Reiter István

C#

jegyzet

devPortal

Előszó

Általában ez az a rész, ahol a szerző bemutatkozik, kifejti a motivációit illetve köszönetet mond a környezete segítségéért. Nem fogok nagy meglepetést okozni, ez most is így lesz.
Az elmúlt két évben, mióta a jegyzet létezik rengeteg levelet kaptam, különféle témában. Egy közös pont viszont mindegyikben volt: a levélírók egy idősebb emberre számítottak, számos esetben tanárnak gondoltak. Ez alapvetően nem zavar, sőt jól esik, hiszen ez is bizonyítja, hogy sikerült egy „érett", mindenki számára emészthető könyvet készítenem. Most viszont - abból az alkalomból, hogy a jegyzet életében ekkora esemény történt, úgy érzem ideje „hivatalosan" bemutatkoznom:
Reiter István vagyok, 24 éves programozó. Bő tíz éve foglalkozom informatikával, az utóbbi hatot pedig már a „sötét" oldalon töltöttem. Elsődlegesen (vastag)kliens oldalra specializálódtam ez ebben a pillanatban a WPF/Silverlight kettőst jelenti - bár előbbi közelebb áll a szívemhez. Jelenleg - munka mellett - az ELTE Programtervező Informatikus szakán folytatok tanulmányokat.
2008-ban elindítottam szakmai blogomat a „régi" msPortal-on - ez ma a devPortal akadémiai szekciójaként szolgál - és ekkor született meg bennem egy kisebb dokumentáció terve, amely összefoglalná, hogy mit kell a C# nyelvről tudni. Elkezdtem írni, de az anyag egyre csak nőtt, terebélyesedett és végül megszületett a jegyzet első százegynéhány oldalas változata. A pozitív fogadtatás miatt folytattam az írást és néhány hónap után az eredeti, viszonylag összeszedett jegyzetből egy óriási, kaotikus, éppenhogy használható „massza" keletkezett.
Ez volt az a pont, ahol lélekben feladtam az egészet, nem volt kedvem, motivációm rendberakni.
Eltelt több mint fél év, megérkezett 2010 és elhatároztam, hogy - Újévi fogadalom gyanánt - feltámasztom a „szörnyeteget". Az eredeti jegyzet túl sokat akart, ezért úgy döntöttem, hogy kiemelem az alapokat - ez gyakorlatilag a legelső változat - és azt bővítem ki. Ez olyannyira jól sikerült, hogy közel háromszoros terjedelmet sikerült elérnem a kiinduláshoz képest.
Már csak egy dologgal tartozom, köszönetet kell mondjak a következőknek:

- Mindenkinek aki az elmúlt két évben tanácsokkal, kiegészítésekkel, javításokkal látott el.
- Mindenkinek aki elolvasta vagy el fogja olvasni ezt a könyvet, remélem tetszeni fog.
- A devPortal közösségének
- A Microsoft Magyarországnak

Tartalomjegyzék

1 Bevezető

Napjainkban egyre nagyobb teret nyer a .NET Framework és egyik fő nyelve a C#. Ez a jegyzet abból a célból született, hogy megismertesse az olvasóval ezt a nagyszerű technológiát.

A jegyzet a C# 2.0, 3.0 és 4.0 verziójával foglalkozik, az utóbbi kettő által bevezetett új eszközöket az adott rész külön jelöli. Néhány fejezet feltételez olyan tudást, amely alapját egy későbbi rész képezi, ezért ne essen kétségbe a kedves olvasó, ha valamit nem ért, egyszerűen olvasson tovább és térjen vissza a kérdéses anyaghoz, ha rátalált a válaszra. A jegyzet megértéséhez nem szükséges programozni tudni, viszont alapvető informatikai ismeretek (pl. számrendszerek) jól jönnek.

A jegyzethez tartozó forráskódok letölthetőek a következő webhelyről:

http://cid-283edaac5ecc7e07.skydrive.live.com/browse.aspx/Nyilv%C3%A1nos/Jegyzet

Bármilyen kérést, javaslatot és hibajavítást szívesen várok a reiteristvan@gmail.com e-mail címre.

1.1 A jegyzet jelölései

Forráskód: szürke alapon, bekeretezve

Megjegyzés: fehér alapon, bekeretezve

```
Parancssor: fekete alapon, keret nélkül
```

1.2 Jogi feltételek

2 Microsoft .NET Framework

A kilencvenes évek közepén a Sun MicroSystems kiadta a Java platform első nyilvános változatát. Az addigi programnyelvek/platformok különböző okokból nem tudták felvenni a Java –val a versenyt, így számtalan fejlesztő döntött úgy, hogy a kényelmesebb és sokoldalúbb Java –t választja.

Részben a piac visszaszerzésének érdekében a Microsoft a kilencvenes évek végén elindította a Next Generation Windows Services fedőnevű projektet, amelyből aztán megszületett a .NET, amely a kissé elavult és nehézkesen programozható COM platformot hívatott leváltani (ettől függetlenül a COM ma is létező viszonylag népszerű eszköz – ez főleg a hatalmas szoftverbázisnak köszönhető, minden Windows rendszer részét képezi és számos .NET könyvtár is épít rá).

2.1 A .NET platform

Maga a .NET platform a Microsoft, a Hewlett Packard, az Intel és mások közreműködésével megfogalmazott CLI (Common Language Infrastructure) egy implementációja. A CLI egy szabályrendszer, amely maga is több részre oszlik:

- A CTS (Common Type System) az adatok kezelését, a memóriában való megjelenést, az egymással való interakciót, stb. írja le.
- A CLS (Common Language Specification) a CLI kompatibilis nyelvekkel kapcsolatos elvárásokat tartalmazza.
- A VES (Virtual Execution System) a futási környezetet specifikálja, nevezik CLR -nek (Common Language Runtime) is.

Általános tévhit, hogy a VES/CLR –t virtuális gépként azonosítják. Ez abból a szintén téves elképzelésből alakult ki, hogy a .NET ugyanaz, mint a Java, csak Microsoft köntösben. A valóságban nincs .NET virtuális gép, helyette ún. felügyelt (vagy managed) kódot használ, vagyis a program teljes mértékben natív módon, közvetlenül a processzoron fut, mellette pedig ott a keretrendszer, amely felelős pl. a memóriafoglalásért vagy a kivételek kezeléséért.

A .NET nem egy programozási nyelv, hanem egy környezet. Gyakorlatilag bármelyik programozási nyelvnek lehet .NET implementációja. Jelenleg kb. 50 nyelvnek létezik hivatalosan .NET megfelelője, nem beszélve a számtalan hobbifejlesztésről.

2.1.1 MSIL/CIL

A “hagyományos” programnyelveken – mint pl. a C++ – megírt programok ún. natív kódra fordulnak le, vagyis a processzor számára – kis túlzással – azonnal értelmezhetőek.
A .NET (akárcsak a Java) más úton jár, a fordító először egy köztes nyelvre (Intermediate Language) fordítja le a forráskódot. Ez a nyelv a .NET világában az

MSIL, illetve a szabványosítás után a CIL (**M***ICRO***S***OFT*/**C***ommon***IL**) – különbség csak az elnevezésben van.

Jogos a kérdés, hogy a két módszer közül melyik a jobb? Ha nagy általánosságban beszélünk, akkor a válasz az, hogy nincs köztük különbség. Igaz, hogy a natív nyelvek hardver-közelibbek és emiatt gyorsabbak tudnak lenni, viszont ez több hibalehetőséggel is jár, amelyek elkerülése a felügyelt környezetben kiegyenlíti az esélyeket.
Bizonyos területeken viszont egyik vagy másik megközelítés jelentős eltérést eredményezhet. Jó példa a számítógépes grafika ahol a natív nyelvek vannak előnyben pont azért, mert az ilyen számításigényes feladathoz minden csepp erőforrást ki kell préselni a hardverből. Másfelől a felügyelt környezet a hatékonyabb memóriakezelés miatt jobban teljesít olyan helyzetekben ahol nagy mennyiségű adatot mozgatunk a memórián belül (pl. számos rendező algoritmus ilyen).

2.1.2 Fordítás és futtatás

A natív programok ún. gépi kódra fordulnak le, míg a .NET forráskódokból egy CIL nyelvű futtatható állomány keletkezik. Ez a kód a feltelepített .NET Framework –nek szóló utasításokat tartalmaz. Amikor futtatjuk ezeket az állományokat, először az ún. JIT (**J**ust–**I**n–**T**ime) fordító veszi kezelésbe, lefordítja őket gépi kódra, amit a processzor már képes kezelni.

Amikor "először" fordítjuk le a programunkat, akkor egy ún. *Assembly* (vagy szerelvény) keletkezik. Ez tartalmazza a felhasznált, illetve megvalósított típusok adatait (ez az ún. *Metadata*) amelyek a futtató környezetnek szolgálnak információval (pl. osztályok szerkezete, metódusai, stb.). Egy Assembly egy vagy több fileból is állhat, tipikusan .exe (futtatható állomány) vagy .dll (osztálykönyvtár) kiterjesztéssel.

2.1.3 BCL

A .NET Framework telepítésével a számítógépre kerül – többek között – a *BCL* (**B**ase **C**lass **L**ibrary), ami az alapvető feladatok (file olvasás/ írás, adatbázis kezelés, adatszerkezetek, stb…) elvégzéséhez szükséges eszközöket tartalmazza. Az összes többi könyvtár (*ADO.NET*, *WCF*, stb…) ezekre épül.

2.2 A C# programozási nyelv

A C# (ejtsd: szí-sárp) a Visual Basic mellett a .NET fő programozási nyelve. 1999 –ben Anders Hejlsberg vezetésével kezdték meg a fejlesztését.

A C# tisztán objektumorientált, típus biztos, általános felhasználású nyelv. A tervezésénél a lehető legnagyobb produktivitás elérését tartották szem előtt. A nyelv elméletileg platform független (létezik Linux és Mac fordító is), de napjainkban a legnagyobb hatékonyságot a Microsoft implementációja biztosítja.

2.3 Alternatív megoldások

A Microsoft .NET Framework jelen pillanatban csak és kizárólag Microsoft Windows operációs rendszerek alatt elérhető. Ugyanakkor a szabványosítás után a CLI specifikáció nyilvános és bárki számára elérhető lett, ezen ismeretek birtokában pedig több független csapat vagy cég is létrehozta a saját CLI implementációját, bár eddig még nem sikerült teljes mértékben reprodukálni az eredetit. Ezen céljukat nehezíti, hogy a Microsoft időközben számos, a specifikációban nem szereplő változtatást végzett a keretrendszeren.

A "hivatalosnak" tekinthető ECMA szabvány nem feltétlenül tekinthető tökéletes útmutatónak a keretrendszer megértéséhez, néhol jelentős eltérések vannak a valósághoz képest. Ehelyett ajánlott a C# nyelv fejlesztői által készített C# referencia, amely – bár nem elsősorban a .NET –hez készült – értékes információkat tartalmaz.

2.3.1 SSCLI

Az *SSCLI* (**S**hared **S**ource **C**ommon **L**anguage **I**nfrastructure) vagy korábbi nevén *Rotor* a Microsoft által fejlesztett nyílt forrású, keresztplatformos változata a .NET Frameworknek (tehát nem az eredeti lebutított változata). Az SSCLI Windows, FreeBSD és Mac OSX rendszereken fut.

Az SSCLI –t kimondottan tanulási célra készítette a Microsoft, ezért a liszensze engedélyez mindenfajta módosítást, egyedül a piaci értékesítést tiltja meg. Ez a rendszer nem szolgáltatja az eredeti keretrendszer teljes funkcionalitását, jelen pillanatban valamivel a .NET 2.0 mögött jár.

Az SSCLI projekt jelen pillanatban leállni látszik. Ettől függetlenül a forráskód és a hozzá tartozó dokumentációk rendelkezésre állnak, letölthetőek a következő webhelyről:

http://www.microsoft.com/downloads/details.aspx?FamilyId=8C09FD61-3F26-4555-AE17-3121B4F51D4D&displaylang=en

2.3.2 Mono

A Mono projekt szülőatyja Miguel de Icaza, 2000 –ben kezdte meg a fejlesztést és egy évvel később mutatta be ez első kezdetleges C# fordítót. A Ximian (amelyet Icaza és Nat Friedman alapított) felkarolta az ötletet és 2001 júliusában hivatalosan is elkezdődött a Mono fejlesztése. 2003 –ban a Novell felvásárolta a Ximian –t, az 1.0 verzió már Novell termékként készült el egy évvel később.

A Mono elérhető Windows, Linux, UNIX, BSD, Mac OSX és Solaris rendszereken is. Napjainkban a Mono mutatja a legígéretesebb fejlődést, mint a Microsoft .NET

jövőbeli “ellenfele”, illetve keresztplatformos társa. A Mono emblémája egy majmot ábrázol, a szó ugyanis spanyolul majmot jelent.

A Mono hivatalos oldala:*http://www.mono-project.com/Main_Page*

2.3.3 DotGNU

A DotGNU a GNU projekt része, amelynek célja egy ingyenes és nyílt alternatívát nyújtani a Microsoft implementáció helyett. Ez a projekt – szemben a Mono –val – nem a Microsoft BCL –lel való kompatibilitást helyezi előtérbe, hanem az eredeti szabvány pontos és tökéletes implementációjának a létrehozását. A DotGNU saját CLI megvalósításának a Portable .NET nevet adta. A jegyzet írásának idején a projekt leállni látszik.

A DotGNU hivatalos oldala: *http://www.gnu.org/software/dotgnu/*

3 “Hello C#!”

A híres “Hello World!” program elsőként Dennis Ritchie és Brian Kerningham “A C programozási nyelv” című könyvében jelent meg és azóta szinte hagyomány, hogy egy programozási nyelv bevezetőjeként ezt a programot mutatják be.

Mi itt most nem a világot, hanem a C# nyelvet üdvözöljük, ezért ennek megfelelően módosítsuk a forráskódot:

```
using System;

class HelloWorld
{
      static public void Main()
      {
            Console.WriteLine("Hello C#!");
            Console.ReadKey();
      }
}
```

Mielőtt lefordítjuk, tegyünk pár lépést a parancssorból való fordítás elősegítésére. Ahhoz, hogy így le tudjunk fordítani egy forrásfilet, vagy meg kell adnunk a fordítóprogram teljes elérési útját (ez a mi esetünkben elég hosszú) vagy a fordítóprogram könyvtárát fel kell venni a *PATH* környezeti változóba.

Utóbbi lelőhelye: Vezérlőpult/Rendszer -> Speciális fül/Környezeti változók. A rendszerváltozók listájából keressük ki a Path –t és kattintsunk a Szerkesztés gombra. Most nyissuk meg a Sajátgépet, C: meghajtó, Windows mappa, azon belül Microsoft.NET/Framework. Nyissuk meg vagy a v2.0…, a v3.5... stb. kezdetű mappát (attól függően, hogy a C# fordító melyik verziójára van szükségünk). Másoljuk ki a címsorból ezt a szép hosszú elérést, majd menjünk vissza a Path –hoz. A változó értékének sorában navigáljunk el a végére, írjunk egy pontosvesszőt (;) és illesszük be az elérési utat. Nyomjuk meg az OK gombot és kész is vagyunk. Ha van megnyitva konzol vagy PowerShell, azt indítsuk újra és írjuk be, hogy *csc*. Azt kell látnunk,hogy:

```
Microsoft ® Visual C# 2008 Compiler Version 3.5 …
```

(Az évszám és verzió változhat, ez itt most a C# 3.0 üzenete.)

Most már fordíthatunk a

```
csc filenév.cs
```

paranccsal. Természetesen a szöveges file kiterjesztése *.txt*, ezért nevezzük is át, mivel a C# forráskódot tartalmazó fileok kiterjesztése: *.cs*

Nézzük, hogy mit is tettünk: az első sor megmondja a fordítónak, hogy használja a *System* névteret. Ezután létrehozunk egy osztályt – mivel a C# teljesen objektum-orientált –, ezért utasítást csak osztályon belül adhatunk meg. A “HelloWorld” osztályon belül definiálunk egy *Main* nevű statikus függvényt, ami a programunk

belépési pontja lesz. Minden egyes C# program a *Main* függvénnyel kezdődik, ezt mindenképpen létre kell hoznunk. Végül meghívjuk a *Console* osztályban lévő *WriteLine* és *ReadKey* függvényeket. Előbbi kiírja a képernyőre a paraméterét, utóbbi vár egy billentyű leütésére.

Ebben a bekezdésben szerepel néhány (sok) kifejezés, amik ismeretlenek lehetnek, de a jegyzet későbbi fejezeteiben mindenre fény derül majd.

3.1 A C# szintaktikája

Amikor egy programozási nyelv szintaktikájáról beszélünk, akkor azokra a szabályokra gondolunk, amelyek megszabják a forráskód felépítését. Ez azért fontos, mert az egyes fordítóprogramok csak ezekkel a szabályokkal létrehozott kódot tudják értelmezni.

A C# úgynevezett C-stílusú szintaxissal rendelkezik (azaz a C programozási nyelv szintaxisát veszi alapul), ez három fontos szabályt von maga után:

- Az egyes utasítások végén pontosvessző (**;**) áll
- A kis- és nagybetűk különböző jelentőséggel bírnak, azaz a “program” és “Program” azonosítók különböznek. Ha a fenti kódban *Console.WriteLine* helyett *console.writeline* –t írtunk volna, akkor a program nem fordulna le.
- A program egységeit (osztályok, metódusok, stb.) ún. blokkokkal jelöljük ki, kapcsos zárójelek (**{** és **}**) segítségével.

3.1.1 Kulcsszavak

Szinte minden programnyelv definiál kulcsszavakat, amelyek speciális jelentőséggel bírnak a fordító számára. Ezeket az azonosítókat a saját meghatározott jelentésükön kívül nem lehet másra használni, ellenkező esetben a fordító hibát jelez. Vegyünk például egy változót, aminek az “int” nevet akarjuk adni. Az “int” egy beépített típus a neve is, azaz kulcsszó, tehát nem fog lefordulni a program.

```
int int;//hiba
```

A legtöbb fejlesztőeszköz beszínezi a kulcsszavakat (is), ezért könnyű elkerülni a fenti hibát.

A C# 77 kulcsszót ismer:

abstract	default	foreach	object	Sizeof	unsafe
as	delegate	goto	operator	stackalloc	ushort
base	do	If	out	Static	using
bool	double	implicit	override	String	virtual
break	else	In	params	Struct	volatile
byte	enum	int	private	Switch	void
case	event	interface	protected	This	while
catch	explicit	internal	public	Throw	
char	extern	Is	readonly	True	
checked	false	lock	ref	Try	
class	finally	long	return	Typeof	
const	fixed	namespace	sbyte	Uint	
continue	float	new	sealed	Ulong	
decimal	for	null	short	unchecked	

Ezeken kívül létezik még 23 azonosító, amelyeket a nyelv nem tart fenn speciális használatra, de különleges jelentéssel bírnak. Amennyiben lehetséges, kerüljük a használatukat “hagyományos” változók, metódusok, osztályok létrehozásánál:

add	equals	group	let	Remove	var
ascending	from	in	on	Select	where
by	get	into	orderby	Set	yield
descending	global	join	partial	Value	

Néhányuk a környezettől függően más-más jelentéssel is bírhat, a megfelelő fejezet bővebb információt ad majd ezekről az esetekről.

3.1.2 Megjegyzések

A forráskódba megjegyzéseket tehetünk. Ezzel egyrészt üzeneteket hagyhatunk (pl. egy metódus leírása) magunknak vagy a többi fejlesztőnek, másrészt a kommentek segítségével dokumentációt tudunk generálni, ami szintén az első célt szolgálja, csak éppen élvezhetőbb formában.

Megjegyzéseket a következőképpen hagyhatunk:

```
using System;

class HelloWorld
{
      static public void Main()
      {
            Console.WriteLine("Hello C#"); // Ez egy egysoros komment
            Console.ReadKey();
            /*
                  Ez egy többsoros komment
            */
      }
}
```

Az egysoros komment a saját sora legvégéig tart, míg a többsoros a “/*” és “*/” párokon belül érvényes. Utóbbiakat nem lehet egymásba ágyazni:

```
/*
        /* */
*/
```

Ez a “kód” nem fordul le.

A kommenteket a fordító nem veszi figyelembe, tulajdonképpen a fordítóprogram első lépése, hogy a forráskódból eltávolít minden megjegyzést.

3.2 Névterek

A .NET Framework osztálykönyvtárai szerény becslés szerint is legalább tízezer nevet, azonosítót tartalmaznak. Ilyen nagyságrenddel elkerülhetetlen, hogy a nevek ne ismétlődjenek. Ekkor egyrészt nehéz eligazodni közöttük, másrészt a fordító sem tudná, mikor mire gondolunk. Ennek a problémának a kiküszöbölésére hozták létre a névterek fogalmát. Egy névtér tulajdonképpen egy virtuális doboz, amelyben a logikailag összefüggő osztályok, metódusok, stb. vannak. Nyilván könnyebb megtalálni az adatbázis-kezeléshez szükséges osztályokat, ha valamilyen kifejező nevű névtérben vannak (pl.*System.Data*).

Névteret magunk is definiálhatunk, a *namespace* kulcsszóval:

```
namespace MyNameSpace
{

}
```

Ezután a névtérre vagy a program elején a *using* kulcsszóval, vagy az azonosító elé írt teljes eléréssel hivatkozhatunk:

```
using MyNameSpace;

//vagy

MyNameSpace.Valami
```

A jegyzet első felében főleg a *System* névteret fogjuk használni.

4 Változók

Amikor programot írunk, akkor szükség lehet tárolókra, ahová az adatainkat ideiglenesen eltároljuk. Ezeket a tárolókat változóknak nevezzük.

A változók a memória egy (vagy több) cellájára hivatkozó leírók. Egy változót a következő módon hozhatunk létre C# nyelven:

Típus változónév;

A változónév első karaktere csak betű vagy alulvonás jel (_) lehet, a többi karakter szám is. Lehetőleg kerüljük az ékezetes karakterek használatát.

Konvenció szerint a változónevek kisbetűvel kezdődnek. Amennyiben a változónév több szóból áll, akkor célszerű azokat a szóhatárnál nagybetűvel “elválasztani” (pl. pirosAlma, vanSapkaRajta, stb.).

4.1 Deklaráció és definíció

Egy változó (illetve lényegében minden objektum) életciklusában megkülönböztetünk deklarációt és definíciót. A deklarációnak tartalmaznia kell a típust és azonosítót, a definícióban pedig megadjuk az objektum értékét. Értelemszerűen a deklaráció és a definíció egyszerre is megtörténhet.

```
int x; // deklaráció
x = 10; // definíció
int y = 11; // delaráció és definíció
```

4.2 Típusok

A C# erősen (statikusan) típusos nyelv, ami azt jelenti, hogy minden egyes változó típusának ismertnek kell lennie fordítási időben, ezzel biztosítva azt, hogy a program pontosan csak olyan műveletet hajthat végre amire valóban képes. A típus határozza meg, hogy egy változó milyen értékeket tartalmazhat, illetve mekkora helyet foglal a memóriában.

A következő táblázat a C# beépített típusait tartalmazza, mellettük ott a .NET megfelelőjük, a méretük és egy rövid leírás:

C# típus	.NET típus	Méret (byte)	Leírás
byte	*System.Byte*	1	Előjel nélküli 8 bites egész szám (0..255)
char	*System.Char*	1	Egy Unicode karakter
bool	*System.Boolean*	1	Logikai típus, értéke igaz(1 vagy true) vagy hamis(0 vagy false)
sbyte	*System.SByte*	1	Előjeles, 8 bites egész szám (-128..127)
short	*System.Int16*	2	Előjeles, 16 bites egész szám (-32768..32767
ushort	*System.Uint16*	2	Előjel nélküli, 16 bites egész szám (0..65535)
int	*System.Int32*	4	Előjeles, 32 bites egész szám (–2147483647.. 2147483647).
uint	*System.Uint32*	4	Előjel nélküli, 32 bites egész szám (0..4294967295)
float	*System.Single*	4	Egyszeres pontosságú lebegőpontos szám
double	*System.Double*	8	Kétszeres pontosságú lebegőpontos szám
decimal	*System.Decimal*	8	Fix pontosságú 28+1 jegyű szám
long	*System.Int64*	8	Előjeles, 64 bites egész szám
ulong	*System.Uint64*	8	Előjel nélküli, 64 bites egész szám
string	*System.String*	N/A	Unicode karakterek szekvenciája
object	*System.Object*	N/A	Minden más típus őse

A forráskódban teljesen mindegy, hogy a “rendes” vagy a .NET néven hivatkozunk egy típusra.

Alakítsuk át a “Hello C#” programot úgy, hogy a kiírandó szöveget egy változóba tesszük:

```
using System;

class HelloWorld
{
      static public void Main()
      {
            //string típusú változó, benne a kiírandó szöveg
            string message="Hello C#";
            Console.WriteLine(message);
            Console.ReadKey();
      }
}
```

A C# 3.0 már lehetővé teszi, hogy egy metódus hatókörében deklarált változó típusának meghatározását a fordítóra bízzuk. Általában olyankor tesszük ezt, amikor hosszú típusnévről van szó, vagy nehéz meghatározni a típust. Ezt az akciót a *var* kulcsszóval kivitelezhetjük.
Ez természetesen nem jelenti azt, hogy úgy használhatjuk a nyelvet, mint egy típustalan környezetet! Abban a pillanatban, amikor értéket rendeltünk a változóhoz

(ráadásul ezt azonnal meg is kell tennünk!), az úgy fog viselkedni, mint az ekvivalens típus. Az ilyen változók típusa nem változtatható meg, de a megfelelő típuskonverziók végrehajthatóak.

```
int x = 10; // int típusú változó
var z = 10; // int típusú változó
z = "string"; // fordítási hiba
var w; //fordítási hiba
```

4.3 Lokális és globális változók

Egy blokkon belül deklarált változó lokális lesz a blokkjára nézve, vagyis a program többi részéből nem látható (úgy is mondhatjuk, hogy a változó hatóköre a blokkjára terjed ki). A fenti példában a *message* egy lokális változó, ha egy másik függvényből vagy osztályból próbálnánk meg elérni, akkor a program nem fordulna le.

Globális változónak azokat az objektumokat nevezzük, amelyek a program bármely részéből elérhetőek. A C# nem rendelkezik a más nyelvekből ismerős globális változóval, mivel deklarációt csak osztályon belül végezhetünk. Áthidalhatjuk a helyzetet statikus változók használatával, erről később szó lesz.

4.4 Referencia- és értéktípusok

A .NET minden típus direkt vagy indirekt módon a *System.Object* nevű típusból származik, és ezen belül szétoszlik érték- és referencia-típusokra (egyetlen kivétel a pointer típus, amelynek semmiféle köze sincs a *System.Object*-hez). A kettő közötti különbség leginkább a memóriában való elhelyezkedésben jelenik meg.

A CLR két helyre tud adatokat pakolni, az egyik a verem (*stack*), a másik a halom (*heap*). A *stack* egy ún. LIFO (last-in-first-out) adattár, vagyis a legutoljára berakott elem lesz a tetején, kivenni pedig csak a mindenkori legfelső elemet tudjuk. A *heap* nem adatszerkezet, hanem a program által lefoglalt nyers memória, amit a CLR tetszés szerint használhat. Minden művelet a *stack*-et használja, pl. ha össze akarunk adni két számot akkor a CLR lerakja mindkettőt a *stack*-be és meghívja a megfelelő utasítást. Ezután kiveszi a verem legfelső két elemét, összeadja őket, majd a végeredményt visszateszi a *stack*-be:

```
int x=10;
int y=11;
x + y
```

A verem:

|11**|**
|10**| --→összeadás művelet--→|**21**|**

A referencia-típusok minden esetben a halomban jönnek létre, mert ezek összetett adatszerkezetek és így hatékony a kezelésük. Az értéktípusok vagy a *stack*-ben vagy a *heap*-ben vannak attól függően, hogy hol deklaráltuk őket.

Metóduson belül, lokálisan deklarált értéktípusok a *stack*-be kerülnek, a referencia-típuson belül adattagként deklarált értéktípusok pedig a *heap*-ben foglalnak helyet.

Nézzünk néhány példát!

```
using System;

class Program
{
	static public void Main()
	{
		int x = 10;
	}
}
```

Ebben a "programban" *x*–et lokálisan deklaráltuk egy metóduson belül, ezért biztosak lehetünk benne, hogy a verembe fog kerülni.

```
class MyClass
{
	private int x = 10;
}
```

Most *x* egy referencia-típuson (esetünkben egy osztályon) belüli adattag, ezért a halomban foglal majd helyet.

```
class MyClass
{
	private int x = 10;

	public void MyMethod()
	{
		int y = 10;
	}
}
```

Most egy kicsit bonyolultabb a helyzet. Az *y* nevű változót egy referencia-típuson belül, de egy metódusban, lokálisan deklaráltuk, így a veremben fog tárolódni, *x* pedig még mindig adattag, ezért marad a halomban.

Végül nézzük meg, hogy mi lesz érték- és mi referencia-típus: értéktípus lesz az összes olyan objektum, amelyeket a következő típusokkal deklarálunk:

- Az összes beépített numerikus típus (*int, byte, double, stb*.)
- A felsorolt típus (*enum*)
- Logikai típus (*bool*)
- Karakter típus (*char*)
- Struktúrák (*struct*)

Referencia-típusok lesznek a következők:

- Osztályok (*class*)
- Interfész típusok (*interface*)
- Delegate típusok (*delegate*)
- Stringek

- Minden olyan típus, amely *közvetlen* módon származik a *System.Object*–ből vagy bármely *class* kulcsszóval bevezetett szerkezetből.

4.5 Referenciák

Az érték- illetve referencia-típusok közötti különbség egy másik aspektusa az, ahogyan a forráskódban hivatkozunk rájuk. Vegyük a következő kódot:

```
int x = 10;
int y = x;
```

Az első sorban létrehoztuk az *x* nevű változót, a másodikban pedig egy új változónak adtuk értékül *x*–et. A kérdés az, hogy *y* hova mutat a memóriában: oda ahol *x* van, vagy egy teljesen más területre?

Amikor egy értéktípusra hivatkozunk, akkor ténylegesen az értékét használjuk fel, vagyis a kérdésünkre a válasz az, hogy a két változó értéke egyenlő lesz, de nem ugyanazon a memóriaterületen helyezkednek el, tehát *y* máshova mutat, teljesen önálló változó.

A helyzet más lesz referencia-típusok esetében. Mivel ők összetett típusok, ezért fizikailag lehetetlen lenne az értékeikkel dolgozni, ezért egy referencia-típusként létrehozott változó tulajdonképpen a memóriának arra a szeletére mutat, ahol az objektum ténylegesen helyet foglal. Nézzük meg ezt közelebbről:

```
using System;

class MyClass
{
      public int x;
}

class Program
{
      static public void Main()
      {
            MyClasss = new MyClass();
            s.x = 10;

            MyClassp = s;
            p.x = 14;

            Console.WriteLine(s.x);
      }
}
```

Vajon mit fog kiírni a program?

Kezdjük az elejéről! Hasonló a felállás, mint az előző forráskódnál, viszont amikor a második változónak értékül adjuk az elsőt, akkor az történik, hogy a *p* nevű referencia ugyanarra a memóriaterületre hivatkozik majd, mint az *s*, vagyis tulajdonképpen *s*-nek egy álneve (*alias*) lesz. Értelemszerűen, ha *p* módosul, akkor *s* is így tesz, ezért a fenti program kimenete 14 lesz.

4.6 Boxing és unboxing

Boxing–nak (bedobozolás) azt a folyamatot nevezzük, amely megengedi egy értéktípusnak, hogy úgy viselkedjen, mint egy referencia-típus. Korábban azt mondtuk, hogy minden típus közvetlenül vagy indirekt módon a *System.Object* –ből származik. Az értéktípusok esetében az utóbbi teljesül, ami egy igen speciális helyzetet jelent. Az értéktípusok alapvetően nem származnak az *Object*–ből, mivel így hatékony a kezelésük, nem tartozik hozzájuk semmiféle "túlsúly" (elméletileg akár az is előfordulhatna ilyenkor, hogy a referencia-típusokhoz "adott" extrák (sync blokk, metódustábla, stb...) több helyet foglalnának, mint a tényleges adat).
Hogy miért van ez így, azt nagyon egyszerű kitalálni: az értéktípusok egyszerű típusok amelyek kis mennyiségű adatot tartalmaznak, ezenkívül ezeket a típusokat különösen gyakran fogjuk használni, ezért elengedhetetlen, hogy a lehető leggyorsabban kezelhessük őket.
A probléma az, hogy az értéktípusoknak a fentiektől függetlenül illeszkedniük kell a típusrendszerbe, vagyis tudnunk kell úgy kezelni őket, mint egy referenciatípust és itt jön képbe a boxing művelet. Nézzünk egy példát: az eddig használt *Console.WriteLine* metódus deklarációja így néz ki:

```
public static void WriteLine(
      Object value
)
```

Látható, hogy a paraméter típusa *object*, leánykori nevén *System.Object* más szóval egy referencia-típus. Mi történik vajon, ha egy *int* típusú változót (egy értéktípust) akarunk így kiírni? A *WriteLine* metódus minden típust úgy ír ki, hogy meghívja rajtuk a *ToString* metódust, amely visszaadja az adott típus *string*-alakját. A baj az, hogy a *ToString*–et a *System.Object* deklarálja, ilyen módon a referencia-típusok mind rendelkeznek vele, de az értéktípusok már nem. Még nagyobb baj, hogy a *ToString* hívásához a sima *object*–ként meghatározott változóknak elő kell keríteniük a tárolt objektum valódi típusát, ami a *GetType* metódussal történik – amelyet szintén a *System.Object* deklarál – ami nem is lenne önmagában probléma, de az értéktípusok nem tárolnak magukról típusinformációt épp a kis méret miatt.
A megoldást a boxing művelet jelenti, ami a következőképpen működik: a rendszer előkészít a halmon egy – az értéktípus valódi típusának megfelelő – keretet (dobozt) amely tartalmazza az eredeti változó adatait, illetve az összes szükséges információt ahhoz, hogy referencia-típusként tudjon működni – lényegében az is.

Első ránézésre azt gondolná az ember, hogy a dobozolás rendkívül drága mulatság, de ez nem feltétlenül van így. A valóságban a fordító képes úgy optimalizálni a végeredményt, hogy nagyon kevés hátrányunk származzon ebből a műveletből, néhány esetben pedig nagyjából ugyanazt a teljesítményt érjük el, mint referencia-típusok esetében.

Vegyük észre, hogy az eddigi *WriteLine* hívásoknál a "konverzió" kérés nélkül – azaz implicit módon – működött annak ellenére, hogy érték- és referencia-típusok között nincs szoros reláció. Az ilyen kapcsolatot implicit konverzábilis kapcsolatnak

nevezzük és nem tévesztendő össze a polimorfizmussal (hamarosan), bár nagyon hasonlónak látszanak.

A következő forráskód azt mutatja, hogy miként tudunk “kézzel” dobozolni:

```
int x = 10;
object boxObject = x; // bedobozolva
Console.WriteLine("X értéke: {0}", boxObject);
```

Itt ugyanaz történik, mintha rögtön az *x* változót adnánk át a metódusnak csak éppen egy lépéssel hamarabb elkészítettük *x* referencia-típus klónját.

Az unboxing (vagy kidobozolás) a boxing ellentéte, vagyis a bedobozolt értéktípusunkból kinyerjük az eredeti értékét:

```
int x = 0;
object obj = x; // bedobozolva
int y = (int)obj; // kidobozolva
```

Az *object* típusú változón explicit típuskonverziót hajtottunk végre (erről hamarosan), így visszakaptuk az eredeti értéket.

