C# + operátorával kapcsolhatók össze nagyobb sztringtípusokká. Ezt a technikát hivatalosan *sztringösszefűzésnek* nevezzük. Vizsgáljuk meg a következő új segédfüggvényt:

```
static void StringConcatenation()
{
  Console.WriteLine("=> String concatenation:");
  string s1 = "Programming the ";
  string s2 = "PsychoDrill (PTP)";
  string s3 = s1 + s2;
  Console.WriteLine(s3);
  Console.WriteLine();
}
```

A C# + szimbólumot a fordító úgy dolgozza fel, hogy meghívja a statikus `String.Concat()` metódust. Igazság szerint, ha az előző kódot lefordítanánk, és a szerelvényt megnyitnánk az `ildasm.exe` (lásd 1. fejezet) programban, a 3.10. ábrán látható köztes nyelvi kód jelenne meg.

Ennek tükrében a sztringösszefűzést végrehajthatjuk közvetlenül a `String.Concat()` meghívásával is (bár ezzel semmit nem nyerünk – csak annyit érünk el, hogy több billentyűt kell leütnünk):

```
static void StringConcatenation()
{
  Console.WriteLine("=> String concatenation:");
  string s1 = "Programming the ";
  string s2 = "PsychoDrill (PTP)";
  string s3 = String.Concat(s1, s2);
  Console.WriteLine(s3);
  Console.WriteLine();
}
```



3.10. ábra: *A C# + operátora a String.Concat() meghívását eredményezi.*

Vezérlőkarakterek

Mint más C-alapú nyelvekben is, a C#-sztringliterálok is tartalmazhatnak különböző *vezérlőkarakter*eket, amelyek megadják, hogy a karakteradatokat hogyan kell kiírni a kimeneti folyamra. Minden vezérlőkarakter egy fordított perjellel kezdődik, amelyet egy speciális token követ. A 3.6. táblázat felsorolja a legáltalánosabb opciókat.

Karakter	Valós jelentése
\'	Egy aposztrófot illeszt be a sztringliterálba.
\"	Egy idézőjelet illeszt be a sztringliterálba.
\\	Egy fordított perjelet illeszt be a sztringliterálba. Ez az elérési útvonalak definiálásakor nagyon hasznos.
\a	Rendszerfigyelmeztetést (sípolást) vált ki. Parancssori programok esetén ez lehet egy audiokulcs a felhasználó számára.
\n	Új sort illeszt be (Win32 platformokon).
\r	Kocsivissza jelet illeszt be.
\t	Vízszintes tabulátort illeszt be a sztringliterálba.

3.6. táblázat: *Sztringtípusú vezérlőkarakterek*

Egy olyan sztring kiíratásához például, amely a szavak között tabulátorjelet tartalmaz, a \t vezérlőkaraktert kell használnunk. Vagy tegyük fel, hogy olyan sztringliterálokat szeretnénk létrehozni, amelyek közül az egyik idézőjeleket tartalmaz, egy másik könyvtárelérési útvonalat definiál, a harmadik pedig három üres sort szúr be a karakteradatok kiírása után. Ha ezt fordítási hiba nélkül szeretnénk végrehajtani, a \", \\ és \n vezérlőkaraktereket kell használnunk. Ráadásul, csak hogy 10 méteres körzeten belül mindenkit idegesítsünk, minden sztringliterálba hangjelzést (sípoló hang kiváltását) ágyaztam be. Vizsgáljuk meg a következőket:

```
static void EscapeChars()
{
  Console.WriteLine("=> Escape characters:\a");
  string strWithTabs = "Model\tColor\tSpeed\tPet Name\a ";
  Console.WriteLine(strWithTabs);

  Console.WriteLine("Everyone loves \"Hello World\"\a ");
  Console.WriteLine("C:\\MyApp\\bin\\Debug\a ");

  // Összesen 4 üres sort illeszt be (majd ismét sípol!).
  Console.WriteLine("All finished.\n\n\n\a ");
  Console.WriteLine();
}
```

Szó szerinti sztringek definiálása

Ha egy sztringliterál prefixeként a @ szimbólumot használjuk, az így előállított elemet *szó szerinti sztring*nek nevezzük. A szó szerinti sztringek használatával kikapcsoljuk a literál vezérlőkaraktereinek feldolgozását, és úgy íratjuk ki a sztringet, ahogy az van. Ez akkor a leghasznosabb, amikor könyvtár- vagy hálózati elérési útvonalakat jelölő sztringekkel dolgozunk. Ilyenkor a \\ vezérlőkarakterek helyett egyszerűen a következőt írhatjuk:

```
// A következő sztringet szó szerint írjuk ki,
// így az összes vezérlőkarakter megjelenik.
Console.WriteLine(@"C:\MyApp\bin\Debug");
```

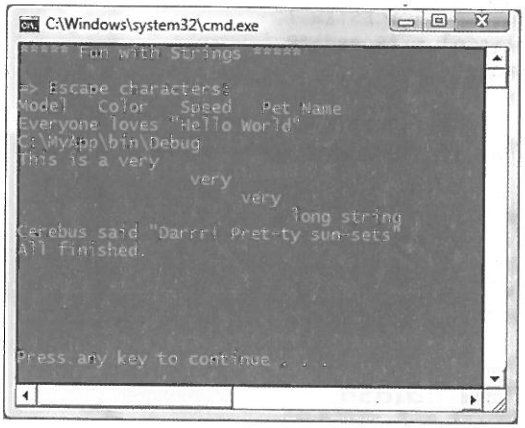
A szó szerinti sztringek a definíciójában található szóközöket, tabulátorokat és újsor-karaktereket is megőrzik.

```
// A szó szerinti sztringek megőrzik a szóközöket.
string myLongString = @"This is a very
    very
        very
            long string";
Console.WriteLine(myLongString);
```

A szó szerinti sztringek használatával közvetlenül tudunk dupla idézőjelet is írni a " token megismétlésével. Például:

```
Console.WriteLine(@"Cerebus said ""Darrr! Pret-ty sun-sets""");
```

A 3.11. ábra mutatja az `EscapeChars()` meghívásának eredményét.



3.11. ábra: *Vezérlőkarakterek és szó szerinti sztringek használat közben*

A sztringek és az egyenlőség

A *referenciatípus* a szemétgyűjtött felügyelt heapen lefoglalt objektum (erről a 4. fejezetben lesz részletesen szó). Az alapértelmezés szerint, amikor a referenciatípusok egyenlőségét vizsgáljuk (a C# == és != operátorai segítségével), akkor kapunk vissza igaz értéket, ha a hivatkozások ugyanarra az objektumra mutatnak a memóriában. Ugyanakkor, annak ellenére, hogy a sztringadattípus referenciatípus, az egyenlőségvizsgáló operátorok új definíciót kaptak azért, hogy a sztringobjektumok *értékeit* vizsgálják, ne pedig azt, hogy a memóriában ugyanarra az objektimra mutatnak-e:

```
static void StringEquality()
{
  Console.WriteLine("=> String equality:");
  string s1 = "Hello!";
  string s2 = "Yo!";
  Console.WriteLine("s1 = {0}", s1);
  Console.WriteLine("s2 = {0}", s2);
  Console.WriteLine();

  // A sztringek egyezésének tesztelése.
  Console.WriteLine("s1 == s2: {0}", s1 == s2);
  Console.WriteLine("s1 == Hello!: {0}", s1 == "Hello!");
  Console.WriteLine("s1 == HELLO!: {0}", s1 == "HELLO!");
  Console.WriteLine("s1 == hello!: {0}", s1 == "hello!");
  Console.WriteLine("s1.Equals(s2): {0}", s1.Equals(s2));
  Console.WriteLine("Yo.Equals(s2): {0}", "Yo!".Equals(s2));
  Console.WriteLine();
}
```

Figyeljük meg, hogy a C# egyenlőségvizsgáló operátorok kis-/nagybetű érzékeny, karakterről karakterre történő egyezésvizsgálatot hajtanak végre, így a „Hello!" nem egyenlő a „HELLO!" sztringgel, amely más, mint a „hello!". Ugyanakkor a sztring és a `System.String` közötti kapcsolatot figyelembe véve, egyenlőséget éppúgy tudtuk a `String Equals()` metódusával is vizsgálni, mint a beépített egyenlőségvizsgáló operátorokkal. Végül, mivel minden sztringliterál (mint pl. a „Yo") érvényes `System.String`-példány, a sztringcentrikus funkcionalitások elérhetők a karakterek egy fix sorozatából.

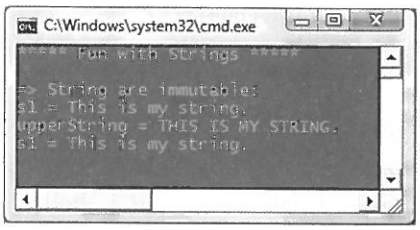
A sztring megváltoztathatatlan

A `System.String` egyik érdekes vonása, hogy ha egyszer egy sztringobjektumnak megadtuk a kezdőértékét, a karakteradatokat már *nem lehet megváltoztatni*. Első pillantásra ez képtelenségnek tűnhet, hiszen a sztringeknek újra és újra más értékeket adunk, és a `System.String` típus is számos olyan metódust definiál, amely látszólag valamilyen úton-módon a sztringeket módosítja (kisbetűs, nagybetűs stb.). Ugyanakkor, ha közelebbről megvizsgáljuk, hogy mi is zajlik a háttérben, egyértelmű, hogy a sztringtípus metódusai tulajdonképpen egy teljesen új, a módosításnak megfelelő formátumú sztringobjektummal térnek vissza.

```
static void StringAreImmutable()
{
  // A sztring kezdőértékének beállítása.
  string s1 = "This is my string.";
  Console.WriteLine("s1 = {0}", s1);
```

```
    // Az s1 nagybetüs?
    string upperString = s1.ToUpper();
    Console.WriteLine("upperString = {0}", upperString);

    // Nem! Az s1 formátuma megegyezik!
    Console.WriteLine("s1 = {0}", s1);
}
```

Ha megvizsgáljuk a 3.12. ábrán látható kimenetet, meggyőződhetünk róla, hogy az eredeti sztringobjektum (s1) nem vált nagybetűssé a `ToUpper()` meghívásának a hatására, hanem a sztring egy *másolatát* kaptuk vissza módosított formátumban.



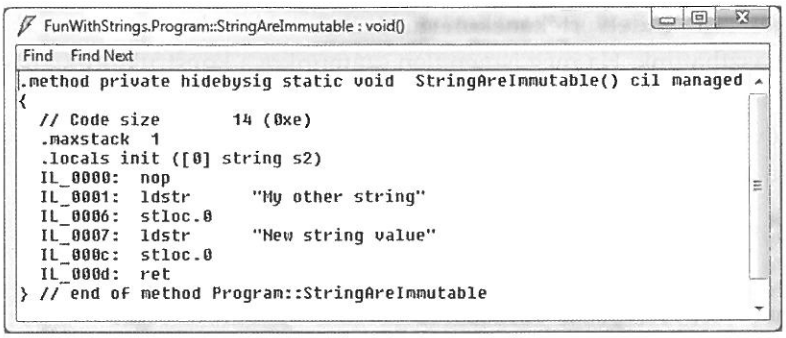
3.12. ábra: *A sztring örök*

Ugyanez a megváltoztathatatlanság érvényes akkor is, amikor a C#-értékadásoperátort használjuk. Ennek bizonyítására tegyük megjegyzések közé (vagy töröljük) a létező kódokat a `StringAreImmutable()` metódusban (a létrehozott köztes nyelvi kód mennyiségének csökkentésére), és írjuk be a következő kódutasítást:

```
static void StringAreImmutable()
{
  string s2 = "My other string";
  s2 = "New string value";
}
```

Majd fordítsuk le az alkalmazást, és töltsük be a szerelvényt az `ildasm.exe` programba (ismét: lásd 1. fejezet). Ha duplán rákattintunk a `Main()` metódusra, a 3.13. ábrán látható köztes nyelvi kódot kapjuk.