A kidobozolás szintén érdekes folyamat: logikusan gondolkodva azt hinnénk, hogy most minden fordítva történik, mint a bedobozolásnál, vagyis a vermen elkészítünk egy új értéktípust és átmásoljuk az értékeket. Ezt majdnem teljesen igaz egyetlen apró kivétellel: amikor vissza akarjuk kapni a bedobozolt értéktípusunkat az unbox IL utasítást hívjuk meg, amely egy ún. value-type-pointert ad vissza, amely a halomra másolt és bedobozolt értéktípusra mutat. Ezt a címet azonban nem használhatjuk közvetlenül a verembe másoláshoz, ehelyett az adatok egy ideiglenes vermen létrehozott objektumba másolódnak majd onnan egy újabb másolás művelettel a számára kijelölt helyre vándorolnak.
A kettős másolás pazarlásnak tűnhet, de ez egyrészt megkerülhetetlen szabály másrészt a JIT ezt is képes úgy optimalizálni, hogy ne legyen nagy teljesítményveszteség.

Fontos még megjegyezni, hogy a bedobozolás után teljesen új objektum keletkezik, amelynek semmi köze az eredetihez:

```
using System;

class Program
{
	static public void Main()
	{
		int x = 10;
		object z = x;
		z = (int)z + 10;

		Console.WriteLine(x);
		Console.WriteLine(z);
	}
}
```

A kimenet 10 illetve 20 lesz. Vegyük észre azt is, hogy *z* –n konverziót kellett végrehajtanunk az összeadáshoz, de az értékadáshoz nem (először kidobozoltuk, összeadtuk a két számot, majd az eredményt visszadobozoltuk).

4.7 Konstansok

A *const* típusmódosító kulcsszó segítségével egy objektumot konstanssá, megváltoztathatatlanná tehetünk. A konstansoknak egyetlen egyszer adhatunk (és ekkor kötelező is adnunk) értéket, mégpedig a deklarációnál. Bármely későbbi próbálkozás fordítási hibát okoz.

```
const int x; // Hiba
const int x = 10; // Ez jó
x = 11; // Hiba
```

A konstans változóknak adott értéket/kifejezést fordítási időben ki kell tudnia értékelni a fordítónak. A következő forráskód éppen ezért nem is fog lefordulni:

```
using System;

class Program
{
	static public void Main()
	{
		Console.WriteLine("Adjon meg egy számot: ");
		const int x = int.Parse(Console.ReadLine());
	}
}
```

A *Console.ReadLine* metódus egy sort olvas be a standard bemenetről (ez alapértelmezés szerint a konzol lesz, de megváltoztatható), amelyet terminálló karakterrel (Carriage Return, Line Feed, stb.), pl. az Enter-rel zárunk.
A metódus egy *string* típusú értékkel tér vissza, amelyből ki kell nyernünk a felhasználó által megadott számot. Erre fogjuk használni az *int.Parse* metódust, ami paraméterként egy *string*et vár, és egész számot ad vissza. A paraméterként megadott karaktersor nem tartalmazhat numerikus karakteren kívül mást, ellenkező esetben a program kivételt dob.

4.8 A felsorolt típus

A felsorolt típus olyan adatszerkezet, amely meghatározott értékek névvel ellátott halmazát képviseli. Felsorolt típust az *enum* kulcsszó segítségével deklarálunk:

```
enum Animal { Cat, Dog, Tiger, Wolf };
```

Ezután így használhatjuk:

```
Animal b = Animal.Tiger;

if(b == Animal.Tiger) // Ha b egy tigris
{
      Console.WriteLine("b egy tigris...");
}
```

Enum típust csakis metóduson kívül (osztályon belül, vagy "önálló" típusként) deklarálhatunk, ellenkező esetben a program nem fordul le:

```
using System;

class Program
{
      static public void Main()
      {
            enum Animal { Cat = 1, Dog = 3, Tiger, Wolf }
      }
}
```

Ez a kód hibás! Nézzük a javított változatot:

```
using System;

class Program
{
      enum Animal { Cat = 1, Dog = 3, Tiger, Wolf }

      static public void Main()
      {

      }
}
```

Most már jó lesz (és akkor is lefordulna, ha a *Program* osztályon kívül deklarálnánk). A felsorolás minden tagjának megfeleltethetünk egy egész (numerikus) értéket. Ha mást nem adunk meg, akkor alapértelmezés szerint a számozás nullától kezdődik és deklaráció szerinti sorrendben (értsd: balról jobbra) eggyel növekszik. Ezen a módon az *enum* objektumokon explicit konverziót hajthatunk végre a megfelelő numerikus értékre:

```
enum Animal { Cat, Dog, Tiger, Wolf }

Animal a = Animal.Cat;
int x = (int)a; // x == 0
a = Animal.Wolf;
x = (int)a; // x == 3
```

A tagok értékei alapértelmezetten *int* típusúak, ezen változtathatunk:

```
enum Animal : byte { Cat, Dog, Tiger, Wolf };
```

Természetesen ez együtt jár azzal, hogy a tagok értékének az adott típus értékhatárai között kell maradniuk, vagyis a példában egy tag értéke nem lehet több mint 255.

Ilyen módon csakis a beépített egész numerikus típusokat használhatjuk (pl. *byte*, *long*, *uint*, stb...)

Azok az "nevek" amelyekhez nem rendeltünk értéket implicit módon, az őket megelőző név értékétől számítva kapják meg azt, növekvő sorrendben. Így a lenti példában Tiger értéke négy lesz:

```
using System;

class Program
{
      enum Animal { Cat = 1, Dog = 3, Tiger, Wolf }

      static public void Main()
      {
            Animal a = Animal.Tiger;

            Console.WriteLine((int)a);
      }
}
```

Az *Enum.TryParse* metódussal *string* értékekből "gyárthatunk" *enum* értékeket:

```
using System;

class Program
{
      enum Animal { Cat = 1, Dog = 3, Tiger, Wolf }

      static public void Main()
      {
            string s1 = "1";
            string s2 = "Dog";

            Animal a1, a2;
            Enum.TryParse(s1, true, out a1);
            Enum.TryParse(s2, true, out a2);
      }
}
```

4.9 Null típusok

A referencia-típusok az inicializálás előtt automatikusan nullértéket vesznek fel, illetve mi magunk is megjelölhetjük őket "beállítatlannak":

```
class RefType{ }
RefType rt = null;
```

Ugyanez az értéktípusoknál már nem működik:

```
int x = null; // ez le sem fordul
```

Azt már tudjuk, hogy a referencia-típusokra referenciákkal, azaz a nekik megfelelő memóriacímmel mutatunk, ezért lehetséges null értéket megadni nekik. Az

értéktípusok pedig az általuk tárolt adatot reprezentálják, ezért ők nem vehetnek fel null értéket.

Ahhoz, hogy meg tudjuk állapítani, hogy egy értéktípus még nem inicializált, egy speciális típust, a nullable típust kell használnunk, amit a "rendes" típus után írt kérdőjellel jelzünk:

```
int? i = null; // ez már működik
```

Egy nullable típusra való konverzió implicit módon (külön kérés nélkül) megy végbe, míg az ellenkező irányban explicit konverzióra lesz szükségünk (vagyis ezt tudatnunk kell a fordítóval):

```
int y = 10;
int? x = y; // implicit konverzió
y = (int)x; // explicit konverzió
```

4.10A dinamikus típus

Ennek a fejezetnek a teljes megértéséhez szükség van az osztályok, illetve metódusok fogalmára, ezeket egy későbbi fejezetben találja meg az olvasó.

A C# 3.0–ig bezárólag minden változó és objektum statikusan típusos volt, vagyis egyrészt a típust fordításkor meg kellett tudnia határozni a fordítónak, másrészt ez futási idő alatt nem változhatott meg.

A C# 4.0 bevezeti a *dynamic* kulcsszót, amely használatával dinamikusan típusossá tehetünk objektumokat. Mit is jelent ez a gyakorlatban? Lényegében azt, hogy minden *dynamic*–cal jelölt objektum bármit megtehet fordítási időben, még olyan dolgokat is, amelyek futásidejű hibát okozhatnának. Ezenkívül az összes ilyen „objektum" futásidőben megváltoztathatja a típusát is:

```
using System;

class Program
{
      static public void Main()
      {
            dynamic x = 10;
            Console.WriteLine(x); // x most 10

            x = "szalámi";
            Console.WriteLine(x); // x most szalámi
      }
}
```

Vegyük a következő osztályt:

```
class Test
{
	public void Method(string s)
	{

	}
}
```

Ha a fenti metódust meg akarjuk hívni, akkor meg kell adnunk számára egy string típusú paramétert is. Kivéve, ha a *dynamic*–ot használjuk:

```
static public void Main()
{
	dynamic t = new Test();
	t.Method(); // ez lefordul
}
```

A fenti forráskód minden további nélkül lefordul, viszont futni nem fog.

A konstruktorok nem tartoznak az „átverhető” metódusok közé, akár használtuk a deklarációnál a *dynamic*–ot, akár nem. A paramétereket minden esetben kötelező megadnunk, ellenkező esetben a program nem fordul le.

Bár a fenti tesztek „szórakoztatóak”, valójában nem túl hasznosak. A *dynamic* „hagyományos” objektumokon való használata lényegében nemcsak átláthatatlanná teszi a kódot, de komoly teljesítményproblémákat is okozhat, ezért mindenképpen kerüljük el az ilyen szituációkat!

A dinamikus típusok igazi haszna a más programnyelvekkel – különösen a script alapú nyelvekkel – való együttműködésben rejlik. A *dynamic* kulcsszó mögött egy komoly platform, a **D**ynamic **L**anguage **R**untime (DLR) áll (természetesen a dynamic mellett jó néhány osztály is helyett kapott a csomagban). A DLR olyan „típustalan”, azaz gyengén típusos nyelvekkel tud együttműködni, mint a Lua, JavaScript, PHP, Python vagy Ruby.

5 Operátorok

Amikor programozunk, utasításokat adunk a számítógépnek. Ezek az utasítások kifejezésekből állnak, a kifejezések pedig operátorokból és operandusokból, illetve ezek kombinációjából jönnek létre:

```
i = x + y;
```

Ebben az utasításban *i*–nek értékül adjuk *x* és *y* összegét. Két kifejezés is van az utasításban:

1 lépés: x + y –> ezt az értéket jelöljük * -al

2 lépés: i = * –> i –nek értékül adjuk a * -ot

Az első esetben *x* és *y* operandusok, a '**+**' jel pedig az összeadás művelet operátora. Ugyanígy a második pontban *i* és * (vagyis *x* + *y*) az operandusok, az értékadás művelet ('**=**') pedig az operátor.

Egy operátornak nem csak két operandusa lehet. A C# nyelv egy- (unáris) és három-operandusú (ternáris) operátorokkal is rendelkezik.

A következő néhány fejezetben átveszünk néhány operátort, de nem az összeset. Ennek oka az, hogy bizonyos operátorok önmagukban nem hordoznak jelentést, egy - egy speciális részterület kapcsolódik hozzájuk, ezért ezeket az operátorokat majd a megfelelő helyen ismerjük meg (pl. az indexelő operátor most kimarad, elsőként a tömböknél találkozhat majd vele a kedves olvasó).

5.1 Operátor precedencia

Amikor több operátor is szerepel egy kifejezésben, a fordítónak muszáj valamilyen sorrendet (precedenciát) felállítani közöttük, hiszen az eredmény ettől is függhet.

Például:

```
10 * 5 + 1
```

Ennél a kifejezésnél, sorrendtől függően az eredmény lehet 51 vagy 60. A jó megoldás az előbbi, az operátorok végrehajtásának sorrendjében a szorzás és az osztás előnyt élvez (természetesen érvényesülnek a matematikai szabályok). A legelső sorrendi helyen szerepelnek pl. a zárójeles kifejezések, utolsón pedig az értékadó operátor. Ha bizonytalanok vagyunk a végrehajtás sorrendjében, akkor mindig használjunk zárójeleket, ez a végleges programra nézve semmilyen hatással nincs (és a forráskód olvashatóságát is javítja).

A fenti kifejezés tehát így nézne ki:

```
(10 * 5) + 1
```

A C# nyelv precedencia szerint 14 kategóriába sorolja az operátorokat (a kisebb sorszámút fogja a fordító hamarabb kiértékelni):

1. Zárójel, adattag hozzáférés (pont (‘**.**’) operátor), metódushívás, postfix inkrementáló és dekrementáló operátorok, a *new* operátor, *typeof*, *sizeof*, *checked és unchecked*
2. Pozitív és negatív operátorok (x = **-**5), logika- és bináris tagadás, prefix inkrementáló és dekrementáló operátorok, explicit típuskonverzió
3. Szorzás, maradékos és maradék nélküli osztás
4. Összeadás, kivonás
5. Bit-eltoló (**>>** és **<<**) operátorok
6. Kisebb (vagy egyenlő), nagyobb (vagy egyenlő), *as*, *is*
7. Egyenlő és nem egyenlő operátorok
8. Logikai ÉS
9. Logikai XOR
10. Logikai VAGY
11. Feltételes ÉS
12. Feltételes VAGY
13. Feltételes operátor (? :)
14. Értékadó operátor, illetve a “rövid formában” használt operátorok (pl: x +=y)

5.2 Értékadó operátor

Az egyik legáltalánosabb művelet, amit elvégezhetünk az az, hogy egy változónak értéket adunk. A C# nyelvben ezt az egyenlőségjel segítségével tehetjük meg:

```
int x = 10;
```

Létrehoztunk egy *int* típusú változót, elneveztük *x* –nek, majd kezdőértékének 10–et adtunk. Természetesen nem kötelező a deklarációnál megadni a definíciót (amikor meghatározzuk, hogy a változó milyen értéket kapjon), ezt el lehet halasztani:

```
int x;
x = 10;
```

Ettől függetlenül a legtöbb esetben ajánlott akkor értéket adni egy változónak, amikor deklaráljuk (persze ez inkább csak esztétikai kérdés, a fordító lesz annyira okos, hogy ugyanazt generálja le a fenti két kódrészletből).

Egy változónak nem csak konstans értéket, de egy másik változót is értékül adhatunk, de csakis abban az esetben, ha a két változó azonos típusú, illetve ha létezik megfelelő konverzió (a típuskonverziókkal egy későbbi fejezet foglalkozik).

```
int x = 10;
int y = x; // y értéke most 10
```

5.3 Matematikai operátorok

A következő példában a matematikai operátorok használatát vizsgáljuk meg:

```
using System;

public class Operators
{
	static public void Main()
	{
		int x = 10;
		int y = 3;
		int z = x + y; // Összeadás: z = 10 + 3
		Console.WriteLine(z); // Kiírja az eredményt: 13
		z = x - y; // Kivonás: z = 10 - 3
		Console.WriteLine(z); // 7
		z = x * y; //Szorzás: z = 10 * 3
		Console.WriteLine(z);//30
		z = x / y; // Maradék nélküli osztás: z = 10 / 3;
		Console.WriteLine(z); // 3
		z = x % y; // Maradékos osztás: z = 10 % 3
		Console.WriteLine(z); // Az osztás maradékát írja ki: 1
		Console.ReadKey(); //Vár egy billentyű leütésére
	}
}
```

5.4 Relációs operátorok

A relációs operátorok segítségével egy adott értékkészlet elemei közötti viszonyt tudjuk lekérdezni. Relációs operátort használó műveletek eredménye vagy igaz (*true*) vagy hamis (*false*) lesz. A numerikus típusokon értelmezve van egy rendezés reláció:

```
using System;

public class RelOp
{
	static public void Main()
	{
		int x = 10;
		int y = 23;

		Console.WriteLine(x > y); // Kiírja az eredményt: false
		Console.WriteLine(x == y); // false
		Console.WriteLine(x != y); // x nem egyenlő y -al: true
		Console.WriteLine(x <= y); // x kisebb-egyenlő mint y: true
	}
}
```

Az első sor egyértelmű, a másodikban az egyenlőséget vizsgáljuk a kettős egyenlőségjellel. Ilyen esetekben figyelni kell, mert egy elütés is nehezen kideríthető

hibát okoz, amikor egyenlőség helyett az értékadó operátort használjuk. Az esetek többségében ugyanis így is le fog fordulni a program, működni viszont valószínűleg rosszul fog.
A relációs operátorok összefoglalása:

x > y	x nagyobb, mint y
x >= y	x nagyobb vagy egyenlő, mint y
x < y	x kisebb, mint y
x <= y	x kisebb vagy egyenlő, mint y
x == y	x egyenlő y-nal
x != y	x nem egyenlő y-nal

5.5 Logikai és feltételes operátorok

Akárcsak a C++, a C# sem rendelkezik „igazi” logikai típussal, helyette 1 és 0 jelzi az igaz és hamis értékeket:

```
using System;

public class RelOp
{
      static public void Main()
      {
            bool l = true;
            bool k = false;

            if(l == true && k == false)
            {
                  Console.WriteLine("Igaz");
            }
      }
}
```

Először felvettünk két logikai (*bool*) változót, az elsőnek „igaz” a másodiknak „hamis” értéket adtunk. Ezután egy elágazás következik, erről bővebben egy későbbi fejezetben lehet olvasni, a lényege az, hogy ha a feltétel igaz, akkor végrehajt egy utasítást (vagy utasításokat). A fenti példában az „ÉS” (*&&*) operátort használtuk, ez két operandust vár és akkor ad vissza „igaz” értéket, ha mindkét operandusa „igaz” vagy nullánál nagyobb értéket képvisel. Ebből következik az is, hogy akár az előző fejezetben megismert relációs operátorokból felépített kifejezések, vagy matematikai formulák is lehetnek operandusok. A program nem sok mindent tesz, csak kiírja, hogy „Igaz”.

Nézzük az „ÉS” igazságtáblázatát:

A	B	Eredmény
hamis	hamis	hamis
hamis	igaz	hamis
igaz	hamis	hamis
igaz	igaz	igaz

A fenti forráskód jó gyakorlás az operátor-precedenciához, az elágazás feltételében először az egyenlőséget fogjuk vizsgálni (a hetes számú kategória) és csak utána a feltételes ÉS –t (tizenegyes kategória).

A második operátor a „VAGY”:

```
using System;

public class RelOp
{
	static public void Main()
	{
		bool l = true;
		bool k = false;

		if(l == true || k == true)
		{
			Console.WriteLine("Igaz");
		}
	}
}
```

A „vagy” (**||**) operátor akkor térít vissza „igaz” értéket, ha az operandusai közül valamelyik „igaz” vagy nagyobb, mint nulla. Ez a program is ugyanazt csinálja, mint az előző, a különbség a feltételben van. Látható, hogy *k* biztosan nem „igaz” (hiszen éppen előtte kapott „hamis” értéket).
A „VAGY” igazságtáblázata:

A	B	Eredmény
hamis	hamis	hamis
hamis	igaz	igaz
igaz	hamis	igaz
igaz	igaz	igaz

Az eredmény kiértékelése az ún. „lusta kiértékelés” (vagy „rövidzár”) módszerével történik, azaz a program csak addig vizsgálja a feltételt, amíg muszáj. Tudni kell azt is, hogy a kiértékelés mindig balról jobbra halad, ezért pl. a fenti példában *k* soha nem fog kiértékelődni, mert *l* van az első helyen, és mivel ő „igaz” értéket képvisel, ezért a feltétel is biztosan teljesül.

A harmadik a „tagadás” (**!**):

```
using System;

public class RelOp
{
	static public void Main()
	{
		int x = 10;

		if(!(x == 11)) // x nem 11, ezért false, de ezt tagadjuk: true
		{
			Console.WriteLine("X nem egyenlő 11 -gyel!");
		}
	}
}
```

Ennek az operátornak egy operandusa van és akkor ad vissza igaz értéket, ha az operandusban megfogalmazott feltétel hamis, vagy – ha numerikus kifejezésről beszélünk - egyenlő nullával.

A „tagadás” (negáció) igazságtáblája:

A	Eredmény
hamis	igaz
igaz	hamis

Ez a három operátor ún. feltételes operátor, közülük pedig az „ÉS” és a „VAGY” operátoroknak létezik „csonkolt” logikai párja is. A különbség annyi, hogy a logikai operátorok az eredménytől függetlenül kiértékelik a teljes kifejezést, nem élnek a lusta kiértékeléssel.

A logikai „VAGY” művelet:

```
if(l == true | k == true)
{
      Console.WriteLine("Igaz");
}
```

A logikai „ÉS”:

```
if(l == true & k == true)
{
      Console.WriteLine("Igaz");
}
```

A logikai operátorok családjához tartozik (ha nem is szorosan) a feltételes operátor. Ez az egyetlen háromoperandusú operátor és a következőképpen működik:

feltétel **?** *igaz-ág* **:** *hamis-ág*;

```
using System;

public class RelOp
{
      static public void Main()
      {
            int x = 10;
            int y = 10;

            Console.WriteLine((x == y) ? "Egyenlő" : "Nem egyenlő");
      }
}
```

Az operátor úgy működik, hogy a kérdőjel előtti kifejezést kiértékeli, majd megnézi, hogy a kifejezés igaz vagy hamis. Ha igaz, akkor az egyenlőség jel utáni érték lesz a teljes kifejezésünk értéke, ha pedig hamis, akkor pedig a kettőspont utáni. Egyszerű *if-else* (erről később) ágakat lehet ezzel a módszerrel kiváltani, sok gépelést megspórolhatunk vele és a kódunk is kompaktabb, áttekinthetőbb lesz tőle.

5.6 Bit operátorok

Az előző fejezetben említett logikai operátorok bitenkénti műveletek elvégzésére is alkalmasak. A számítógép az adatokat kettes számrendszerbeli alakban tárolja, így például, ha van egy *byte* típusú változónk (ami egy byte, azaz 8 bit hosszúságú), aminek a „2” értéket adjuk, akkor az a következőképpen jelenik meg a memóriában:

2 → 00000010

A bit operátorok ezzel a formával dolgoznak. Az eddig megismert kettő mellé még jön négy másik operátor is. A műveletek:

Bitenkénti „ÉS”: veszi a két operandus bináris alakját és a megfelelő bitpárokon elvégzi az „ÉS” műveletet, azaz ha mindkét bit *1* állásban van, akkor az adott helyen az eredményben is az lesz, egyébként pedig *0*:

```
01101101
00010001 AND
00000001
```

Elég egyértelmű, hogy az „ÉS” igazságtáblát használtuk az eredmény kiszámolásához.

Példa:

```
using System;

public class Program
{
	static public void Main()
	{
		Console.WriteLine(10 & 2);
		//1010 & 0010 = 0010 = 2
	}
}
```

A programot futtatva látni fogjuk, hogy a *Console.WriteLine* a művelet eredményét tízes számrendszerben írja majd ki.

A bitenkénti „VAGY” hasonlóan működik, mint az „ÉS”, de a végeredményben egy bit értéke akkor lesz 1, ha a két operandus adott bitje közül valamelyik legalább az:

```
01101101
00010001 OR
01111101
```

Példa a „VAGY”-ra:

```
using System;

public class Program
{
	static public void Main()
	{
		Console.WriteLine(10 | 2);
		// 1010 | 0010 = 1010 = 10
	}
}
```

Biteltolás balra: a kettes számrendszerbeli alak „felső” bitjét eltoljuk, majd a jobb oldalon keletkező „üres” bitet nullára állítjuk. Az operátor jele: **<<**.

10001111 LEFT SHIFT
100011110

Példa:

```
using System;

public class Program
{
	static public void Main()
	{
		int x = 143;
		Console.WriteLine(x << 1);
		// 10001111 (=143) << 1 = 100011110 = 286
	}
}
```

Amikor biteltolást végzünk, figyelnünk kell arra, hogy a művelet végeredménye minden esetben 32 bites, előjeles szám (*int*) lesz. Ennek nagyon egyszerű oka az, hogy így biztosítja a .NET, hogy az eredmény elférjen (a fenti példában használt 143 pl. pontosan 8 biten felírható szám, azaz egy byte típusban már nem férne el az eltolás után, hiszen akkor 9 bitre lenne szükségünk).

Biteltolás jobbra: most az alsó bitet toljuk el, és felül pótoljuk a hiányt. Az operátor: **>>**.

Példa:

```
using System;

public class Program
{
	static public void Main()
	{
		byte x = 143;
		Console.WriteLine(x >> 1);
		// 10001111 (=143) >> 1 = 01000111 = 71
	}
}
```

A legtöbb beépített típust könnyen konvertálhatjuk át különböző számrendszerekre a *Convert.ToString(x, y)* metódussal, ahol *x* az az objektum, amit konvertálunk, *y* pedig a cél-számrendszer:

```
using System;

public class Program
{
	static public void Main()
	{
		byte x = 10;
		Console.WriteLine(Convert.ToString(x, 2)); // 1010

		int y = 10;
		Console.WriteLine(Convert.ToString(y, 2)); // 1010

		char z = 'a';
		Console.WriteLine(Convert.ToString(z, 2)); // 1100001
		Console.WriteLine(Convert.ToString(z, 16)); // 61
		Console.WriteLine(Convert.ToString(z, 10)); // 97
	}
}
```

Ez a metódus a konvertálás után csak a „hasznos” részt fogja visszaadni, ezért fog az *int* típusú - egyébként 32 bites - változó csak 4 biten megjelenni (hiszen a 10 egy pontosan 4 biten felírható szám).
A *char* típus numerikus értékre konvertálásakor az Unicode táblában elfoglalt helyét adja vissza. Ez az ’a’ karakter esetében 97 (tízes számrendszer), 1100001 (kettes szr.) vagy 0061 (tizenhatos szr.) lesz. Ez utóbbinál is csak a hasznos részt kapjuk vissza, hiszen a felső nyolc bit itt nullákból áll.

Vegyük észre, hogy amíg a balra való eltolás ténylegesen – fizikailag – hozzátett az eredeti számunkhoz, addig a jobbra tolás elvesz belőle, hiszen a „felülre” érkező nulla bitek nem hasznosulnak az eredmény szempontjából. Értelemszerűen a balra tolás ezért mindig növelni, a jobbra tolás pedig mindig csökkenteni fogja az eredményt.

Ennél is tovább mehetünk, felfedezve a biteltolások valódi hasznát: egy *n* bittel balra tolás megfelel az alapszám 2 az *n*–edik hatványával való szorzásnak:

143 << 1 = 143 * (2^1) = 286
143 << 2 = 143 * (2^2) = 572

Ugyanígy a jobbra tolás ugyanazzal a hatvánnyal oszt (nullára kerekítéssel):

143 >> 1 = 143 / (2^1) =71
143 >> 2 = 143 / (2^2) = 35

Amikor olyan programot készítünk, amely erősen épít kettővel vagy hatványaival való szorzásra/osztásra, akkor ajánlott bitműveleteket használni, mivel ezeket a processzor sokkal gyorsabban végzi el, mint a hagyományos szorzást (tulajdonképpen a szorzás a processzor egyik leglassabb művelete).

5.7 Rövid forma

Vegyük a következő példát:

```
x = x + 10;
```

Az *x* nevű változót megnöveltük tízzel. Csakhogy van egy kis baj: ez a megoldás nem túl hatékony. Mi történik valójában? Elsőként értelmezni kell a jobb oldalt, azaz ki kell értékelni *x*–et, hozzá kell adni tízet és eltárolni a veremben. Ezután ismét kiértékeljük *x*–et, ezúttal a bal oldalon.
Szerencsére van megoldás, mégpedig az ún. rövid forma. A fenti sorból ez lesz:

```
x += 10;
```

Rövidebb, szebb és hatékonyabb. Az összes aritmetikai operátornak létezik rövid formája.

Az igazsághoz azért az is hozzátartozik, hogy a fordítóprogram elvileg felismeri a fent felvázolt szituációt és a rövid formával egyenértékű IL –t készít belőle (más kérdés, hogy a forráskód így viszont szebb és olvashatóbb).

A probléma ugyanaz, de a megoldás más a következő esetben:

```
x = x + 1;
```

Szemmel láthatóan ugyanaz a baj, azonban az eggyel való növelésre/csökkentésre van önálló operátorunk:

++x és **--**x
x**++** és x**--**

Ebből az operátorból rögtön két verziót is kapunk, prefixes (++/-- elől) és postfixes (++/-- hátul) formát. A prefixes alak pontosan azt teszi, amit elvárunk tőle, azaz megnöveli(vagy értelemszerűen csökkenti) az operandusát eggyel.

A postfixes forma egy kicsit bonyolultabb, elsőként létrehoz egy átmeneti változót, amiben eltárolja az operandusa értékét, majd megnöveli eggyel az operandust, végül visszaadja az átmeneti változót. Ez elsőre talán nem tűnik hasznosnak, de vannak helyzetek, amikor lényegesen megkönnyíti az életünket a használata.

Attól függően, hogy növeljük vagy csökkentjük az operandust, inkrementáló illetve dekrementáló operátorról beszélünk. Ez az operátor használható az összes beépített numerikus típuson, valamint a *char* illetve *enum* típusokon is.

5.8 Egyéb operátorok

Unáris ('+' és '-'): az adott szám pozitív illetve negatív értékét jelezzük vele:

```
using System;

public class Program
{
	static public void Main()
	{
		int x = 10;
		int y = 10 + (-x);

		Console.WriteLine(y);
	}
}
```

Ez a program nullát fog kiírni (természetesen érvényesülnek a matematikai szabályok). Ezeket az operátorokat csakis előjeles típusokon használhatjuk, mivel az operátor *int* típussal tér vissza (akkor is, ha pl. *byte* típusra alkalmaztuk). A következő program le sem fordul:

```
using System;

public class Program
{
	static public void Main()
	{
		byte x = 10;
		byte y = -x;
	}
}
```

A *typeof* az operandusa típusát adja vissza:

```
usingSystem;

classProgram
{
	staticpublicvoidMain()
	{
		int x = 143;
		if(typeof(int) == x.GetType())
		{
			Console.WriteLine("x típusa int");
		}
	}
}
```

A változón meghívott *GetType* metódus a változó típusát adja vissza (ez egy *System.Object*–hez tartozó metódus, így a használatához dobozolni kell az objektumot).

A *sizeof* operátor a „paramétereként” megadott értéktípus méretét adja vissza byte – ban. Ez az operátor kizárólag *unsafe* módban használható és csakis értéktípusokon (illetve pointer típusokon):

```
using System;

public class Program
{
	static public void Main()
	{
		unsafe
		{
			Console.WriteLine(sizeof(int));
		}
	}
}
```

Ez a program négyet fog kiírni, hiszen az *int* típus 32 bites, azaz 4 byte méretű típus.

A programot az *unsafe* kapcsolóval kell lefordítanunk:

```
csc /unsafe main.cs
```

6 Vezérlési szerkezetek

Vezérlési szerkezetnek a program utasításainak sorrendiségét szabályozó konstrukciókat nevezzük.

6.1 Szekvencia

A legegyszerűbb vezérlési szerkezet a szekvencia. Ez tulajdonképpen egymás után megszabott sorrendben végrehajtott utasításokból áll.

6.2 Elágazás

Gyakran előfordul, hogy meg kell vizsgálnunk egy állítást, és attól függően, hogy igaz vagy hamis, a programnak más-más utasítást kell végrehajtania. Ilyen esetekben elágazást használunk:

```
using System;

public class Program
{
	static public void Main()
	{
		int x = 10;

		if(x == 10) // Ha x egyenlő 10 -zel
		{
			Console.WriteLine("x értéke 10");
		}
	}
}
```

Természetes az igény arra is, hogy azt a helyzetet is kezelni tudjuk, amikor *x* értéke nem tíz. Ilyenkor használjuk az *else* ágat:

```
using System;

public class Program
{
	static public void Main()
	{
		int x = 11;

		if(x == 10) // Ha x egyenlő 10 -zel
		{
			Console.WriteLine("x értéke 10");
		}
		else // Ha pedig nem
		{
			Console.WriteLine("x értéke nem 10");
		}
	}
}
```

Az *else* szerkezet akkor lép életbe, ha a hozzá kapcsolódó feltétel(ek) nem igaz(ak). Önmagában *else* ág nem állhat (nem is lenne sok értelme).

A fenti helyzetben írhattuk volna ezt is:

```
using System;

public class Program
{
	static public void Main()
	{
		int x = 11;

		if(x == 10) // Ha x egyenlő 10 -zel
		{
			Console.WriteLine("x értéke 10");
		}

		if(x != 10)//Ha pedig x nem 10
		{
			Console.WriteLine("x értéke nem 10");
		}
	}
}
```

Ez a program pontosan ugyanazt csinálja, mint az előző, de van egy nagy különbség a kettő között: mindkét feltételt ki kell értékelnie a programnak, hiszen két különböző szerkezetről beszélünk (ez egyúttal azzal is jár, hogy a feltételtől függően mindkét állítás lehet igaz).

Arra is van lehetőségünk, hogy több feltételt is megvizsgáljunk, ekkor *else-if* –et használunk:

```
using System;

public class Program
{
	static public void Main()
	{
		int x = 13;

		if(x == 10) // Ha x == 10 -zel
		{
			Console.WriteLine("x értéke 10");
		}
		elseif(x == 12) // Vagy 12 -vel
		{
			Console.WriteLine("x értéke 12");
		}
		else // De ha egyik sem
		{
			Console.WriteLine("x értéke nem 10 vagy 12");
		}
	}
}
```

A program az első olyan ágat fogja végrehajtani, amelynek a feltétele teljesül (vagy ha egyik feltétel sem bizonyult igaznak, akkor az *else* ágat – ha adtunk meg ilyet).

Egy elágazásban pontosan egy darab *if,* bármennyi *else-if* és pontosan egy *else* ág lehet. Egy elágazáson belül is írhatunk elágazást.

Az utolsó példában olyan változót vizsgáltunk, amely nagyon sokféle értéket vehet fel. Nyílván ilyenkor nem tudunk minden egyes állapothoz feltételt írni (pontosabban tudunk, csak az nem lesz szép). Ilyen esetekben azonban van egy egyszerűbb és elegánsabb megoldás, mégpedig a *switch-case* szerkezet. Ezt akkor használjuk, ha egy változó több lehetséges állapotát akarjuk vizsgálni:

```
using System;

public class Program
{
	static public void Main()
	{
		int x = 11;

		switch(x)
		{
			case 10:
				Console.WriteLine("x értéke 10");
				break;
			case 11:
				Console.WriteLine("x értéke 11");
				break;
		}
	}
}
```

A *switch* szerkezeten belül megadhatjuk azokat az állapotokat, amelyekre reagálni szeretnénk. Az egyes esetek utasításai után meg kell adnunk, hogy mi történjen ezután. Az egyes ágak a kijelölt feladatuk végrehajtása után a *break* utasítással kilépnek a szerkezetből:

```
using System;

public class Program
{
	enum Animal { TIGER, WOLF, CAT, DOG};

	static public void Main()
	{
		Animal animal = Animal.DOG;

		switch(animal)
		{
			case Animal.TIGER:
				Console.WriteLine("Tigris");
				break;
			default:
				Console.WriteLine("Nem ismerem ezt az állatot!");
				break;
		}
	}
}
```

Újdonságként megjelenik a *default* állapot, ez lényegében az *else* ág testvére lesz, akkor kerül ide a vezérlés, ha a *switch* nem tartalmazza a vizsgált változó állapotát (vagyis a *default* biztosítja, hogy a *switch* egy ága mindenképpen lefusson).