Bár még nem vizsgáltuk meg a köztes nyelv (CIL) alacsony szintű részleteit, figyeljük meg, hogy a `Main()` metódus sokszor meghívja az `ldstr` (sztring betöltése) vezérlőkódot. Leegyszerűsítve a köztes nyelv ldstr vezérlőkódja egy új sztring-objektumot tölt be a felügyelt heapre. A korábbi sztringobjektum, amely a „My other string" értéket tárolta, végül szemétgyűjtésre kerül.



3.13. ábra: *A sztringobjektumok értékadásának eredménye egy új sztringobjektum*

Vagyis mit kell ebből leszűrnünk? Röviden annyit, hogy ha nem megfelelően használjuk a sztringtípust, akkor nem hatékony és nagyméretű kódot kapunk, különösen sztringösszefűzés esetén. Ha egyszerű karakteres adatokat, például taj számokat, kereszt- és vezetékneveket vagy az alkalmazásban használt egyszerű sztringliterálokat kell ábrázolnunk, a string típus a tökéletes választás.

Ha azonban olyan alkalmazást készítünk, amely intenzíven használ szöveges adatokat (mint például a szövegszerkesztő programok), nem lenne túl jó ötlet a megszerkesztendő szöveget sztringtípussal reprezentálni, hiszen így teljesen nyilvánvalóan (és sokszor közvetett módon) rengeteg felesleges másolatot kellene készítenünk a sztringadatokról. Akkor mit is tehet a programozó?

A System.Text.StringBuilder típus

A sztringtípus tehát nem bizonyul hatékonynak akkor, ha tömegével kell használni őket. A .NET-alaposztálykönyvtár azonban rendelkezik egy `System.Text` névtérrel, és ebben a (viszonylag kicsi) névtérben található a `StringBuilder` nevű osztály. A `System.String` osztályhoz hasonlóan a `StringBuilder` is definiál olyan osztályokat, amelyek szegmensek helyettesítését, formázását stb. teszik lehetővé. Ha ezt a típust akarjuk a C#-kódban használni, az első lépés importálni a megfelelő névteret:

```
// A StringBuilder itt található!
using System.Text;
```

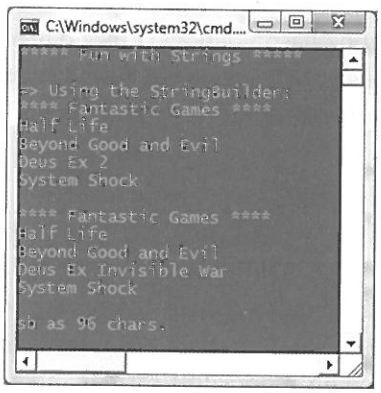
A `StringBuilderhez` kapcsolódó különlegesség, hogy az ilyen típusú tagok meghívásakor közvetlenül az objektum belső karakteradatait módosítjuk (ezáltal jelentősen nő a hatékonysága), nem pedig módosított formában készítünk másolatot az adatokról. A `StringBuilder` egy példányának létrehozá-

sakor az objektum kezdeti értékének megadásához több különböző *konstruktor* közül választhatunk. Ha még ismeretlen számunkra a konstruktorok világa, röviden annyit érdemes tudni róluk, hogy egy adott kiinduló állapottal teszik lehetővé objektumok létrehozását a `new` kulcsszó használatakor. Vizsgáljuk meg a `StringBuilder` következő alkalmazását:

```
static void FunWithStringBuilder()
{
  Console.WriteLine("=> Using the StringBuilder:");
  StringBuilder sb = new StringBuilder("**** Fantastic Games ****");
  sb.Append("\n");
  sb.AppendLine("Half Life");
  sb.AppendLine("Beyond Good and Evil");
  sb.AppendLine("Deus Ex 2");
  sb.AppendLine("System Shock");
  Console.WriteLine(sb.ToString());

  sb.Replace("2", "Invisible War");
  Console.WriteLine(sb.ToString());
  Console.WriteLine("sb as {0} chars.", sb.Length);
  Console.WriteLine();
}
```

Az itt létrehozott `StringBuilder` kezdőértéke „`**** Fantastic Games ****`". Mint láthatjuk, a belső pufferhez csatoljuk hozzá, így tetszés szerint cserélhetjük (vagy törölhetjük) a karaktereket. Az alapértelmezés szerint a `StringBuilder` legfeljebb 16 karakterből álló `sztring` tárolására képes; viszont ez a kezdőérték további konstruktorparaméterek segítségével módosítható:

```
// A StringBuilder kezdőértéke legyen 256.
StringBuilder sb =
  new StringBuilder("**** Fantastic Games ****", 256);
```



3.14. ábra: *A StringBuilder sokkal hatékonyabb, mint a sztring*

Ha a megadott korlátnál több karaktert csatolunk, a `StringBuilder` objektum átmásolja a benne lévő adatokat egy új példányba, és a megadott korlátra növeli a puffert. A 3.14. ábrán látható az előbbi segédfüggvény kimenete.

Forráskód A FunWithStrings projekt megtalálható a 3. fejezet alkönyvtárában.

System.DateTime és System.TimeSpan

Az alapvető adattípusok vizsgálatának lezárásaként hadd hívjam fel a figyelmet arra, hogy a `System` névtér néhány olyan hasznos adattípust definiál, amelyekre nincsenek C#-kulcsszavak – konkrétan a `DateTime` és a `TimeSpan` struktúrákat (a `System.Guid` és a `System.Void` a 3.6. ábrán látható).

A `DateTime` egy adott dátumot (hónap, nap, év) és időértéket reprezentáló adatokat tartalmaz, ezek az adott tagok segítségével többféle formátumban megjeleníthetők. Ehhez egyszerű példaként vizsgáljuk meg az alábbi utasításokat:

```
// A konstruktor az (év, hónap, nap) formátummal dolgozik.
DateTime dt = new DateTime(2004, 10, 17);

// A hónap mely napja van ma?
Console.WriteLine("The day of {0} is {1}", dt.Date, dt.DayOfWeek);
dt = dt.AddMonths(2);  // A hónap most december.
Console.WriteLine("Daylight savings: {0}",
  dt.IsDaylightSavingTime());
```

A `TimeSpan` struktúra különböző tagjainak segítségével egyszerűen definiálhatunk és alakíthatunk át időegységeket. Például:

```
// A konstruktor az (óra, perc, másodperc) formátummal dolgozik.
TimeSpan ts = new TimeSpan(4, 30, 0);
Console.WriteLine(ts);

// Vonjunk le 15 percet az aktuális TimeSpan értékből, és
// írjuk ki az eredményt.
Console.WriteLine(ts.Subtract(new TimeSpan(0, 15, 0)));
```

Nyilvánvaló tehát, hogy minden C#-adattípus kulcsszavának megvan a megfelelője a .NET-alaposztálykönyvtárakban, s ezek mindegyike állandó szereppel rendelkezik. Bár nem részleteztük az alaptípusok minden tagját, ám a beépített .NET-adattípusokról további részletekért tekintsük át a .NET Framework 3.5 SDK dokumentációját.

Szűkítő és bővítő adattípus-konverziók

A belső adattípusok használatához kapcsolódva a következőkben vizsgáljuk meg az *adattípus-konverzió* témakörét. Tegyük fel, hogy van egy új parancssorialkalmazás-projektünk (TypeConversions néven), amely a következő osztálytípust definiálja:

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

    // Adjunk össze két short változót, és írjuk ki az eredményt.
    short numb1 = 9, numb2 = 10;
    Console.WriteLine("{0} + {1} = {2}",
      numb1, numb2, Add(numb1, numb2));
    Console.ReadLine();
  }

  static int Add(int x, int y)
  { return x + y; }
}
```

Figyeljük meg, hogy az Add() metódus két int paramétert vár. Ezzel szemben a Main() metódus valójában két short típusú változót ad át. Bár ez az adattípusok teljesen rossz párosításának tűnhet, a programot hiba nélkül fordíthatjuk és futtathatjuk, és a várt 19-es eredményt kapjuk vissza.

A fordító azért tekinti ezt a kódot szintaktikailag helyesnek, mert nincs meg a lehetőség az adatvesztésre. Mivel a short maximális értéke (32767) jócskán az int típus tartományán (2147483647) belül van, a fordító implicit módon *kibővít* minden short típust int típusra. Formálisan a *kibővítés* kifejezést használjuk az adatvesztéssel nem járó, implicit „felfelé kasztolás"-ra

Megjegyzés Lapozzuk fel a „Type Conversion Tables (típuskonverziós tábla)" témakört a .NET Framework 3.5 SDK dokumentációban az egyes C#-adattípusok megengedett bővítő konverzióiért.

Bár az előző példában ez az implicit bővítés jól működött, más esetekben ez a „szolgáltatás" fordítási idejű hibák forrása lehet. Tegyük fel például, hogy a numb1 és numb2 értékét úgy állítottuk be, hogy (összeadva) túlcsordulnak a short maximális értékén.

Tegyük fel továbbá, hogy az `Add()` metódus visszatérési értékét egy új lokális `short` típusú változóban tároljuk ahelyett, hogy közvetlenül kiíratnánk az eredményt a konzolra:

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

  // Fordítási hiba következik!
  short numb1 = 30000, numb2 = 30000;
  short answer = Add(numb1, numb2);
  Console.WriteLine("{0} + {1} = {2}",
    numb1, numb2, answer);
  Console.ReadLine();
}
```

Ebben az esetben a fordító a következő hibaüzenetet adja:

```
Cannot implicitly convert type 'int' to 'short'. An explicit conversion
exists (are you missing a cast?)
```

A probléma az, hogy bár az `Add()` metódus képes 60000 értékű `int` visszaadására (hiszen belefér a `System.Int32` tartományába), az értéket nem lehet `short` típusban tárolni (mivel túlcsordul a típus határán). Formálisan: a CLR nem tudott *szűkítő műveletet* végrehajtani. A szűkítés értelemszerűen a bővítés logikai ellentéte, amelyben egy nagyobb értéket egy kisebb változóban tárolunk.

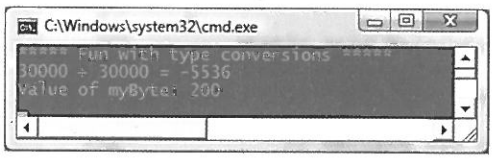
Fontos tudni, hogy minden szűkítő konverzió fordítói hibát eredményez, még akkor is, ha a szűkítő konverziónak valójában sikerülnie kellene. A következő kód is például fordítási hibát eredményez:

```
// Újabb fordítási hiba!
static void NarrowingAttempt()
{
  byte myByte = 0;
  int myInt = 200;
  myByte = myInt;
  Console.WriteLine("Value of myByte: {0}", myByte);
}
```

Itt az int változóban (`myInt`) tárolt érték bőven elfér a byte tartományában, ezért azt várnánk, hogy a szűkítő művelet ne okozzon futásidejű hibát. Ezzel szemben mivel a C# nyelvet a típusbiztonság szemléletében tervezték, mégis fordítási hibát kapunk.

Ha a fordítónak azt szeretnénk közvetíteni, hogy a szűkítő műveletből eredő lehetséges adatvesztéssel szeretnénk dolgozni, *explicit kasztot* kell alkalmaznunk a C#-kasztoperátor segítségével. A 3.15. ábrán látható a `Program` típus következő módosítása, és az így kapott kimenet.

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with type conversions *****");
    short numb1 = 30000, numb2 = 30000;

    // int érték explicit kasztolása short típusba
    // (lehetséges adatvesztés).
    short answer = (short)Add(numb1, numb2);

    Console.WriteLine("{0} + {1} = {2}",
      numb1, numb2, answer);
    NarrowingAttempt();
    Console.ReadLine();
  }

  static int Add(int x, int y)
  { return x + y; }

  static void NarrowingAttempt()
  {
    byte myByte = 0;
    int myInt = 200;
    // int érték explicit kasztolása byte adatba (nincs adatvesztés).
    myByte = (byte)myInt;
    Console.WriteLine("Value of myByte: {0}", myByte);
  }
}
```



3.15. ábra *Hoppá! A számok összeadásánál elvesztettünk néhány adatot*

Hibacsapda állítása a szűkítő adatkonverzióknak

Mint láthattuk, az explicit kaszt lehetővé teszi, hogy szűkítő konverzió alkalmazására kényszerítsük a fordítót még akkor is, ha ezzel adatot vesztünk. A `NarrowingAttempt()` metódus esetében ez nem is volt probléma, hiszen a 200-as érték bőségesen elfér a byte tartományában. Viszont a `Main()`-en belül

a két short összeadásakor az eredmény teljességgel elfogadhatatlan lesz (30000 + 30000 = -5536?). Ha olyan alkalmazást készítünk, amelyben az adatvesztés egyáltalán nem fogadható el, a C# checked és unchecked kulcsszava segítségével biztosíthatjuk azt, hogy ne történhessen észrevétlen adatvesztés.

A kulcsszavak használatának illusztrálására tegyük fel, hogy a Program osztályon belül van egy olyan új metódusunk, amely megpróbál két bytetípust összeadni, amelyek értéke bőven a maximális érték (255) alatt van. Ha az ilyen típusokat összeadnánk (a visszaadott int típust byte-tá kasztolva), azt várhatnánk, hogy az eredmény pontosan a két tag összege legyen:

```
static void ProcessBytes()
{
  byte b1 = 100;
  byte b2 = 250;
  byte sum = (byte)Add(b1, b2);

  // A sum várt értéke 350. Ehelyett az értéke 94!
  Console.WriteLine("sum = {0}", sum);
}
```

Ennek az alkalmazásnak a kimenetében meglepődve tapasztalnánk, hogy az összeg (a várt 350 helyett) a 94-es értéket tárolja. A magyarázat egyszerű. Mivel a System.Byte csak 0 és 255 közötti érték tárolására képes (mert 256 értéket vehet fel), a sum a fenti esetben a túlcsordulás értékét tárolja (350 - 256 = 94). Az alapértelmezés szerint, ha nem alkalmazunk javító megoldást, a túlcsordulás/alulcsordulás hiba nélkül szerepel.

A fenti alkalmazásunkban a túlcsordulás és az alulcsordulás kezelésére két lehetőségünk adódik. Az egyik, hogy manuálisan kezeljük az összes túlcsordulást/alulcsordulást. Feltételezve, hogy megtaláltuk az összes túlcsordulást a programunkban, a következőképpen oldhatjuk fel az előző túlcsordulási hibát:

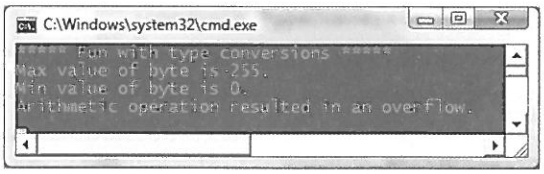
```
// A túlcsordulás megakadályozásához a sum értékét int tárolja.
byte b1 = 100;
byte b2 = 250;
int sum = (byte)Add(b1, b2);
```

Ezzel a technikával csak az a gond, hogy emberek vagyunk, és még a legjobb szándékunk ellenére is maradhatnak olyan hibák, amelyeket nem veszünk észre. Éppen ezért létezik a C#-ban a checked kulcsszó. Ha egy utasítást (vagy utasításblokkot) a checked kulcsszó hatókörébe burkolunk, a C#-fordító további CIL-utasításokat ad ki, s ezek ellenőrzik a két numerikus adattípus összeadása, szorzása, kivonása vagy osztása esetén esetlegesen fellépő túlcsordulást.

Ha túlcsordulás történik, futásidejű kivételt kapunk (`System.OverflowExceptiont`, a pontosság kedvéért). A 7. fejezet fog a strukturált kivételkezelés részleteivel, valamint a `try` és `catch` kulcsszavak használatával foglalkozni. Röviden nézzük meg most a következő módosítást:

```
static void ProcessBytes()
{
  byte b1 = 100;
  byte b2 = 250;

  // Utasítjuk a fordítót a CIL-kód hozzáadásával
  // kivétel kiváltására túlcsordulás/alulcsordulás
  // esetén.
  try
  {
    byte sum = checked((byte)Add(b1, b2));
    Console.WriteLine("sum = {0}", sum);
  }
  catch (OverflowException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Az `Add()` visszatérési értékét a `checked` kulcsszó hatókörén belülre helyeztük. Mivel az összeg nagyobb, mint egy byte, futásidejű kivételt váltunk ki. Figyeljük meg a `Message` tulajdonság segítségével kiírt hibaüzenetet a 3.16. ábrán.



3.16. ábra: *A checked kulcsszó futásidejű kivételt kényszerít ki adatvesztés esetén*

Ha egy kódutasításblokkban túlcsordulás-ellenőrzést szeretnénk kikényszeríteni, a következőképpen kell definiálnunk a *checked hatókörét:*

```
try
{
  checked
  {
    byte sum = (byte)Add(b1, b2);
    Console.WriteLine("sum = {0}", sum);
  }
}
```

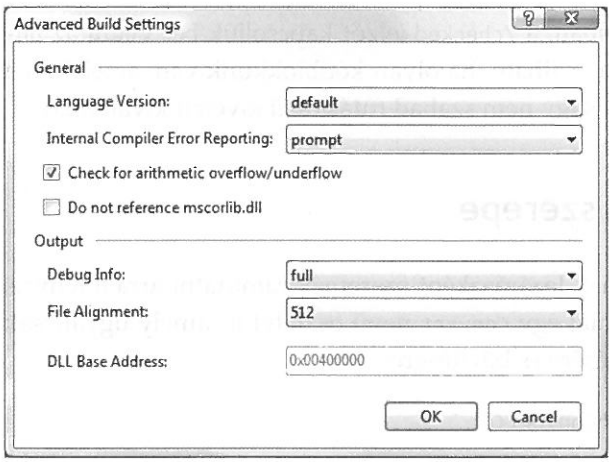
```
catch (OverflowException ex)
{
  Console.WriteLine(ex.Message);
}
```

Mindkét esetben a kérdéses kódban automatikusan ellenőrzés történik a lehetséges túlcsordulásra, és ha van, az esemény túlcsordulás-kivételt vált ki.

Projektszintű túlcsordulás-ellenőrzés beállítása

Ha olyan alkalmazást készítünk, amely soha nem engedheti meg a csendes túlcsordulást, akkor rengeteg sornyi kódot kell a `checked` kulcsszó hatókörébe burkolnunk. Alternatív megoldásként a C# támogatja a `/checked` jelző használatát. Ha ezt bekapcsoljuk, minden aritmetikai kifejezés túlcsordulását ellenőrizhetjük anélkül, hogy használnunk kellene a `checked` C#-kulcsszót. Ha a fordító túlcsordulást talál, futásidejű kivételt dob.

A jelző bekapcsolásához Visual Studio 2008-ban nyissuk meg a projektünk tulajdonságablakát, majd a Build fülön kattintsunk az Advanced gombra. A megjelenő párbeszédablakban az aritmetikai túlcsordulás/alulcsordulás jelölőnégyzetet pipáljuk be. (lásd 3.17. ábra).



3.17. ábra: *Projektszintű túlcsordulás/alulcsordulás adatellenőrzésének bekapcsolása*

Ennek a beállításnak az engedélyezése akkor lehet nagyon hasznos, amikor hibakeresési verziót hozunk létre. Ha már az összes túlcsordulás kivételt kiszórítottuk az alapkódból, nyugodtan kikapcsolhatjuk a `/checked` jelzőt a további verziókhoz (ez javítani fogja az alkalmazásunk futásidejű teljesítményét).

Az unchecked kulcsszó

Ha bekapcsoltuk ezt a projektszintű beállítást, mit tegyünk abban az esetben, ha olyan kódblokkunk van, amelyben viszont az adatvesztés *elfogadható*? Mivel a `/checked` jelző kiértékel minden aritmetikai logikát, a C#-ban létezik `unchecked` kulcsszó is, amellyel esetről esetre kikapcsolható a túlcsordulás-kivétel dobása. Ennek a kulcsszónak a használata megegyezik a `checked` kulcsszóéval abban, hogy megadhatunk egyetlen utasítást vagy utasítások blokkját is, például:

```
// Ha a /checked kulcsszó engedélyezett,
// a kódblokk nem vált ki
// futásidejű kivételt.
unchecked
{
  byte sum = (byte)(b1 + b2);
  Console.WriteLine("sum = { 0} ", sum);
}
```

A `checked` és `unchecked` C#-kulcsszavakkal kapcsolatban tehát tudni kell, hogy a .NET alapértelmezett futásidejű viselkedése figyelmen kívül hagyja az aritmetikai túlcsordulást. Amikor külön akarunk egyes utasításokat kezelni, a `checked` kulcsszót használjuk. Ha csapdába akarjuk ejteni a túlcsordulási hibákat a teljes alkalmazásban, a `/checked` jelzőt kapcsoljuk be. Végül az `unchecked` kulcsszó akkor használható, ha olyan kódblokkunk van, amelyben a túlcsordulás elfogadható (és így nem szabad futásidejű kivételt kiváltania).

A System.Convert szerepe

Az adatkonverzió témájának lezárásaként szeretnék rámutatni arra a tényre, hogy a `System` névtér definiál egy `Convert` nevű osztályt is, amely ugyancsak használható adatok szűkítésére és bővítésére:

```
static void NarrowWithConvert()
{
  byte myByte = 0;
  int myInt = 200;
  myByte = Convert.ToByte(myInt);
  Console.WriteLine("Value of myByte: {0}", myByte);
}
```

A `System.Convert` alkalmazásának egyik előnye, hogy nyelvsemleges módon biztosítja a konverziót az adattípusok között. Minthogy azonban a C# rendelkezik explicit típuskonverziós operátorral, a `Convert` típus alkalmazása az adattípus-konverzióra általában személyes preferencia függvénye.

Forráskód A TypeConversions projekt megtalálható a 3. fejezet alkönyvtárában.

Ciklus készítése a C#-ban

Minden programozási nyelv biztosít valamilyen módot a kódblokkok ismétlésére egy befejezési feltétel teljesüléséig. Az eddigi nyelvhasználattól függetlenül a C# iterációs utasítások nem fognak nagy meglepetéssel szolgálni, és nem igényelnek sok magyarázatot sem. A C# a következő négy ciklusszerkezetet tartalmazza:

- for ciklus,
- foreach/in ciklus,
- while ciklus,
- do/while ciklus.

Vizsgáljunk meg sorban minden ciklusszerkezetet egy `IterationsAndDecisions` nevű új parancssorialkalmazás-projekt segítségével.

A for ciklus

Egy kódblokkon adott számú iteráció végrehajtásához a for utasítás kellően nagy rugalmasságot biztosít. Megadhatjuk, hogy a kódblokk hányszor ismételje meg önmagát, valamint a kilépési feltételt is. Nézzünk erre egy szintaxismintát:

```
// Egyszerű for ciklus.
static void ForAndForEachLoop()
{
  // Figyelem! Az "i" csak a for ciklus hatókörében látható.
  for(int i = 0; i < 4; i++)
  {
    Console.WriteLine("Number is: {0} ", i);
  }
  // Az "i" itt nem látható.
}
```

Minden régi jól bevált C-, C++- és Java-trükk működik a C# `for` utasításainál is. Létrehozhatunk bonyolult kilépési feltételeket, végtelen ciklusokat, és használhatjuk a `goto`, `continue` és `break` kulcsszavakat. Ezt a ciklusszerkezetet kedvünk szerint alkalmazhatjuk. Ha további információra lenne szükségünk a C# `for` kulcsszaváról, lapozzuk fel a .NET Framework 3.5 SDK dokumentációját.