A C++ nyelvtől eltérően a C# nem engedélyezi, hogy *break* utasítás hiányában egyik állapotból átcsússzunk egy másikba. Ez alól a szabály alól egyetlen kivétel, ha az adott ág nem tartalmaz semmilyen utasítást:

```
using System;

public class Program
{
	enum Animal { TIGER, WOLF, CAT, DOG };

	static public void Main()
	{
		Animal animal = Animal.DOG;

		switch(animal)
		{
			case Animal.TIGER:
			case Animal.DOG:
			default:
				Console.WriteLine("Ez egy állat!");
				break;
		}
	}
}
```

A break utasításon kívül használhatjuk a *goto* –t is, ekkor átugrunk a megadott ágra:

```
using System;

public class Program
{
	enum Animal { TIGER, WOLF, CAT, DOG};

	static public void Main()
	{
		Animal animal = Animal.DOG;

		switch(animal)
		{
			case Animal.TIGER:
				goto default;
			case Animal.DOG:
				goto default;
			default:
				Console.WriteLine("Ez egy állat!");
				break;
		}
	}
}
```

6.3 Ciklus

Amikor egy adott utasítássorozatot egymás után többször kell végrehajtanunk, akkor ciklust használunk. A C# négyféle ciklust biztosít számunkra.

Az első az ún. számlálós ciklus (nevezzük *for*-ciklusnak). Nézzük a következő programot:

```
using System;

public class Program
{
	static public void Main()
	{
		for(int i = 0;i < 10;++i)
		{
			Console.WriteLine(i);
		}
	}
}
```

Vajon mit ír ki a program? Mielőtt ezt megmondanám, először inkább nézzük meg azt, hogy mit csinál: a *for* utáni zárójelben találjuk az ún. ciklusfeltételt, ez minden ciklus része lesz, és azt adjuk meg benne, hogy hányszor fusson le a ciklus. A számlálós ciklus feltétele első ránézésre eléggé összetett, de ez ne tévesszen meg minket, valójában nem az. Mindössze három kérdésre kell választ adnunk: Honnan? Meddig? És Hogyan?

Menjünk sorjában: a honnanra adott válaszban megmondjuk azt, hogy milyen típust használunk a számoláshoz és azt, hogy honnan kezdjük a számolást. Tulajdonképpen ebben a lépésben adjuk meg az ún. Ciklusváltozót, amelyre a ciklusfeltétel épül. A fenti példában egy *int* típusú ciklusváltozót hoztunk létre a ciklusfeltételen belül és nulla kezdőértéket adtunk neki.

A ciklusváltozó neve konvenció szerint i lesz az angol iterate – ismétel szóból. Több ciklusváltozó használatakor általában i, j, k ... sorrendet követünk.

Mivel a ciklusfeltétel után blokkot nyitunk, azt hinné az ember, hogy a ciklusváltozó a lokális lesz a ciklus blokkjára nézve, de ez nem fedi a valóságot. A ciklusfeltételen belül deklarált ciklusváltozó lokális lesz a *ciklust tartalmazó blokkra* nézve. Épp ezért a következő forráskód nem fordulna le:

```
using System;

public class Program
{
	static public void Main()
	{
		for(int i = 0;i < 10;++i)
		{
			Console.WriteLine(i);
		}

		int i = 10; // itt a hiba
	}
}
```

Következzen a „Meddig?”! Most azt kell megválaszolnunk, hogy a ciklusváltozó milyen értéket vehet fel, ami kielégíti a ciklusfeltételt. Most azt adtuk meg, hogy *i*-nek kisebbnek kell lennie tíznél, vagyis kilenc még jó, de ha *i* ennél nagyobb, akkor a ciklust be kell fejezni.

Természetesen bonyolultabb kifejezést is megadhatunk:

```
using System;

public class Program
{
	static public void Main()
	{
		for(int i = 1;i < 10 && i != 4;++i)
		{
			Console.WriteLine(i);
		}
	}
}
```

Persze ennek a programnak különösebb értelme nincs, de a ciklusfeltétel érdekesebb. Addig megy a ciklus, amíg *i* kisebb tíznél és nem egyenlő néggyel. Értelemszerűen csak háromig fogja kiírni a számokat, hiszen mire a négyhez ér, a ciklusfeltétel már nem lesz igaz.

Utoljára a „Hogyan?” kérdésre adjuk meg a választ, vagyis azt, hogy milyen módon változtatjuk a ciklusváltozó értékét. A leggyakoribb módszer a példában is látható inkrementáló (dekrementáló) operátor használata, de itt is megadhatunk összetett kifejezést:

```
using System;

public class Program
{
	static public void Main()
	{
		for(int i = 0;i < 10;i += 2)
		{
			Console.WriteLine(i);
		}
	}
}
```

Ebben a kódban kettesével növeljük a ciklusváltozót, vagyis a páros számokat íratjuk ki a képernyőre.

Most már meg tudjuk válaszolni, hogy az első programunk mit csinál: nullától kilencig kiírja a számokat.

A *for* ciklusból tetszés szerint elhagyhatjuk a ciklusfej bármely részét – akár az egészet is, ekkor végtelen ciklust készíthetünk.
Végtelen ciklusnak nevezzük azt a ciklust, amely soha nem ér véget. Ilyen ciklus születhet programozási hibából, de szándékosan is, mivel néha erre is szükségünk lesz.

```
using System;

public class Program
{
	static public void Main()
	{
		for(;;)
		{
			Console.WriteLine("Végtelen ciklus");
		}
	}
}
```

Ez a forráskód lefordul, de a fordítótól figyelmeztetést kapunk (warning), hogy „gyanús” kódot észlelt.
A „program” futását a Ctrl+C billentyűkombinációval állíthatjuk le, ha parancssorból futtattuk.

Második kliensünk az elől-tesztelős ciklus (mostantól hívjuk *while*-ciklusnak), amely onnan kapta a nevét, hogy a ciklusmag végrehajtása előtt ellenőrzi a ciklusfeltételt, ezért előfordulhat az is, hogy a ciklus-törzs egyszer sem fut le:

```
using System;

public class Program
{
	static public void Main()
	{
		int i = 0; // ciklusváltozó deklaráció
		while(i < 10) // ciklusfeltétel: fuss amíg i kisebb, mint 10
		{
			Console.WriteLine("i értéke: {0}", i);
			++i; // ciklusváltozó növelése
		}
	}
}
```

A program ugyanazt csinálja mint az előző, viszont itt jól láthatóan elkülönülnek a ciklusfeltételért felelős utasítások (kezdőérték, ciklusfeltétel, növel/csökkent).

Működését tekintve az elől-tesztelős ciklus hasonlít a számlálósra (mindkettő először a ciklusfeltételt ellenőrzi), de az előbbi sokkal rugalmasabb, mivel több lehetőségünk van a ciklusfeltétel megválasztására.

A változó értékének kiíratásánál a *Console.WriteLine* egy másik verzióját használtuk, amely ún. formátum-sztringet kap paraméterül. Az első paraméterben a kapcsos zárójelek között megadhatjuk, hogy a további paraméterek közül melyiket helyettesítse be a helyére (nullától számozva).

A harmadik versenyző következik, őt hátul-tesztelős ciklusnak hívják (legyen *do-while*), nem nehéz kitalálni, hogy azért kapta ezt a nevet, mert a ciklusmag végrehajtása után ellenőrzi a ciklusfeltételt, így legalább egyszer biztosan lefut:

```
using System;

public class Program
{
	static public void Main()
	{
		int i = 0;
		do
		{
			Console.WriteLine("i értéke: {0}", i);
			++i;
		}while(i < 10);
	}
}
```

Végül – de nem utolsósorban – a *foreach* (neki nincs külön neve) ciklus következik. Ezzel a ciklussal végigiterálhatunk egy tömbön vagy gyűjteményen, illetve minden olyan objektumon, ami megvalósítja az *IEnumerable* és *IEnumerator* interfészeket (interfészekről egy későbbi fejezet fog beszámolni, ott lesz szó erről a kettőről is).

A példánk most nem a már megszokott „számoljunk el kilencig” lesz, helyette végigmegyünk egy *string*en:

```
using System;

public class Program
{
	static public void Main()
	{
		string str = "abcdefghijklmnopqrstuvwxyz";

		foreach(char ch in str)
		{
			Console.Write(ch);
		}
	}
}
```

A ciklusfejben felveszünk egy *char* típusú változót (egy *string* karakterekből áll), utána az *in* kulcsszó következik, amivel kijelöljük, hogy min megyünk át. A példában használt *ch* változó nem ciklusváltozó, hanem ún. iterációs változó, amely felveszi az iterált gyűjtemény aktuális elemének értékét. Éppen ezért egy *foreach* ciklus nem módosíthatja egy gyűjtemény elemeit (le sem fordulna ebben az esetben a program).

A foreach ciklus kétféle módban képes működni: ha a lista, amin alkalmazzuk, megvalósítja az *IEnumerable* és *IEnumerator* interfészeket, akkor azokat fogja használni, de ha nem, akkor hasonló lesz a végeredmény, mint egy számlálós ciklus esetében (leszámítva az iterációs változót, az mindenképpen megmarad).
A *foreach* pontos működésével az interfészekről szóló fejezet foglalkozik majd, ahol többek között megvalósítunk egy osztályt, amelyen a *foreach* képes végigiterálni (azaz megvalósítjuk az *IEnumerable* és *IEnumerator* interfészeket).

6.3.1 Yield

A *yield* kifejezés lehetővé teszi, hogy egy ciklusból olyan osztályt generáljon a fordító, amely megvalósítja az *IEnumerable* interfészt és ezáltal használható legyen pl. a *foreach* ciklussal:

```
using System;
using System.Collections;

public class Program
{
	static public IEnumerable EnumerableMethod(int max)
	{
		for(int i = 0;i < max;++i)
		{
			yield return i;
		}
	}

	static public void Main()
	{
		foreach(int i in EnumerableMethod(10))
		{
			Console.Write(i);
		}
	}
}
```

A yield működési elve a következő: a legelső metódushívásnál a ciklus megtesz egy lépést, ezután „kilépünk” a metódusból – de annak állapotát megőrizzük, azaz a következő hívásnál nem újraindul a ciklus, hanem onnan folytatja ahol legutóbb abbahagytuk.

6.3.2 Párhuzamos ciklusok

Ennek a fejezetnek a megértéséhez szükség van a generikus listák és a lambda kifejezések ismeretére, ezekről egy későbbi fejezet szól.

A több processzormaggal rendelkező számítógépek teljesítményének kihasználása céljából a Microsoft elkészítette a Task Parallel Library –t (illetve a PLINQ –t, erről egy későbbi fejezetben), amely a .NET 4.0 verziójában kapott helyet, ezért ehhez a fejezethez a C# 4.0 –hoz készült fordító szükséges.

A TPL számunkra érdekes része a párhuzamos ciklusok megjelenése. A NET 4.0 a *for* és a *foreach* ciklusok párhuzamosítását támogatja a következő módon:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks; // ez kell

class Program
{
	static public void Main()
	{
		List<int> list = new List<int>()
		{
			1, 2, 4, 56, 78, 3, 67
		};

		Parallel.For(0, list.Count, (index) =>
			{
				Console.Write("{0}, ", list[index]);
			});

		Console.WriteLine();

		Parallel.ForEach(list, (item) => Console.Write("{0}, ", item));
	}
}
```

A *For* első paramétere a ciklusváltozó kezdőértéke, második a maximumérték, míg a harmadik helyen a ciklusmagot jelentő *Action<int>* generikus *delegate* áll, amely egyetlen bemenő paramétere a ciklusváltozó aktuális értéke.

A *ForEach* két paramétere közül az első az adatforrás, míg a második a ciklusmag.

Mindkét ciklus számos változattal rendelkezik, ezek megtalálhatóak a következő MSDN oldalon:

http://msdn.microsoft.com/en-us/library/system.threading.tasks.parallel_members.aspx

7 Gyakorló feladatok

7.1 Szorzótábla

Készítsünk szorzótáblát! A program vagy a parancssori paraméterként kapott számot használja, vagy ha ilyet nem adtunk meg, akkor generáljon egy véletlen számot.

Megoldás (7/Mtable.cs)

Elsőként készítsük el a program vázát:

```
using System;

class Program
{
      static public void Main(string[] args)
      {

      }
}
```

Vegyük észre, hogy a *Main* metódus kapott egy paramétert, mégpedig egy *string* típusú elemekből álló tömböt (tömbökről a következő fejezetek adnak több tájékoztatást, most ez nem annyira lesz fontos). Ebben a tömbben lesznek az ún. parancssori paramétereink.

De mi is az a parancssori paraméter? Egy nagyon egyszerű példát nézzünk meg, azt, amikor lefordítunk egy C# forráskódot:

```
csc main.cs
```

Ebben az esetben a *csc* a fordítóprogram neve, míg a forráskódot tartalmazó file neve pedig a paraméter. Ha ezt vesszük alapul, akkor az *args* tömb egyetlen elemet tartalmaz a *csc*-re nézve, mégpedig a „main.cs” –t. Lássuk, hogy hogyan fog ez kinézni a mi programunkban (feltesszük, hogy mul.exe lesz a neve):

```
mul.exe 12
```

Vagyis a 12 –es szorzótáblát szeretnénk látni.

A következő lépésben fejlesszük tovább a programot, hogy írja ki a paraméterként megadott szám kétszeresét. Ehhez még szükségünk van arra is, hogy szám típussá alakítsuk a paramétert, hiszen azt *string*ként kapjuk meg. Erre a feladatra az *int.Parse* metódust használjuk majd, amely számmá konvertálja a paramétereként kapott szöveget (persze csak akkor, ha ez lehetséges, egyébként kivételt dob).

A forráskód most így alakul:

```
using System;

class Program
{
	static public void Main(string[] args)
	{
		int number = int.Parse(args[0]);
		Console.WriteLine(number * 2);
	}
}
```

Mivel a tömböket mindig nullától kezdve indexeljük, ezért az első parancssori paraméter – a megadott szám – a nulladik helyen lesz.

A programot most így tudjuk futtatni:

```
mul.exe 10
```

Erre az eredmény 20 lesz. Egyetlen probléma van, a program „összeomlik”, ha nem adunk meg paramétert.

Most módosítsuk úgy, hogy figyelmeztesse a felhasználót, hogy meg kell adnia egy számot is!

Ezt úgy fogjuk megoldani, hogy lekérdezzük a paramétereket tartalmazó tömb hosszát, és ha ez az érték nulla, akkor kiírjuk az utasítást:

```
using System;

class Program
{
	static public void Main(string[] args)
	{
		if(args.Length == 0)
		{
			Console.WriteLine("Adjon meg egy paramétert!");
		}
		else
		{
			int number = int.Parse(args[0]);
			Console.WriteLine(number * 2);
		}
	}
}
```

Egy kicsit szebb lesz a forráskód, ha az *else* ág használata helyett az *if* ágba teszünk egy *return* utasítást, amely visszaadja a vezérlést annak a „rendszernek” amely az őt tartalmazó metódust hívta (ez a metódus jelen esetben a *Main*, őt pedig mi – vagyis inkább az operációs rendszer – hívta, azaz a program befejezi a futását):

```
using System;

class Program
{
	static public void Main(string[] args)
	{
		if(args.Length == 0)
		{
			Console.WriteLine("Adj meg egy paramétert!");
			return;
		}

		int number = int.Parse(args[0]);
		Console.WriteLine(number * 2);
	}
}
```

A következő lépésben ahelyett, hogy kilépünk, ha nincs paraméter, inkább generálunk egy véletlen számot. Ehhez szükségünk lesz egy *Random* típusú objektumra.

A forráskód most ilyen lesz:

```
using System;

class Program
{
	static public void Main(string[] args)
	{
		int number;

		if(args.Length == 0)
		{
			Random r = newRandom();
			number = r.Next(100);
		}
		else
		{
			number = int.Parse(args[0]);
		}

		Console.WriteLine(number * 2);
	}
}
```

Véletlenszámot a *Next* metódussal generáltunk, a fenti formájában 0 és 100 között generál egy számot, de használhatjuk így is:

```
number = r.Next(10, 100);
```

Ekkor 10 és 100 közötti lesz a szám.

Már nincs más dolgunk, mint megírni a feladat lényegét, a szorzótáblát:

```
using System;

class Program
{
	static public void Main(string[] args)
	{
		int number;

		if(args.Length == 0)
		{
			Randomr = new Random();
			number = r.Next(100);
		}
		else
		{
			number = int.Parse(args[0]);
		}

		for(int i = 1;i <= 10;++i)
		{
			Console.WriteLine("{0} x {1} = {2}",
				i, number, i * number);
		}
	}
}
```

7.2 Számológép

Készítsünk egy egyszerű számológépet! A program indításakor kérjen be két számot és egy műveleti jelet, majd írja ki az eredményt.

Ezután bővítsük ki a programot, hogy a két számot illetve a műveleti jelet parancssori paraméterként is megadhassuk (ekkor nincs külön műveletválasztó menü, hanem írjuk ki rögtön az eredményt):

```
main.exe 12 23 +
```

(az eredmény pedig 35 lesz).

Megoldás (7/Calculator.cs)

Most is két részből áll a programunk, először hozzá kell jutnunk a számokhoz és az operátorhoz, majd elvégezzük a megfelelő műveletet és kiírjuk az eredményt.

Ezúttal is a szükséges változók deklarációjával kezdjük a programírást, három darab kell, két numerikus (legyen most int) és egy karaktertípus. Ezután bekérjük a felhasználótól a szükséges adatokat, vagy pedig felhasználjuk a paramétereket. Ez utóbbi esetben végezzünk egy kis hibaellenőrzést, vizsgáljuk meg, hogy pontosan három paramétert kaptunk–e!

A forráskód eddig így néz ki:

```
using System;

class Program
{
	static public void Main(string[] args)
	{
		int x, y;
		char op;

		if(args.Length == 0)
		{
			Console.WriteLine("Az első szám: ");
			x = int.Parse(Console.ReadLine());

			Console.WriteLine("A második szám: ");
			y = int.Parse(Console.ReadLine());

			Console.WriteLine("A művelet(+, -, *, /): ");
			op = Convert.ToChar(Console.Read());
		}
		else
		{
			if(args.Length != 3)
			{
				Console.WriteLine("Túl sok/kevés paraméter!");
				return;
			}
			else
			{
				x = int.Parse(args[0]);
				y = int.Parse(args[1]);
				op = Convert.ToChar(args[2]);
			}
		}
	}
}
```

Az operátor „megszerzéséhez” egyrészt a *Console.Read* metódust használtuk (mivel csak egyetlen karakterre van szükség), másrészt ennek a metódusnak a visszatérési értékét – amely egy egész szám – át kellett konvertálnunk karaktertípussá, ehhez a *Convert.ToChar* metódus nyújtott segítséget.

Most már nagyon egyszerű dolgunk van, mindössze ki kell számolnunk az eredményt. Ezt nyílván a beolvasott operátor alapján fogjuk megtenni, ebben a helyzetben pedig a legkézenfekvőbb, ha a *switch* szerkezetet használjuk:

```
int result=0;

switch(op)
{
      case '+':
            result = x + y;
            break;
      case '-':
            result = x - y;
            break;
      case '*':
            result = x * y;
            break;
      case '/':
            result = x / y;
            break;
}

Console.WriteLine("A művelet eredménye: {0}", result);
```

Amire figyelni kell az az, hogy a *result* változó kapjon kezdőértéket, ellenkező esetben ugyanis nem fordul le a program (*uninitialized variable* kezdetű hibaüzenetet kapunk). Ez azért van így, mert a változó-deklaráció és az utolsó sorban levő kiíratás között nem biztos, hogy megtörténik a változó-definíció.

Ugyan az értéktípusok bizonyos helyzetekben automatikusan nulla értéket kapnak, de ez nem minden esetben igaz és lokális változók esetén épp ez a helyzet. Ezért minden olyan esetben, amikor egy lokális változó deklarációja és definíciója között használni akarjuk a változót, akkor hibaüzenetet fogunk kapni.

A fenti esetben megoldást jelenthet az is, ha bevezetünk a *switch* szerkezetben egy *default* címkét, amivel minden esetben megtörténik - valamilyen formában – az értékadás.

7.3 Kő – Papír – Olló

Készítsünk kő-papír-olló játékot! Ebben az esetben is használjuk a véletlenszám-generátort. A játék folyamatos legyen, vagyis addig tart, amíg a felhasználó kilép (nehezítésképpen lehet egy karakter, amelyre a játék végetér). Tartsuk nyílván a játék állását és minden forduló után írjuk ki az aktuális eredményt.

Megoldás (7/SPS.cs)

A programunk lényegében három részből fog állni: elsőként megkérdezzük a felhasználótól, hogy mit választott, majd sorsolunk a „gépnek” is valamit, végül pedig kiértékeljük az eredményt.
Első dolgunk legyen, hogy deklarálunk öt darab változót, egy *Random* objektumot, két *string*et (ezekben tároljuk, hogy mit választottak a „versenyzők”), és két *byte* típust (ezekben pedig az eredményeket tartjuk számon, itt elég lesz a *byte* is, mert feltesszük, hogy senki nem fog több százszor játszani egymás után).
Szükségünk lesz még egy logikai típusra is, ezzel fogjuk jelezni a programnak, hogy akarunk –e még játszani.

Készítsük el a program vázát a változó-deklarációkkal, illetve a főciklussal (a ciklustörzsben kérdezzünk rá, hogy akarunk –e még játszani):

```
using System;

class Program
{
	static public void Main()
	{
		Random r = new Random();

		string compChoice = "";
		string playerChoice = "";

		int compScore = 0;
		int playerScore = 0;

		bool l = true;
		do
		{
			Console.WriteLine("Akarsz még játszani? i/n");
			if(Console.ReadKey(true).KeyChar == 'n'){ l = false; }
		}while(l);
	}
}
```

A *Console.ReadKey* metódussal egyetlen karaktert olvasunk be a standard bemenetről, de ezt az adatot nem használhatjuk fel azonnal, mivel nem *char* típust hanem *ConsoleKeyInfo* objektumot ad vissza. Ez utóbbi *KeyChar* tulajdonságával kapjuk vissza a beírt karaktert. A *ReadKey* most kapott egy paramétert is, amellyel azt jeleztük, hogy az adott karakter ne jelenjen meg a konzolon.

Most kérjük be az adatokat:

```
Console.WriteLine("Mit választasz? (k/p/o)");

switch(Console.ReadKey(true).KeyChar)
{
	case 'k':
		playerChoice = "kő";
		break;
	case 'p':
		playerChoice = "papír";
		break;
	case 'o':
		playerChoice = "olló";
		break;
}

switch(r.Next(0,3))
{
	case 0:
		compChoice = "kő";
		break;
	case 1:
		compChoice = "papír";
		break;
	case 2:
		compChoice = "olló";
		break;
}
```

Ez a kódrészlet természetesen a ciklustörzsben a kérdés elé kerül. Az egyes lehetőségeket a k/p/o billentyűkre állítottuk.

Már csak az eredmény kiértékelése van hátra:

```
if((playerChoice == "kő" && compChoice == "papír")
      ||
   (playerChoice == "papír" && compChoice == "olló")
    ||
   (playerChoice == "olló" && compChoice == "kő"))
{
      Console.WriteLine("Veszítettél! Az állás:\nSzámítógép:
{0}\nJátékos:{1}",
            ++compScore, playerScore);
}
elseif(playerChoice == compChoice)
{
      Console.WriteLine("Döntetlen! Az állás:\nSzámítógép:
{0}\nJátékos:{1}",
            compScore, playerScore);
}
else
{
      Console.WriteLine("Nyertél! Az állás:\nSzámítógép: {0}\nJátékos:{1}",
            compScore, ++playerScore);
}
```

Érdekesebb megoldás bitműveletekkel elkészíteni a kiértékelést, ehhez egy kis segítség: jelöljük a három lehetőséget (k/p/o) a következőképpen: 100, 010, 001. Nyilvánvaló, hogy egy bitenkénti VAGY művelettel ellenőrizni tudjuk a döntetlen eredményt mivel ekkor 100 | 100 = 100, vagyis ugyanazt kapjuk vissza. Ha az eredmény nem döntetlen, akkor a VAGY háromféle eredményt adhat vissza (KP, KO, OP): 110, 101 és 011, amelyeket pl. egy *switch* szerkezettel dolgozhatunk fel.

7.4 Számkitaláló játék

Készítsünk számkitaláló játékot, amely lehetőséget ad kiválasztani, hogy a felhasználó próbálja kitalálni a program által kisorsolt számot, vagy fordítva. A kitalált szám legyen 1 és 100 között. Öt próbálkozása lehet a játékosnak, minden tipp után írjuk ki, hogy a tippelt szám nagyobb, vagy kisebb-e, mint a kitalált szám.
Ha a gépen van a sor, akkor használjunk véletlenszám-generátort a szám létrehozására. A gépi játékos úgy találja ki a számot, hogy mindig felezi az intervallumot (pl.először 50–et tippel, ha a kitalált szám nagyobb, akkor 75 jön, és így tovább).
A felhasználótól a játék végén kérdezzük meg, hogy akar–e ismét játszani.

Megoldás (7/Number.cs)

Ennek a feladatnak a legnagyobb kihívása, hogy olyan programszerkezetet rakjunk össze, amely átlátható és könnyen módosítható. Legyen az alapötlet az, hogy elágazásokat használunk. Nézzük meg így a program vázát:

```
static public void Main()
{
	// Itt kiválasztjuk, hogy ki választ számot

	if(/* A játékos választ */)
	{
		// A számítógép megpróbálja kitalálni a számot
	}
	else// A számítógép választ
	{
		// A játékos megpróbálja kitalálni a számot
	}

	// Megkérdezzük a játékost, hogy akar-e még játszani
}
```

Nem néz ki rosszul, de a probléma az, hogy a két feltétel blokkja nagyon el fog „hízni”, emiatt pedig kevésbé lesz olvasható a forráskód. Ez persze nem olyan nagy baj, de ahhoz elég, hogy valami mást próbáljunk ki.

A procedurális és alacsonyszintű nyelvek az ilyen feladatokat „ugró” utasításokkal oldják meg, vagyis a forráskódban elhelyezett címkék között ugrálnak (tulajdonképpen a magas szintű nyelveknél is ez történik, csak ezt mi nem látjuk mivel az *if*/*switch*/stb. elfedi előlünk). Ez a módszer a magas szintű nyelvek esetén nem igazán ajánlott (főleg, mert megvannak az eszközök az ugrálás kikerülésére), de jelenleg nem is használjuk ki teljesen a nyelv adta lehetőségeket, ezért most szabad „rosszalkodnunk”. Írjuk át a fenti vázat egy kicsit:

```
using System;

class Program
{
	static public void Main()
	{
		START:

			Console.WriteLine("Válassz játékmódot!");
			Console.WriteLine("1 - Te gondolsz egy számra");
			Console.WriteLine("2 - A számítógép gondol egy számra");

			switch(Console.ReadKey(true).KeyChar)
			{
				case '1': goto PLAYER;
				case '2': goto COMPUTER;
			}

			PLAYER: goto END;

			COMPUTER: goto END;

		END:
			Console.WriteLine("\nAkarsz még játszani? i/n");

			switch(Console.ReadKey(true).KeyChar)
			{
				case 'i': goto START;
				case 'n': break;
			}
	}
}
```

Közben ki is egészítettük a kódot egy kicsit, lehet vele kísérletezni, amíg el nem készítjük a lényeget. Jól látható, hogy a címkékkel kellemesen olvashatóvá és érthetővé vált a kód (persze ennél nagyobb terjedelmű forrásnál már problémásabb lehet).
Már elkészítettük a programrészt, amely megkérdezi a játékost, hogy szeretne-e még játszani. Egy apró hibája van, mégpedig az, hogy ha **i** vagy **n** helyett más billentyűt nyomunk le, akkor a program végetér. Ezt könnyen kijavíthatjuk, ha egy kicsit gondolkodunk. Nyílván a *default* címkét kell használni és ott egy ugró utasítással a vezérlést a megfelelő helyre tenni:

```
END:
      Console.WriteLine("\nAkarsz még játszani? i/n");

      switch(Console.ReadKey(true).KeyChar)
      {
            case 'i': goto START;
            case 'n': break;
            default: goto END;
      }
```

Most következik a program lényege: a játék elkészítése. Elsőként azt a szituációt implementáljuk, amikor a játékos próbálja kitalálni a számot, mivel ez az egyszerűbb. Szükségünk lesz természetesen változókra is, de érdemes átgondolni, hogy úgy vegyük fel őket, hogy mindkét programrész használhassa őket. Ami biztosan mindkét esetben kell az a véletlenszámgenerátor, valamint két *int* típusú változó az egyikben a játékos és a számítógép tippjeit tároljuk, a másik pedig a ciklusváltozó lesz, neki adjunk azonnal nulla értéket. Ezt a kettőt deklaráljuk egyelőre, rögtön a START címke előtt. Készítsük is el a programot, nem lesz nehéz dolgunk, mindössze egy ciklusra lesz szükségünk:

```
COMPUTER:
      int number = r.Next(100);
      i = 0;
      while(i < 5)
      {
            Console.WriteLine("\nA tipped: ");
            x = int.Parse(Console.ReadLine());

            if(x < number)
            {
                  Console.WriteLine("A szám ennél nagyobb!");
            }
            elseif(x > number)
            {
                  Console.WriteLine("A szám ennél kisebb!");
            }
            else
            {
                  Console.WriteLine("Nyertél!");
                  goto END;
            }
            ++i;
      }

      Console.WriteLine("\nVesztettél, a szám {0} volt.", number);
goto END;
```

Ennek megértése nem okozhat gondot, léphetünk a következő állomásra, ami viszont kicsit nehezebb lesz. Ahhoz, hogy megfelelő stratégiát készítsünk a számítógép számára, magunknak is tisztában kell lennünk azzal, hogy hogyan lehet megnyerni ezt a játékot. A legáltalánosabb módszer, hogy mindig felezzük az intervallumot, így az utolsó tippre már elég szűk lesz az a számhalmaz, amiből választhatunk (persze így egy kicsit szerencsejáték is lesz).
Nézzünk meg egy példát: a gondolt szám legyen a 87 és tudjuk, hogy a szám egy és száz között van. Az első tippünk 50 lesz, amire természetesen azt a választ kapjuk, hogy a szám ennél nagyobb. Már csak 50 lehetséges szám maradt, ismét felezünk, a következő tippünk így a 75 lesz. Ismét azt kapjuk vissza, hogy ez nem elég. Ismét felezünk, méghozzá maradék nélkül, vagyis tizenkettőt adunk hozzá a hetvenöthöz és így ki is találtuk a gondolt számot.

Most már könnyen fel tudjuk írni, hogy mit kell tennie a számítógépnek: az első négy kísérletnél felezzük az intervallumot, az utolsó körben pedig tippelünk. Nézzük a kész kódot:

```
PLAYER:
      Console.WriteLine("Gondolj egy számra! (1 - 100)");
      Console.ReadLine();

      x = 50;
      int min = 0;
      int max = 100;
      while(i < 5)
      {
            Console.WriteLine("A számítógép szerint a szám {0}",x);
            Console.WriteLine("Szerinted? (k/n/e)");

            switch(Console.ReadKey(true).KeyChar)
            {
                  case 'k':
                        if(i == 3){ x = r.Next(min, x); }
                        else
                        {
                              max = x;
                              x -= (max - min) / 2;
                        }
                        break;
                  case 'n':
                        if(i == 3){ x = r.Next(x + 1,max); }
                        else
                        {
                              min = x;
                              x += (max - min) / 2;
                        }
                        break;
                  case 'e':
                        Console.WriteLine("A számítógép nyert!");
                        goto END;
            }

            ++i;
      }

      Console.WriteLine("A számítógép nem tudta kitalálni a számot. ☹");
goto END;
```

A *min* illetve *max* változókkal tartjuk számon az intervallum alsó, illetve felső határát. Az *x* változóban tároljuk az aktuális tippet, neki meg is adtuk a kezdőértéket.

Egy példán keresztül nézzük meg, hogy hogyan működik a kódunk. Legyen a gondolt szám ismét 87. A számítógép első tippje 50, mi erre azt mondjuk, hogy a szám ennél nagyobb, ezért a *switch* 'n' ága fog beindulni. Az intervallum alsó határa ezután *x* (vagyis 50) lesz, mivel tudjuk, hogy a szám ennél biztosan nagyobb. A felső határ nyílván nem változik, már csak az új *x*–et kell kiszámolni, vagyis *x*–hez hozzá kell adni a felső és alsó határok különbségének a felét: (100 – 50) / 2 = 25 (ellenkező esetben pedig nyílván le kellene vonni ugyanezt).
Amiről még nem beszéltünk az az a feltétel, amelyben *x* egyenlőségét vizsgáljuk hárommal (vagyis azt akarjuk tudni, hogy eljött –e az utolsó tipp ideje). Ez az elágazás fogja visszaadni az utolsó tippet a véletlenszám-generátorral a megfelelően leszűkített intervallumban.

8 Típuskonverziók

Azt már tudjuk, hogy az egyes típusok másként jelennek meg a memóriában, azonban gyakran kerülünk olyan helyzetbe, hogy egy adott típusnak úgy kellene viselkednie, mint egy másiknak. Ilyen helyzetekben típuskonverziót (vagy *cast*olást) kell elvégeznünk.
Kétféleképpen konvertálhatunk: implicit és explicit módon. Az előbbi esetben nem kell semmit tennünk, a fordító kérés nélkül elvégzi helyettünk. Implicit konverzió általában „hasonló” típusokon működik, szinte minden esetben a szűkebb típusról a tágabbra:

```
int x = 10;
long y = x; // y == 10, implicit konverzió
```

Ebben az esetben a *long* és *int* mindketten egész numerikus típusok, és mivel a long tágabb (az *int* 32 bites míg a *long* 64 bites egész szám), ezért a konverzió gond nélkül végbe megy. Egy implicit konverzió minden esetben sikeres és nem jár adatvesztéssel.

Egy explicit konverzió nem feltétlenül fog működni, és ha mégis akkor adatvesztés is felléphet. Vegyük a következő példát:

```
int x = 300;
byte y = (byte)x; // explicit konverzió, y == ??
```

A *byte* szűkebb típus mint az *int* (8 illetve 32 bitesek), ezért explicit konverziót hajtottunk végre, ezt a változó előtti zárójelbe írt típussal jelöltük. Ha lehetséges a konverzió, akkor végbemegy, egyébként a fordító figyelmeztetni fog.
Vajon mennyi most az *y* változó értéke? A válasz elsőre meglepő lehet: 44. A magyarázat: a 300 egy kilenc biten felírható szám (100101100), persze az *int* kapacitása ennél nagyobb, de most csak a hasznos részre van szükség. A *byte* viszont (ahogy a nevében is benne van) egy nyolcbites értéket tárolhat (vagyis a maximum értéke 255 lehet), ezért a 300–nak csak az első 8 bitjét (00101100) adhatjuk át *y*–nak, ami pont 44.

8.1 Ellenőrzött konverziók

A programfejlesztés alatt hasznos lehet tudnunk, hogy minden konverzió gond nélkül lezajlott-e vagy sem. Ennek ellenőrzésére ún. ellenőrzött konverziót fogunk használni, amely kivételt dob (erről hamarosan), ha a forrás nem fér el a célváltozóban:

```
checked
{
      int x = 300;
      byte y = (byte)x;
}
```

Ez a kifejezés kivételt (*System.OverflowException*) fog dobni (ha elindítjuk a lefordított programot hibaüzenet fogad majd). Figyeljünk arra, hogy ilyen esetekben csak a blokkon belül deklarált, statikus és tagváltozókat vizsgálhatjuk.

Előfordul, hogy csak egy-egy konverziót szeretnénk vizsgálni, amihez nincs szükség egy egész blokkra:

```
int x = 300;
byte y = checked((byte)x);
```

Az ellenőrzés kikapcsolását is megtehetjük az *unchecked* használatával:

```
int x = 300;
byte y = unchecked((byte)x);
```

Az ajánlás szerint ellenőrzött konverziókat csak a fejlesztés ideje alatt (tesztelésre) használjunk, mivel némi teljesítményvesztéssel jár.

8.2 Is és as

Az *is* operátort futásidejű típus-lekérdezésre használjuk:

```
using System;

public class Program
{
	static public void Main()
	{
		int x = 10;

		if(x is int) // ha x egy int
		{
			Console.WriteLine("x típusa int");
		}
	}
}
```

Ez a program lefordul, de figyelmeztetést kapunk, mivel a fordító felismeri, hogy a feltétel mindig igaz lesz.

Ennek az operátornak a leggyakoribb felhasználási területe az interfész-megvalósítás lekérdezése (erről később).

Párja az *as* az ellenőrzés mellett egy explicit típuskonverziót is végrehajt. Ezzel az operátorral csakis referencia-típusra konvertálhatunk, értéktípusra nem (ekkor le sem fordul a program).

Nézzünk egy példát:

```
using System;

public class Program
{
	static public void Main()
	{
		object a = "123";
		object b = "Hello";
		object c = 10;

		string aa = a as string;
		Console.WriteLine(aa == null ? "NULL" : aa); // 123

		string bb = b as string;
		Console.WriteLine(bb == null ? "NULL" : bb); // Hello

		string cc = c as string;
		Console.WriteLine(cc == null ? "NULL" : cc); // NULL
	}
}
```

Amennyiben ez a konverzió nem hajtható végre, a célváltozóhoz *null* érték rendelődik (ezért is van a referencia-típusokhoz korlátozva ez az operátor).

8.3 Karakterkonverziók

A *char* típust implicit módon tudjuk numerikus típusra konvertálni, ekkor a karakter unicode értékét kapjuk vissza:

```
using System;

class Program
{
	static public void Main()
	{
		for(char ch = 'a';ch <= 'z';++ch)
		{
			Console.WriteLine((int)ch);
		}
	}
}
```

Erre a kimeneten a következő számok jelennek meg: 97, 98, 99, 100, ...

A kis 'a' betű hexadecimális unicode száma 0061h, ami a 97 decimális számnak felel meg, tehát a konverzió a tízes számrendszerbeli értéket adja vissza.

9 Tömbök

Gyakran van szükségünk arra, hogy több azonos típusú objektumot tároljunk el, ilyenkor kényelmetlen lenne mindegyiknek külön változót foglalnunk (képzeljünk el 30 darab *int* típusú változót, még leírni is egy örökkévalóság lenne), de ezt nem is kell megtennünk, hiszen rendelkezésünkre áll a tömb adatszerkezet.

A tömb meghatározott számú, azonos típusú elemek halmaza. Minden elemre egyértelműen mutat egy index (egész szám). A tömbök referencia-típusok. A C# mindig folytonos memóriablokkokban helyezi el egy tömb elemeit.

Tömböt a következőképpen deklarálhatunk:

```
int[] array = new int[10];
```

Ez a tömb tíz darab *int* típusú elem tárolására alkalmas. A tömb deklarációja után az egyes indexeken lévő elemek automatikusan a megfelelő nullértékre inicializálódnak (ebben az esetben 10 darab nullát fog tartalmazni a tömbünk). Ez a szabály referencia-típusoknál kissé máshogy működik, mivel ekkor a tömbelemek *null* -ra inicializálódnak. Ez nagy különbség, mivel értéktípusok esetében szimpla nullát kapnánk vissza az általunk nem beállított indexre hivatkozva (vagyis ez egy teljesen szabályos művelet), míg referencia-típusoknál ugyanez *NullReferenceException* típusú kivételt fog generálni.

Az egyes elemekre az indexelő operátorral (szögletes zárójelek: [ ]) és az elem indexével (sorszámával) hivatkozunk. A számozás mindig nullától kezdődik, így a legutolsó elem indexe: az elemek száma mínusz egy. A következő példában feltöltünk egy tömböt véletlen számokkal és kiíratjuk a tartalmát:

```
using System;

class Program
{
	static public void Main()
	{
		int[] array = new int[10];

		Random r = new Random();
		for(int i = 0;i < array.Length;++i)
		{
			array[i] = r.Next();
		}

		foreach(int item in array)
		{
			Console.WriteLine(item);
		}
	}
}
```

A példában a ciklusfeltétel megadásakor a tömb *Length* nevű tulajdonságát használtuk, amely visszaadja a tömb hosszát. Látható, az indexelő-operátor használata is, az *array[i]*a tömb *i*–edik elemét jelenti. Az indexeléssel vigyázni kell, ugyanis a fordító nem ellenőrzi fordítási időben az indexek helyességét, viszont

helytelen indexelés esetén futás időben *IndexOutOfRangeException* kivételt fog dobni a program.

Egy tömböt akár a deklaráció pillanatában is feltölthetünk a nekünk megfelelő értékekkel:

```
char[] charArray = new char[]{ 'b', 'd', 'a', 'c' };
```

Ekkor az elemek számát a fordító fogja meghatározni. Az elemek számát a deklarációval azonnal meghatározzuk, ezen a későbbiekben nem lehet változtatni. Dinamikusan bővíthető adatszerkezetekről a **Gyűjtemények** című fejezet szól.

Minden tömb a *System.Array* osztályból származik, ezért néhány hasznos művelet azonnal rendelkezésünkre áll (pl. rendezhetünk egy tömböt a *Sort* metódussal):

```
chararray.Sort(); // tömb rendezése
```

9.1 Többdimenziós tömbök

Eddig az ún. egydimenziós tömböt (vektort) használtuk. Lehetőségünk van azonban többdimenziós tömbök létrehozására is, ekkor nem egy indexszel hivatkozunk egy elemre, hanem annyival, ahány dimenziós a tömb. Vegyük például a matematikából már ismert mátrixot:

$$A = \begin{bmatrix} 12, 23, 2 \\ 13, 67, 52 \\ 45, 55, 1 \end{bmatrix}$$

Ez egy kétdimenziós tömbnek (mátrix a neve – ki hinné?) felel meg, az egyes elemekre két indexxel hivatkozunk, első helyen a sor áll és utána az oszlop. Így a 45 indexe: [2, 0] (ne feledjük, még mindig nullától indexelünk).

Multidimenziós tömböt a következő módon hozunk létre C# nyelven:

```
int[,] matrix = new int[3, 3];
```

Ez itt egy 3x3–as mátrix, olyan mint a fent látható. Itt is összeköthetjük az elemek megadását a deklarációval, bár egy kicsit trükkösebb a dolog:

```
int[,] matrix = new int[,]
{
      {12, 23, 2},
      {13, 67, 52},
      {45, 55, 1}
};
```

Ez a mátrix pontosan olyan, mint amit fönt leírtunk. Az elemszám most is meghatározott, nem változtatható.

Nyílván nem akarjuk mindig kézzel feltölteni a tömböket, viszont ezúttal nem olyan egyszerű a dolgunk, hiszen egy ciklus biztosan nem lesz elég ehhez, vagyis gondolkodnunk kell: az index első tagja a sort, a második az oszlopot adja meg, pontosabban az adott sorban elfoglalt indexét. Ez alapján pedig jó ötletnek tűnik, ha egyszerre csak egy dologgal foglalkozunk, azaz szépen végig kell valahogyan mennünk minden soron egyesével. Erre a megoldást az ún. egymásba ágyazott ciklusok jelentik: a külső ciklus a sorokon megy át, a belső pedig a sorok elemein:

```
using System;

class Program
{
	static public void Main()
	{
		int[,] matrix = new int[3, 3];
		Random r = new Random();

		// sorok
		for(int i = 0;i < matrix.GetLength(0);++i)
		{
			// oszlopok
			for(int j = 0;j < matrix.GetLength(1);++j)
			{
				matrix[i, j] = r.Next();
			}
		}
	}
}
```

Most nem írjuk ki a számokat, ez nem okozhat gondot. A tömbök *GetLength* metódusa a paraméterként megadott dimenzió hosszát adja vissza (nullától számozva), tehát a példában az első esetben a sor, a másodikban az oszlop hosszát adjuk meg a ciklusfeltételben.

A többdimenziós tömbök egy variánsa az ún. egyenetlen (jagged) tömb. Ekkor legalább egy dimenzió hosszát meg kell adnunk, ez konstans marad viszont a belső tömbök hossza tetszés szerint megadható:

```
int[][] jarray = new int[3][];
```

Készítettünk egy három sorral rendelkező tömböt, azonban a sorok hosszát (az egyes sorok nyílván önálló vektorok) ráérünk később megadni, és nem kell ugyanolyan hosszúnak lenniük.

```
using System;

class Program
{
	static public void Main()
	{
		int[][] jarray = new int[3][];
		Random r = new Random();

		for(int i = 0;i < 3;++i)
		{
			jarray[i] = new int[r.Next(1, 5)];
			for(int j = 0;j < jarray[i].Length;++j)
			{
				jarray[i][j] = i + j;
			}
		}
	}
}
```

Véletlenszám-generátorral adjuk meg a belső tömbök hosszát, persze értelmes kereteken belül. A belső ciklusban jól látható, hogy a tömb elemei valóban tömbök, hiszen használtuk a *Length* tulajdonságot (persze a hagyományos többdimenziós tömbök esetében is ez a helyzet, de ott nem lenne értelme külön elérhetővé tenni az egyes sorokat).

Az inicializálás a következőképpen alakul ebben az esetben:

```
int[][] jarray = new int[][]
{
	newint[]{ 1, 2, 3, 4, 5 },
	newint[]{ 1, 2, 3 },
	newint[]{ 1 }
};
```

10 Stringek

A C# beépített karaktertípusa (*char*) egy unicode karaktert képes tárolni két byteon. A szintén beépített *string* típus ilyen karakterekből áll (tehát az egyes betűket *char*–ként kezelhetjük).

```
using System;

class Program
{
	static public void Main()
	{
		string s = "ezegystring";
		Console.WriteLine(s);
	}
}
```

A látszat ellenére a *string* referencia-típus, viszont nem kötelező használnunk a *new* operátort.

Egy *string* egyes betűire az indexelő operátorral hivatkozhatunk (vagyis minden *string*et kezelhetünk tömbként is):

```
using System;

class Program
{
	static public void Main()
	{
		string s = "ezegystring";
		Console.WriteLine(s[0]); // e
	}
}
```

Ekkor a visszaadott objektum típusa *char* lesz. A *foreach* ciklussal indexelő operátor nélkül is végigiterálhatunk a karaktersorozaton:

```
foreach(char ch in s)
{
	Console.WriteLine(ch);
}
```

Az indexelő operátort nem csak változókon, de „nyers” szövegen is alkalmazhatjuk:

```
Console.WriteLine("ezegystring"[4]); // y
```

Ilyenkor egy „névtelen” változót készít a fordító és azt használja.

Nagyon fontos tudni, hogy mikor egy létező *string* objektumnak új értéket adunk, akkor nem az eredeti példány módosul, hanem egy teljesen új objektum keletkezik a memóriában (vagyis a *string* ún. immutable – megváltoztathatatlan típus). Ez a viselkedés főleg akkor okozhat (teljesítmény)problémát, ha sokszor van szükségünk erre a műveletre.

10.1 Metódusok

A .NET számos hasznos metódust biztosít a *string*ek hatékony kezeléséhez. Most megvizsgálunk néhányat, de tudni kell, hogy a metódusoknak számos változata lehet, itt most a leggyakrabban használtakat nézzük meg:

Összehasonlítás:

```
using System;

class Program
{
	static public void Main()
	{
		string a = "egyik";
		string b = "másik";

		int x = String.Compare(a, b);

		if(x == 0)
		{
			Console.WriteLine("A két string egyenlő");
		}
		elseif(x < 0)
		{
			Console.WriteLine("Az 'a' a kisebb");
		}
		else
		{
			Console.WriteLine("A 'b' a kisebb");
		}
	}
}
```

A *String.Compare* metódus nullát ad vissza, ha a két *string* egyenlő, és nullánál kisebbet/nagyobbat, ha nem (pontosabban, ha lexikografikusan – lényegében ábécésorrend szerint – kisebb/nagyobb).

Keresés:

```
using System;

class Program
{
	static public void Main()
	{
		string s = "verylonglongstring";
		char[] chs = new char[]{ 'y', 'z', 'o' };

		Console.WriteLine(s.IndexOf('r')); // 2
		Console.WriteLine(s.IndexOfAny(chs)); // 3
		Console.WriteLine(s.LastIndexOf('n')); // 16
		Console.WriteLine(s.LastIndexOfAny(chs)); // 9
		Console.WriteLine(s.Contains("long")); // true
	}
}
```

Az *IndexOf* és *LastIndexOf* metódusok egy *string* vagy karakter első illetve utolsó előfordulási indexét (*string*ek esetén a kezdőindexet) adják vissza. Ha nincs találat, akkor a visszaadott érték -1 lesz. A két metódus *Any*–re végződő változata egy karaktertömböt fogad paramétereként és az abban található összes karaktert próbálja megtalálni. A *Contains* igaz értékkel tér vissza, ha a paramétereként megadott karakter(sorozat) benne van a *string*ben.

Módosítás:

```
using System;

class Program
{
static public void Main()
{
      string s = "smallstring";
      char[] chs = new char[]{ 's', 'g' };

      Console.WriteLine(s.Replace('s', 'l')); // lmallltring
      Console.WriteLine(s.Trim(chs)); // mallstrin
      Console.WriteLine(s.Insert(0, "one")); // onesmallstring
      Console.WriteLine(s.Remove(0, 2)); // allstring
      Console.WriteLine(s.Substring(0, 3)); // sma
      Console.WriteLine(s.ToUpper()); // SMALLSTRING
      Console.WriteLine(s.ToLower()); // smallstring
}
}
```

A *Replace* metódus az első paraméterének megfelelő karaktereket lecseréli a második paraméterre. A *Trim* string elején és végén lévő karaktereket vágja le, a *Substring* pedig kivág egy karaktersorozatot, paraméterei a kezdő és végindexek (van egyparaméteres változata is, ekkor a csak a kezdőindexet adjuk meg és a végéig megy). Az *Insert/Remove* metódusok hozzáadnak, illetve elvesznek a *string*ből (a megadott indexeken). Végül a *ToLower* és *ToUpper* metódusok kis- illetve nagybetűssé alakítják az eredeti *string*et.

Fontos megjegyezni, hogy ezek a metódusok soha nem az eredeti *string*en végzik a módosításokat, hanem egy új példányt hoznak létre, és azt adják vissza.

10.2 StringBuilder

Azt már tudjuk, hogy amikor módosítunk egy *string*et, akkor automatikusan egy új példány jön létre a memóriában, ez pedig nem feltétlenül „olcsó” művelet.

Ha sokszor (legalább 10+) van szükségünk erre, akkor használjuk inkább a *StringBuilder* típust, ez automatikusan lefoglal egy nagyobb darab memóriát, és ha ez sem elég, akkor allokál egy megfelelő méretű területet és átmásolja magát oda. A *StringBuilder* a *System.Text* névtérben található.

Példa a *StringBuilder* használatára:

```
using System;
using System.Text;

class Program
{
	static public void Main(string[] args)
	{
		StringBuilder sb = new StringBuilder(50);

		for(char ch = 'a';ch <= 'z';++ch)
		{
			sb.Append(ch);
		}

		Console.WriteLine(sb);
	}
}
```

A *StringBuilder* fenti konstruktora (van több is) helyet foglal ötven karakter számára (létezik alapértelmezett konstruktora is, ekkor tizenhat karakternek foglal helyet). Az *Append* metódussal tudunk karaktereket (vagy egész *string*eket) hozzáfűzni.

10.3 Reguláris kifejezések

Reguláris kifejezések segítségével vizsgálhatjuk, hogy egy karaktersorozat megfelel-e egy adott mintának. Nézzük is egy példát: készítsünk olyan reguláris kifejezést, amely természetes számokra illik! A példában a nullát nem soroljuk a természetes számok közé, vagyis a minta a következő lesz: az első számjegy egy és kilenc közötti lesz, ezután pedig bármennyi nulla és kilenc közötti számjegy jöhet. A forráskód:

```
using System;
using System.Text.RegularExpressions; // ez kell

class Program
{
	static public void Main()
	{
		Regex pattern = new Regex("^[1-9][0-9]*");

		string s1 = "012345";
		string s2 = "112345";
		string s3 = "2";

		Console.WriteLine("{0} : {1}", s1, (pattern.IsMatch(s1) ?
			"természetes szám" : "nem természetes szám"));

		Console.WriteLine("{0} : {1}", s2, (pattern.IsMatch(s2) ?
			"természetes szám" : "nem természetes szám"));

		Console.WriteLine("{0} : {1}", s3, (pattern.IsMatch(s3) ?
			"természetes szám" : "nem természetes szám"));
	}
}
```

A reguláris kifejezést egy *Regex* példánynak adjuk át, lássuk, hogy hogyan épül fel:

^: ezzel a karakterrel (Alt Gr + 3) megmondjuk, hogy a mintaillesztést a *string* elejétől kell kezdeni.

[1-9]: a *string* elején egy darab numerikus karakter áll – kivéve a nullát.

[0-9]*: a csillag(*) nulla vagy több előfordulást jelent, vagyis az első számjegy után nulla vagy több nulla és kilenc közötti szám állhat.

A programunk a következő kimenetet produkálja:

```
012345 : nem természetes szám
112345 : természetes szám
2 : természetes szám
```

A következő példánk egy kicsit bonyolultabb lesz a reguláris kifejezés pedig már-már ijesztő: képzeljük el, hogy a világon uniformizálják a telefonszámokat, a következő módon: minden szám az ország előhívójával kezdődik amelyet egy '+' jel előz meg, az előhívószám két vagy három számjegyből áll, az első szám nem lehet nulla. Ezután egy szóköz jön, amelyet a körzetszám követ, ami szintén két vagy három számjegy az első helyen itt sem állhat nulla. Végül következik a telefonszám, ami a körzetszámot szóközzel elválasztva követi. Minden telefonszám három darab három számjegyből álló blokkból áll ezeket kötőjel választja el. Egy példa:

+36 30 123-456-789

Lássuk a forráskódot:

```
string s = @"^(\+)[1-9][0-9]{1,2}\s[1-9][0-9]{1,2}\s(\d{3}(-)){2}\d{3}$";

Regex pattern = new Regex(s);

string s1 = "+36 30 661-345-612";
string s2 = "+3630 661-567-233";
string s3 = "+56 30 667-876-987-456";

Console.WriteLine(pattern.IsMatch(s1)); // true
Console.WriteLine(pattern.IsMatch(s2)); // false
Console.WriteLine(pattern.IsMatch(s3)); // false
```

A reguláris kifejezés elé kötelezően oda kell tennünk a @ jelet, mivel speciális karaktereket is tartalmaz (erre a fordító is figyelmeztet).
A kifejezésünk két részből áll:

^(\+)[1-9][0-9]{1,2}\s: ez a minta lesz az első két helyen és ő hívatott az előhívót illetve a körzetszámot ellenőrizni: a már ismert ^-vel az illesztést a

karaktersor elejétől kezdjük, az ezt követő (\+) pedig megmondja, hogy az első helyen egy '+' jel van (ez persze nem lesz ott a körzetszám előtt). A '+' elé '\' –t kell tennünk, mivel ennek a jelnek önálló jelentése is van reguláris kifejezésben. A zárójelek közé „nyers" karaktereket (is) írhatunk. Ezután ismerős következik: az [1-9][0-9] –et már nem okozhat gondot megérteni az utána következő {1,2} –t már inkább. Ezzel azt közöltük, hogy az előtte lévő meghatározás ([0-9]) legalább egy legfeljebb két alkalommal szerepel egymás után. Végül a \s egy szóközt jelöl.

(\d{3}(-)){2}\d{3}$: ez a kifejezés a kötőjellel elválasztott számblokkokat jelöli. A végén lévő $ jel jelzi, hogy a karaktersorozat végéig kell illesztenie, vagyis az egészet ellenőrizze (máskülönben a 345-345-345-456 sorozat érvényes lenne, hiszen benne van az amit kerestünk).
Haladjunk továbbra is a végéről a {3}–ról már tudjuk, hogy azt jelenti: az előtte lévő kifejezésnek pontosan háromszor kell szerepelnie, ez jelen esetben azt jelenti, hogy a minta végén három számjegy áll, amelyet a \d–vel (d mint digit) jelölünk.
A \d előtt szereplő {2} az egész zárójelben lévő kifejezésre vonatkozik, ahol megmondtuk, hogy három számjegy után egy kötőjel következik.

A .NET reguláris kifejezései meglepően sokrétűek, rengeteg lehetőségünk van, a fenti két példa csak ízelítő volt. További információt találhatunk az MSDN megfelelő oldalán:

http://msdn.microsoft.com/en-us/library/az24scfc(v=VS.90).aspx

11 Gyakorló feladatok II.

11.1 Minimum- és maximumkeresés

Keressük ki egy tömb legnagyobb, illetve legkisebb elemét és ezek indexeit (ha több ilyen elem van, akkor elég az első előfordulás).

Megoldás (11/MinMax.cs)

A mininum/maximum-keresés a legalapvetőbb algoritmusok egyike. Az alapelv rendkívül egyszerű, végigmegyünk a tömb elemein, és minden elemet összehasonlítunk az aktuális legkisebbel/legnagyobbal. Nézzük is meg a forráskódot (csak a lényeg szerepel itt, a tömb feltöltése nem okozhat gondot):

```
int min = 1000;
int max = -1;
int minIdx = 0;
int maxIdx = 0;

for(int i = 0;i < 30;++i)
{
	if(array[i] < min)
	{
		min = array[i];
		minIdx = i;
	}

	if(array[i] > max)
	{
		max = array[i];
		maxIdx = i;
	}
}
```

A *min* és *max* változóknak kezdőértéket is adtunk, ezeket úgy kell megválasztani, hogy ezek biztosan kisebbek illetve nagyobbak legyenek, mint a tömb elemei (feltesszük, hogy nullánál kisebb és ezernél nagyobb szám nincs a tömbben).

Értelemszerűen, mivel minden elemet kötelezően meg kell vizsgálnunk, ez az algoritmus nem túl gyors nagy elemszám esetén, viszont nagyon jól működik párhuzamos környezetben – a tömb egyes részeit külön feldolgozóegység kapja meg.

11.2 Szigetek

Egy szigetcsoport fölött elrepülve bizonyos időközönként megnéztük, hogy épp hol vagyunk. Ha sziget (szárazföld) fölött, akkor leírtunk egy egyest, ha tenger fölött, akkor nullát.
A programunk ezt az adatot dolgozza föl, amelyet vagy a billentyűzetről vagy – ha van – a parancssori paraméterből kap meg. A feladatok:

- ✓ Adjuk meg a leghosszabb egybefüggő szárazföld hosszát.
- ✓ Adjuk meg, hogy hány szigetet találtunk.

Megoldás (11/Islands.cs)

A megoldásban csak az adatok feldolgozását nézzük meg, a beolvasásuk nem okozhat mostanra gondot. A szigeteken végzett méréseket a *data* nevű, *string* típusú változóban tároltuk el.

A két feladatot egyszerre fogjuk megoldani, mivel a programunk a következő elven alapul: a *data string*en fogunk keresztülmenni egy ciklus segítségével. Ha a *string* adott indexén egyest találunk, akkor elindítunk egy másik ciklust, amely attól az indextől megy egészen addig, amíg nullához nem ér. Ekkor egyrészt megnöveljük eggyel a szigetek számát tároló változót, másrészt tudni fogjuk, hogy milyen hosszú volt a sziget és összehasonlíthatjuk az eddigi eredményekkel.
Nézzük is meg a forráskódot:

```
int islandCount = 0;
int maxIslandLength = 0;
int i = 0;

while(i < data.Length)
{
	if(data[i] == '1')
	{
		++islandCount;
		int j = i;
		int tmp = 0;
		while(j < data.Length && data[j] == '1')
		{
			++j;
			++tmp;
		}

		i = j;

		if(tmp > maxIslandLength){ maxIslandLength = tmp; }
	}
	else
	{
		++i;
	}
}
```

A kódban két érdekes dolog is van, az első a belső ciklus feltétele: ellenőrizzük, hogy még a szigetet mérjük és azt is, hogy nem-e értünk a *string* végére. Ami fontos, az a feltételek sorrendje: mivel tudjuk, hogy a feltételek kiértékelése balról jobbra halad, ezért először azt kell vizsgálnunk, hogy helyes indexet használunk-e, ellenkező esetben ugyanis kivételt kapnánk (hiszen ha a *string* vége után vagyunk ott már nincs semmi).

A másik érdekesség a két ciklusváltozó. Amikor befejezzük a belső ciklust, akkor a külső ciklusváltozót új pozícióba kell helyeznünk, méghozzá oda, ahol a belső ciklus abbahagyta a vizsgálatot.

11.3 Átlaghőmérséklet

Az év minden napján megmértük az átlaghőmérsékletet, az eredményeket pedig egy mátrixban tároltuk (az egyszerűség kedvéért tegyük fel, hogy minden hónap harminc napos, az eredményeket pedig véletlenszám-generátorral (ésszerű kereteken belül) sorsoljuk ki).

- ✓ Keressük meg az év legmelegebb és leghidegebb napját!
- ✓ Adjuk meg az év legmelegebb és leghidegebb hónapját!
- ✓ Volt –e egymást követő öt nap (egy hónapon belül), amikor mínusz fokot mértünk?

Megoldás (11/Temperature.cs)

Ehhez a feladathoz nem tartozik írásos megoldás, tulajdonképpen egy minimum- és maximum-kiválasztásról van szó, csak éppen kétdimenziós tömbben (viszont a jegyzethez csatolva van egy lehetséges megoldás).

11.4 Buborékrendezés

Valósítsuk meg egy tömbön a buborékrendezést.

Megoldás (11/BubbleSort.cs)

A buborékos rendezés egy alapvető rendezési algoritmus, amelynek alapelve, hogy a kisebb elemek – buborék módjára – felszivárognak, míg a nagyobb elemek lesüllyednek. Ennek az algoritmusnak többféle implementációja is létezik, mi most két változatát is megvizsgáljuk. Az első így néz ki:

```
for(int i = 0;i < array.Length - 1;++i)
{
      for(int j = array.Length - 1;j > i;--j)
      {
            if(array[j - 1] > array[j])
            {
                  int tmp = array[j];
                  array[j] = array[j - 1];
                  array[j - 1] = tmp;
            }
      }
}
```

Kezdjük a belső ciklussal. Ez a tömb végéről fog visszafelé menni és cserélgeti az elemeket, hogy a legkisebbet vigye tovább magával. Legyen pl. a tömb utolsó néhány eleme:

10 34 5

Fogjuk az 5 –öt (*array[j]*) és összehasonlítjuk az előtte levő elemmel, ami a 34 (*array[j-1]*). Mivel nagyobb nála, ezért megcseréljük a kettőt:

10 5 34

Ezután csökkentjük a ciklusváltozót, ami most megint az eddigi legkisebb elemre az 5 –re fog mutatni és cserélhetjük tovább. Természetesen, ha kisebb elemet találunk, akkor ezután őt fogjuk tovább vinni, egészen addig, amíg a legkisebb elem elfoglalja a tömb első indexét. Itt jön képbe a külső ciklus, ami azt biztosítja, hogy a rendezett elemeket már ne vizsgáljuk, hiszen a belső ciklus minden futásakor a tömb elejére tesszük az aktuális legkisebb elemet.

Nézzünk meg egy másik megoldást is:

```
for(int i = 1;i < array.Length;++i)
{
      int y = array[i];
      int j = i - 1;
      while(j > -1 && y < array[j])
      {
            array[j + 1] = array[j];
            --j;
      }

      array[j + 1] = y;
}
```

Itt lényegében ugyanarról van szó, csak most elölről vizsgáljuk az elemeket.
Nem árt tudni, hogy a buborékos rendezés csak kis elemszám esetében hatékony, nagyjából O(n^2) nagyságrendű.

Az O() (ún. nagy ordó) jelölést használjuk egy algoritmus futásidejének megbecsülésére (illetve használják a matematika más területein is).

12 Objektum-orientált programozás - elmélet

A korai programozási nyelvek nem az adatokra, hanem a műveletekre helyezték a hangsúlyt, mert akkoriban még főleg matematikai számításokat végeztek a számítógépekkel. Ahogy aztán a számítógépek széles körben elterjedtek, megváltoztak az igények, az adatok pedig túl komplexekké váltak ahhoz, hogy a procedurális módszerrel kényelmesen és hatékonyan kezelni lehessen őket.

Az első objektum-orientált programozási nyelv a Simula 67 volt. Tervezői (Ole-Johan Dahl és Kristen Nygaard) hajók viselkedését szimulálták és ekkor jött az ötlet, hogy a különböző hajótípusok adatait egy egységként kezeljék, így egyszerűsítve a munkát.

Az OOP már nem a műveleteket helyezi a középpontba, hanem az egyes adatokat (adatszerkezeteket) és a közöttük levő kapcsolatokat (hierarchiát).

Ebben a fejezetben az OOP elméleti oldalával foglalkozunk, a cél a paradigma megértése, gyakorlati példákkal a következő részekben találkozhatunk (szintén a következő részekben található meg néhány elméleti fogalom is, amelyek gyakorlati példákon keresztül érthetőbben megfogalmazhatók, ezért ezek csak később lesznek tárgyalva,pl. polimorfizmus).

12.1 UML

Az OOP tervezés elősegítésére hozták létre az UML –t (Unified Modelling Language). Ez egy általános tervezőeszköz, a célja, hogy egy minden fejlesztő által ismert közös jelrendszert valósítson meg. A következőkben az UML eszközeit fogjuk felhasználni az adatok közötti relációk grafikus ábrázolásához.

12.2 Osztály

Az OOP világában egy osztály olyan adatok és műveletek összessége, amellyel leírhatjuk egy modell (vagy entitás) tulajdonságait és működését. Legyen például a modellünk a kutya állatfaj. Egy kutyának vannak tulajdonságai (pl. életkor, súly, stb.) és van meghatározott viselkedése (pl. csóválja a farkát, játszik, stb.)

Az UML a következőképpen jelöl egy osztályt:

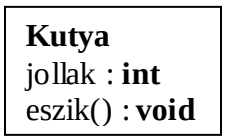
Amikor programot írunk, akkor az adott osztályból (osztályokból) létre kell hoznunk egy (vagy több) példányt, ezt pédányosításnak nevezzük. Az osztály és példány közötti különbségre jó példa a recept (osztály) és a sütemény (példány).

12.3 Adattag és metódus

Egy objektumnak az életciklusa során megváltozhat az állapota, tulajdonságai. Ezt az állapotot valahogy el kell tudnunk tárolni, illetve biztosítani kell a szükséges műveleteket a tulajdonságok megváltoztatásához (pl. a kutya eszik (ez egy művelet), ekkor megváltozik a „jóllakottság” tulajdonsága/állapota).

A tulajdonságokat tároló változókat adattagoknak (vagy mezőnek), a műveleteket metódusoknak nevezzük. A műveletek összességét felületnek is hívjuk.

Módosítsuk a diagramunkat:



Az adattagokat *név: típus* alakban ábrázoljuk, a metódusokat pedig *név(paraméterlista): visszatérési_érték* formában. Ezekkel a fogalmakkal egy későbbi fejezet foglalkozik majd.

12.4 Láthatóság

Az egyes tulajdonságokat és metódusokat nem feltétlenül kell közszemlére bocsátani. Az OOP egyik alapelve, hogy a felhasználó csak annyi adatot kapjon meg, amennyi feltétlenül szükséges. A kutyás példában az *eszik()* művelet magába foglalja a rágást, nyelést, emésztést is, de erről nem fontos tudniuk, csak az evés ténye számít.

Ugyanígy egy tulajdonság (adattag) esetében sem jó, ha mindenki hozzájuk fér (az elfogadható, ha a közvetlen család hozzáfér a számlámhoz, de idegenekkel nem akarom megosztani).

Az ős-OOP szabályai háromféle láthatóságot fogalmaznak meg (ez nyelvtől függően bővülhet), a C# láthatóságairól a következő részekben lesz szó.

A háromféle láthatóság:

- **Public**: mindenki láthatja (UML jelölés: **+**).
- **Private**: csakis az osztályon belül elérhető, illetve a leszármazott osztályok is láthatják, de nem módosíthatják (a származtatás és öröklődés hamarosan jön) (UML jelölés: **-**).
- **Protected**: ugyanaz, mint a private, de a leszármazott osztályok módosíthatják is (UML jelölés: #).

A *Kutya* osztály most így néz ki:

Kutya
-jollak : **int** +eszik() : **void**

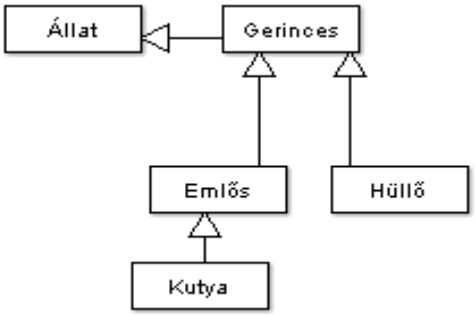
12.5 Egységbezárás

A „hagyományos”, nem OO programnyelvek (pl. a C) az adatokat és a rajtuk végezhető műveleteket a program külön részeiként kezelik. Bevett szokás ezeket elkülöníteni egy önálló forrásfileba, de ez még mindig nem elég biztonságos. A kettő között nincs összerendelés, ezért más programozók gond nélkül átírhatják egyiket vagy másikat, illetve hozzáférnek a struktúrákhoz és nem megfelelően használhatják fel azokat.

Az OO paradigma egységbe zárja az adatokat és a hozzájuk tartozó felületet, ez az ún. egységbezárás (*encapsulation* vagy *information hiding*). Ennek egyik nagy előnye, hogy egy adott osztály belső szerkezetét gond nélkül megváltoztathatjuk, mindössze arra kell figyelni, hogy a felület ne változzon (pl. egy autót biztosan tudunk kormányozni, attól függetlenül, hogy az egy személyautó, traktor vagy Forma-1–es gép).

12.6 Öröklődés

Az öröklődés vagy származtatás az új osztályok létrehozásának egy módja. Egy (vagy több) már létező osztályból hozunk létre egy újat úgy, hogy az minden szülőjének tulajdonságát örökli vagy átfogalmazza azokat. A legegyszerűbben egy példán keresztül érthető meg. Legyen egy „Állat” osztályunk. Ez egy eléggé tág fogalom, ezért szűkíthetjük a kört, mondjuk a „Gerinces Állatok”–ra. Ezen belül megkülönböztethetünk „Emlős” –t vagy „Hüllő” –t. Az „Emlős” osztály egy leszármazottja lehet a „Kutya” és így tovább. Az öröklődést specializálásnak is nevezik. A specializálás során az osztályok között ún. „az-egy” (is-a) reláció áll fenn. Így amikor azt mondjuk, hogy a „Kutya” az egy „Állat” akkor arra gondolunk, hogy a „Kutya” egy specializáltabb forma, amelynek megvan a saját karakterisztikája, de végeredményben egy „Állat”. Ennek a gondolatmenetnek a gyakorlati felhasználás során lesz jelentősége.



A diagram a fenti példát ábrázolja. UML-ül az öröklődést „üres” nyíllal jelöljük, amely a specializált osztály felől mutat az általánosabbra. Az osztályok között fennálló kapcsolatok összességét hierarchiának nevezzük.

Előfordul, hogy nem fontos számunkra a belső szerkezet, csak a felületet szeretnénk átörökíteni, hogy az osztályunkat fel tudja használni a programunk egy másik része (ilyen például a már említett *foreach* ciklus). Ilyenkor nem egy „igazi” osztályról, hanem egy interfészről – felületről – beszélünk, amelynek nincsenek adattagjai, csakis a műveleteket deklarálja.

A C# rendelkezik önálló interfészekkel, de ez nem minden programnyelvre igaz, ezért ők egy hasonló szerkezetet, ún. absztrakt osztályokat használnak. Ezekben előfordulnak adattagok is, de leginkább a felület definiálására koncentrálnak. A C# nyelvi szinten támogat absztrakt osztályokat is, a kettő között ugyanis lényegi különbség van. Az interfészek az osztálytól függetlenek, csakis felületet biztosítanak (például az *IEnumerable* és *IEnumerator* a *foreach*–nek (is) biztosítanak felületet, az nem lényeges, hogy milyen osztályról van szó). Az absztrakt osztályok viszont egy őshöz kötik az utódokat (erre az esetre példa az „Állat” osztály, amelyben mondjuk megadunk egy absztrakt „evés” metódust, amit az utódok megvalósítanak - egy krokodil nyílván máshogy eszik mint egy hangya, de az evés az állatokhoz köthető valami, ezért közös).

13 Osztályok

Osztályt a *class* kulcsszó segítségével deklarálhatunk:

```
using System;

class Dog
{
}

class Program
{
      static public void Main(string[] args)
      {

      }
}
```

Látható, hogy a főprogram és a saját osztályunk elkülönül.

Konvenció szerint az osztálynév mindig nagybetűvel kezdődik.

Felmerülhet a kérdés, hogy honnan tudja a fordító, hogy melyik a „fő” osztály? A helyzet az, hogy a *Main* egy speciális metódus, ezt a fordító automatikusan felismeri és megjelöli, mint a program belépési pontját. Igazából lehetséges több *Main* nevű metódust is létrehozni, ekkor a fordítóprogramnak meg kell adni (a */main* kapcsoló segítségével), hogy melyik az igazi belépési pont. Ettől függetlenül ezt a megoldást, ha csak lehet, kerüljük el.

Nézzünk egy példát:

```
using System;

class Program1
{
      static public void Main()
      {
            Console.WriteLine("Program1");
      }
}

class Program2
{
      static public void Main()
      {
            Console.WriteLine("Program2");
      }
}
```

Ezt a forráskódot így fordíthajuk le:

```
csc /main:Program1 main.cs
```

Futtatáskor a „Program1” szöveget fogja kiírni a program.

A fordító további kapcsolóiról tudhatunk meg többet, ha parancssorba beírjuk, hogy:

```
csc -help
```

A fenti példában a lehető legegyszerűbb osztályt hoztuk létre. A C++ nyelvet ismerők figyeljenek arra, hogy az osztály-deklaráció végén nincs pontosvessző.

Az osztályunkból a *new* operátor segítségével tudunk készíteni egy példányt.