A foreach ciklus

A C# `foreach` kulcsszó egy tömb elemeinek iterációját teszi lehetővé anélkül, hogy a tömb felső határát ellenőriznünk kellene. Íme, két példa a `foreach` használatára; az egyik egy sztringtömböt, a másik egy egész tömböt jár be:

```
// A tömb elemeinek iterációja a foreach kulcsszóval.
static void ForAndForEachLoop()
{
...
  string[] carTypes = {"Ford", "BMW", "Yugo", "Honda" };
  foreach (string c in carTypes)
    Console.WriteLine(c);

  int[] myInts = { 10, 20, 30, 40 };
  foreach (int i in myInts)
    Console.WriteLine(i);
}
```

Az egyszerű tömbökön végrehajtott iteráción túl a `foreach` használható a rendszer által biztosított és a felhasználók által definiált gyűjteményeken történő iterációra is. Ennek részleteit a 9. fejezetben tárgyaljuk, ugyanis a `foreach` kulcsszó ezen vonásának megértéséhez szükség van az interfészalapú programozási ismeretekre, valamint az `IEnumerator` és az `IEnumerable` interfészek szerepének ismeretére.

A while és a do/while ciklusszerkezetek

A `while` ciklusszerkezet akkor használható, ha egy utasításblokk végrehajtása addig tart, amíg el nem érünk egy kilépési feltételt. A `while` ciklus hatókörén belül természetesen biztosítanunk kell, hogy ez a kilépési feltétel tényleg bekövetkezzen, másképp beragadunk egy végtelen ciklusba. A következő példában az „In while loop" üzenet jelenik meg mindaddig, amíg a felhasználó a ciklusból nem lép ki a parancssorba begépelt a **yes** szóval:

```
static void ExecuteWhileLoop()
{
  string userIsDone = "";

  // A sztring alacsonyabb osztálybeli másolatának ellenőrzése.
  while(userIsDone.ToLower() != "yes")
  {
    Console.Write("Are you done? [yes] [no]: ");
    userIsDone = Console.ReadLine();
    Console.WriteLine("In while loop");
  }
}
```

A `while` ciklushoz szorosan kötődik a `do/while` utasítás. Az egyszerű `while` ciklushoz hasonlóan a `do/while` is akkor használható, amikor egy adott műveletet nem meghatározott alkalommal kell végrehajtani. A különbség csupán annyi, hogy a `do/while` ciklus esetén a megfelelő kódblokk garantáltan legalább egyszer végrehajtódik (ezzel szemben előfordulhat, hogy a rendszer egy egyszerű `while` ciklust soha nem hajt végre, ha a kilépési feltétel mindjárt az induláskor igaz).

```
static void ExecuteDoWhileLoop()
{
  string userIsDone = "";

  do
  {
    Console.WriteLine("In do/while loop");
    Console.Write("Are you done? [yes] [no]: ");
    userIsDone = Console.ReadLine();
  }while(userIsDone.ToLower() != "yes");
                              // Figyeljünk a pontosvesszőre!
}
```

Elágazó szerkezetek és a reláció/egyenlőség-operátorok

Az utasításblokkok ciklusba szervezéséhez kapcsolódik a következő téma: hogyan vezéreljük a program futását. A C# két egyszerű szerkezetet kínál a program futásának vezérlésére, különböző feltételek alapján.

- az `if/else` utasítás,
- a `switch` utasítás.

Az if/else utasítás

Először vizsgáljuk meg az `if/else` utasítást. A C és C++ nyelvektől eltérően a C# `if/else` utasítása csak logikai kifejezéseken használható, nem olyan ad hoc értékeken, mint a -1 vagy 0. Éppen ezért az `if/else` utasítást a 3.7. táblázatban bemutatott C#-operátorokkal használjuk együtt, hogy tényleges logikai értéket kapjunk.

C# egyenlőségvizsgáló/ relációs operátorok	Példa a használatra	Valós jelentése
`==`	`if(age == 30)`	Visszatérési értéke csak akkor igaz, ha mindkét kifejezés megegyezik.
`!=`	`if("Foo" != myStr)`	A visszatérési értéke csak akkor igaz, ha mindkét kifejezés különböző.

`<`
`>`
`<=`
`>=`
	`if(bonus < 2000`
	`if(bonus > 2000)`
	`if(bonus <= 2000)`
	`if(bonus >= 2000)`
		A visszatérési értéke igaz, ha az A kifejezés kisebb, nagyobb, kisebb vagy egyenlő, illetőleg nagyobb vagy egyenlő, mint a B kifejezés.

3.7. táblázat: *C# relációs és egyenlőségvizsgáló operátorok*

Megint csak a C- és C++-programozóknak figyelniük kell arra, hogy a „nem egyenlő nulla" értékvizsgálatra alkalmazott régi jól bevált trükkök nem használhatók a C#-ban. Tegyük fel, hogy azt szeretnénk megtudni, hogy a vizsgált sztring nulla karakternél hosszabb-e. Csábító lenne ezt írni:

```
static void ExecuteIfElse()
{
  // Érvénytelen, a Length int értéket ad vissza, nem logikait.
  string stringData = "My textual data";
  if(stringData.Length)
  {
    Console.WriteLine("string is greater than 0 characters");
  }
}
```

Ha a `String.Length` tulajdonság segítségével szeretnénk eldönteni a feltételes kifejezés igaz vagy hamis voltát, módosítanunk kell a kifejezést ahhoz, hogy logikai típusúvá váljon. Például:

```
// Érvényes, mivel a kifejezés értéke igaz vagy hamis.
if(stringData.Length > 0)
{
  Console.WriteLine("string is greater than 0 characters");
}
```

Egy `if` utasítás állhat bonyolult kifejezésekből, és tartalmazhat `else` utasításokat a bonyolultabb ellenőrzések elvégzése érdekében. A szintaxis ugyanaz, mint a C(++) és a Java esetében (és nem áll messze a Visual Basictől sem). Az összetett kifejezések készítéséhez a C# a logikai operátorok szokásos készletét biztosítja, ez látható a 3.8. táblázatban.

Operátorok	Példa	Valós jelentése
&&	`if((age == 30) && (name == "Fred"))`	logikai AND operátor
||	`if((age == 30) || (name == "Fred"))`	logikai OR operátor
!	`if(!myBool)`	logikai NOT operátor

3.8. táblázat: *A C# logikai operátorai*

A switch utasítás

A C# által nyújtott másik egyszerű választási szerkezet a `switch` utasítás. Mint a többi C-alapú nyelvben is, a `switch` utasítás teszi lehetővé a programvégrehajtási folyamat kezelését előre definiált választási lehetőségek alapján. A következő `Main()` például egy adott sztringüzenetet ír ki a két lehetséges választás egyike alapján (az alapértelmezett eset az érvénytelen választást kezeli):

```
// switch utasítás numerikus értékkel.
static void ExecuteSwitch()
{
  Console.WriteLine("1 [C#], 2 [VB]");
  Console.Write("Please pick your language preference: ");

  string langChoice = Console.ReadLine();
  int n = int.Parse(langChoice);

  switch (n)
  {
    case 1:
      Console.WriteLine("Good choice, C# is a fine language.");
    break;
    case 2:
      Console.WriteLine("VB .NET: OOP, multithreading, and more!");
    break;
    default:
      Console.WriteLine("Well...good luck with that!");
    break;
  }
}
```

Megjegyzés A C# megköveteli, hogy minden eset (az alapértelmezett is), amely végrehajtható utasításokat tartalmaz, lezáró `break` vagy `goto` utasítással rendelkezzen, hogy megakadályozza a program másik ágba futását.

A C# `switch` utasítás egyik jelentős tulajdonsága, hogy a numerikus adatokon kívül sztringadatokat is tud értékelni. Nézzünk meg egy sztringadatokat értékelő módosított `switch` utasítást (figyeljük meg, hogy ezzel a megközelítéssel nem szükséges a felhasználói adatokat numerikus értékként elemezni):

```
static void ExecuteSwitchOnString()
{
  Console.WriteLine("C# or VB");
  Console.Write("Please pick your language preference: ");
```

```
    string langChoice = Console.ReadLine();
    switch (langChoice)
    {
      case "C#":
        Console.WriteLine("Good choice, C# is a fine language.");
      break;
      case "VB":
        Console.WriteLine("VB .NET: OOP, multithreading and more!");
      break;
      default:
        Console.WriteLine("Well...good luck with that!");
      break;
    }
  }
```

Forráskód Az IterationsAndDecisions projekt megtalálható a 3. fejezet alkönyvtárában.

Összefoglalás

A fejezet célja az volt, hogy megismerjük a C# programozási nyelv több különböző szempontját. Megvizsgáltuk azokat a szerkezeteket, amelyek általánosan használatosak minden alkalmazásban. Az alkalmazásobjektum szerepének vizsgálata után megtudtuk, hogy minden végrehajtható C#-programnak kell egy `Main()` metódust definiáló típussal rendelkeznie, amely a program belépési pontjaként szolgál. A `Main()` hatókörén belül általában bármennyi objektum létrehozható, amelyek együttműködve működtetik az alkalmazást.

Ezután a C# beépített adattípusainak tanulmányozásában mélyedtünk el. Minden adattípuskulcsszó (pl. `int`) tulajdonképpen a `System` névtérben (`System.Int32` ebben az esetben) szereplő teljes típus egy rövidített jelölése. Így minden C#-adattípus számos beépített taggal rendelkezik. Hasonlóképpen megtanultuk a *bővítés* és *szűkítés*, valamint a `checked` és `unchecked` kulcsszavak szerepét.

Végezetül megvizsgáltuk a C# által támogatott különböző iterációs és elágazási konstrukciókat. Most, hogy már tisztában vagyunk az alapvető elemekkel, a következő fejezetben befejezzük a nyelv központi elemeinek vizsgálatát. Ha ez megtörtént, akkor a C# objektumorientált elemeivel való ismerkedést kezdhetjük meg.

NEGYEDIK FEJEZET

A C# alapvető építőelemei, II. rész

Ez a fejezet az előző folytatásaként befejezi a C# programozási nyelv alapvető elemeinek a vizsgálatát. Először a C#-metódusok konstrukciójának különböző részleteit tekintjük át, bemutatva az `out`, a `ref` és a `params` kulcsszavakat. A *metódus-túlterhelés* témáját követően a tömbtípusok kezelésének vizsgálata következik a C#-szintaxis használatával, valamint megismerkedünk a kapcsolódó `System.Array` osztálytípusban lévő funkcionalitással is.

Emellett ez a fejezet tárgyalja a felsoroláskészítését és a struktúratípusokat, beleértve az *értéktípus* és a *referenciatípus* közti különbségek részleteit is. Vizsgálódásunkat végül a nullázható adattípusok, valamint a ? és a ?? operátorok szerepének áttekintésével zárjuk.

Metódusok és paramétermódosítók

Először megvizsgáljuk a típusmetódusok definiálásának részleteit. A `Main()` metódus (lásd a 3. fejezetet) és ugyanígy az egyedi metódusok is felvehetnek paramétereket, és visszaadhatnak értékeket. Azután megnézzük azt, hogy a metódusokat hogyan lehet osztályok vagy struktúrák hatókörén belül megvalósítani (és interfésztípusokon belül definiálni), valamint hogy különböző kulcsszavakkal (`internal`, `virtual`, `public`, `new` stb.) hogyan lehet megadni a viselkedésüket. Jelenleg metódusaink mindegyike ezt az alapvető formát követi:

```
// Statikus metódusokat közvetlenül,
// osztálypéldány létrehozása nélkül lehet hívni.
class Program
{
  // static returnVal MethodName(args) {...}
  static int Add(int x, int y){ return x + y; }
}
```

Noha a C#-ban elég egyértelműen meg lehet egy metódust határozni, van néhány kulcsszó, amelyekkel azt vezérelhetjük, hogy a paramétereket hogyan lehet átadni a szóban forgó metódusnak; ezek listáját tartalmazza a 4.1. táblázat.