```
Dog d = new Dog();
```

A *new* hívásakor lefut a konstruktor, megfelelő nagyságú hely foglalódik a memóriában, ezután pedig megtörténik az adattagok inicializálása is.

13.1 Konstruktorok

Minden esetben amikor példányosítunk egy speciális metódus a konstruktor fut le, melynek feladata, hogy „beállítsa” az osztály értékeit. Bár a fenti osztályunkban nem definiáltunk semmi ilyesmit ettől függetlenül rendelkezik alapértelmezett (azaz paraméter nélküli) konstruktorral. Ez igaz minden olyan osztályra, amelynek nincs konstruktora (amennyiben bármilyen konstruktort létrehoztunk, akkor ez a lehetőség megszűnik).

Az alapértelmezett konstruktor legelőször meghívja a saját ősosztálya alapértelmezett konstruktorát. Ha nem származtattunk direkt módon (mint a fenti programban), akkor ez a *System.Object* konstruktora lesz (tulajdonképpen ez előbb vagy utóbb mindenképpen meghívódik, hiszen az ősosztály konstruktora is meghívja a saját ősét és így tovább).

Abban az esetben, ha az ősosztály nem tartalmaz alapértelmezett konstruktort (mert van neki paraméteres), akkor valamely másik konstruktorát explicit módon hívni kell a leszármazott osztály konstruktorából a *base* metódussal, minden más esetben fordítási hiba az eredmény.

```
class Base
{
      public Base(string s){ }
}

class Derived : Base
{

}
```

Ez a forráskód(részlet) nem fordul le, mivel a *Base* osztály csakis paraméteres konstruktorral rendelkezik.

```
class Base
{
      public Base(string s){ }
}

class Derived : Base
{
      public Derived() : base("abc"){ }
}
```

Így már viszont működni fog.

A *base* nem összekeverendő a *Base* nevű osztállyal, ez egy önálló metódus, amely minden esetben az ősosztály valamely konstruktorát hívja.

Az alapértelmezett konstruktor valamilyen formában minden esetben lefut, akkor is, ha az osztályban deklaráltunk paraméterest, hiszen továbbra is ez felel a memóriafoglalásért.

Egy osztály példányosításához legalább egy *public* elérhetőségű konstruktorra van szükség, egyébként nem fordul le a program.

Az adattagok – ha vannak – automatikusan a nekik megfelelő nullértékre inicializálódnak(pl: *int*-> 0, *bool* ->false, referencia- és nullabletípusok ->*null*).

A C++ nyelvet ismerők vigyázzanak, mivel itt csak alapértelmezett konstruktort kapunk automatikusan, értékadó operátort illetve másoló konstruktort nem. Ugyanakkor minden osztály a *System.Object* –ből származik (még akkor is ha erre nem utal semmi), ezért néhány metódust (például a típus lekérdezéséhez) a konstruktorhoz hasonlóan azonnal használhatunk.

Jelen pillanatban az osztályunkat semmire nem tudjuk használni, ezért készítsünk hozzá néhány adattagot és egy konstruktort:

```
using System;

class Dog
{
      private string _name;
      private int _age;

      public Dog(string name, int age)
      {
            this._name = name;
            this._age = age;
      }
}

class Program
{
      static public void Main()
      {
            Dog d = new Dog("Rex", 2);
      }
}
```

A konstruktor neve meg kell egyezzen az osztály nevével és semmilyen visszatérési értéke nem lehet. A mi konstruktorunk két paramétert vár, a nevet és a kort (metódusokkal és paramétereikkel a következő rész foglalkozik bővebben).

Egy osztálynak paraméterlistától függően bármennyi konstruktora lehet és egy konstruktorból hívhatunk egy másikat a *this*–szel:

```
class Test
{
      public Test() : this(10){ }
      public Test(int x){ }
}
```

Ha több konstruktor is van, akkor a paraméter típusához leginkább illeszkedő fut le.

A példában a konstruktor törzsében értéket adtunk a mezőknek a *this* hivatkozással, amely mindig arra a példányra mutat, amelyen meghívták (a *this* kifejezés így minden olyan helyen használható, ahol az osztálypéldányra van szükség). Nem kötelező használni, ugyanakkor hasznos lehet, hogy ha sok adattag/metódus van, illetve ha a paraméterek neve megegyezik az adattagokéval. A fordítóprogram automatikusan „odaképzeli” magának a fordítás során, így mindig tudja mivel dolgozik.

Az adattagok *private* elérésűek (ld. elméleti rész), azaz most csakis az osztályon belül használhatjuk és módosíthatjuk őket, például a konstruktorban, ami viszont publikus.

Nem csak a konstruktorban adhatunk értéket a mezőknek, hanem használhatunk ún. inicializálókat is:

```
class Dog
{
      private string _name = "Rex";
      private int _age = 5;

      public Dog(string name, int age)
      {
            this._name = name;
            this._age = age;
      }
}
```

Az inicializálás mindig a konstruktor előtt fut le, ez egyben azt is jelenti, hogy az utóbbi felülbírálhatja. Ha a *Dog* osztálynak ezt a módosított változatát használtuk volna fentebb, akkor a példányosítás során minden esetben felülírnánk az alapértelmezettnek megadott kort.

Az inicializálás sorrendje megegyezik a deklarálás sorrendjével (felülről lefelé halad).

A konstruktorok egy speciális változata az ún. másoló- vagy copy-konstruktor. Ez paramétereként egy saját magával megegyező típusú objektumot kap és annak értékeivel inicializálja magát. Másoló konstruktort általában az értékadó operátorral szoktak implementálni, de az operátor-kiterjesztés egy másik fejezet témája, így most egyszerűbben oldjuk meg:

```
class Dog
{
      private string _name = "Rex";
      private int _age = 5;

      public Dog(string name, int age)
      {
            this._name = name;
            this._age = age;
      }

      public Dog(Dog otherDog)
            : this(otherDog._name, otherDog._age)
      { }
}
```

A program első ránézésre furcsa lehet, mivel privát elérhetőségű tagokat használunk, de ezt minden gond nélkül megtehetjük, mivel a C# a privát elérhetőséget csak osztályon kívül érvényesíti, ugyanolyan típusú objektumok látják egymást.

Most már használhatjuk is az új konstruktort:

```
Dog d = new Dog("Rex", 2);
Doge = new Dog(d);
```

13.2 Adattagok

Az adattagok vagy mezők olyan változók, amelyeket egy osztályon (vagy struktúrán) belül deklaráltunk. Az adattagok az osztálypéldányhoz tartoznak (azaz minden egyes példány különálló adattagokkal rendelkezik) vele születnek és „halnak” is meg. Az eddigi példáinkban is használtunk már adattagokat, ilyenek voltak a *Dog* osztályon belüli *_name* és *_age* változók.

Az adattagokon használhatjuk a *const* típusmódosítót is, ekkor a deklarációnál értéket kell adnunk a mezőnek, hasonlóan az előző fejezetben említett inicializáláshoz. Ezek a mezők pontosan ugyanúgy viselkednek mint a hagyományos konstansok.

Egy konstans mezőt nem lehet statikusnak (statikus tagokról hamarosan) jelölni, mivel a fordító egyébként is úgy fogja kezelni (ha egy adat minden objektumban változatlan, felesleges minden alkalommal külön példányt készíteni belőle), vagyis minden konstans adattagból globálisan – minden példányra vonatkozóan egy darab van.

A mezőkön alkalmazható a *readonly* módosító is, ez két dologban különbözik a konstansoktól: az értékadás elhalasztható a konstruktorig és az értékül adott kifejezés eredményének nem szükséges ismertnek lennie fordítási időben.

13.3 Láthatósági módosítók

A C# ötféle módosítót ismer:

- **public**: az osztályon/struktúrán kívül és belül teljes mértékben hozzáférhető.
- **private**: csakis a tartalmazó osztályon belül látható, a leszármazottak sem láthatják, osztályok/struktúrák esetében az alapértelmezés.
- **protected**: csakis a tartalmazó osztályon és leszármazottain belül látható.
- **internal**: csakis a tartalmazó (és a barát) assembly(ke)n belül látható.
- **protected internal**: a *protected* és *internal* keveréke.

Ezek közül leggyakrabban az első hármat fogjuk használni.

13.4 Parciális osztályok

C# nyelvben létrehozhatunk ún. parciális (darab, töredék) osztályokat (*partial class*), ha egy osztály-deklarációban használjuk a *partial* kulcsszót (ezt minden darabnál meg kell tennünk). Egy parciális osztály definíciója több részből (tipikusan több forrásfile-ból) állhat. Egy parciális osztály minden töredékének ugyanazzal a láthatósági módosítóval kell rendelkeznie, valamint az egyik résznél alkalmazott egyéb módosítók (pl. *abstract*), illetve az ősosztály deklaráció a teljes osztályra (értsd: minden töredékre) érvényes lesz (ebből következik, hogy ezeket nem kötelező feltüntetni minden darabnál). Ugyanakkor ennél az utolsó feltételnél figyelni kell arra, hogy ne adjunk meg egymásnak ellentmondó módosítókat (pl. egy osztály nem kaphat *abstract* és *sealed* módosítókat egyidőben).

Nézzünk egy példát:

```
// main.cs
using System;

partial class PClass
{

}

class Program
{
      static public void Main()
      {
            PClass p = new PClass();
            p.Do();
      }
}
```

Látható, hogy egy olyan metódust hívtunk, amelynek hiányzik a deklarációja.

Készítsünk egy másik forrásfilet is:

```
// partial.cs
using System;

partial class PClass
{
	public void Do()
	{
		Console.WriteLine("Hello!");
	}
}
```

A két filet így tudjuk fordítani:

```
csc main.cs partial.cs
```

A .NET a parciális osztályokat főként olyan esetekben használja, amikor az osztály egy részét a fordító generálja (pl. a grafikus felületű alkalmazásoknál a kezdeti beállításokat az *InitializeComponent* metódus végzi, ezt teljes egészében a fordító készíti el). Ennek a megoldásnak az a nagy előnye, hogy könnyen ki tudjuk egészíteni ezeket a generált osztályokat.

Bármelyik osztály (tehát a nem-parciális is) tartalmazhat beágyazott parciális osztályt, ekkor értelemszerűen a töredékek a tartalmazó osztályon belül kell legyenek (ugyanez nem vonatkozik a parciális osztályon belül lévő parciális osztályokra, ott a beágyazott osztályok töredékei szétoszolhatnak a tartalmazó osztály darabjai között).

Egy parciális osztály darabjainak ugyanabban az assemblyben kell lenniük.

A C# 3.0 már engedélyezi parciális metódusok használatát is, ekkor a metódus deklarációja és definíciója szétoszlik:

```
partial class PClass
{
	partial void Do();
}

partial class PClass
{
	partial void Do()
	{
		Console.WriteLine("Hello!");
	}
}
```

Parciális metódusnak nem lehet elérhetőségi módosítója (épp ezért minden esetben *private* elérésű lesz) valamint *void*-dal kell visszatérnie.

A *partial* kulcsszót ilyenkor is ki kell tenni minden előfordulásnál. Csakis parciális osztály tartalmazhat parciális metódust.

13.5 Beágyazott osztályok

Egy osztály tartalmazhat metódusokat, adattagokat és más osztályokat is. Ezeket a „belső” osztályokat beágyazott (*nested*) osztálynak nevezzük. Egy ilyen osztályt általában elrejtünk, de ha mégis publikus elérésűnek deklaráljuk, akkor a „külső” osztályon keresztül érhetjük el. A beágyazott osztályok alapértelmezés szerint privát elérésűek.

```
class Outer
{
      class Inner
      {
            // a beágyazott osztály nem látható
      }
}

class Outer
{
      public class Inner
      {
            // így már látszik
      }
}

Outer.Inner x = new Outer.Inner(); // példányosítás
```

Egy beágyazott osztály hozzáfér az őt tartalmazó osztálypéldány minden tagjához (beleértve a *private* elérésű tagokat és más beágyazott osztályokat is), de csakis akkor, ha a beágyazott osztály tárol egy, a külső osztályra hivatkozó referenciát:

```
class Outer
{
      private int value = 11;
      private Inner child;

      public Outer()
      {
            child = new Inner(this);
      }

      public void Do()
      {
            child.Do();
      }

      class Inner
      {
            Outer parent;

            public Inner(Outer o)
            {
                  parent = o;
            }

            public void Do()
            {
                  Console.WriteLine(parent.value);
            }
      }
}
```

13.6 Objektum inicializálók

A C# 3.0 objektumok példányosításának egy érdekesebb formáját is tartalmazza:

```
using System;

class Person
{
      public Person(){ }

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

class Program
{
      static public void Main()
      {
            Person p = new Person()
            {
                  Name = "István"
            };

            Console.WriteLine(p.Name);
      }
}
```

Ilyen esetekben vagy egy nyilvános tagra, vagy egy tulajdonságra hivatkozunk (ez utóbbit használtuk). Természetesen, ha létezik paraméteres konstruktor, akkor is használhatjuk ezt a beállítási módot.

13.7 Destruktorok

A destruktorok a konstruktorokhoz hasonló speciális metódusok, amelyek az osztály által használt erőforrások felszabadításáért felelősek.

A .NET ún. automatikus szemétgyűjtő (*garbage collector*) rendszert használ, amelynek lényege, hogy a hivatkozás nélküli objektumokat (nincs rájuk mutató érvényes referencia) a keretrendszer automatikusan felszabadítja.

```
MyClass mc = new MyClass(); // mc egy MyClass objektumra mutat
mc = null; // az objektumra már nem mutat semmi, felszabadítható
```

Objektumok alatt ebben a fejezetben csak és kizárólag referencia-típusokat értünk, az értéktípusokat nem a GC kezeli.

A szemétgyűjtő működése nem determinisztikus, azaz előre nem tudjuk megmondani, hogy mikor fut le, ugyanakkor kézzel is meghívható, de ez nem ajánlott. A következő példában foglalunk némi memóriát, majd megvizsgáljuk, hogy mi történik felszabadítás előtt és után:

```
using System;

class Program
{
	static public void Main()
	{
		Console.WriteLine("Foglalt memória: {0}",
			GC.GetTotalMemory(false));

		for(int i = 0;i < 10;++i)
		{
			int[] x = new int[1000];
		}

		Console.WriteLine("Foglalt memória: {0}",
			GC.GetTotalMemory(false));

		GC.Collect(); // meghívjuk a szemétgyűjtőt

		Console.WriteLine("Foglalt memória: {0}",
			GC.GetTotalMemory(false));
	}
}
```

A GC osztály *GetTotalMemory* metódusa a program által lefoglalt byte -ok számát adja vissza, paraméterként megadhatjuk, hogy meg szeretnénk –e hívni a szemétgyűjtőt.

A fenti program kimenete valami ilyesmi kell legyen:

```
Foglalt memória: 21060
Foglalt memória: 70212
Foglalt memória: 27144
```

Vegyük észre, hogy a ciklusban létrehozott tömbök minden ciklus végén eltüntethetőek, mivel nincs többé rájuk hivatkozó referencia (hiszen a lokális objektumok hatóköre ott véget ér).

De hogyan is működik a szemétgyűjtő? A .NET ún. generációs garbage collector-t használ, amely abból a feltevésből indul ki, hogy a legfrissebben létrehozott objektumok lesznek leghamarabb felszabadíthatóak (ez az ún. generációs hipotézis) (gondoljunk csak a lokális változókra, amelyeket viszonylag sokat használunk).

Ez alapján a következő történik: minden friss objektum a nulladik – legfiatalabb – generációba kerül. Amikor eljön a szemétgyűjtés ideje, a GC először ezt a generációt vizsgálja meg, és ha talál hivatkozás nélküli objektumot azt törli (pontosabban az elfoglalt memóriaterületet szabadnak jelöli), a maradékot pedig átrakja az első generációba. Ezután sorban átvizsgálja a többi generációt (még kettő van) és elvégzi a megfelelő módosításokat. Értelemszerűen a második generációs objektumok akkor törlődnek, ha a program megáll (illetve elfogyhat a memória is, ekkor *OutOfMemoryException* kivétel keletkezik). Az egyes generációk összefüggő memóriaterületen vannak, így kevesebbet kell dolgoznia a gyűjtőnek.

Az is érdekes kérdés, hogy honnan tudja a GC, hogy melyik objektumok feleslegesek. Ehhez be kell vezetnünk két fogalmat, a gyenge- illetve erős referenciák (*weak*- és *strong-reference*) intézményét:

- Minden olyan objektumra, amelyet a *new* operátorral hozunk létre, erős referenciával hivatkozunk, ezek „normális" objektumok, amelyek akkor és csakis akkor kerülnek hatókörön kívülre (és takaríthatóak el a GC által), ha nincs rájuk hivatkozó érvényes referencia.

- A gyenge referenciák ennek épp ellenkezőjét nyújtják, bármikor törölhető az általuk mutatott objektum ha nincs elég memória – akkor is ha létezik rá mutató érvényes – gyenge – hivatkozás.

A .NET nyelvi szinten támogatja a gyenge referenciákat:

```
int[][] array = new int[10][];
for(int i = 0;i < 10;++i)
{
      array[i] = new int[1000];
}

WeakReference wr = new WeakReference(array);
array = null;
```

Ebben a példában miután az eredeti tömbhivatkozást *null*–ra állítottuk, a tömb objektumokra már csak gyenge referencia mutat, azaz bármikor eltakaríthatóak.

Egy *WeakReference* objektumból „visszanyerhető" az eredeti erős referencia a *Target* tulajdonság segítségével, viszont ne felejtsük el ellenőrizni ennek nullértékét, mert lehet, hogy már átment rajta a GC (és konvertálnunk is kell, mivel *object* típussal tér vissza):

```
WeakReference wr = new WeakReference(array);
array = null;

if(wr.Target != null)
{
      int[][]array = (int[][])wr.Target;
}
```

A másik fogalom, amelyet ismernünk kell, az az ún. *application root* objektumok. Ezek olyan objektumok, amelyekről feltételezhetjük, hogy elérhetőek (ilyen objektumok lesznek pl. az összes lokális és globális változó). A GC mindig a root objektumokat vizsgálja meg először, és rajtuk keresztül építi fel a memóriatérképet.

Most már tisztában vagyunk az alapokkal, vizsgáljuk meg, hogy mi történik valójában. Azt mondtuk, hogy a GC átvizsgálja a generációkat, ennek azonban van egy kis hátránya, mégpedig az, hogy lassú. Ahhoz, hogy a takarítás valóban hatékony legyen, fel kell függeszteni a program futását, vagy azzal párhuzamosan dolgozni. Mindkét esetben rosszul járunk, hiszen vagy „lefagy" a program egy időre, vagy kevesebb erőforráshoz jut (ugyanakkor azt is számításba kell venni, hogy a GC a lehető legjobb időben – értsd: akkor amikor a program a legkevesebb erőforrást használja – fog beindulni, tehát nem feltétlenül fog igazán nagy gondot jelenteni az

alkalmazás szempontjából). Nyílván többmagos processzorral szerelt PC –knél jobb a helyzet, de attól még fennáll a hatékonyság problémája.

Épp ezért, ahelyett, hogy minden alkalommal teljes vizsgálatot végezne a GC, bevezették a részleges takarítás fogalmát, amely a következő feltevésre épül: az egyes illetve kettes generációkban lévő objektumok nagy valószínűséggel nem módosultak, vagyis feltehetjük, hogy van rájuk hivatkozó referencia (gondoljunk arra, hogy pl. a lokális változók szinte soha nem fognak átkerülni még az egyes generációba sem, vagyis az egyes és kettes generáció tagjai tipikusan hosszú életű objektumok lesznek). Természetesen ezt nem tudhatjuk biztosan, ezért minden .NET alkalmazáshoz automatikusan létrejön egy adatszerkezet (képzeljük el tömbként), amelynek egyes indexei a memória egy bizonyos nagyságú területének állapotát mutatják (nem az objektumokét!).

Tehát eljön a GC ideje, átválogatja a nulladik generációt, majd fogja a fenti adatszerkezetet (ún. *card table*) és megvizsgál minden olyan objektumot, amely olyan memóriaterületen fekszik amelyet a *card table* módosítottnak jelölt. Ez drámaian megnöveli a GC hatékonyságát, hiszen a teljes felhasznált memóriának csak kis részét kell megvizsgálnia.

A GC a nevével ellentétben nem csak ennyit tesz, valójában az ő dolga az objektumok teljes életciklusának a kezelése és a memória megfelelő szervezése is. Amikor elindítunk egy .NET programot, akkor a GC elsőként szabad memóriát kér az operációs rendszertől (a .NET ún. szegmensekre osztja a memóriát, minden szegmens 16 MB méretű), mégpedig kétszegmensnyit: egyet a hagyományos objektumoknak (GC Heap) és egyet a nagyméretű (100+ kilobyte) objektumoknak (LOH – Large Object Heap) (ez utóbbit csakis teljes takarításnál vizsgálja a GC). Ezután nyugodtan készíthetünk objektumokat, mert ha elfogy a hely, a GC automatikusan új szegmenseket fog igényelni.

Azt gondolná az ember, hogy ennyi az egész, de minden objektum életében eljön a pillanat, amikor visszaadja a lelkét a teremtőjének, nevezetesen a GC–nek. Ilyenkor rá hárul az a rendkívül fontos feladat is, hogy rendbe rakja a memóriát. Mit is értünk ez alatt? Hatékonyság szempontjából az a legjobb, ha az osztálypéldányok egymáshoz közel – lehetőleg egymás mellett – vannak a memóriában. Épp ezért a GC minden (fő)gyűjtőciklus (tehát teljes takarítás) alkalmával átmozgatja az objektumokat, hogy a lehető legoptimálisabban érjük el őket.

Ennek a megoldásnak egy hátulütője, hogy ilyen módon nem használhatunk unmanaged kódot, mivel ez teljes mértékben megakadályozza a pointerműveleteket. A megoldást az objektumok rögzítése (ún. pinning) jelenti, erről egy későbbi fejezet számol be.

A managelt kód éppen a fenti tények miatt tudja felvenni a versenyt a natív programokkal. Sőt, olyan alkalmazások esetében, ahol sokszor foglalunk és szabadítunk fel memóriát, a natív kód hátrányba is kerül(het).

Összességében azt mondhatjuk, hogy natív és managelt program között nincs nagy különbég sebesség tekintetében.

A GC háromféle módban tud működni:

- A GC párhuzamosan fut az alkalmazással
- A GC felfüggesztheti az alkalmazást
- Szerver mód

Nézzük az elsőt: az objektumok allokálását egy különálló szál végzi, ha szükség van tisztításra, akkor a program többi szálát csak nagyon rövid ideig függeszti fel és a takarítással egyidejűleg a program továbbra is helyet foglalhat a memóriában, kivéve ha az átlépte a maximálisan kiszabott keretet. Ez a limitálás a nulladik generációra vonatkozik, tehát azt szabja meg, hogy mennyi memóriát használhat fel egyszerre a G0 (amennyiben ezt az értéket elérjük, a GC beindul).Ezt a módszert olyan alkalmazásoknál használjuk, amikor fontos, hogy felhasználói felület reszponzív maradjon. Ez az alapértelmezett mód.

Hasonlóan működik a második is, viszont ő teljes mértékben „leállítja" az alkalmazást (ún. Stop-The-World módszer) a tisztítás idejére. Ez a mód sokkal kisebb G0 kerettel rendelkezik.

Szerver módban minden egyes processzor külön heap–pel és GC–vel rendelkezik. Ha egy alkalmazás kifut a memóriából, szól a GC–nek, amely felfüggeszti a program futását a tisztítás idejére.

Ha meg akarjuk változtatni egy program GC módját, szükségünk lesz egy konfigurációs filera (erről egy későbbi fejezetben), amely a következőket tartalmazza (a példában kikapcsoljuk a párhuzamos futást):

```
<configuration>
   <runtime>
      <gcConcurrent enabled="false"/>
   </runtime>
</configuration>
```

Vagy:

```
<configuration>
   <runtime>
      <gcServer enabled="true"/>
   </runtime>
</configuration>
```

Ezzel pedig a szervermódot állítottuk be.

Most pedig megnézzük, hogy hogyan használhatjuk a GC –t a gyakorlatban: A konstruktor(ok) mellett egy másik speciális metódus is jár minden referencia-típushoz, ez pedig a destruktor.

A GC megsemmisítés előtt az objektumokon meghívja a hozzájuk tartozó destruktort, másnéven *Finalizer*-t. Ennek a metódusnak a feladata, hogy felszabadítsa az osztály

által használt erőforrásokat (pl., hogy lezárja a hálózati kapcsolatokat, bezárjon minden egyes megnyitott file-t, stb.). Vegyük a következő kódot:

```
using System;

class DestructableClass
{
      public DestructableClass()
      {
            Console.WriteLine("Konstruktor");
      }

      ~DestructableClass()
      {
            Console.WriteLine("Destruktor");
      }
}

class Program
{
      static public void Main()
      {
            DestructableClass dc = newDestructableClass();
            Console.ReadKey();
      }
}
```

A destruktor neve *tilde* jellel (~) kezdődik, neve megegyezik az osztályéval, és nem lehet semmilyen módosítója vagy paramétere (értelemszerűen egy destruktor mindig privát elérhetőségű lesz, vagyis közvetlenül soha nem hívhatjuk, ez csakis a GC előjoga).

Soha ne készítsünk üres destruktort, mivel a GC minden destruktorról bejegyzést készít, és mindenképpen meghívja mindegyiket – akkor is, ha üres, vagyis ez egy felesleges metódushívás lenne.

Ha lefordítjuk ezt a kódot és elindítjuk a programot, először a „Konstruktor” szót fogjuk látni, majd egy gomb lenyomása után megjeleni a párja is (ha látni is akarjuk, nem árt parancssorból futtatni). A fenti kód valójában a következő formában létezik:

```
class DestructableClass
{
      public DestructableClass()
      {
            Console.WriteLine("Konstruktor");
      }

      protected override void Finalize()
      {
            try
            {
                  Console.WriteLine("Destruktor");
            }
            finally
            {
                  base.Finalize();
            }
      }
}
```

Ez a forráskód csak példa, nem fordul le, mivel a *Finalize* metódust nem definiálhatjuk felül, erre való a destruktor.

A *Finalize*-t minden referencia-típus örökli a *System.Object* –től. Először felszabadítja az osztály erőforrásait (a destruktorban általunk megszabott módon), azután meghívja az ősosztály *Finalize* metódusát (ez legalább a *System.Object* destruktora lesz) és így tovább, amíg a lánc végére nem ér:

```
using System;

class Base
{
      ~Base()
      {
            Console.WriteLine("Base");
      }
}

class Derived : Base
{
      ~Derived()
      {
            Console.WriteLine("Derived");
      }
}

class Program
{
      static public void Main()
      {
            Derived d = new Derived();
      }
}
```

A destruktorokra vonatkozik néhány szabály, ezek a következőek:

- Egy osztálynak csak egy destruktora lehet
- A destruktor nem örökölhető
- A destruktort nem lehet direkt hívni, a hívás mindig automatikusan történik
- Destruktora csakis osztálynak lehet, struktúrának nem

Legtöbbször felesleges destruktort készíteni, ez csak néhány speciális esetben szükséges, pl. amikor valamilyen unmanaged erőforrást (memória, file, stb...) használunk.

13.7.1 IDisposable

Az *IDisposable* interfész segítségével egy osztály által használt erőforrások felszabadítása kézzel – előre meghatározott időpontban – is megtörténhet, tehát nem kell a GC –re várni.

```
using System;

class DisposableClass : IDisposable
{
	public void Dispose()
	{
		// Takarítunk
		GC.SuppressFinalize(this);
	}
}

class Program
{
	static public void Main()
	{
		DisposableClass dc = new DisposableClass();
	}
}
```

Az interfész által deklarált *Dispose* metódusban meg kell hívnunk a *GC.SuppressFinalize* metódust, hogy jelezzük a GC –nek, hogy ez az osztály már felszabadította az erőforrásait, és nem kell destruktort hívnia.

A fenti kódban egy kicsit csaltunk: a *SuppressFinalize* csak akkor kell, ha valóban definiáltunk destruktort, egyébként felesleges.

Destruktorok használata helyett általában az ún. Dispose tervezési mintát alkalmazzuk, amely megvalósításához nyílván az *IDisposable* interfész lesz segítségünkre.

```
class DisposableClass : IDisposable
{
	private bool disposed = false;

	public void Dispose()
	{
		Dispose(true);
		GC.SuppressFinalize(this);
	}

	private void Dispose(bool disposing)
	{
		if(!disposed) {
			if(disposing) { // managed erõforrások felszabadítása }
			// unmanaged erõforrások felszabadítása
			disposed = true;
		}
	}

	~DisposableClass() { Dispose(false); }

}
```

Ebben a forráskódban két *Dispose* metódust készítettünk, az első paraméter nélküli az, amit az interfésztől kaptunk, míg a másik arra szolgál, hogy összehangoljuk a GC munkáját a kézi erőforrás-felszabadítással.
Ha a második metódus paramétere *true* értékű a híváskor, akkor tudjuk, hogy kézzel hívtuk a *Dispose* metódust, vagyis mind a menedzselt, mind a natív erőforrások felszabadíthatóak. Ha a paraméter értéke *false*, akkor pedig a hívás a destruktorból származik, vagyis csak az unmanaged erőforrásokkal kell törődnünk.

Az *IDisposable* interfészt megvalósító osztályok használhatóak ún. *using* blokk-ban, ami azt jelenti, hogy a blokk hatókörén kívülre érve a *Dispose* metódus automatikusan meghívódik:

```
using System;

class DisposableClass : IDisposable
{
      public void Dispose()
      {
            Console.WriteLine("Takarítunk...");

            GC.SuppressFinalize(this);
      }
}

class Program
{
      static public void Main()
      {
            using(DisposableClass dc = new DisposableClass())
            {
                  Console.WriteLine("Using blokk...");
            }
      }
}
```

A legtöbb I/O művelettel (file- és hálózatkezelés) kapcsolatos osztály megvalósítja az *IDisposable* –t, ezért ezeket ajánlott mindig *using* blokkban használni.

14 Metódusok

Az objektum-orientált programozásban egy metódus olyan programrész, amely vagy egy objektumhoz, vagy egy osztályhoz köthető. Előbbi az ún. osztály metódus, utóbbi pedig a statikus metódus. Ebben a fejezetben az osztály (*instance*) metódusokról lesz szó.

Egy metódussal megváltoztathatjuk egy objektum állapotát, vagy információt kaphatunk annak adatairól. Optimális esetben egy adattaghoz csakis metódusokon keresztül férhetünk hozzá (ez akkor is igaz, ha látszólag nem így történik, pl. minden operátor valójában metódus formájában létezik, ezt majd látni fogjuk).

Bizonyára szeretnénk, ha a korábban elkészített kutya osztályunk nem csak lógna a semmiben, hanem tenne is valamit. Készítsünk néhány metódust a legfontosabb műveletekhez: az evéshez és az alváshoz:

```
using System;

class Dog
{
	string name;
	int age;

	public Dog(string name, int age)
	{
		this.name = name;
		this.age = age;
	}

	public void Eat()
	{
		Console.WriteLine("A kutya eszik...");
	}

	public void Sleep()
	{
		Console.WriteLine("A kutya alszik...");
	}
}

class Program
{
	static public void Main()
	{
		Dog d = new Dog("Rex", 2);
		d.Eat();
		d.Sleep();
	}
}
```

Nézzük meg, hogyan épül fel egy metódus: elsőként megadjuk a láthatóságot és itt is érvényes a szabály, hogy ennek hiányában az alapértelmezett privát elérés lesz érvényben. Ezután a visszatérési érték típusa áll, jelen esetben a *void*–dal jeleztük, hogy nem várunk ilyesmit. Következik a metódus neve, ez konvenció szerint nagybetűvel kezdődik, végül a sort a paraméterlista zárja.

Egy metódust az objektum neve után írt pont operátorral hívhatunk meg (ugyanez érvényes a publikus adattagokra, tulajdonságokra, stb. is).

A „hagyományos” procedurális programozás (pl. a C vagy Pascal nyelv) a metódusokhoz hasonló, de filozófiájában más eszközöket használ, ezek a függvény (function) és az eljárás (procedure). Mi a különbség? Azt mondtuk, hogy a metódusok egy osztályhoz köthetőek, annak életciklusában játszanak szerepet.

Nézzünk egy példát:

```
using System;

class NewString1
{
	private string aString;

	public NewString1(string s)
	{
		this.aString = s;
	}

	public void PrintUpper()
	{
		Console.WriteLine(this.aString.ToUpper());
	}
}

class NewString2
{
	public void PrintUpper(string s)
	{
		Console.WriteLine(s.ToUpper());
	}
}

class Program
{
	static public void Main()
	{
		NewString1 ns1 = new NewString1("baba");
		NewString2 ns2 = new NewString2();

		ns1.PrintUpper();
		ns2.PrintUpper("baba");
	}
}
```

Pontosan ugyanaz történik mindkét esetben, de van egy nagy különbség. Az első osztály „valódi” osztály: adattaggal, konstruktorral, stb. Van állapota, végezhetünk rajta műveleteket. A második osztály nem igazi osztály, csak egy doboz, amelyben egy teljesen önálló, egyedül is életképes szerkezet van, mindössze azért kell az osztály-definíció, mert egyébként nem fordulna le a program (ebben az esetben egy statikus „metódust” kellett volna készítenünk, erről hamarosan).

Az első osztályban metódust definiáltunk, a másodikban eljárást (eljárás és függvény között a lényegi különbség, hogy utóbbinak van visszatérési értéke).

14.1 Paraméterek

Az objektummal való kommunikáció érdekében képesnek kell lennünk kívűlről megadni adatokat, vagyis paramétereket. A paraméterek számát és típusait a metódus deklarációjában, vesszővel elválasztva adjuk meg. Egy metódusnak gyakorlatilag bármennyi paramétere lehet.
A metódus nevét és paraméterlistáját aláírásnak, szignatúrának vagy prototípusnak nevezzük. Egy osztály bármennyi azonos nevű metódust tartalmazhat, ameddig a paraméterlistájuk különbözik. A paraméterek a metóduson belül lokális változókként viselkednek, és a paraméter nevével hivatkozunk rájuk.

```
using System;

class Test
{
	public void Method(string param)
	{
		Console.WriteLine("A paraméter: {0}", param);
	}
}

class Program
{
	static public void Main()
	{
		Test t = new Test();
		t.Method("Paraméter");
	}
}
```

A C# nyelvben paraméterek átadhatunk érték és cím szerint is. Előbbi esetben egy teljesen új példány jön létre az adott osztályból, amelynek értékei megegyeznek az eredetiével. A másik esetben egy az objektumra mutató referencia adódik át, tehát az eredeti objektummal dolgozunk.

Az érték- és referencia-típusok különbözően viselkednek az átadás szempontjából. Az értéktípusok alapértelmezetten érték szerint adódnak át, míg a referencia-típusoknál a cím szerinti átadás az előre meghatározott viselkedés. Utóbbi esetben van azonban egy kivétel, mégpedig az, hogy míg a referencia-típus értékeit megváltoztathatjuk (és ez az eredeti objektumra is hat) addig magát a referenciát már nem (tehát nem készíthetünk új példányt, amelyre az átadott referencia mutat). Ha ezt mégis megtesszük, az nem eredményez fordítási hibát, de a változás csakis a metóduson belül lesz észlelhető.

Erre a magyarázat nagyon egyszerű: már említettük, hogy egy metódusparaméter lokális változóként viselkedik, vagyis ebben az esetben egyszerűen egy lokális referenciával dolgoznánk.

```
using System;

class Test
{
      public int x = 10;

      public void TestMethod(Test t)
      {
            t = new Test();
            t.x = 11;
      }
}

class Program
{
      static public void Main()
      {
            Test t = new Test();
            Console.WriteLine(t.x); // 10;
            t.TestMethod(t);
            Console.WriteLine(t.x); // 10
      }
}
```

Ha mégis módosítani akarjuk egy referencia-típus referenciáját, akkor külön jeleznünk kell azt, hogy „valódi” referenciaként akarjuk átadni.

Kétféleképpen adhatunk át paramétert referencia szerint. Az első esetben az átadott objektumnak inicializáltnak kell lennie (tehát mindenképpen mutatnia kell valahová, használnunk kellett a *new* operátort). Ha ezt nem tettük meg, attól a program még lefordul, de a metódus hívásakor kivételt fogunk kapni (*NullReferenceException*).A referencia szerinti átadást a forráskódban is jelölni kell, mind a metódus prototípusánál, mind a hívás helyén a *ref* módosítóval.

Referencia-típust gyakorlatilag soha nem kell ilyen módon átadnunk (persze nincs megtiltva, de gondos tervezéssel elkerülhető), kivételt képez, ha ezt valamilyen .NET–en kívüli eszköz megköveteli (a lényeg, hogy már allokált objektumra mutató referenciát optimális esetben nem állítunk máshová).

A *ref* értéktípusok esetében már sokkal hasznosabb, nézzük a következő forráskódot:

```
using System;

class Test
{
	public void Swap(int x, int y)
	{
		int tmp = x;
		x = y;
		y = tmp;
	}
}

class Program
{
	static public void Main()
	{
		int x = 10;
		int y = 20;
		Test t = new Test();

		t.Swap(x, y);

		Console.WriteLine("x = {0}, y = {1}", x, y);
	}
}
```

A *Swap* eljárással megpróbáljuk felcserélni *x* és *y* értékeit. Azért csak próbáljuk, mert *int* típusok (mivel értéktípusról van szó) érték szerint adódnak át, vagyis a metódus belsejében teljesen új változókkal dolgozunk.

Írjuk át egy kicsit a forrást:

```
using System;

class Test
{
	public void Swap(ref int x, ref int y)
	{
		int tmp = x;
		x = y;
		y = tmp;
	}
}

class Program
{
	static public void Main()
	{
		int x = 10;
		int y = 20;
		Test t = new Test();

		t.Swap(ref x, ref y);

		Console.WriteLine("x = {0}, y = {1}", x, y);
	}
}
```

Most már az történik, amit szeretnénk: *x* és *y* értéke megcserélődött.

Egy érdekesebb módszer két szám megcserélésére: használjuk a kizáró vagy operátort, ami akkor ad vissza igaz értéket, ha a két operandusa közül pontosan az egyik igaz. Nézzük először a kódot:

```
public void Swap(ref int x, ref int y)
{
      if(x != y)
      {
            x ^= y;
            y ^= x;
            x ^= y;
      }
}
```

A két számot írjuk fel kettes számrendszerben: x (= 10) = 01010 és y (= 20) =10100. Most lássuk, hogy mi történik! Az első sor:

01010
10100 XOR

11110 (ez lesz most x)

A második sor:

10100
11110 XOR

01010 (ez most y, ez az érték a helyén van)

Végül a harmadik sor:

11110
01010 XOR

10100 (kész vagyunk)

Hogy ez a módszer miért működik, azt mindenki gondolja át maga, egy kis segítség azért jár: felhasználjuk a XOR következő tulajdonságait:

Kommutatív: A XOR B = B XOR A
Asszociatív: (A XOR B) XOR C = A XOR (B XOR C)
Létezik neutrális elem (jelöljük NE –vel): A XOR NE = A
Minden elem saját maga inverze: A XOR A = 0 (ez az állítás az oka annak, hogy ellenőriznünk kell, hogy x és y ne legyen egyenlő)

Bár ez az eljárás hatékonyabbnak tűnik, igazából ez egy hagyományos PC –n még lassúbb is lehet, mint az átmeneti változót használó társa. A XOR – Swap olyan limitált helyzetekben hasznos, ahol nincs elég memória/regiszter manőverezni – tipikusan mikrokontrollerek esetében.

A cím szerinti átadás másik formájában nem inicializált paramétert is átadhatunk, de ekkor feltétel, hogy a metóduson belül állítsuk be (átadhatunk így már inicializált

paramétert is, de ekkor is feltétel, hogy új objektumot készítsünk). A használata megegyezik a *ref*–fel, azaz a szignatúrában és a hívásnál is jelezni kell a szándékunkat. A használandó kulcsszó az *out* (Nomen est omen – A név kötelez):

```
using System;

class Init
{
	public void TestInit(out Test t)
	{
		t = new Test() { s = "Hello!"};
	}
}

class Test
{
	public string s = null;
}

class Program
{
	static public void Main()
	{
		Test t = null;
		Init i = new Init();

		i.TestInit(out t);
		Console.WriteLine(t.s); // Hello!
	}
}
```

A fenti programokban pontosan tudtuk, hogy hány paramétere van egy metódusnak. Előfordul viszont, hogy ezt nem tudjuk egyértelműen megmondani, ekkor ún. paraméter-tömböket kell használnunk. Ha ezt tesszük, akkor az adott metódus paraméter-listájában a paraméter-tömbnek kell az utolsó helyen állnia, illetve egy paraméter-listában csak egyszer használható ez a szerkezet.

```
using System;

class Test
{
	public void PrintElements(params object[] list)
	{
		foreach(var item in list)
		{
			Console.WriteLine(item);
		}
	}
}

class Program
{
	static public void Main()
	{
		Test t = new Test();
		t.PrintElements("alma", "körte", 4, 1, "dió");
		t.PrintElements(); // ez is működik
	}
}
```

A paraméter-tömböt a *params* kulcsszóval vezetjük be, ezután a metódus belsejében pontosan úgy viselkedik, mint egy normális tömb. Paraméter-tömbként átadhatunk megfelelő típusú tömböket is.

14.1.1 Alapértelmezett paraméterek

A C# 4.0 bevezeti az alapértelmezett paramétereket, amelyek lehetővé teszik, hogy egy paramétereknek alapértelmezett értékeket adjunk, ezáltal nem kell kötelezően megadnunk minden paramétert a metódus hívásakor. Nézzük a következő példát:

```
class Person
{
      public Person(string firstName, string lastName)
      {
            FirstName = firstName;
            LastName = lastName;
      }

      public string Person(string firstName, string lastName, string job)
            : this(firstName, lastName)
      {
            Job = job;
      }

      public string FirstName {get; private set; }
      public string LastName {get; private set; }
      public string Job {get; private set; }
}
```

Mivel nem tudunk biztosan minden emberhez munkahelyet rendelni, ezért két konstruktort kellett készítenünk. Ez alapvetően nem nagy probléma, viszont az gondot okozhat, ha valaki csak az első konstruktort használja, majd megpróbál hozzáférni a munka tulajdonsághoz. Nyílván ezt sem nagy gond megoldani, de miért fáradnánk, ha rendelkezésünkre állnak az alapértelmezett paraméterek? Írjuk át a forráskódot:

```
class Person
{
      public Person(string firstName, string lastName, string job = "N/A")
      {
            FirstName = firstName;
            LastName = lastName;
            Job = job;
      }

      public string FirstName { get; private set; }
      public string LastName { get; private set; }
      public string Job { get; private set; }
}
```

A „*job*” paraméterhez most alapértelmezett értéket rendeltünk, így biztosak lehetünk benne, hogy minden adattag megfelelően inicializált. Az osztályt most így tudjuk használni:

```
Personp1=newPerson("István","Reiter");
Personp2=newPerson("István","Reiter","börtönõr");
```

14.1.2 Nevesített paraméterek

A C# 4.0 az alapértelmezett paraméterek mellett bevezeti a nevesített paraméter (*named parameter*) fogalmát, amely segítségével explicit megadhatjuk, hogy melyik paraméternek adunk értéket. Nézzük az előző fejezet *Person* osztályának konstruktorát:

```
Person p = new Person(firstName:"István", lastName:"Reiter");
```

Mivel tudatjuk a fordítóval, hogy pontosan melyik paraméterre gondolunk, ezért nem kell betartanunk az eredeti metódus-deklarációban előírt sorrendet:

```
Person p = new Person(lastName:"Reiter", firstName:"István");
```

14.2 Visszatérési érték

Az objektumainkon nem csak műveleteket végzünk, de szeretnénk lekérdezni az állapotukat is és felhasználni ezeket az értékeket. Ezenkívül szeretnénk olyan függvényeket is készíteni, amelyek nem kapcsolódnak közvetlenül egy osztályhoz, de hasznosak lehetnek (pl. az *Int.Parse* függvény ilyen).

Készítsünk egy egyszerű függvényt, amely összead két számot, az eredményt pedig visszatérési értékként kapjuk meg:

```
using System;

class Test
{
      public int Add(int x, int y)
      {
            return x + y;
      }
}

class Program
{
      static public void Main()
      {
            Test t = new Test();
            int result = t.Add(10, 11);
      }
}
```

A végeredményt a *return* utasítással adhatjuk vissza. A metódus-deklarációnál meg kell adnunk a visszatérési érték típusát is. Amennyiben ezt megtettük, a metódusnak mindenképpen tartalmaznia kell egy *return* utasítást a megfelelő típusú elemmel (ez

lehet *null* is referencia- és nullable típusok esetében) és ennek az utasításnak mindenképpen le kell futnia:

```
public int Add(int x, int y)
{
      if(x != 0 && y != 0)
      {
            return x + y;
      }
}
```

Ez a metódus nem fordul le, mivel nem lesz minden körülmények között visszatérési érték. A visszatérített érték típusának vagy egyeznie kell a visszatérési érték típusával, vagy a kettő között léteznie kell implicit típuskonverziónak:

```
public int Add(int x, int y)
{
      return (byte)(x + y); // ez működik, bár nincs sok értelme
}

public int Add(int x, int y)
{
      return (long)(x + y);// ez le sem fordul
}
```

Visszatérési értékkel rendelkező metódust használhatunk minden olyan helyen, ahol a program valamilyen típust vár (értékadás, logikai kifejezések, metódus paraméterei, ciklusfeltétel, stb.).

Valójában a *Main* metódus két formában létezik: visszatérési értékkel és a nélkül. A *Main* esetében a visszatérési érték azt jelöli, hogy a program futása sikeres volt–e (1) vagy sem (0). Ezt a gyakorlatban akkor tudjuk használni, ha egy külső program/scipt hívta meg a programunkat és kíváncsiak vagyunk, hogy sikeres volt–e a futása.

```
static public int Main()
{
      return 0;
}
```

14.3 Kiterjesztett metódusok

A C# 3.0 lehetőséget ad arra, hogy egy már létező típushoz új metódusokat adjunk, anélkül, hogy azt közvetlenül módosítanánk, vagy származtatnánk belőle. Egy kiterjesztett metódus (*extension method*) minden esetben egy statikus osztály statikus metódusa kell, hogy legyen (erről a következő fejezetben).

Egészítsük ki a *string* típust egy metódussal, ami kiírja a képernyőre az adott karaktersorozatot:

```
using System;

static public class StringHelper
{
	static public void Print(this string s)
	{
		Console.WriteLine(s);
	}
}

class Program
{
	static public void Main()
	{
		string s = "ezegystring";
		s.Print();
		StringHelper.Print(s); // így is használhatjuk
	}
}
```

A *this* módosító után a paraméter típusa következik, amely meghatározza a kiterjesztett osztály típusát. A fenti példában látható, hogy „rendes” statikus metódusként is használható egy extension method.

Ha két kiterjesztett metódus ugyanazzal a szignatúrával rendelkezik, akkor a hagyományos, statikus úton kell hívnunk őket. Ha nem így teszünk, akkor a speciálisabb (szűkebb típusú) paraméterű metódus fog meghívódni.

Kiterjesztett metódust nem definiálhatunk beágyazott osztályban.

15 Tulajdonságok

A tulajdonságokat (*property*) a mezők közvetlen módosítására használjuk, anélkül, hogy megsértenénk az egységbezárás elvét. A tulajdonságok kívülről nézve pontosan ugyanolyanok, mint a hagyományos változók, de valójában ezek speciális metódusok. Minden tulajdonság rendelkezhet ún. *getter* és *setter* blokkal, előbbi a property mögött lévő mező értékét adja vissza, utóbbi pedig értéket ad neki:

```
using System;

class Person
{
	public Person(string name)
	{
		this.name = name;
	}

	string name;
	public string Name
	{
		get { return this.name; }
		set { this.name = value; }
	}
}

class Program
{
	static public void Main()
	{
		Person p = new Person("István");
		Console.WriteLine(p.Name);
	}
}
```

Láthatjuk, hogy egy property deklaráció hasonlóan épül fel mint a metódusoké, azzal a kivétellel, hogy nincs paraméterlista. Vegyük észre, hogy a setter–ben egy ismeretlen, *value* nevű változót használtunk. Ez egy speciális elem, azt az értéket tartalmazza amelyet hozzárendeltünk a setter-hez:

```
Person p = new Person("István");
p.Name = "Béla"; // value == "Béla"
```

A getter és setter elérhetőségének nem muszáj megegyeznie, de a getternek minden esetben publikusnak kell lennie:

```
public string Name
{
	private get { return this._name; } // ez nem működik
	set { this.name = value; }
}

public string Name
{
	get { return this.name; }
	private set { this.name = value; }//ez viszont jó
}
```

Arra is van lehetőség, hogy csak az egyiket használjuk, ekkor csak írható/olvasható tulajdonságokról beszélünk:

```
public string Name
{
      get { return this._name; }
}
```

Egyik esetben sem vagyunk rákényszerítve, hogy azonnal visszaadjuk/beolvassuk az adattag értékét, tetszés szerint végezhetünk műveleteket is rajtuk:

```
public string Name
{
      get { return "Mr. " + this._name; }
}
```

A C# 3.0 rendelkezik egy nagyon érdekes újítással, az ún. automatikus tulajdonságokkal. Nem kell létrehoznunk sem az adattagot, sem a teljes tulajdonságot, a fordító mindkettőt legenerálja nekünk:

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

A fordító automatikusan létrehoz egy *private* elérésű, *string* típusú „name” nevű adattagot és elkészíti hozzá a getter-t/setter-t is. Van azonban egy probléma, méghozzá az, hogy a fordítás pillanatában ez a változó még nem létezik, vagyis közvetlenül nem hivatkozhatunk rá pl. a konstruktorban. Ilyenkor a setter-en keresztül kell értéket adnunk.

```
class Person
{
      public Person(string name)
      {
            this.Name = name;
      }

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

16 Indexelők

Az indexelők hasonlóak a tulajdonságokhoz, azzal a különbséggel, hogy nem névvel, hanem egy indexxel férünk hozzá az adott információhoz. Általában olyan esetekben használják, amikor az osztály/struktúra tartalmaz egy tömböt vagy valamilyen gyűjteményt (vagy olyan objektumot, amely maga is megvalósít egy indexelőt).

Egy indexelőt így implementálhatunk:

```
using System;
using System.Collections;

class Names
{
	private ArrayList nameList;

	public Names()
	{
		nameList = new ArrayList();
		nameList.Add("István");
		nameList.Add("Judit");
		nameList.Add("Béla");
		nameList.Add("Eszter");
	}

	public int Count
	{
		get
		{
			return nameList.Count;
		}
	}

	public string this[int idx]
	{
		get
		{
			if(idx >= 0 && idx <n ameList.Count)
			{
				return nameList[idx].ToString();
			}

			return null;
		}
	}
}

class Program
{
	static public void Main()
	{
		Names n = new Names();

		for(int i = 0;i < n.Count;++i)
		{
			Console.WriteLine(n[i]);
		}
	}
}
```

Ez gyakorlatilag egy „névtelen tulajdonság”, a *this* mutat az aktuális objektumra, amin az indexelőt definiáltuk. Több indexet is megadhatunk, amelyek különböző típusú indexxel vagy visszatérési értékkel rendelkezhetnek.

17 Statikus tagok

A hagyományos adattagok és metódusok objektumszinten léteznek, azaz minden objektum minden adattagjából saját példánnyal rendelkezik. Gyakran van azonban szükségünk objektumtól független mezőkre/metódusokra, pl., ha meg szeretnénk számolni, hogy hány objektumot hoztunk létre. Erre a célra szolgálnak az ún. statikus tagok, amelyekből osztályszinten összesen egy darab létezik. Statikus tagokat akkor is használhatunk, ha az osztályból nem készült példány.

A statikus tagok jelentősége a C# nyelv tisztán objektum orientált mivoltában rejlik, ugyanis nem definiálhatunk globális (mindenki számára egyformán elérhető) tagokat. Ezt (is) váltják ki a statikus adattagok és metódusok.

17.1 Statikus adattag

Statikus tagot a *static* kulcsszó segítségével hozhatunk létre:

```
using System;

class Animal
{
      static public int AnimalCounter = 0;

      public Animal()
      {
            ++Animal.AnimalCounter;
      }

      ~Animal()
      {
            --Animal.AnimalCounter;
      }
}

class Program
{
      static public void Main()
      {
            Animal a = new Animal();
            Console.WriteLine(Animal.AnimalCounter);
      }
}
```

A példában a statikus adattag értékét minden alkalommal megnöveljük eggyel, amikor meghívjuk a konstruktort és csökkentük, amikor az objektum elpusztul, vagyis az aktív példányok számát tartjuk számon vele.
A statikus tagokhoz az osztály nevén (és nem egy példányán) keresztül férünk hozzá (a statikus tag „gazdaosztályából” az osztály neve nélkül is hivatkozhatunk rájuk, de ez nem ajánlott, mivel rontja az olvashatóságot).

A statikus tagok – ha az osztálynak nincs statikus konstruktora – rögtön a program elején inicializálódnak. Az olyan osztályok statikus tagjai, amelyek rendelkeznek

statikus konstruktorral, az inicializálást elhalasztják addig a pontig, amikor először használjuk az adott osztály egy példányát.

Konvenció szerint minden statikus tag (adattagok is) neve nagybetűvel kezdődik. A statikus és láthatósági módosító megadásának sorrendje mindegy.

17.2 Statikus konstruktor

A statikus konstruktor a statikus tagok beállításáért felel. A statikus konstruktor közvetlenül azelőtt fut le, hogy egy példány keletkezik az adott osztályból vagy hozzáfértek valamely tagjához.

A statikus konstruktornak nem lehet láthatóságot adni, illetve nincsenek paraméterei sem. Nem férhet hozzá példánytagokhoz sem.

```
using System;

class Test
{
      static public int Var = Test.Init();

      static public int Init()
      {
            Console.WriteLine("Var = 10");
            return 10;
      }

      static Test()
      {
            Console.WriteLine("Statikus konstruktor");
      }

      public Test()
      {
            Console.WriteLine("Konstruktor");
      }
}

class Program
{
      static public void Main()
      {
            Console.WriteLine("Start...");
            Test t = new Test();

      }
}
```

Ha elindítjuk a programot, a következő kimenetet kapjuk:

```
Start...
Var = 10
Statikus konstruktor
Konstruktor
```

A statikus konstruktoroknak van azonban egy hatalmas hátulütőjük, amelyet a következő példában láthatunk:

```
static class A1
{
	static public int x = 10;
}

static class A2
{
	static public int x;

	static A2()
	{
		x = 10;
	}
}
```

A két osztály látszólag ugyanazt teszi, mégis óriási teljesítménykülönbség van köztük:

```
class Program
{
	static public void Main()
	{
		Stopwatch sw = Stopwatch.StartNew();

		for(int i = 0;i < 10000000;++i)
		{
			int x = A1.x;
		}

		Console.WriteLine("Eltelt idõ: {0}ms", sw.ElapsedMilliseconds);

		sw = Stopwatch.StartNew();

		for(int i = 0;i < 10000000;++i)
		{
			int x = A2.x;
		}

		Console.WriteLine("Eltelt idõ: {0}ms", sw.ElapsedMilliseconds);
	}
}
```

A *Stopwatch* osztály a *System.Diagnostics* névtérben van és – ahogyan látszik is – időt tudunk mérni vele.

Mindkétszer tízmillió alkalommal kértük el a statikus tag értékét, lássuk az eredményt:

```
Eltelt idő: 12ms
Eltelt idő: 88ms
```

A különbség elképesztően nagy, az ok pedig a következő: ha definiáltunk statikus konstruktort akkor a rendszer minden egyes alkalommal, amikor statikus taghoz próbálunk hozzáférni, ellenőrzi, hogy meghívódott-e már a statikus konstruktor, ez pedig a fenti teljesítményveszteséget eredményezi.

17.3 Statikus metódus

Statikus metódust a hagyományos metódusokhoz hasonlóan készítünk, mindössze a *static* kulcsszóra van szükségünk. Ilyen metódus volt az előző példában az *Init* metódus is. A statikus konstruktortól eltérően rá nem vonatkozik, hogy nem lehetnek paraméterei. Statikus metódusok nem férnek hozzá az osztály „normális” tagjaihoz, legalábbis direkt módon nem (az minden további nélkül működik, ha egy példány referenciáját adjuk át neki).

Statikus metódust általában akkor használunk, ha nem egy példány állapotának a megváltoztatása a cél, hanem egy osztályhoz kapcsolódó művelet elvégzése. Ilyen metódus például az *Int32* osztályhoz tartozó *Parse* statikus metódus is.

A „leghíresebb” statikus metódus a *Main*.

17.4 Statikus tulajdonság

A statikus tulajdonságok a C# egy viszonylag ritkán használt lehetősége. Általában osztályokhoz kapcsolódó konstans értékek lekérdezésére használjuk (lényegében a statikus metódusok egy olvashatóbb verziója).

Példa:

```
class Math
{
      static public double PI
      {
            get { return 3.14; }
      }
}
```

17.5 Statikus osztály

Egy osztályt statikusnak jelölhetünk, ha csak és kizárólag statikus tagjai vannak. Egy statikus osztályból nem hozható létre példány, nem lehet példány-konstruktora (de statikus igen) és mindig lezárt (ld. Öröklődés). A fordító minden esetben ellenőrzi ezeknek a feltételeknek a teljesülését.

Példa:

```
using System;

static class MathHelper
{
      static public double PI
      {
            get { return 3.14; }
      }

      static public double Cos(double x)
```

```
        {
            return Math.Cos(x);
        }
}

class Program
{
        static public void Main()
        {
            Console.WriteLine(MathHelper.Cos(1));
        }
}
```

18 Struktúrák

A struktúrák – szerkezetüket tekintve – hasonlóak az osztályokhoz, viszont azoktól eltérően nem referencia-, hanem értéktípusok.
Minden struktúra indirekt módon a *System.ValueType* osztályból származik. Ez egy speciális típus, amely lehetőséget biztosít értéktípusok számára, hogy referencia-típusként viselkedjenek (lásd: Boxing).
A struktúrák közvetlenül tartalmazzák a saját értékeiket, míg az osztályok „csak” referenciákat tárolnak. Épp ezért struktúrát általában akkor használunk, ha egyszerű adatokkal kell dolgoznunk, de nincs szükségünk egy osztály minden szolgáltatására.

18.1 Konstruktor

Minden struktúra alapértelmezetten rendelkezik egy konstruktor-szerűséggel (vigyázat, *nem igazi* konstruktor), amely elvégzi a tagok nullára inicializálását (lényegében nullákkal tölti fel az adott memóriaterületet). Ez a lehetőség mindig él, nem rejthető el.

```
using System;

struct Test
{
	public int x;
}

class Program
{
	static public void Main()
	{
		Test t = new Test();
		Console.WriteLine(t.x); // x == 0
	}
}
```

Nem kötelező használni a *new* operátort, de ha így teszünk, akkor a struktúra tagjainak használata előtt definiálni kell az értéküket, ellenkező esetben a program nem fordul le:

```
using System;

struct Test
{
	public int x;
}

class Program
{
	static public void Main()
	{
		Test t;
		Console.WriteLine(t.x); // nem jó, x inicializálatlan
	}
}
```

Készíthetünk saját konstruktort, de ekkor minden mező értékadásáról gondoskodnunk kell. Egy struktúra mezőit nem inicializálhatjuk:

```
structTest
{
      int _x = 10; // ez nem jó
      int _y;

      public Test(int x, int y)
      {
            _y = y;// ez sem jó, x nem kap értéket
      }
}
```

Struktúrának csakis paraméteres konstruktort definiálhatunk, praméter nélküli alapértelmezettet nem. Viszont ha ezt megtettük, attól az alapértelmezett konstruktor még használható marad:

```
using System;

struct Test
{
      int _x;
      int _y;

      public Test(int x, int y)
      {
            _y = y;
            _x = x;
      }
}

class Program
{
      static public void Main()
      {
            Test t1 = new Test(10, 11);
            Test t2 = new Test(); //ez is működik
      }
}
```

18.2 Destruktor

Struktúrák nem rendelkezhetnek destruktorral. Egy struktúra két helyen lehet a memóriában: a *stack*-ben és a *heap*-ben (ha egy referencia-típus tagja).

Ahhoz, hogy megértsük, hogy miért nincs destruktor, szükségünk van a következőre: egy struktúrában lévő referenciatípusnak csak a referenciáját tároljuk. Ha a veremben van a struktúra, akkor előbb vagy utóbb kikerül onnan, és mivel így a benne lévő referenciatípusra már nem mutat referencia (legalábbis a struktúrából nem) ezért eltakarítható. Ugyanez a történet akkor is, ha a struktúra példány egy referenciatípusban foglal helyet.

18.3 Adattagok

A struktúrák az adattagokat közvetlenül tárolják (míg osztályok esetében mindig referenciákat tartunk számon). Egy struktúra minden adattagja – amennyiben a konstruktorban nem adunk meg mást – automatikusan a megfelelő nulla értékre inicializálódik.

Struktúra nem tartalmazhat saját magával megegyező típusú adattagot. Ugyanígy, egy struktúra nem tartalmazhat olyan típusú tagot, amely típus hivatkozik az eredeti struktúrára:

```
struct Test
{
	Test t;
}

struct Test1
{
	Test2 t;
}

struct Test2
{
	Test1 t;
}
```

Mindhárom struktúra hibás. Az ok nagyon egyszerű: mivel a struktúrák direkt módon – nem referenciákon keresztül – tárolják az adattagjaikat, valamint mivel a struktúrák nem vehetnek fel null értéket a fenti szerkezetek mind végtelen hurkot (és végtelen memóriafoglalást) okoznának (Test1 struktúra amiben Test2 amiben Test1 és így tovább) (lásd: tranzitív lezárt).

18.4 Hozzárendelés

Amikor egy struktúra példánynak egy másik példányt adunk értékül akkor egy teljesen új objektum keletkezik:

```
using System;

struct Test
{
	public int x;
}

class Program
{
	static public void Main()
	{
		Test t1 = new Test();
		Test t2 = t1;
		t2.x = 10;
		Console.WriteLine("t1.x = {0}, t2.x = {1}", t1.x, t2.x);
	}
}
```

Ha lefordítjuk ezt a kódot, azt fogjuk látni, hogy *t1.x* értéke nem változott, tehát nem referenciát adtunk át *t2*–nek.

Most nézzünk meg egy nagyon gyakori hibát, amibe belefuthatunk. Adott a következő programkód:

```
using System;

struct Point
{
	int _x;
	public int X
	{
		get { return _x; }
		set { _x = value; }
	}

	int _y;
	public int Y
	{
		get { return _y; }
		set { _y = value; }
	}

	public Point(int x, int y)
	{
		_x = x;
		_y = y;
	}
}

struct Line
{
	Point a;
	public Point A
	{
		get { return a; }
		set { a = value; }
	}

	Point b;
	public Point B
	{
		get { return b; }
		set { b = value; }
	}
}

class Program
{
	static public void Main()
	{
		Line l = new Line();
		l.A = new Point(10, 10);
		l.B = new Point(20, 20);
	}
}
```

Teljesen szabályos forrás, le is fordul. Látható, hogy a *Point* struktúra publikus tulajdonságokkal bír, vagyis jogosnak tűnik, hogy a *Line* struktúrán keresztül módosítani tudjuk a koordinátákat. Egészítsük ki a kódot:

```
static public void Main()
{
      Line l = new Line();
      l.A = new Point(10, 10);
      l.B = new Point(20, 20);
      l.A.X = 5;
}
```

Ez a forráskód nem fog lefordulni, mivel nem változónak akarunk értéket adni. Mi lehet a hiba oka? A probléma ott van, hogy rosszul értelmeztük ezt a kifejezést. Az *l.A* valójában a getter-t hívja meg, ami az eredeti struktúra egy másolatával tér vissza, amelynek a tagjait módosítani viszont nincs értelme. Ilyen esetekben mindig új struktúrát kell készítenünk:

```
static public void Main()
{
      Line l = new Line();
      l.A = new Point(10, 10);
      l.B = new Point(20, 20);
      l.A = new Point(5, 10);
}
```

Ez a hiba viszonylag gyakran fordul elő grafikus felületű alkalmazások készítése közben, mivel a .NET beépített *Point* típusa szintén struktúra.

18.5 Öröklődés

Struktúrák számára az öröklődés tiltott, minden struktúra automatikusan *sealed* módosítót kap. Ilyen módon egy struktúra nem lehet absztrakt, tagjainak elérhetősége nem lehet *protected/protected internal*, metódusai nem lehetnek virtuálisak illetve csak a *System.ValueType*) metódusait definiálhatja át. Ez utóbbi esetben a metódushívások nem járnak bedobozolással:

```
using System;

class Test
{
      public int x;

      public override string ToString()
      {
            return "X == " + x.ToString();
      }
}

class Program
{
      static public void Main()
      {
            Test t = new Test();
            t.x = 10;

            Console.WriteLine(t.ToString());
      }
}
```

19 Gyakorló feladatok III.

19.1 Faktoriális és hatvány

Készítsünk rekurzív faktoriális és hatványt számító függvényeket!

Megoldás (19/RecFact.cs és 19/RecPow.cs)

Mi is az a rekurzív függvény? Egy függvény, amely önmagát hívja. Rengeteg olyan probléma van, amelyeket több, lényegében azonos feladatot végrehajtó részre lehet osztani. Vegyük pl. a hatványozást: semmi mást nem teszünk, mint meghatározott számú szorzást végzünk, méghozzá ugyanazzal a számmal. Írhatunk persze egy egyszerű ciklust is, de ez egy kicsit atombombával egérre típusú megoldás lenne. Nézzük meg a hatványozás rekurzív megfelelőjét:

```
using System;

class Program
{
	static public double Pow(double x, int y)
	{
		if(y == 0){ return 1.0; }
		else return x * Pow(x, y - 1);
	}

	static public void Main()
	{
		double result = Pow(2, 10);
		Console.WriteLine(result); // 1024
	}
}
```

Látható, hogy *x*–et (az alapot) érintetlenül hagyjuk, míg a kitevőt (y) a függvény minden hívásakor eggyel csökkentjük, egészen addig, amíg értéke nulla nem lesz, ekkor befejezzük az „ördögi” kört és visszaadjuk az eredményt.

Hasonlóképpen készíthetjük el a faktoriálist számoló programot is:

```
using System;

class Program
{
	static public int Fact(int x)
	{
		if(x == 0){ return 1; }
		else return x * Fact(x - 1);
	}

	static public void Main()
	{
		int result = Fact(10);
		Console.WriteLine(result);
	}
}
```

19.2 Gyorsrendezés

Valósítsuk meg a gyorsrendezést!

Megoldás (19/QuickSort.cs)

A gyorsrendezés a leggyorsabb rendező algoritmus, nagy elemszám esetén O(n*logn) nagyságrendű átlaggal. Az algoritmus lényege, hogy a rendezendő elemek közül kiválaszt egy ún. pivot elemet, amely elé a nála nagyobb, mögé pedig a nála kisebb elemeket teszi, majd az így kapott két csoportra ismét meghívja a gyorsrendezést (tehát egy rekurzív algoritmusról beszélünk). Lássuk, hogy hogyan is néz ez ki a gyakorlatban!

Nagy elemszámnál ugyan jól teljesít ez az algoritmus, de kevés elem esetén fordul a kocka. Éppen ezért amikor a rendezésre átadott tömbrészlet kellően kicsi akkor egy általánosabb rendezést, pl. buborékrendezést érdemes használni.

A legtöbb programozási nyelv „beépített” rendezései általában a gyorsrendezés egy variációját használják.

```
class Array
{
      private int[] array;

      public Array(int length)
      {
            array = new int[length];
      }

      public int this[int idx]
      {
            get { return array[idx]; }
            set { array[idx] = value; }
      }

      public int Length
      {
            get { return array.Length; }
      }

      public void Sort()
      {
            QuickSort(0, array.Length - 1);
      }

      private void QuickSort(int left, int right)
      {
            // rendezés...
      }

}
```

Készítettünk egy osztályt, amely reprezentálja a rendezendő tömböt. A *Sort* metódussal fogjuk meghívni a tényleges rendező metódust amely két paramétert kap, a rendezésre kiválasztott tömbrészlet alsó és felső indexét. A metódus implementációja a következőképpen néz ki:

```
private void QuickSort(int left, int right)
{
      int pivot = array[left];
      int lhold = left;
      int rhold = right;

      while(left < right)
      {
            while(array[right] >= pivot && left < right)
            {
                  --right;
            }

            if(left != right)
            {
                  array[left] = array[right];
                  ++left;
            }

            while(array[left] <= pivot && left < right)
            {
                  ++left;
            }

            if(left != right)
            {
                  array[right] = array[left];
                  --right;
            }
      }

      array[left] = pivot;
      pivot = left;
      left = lhold;
      right = rhold;

      if(left < pivot)
      {
            QuickSort(left, pivot - 1);
      }

      if(right > pivot)
      {
            QuickSort(pivot + 1, right);
      }
}
```

A tömb mindkét oldaláról behatároljuk a kisebb/nagyobb elemeket majd továbbhívjuk a rendezést. Nézzük meg az algoritmus működését egy példán keresztül! A rendezendő számsorozat legyen:

3, 9, 4, 6, 8, 11

Left és *right* 0 és 5 (ugye hat elem van a tömbben, és nullától indexelünk), ezeket az értékeket eltároljuk, mivel az értékük módosulni fog, de a metódus végén szükség van az eredetiekre.
Az első ciklus – mivel nem fog a *left* indexen lévő számnál (3) kissebbet találni – úgy végződik, hogy *right* értéke 0 lesz. Az elágazásba – mivel *right* és *left* egyenlő – nem megyünk bele, ugyanúgy ahogyan a második ciklusba sem, mivel a hármasnál kisebb elem nincs a nála nagyobbak pedig utána helyezkednek el. A következő

elágazás szintén kimarad és nekiállhatunk kiszámolni, hogy miként hívjuk meg újra a metódust. A tömb változatlan marad, a *pivot* változó nulla értéket kap, *right* és *left* pedig visszakapják az eredeti értéküket.
Ezután a második elágazást fogjuk használni (ne feledjük, hogy *right* értéke ismét 5) és meghívjuk a *QuickSort* metódust 1 illetve 5 paraméterekkel, vagyis az első elemet (3) – mivel ő már a helyén van – átugorjuk.