Paramétermódosító	Valós jelentés
(None)	Ha egy paraméter nem rendelkezik paramétermódosítóval, akkor az átadása értékként történik, ez pedig azt jelenti, hogy a meghívott metódus megkapja az eredeti adat egy másolatát.
`out`	A kimeneti paramétereknek a meghívott metódusban kell értéket adni (ezért referenciaként továbbítódik). Ha a meghívott metódusban nem kap értéket, akkor fordítási hibával szembesülünk.
`ref`	A paraméterhez a hívó hozzárendel egy értéket, ám a meghívott metódus ezt módosíthatja (minthogy az adat szintén referenciaként továbbítódik). Nincs fordítási hiba, ha a meghívott metódus nem ad új értéket a referenciaként átvett paraméterhez.
`params`	Ez a paramétermódosító lehetővé teszi, hogy változó számú argumentumot egyedüli logikai paraméterként küldjünk be. Egy metódusnak csak egyetlen `params` módosítója lehet, ez pedig a metódus utolsó paramétere kell, hogy legyen.

4.1. táblázat: *C#-paramétermódosítók*

Ahhoz, hogy ezeket a kulcsszavakat bemutassuk, hozzunk létre egy új parancssorialkalmazás-projektet `FunWithMethods` néven. A következőkben megvizsgáljuk az egyes kulcsszavak szerepét.

Az alapértelmezett paraméterátadási viselkedés

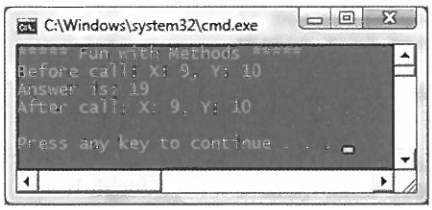
Az alapértelmezés szerint egy paramétert *értékként* adunk át egy függvénynek. Leegyszerűsítve, ha egy argumentumot nem jelölünk meg paramétermódosítóval, az adat másolatának átadása következik be. A fejezet végén látni fogjuk: az, hogy pontosan *mit* másol a rendszer, függ attól, hogy a paraméter érték- vagy referenciatípus-e. Vizsgáljuk meg a `Program` osztály következő metódusát, amely két értékként átadott numerikus adattípuson működik:

```
// A paramétereket alapértelmezés szerint értékként adjuk át.
static int Add(int x, int y)
{
  int ans = x + y;

  // A hívó nem látja ezeket a módosításokat,
  // mivel az eredeti adatok másolatát
  // változtatjuk.
  x = 10000; y = 88888;
  return ans;
}
```

A numerikus adatok az értéktípusok kategóriájába esnek. Ezért, ha a metódus hatókörén belül módosítjuk a paraméterek értékeit, annak a hívó nem lesz tudatában, ugyanis a hívó adatairól készült másolatban módosítjuk az értékeket:

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

  // Két paraméter átadása értékként.
  int x = 9, y = 10;
  Console.WriteLine("Before call: X: {0}, Y: {1}", x, y);
  Console.WriteLine("Answer is: {0}", Add(x, y));
  Console.WriteLine("After call: X: {0}, Y: {1}", x, y);
  Console.ReadLine();
}
```

Az x és az y értékei ugyanazok maradtak az `Add()` metódus meghívása előtt és után is (lásd a 4.1. ábrát).



4.1. ábra: *A paraméterek átadása alapértelmezés szerinti értékként történik*

Az out módosító

A következőkben a *kimeneti paramétereket* fogjuk használni. Az úgy meghatározott metódusoknak, amelyek felvesznek kimeneti paramétereket (az `out` kulcsszó révén), hozzá kell rendelniük azokat egy megfelelő értékhez, mielőtt a metódus véget érne (ha ezt nem tesszük meg, fordítási hibákat kapunk).

Ennek bemutatására nézzük meg az `Add()` metódus egy olyan alternatív verzióját, amely két számjegy összegét adja vissza a C# `out` módosítójának használatával (figyeljük meg, hogy a metódus visszatérési értéke `void`):

```
// A kimeneti paraméterek hozzárendelését a meghívott metódus végzi.
static void Add(int x, int y, out int ans)
{
  ans = x + y;
}
```

Kimeneti paraméterekkel rendelkező metódus meghívása az out módosító használatát is megköveteli. A kimeneti változóként átadott lokális változók hozzárendelésére használat előtt nincs szükség (ha így teszünk, akkor elvesztjük az eredeti értéket a hívás után):

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
  ...
  // A kimeneti paraméterként alkalmazott lokális változókhoz
  // nem kell kezdőértéket rendelni.
  int ans;
  Add(90, 90, out ans);
  Console.WriteLine("90 + 90 = {0}", ans);
  Console.ReadLine();
}
```

Az előző példát csak bemutatásra szántuk; valójában semmi szükség a kimeneti paraméter használatára ahhoz, hogy visszakapjuk az összeadás értékét. A C# `out` módosítója azonban nagyon hasznos célt is szolgál: lehetővé teszi, hogy a hívó egyetlen metódushívásból több visszatérési értéket is megszerezzen.

```
// Több kimeneti paramétert kapunk vissza.
static void FillTheseValues(out int a, out string b, out bool c)
{
  a = 9;
  b = "Enjoy your string.";
  c = true;
}
```

Ez a metódus a következőképpen hívható:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
  ...
  int i; string str; bool b;

  FillTheseValues(out i, out str, out b);
  Console.WriteLine("Int is: {0}", i);
  Console.WriteLine("String is: {0}", str);
  Console.WriteLine("Boolean is: {0}", b);
  Console.ReadLine();
}
```

Végül hangsúlyoznunk kell, hogy a kimeneti paramétereket meghatározó metódusoknak hozzá *kell* rendelniük a paramétert egy érvényes értékhez, mielőtt a metódus véget ér. Ezért a következő metódus fordítási hibát eredményez, mivel az egészparaméterhez nem rendeltünk értéket a metódus hatókörén belül:

```
static void ThisWontCompile(out int a)
{
  Console.WriteLine("This is an error...");
}
```

A ref módosító

A referenciaparaméterekre akkor van szükség, amikor lehetővé szeretnénk tenni, hogy egy metódus különböző, a hívó hatókörében deklarált (pl. rendező- vagy cserélőrutinok) adatelemeken működjön (és általában módosítsa az értéküket). Figyeljük meg a különbséget a kimeneti és a referenciaparaméterek között:

- A kimeneti paramétereknek nem kell kezdőértéket megadnunk, mielőtt átadjuk őket a metódusnak. Mindezt miért nem? Azért, mert a kimeneti paraméterek hozzárendelését a metódusnak magának kell biztosítania a kilépés előtt.
- A referenciaparamétereknek meg kell adnunk a kezdőértéket, mielőtt átadjuk őket a metódusnak. Mindezt miért? A referenciát egy már létező változónak adjuk át. Ha nem állítunk be kezdőértéket, az olyan, mintha érték nélküli lokális változóval dolgoznánk.

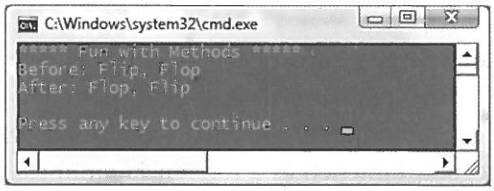
Nézzük meg a `ref` kulcsszó használatát egy olyan metódusban, amely felcserél két sztringet:

```
// Referenciaparaméterek.
public static void SwapStrings(ref string s1, ref string s2)
{
  string tempStr = s1;
  s1 = s2;
  s2 = tempStr;
}
```

Ezt a metódust így lehet meghívni:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
  ...
  string s1 = "Flip";
  string s2 = "Flop";
  Console.WriteLine("Before: {0}, {1} ", s1, s2);
  SwapStrings(ref s1, ref s2);
  Console.WriteLine("After: {0}, {1} ", s1, s2);
  Console.ReadLine();
}
```

A hívó itt hozzárendelt kezdőértéket a helyi sztringadatokhoz (`s` és `s2`). Amint a `SwapString()` hívása visszatér, már az `s1` már tartalmazza a „`Flop`" értéket, míg az `s2` a „`Flip`" értéket adja vissza (lásd a 4.2. ábrát).



4.2. ábra: *A referenciaparamétereket módosíthatja a meghívott metódus*

Megjegyzés A C# `ref` kulcsszót később újra megvizsgáljuk az „Érték és referenciatípusok megértése" című részben. A kulcsszó viselkedése némileg változni fog attól függően, hogy a paraméter „értéktípus" (struktúra) vagy „referenciatípus" (osztály).

A params módosító

Végül, de nem utolsósorban, a C# támogatja a *paramétertömbök* használatát is. A `params` kulcsszó működésének megértéséhez tisztában kell lennünk azzal (ahogy a neve is mutatja), hogy a C# hogyan kezeli a tömböket. (Később formálisan is megvizsgáljuk a `System.Array` típust a „Tömbök kezelése C#-ban" című részben.

A `params` kulcsszó lehetővé teszi, hogy egy metódusnak változó számú (ugyanolyan típusú) paramétert adjunk át *egyedüli logikai paraméterként*. A `params` kulcsszóval jelölt paramétereket akkor lehet feldolgozni, ha a hívó egy erősen tipizált tömböt küld, avagy elemek vesszővel elválasztott listáját. Ez némileg zavaró lehet. Ennek kiküszöbölésére hozzunk létre egy olyan függvényt, amely lehetővé teszi a hívónak, hogy bármennyi argumentumot továbbadjon, majd visszaadja a kiszámított átlagot.

Ha úgy hoznánk létre a metódus prototípusát, hogy felvegye a `double` típusú adatok tömbjét, ez arra kényszerítené a hívót, hogy először létrehozza a tömböt, majd kitöltse, végül átadja a metódusnak. Ha azonban úgy definiáljuk a `CalculateAverage()` metódust, hogy felvegye egész adatok paraméterét, a hívó egyszerűen át tudja adni a `double` adatok vesszővel elválasztott listáját. A .NET-futtatórendszer a háttérben automatikusan `double` típusú tömbbe csomagolja a `double` adatok készletét:

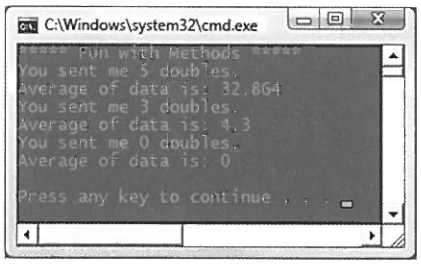
```
// Néhány double adat átlagát kapjuk vissza.
static double CalculateAverage(params double[] values)
{
  Console.WriteLine("You sent me {0} doubles.", values.Length);

  double sum = 0;
  if(values.Length == 0)
    return sum;

  for (int i = 0; i < values.Length; i++)
    sum += values[i];
  return (sum / values.Length);
}
```

Ez a metódus felveszi a `double` adatok paramétertömbjét. A metódus valójában ezt jelzi: „Küldj nekem bármennyi `double` adatot, és én kiszámolom az átlagot." Így a következő módok bármelyikével meghívhatjuk a `CalculateAverage()` metódust (ha nem használtuk a `params` módosítót a `CalculateAverage()` definiálásakor, akkor a metódus első meghívása fordítási hibát eredményez):

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

  // A bemenet double értékek vesszővel elválasztott listája...
  double average;
  average = CalculateAverage(4.0, 3.2, 5.7, 64.22, 87.2);
  Console.WriteLine("Average of data is: {0}", average);

  // ...vagy double adatok tömbje.
  double[] data = { 4.0, 3.2, 5.7 };
  average = CalculateAverage(data);
  Console.WriteLine("Average of data is: {0}", average);

  // A 0 átlaga 0!
  Console.WriteLine("Average of data is: {0}", CalculateAverage());