Következik a második forduló, a *pivot* változó értéke most kilenc lesz, míg *left* és *right* értéke 1 és 5. Az első ciklus egyszer fog lefutni hiszen a tömb negyedik indexén ülő nyolcas szám már kissebb mint a *pivot* elem. *Left* nem egyenlő *right* –tal ezért a következő elágazásba is bemegyünk és a *left* indexre helyezzük a *right* indexen lévő nyolcast (a *pivot* elem pedig pont az itt „már nem" lévő kilencest tárolja). *Left* előrelép eggyel, hogy a tömb második indexére (4) mutasson.
A második ciklus is lefut, egészen addig fogjuk növelni *left* értékét, amíg eléri a *right* által mutatott nyolcast, hiszen ott a ciklus feltétel második fele sérül. Most *left* és *right* értéke egyenlő: 4. Éppen ezért a második elágazást kihagyjuk és tovább lépünk. A *left* által mutatott indexre behelyezzük a pivotban lévő kilencest, amivel helyre is áll a rend. *Pivot* értéke ezután négy lesz és a két másik változó is visszakapja az értékét. *Left* kissebb most, mint a *pivot* és *right* pedig nagyobb nála így mindkét elágazás feltétele teljesül, vagyis most mindkét oldalra hívjuk a metódust.
Az ez utáni események átgondolása pedig az olvasó feladata.

19.3 Láncolt lista

Valósítsuk meg a láncolt lista adatszerkezetet!

Megoldás (19/LinkedList.cs)

Amikor tömbökkel dolgozunk akkor a tömb elemeit indexekkel érjük el. A láncolt lista olyan adatszerkezet, amelynek elemei a soron következő elemre hivatkozó *referenciát* tartalmaznak. A láncolt listát az első fej- vagy gyökérelemen keresztül érjük el. Ha az elemek csak a következő tagra mutatnak, akkor egyszeresen-, ha a megelőző elemre is, akkor kétszeresen láncolt listáról beszélünk:

Elsőként valósítsuk meg az elemeket jelképező osztályt:

```
class Node
{
      public Node(int value)
      {
            this.Value = value;
      }

      public int Value{ get; set; }

      public Node Next{ get; set; }
      public NodePrevious { get; set; }
}
```

Most pedig jöjjön a láncolt lista osztály:

```
class LinkedList
{
      public LinkedList(){ }

      public LinkedList(int[] values)
      {
            foreach(int value in values)
            {
                  this.Add(value);
            }
      }

      public void Add(int value)
      {
            if(Root == null)
            {
                  Root = new Node(value);
            }
            else
            {
                  Node current = Root;
                  while(current.Next != null)
                  {
                        current = current.Next;
                  }

                  current.Next = new Node(value);
                  current.Next.Previous = current;
            }
      }

      public NodeRoot { get; private set; }
}
```

Az *Add* metódus az, amelyik számunkra érdekes. Elsőként megvizsgáljuk, hogy létezik-e gyökérelem, ha nem akkor létrehozzuk és nincs is más dolgunk (ugye ilyenkor még nincs se előző, se rákövetkező elem). Más a helyzet, ha van már néhány elem a listában, ekkor meg kell keresnünk a legutolsó elemet és utána fűzni az újat (megvalósíthattuk volna úgy is a listát, hogy tárolunk egy referenciát az utolsó elemre, ez lényegesen gyorsabb lenne – de kevésbé érdekes).

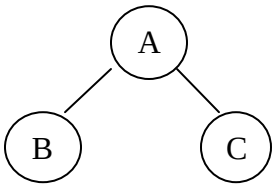
Ahhoz, hogy megkeressük az utolsó elemet szükségünk lesz egy átmeneti referenciára, amely mindig az aktuális elemet mutatja majd. A ciklust addig kell futtatni, ameddig az aktuális elem rákövetkezője null értékre nem mutat, ekkor beállítjuk a *Next* és *Previous* értékeket is.

19.4 Bináris keresőfa

Készítsünk bináris keresőfát!

Megoldás (19/BinaryTree.cs)

A fa típus olyan adatszerkezet, amelynek elemei nulla vagy több gyermekelemmel és maximum egy szülőelemmel rendelkeznek:



A képen az A elem gyermekei B illetve C akiknek természetesen A lesz a közös szülőelemük.
A fa típus egy speciális esete a bináris fa, amely minden elemének pontosan egy szülő és maximum kettő gyermek eleme lehet. A bináris fa speciális esete pedig a bináris keresőfa, amelynek jellemzője, hogy egy szülő elem bal oldali részfájában a szülőelemnél kisebb jobboldali részfájában pedig a szülőelemnél nagyobb elemek vannak, ezáltal egyértelműen meghatározható, hogy egy elem benne van-e a fában vagy nincs (értelemszerűen a részfák is keresőfák, vagyis rájuk is ugyanez vonatkozik). A bináris keresőfa minden elemének egyedi kulcssal kell rendelkeznie, vagyis ugyanaz az elem kétszer nem szerepelhet a fában. A keresés művelet O(logn) nagyságrendű.

Ahogyan az előző feladatban, most is készítsük el a fa csúcsait jelképező osztályt:

```
class TreeNode
{
      public TreeNode(int value)
      {
            this.Value = value;
      }

      public int Value { get; set; }

      public TreeNode Parent { get; set; }
      public TreeNode Left { get; set; }
      public TreeNode Right { get; set; }
}
```

Most pedig készítsük el a fa osztályt!

```
class BinaryTree
{
      public BinaryTree() { }

      public BinaryTree(int[] values)
      {
            foreach(int value in values)
            {
                  this.Insert(value);
            }
      }

      public void Insert(int value) { }

      public TreeNode Root { get; private set; }
}
```

Az osztály váza hasonlít a láncolt listáéhoz, itt is szükségünk van egy gyökérelemre, ez tulajdonképpen a legelső beszúrt csúcs lesz.

Írjuk meg a hiányzó Insert metódust:

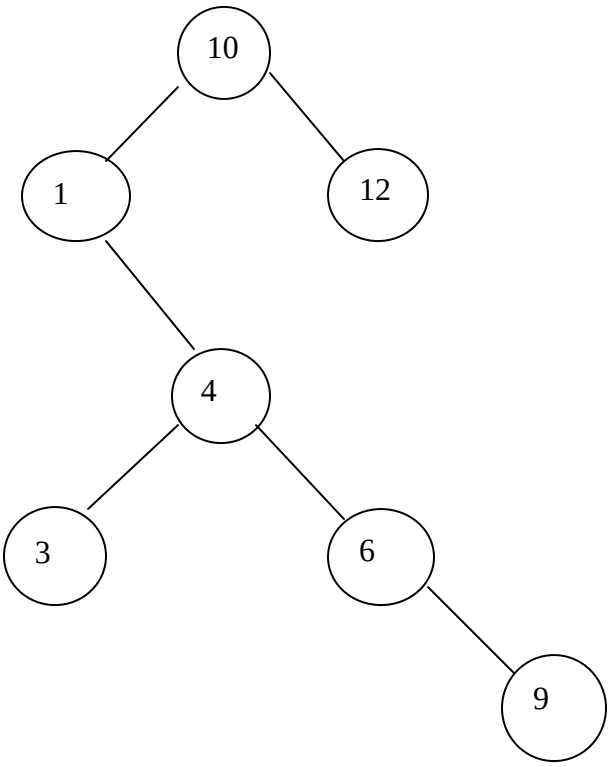
```
public void Insert(int value)
{
      if(Root == null)
      {
            Root = new TreeNode(value);
      }
      else
      {
            TreeNode current = Root;

            while(current != null)
            {
                  if(current.Value > value)
                  {
                        if(current.Left == null)
                        {
                              current.Left = new TreeNode(value);
                              current.Left.Parent = current;
                              return;
                        }
                        else
                        {
                              current = current.Left;
                        }
                  }
                  else if(current.Value < value)
                  {
                        if(current.Right == null)
                        {
                              current.Right = new TreeNode(value);
                              current.Right.Parent = current;
                              return;
                        }
                        else
                        {
                              current = current.Right;
                        }
                  }
                  else return;
            }
      }
}
```

Ha a gyökérelem null értéken áll, akkor készítünk egy új *TreeNode* objektumot, egyébként megkeressük az új elem helyét oly módon, hogy minden csúcsnál a megfelelő irányba „fordulunk”. Amennyiben az adott érték már szerepel a fában egyszerűen elhagyjuk a ciklust. Tegyük fel, hogy az elemek a következő sorrendben érkeznek:

10, 1, 4, 6, 6, 3, 9, 12

Ekkor a bináris keresőfa így fog kinézni:



A következő feladatunk, hogy kiírjuk a fa elemeit a konzolra. Persze ez nem is olyan egyszerű, hiszen megfelelő algoritmusra lesz szükségünk ahhoz, hogy a fa elemeit bejárhassuk. Egy rekurzív algoritmust fogunk használni, amely stratégiától függően az egyes csúcsok részfáit majd magát a csúcsot látogatja meg. Háromféle stratégiát ismerünk: preorder, inorder és postorder. A preorder elsőként a csúcsot majd a bal és jobb oldali részfát veszi kezelésbe. Inorder módon a bal oldali részfa, a csúcs és a jobb oldali részfa lesz a sorrend végül pedig a postorder bejárás sorrendje a bal oldali részfa, a jobb oldali részfa végül pedig a csúcs. Nézzük meg, hogyan is működik mindez a gyakorlatban! A fent felépített fán fogunk inorder módon végigmenni.

Az algoritmust a gyökérelemre (10) fogjuk meghívni, amely elsőként a bal oldali részfa csúcsát (1) fogja meglátogatni. Mivel neki nincsen bal oldali részfája, ezért kiírjuk az egyes számot és lépünk a jobb oldali részfára (4). Neki már van bal oldali ága ezért őt vizsgáljuk a továbbiakban. Itt nincs gyermekelem, ezért a következő szám, amit kiírhatunk a három. Visszalépünk a szülőelemre és kiírjuk őt (4) majd lépünk jobbra. A hatos csúcsnak sincs bal oldali fája, ezért kiírjuk, majd jön a jobb fában a kilences amit megint csak kiírunk hiszen nem rendelkezik gyermekelemmel. Ezen a ponton végeztünk a gyökérelem bal oldali fájával ezért őt jelenítjük meg, ezután pedig már csak egyetlen elem marad. A végső sorrend tehát:

1, 3, 4, 6, 9, 10, 12

A forráskódban ez az algoritmus meglehetősen egyszerűen jelenik meg, következzen az inorder bejárás:

```
public void InOrder(Action<int> action)
{
      _inOrder(Root, action);
}

private void _inOrder(TreeNode root, Action<int> action)
{
      if(root == null) { return; }

      _inOrder(root.Left, action);
      action(root.Value);
      _inOrder(root.Right, action);
}
```

Az *Action<T>* osztályról a Lambda kifejezések c. fejezetben olvashat többet az olvasó.

20 Öröklődés

Öröklődéssel egy már létező típust terjeszthetünk ki vagy bővíthetjük tetszőleges szolgáltatással. A C# csakis egyszeres öröklődést engedélyez (vagyis minden osztály egyetlen ősosztályból származhat, leszámítva a *System.Object* –et), ugyanakkor megengedi több interfész impementálását (interfészekről hamarosan).

Készítsük el az elméleti rész példáját (Állat-Kutya-Krokodil) C# nyelven. Az egyszerűség kedvéért hagyjuk ki az Állat és Kutya közti speciálisabb osztályokat:

```
class Animal
{

}

class Dog : Animal
{

}

class Crocodile : Animal
{

}
```

Az ősosztályt az osztálydeklaráció után írt kettőspont mögé kell tenni, szintén itt lesznek majd az osztály által megvalósított interfészek is.

A Kutya és Krokodil osztályok egyaránt megvalósítják az ősosztály (egyelőre szegényes) funkcionalitását. Bővítsük hát ki:

```
class Animal
{
      public Animal(string name)
      {
            this.Name = name;
      }

      public string Name { get; set; }

      public void Eat()
      {
            Console.WriteLine("Hamm - Hamm");
      }
}
```

Vegyük észre, hogy paraméteres konstruktort készítettünk az ősosztálynak, vagyis át kell gondolnunk a példányosítást. Az első változatot (az alapértelmezett konstruktorral) így használhattuk:

```
static public void Main()
{
      Dog d = new Dog();
      Crocodile c = new Crocodile();
}
```

Ha ezt az új *Animal* osztállyal próbálnánk meg akkor meglepetés fog érni, mivel nem fordul le a program. Ahhoz, hogy ki tudjuk javítani a hibát tudnunk kell, hogy a leszármazott osztályok először mindig a közvetlen ősosztály konstruktorát hívják meg, vagyis – ha nem adunk meg mást – az alapértelmezett konstruktort. A probléma az, hogy az ősosztálynak már nincs ilyenje, ezért a leszármazott osztályokban explicit módon hívni kell a megfelelő konstruktort:

```
class Animal
{
	public Animal(string name)
	{
		this.Name = name;
	}

	public string Name { get; set; }

	public void Eat()
	{
		Console.WriteLine("Hamm - Hamm");
	}
}

class Dog : Animal
{
	public Dog(string name) : base(name)
	{

	}
}

class Crocodile : Animal
{
	public Crocodile(string name) : base(name)
	{

	}
}
```

Ezután így példányosítunk:

```
static public void Main()
{
	Dog d = new Dog("Bundás");
	Crocodile c = new Crocodile("Aladár");

	Console.WriteLine("{0} és {1}", d.Name, c.Name);
}
```

Ugyanígy használhatjuk az ősosztály metódusát is:

```
static public void Main()
{
	Dog d = new Dog("Bundás");
	Crocodile c = new Crocodile("Aladár");

	d.Eat();
	c.Eat();
}
```

Honnan tudja vajon a fordító, hogy egy ősosztálybeli metódust kell meghívnia? A referenciatípusok speciális módon jelennek meg a memóriában, rendelkeznek többek közt egy ún. metódus-táblával, ami mutatja, hogy az egyes metódushívásoknál melyik metódust kell meghívni. Persze ezt is meg kell határozni valahogy, ez nagy vonalakban úgy történik, hogy a fordító a fordítás pillanatában megkapja a metódus nevét és elindul „visszafelé" az osztályhierarchia mentén. A fenti példában a hívó osztály nem rendelkezik *Eat* nevű metódussal és nem is definiálja át annak a viselkedését (erről hamarosan), ezért az eggyel feljebbi őst kell megnéznünk. Ez egészen a lehető „legújabb" metódusdefinícióig megy, és amikor megtalálja a megfelelő implementációt, bejegyzi azt a metódustáblába.

20.1 Virtuális metódusok

Az ősosztályban deklarált virtuális (vagy polimorfikus) metódusok viselkedését a leszármazottak átdefiniálhatják. Virtuális metódust a szignatúra elé írt *virtual* kulcsszó segítségével deklarálhatunk:

```
using System;

class Animal
{
      public virtual void Eat()
      {
            Console.WriteLine("Hamm - Hamm");
      }
}

class Dog : Animal
{
      public override void Eat()
      {
            Console.WriteLine("Vau - Vau - Hamm - Hamm");
      }
}

class Crocodile : Animal
{
      public override void Eat()
      {
            Console.WriteLine("Kro - Kro - Hamm - Hamm");
      }
}

class Program
{
      static public void Main()
      {
            Animal a = new Animal();
            Dog d = new Dog();
            Crocodile c = new Crocodile();

            a.Eat();
            d.Eat();
            c.Eat();
      }
}
```

A leszármazott osztályokban az *override* kulcsszóval mondjuk meg a fordítónak, hogy szándékosan hoztunk létre az ősosztályéval azonos szignatúrájú metódust és a leszármazott osztályon ezt kívánjuk használni mostantól. Egy *override*–dal jelölt metódus automatikusan virtuális is lesz, így az ő leszármazottai is átdefiniálhatják a működését:

```
class Crocodile : Animal
{
	public override void Eat()
	{
		Console.WriteLine("Kro - Kro - Hamm - Hamm");
	}
}

class BigEvilCrocodile : Crocodile
{
	public override void Eat()
	{
		Console.WriteLine("KRO - KRO - HAMM - HAMM");
	}
}
```

Az utódosztály metódusának szignatúrája és láthatósága meg kell egyezzen azzal amit át akarunk definiálni.

Tegyük fel, hogy nem ismerjük az ősosztály felületét és a hagyományos módon deklaráljuk az *Eat* metódust (ugye nem tudjuk, hogy már létezik). Ekkor a program ugyan lefordul, de a fordító figyelmezet minket, hogy eltakarjuk az öröklött metódust. És valóban, ha meghívnánk, akkor az új metódus futna le. Ezt a jelenséget árnyékolásnak (shadow) nevezik.
Természetesen mi azt szeretnénk, hogy a fordítás hiba nélkül menne végbe, így tájékoztatnunk kell a fordítót, hogy szándékosan takarjuk el az eredeti implementációt. Ezt a *new* kulcsszóval tehetjük meg:

```
class Animal
{
	public virtual void Eat()
	{
		Console.WriteLine("Hamm - Hamm");
	}
}

class Dog : Animal
{
	public new void Eat()
	{
		Console.WriteLine("Vau - Vau - Hamm - Hamm");
	}
}
```

Ezután a *Dog* utódjai már nem látják az eredeti *Eat* metódust. Viszont készíthetünk belőle virtuális metódust, amelyet az utódai már kedvükre használhatnak. Azaz, a *new* módosítóval ellátott metódus új sort kezd, amikor a fordító felépíti a metódustáblát, vagyis a *new virtual* kulcsszavakkal ellátott metódus lesz az új metódussorozat gyökere.

```
class Dog : Animal
{
	public new virtual void Eat()
	{
		Console.WriteLine("Vau - Vau - Hamm - Hamm");
	}
}
```

Nem jelölhetünk virtuálisnak statikus, absztrakt és *override*–dal jelölt tagokat (az utolsó kettő egyébként virtuális is lesz, de ezt nem kell külön jelölni).

20.2 Polimorfizmus

Korábban már beszéltünk arról, hogy az ős és leszármazottak közt *az-egy* (is-a) reláció áll fent. Ez a gyakorlatban azt jelenti, hogy minden olyan helyen, ahol egy őstípust használunk, ott használhatunk leszármazottat is (pl. egy állatkertben állatok vannak, de az állatok helyére (nyílván) behelyettesíthetek egy speciálisabb fajt).
Például gond nélkül írhatom a következőt:

```
Animal d = new Dog("Bundás");
```

A *new* operátor meghívása után *d* úgy fog viselkedni, mint a *Dog* osztály egy példánya (elvégre az is lesz), használhatja annak metódusait, adattagjait. Arra azonban figyeljünk, hogy ez visszafelé nem működik, a fordító hibát jelezne.

Abban az esetben ugyanis, ha a fordító engedné a visszafelé konverziót az ún. leszeletelődés (slicing) effektus lépne fel, azaz az adott objektum elveszítené a speciálisabb osztályra jellemző karakterisztikáját. A C++ nyelvben sokszor jelent gondot ez a probléma, mivel ott egy pointeren keresztül megtehető a „lebutítás”. Szerencsére a C# nyelvben ezt megoldották, így nem kell aggódnunk miatta.

Mi történik vajon a következő esetben:

```
static public void Main()
{
	Animal[] animalArray = new Animal[2];

	animalArray[0] = new Animal();
	animalArray[1] = new Dog();

	animalArray[0].Eat();
	animalArray[1].Eat();
}
```

Amit a fordító lát az az, hogy készítettünk egy *Animal* típusú elemekből álló tömböt és, hogy az elemein meghívtuk az *Eat* metódust. Csakhogy az Eat egy virtuális metódus, ráadásul van leszármazottbeli implementációja is, amely átdefinálja az eredeti viselkedést, és ezt explicit jelöltük is az *override* kulcsszóval. Így a fordító el tudja dönteni a futásidejű típust, és ez által ütemezi a metódushívásokat. Ez az ún. késői kötés (late binding). A kimenet így már nem lehet kétséges.
Már beszéltünk arról, hogyan épül fel a metódustábla, a fordító megkeresi a legkorábbi implementációt, és most már azt is tudjuk, hogy az első ilyen

implementáció egy virtuális metódus lesz, azaz a keresés legkésőbb az első virtuális változatnál megáll.

20.3 Lezárt osztályok és metódusok

Egy osztályt lezárhatunk, azaz megtilthatjuk, hogy új osztályt származtassunk belőle:

```
sealed class Dobermann : Dog
{

}

class MyDobermann : Dobermann // ez nem jó
{

}
```

Egy metódust is deklarálhatunk lezártként, ekkor a leszármazottak már nem definiálhatják át a működését:

```
class Dog : Animal
{
	public sealed override void Eat()
	{
		Console.WriteLine("Vau - Vau - Hamm - Hamm");
	}
}

sealed class Dobermann : Dog
{
	public override void Eat() // ez sem jó
	{

	}
}
```

20.4 Absztrakt osztályok

Egy absztrakt osztályt nem lehet példányosítani. A létrehozásának célja az, hogy közös felületet biztosítsunk a leszármazottainak:

```
using System;

abstract class Animal
{
	abstract public void Eat();
}

class Dog : Animal
{
	public override void Eat()
	{
		Console.WriteLine("Vau - Vau - Hamm - Hamm");
	}
}
```

```
class Program
{
	static public void Main()
	{
		// Animal a = new Animal(); //ez nem fordul le

		Dog d = new Dog();
		d.Eat();
	}
}
```

Látható, hogy mind az osztály, mind a metódus absztraktként lett deklarálva, ugyanakkor a metódus (látszólag) nem virtuális és nincs definíciója.
Egy absztrakt osztály csak a fordítás közben absztrakt, a lefordított kódban teljesen normális osztályként szerepel, virtuális metódusokkal. A fordító feladata az, hogy betartassa a rá vonatkozó szabályokat. Ezek a szabályok a következők:

- absztrakt osztályt nem lehet példányosítani
- absztrakt metódusnak nem lehet definíciója
- a leszármazottaknak definiálnia *kell* az öröklött absztrakt metódusokat.

Absztrakt osztály tartalmazhat nem absztrakt metódusokat is, ezek pont úgy viselkednek, mint a hagyományos nem-virtuális társaik. Az öröklött absztrakt metódusokat az *override* kulcsszó segítségével tudjuk definiálni (hiszen virtuálisak, még ha nem is látszik).
Amennyiben egy osztálynak van legalább egy absztrakt metódusa az osztályt is absztraktként kell jelölni.
Annak ellenére, hogy egy absztrakt osztályt nem példányosíthatunk még lehet konstruktora, mégpedig azért, hogy beállíthassuk vele az adattagokat:

```
abstract class Animal
{
	public Animal(string name)
	{
		this.Name = name;
	}

	public string Name { get; set; }

	abstract public void Eat();
}

class Dog : Animal
{
	public Dog(string name) : base(name) { }

	public override void Eat()
	{
		Console.WriteLine("Vau - Vau - Hamm - Hamm");
	}
}
```

Vajon, hogyan működik a következő példában a polimorfizmus elve? :

```
Animal[] animalArray = new Animal[2];

animalArray[0] = new Dog("Bundás");
animalArray[1] = new Crocodile("Aladár");
```

Ennek a kódnak hiba nélkül kell fordulnia, hiszen ténylegesen egyszer sem példányosítottuk az absztrakt ősosztályt. A fordító csak azt fogja megvizsgálni, hogy mi van a *new* operátor jobb oldalán, az alaposztály nem érdekli. Természetesen a következő esetben nem fordulna le:

```
animalArray[0] = new Animal("Animal");
```

21 Interfészek

Az interfészek hasonlóak az absztrakt osztályokhoz, abban az értelemben, hogy meghatározzák egy osztály viselkedését, felületét. A nagy különbség a kettő közt az, hogy míg előbbi eleve meghatároz egy osztályhierarchiát, egy interfész nem köthető közvetlenül egy osztályhoz, mindössze előír egy mintát, amit meg kell valósítani. Egy másik előnye az interfészek használatának, hogy míg egy osztálynak csak egy őse lehet, addig bármennyi interfészt megvalósíthat. Ezen felül interfészt használhatunk struktúrák esetében is. A következő példában átírjuk az *Animal* ősosztályt interfészre:

```
using System;

interface IAnimal
{
      void Eat();
}

class Dog : IAnimal
{
      public void Eat()
      {
            Console.WriteLine("Vau - Vau - Hamm - Hamm");
      }
}

class Program
{
      static public void Main()
      {
            Dog d = new Dog();
            d.Eat();
      }
}
```

Az interfész nevét konvenció szerint nagy I betűvel kezdjük. Látható, hogy a metódusokhoz nem tartozik definíció, csak deklaráció. A megvalósító osztály dolga lesz majd implementálni a tagjait. Egy interfész a következőket tartalmazhatja: metódusok, tulajdonságok, indexelők és események (erről hamarosan). A tagoknak nincs külön megadott láthatóságuk, mindannyiuk elérhetősége publikus. Magának az interfésznek az elérhetősége alapesetben publikus, illetve jelölhetjük *internal*–ként, másféle láthatóságot nem adhatunk meg (illetve osztályon belül deklarált (beágyazott) interfész elérhetősége lehet privát).
Egy interfészt implementáló osztálynak meg kell valósítania az interfész metódusait, egyetlen kivétellel, ha a szóban forgó osztály egy absztrakt osztály, ekkor az interfész metódusait abszraktként jelölve elhalaszthatjuk a metódusdefiníciót az absztrakt osztály leszármazottainak implementálásáig:

```
interface IAnimal
{
      void Eat();
}

abstract class AbstractAnimal : IAnimal
{
      public abstract void Eat();
}
```

Fontos, hogy amennyiben egy osztályból is származtatunk, akkor a felsorolásnál az ősosztály nevét kell előrevenni, utána jönnek az interfészek:

```
class Base { }

interface IFace { }

class Derived : IFace, Base { } // ez nem fordul le
```

Egy interfészt származtathatunk más interfészekből:

```
using System;

interface IAnimal
{
      void Eat();
}

interface IDog : IAnimal
{
      void Vau();
}

class Dog : IDog
{
      public void Eat()
      {
            Console.WriteLine("Vau - Vau - Hamm - Hamm");
      }

      public void Vau()
      {
            Console.WriteLine("Vau - Vau");
      }
}

class Program
{
      static public void Main()
      {
            Dog d = new Dog();
            d.Eat();
            d.Vau();
      }
}
```

Ekkor természetesen az összes interfészt meg kell valósítanunk.

Egy adott interfészt megvalósító objektumot implicit módon átkonvertálhatjuk az interfész „típusára”:

```
static public void Main()
{
      Dog d = new Dog();
      IAnimal ia = d;
      IDog id = d;

      ia.Eat();
      id.Vau();
}
```

Az *is* és *as* operátorokkal pedig azt is megtudhatjuk, hogy egy adott osztály megvalósít–e egy interfészt:

```
static public void Main()
{
      Dog d = new Dog();

      IAnimal ia = d as IAnimal;
      if(ia != null)
      {
            Console.WriteLine("Az objektum megvalósítja az IAnimal -t");
      }

      if(d is IDog)
      {
            Console.WriteLine("Az objektum megvalósítja az IDog -ot");
      }
}
```

21.1 Explicit interfészimplementáció

Ha több interfészt implementálunk, az névütközéshez is vezethet. Ennek kiküszöbölésére explicit módon megadhatjuk a megvalósítani kívánt funkciót:

```
using System;

interface IOne
{
      void Method();
}

interface ITwo
{
      void Method();
}

class Test : IOne, ITwo
{
      public void Method()
      {
            Console.WriteLine("Method!");
      }
}

class Program
{
      static public void Main()
      {
            Test t = new Test();
            t.Method();
      }
}
```

Ez a forráskód lefordul, és a metódust is meg tudjuk hívni, a probléma ott van, hogy két metódust kellene implementálnunk, de csak egy van – viszont a program működik. Nyílván nem ez az elvárt viselkedés, ezért ilyen esetekben explicit módon meg kell mondanunk, hogy melyik metódus/tulajdonság/etc... melyik interfészhez tartozik. Írjuk át a fenti kódot:

```
class Test : IOne, ITwo
{
	void IOne.Method()
	{
		Console.WriteLine("IOne Method!");
	}

	void ITwo.Method()
	{
		Console.WriteLine("ITwo Method!");
	}
}
```

Vegyük észre, hogy nem használtunk láthatósági módosítót, ilyenkor az interfész láthatósága érvényes ezekre a tagokra.
Újabb problémánk van, méghozzá az, hogy hogyan fogjuk meghívni a metódusokat? Most fogjuk kihasználni, hogy egy osztály konvertálható a megvalósított interfészek típusára:

```
static public void Main()
{
	Test t = new Test();

	((IOne)t).Method(); // ez mûködik

	ITwo it = t;
	it.Method(); // ez is mûködik
}
```

21.2 Virtuális tagok

Egy interfész tagjai alapértelmezés szerint lezártak, de a megvalósításnál jelölhetjük őket virtuálisnak. Ezután az osztály leszármazottjai tetszés szerint módosíthatják a definíciót, a már ismert *override* kulcsszóval:

```
class Dog : IDog, IAnimal
{
	public void Eat()
	{
		Console.WriteLine("Vau - Vau - Hamm - Hamm");
	}

	public virtual void Vau()
	{
		Console.WriteLine("Vau - Vau");
	}
}

class WuffDog : Dog
{
	public override void Vau()
	{
		Console.WriteLine("Wuff - Wuff - Vau - Vau");
	}
}
```

Egy leszármazott újraimplementálhatja az adott interfészt, amennyiben nemcsak az ősnél, de az utódnál is jelöljük a megvalósítást:

```
class WuffDog : Dog, IAnimal
{
      public new void Eat()
      {
            Console.WriteLine("Wuff - Wuff - Hamm - Hamm");
      }

      public override void Vau()
      {
            Console.WriteLine("Wuff - Wuff - Vau - Vau");
      }
}
```

Ez esetben használnunk kell a *new* kulcsszót annak jelölésére, hogy eltakarjuk az ős megvalósítását.

22 Operátor kiterjesztés

Nyílván szeretnénk, hogy az általunk készített típusok hasonló funkcionalitással rendelkezzenek, mint a beépített típusok (*int*, *string*, stb…).
Vegyük pl. azt a példát, amikor egy mátrix típust valósítunk meg. Jó lenne, ha az összeadás, kivonás, szorzás, stb. műveleteket úgy tudnánk végrehajtani, mint egy egész szám esetében, nem pedig metódushívásokkal. Szerencsére a C# ezt is lehetővé teszi számunkra, ugyanis engedi az operátorok kiterjesztését (operator overloading), vagyis egy adott operátort tetszés szerinti funkcióval ruházhatunk fel az osztályunkra vonatkoztatva.

```
static public void Main()
{
      Matrix m1 = new Matrix(20, 20);
      Matrix m2 = new Matrix(20, 20);

      //ez is jó
      m1.Add(m2);

      //de ez még jobb lenne
      m1 += m2;
}
```

A kiterjeszthető operátorok listája:

+(unáris)	-(unáris)	!	~	++
--	+	-	*	/
%	&	|	^	<<
>>	==	!=	>	<
>=	<=			

A C# nyelvben az operátorok valójában statikus metódusok, paramétereik az operandusok, visszatérési értékük pedig az eredmény. Egy egyszerű példa:

```
class MyInt
{
      public MyInt(int value)
      {
            this.Value = value;
      }

      public int Value { get; private set; }

      static public MyInt operator+(MyInt lhs, MyInt rhs)
      {
            return new MyInt(lhs.Value + rhs.Value);
      }
}
```

A ’+’ operátort működését fogalmaztuk át. A paraméterek (operandusok) nevei konvenció szerint *lhs* (left-hand-side) és *rhs* (right-hand-side), utalva a jobb és baloldali operandusra.
Tehát most már nyugodtan írhatom a következőt:

```
static public void Main()
{
      MyInt x = new MyInt(10);
      MyInt y = new MyInt(20);

      MyInt result = x + y;

      Console.WriteLine(result.Value);
}
```

Mivel definiáltunk az osztályunkon egy saját operátort így a fordító tudni fogja, hogy azt használja és átalakítja a műveletet:

```
MyInt result = MyInt.operator+(x, y);
```

22.1 Egyenlőség operátorok

A .NET megfogalmaz néhány szabályt az operátor-kiterjesztéssel kapcsolatban. Ezek egyike az, hogy ha túlterheljük az egyenlőség operátort (==) akkor definiálnunk kell a nem-egyenlő (!=) operátort is:

```
class MyInt
{
      public MyInt(int value)
      {
            this.Value = value;
      }

      public int Value { get; private set; }

      static public MyInt operator+(MyInt lhs, MyInt rhs)
      {
            return new MyInt(lhs.Value + rhs.Value);
      }

      static public bool operator==(MyInt lhs, MyInt rhs)
      {
            return lhs.Value == rhs.Value;
      }

      static public bool operator!=(MyInt lhs, MyInt rhs)
      {
            return !(lhs == rhs);
      }
}
```

A nem-egyenlő operátor esetében a saját egyenlőség operátort használtuk fel (a megvalósítás elve nem feltétlenül világos, elsőként megvizsgáljuk, hogy a két elem egyenlő–e, de mi a nem-egyenlőségre vagyunk kíváncsiak, ezért tagadjuk az eredményt, ami pontosan ezt a választ adja meg).
Ezekhez az operátorokhoz tartozik az *object* típustól örökölt virtuális *Equals* metódus is, ami a CLS kompatibilitást hívatott megőrizni, erről később még lesz szó. A fenti esetben ezt a metódust is illik megvalósítani. Az *Equals* azonban egy kicsit különbözik, ő egyetlen *object* típusú paramétert vár, ezért meg kell majd győződnünk arról, hogy valóban a saját objektumunkkal van –e dolgunk:

```
public override bool Equals(object rhs)
{
      if(!(rhs is MyInt))
      {
            return false;
      }

      return this == (MyInt)rhs;
}
```

Mivel ez egy példány tag ezért a *this*–t használjuk az objektum jelölésére, amin meghívtuk a metódust.