  Console.ReadLine();
}
```

Megjegyzés A félreérthetőség elkerülése érdekében a C#-ban egy metódus csak egyetlen `params` argumentumot támogat, ennek az utolsó paraméternek a paraméterlistában kell lennie.

Ez a technika kényelmes a hívónak, ugyanis a tömböt szükség esetén a CLR hozza létre. Amikor a tömb a meghívandó metódus hatókörébe kerül, teljes .NET-tömbként kezeljük, amely a `System.Array` alaposztálykönyvtár-típus minden szerepét tartalmazza. A 4.3. ábra mutatja a kimenetet.



4.3. ábra: *A params kulcsszó lehetővé teszi változó számú argumentummal rendelkező metódusok készítését*

Forráskód A `FunWithMethods` alkalmazás megtalálható a 4. fejezet alkönyvtárában.

A tag túlterhelésének megértése

Más modern objektumorientált nyelvekhez hasonlóan a C# is lehetővé teszi egy metódus *túlterhelését*. Amikor olyan, egyformán elnevezett tagok készletét határozzuk meg, amelyek a paraméterek számában (vagy típusában) különböznek, akkor erre azt mondjuk, hogy a szóban forgó tag túlterhelődött. Ennek ellenőrzésére hozzunk létre egy új parancssorialkalmazás-projektet `MethodOverloading` névvel.

Hogy megértsük, miért hasznos a túlterhelés, tételezzük fel, hogy Visual Basic 6.0 fejlesztők vagyunk, VB6-ot használunk egy olyan metóduskészlet létrehozására, amely visszaadja különböző bejövő típusok (egész számok, `double` adatok stb.) összegét. Mivel a VB6 nem támogatja a metódus-túlterhelést, definiálnunk kell egy olyan egyedi metódusgyűjteményt, amely lényegében ugyanazt csinálja (visszaadja az argumentumok összegét):

```
' VB6-kód.
Public Function AddInts(ByVal x As Integer,
    ByVal y As Integer) As Integer
  AddInts = x + y
End Function
Public Function AddDoubles(ByVal x As Double,
    ByVal y As Double) As Double
  AddDoubles = x + y
End Function
Public Function AddLongs(ByVal x As Long,
    ByVal y As Long) As Long
  AddLongs = x + y
End Function
```

Nemcsak nehéz karbantartani az ilyen kódokat, hanem a hívónak pontosan tudnia kell az egyes metódusok nevét is. A túlterheléssel lehetővé tehetjük a hívó számára, hogy egyedül az `Add()` nevű metódust hívja meg. A lényeg annak biztosítása, hogy a metódus minden verziója rendelkezzen különböző argumentumokkal (egyedül a visszatérési típusban különböző tagok *nem* elég egyediek). Figyeljük meg a következő osztálydefiníciót:

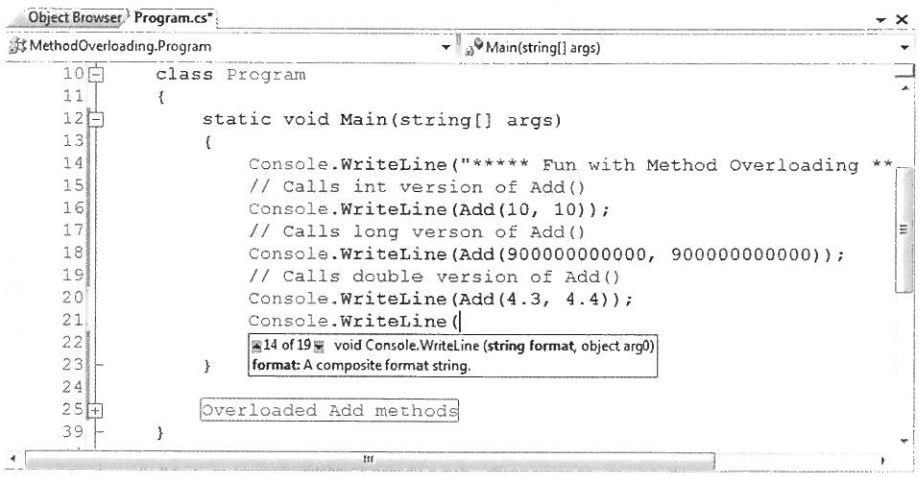
```
// C#-kód.
class Program
{
  static void Main(string[] args) { }

  // Túlterhelt Add() metódus.
  static int Add(int x, int y)
  { return x + y; }
```

```
  static double Add(double x, double y)
  { return x + y; }
  static long Add(long x, long y)
  { return x + y; }
}
```

Megjegyzés A 10. fejezetben szó lesz róla, hogy lehetséges olyan generikus metódusokat készíteni, amelyek a túlterhelés koncepcióját egy következő szintre emelik. Generikus típusok használatával definiálhatunk „helyőrzőket" egy metódusmegvalósítás számára. Ezeket a tag aktivizálásakor adjuk meg.

A hívó a szükséges paraméterekkel egyszerűen meghívhatja az Add() metódust, a fordítás pedig rendben megtörténik, ugyanis a fordító a megadott paraméterekkel képes lesz feloldani a meghívandó metódus helyes megvalósítását:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Method Overloading *****
  // Az Add() int verzióját hívja.
  Console.WriteLine(Add(10, 10));
  // Az Add() long verzióját hívja.
  Console.WriteLine(Add(900000000000, 900000000000));
  // Az Add() double verzióját hívja.
  Console.WriteLine(Add(4.3, 4.4));
  Console.ReadLine();
}
```



4.4. ábra: *A Visual Studio IntelliSense és a túlterhelt tagok*

A Visual Studio 2008 IDE segítséget biztosít túlterhelt tagok meghívásakor . Amikor beírjuk egy túlterhelt metódus (pl. a `Console.Writeline()`) nevét, akkor az IntelliSense listázza a szóban forgó metódus összes verzióját. Egy túlterhelt metódus összes verziójához ciklikusan visszatérhetünk a fel és a le nyilak használatával (lásd a 4.4. ábrát).

Forráskód A `MethodOverloading` alkalmazás megtalálható a 4. fejezet alkönyvtárában.

A következő témánk a tömbök, a felsorolások és a struktúrák készítése és kezelése lesz.

Tömbök kezelése a C#-ban

A *tömb* olyan adatelem-gyűjtemény, amelyhez numerikus index használatával lehet hozzáférni. Specifikusabban: a tömb ugyanolyan típusú folyamatos adatelemek gyűjteménye (számok tömbje, sztring tömb, `SportsCar`-tömb stb.). A tömbök deklarálása a C#-ban viszonylag egyértelmű. Hozzunk létre egy új parancssorialkalmazás-projektet (`FunWithArrays` néven), amely tartalmaz egy `SimpleArrays()` nevű segítőmetódust, a `Main()` metódusból meghívva:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Arrays *****");
    SimpleArrays();
  }

  static void SimpleArrays()
  {
    Console.WriteLine("=> Simple Array Creation.");
    // 3 elemet {0 - 2} tartalmazó int tömb hozzárendelése.
    int[] myInts = new int[3];

    // 100 elemből álló számozott sztring {0 - 99} tömb
    // inicializálása.
    string[] booksOnDotNet = new string[100];
    Console.WriteLine();
  }
}
```

Nézzük meg figyelmesen az előző kód megjegyzéseit. Amikor ezzel a szintaxissal deklarálunk egy C#-tömböt, akkor az abban használt számok az elemek számát jelképezik, nem a felső korlátot. A tömb mindig a 0. elemmel kezdődik. Ezért amikor azt írjuk, hogy `int [] myInts[3]`, akkor három elemből álló tömböt kapunk ({0, 1, 2}).

Ha meghatároztunk egy tömböt, akkor indexről indexre feltölthetjük az elemeket, ahogy ez a módosított `SimpleArrays()` metódusban is látható:

```
static void SimpleArrays()
{
  Console.WriteLine("=> Simple Array Creation.");
  // 3 egész számot tartalmazó tömb létrehozása és feltöltése.
  int[] myInts = new int[3];
  myInts[0] = 100;
  myInts[1] = 200;
  myInts[2] = 300;

  // Az értékek kiírása.
  foreach(int i in myInts)
  Console.WriteLine(i);
  Console.WriteLine();
}
```

Megjegyzés Figyeljünk arra, hogy ha deklarálunk egy tömböt, de nem töltjük fel az összes elemet, akkor minden elem az adattípus alapértelmezett értékét kapja meg (pl. egy logikai tömb hamisra lesz állítva, egy számtömb 0-ra stb.).

A C# tömbinicializáló szintaxisa

A tömbök elemről elemre történő feltöltése mellett megtehetjük azt is, hogy a tömböt a C# tömbinicializáló szintaxisát használva töltjük fel. Ehhez a tömb minden elemét a kapcsos zárójelen belül ({}) kell megadni. Ez a szintaxis akkor lehet hasznos, ha ismert méretű tömböt hozunk létre, és gyorsan szeretnénk megadni a kezdőértékeket. Figyeljük meg például a következő alternatív tömbinicializálást:

```
static void ArrayInitialization()
{
  Console.WriteLine("=> Array Initialization.");

  // Tömbinicializáló szintaxis a new kulcsszóval.
  string[] stringArray = new string[]
    { "one", "two", "three" };
  Console.WriteLine("stringArray has {0} elements",
    stringArray.Length);
```

```
    // Tömbinicializáló szintaxis a new kulcsszó nélkül.
    bool[] boolArray = { false, false, true };
    Console.WriteLine("boolArray has {0} elements", boolArray.Length);

    // Tömbinicializáló szintaxis a new kulcsszóval, a tömb méretével.
    int[] intArray = new int[4] { 20, 22, 23, 0 };
    Console.WriteLine("intArray has {0} elements", intArray.Length);
    Console.WriteLine();
}
```

Amikor ezt a „kapcsos zárójel" szintaxist használjuk, akkor nem kell megadnunk a tömb méretét (lásd a `stringArray` típus létrehozásakor), ugyanis ez a kapcsos zárójel hatókörében megadott elemek számából adódik. Ebben a szintaxisban a `new` kulcsszó használata tetszőleges (a `boolArray` típus létrehozásakor látható).

Az `IntArray` deklarációjakor a megadott numerikus érték ugyancsak a tömbben lévő elemek számát jelzi, nem pedig a felső korlátot. Ha a deklarált méret és az inicializálók száma nem egyezik, akkor fordítási hibát kapunk:

```
// HOPPÁ! A méret és az elemek száma nem egyezik!
int[] intArray = new int[2] { 20, 22, 23, 0 };
```

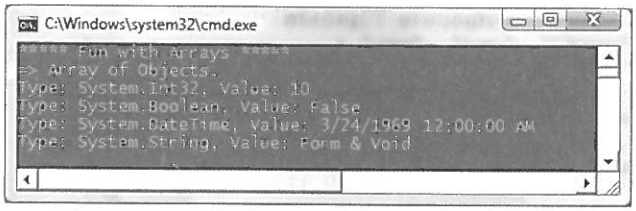
Objektumtömb definiálása

Egy tömböt úgy definiálunk tehát, hogy megadjuk azt az elemtípust, amely a tömbváltozóban lehet. Bár ez elég egyértelműnek tűnik, van még egy említésre méltó tényező. Az alapvető ősosztály minden egyes típus számára a .NET-típusrendszerben (beleértve az alapvető adattípusokat is) a `System.Object`. Így tehát, ha megadunk egy objektumtömböt, a benne lévő elemek lényegében bármilyen típusú adatok lehetnek. Nézzük meg az alábbi `ArrayOfObjects()` metódust (amelyet szintén a `Main()` metódusból meghívhatunk tesztelésre):

```
static void ArrayOfObjects()
{
  Console.WriteLine("=> Array of Objects.");

  // Az objektumtömb bármilyen típusú adatot tartalmazhat.
  object[] myObjects = new object[4];
  myObjects[0] = 10;
  myObjects[1] = false;
  myObjects[2] = new DateTime(1969, 3, 24);
  myObjects[3] = "Form & Void";
```

```
    foreach (object obj in myObjects)
    {
      // A tömb elemek típusának és értékének kiírása.
      Console.WriteLine("Type: {0}, Value: {1}", obj.GetType(), obj);
    }
    Console.WriteLine();
  }
```

Itt, a `myObjects` tartalmának iterálásakor, kiíratjuk az egyes elemek alapjául szolgáló típust a `System.Object GetType()` metódusával, valamint az aktuális elem értékét is. Röviden: a `System.Object.GetType()` metódust arra használhatjuk, hogy megszerezzük az elem teljesen meghatározott nevét. (A típusinformációkat és a reflexiós szolgáltatásokat lásd a 16. fejezetben.) A 4.5. ábra mutatja az előző kódrészlet kimenetét.



4.5. ábra: *Objektumtípus bármit tartalmazható tömbje*

Munka a többdimenziós tömbökkel

Az eddig látott egydimenziós tömbök mellett a C# két különböző típusú többdimenziós tömböt is támogat. Az elsőt *derékszögű tömbnek* (rectangular array) nevezzük, ez egy olyan többdimenziós tömb, ahol minden sor ugyanolyan hosszúságú. A többdimenziós derékszögű tömb deklarálása és feltöltése a következőképpen alakul:

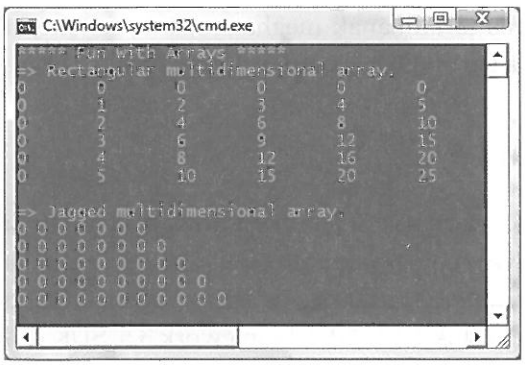
```
static void RectMultidimensionalArray()
{
  Console.WriteLine("=> Rectangular multidimensional array.");
  // Derékszögű többdimenziós tömb.
  int[,] myMatrix;
  myMatrix = new int[6,6];

  // (6 * 6) tömb feltöltése.
  for(int i = 0; i < 6; i++)
    for(int j = 0; j < 6; j++)
      myMatrix[i, j] = i * j;
```

```
    // (6 * 6) tömb kiírása.
    for(int i = 0; i < 6; i++)
    {
      for(int j = 0; j < 6; j++)
        Console.Write(myMatrix[i, j] + "\t");
      Console.WriteLine();
    }
    Console.WriteLine();
}
```

A második típusú többdimenziós tömb neve: *fűrészfogas tömb* (jagged array). Ahogy a név is mutatja, a fűrészfogas tömbök több belső tömböt is tartalmazhatnak, ezek mindegyike rendelkezhet egyedi felső határral. Például:

```
static void JaggedMultidimensionalArray()
{
  Console.WriteLine("=> Jagged multidimensional array.");
  // Fűrészfogas többdimenziós tömb (pl. tömbök tömbje).
  // 5 különböző tömb tömbje áll rendelkezésünkre.
  int[][] myJagArray = new int[5][];

  // A fűrészfogas tömb létrehozása.
  for (int i = 0; i < myJagArray.Length; i++)
    myJagArray[i] = new int[i + 7];

  // A sorok kiírása (minden elem alapértelmezett értéke 0!)
  for(int i = 0; i < 5; i++)
  {
    for(int j = 0; j < myJagArray[i].Length; j++)
      Console.Write(myJagArray[i][j] + " ");
    Console.WriteLine();
  }
  Console.WriteLine();
}
```

A 4.6. ábra mutatja a `Main()` metóduson belül lévő összes metódus meghívásának kimenetét.



4.6. ábra: *Derékszögű és fűrészfogas többdimenziós tömbök*

Tömbök mint paraméterek (és visszatérési értékek)

Ha létrehoztunk egy tömböt, nyugodtan átadhatjuk paraméterként, és megkaphatjuk tag visszatérési értékeként. A következő `PrintArray()` metódus például átveszi számok bemeneti tömbjét, és kiírja az összes elemét a konzolra, a `GetStringArray()` metódus pedig feltölti a sztringtömböt, és visszaadja a hívónak:

```
static void PrintArray(int[] myInts)
{
  for(int i = 0; i < myInts.Length; i++)
    Console.WriteLine("Item {0} is {1}", i, myInts[i]);
}

static string[] GetStringArray()
{
  string[] theStrings = { "Hello", "from", "GetStringArray" };
  return theStrings;
}
```

Ezeket a metódusokat a feltételezés szerint lehet meghívni:

```
static void PassAndReceiveArrays()
{
  Console.WriteLine("=>Arrays as params and return values.");
  // Tömb átadása paraméterként.
  int[] ages = {20, 22, 23, 0} ;
  PrintArray(ages);

  // Tömb beolvasása visszatérési értékként.
  string[] strs = GetStringArray();
  foreach(string s in strs)
    Console.WriteLine(s);
  Console.WriteLine();
}
```

Miután megnéztük a C#-tömbtípus tartalmának meghatározását, feltöltését és vizsgálatát, ismerkedjünk meg a `System.Array` osztály szerepével.

A System.Array ősosztály

Minden létrehozott tömb a `System.Array` osztályból kapja funkcionalitásának nagy részét. A gyakori tagok révén egy objektummodell használatával is dolgozhatunk a tömbökön. A 4.2. táblázat tartalmaz néhányat az érdekesebb tagok közül (teljes áttekintésért lapozzuk fel a .NET Framework 3.5. SDK dokumentációját).

Az Array osztály tagjai	Valós jelentés
`Clear ()`	Ez a statikus metódus üres értékre állítja a tömb meghatározott elemtartományát (0-ra az értékelemeket, statikusra az objektumreferenciákat).
`CopyTo()`	Ezzel a metódussal a forrástömbből másolhatunk elemeket a céltömbbe.
`GetEnumerator()`	Ez a metódus egy tömb `IEnumerator` interfészét adja vissza. Erre az interfészre a `foreach` szerkezetnek van szüksége. (Az interfészeket lásd a 9. fejezetben.)
`Length`	Ez a tulajdonság visszaadja a tömbben levő elemek számát.
`Rank`	Ez a tulajdonság az aktuális tömb dimenzióinak számát adja vissza.
`Reverse()`	Ez a statikus metódus megfordítja egy egydimenziós tömb tartalmát.
`Sort()`	Ez a statikus metódus rendezi a belső típusok egydimenziós tömbjét. Ha a tömbben lévő elemek megvalósítják az `IComparer` interfészt, akkor az egyedi típusokat is rendezhetjük (lásd a 9. fejezetet).

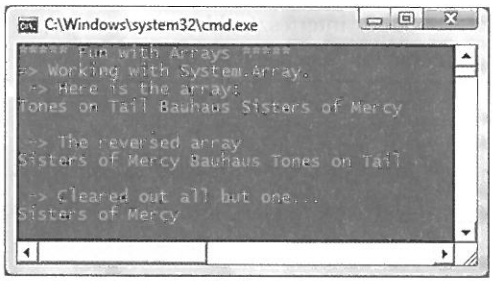
4.2. táblázat: *A System.Array néhány tagja*

Nézzünk meg néhány elemet működés közben. A következő kisegítő metódus a statikus `Reverse()` és `Clear()` metódusok használatával írja ki egy sztringtömb információit a konzolra:

```
static void SystemArrayFunctionality()
{
  Console.WriteLine("=> Working with System.Array.");
  // Elemek inicializálása indítás során.
  string[] gothicBands = {"Tones on Tail", "Bauhaus",
    "Sisters of Mercy"};

  // Nevek kiírása a meghatározott sorrendben.
  Console.WriteLine(" -> Here is the array:");
  for (int i = 0; i <= gothicBands.GetUpperBound(0); i++)
  {
    // Név kiírása
    Console.Write(gothicBands[i] + " ");
  }
  Console.WriteLine("\n");
```

```
    // Az elemek fordított sorrendje...
    Array.Reverse(gothicBands);
    Console.WriteLine(" -> The reversed array");
    // ... és kiírása.
    for (int i = 0; i <= gothicBands.GetUpperBound(0); i++)
    {
      // Név kiírása
      Console.Write(gothicBands[i] + " ");
    }
    Console.WriteLine("\n");

    // Az utolsó elem kivételével töröljük a tömb elemeit.
    Console.WriteLine(" -> Cleared out all but one...");
    Array.Clear(gothicBands, 1, 2);
    for (int i = 0; i <= gothicBands.GetUpperBound(0); i++)
    {
      // Név kiírása
      Console.Write(gothicBands[i] + " ");
    }
    Console.WriteLine();
  }
```

Ha meghívjuk ezt a metódust a `Main()` metódusból, akkor a 4.7. ábrán láthatóhoz hasonló kimenetet kapunk.



4.7. ábra: *A System.Array osztály minden .NET-tömbnek biztosít funkcionalitást*

Figyeljük meg, hogy a `System.Array` sok tagját statikus tagként definiálták, ezért osztályszinten történik a meghívásuk (például az `Array.Sort()` vagy az `Array.Reverse()` metódusok). Az ilyen metódusoknál jön létre a feldolgozni kívánt tömb átadása. A `System.Array` más metódusai (pl. a `GetUpperBound()` metódus vagy a `Length` tulajdonság) objektumszintűek, ezért a tagot közvetlenül a tömbön hívhatjuk.

Forráskód A `FunWithArrays` alkalmazás megtalálható a 4 fejezet alkönyvtárában.

Az Enum típus

A .NET-típusrendszer osztályokból, struktúrákból, felsorolásokból, interfészekből és metódusreferenciákból áll (lásd az 1. fejezetet). Kezdjük a *felsorolás* (vagy egyszerűen *enum*) szerepének vizsgálatával úgy, hogy létrehozunk egy parancssorialkalmazás-projektet `FunWithEnums` névvel. Rendszer építésekor gyakori kényelmes megoldás olyan szimbolikus nevek készletét megalkotni, amelyek leképezhetők az ismert numerikus értékekre. Ha például létrehozunk egy bérjegyzékrendszert, akkor az alkalmazott-típusokra olyan konstansok segítségével hivatkozhatunk, mint alelnök, menedzser, beszállító és dolgozó. A C# ezért támogatja az egyedi felsorolást. Nézzünk például egy `EmpType` nevű felsorolást:

```
// Egyedi felsorolás.
enum EmpType
{
  Manager, // = 0
  Grunt, // = 1
  Contractor, // = 2
  VicePresident // = 3
}
```

Az `EmpType` felsorolás négy megnevezett konstanst határoz meg, megfeleltetve egy diszkrét numerikus értéknek. Az alapértelmezés szerint az első elem értéke nulla (0), amelyet mindig eggyel nagyobb érték követ. A kezdőértéket nyugodtan meg lehet változtatni. Ha például az `EmpType` tagjainak számát 102-től 105-ig szeretnénk beállítani, ezt a következőképpen tehetjük meg:

```
// A kezdőérték legyen 102.
enum EmpType
{
  Manager = 102,
  Grunt,          // = 103
  Contractor,     // = 104
  VicePresident   // = 105
}
```

A felsorolásoknak nem feltétlenül kell szekvenciális sorrendet követniük, és nem kell egyedi értékekkel rendelkezniük. Ha (valamilyen oknál fogva) az alábbiak szerint hozzuk létre az `EmpType` típust, a fordítónak akkor sem lesz problémája:

```
// A felsorolás elemeinek nem kell szekvenciális sorrendet követni!
enum EmpType
{
  Manager = 10,
  Grunt = 1,
  Contractor = 100,
  VicePresident = 9
}
```

Az enum alapjául szolgáló tároló vezérlése

Az alapértelmezés szerint a felsorolás értékeinek tartalmazására szolgáló tároló típus egy `System.Int32` (a C# `int`); ám ezt is nyugodtan módosíthatjuk. A C#-felsorolások bármilyen alapvető rendszertípushoz (`byte`, `short`, `int` vagy `long`) hasonlóan definiálhatók. Ha például az `EmpType` alapjául szolgáló értéktárolót `byte`-ra állítanánk `int` helyett, akkor a következőt írhatnánk:

```
// Az EmpType alapja a mögöttes byte értéke.
enum EmpType : byte
{
  Manager = 10,
  Grunt = 1,
  Contractor = 100,
  VicePresident = 9
}
```

A felsorolás alapjául szolgáló típus megváltoztatása akkor lehet hasznos, ha olyan .NET-alkalmazást készítünk, amelyet kevés memóriával rendelkező eszközre szánunk (pl. egy .NET-képes mobiltelefonra vagy PDA-ra), és minden lehetséges helyen spórolunk a memóriával. Természetesen, ha úgy hozzuk létre a felsorolást, hogy tárolóként `byte`-ot használjon, akkor minden értéknek a tartományon belül kell lennie.

Felsorolások deklarálása és használata

Amint létrehoztuk a felsorolás tartományát és tárolótípusát, használhatjuk az úgynevezett „bűvös" számok helyett. Mivel a felsorolások csupán a felhasználó által meghatározott típusok, használhatjuk őket függvény-visszatérési értékként, metódusparaméterként, lokális változóként stb. Tegyük fel, hogy van egy `AskForBonus()` metódusunk, amely egy `EmpType` változót kap egyedüli paraméterként. A bejövő paraméter értékére alapozva kiíratunk egy megfelelő választ a jutalom kérésére:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("**** Fun with Enums *****");
    // Szerződéses alkalmazott típus készítése.
    EmpType emp = EmpType.Contractor;
    AskForBonus(emp);
    Console.ReadLine();
  }

  // Felsorolások használata paraméterként.
  static void AskForBonus(EmpType e)
  {
    switch (e)
    {
      case EmpType.Manager:
        Console.WriteLine("How about stock options instead?");
      break;
      case EmpType.Grunt:
        Console.WriteLine("You have got to be kidding...");
      break;
      case EmpType.Contractor:
        Console.WriteLine("You already get enough cash...");
      break;
      case EmpType.VicePresident:
        Console.WriteLine("VERY GOOD, Sir!");
      break;
    }
  }
}
```

Figyeljük meg, hogy amikor hozzárendelünk egy értéket egy `enum` változóhoz, akkor az `enum` nevet (`EmpType`) az érték (`Grunt`) hatókörébe kell utalni. Mivel a felsorolások fix név/érték párok, ezért egy `enum` változót nem szabad olyan értékre állítani, amelyet nem határoz meg közvetlenül a felsorolt típus:

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

  // Hiba! A SalesManager nincs benne az EmpType felsorolásban!
  EmpType emp = EmpType.SalesManager;

  // Hiba! A Grunt értéket nem utaltuk az EmpType hatókörébe!
  emp= Grunt;
  Console.ReadLine();
}
```

A System.Enum típus

A .NET-felsorolások érdekessége, hogy funkcionalitásukat a `System.Enum` osztálytípustól szerzik. Ez az osztály olyan metódusokat definiál, amelyek lehetővé teszik egy adott felsorolás lekérdezését és átalakítását. Hasznos metódus a statikus `Enum.GetUnderlyingType()`, amely – ahogy a neve is mutatja – visszaadja a felsorolt típus értékeinek tárolására használatos adattípust (`System.Byte` a jelenlegi `EmpType` deklaráció esetében).

```
static void Main(string[] args)
{
  Console.WriteLine("**** Fun with Enums *****");
  // Szerződéses alkalmazott típus készítése.
  EmpType emp = EmpType.Contractor;
  AskForBonus(emp);

  // A felsorolás tárolására használt típus kiírása.
  Console.WriteLine("EmpType uses a {0} for storage",
    Enum.GetUnderlyingType(emp.GetType()));
  Console.ReadLine();
}
```

A Visual Studio 2008 objektumböngészőjével ellenőrizhetjük, hogy az `Enum.GetUnderlyingType()` metódus első paraméterként a `System.Type` átadását követeli meg. A `Type` egy adott .NET-entitás metaadat-leírását jelképezi (lásd a 16. fejezetben).

A metaadatok megszerzésének egyik lehetséges módja (ahogy korábban már volt róla szó) a `GetType()` metódus használata, amely közös a .NET-alaposztálykönyvtárak minden típusánál. Egy másik megközelítés a C# `typeof` operátor használata. Ennek az az előnye, hogy nem szükséges annak az entitásnak a változójával rendelkeznünk, amelynek szeretnénk megszerezni a metaadat-leírását:

```
// A Type metaadat-leírás beolvasása a typeof operátorral.
Console.WriteLine("EmpType uses a {0} for storage",
  Enum.GetUnderlyingType(typeof(EmpType)));
```

Felsorolás név/érték párjainak dinamikus feltárása

Az `Enum.GetUnderlyingType()` metódus mellett minden C#-felsorolás támogatja a `ToString()` nevű metódust, amely visszaadja az aktuális felsorolás értékének sztringnevét, például:

```
static void Main(string[] args)
{
  Console.WriteLine("**** Fun with Enums *****");
  EmpType emp = EmpType.Contractor;

  // A következőket írja ki: "emp is a Contractor".
  Console.WriteLine("emp is a {0}.", emp.ToString());
  Console.ReadLine();
}
```

Ha név helyett az adott felsorolási változó értékét szeretnénk kideríteni, akkor egyszerűen kasztoljuk az `enum` változót az alap-tárolótípussal szemben. Például:

```
static void Main(string[] args)
{
  Console.WriteLine("**** Fun with Enums *****");
  EmpType emp = EmpType.Contractor;

// A következőket írja ki: "Contractor = 100".
  Console.WriteLine("{0} = {1}", emp.ToString(), (byte)emp);
  Console.ReadLine();
}
```

Megjegyzés A statikus `Enum.Format()` metódus finomabb formázási szintet biztosít a kívánt formázási kapcsoló megadásával. A `System.Enum.Format()` metódus részletezését a .NET Framework 3.5 SDK dokumentációjában találjuk meg.

A `System.Enum` egy másik statikus metódust is definiál, amelynek `GetValues()` a neve. Ez a metódus a `System.Array` egy példányát adja vissza. A tömb minden eleme megfelel az adott felsorolás egy tagjának. Figyeljük meg az alábbi metódust, amely kiírja minden, paraméterként átadott felsorolás név/érték párját:

```
// A metódus a felsorolások részleteit írja ki.
static void EvaluateEnum(System.Enum e)
{
  Console.WriteLine("=> Information about {0}", e.GetType().Name);

  Console.WriteLine("Underlying storage type: {0}",
    Enum.GetUnderlyingType(e.GetType()));

  // A bejövő paraméter név/érték párjainak beolvasása.
  Array enumData = Enum.GetValues(e.GetType());
  Console.WriteLine("This enum has {0} members.", enumData.Length);
```

```
    // A sztring nevének és a kapcsolódó értékének kiírása.
    for(int i = 0; i < enumData.Length; i++)
    {
      Console.WriteLine("Name: {0}, Value: {0:D}",
        enumData.GetValue(i));
    }
    Console.WriteLine();
}
```

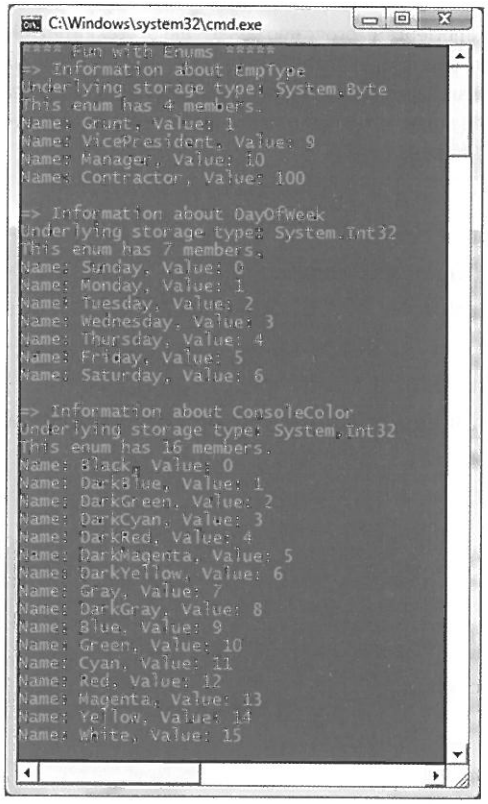
Az új metódus teszteléséhez módosítsuk úgy a `Main()` metódust, hogy létrehozza a `System` névtérben deklarált felsorolástípusok változóit (és ráadásul egy `EmpType` felsorolást is), például:

```
static void Main(string[] args)
{
  Console.WriteLine("**** Fun with Enums *****");
  EmpType e2;

  // Ezek a típusok felsorolások a System névtérben.
  DayOfWeek day;
  ConsoleColor cc;

  EvaluateEnum(e2);
  EvaluateEnum(day);
  EvaluateEnum(cc);
  Console.ReadLine();
}
```

A kimenet a 4.8. ábrán látható.

A felsorolások széles körben használatosak a .NET-alaposztálykönyvtárakban. Az ADO.NET például számos felsorolást használ, hogy megjelenítse egy adatbázis-kapcsolat állapotát (megnyitott, bezárt stb.), a `DataTable` egy sorának állapotát (módosított, új, lecsatlakoztatott stb.), és így tovább. Így ha felsorolást használunk, a `System.Enum` tagjait használva dolgozhatunk a név/érték párokkal.

Forráskód A `FunWithEnums` projekt megtalálható a 4. fejezet alkönyvtárában.



4.8. ábra: *A felsorolástípusok név/érték párjának dinamikus feltárása*

A struktúratípus

Miután megismerkedtünk a felsorolástípusok szerepével, vizsgáljuk meg a .NET-*struktúrák* (vagy egyszerűen *structs*) használatát. A struktúratípusok kiválóan alkalmasak matematikai, geometriai és egyéb „elemi" entitások modellezésére az alkalmazásunkban. A struktúra (mint ahogy a felsorolás) a felhasználó által definiált típus; a struktúrák azonban nem egyszerűen kulcs/érték párok gyűjteményei, hanem olyan típusok, amelyek bármennyi adatmezőt tartalmazhatnak, továbbá olyan tagokat, amelyek ezeken a mezőkön dolgoznak.

A struktúrák emellett meghatározhatnak konstruktorokat, implementálhatnak interfészeket, valamint tartalmazhatnak bármennyi tulajdonságot, metódust, eseményt és túlterhelt operátort.

Megjegyzés Ha már foglalkoztunk objektumorientált programozással, akkor tekinthetjük „egyszerű osztálytípusnak" a struktúrát, hiszen lehetőséget biztosít magába foglalást támogató típus meghatározására, nem használható azonban kapcsolódó típusok családjának létrehozására (ugyanis a struktúrák implicit módon lezártak). Amikor öröklés révén építjük a kapcsolódó típusok családját, akkor az osztálytípusokat kell használnunk.

Első ránézésre nagyon könnyűnek tűnhet a struktúrák definiálása és használata, de ahogy mondani szokás, az ördög a részletekben lakozik. A struktúratípusok megértéséhez elsőként hozzunk létre egy új projektet `FunWithStructures` néven. C#-ban a struktúrákat a `struct` kulcsszó használatával lehet kialakítani. Definiáljunk egy olyan új struktúrát `Point` néven, amely két, int típusú tagváltozót határoz meg, valamint egy olyan metóduskészletet, amely a megadott adatokkal dolgozik:

```
struct Point
{
  // A struktúra mezői.
  public int X;
  public int Y;

  // Adjunk 1-et az (X, Y) pozícióhoz.
  public void Increment()
  {
    X++; Y++;
  }

  // Vonjunk le 1-et az (X, Y) pozícióból.
  public void Decrement()
  {
    X--; Y--;
  }

  // Jelenítsük meg az aktuális pozíciót.
  public void Display()
  {
    Console.WriteLine("X = {0}, Y = {1}", X, Y);
  }
}
```

A `public` kulcsszó használatával, amely egy hozzáférés-szabályozó (lásd részletesen a következő fejezetben), meghatároztunk két egész adatot (X és Y). `Public` kulcsszóval történő deklarálásuk biztosítja, hogy a hívó közvetlenül hozzáfér az adatokhoz egy adott `Point` változóból (a pont operátor révén).