Ha az *Equals* –t megvalósítottuk, akkor ezt kell tennünk a szintén az object –től öröklött *GetHashCode* metódussal is, így az osztály használható lesz gyűjteményekkel és a *HashTable* típussal is. A legegyszerűbb implementáció visszaad egy számot az adattag(ok)ból számolva (pl.: hatványozás, biteltolás, stb…):

```
public override int GetHashCode()
{
      return this.Value << 2;
}
```

22.2 A ++/-- operátorok

Ez a két operátor elég nagy fejfájást tud okozni, nézzük meg a következő kódot:

```
using System;

class MyInt
{
      public MyInt(int value)
      {
            this.Value = value;
      }

      public int Value { get; private set; }

      static public MyInt operator++(MyInt rhs)
      {
            ++rhs.Value;
            return rhs;
      }
}

class Program
{
      static public void Main()
      {
            MyInt x = new MyInt(10);
            Console.WriteLine(x.Value); // 10
            ++x;
            Console.WriteLine(x.Value); // 11
            MyInt y = x++;
            Console.WriteLine(x.Value); // 12
            Console.WriteLine(y.Value); // 12 (!)
      }
}
```

Nézzük az utolsó sort! Mivel *y*–nak a postfixes formában adtunk értéket, ezért 11–et kellene tartalmaznia, ehelyett x++ esetében pontosan ugyanaz történik, mint ha ++x-et írtunk volna. A probléma gyökere, hogy postfixes operátort egyszerűen nem készíthetünk (ezt egyébként maga a Microsoft sem ajánlja) (illetve készíthetünk, de nem fog úgy működni, ahogyan szeretnénk).

22.3 Relációs operátorok

Hasonlóan a logikai operátorokhoz a relációs operátorokat is csak párban lehet elkészíteni, vagyis (<, >) és (<=, >=):

```
static public bool operator<(MyInt lhs, MyInt rhs)
{
      return lhs.Value < rhs.Value;
}

static public bool operator>(MyInt lhs, MyInt rhs)
{
      return lhs.Value > rhs.Value;
}
```

Ebben az esetben az *IComparable* és *IComparable<T>* interfészek megvalósítása is szükséges lehet a különböző gyűjteményekkel való együttműködés érdekében. Ezekkel hamarosan többet is foglalkozunk.

22.4 Konverziós operátorok

A C# a szűkebbről tágabbra konverziókat implicit módon (azaz különösebb jelölés nélkül), míg a tágabbról szűkebbre konvertálást explicite (ezt jelölnünk kell) végzi. Természetesen szeretnénk, ha a saját típusunk ilyesmire is képes legyen, és bizony erre is létezik operátor. Ezeknél az operátoroknál az *implicit* illetve *explicit* kulcsszavakkal fogjuk jelölni a konverzió típusát:

```
static public implicit operator MyInt(int rhs)
{
      return new MyInt(rhs);
}

static public explicit operator MyInt(string rhs)
{
      return new MyInt(int.Parse(rhs));
}
```

Ezeket most így használhatjuk:

```
static public void Main()
{
      MyInt x = 10; // implicit konverzió
      MyInt y = (MyInt)"20"; //explicit konverzió

      Console.WriteLine("x == {0}, y == {1}", x.Value, y.Value);
}
```

Fontos, hogy a konverziós operátorok mindig statikusak.

22.5 Kompatibilitás más nyelvekkel

Mivel nem minden nyelv teszi lehetővé az operátorok kiterjesztését ezért a CLS javasolja, hogy készítsük el a hagyományos változatot is:

```
static public MyInt Add(MyInt lhs, MyInt rhs)
{
      return new MyInt(lhs + rhs);
}

static public MyInt operator+(MyInt lhs, MyInt rhs)
{
      return new MyInt(lhs.Value + rhs.Value);
}
```

Ez természetesen nem kötelező, de bizonyos helyzetekben jól jön.

23 Kivételkezelés

Nyílván vannak olyan esetek, amikor az alkalmazásunk, bár gond nélkül lefordul, mégsem úgy fog működni, ahogy elképzeltük. Az ilyen „abnormális” működés kezelésére találták ki a kivételkezelést. Amikor az alkalmazásunk „rossz” állapotba kerül, akkor egy ún. kivételt fog dobni, ilyennel már találkoztunk a tömböknél, amikor túlindexeltünk:

```
using System;

class Program
{
	static public void Main()
	{
		int[] array = new int[2];
		array[2] = 10;
	}
}
```

Itt az utolsó érvényes index az 1 lenne, így kivételt kapunk, mégpedig egy *System.IndexOutOfRangeException*–t. Ezután a program leáll. Természetesen mi azt szeretnénk, hogy valahogy kijavíthassuk ezt a hibát, ezért el fogjuk kapni a kivételt. Ehhez a művelethez három dologra van szükségünk: kijelölni azt a programrészt, ami dobhat kivételt, elkapni azt és végül kezeljük a hibát:

```
using System;

class Program
{
	static public void Main()
	{
		int[] array = new int[2];

		try
		{
			array[2] = 10;
		}
		catch(System.IndexOutOfRangeException e)
		{
			Console.WriteLine(e.Message);
		}
	}
}
```

A *try* blokk jelöli ki a lehetséges hibaforrást, a *catch* pedig elkapja a megfelelő kivételt (arra figyeljünk, hogy ezek is blokkok, azaz a blokkon belül deklarált változók a blokkon kívül nem láthatóak). A fenti programra a következő lesz a kimenet:

```
A hiba: Index was outside the bounds of the array.
```

Látható, hogy a kivétel egy objektum formájában létezik. Minden kivétel őse a *System.Exception* osztály, így ha nem speciálisan egy kivételt akarunk elkapni, akkor írhattuk volna ezt is:

```
try
{
      array[2] = 10;
}
catch(System.Exception e)
{
      Console.WriteLine(e.Message);
}
```

Ekkor minden kivételt el fog kapni a *catch* blokk. A *System.Exception* tulajdonságai közül kettőt kell megemlítenünk:

- *Message*: ez egy olvashatóbb formája a kivétel okának.
- *InnerException*: ez alapértelmezetten null értékkel rendelkezik, akkor kap értéket, ha több kivétel is történik. Értelemszerűen ekkor a legújabb kivételt kaphatjuk el és az *InnerException* –ön keresztül követhetjük vissza az eredeti kivételig.

Nézzük meg, hogyan működnek a kivételek. Kivétel két módon keletkezhet: vagy a *throw* utasítással szándékosan mi magunk idézzük elő, vagy az alkalmazás hibás működése miatt.
Abban a pillanatban, amikor a kivétel megszületik a rendszer azonnal elkezdi keresni a legközelebbi megfelelő *catch* blokkot, elsőként abban a metódusban amelyben a kivétel keletkezett, majd ha ott nem volt sikeres, akkor abban amely ezt a metódust hívta (és így tovább amíg nem talál olyat amely kezelné).
A keresés közben két dolog is történhet: ha egy statikus tag vagy konstruktor inicializálása történik, az ekkor szintén kivétellel jár, méghozzá egy *System.TypeInitializationException*–nel, amely kivételobjektum *InnerException* tulajdonságába kerül az eredeti kivétel. A másik lehetőség, hogy nem talál megfelelő catch blokkot, ekkor a program futása – hibaüzenet társaságában – leáll.
Ha mégis talált használható *catch* blokkot, akkor a kivétel helyéről a vezérlés a talált catch –re kerül. Ha több egymásba ágyazott kivételről van szó, akkor a megelőző *catch* blokkhoz tartozó *finally* blokk fut le, majd ezután következik a *catch* blokk.

Kivételt a *throw* utasítással dobhatunk:

```
using System;

class Program
{
      static public void Main()
      {
            try
            {
                  throw new System.Exception("Kivétel. Hurrá!");
            }
            catch(Exception e)
            {
                  Console.WriteLine(e.Message);
            }
      }
}
```

A C++-tól eltérően itt példányosítanunk kell a kivételt.

A *catch*–nek nem kötelező megadni a kivétel típusát, ekkor minden kivételt elkap:

```
try
{
	throw new System.Exception();
}
catch
{
	Console.WriteLine("Kivétel. Hurrá!");
}
```

Ilyenkor viszont nem használhatjuk a kivételobjektumot.

23.1 Kivétel hierarchia

Amikor kivétel dobódik, akkor a vezérlést az első alkalmas *catch* blokk veszi át. Mivel az összes kivétel a *System.Exception* osztályból származik, így ha ezt adjuk meg a *catch*–nél, akkor az összes lehetséges kivételt el fogjuk kapni vele. Egyszerre több *catch* is állhat egymás után, de ha van olyan, amelyik az ős kivételt kapja el, akkor a program csak akkor fog lefordulni, ha az az utolsó helyen áll, hiszen a többinek esélye sem lenne.

```
using System;

class Program
{
	static public void Main()
	{
		try
		{
			int[] array = new int[2];
			array[3] = 10;
		}
		catch(System.IndexOutOfRangeException)
		{
			Console.WriteLine("OutOfRange");
		}
		catch(System.Exception)
		{
			Console.WriteLine("Exception");
		}
	}
}
```

Látható, hogy a *catch*–nek elég csak a kivétel típusát megadni, persze ekkor nem használhatjuk a kivételobjektumot.

23.2 Kivétel készítése

Mi magunk is készíthetünk kivételt, a *System.Exception*–ból származtatva:

```
using System;

class MyException : System.Exception
{
	public MyException() { }

	public MyException(string message)
		: base(message)
	{
	}

	public MyException(string message, Exception inner)
		: base(message, inner)
	{
	}
}

class Program
{
	static public void Main()
	{
		try
		{
			throw new MyException("Kivétel. Hurrá!");
		}
		catch(MyException e)
		{
			Console.WriteLine(e.Message);
		}
	}
}
```

23.3 Kivételek továbbadása

Egy kivételt az elkapása után ismét eldobhatunk. Ez hasznos olyan esetekben, amikor feljegyzést akarunk készíteni illetve, ha egy specifikusabb kivételkezelőnek akarjuk átadni a kivételt:

```
try
{
}
catch(System.ArgumentException e)
{
	throw; //továbbadjuk
	throw(new System.ArgumentNullException()); //vagy egy újat dobunk
}
```

Természetesen a fenti példában csak az egyik *throw* szerepelhetne „legálisan”, ebben a formában nem fog lefordulni.
Ilyenkor beállíthatjuk az *Exception InnerException* tulajdonságát is:

```
try
{
	throw new Exception();
}
catch(System.ArgumentException e)
{
	throw(new System.ArgumentNullException("Tovább", e));
}
```

Itt az *Exception* osztály harmadik konstruktorát használtuk, az új kivétel már tartalmazni fogja a régit is.

23.4 Finally blokk

A kivételkezelés egy problémája, hogy a kivétel keletkezése után az éppen végrehajtott programrész futása megszakad, így előfordulhat, hogy nem szabadulnak fel időben az erőforrások (megnyitott file, hálózati kapcsolat, stb), illetve objektumok olyan formában maradnak meg a memóriában, amely hibát okozhat.
Megoldást a *finally*–blokk használata jelent, amely függetlenül attól, hogy történt–e kivétel mindig lefut (kivéve, ha a *try* blokk tartalmaz *return* kifejezést):

```
using System;

class Program
{
	static public void Main()
	{
		int x = 10;

		try
		{
			Console.WriteLine("x értéke a kivétel elõtt: {0}", x);
			throw new Exception();
		}
		catch(Exception)
		{
			Console.WriteLine("Kivétel. Hurrá!");
		}
		finally
		{
			Console.WriteLine("Finally blokk");
			x = 11;
		}

		Console.WriteLine("x értéke a kivétel után {0}", x);
	}
}
```

„Valódi” erőforrások kezelésekor kényelmesebb a *using*–blokk használata, mivel az automatikusan lezárja azokat. Lényegében a *using*-blokkal használt erőforrások fordítás után a megfelelő *try-catch-finally* blokkokká alakulnak.

24 Gyakorló feladatok IV.

24.1 IEnumerator és IEnumerable

Készítsünk osztályt, amely megvalósítja az *IEnumerator* és *IEnumerable* interfészeket.

Megoldás

Korábban már találkoztunk a *foreach* ciklussal, és már tudjuk, hogy csak olyan osztályokon képes végigiterálni, amelyek megvalósítják az *IEnumerator* és *IEnumerable* interfészeket. Mindkettő a *System.Collections* névtérben található.

Elsőként nézzük az *IEnumerable* interfészt:

```
public interface IEnumerable
{
      IEnumerator GetEnumerator();
}
```

Ez a *foreach*–nek fogja szolgáltatni a megfelelő felületet, ugyanis a ciklus meghívja a metódusát, és annak vissza kell adnia az osztályt *IEnumerator*–ként (ld. implicit konverzió). Ezért kell megvalósítani egyúttal az *IEnumerator* interfészt is, ami így néz ki:

```
public interface IEnumerator
{
   bool MoveNext();
   void Reset();

   object Current { get; }
}
```

A *MoveNext* a következő elemre mozgatja a mutatót, ha tudja, ellenkező esetben (vagyis ha a lista végére ért) *false* értékkel tér vissza. A *Reset* alapértelmezésre állítja a mutatót, azaz -1 –re. Végül a *Current* (read-only) tulajdonság az aktuális pozicióban lévő elemet adja vissza. Ennek *object* típussal kell visszatérnie, hiszen minden típusra működnie kell (létezik generikus változata is, de erről később).

Használjuk az *Animal* osztályunk egy kissé módosított változatát:

```
public class Animal
{
   public Animal(string name)
   {
         this.Name = name;
   }

   public string Name { get; private set; }
}
```

Most készítsünk egy osztályt, amelyen megvalósítjuk a két interfészt, és ami tartalmaz egy *Animal* objektumokból álló listát:

```
public class AnimalContainer : IEnumerable, IEnumerator
{
   private ArrayList container = new ArrayList();
   private int currPosition = -1;

   public AnimalContainer()
   {
         container.Add(new Animal("Rex"));
         container.Add(new Animal("Bundás"));
         container.Add(new Animal("Parizer"));
   }
}
```

Ez persze még nem az egész osztály, felvettünk egy *ArrayListet*, amiben eltároljuk az objektumokat, illetve deklaráltunk egy egész számot, ami az aktuális pozíciót tárolja el és kezdőértékéül -1–et adtunk (ld. *Reset*).
Készítsük el az *IEnumerator* által igényelt metódusokat:

```
public bool MoveNext()
{
      return (++currPosition < container.Count);
}

public object Current
{
      get { return container[currPosition]; }
}

public void Reset()
{
      currPosition = -1;
}
```

Végül az *IEnumerable* interfészt valósítjuk meg:

```
public IEnumerator GetEnumerator()
{
      return (IEnumerator)this;
}
```

Ezután használhatjuk is az osztályt:

```
static public void Main()
{
      AnimalContainer ac = new AnimalContainer();

      foreach(Animal animal in ac)
      {
            Console.WriteLine(animal.Name);
      }
}
```

Amennyiben a *foreach*-en kívül akarjuk használni az osztályt pl. ha készítettünk indexelőt is, akkor gondoskodnunk kell a megfelelő konverzióról is (a *foreach* kivételt képez, mivel ő ezt megteszi helyettünk).

24.2 IComparable és IComparer

Valósítsuk meg az *IComparable* illetve *IComparer* interfészt!

Megoldás

A második gyakorlati példánkban az *IComparable* interfészt fogjuk megvalósítani, amelyre gyakran van szükségünk. Ez az interfész általában olyan adatszerkezeteknél követelmény, amelyek az elemeiken megvalósítanak valamilyen rendezést. A generikus *List* típusmak is van rendező metódusa (*Sort*), amely ezzel a metódussal dolgozik. Az *IComparable* egyetlen metódussal a *CompareTo*–val rendelkezik, amely egy *object* típust kap paraméteréül:

```
class ComparableClass : IComparable
{
   public ComparableClass(int value)
   {
         this.Value = value;
   }

   public int Value { get; private set; }

   public int CompareTo(object o)
   {
         if(o is ComparableClass)
         {
               ComparableClass c = (ComparableClass)o;
               return Value.CompareTo(c.Value);
         }
         else throw(new Exception("Nem megfelelõ objektum..."));
   }
}
```

Az osztályban a beépített típusok *CompareTo* metódusát használtuk, hiszen ők mind megvalósítják ezt az interfészt. Ez a metódus -1–et ad vissza, ha a hívó fél kisebb, 0–át, ha egyenlő és 1–et ha nagyobb. A használata:

```
static public void Main()
{
      List<ComparableClass> list = new List<ComparableClass>();
      Random r = new Random();

      for(int i = 0;i < 10;++i)
      {
         list.Add(new ComparableClass(r.Next(1000)));
      }

      foreach(ComparableClass c in list)
      {
         Console.Write("{0} ", c.Value);
      }

      Console.WriteLine("\nA rendezett lista:");

      list.Sort();

      foreach(ComparableClass c in list)
      {
         Console.Write("{0} ", c.Value);
```

```
        }

        Console.WriteLine();
}
```

A *List<T>* típusról bővebben a Generikusok című fejezetben olvashatunk.

Hasonló feladatot lát el, de jóval rugalmasabb az *IComparer* interfész. Az *IComparer* osztályok nem részei az „eredeti” osztályoknak, így olyan osztályok esetén is használhatunk ilyet, amelyek implementációjához nem férünk hozzá.
Például a *List<T>* rendezésénél megadhatunk egy összehasonlító osztályt, amely megvalósítja az *IComparer* interfészt, így tetszőleges rendezést valósíthatunk meg anélkül, hogy ki kellene egészíteni a *List<T>* -t. Most is csak egy metódust kell elkészítenünk, ez a *Compare,* amely két *object* típust vár paramétereként:

```
class ComparableClassComparer : IComparer
{
        public int Compare(object x, object y)
        {
                if(x is ComparableClass && y is ComparableClass)
                {
                        ComparableClass _x = (ComparableClass)x;
                        ComparableClass _y = (ComparableClass)y;

                        return _x.CompareTo(_y);
                }
                else throw(new Exception("Nem megfelelõ paraméter..."));
        }
}
```

A metódus elkészítésénél az egyszerűség miatt feltételeztük, hogy az összehasonlított osztály megvalósítja az *IComparable* interfészt (a beépített típusok ilyenek), természetesen mi magunk is megírhatjuk az eredményt előállító programrészt. Ezután a következőképpen rendezhetjük a listát:

```
list.Sort(new ComparableClassComparer());
```

Az *IComparer* előnye, hogy nem kötődik szorosan az osztályhoz (akár a nélkül is megírhatjuk, hogy ismernénk a belső szerkezetét), így többféle megvalósítás is lehetséges.

24.3 Mátrix típus

Készítsük el a mátrix típust és valósítsuk meg rajta az összeadás műveletet! Az egyszerűség kedvéért tegyük fel, hogy a mátrix csak egész számokat tárol.

Megoldás

```
class Matrix
{
	int[,] matrix;

	public Matrix(int n, int m)
	{
		matrix = new int[n, m];
	}

	public int N
	{
		get { return matrix.GetLength(0); }
	}

	public int M
	{
		get { return matrix.GetLength(1); }
	}

	public int this[int idxn, int idxm]
	{
		get { return matrix[idxn, idxm]; }
		set { matrix[idxn, idxm] = value; }
	}

	static public Matrix operator+(Matrix lhs, Matrix rhs)
	{
		if(lhs.N != rhs.N || lhs.M != rhs.M) return null;

		Matrix result = new Matrix(lhs.N, lhs.M);

		for(int i = 0;i < lhs.N;++i)
		{
			for(int j = 0;j < lhs.M;++j)
			{
				result[i, j] = lhs[i, j] + rhs[i, j];
			}
		}
		return result;
	}
}
```

Mátrixokat úgy adunk össze, hogy az azonos indexeken lévő értékeket összeadjuk:

```
123   145   1+1 2+4 3+5
456 + 532 = (stb…)
789   211
```

Az összeadás műveletet csakis azonos nagyságú dimenziók mellet lehet elvégezni (3x3–as mátrixhoz nem lehet hozzáadni egy 4x4 –eset). Ezt ellenőriztük is az operátor megvalósításánál.

Most csak az összeadás műveletet valósítottuk meg, a többi a kedves olvasóra vár. Plusz feladatként indexellenőrzést is lehet végezni az indexelőnél.

25 Delegate–ek

A delegate–ek olyan típusok, amelyek egy vagy több metódusra hivatkoznak. Minden delegate különálló objektum, amely egy listát tárol a meghívandó metódusokról (értelemszerűen ez egyúttal erős referencia is lesz a metódust szolgáltató osztályra). Nemcsak példány-, hanem statikus metódusokra is mutathat. Egy delegate deklarációjánál megadjuk, hogy milyen szignatúrával rendelkező metódusok megfelelőek:

```
delegate int TestDelegate(int x);
```

Ez a delegate olyan metódusra mutathat, amelynek visszatérési értéke *int* típusú és egyetlen *int* típusú paramétere van:

```
public int Pow(int x)
{
      return (x * x);
}
```

A használata:

```
TestDelegate dlgt = Pow;
int result = dlgt(10);
```

A delegate hívásakor az összes a listáján lévő metódust neghívja. A delegate–ekhez a += illetve + operátorokkal hozzáadni a -= és – operátorokkal elvenni tudunk metódusokat:

```
class Test
{
      public delegate void TestDelegate(string msg);
      private TestDelegate handler;

      public Test()
      {
            handler += Test.StaticMethod;
            handler += this.InstanceMethod;
      }

      static public void StaticMethod(string msg)
      {
            Console.WriteLine(msg);
      }

      public void InstanceMethod(string msg)
      {
            Console.WriteLine(msg);
      }

      public void CallDelegate(string msg)
      {
            handler(msg);
      }
}
```

A delegate–ek legnagyobb haszna, hogy nem kell előre megadott metódusokat használnunk, ehelyett dinamikusan adhatjuk meg az elvégzendő műveletet:

```
class Array
{
      public delegate void Transformer(ref int item);

      private int[] array;

      public Array(int length)
      {
            Length = length;
            array = new int[Length];
      }

      public int Length { get; set; }

      public int this[int idx]
      {
            get { return array[idx]; }
            set { array[idx] = value; }
      }

      public void Transform(Transformer t)
      {
            for(int i = 0;i < array.Length;++i)
            {
                  t(ref array[i]);
            }
      }
}
```

A *Transform* metódus egy delegate–et kap paraméteréül, amely elvégzi a változtatásokat a tömbön. Pl.:

```
class Program
{
      static public void TransformerMethod(ref int item)
      {
            item *= item;
      }

      static public void Main()
      {
            Array array = new Array(10);

            for(int i = 0;i < array.Length;++i)
            {
                  array[i] = i;
            }

            array.Transform(Program.TransformerMethod);

            for(int i = 0;i < array.Length;++i)
            {
                  Console.WriteLine(array[i]);
            }
      }
}
```

Két delegate szerkezetileg nem egyenlő, még akkor sem ha a szignatúrájuk megegyezik:

```
using System;

class Program
{
      public delegate void Dlgt1();
      public delegate void Dlgt2();

      static public void Method() { }

      static public void Main()
      {
            Dlgt1 d1 = Program.Method;
            Dlgt2 d2 = d1; // ez hiba
      }
}
```

Ugyanakkor ugyanazon delegate „típus” példányai között használhatjuk az == és != operátorokat. Két delegate egyenlő, ha mindkettő értéke null, illetve ha a híváslistájukon ugyanazon objektumok ugyanazon metódusai szerepelnek (vagy ugyanazok a statikus metódusok):

```
using System;

class C1
{
      public void CMethod() { }
}

class Program
{
      public delegate void Test1();

      static public void Method1() { }
      static public void Method2() { }

      static public void Main()
      {
            Test1 t1 = null;
            Test1 t2 = null;

            Console.WriteLine(t1 == t2); // True

            t1 = Program.Method1;
            t2 = Program.Method1;

            Console.WriteLine(t1 == t2); // True

            t1 += Program.Method2;

            Console.WriteLine(t1 == t2); // False

            t1 -= Program.Method2;

            C1 c1 = new C1();
            C1 c2 = new C1();
            t1 += c1.CMethod;
            t2 += c2.CMethod;

            Console.WriteLine(t1 == t2); // False
      }
}
```

25.1 Paraméter és visszatérési érték

Egy delegate–nek átadott metódus paraméterei lehetnek olyan típusok, amelyek az eredeti paraméternél általánosabbak:

```
using System;

class Animal { }

class Dog : Animal { }

class Cat : Animal { }

class Program
{
      public delegate void DogDelegate(Dog d);

      static public void AnimalMethod(Animal a) { }

      static public void Main()
      {
            DogDelegate d = AnimalMethod;
      }
}
```

Ez az ún. kontravariáns (contravariant) viselkedés.
Ennek a fordítottja igaz a visszatérési értékre, azaz az átadott metódus visszatérési értéke lehet specifikusabb az eredetinél:

```
using System;

class Animal { }

class Dog : Animal { }

class Cat : Animal { }

class Program
{
      public delegate Animal GetAnimal();

      static public Animal AnimalMethod() { return new Animal(); }
      static public Dog DogMethod() { return new Dog(); }
      static public Cat CatMethod() { return new Cat(); }

      static public void Main()
      {
            GetAnimal ga = AnimalMethod;
            Animal a = ga();

            ga = DogMethod;
            Dog d = (Dog)ga();

            ga = CatMethod;
            Cat c = (Cat)ga();

            Console.WriteLine("{0}, {1}, {2}",
                  a.GetType(), d.GetType(), c.GetType());
      }
}
```

Ezt pedig kovariáns (covariant) viselkedésnek nevezzük.

25.2 Névtelen metódusok

Egy delegate–nek nem kötelező létező metódust megadnunk, lehetőségünk van helyben kifejteni egyet. Természetesen ez a névtelen metódus a program többi részéből közvetlenül nem hívható, csak a delegate–en keresztül.

```
using System;

class Program
{
      public delegate void Test(int x);

      static public void Main()
      {
            Test t = delegate(int x)
                  {
                        Console.WriteLine(x);
                  };

            t(10);
      }
}
```

Egy névtelen metódus eléri az őt tároló blokk lokális változóit és módosíthatja is őket:

```
using System;

class Program
{
      public delegate void Test();

      static public void Main()
      {
            int x = 10;

            Test t = delegate()
                  {
                        x = 11;
                        Console.WriteLine(x);
                  };

            t();
      }
}
```

Ilyenkor figyelni kell arra, hogy a külső változóra a delegate is erős referenciával mutat, vagyis a változó akkor válik eltakaríthatóvá, ha a delegate maga is érvényét veszti.
Névtelen metódus nem használhat semmilyen ugró utasítást (pl.: *goto*, *break*, stb...).

25. Események

Egy osztály eseményeket (*event*) használhat, hogy a saját állapota megváltozásakor értesítsen más osztályokat. Ehhez a „megfigyelő” osztályoknak fel kell iratkozni a „megfigyelt” osztály eseményére azáltal, hogy az előbbiek rendelkeznek egy, az eseménynek megfelelő szignatúrájú metódussal, ún. eseménykezelővel. Az esemény megtörténtekor ezek a metódusok fognak lefutni. Eddig ez nagyon úgy hangzik, mintha a delegate–ekről beszéltünk volna, és valóban egy esemény tulajdonképpen egy speciális delegate, mindössze három dologban különböznek:

- Esemény lehet része interfésznek míg delegate nem.
- Egy eseményt csakis az az osztály „hívhat” meg amely deklarálta.
- Egy esemény rendelkezik *add* és *remove* „metódusokkal” amelyek felülbírálhatóak.

Egy esemény deklarációjában meg kell adnunk azt a delegate–et amely az eseményhez szükséges szignatúrát definiálja. Nézzünk egy egyszerű példát:

```
class Test
{
      public delegate void EventHandlerDelegate(string message);
      public event EventHandlerDelegate TestStatusChange;

      private int data = 10;
      public int Data
      {
            get { return data; }
            set
            {
                  data = value;
                  this.OnStatusChange();
            }
      }

      private void OnStatusChange()
      {
            if(TestStatusChange != null)
            {
                  TestStatusChange("Az osztály állapota megváltozott!");
            }
      }
}
```

Nem feltétlenül kell delegate–et deklarálnunk, mivel rendelkezésünkre áll a beépített általános *EventHandler* delegate, amely két paraméterrel (erről hamarosan) rendelkezik és *void* visszatérési típussal bír.

Az esemény akkor fog beindulni, amikor a *data* mező értéke megváltozik. Ekkor meghívjuk az *OnStatusChanged* metódust, amely elsőként megvizsgálja, hogy az eseményre feliratkoztak–e vagy sem - utóbbi esetben a hívás kivételt váltana ki. Ezt az osztály így használhatjuk:

```
class Program
{
	static public void Handler(string message)
	{
		Console.WriteLine(message);
	}

	static public void Main()
	{
		Test t = new Test();
		t.TestStatusChanged += Program.Handler;
		t.Data = 11;
	}
}
```

Az eseményekhez rendelt eseménykezelőknek konvenció szerint (ettől eltérhetünk, de a Framework eseményei mind ilyenek) két paramétere van, az első az az objektum, amely kiváltotta az eseményt, a második pedig az eseményhez kapcsolódó információk. A második paraméter ekkor olyan típus lehet, amely az *EventArgs* osztályból származik. Módosítsuk ennek megfelelően a fenti programot. Elsőként készítünk egy *EventArgs* osztályból származó új osztályt, amely képes tárolni az eseményhez kapcsolódó üzenetet (az *EventArgs* alapértelmezetten nem redelekezik ilyesmivel csak egy alaposztály a specializált eseményekhez):

```
class TestEventArgs : EventArgs
{
	public TestEventArgs(string message)
		: base()
	{
		this.Message = message;
	}

	public string Message { get; set; }
}
```

Ezután már csak módosítani kell a delegate–et és az esemény kiváltását:

```
public delegate void EventHandlerDelegate(object sender, TestEventArgs e);

private void OnStatusChange()
{
	if(TestStatusChanged != null)
	{
		TestStatusChanged(this, new TestEventArgs("Az osztály állapota
megváltozott"));
	}
}
```

A *sender* paraméternek a *this*–szel adjuk meg az értékét, ezután explicit konverzióval visszakaphatjuk belőle a küldő példányt (az első paraméter szintén konvenció szerint minden esetben *object* típusú lesz, mivel ugyanazt az eseményt használhatjuk különböző osztályokkal).

Még módosítsuk az eseménykezelőt is:

```
static public void Handler(object sender, TestEventArgs e)
{
	Console.WriteLine(e.Message);
}
```

A következő példában az események valódi hasznát fogjuk látni. Készítsünk egy egyszerű kliens-szerver alkalmazást (persze nem a hálózaton, csak szimuláljuk). A kliensek csatlakozhatnak a szerverhez (ezután pedig kiléphetnek). A feladat, hogy minden ilyen eseményről küldjünk értesítést az összes csatlakozott kliensnek.
Normális esetben a szerver osztálynak tárolnia kellene a kliensek hivatkozásait pl. egy tömbben. A probléma, hogy ez azért nem olyan egyszerű, hiszen gondoskodni kell arról, hogy a kilépett klienseket töröljük a listából, illetve a lista mérete is gondot jelenthet. Események alkalmazásával viszont nagyon egyszerű lesz a dolgunk.
Készítsünk egy *EventArgs* osztályt, amely segítségünkre lesz az eseménykezelők értesítésében:

```
class ServerEventArgs : EventArgs
{
	public ServerEventArgs(string message)
		: base()
	{
		this.Message = message;
	}

	public string Message { get; set; }
}
```

Most pedig a szervert készítjük el:

```
class Server
{
	public delegate void ServerEvent(object sender, ServerEventArgs e);
	public event ServerEvent ServerChange;

	public Server() { }

	public void Connect(Client client)
	{
		this.ServerChange += client.ServerMessageHandler;
		OnServerChange(string.Format("Felhasználó <{0}> csatlakozott!",
			client.Name));
	}

	public void Disconnect(Client client)
	{
		OnServerChange(string.Format("Felhasználó <{0}> kilépett!",
			client.Name));
		this.ServerChange -= client.ServerMessageHandler;
	}

	protected void OnServerChange(string message)
	{
		if(ServerChange != null)
		{
			ServerChange(this, new ServerEventArgs(message));
		}
	}
}
```

Látható, hogy a kliensek kezelése nagyon egyszerű, mindössze egy műveletet kell elvégeznünk az eseményekre való feliratkozásért/leiratkozásért. Nézzük meg a kliens osztályt:

```
class Client
{
      public Client(string name)
      {
            Name = name;
      }

      public string Name { get; set; }

      public void ServerMessageHandler(object sender, ServerEventArgs e)
      {
            Console.WriteLine("{0} üzenetet kapott: {1}",
                  this.Name, e.Message);
      }
}
```

Végül a *Main*:

```
static public void Main()
{
      Server server = new Server();

      Client c1 = new Client("Józsi");
      Client c2 = new Client("Béla");
      Client c3 = new Client("Tomi");

      server.Connect(c1);
      server.Connect(c2);
      server.Disconnect(c1);
      server.Connect(c3);
}
```

26 Generikusok

Az objektum-orientált programozás egyik alapköve a kód-újrafelhasználás, vagyis, hogy egy adott kódrészletet elég általánosra írjunk meg ahhoz, hogy minél többször felhasználhassuk. Ennek megvalósítására két eszköz áll rendelkezésünkre, az egyik az öröklődés, a másik pedig jelen fejezet tárgya a generikusok.

26.1 Generikus metódusok

Vegyük a következő metódust:

```
static public void Swap(ref int x, ref int y)
{
      int tmp = x;
      x = y;
      y = tmp;
}
```

Ha szeretnénk, hogy ez a metódus más típusokkal is működjön, akkor bizony sokat kell gépelnünk. Kivéve, ha írunk egy generikus metódust:

```
static public void Swap<T>(ref T x, ref T y)
{
      T tmp = x;
      x = y;
      y = tmp;
}
```

A T fogja jelképezni az aktuális típust (lehet más nevet is adni neki, eredetileg a Template szóból jött), ezt generikus paraméternek hívják. Generikus paramétere csakis osztálynak vagy metódusnak lehet és ebből többet is használhatnak. Ezután a metódust a hagyományos úton használhatjuk, a fordító felismeri, hogy melyik típust használjuk (ezt megadhatjuk mi magunk is explicite):

```
static public void Main()
{
      int x = 10;
      int y = 20;

      Program.Swap<int>(ref x, ref y);

      Console.WriteLine("x == {0} és y == {1}", x, y);

      string s1 = "alma";
      string s2 = "dió";

      Program.Swap<string>(ref s1, ref s2);

      Console.WriteLine("s1 == {0} és s2 == {1}", s1, s2);
}
```

A C# generikusai hasonlítanak a C++ sablonjaira, de annál kevésbé hatékonyabbak, cserében sokkal biztonságosabb a használatuk. Két fontos különbség van a kettő

közt: míg a C++ fordítási időben készíti el a specializált metódusokat/osztályokat, addig a C# ezt a műveletet futási időben végzi el. A másik eltérés az elsőből következik, mivel a C++ fordításkor ki tudja szűrni azokat az eseteket, amelyek hibásak, pl., összeadunk két típust a sablonmetódusban, amelyeken nincs értelmezve összeadás. A C# ezzel szemben kénytelen az ilyen problémákat megelőzni, a fordító csakis olyan műveletek elvégzését fogja engedélyezni, amelyek mindenképpen működni fognak. A következő példa nem fog lefordulni:

```
static public T Sum<T>(T x, T y)
{
      return x + y;
}
```

Fordításkor csak azt tudja ellenőrizni, hogy létező típust adtunk–e meg, így nem tudhatja a fordító, hogy sikeres lesz–e a végrehajtás, ezért a fenti kód az összeadás miatt (nem feltétlenül valósítja meg minden típus) „rossz”.

26.2 Generikus osztályok

Képzeljük el, hogy azt a feladatot kaptunk, hogy készítsünk egy verem típust, amely bármely típusra alkalmazható. Azt is képzeljük el, hogy még nem hallottunk generikusokról. Így a legkézenfekvőbb megoldás, ha az elemeket egy *object* típusú tömbben tároljuk:

```
class Stack
{
   object[] t;
   int pointer;
   readonly int size;

   public Stack(int capacity)
   {
         t = new object[capacity];
         size = capacity;
         pointer = 0;
   }

   public void Push(object item)
   {
         if(pointer >= size)
         {
                  throw(new StackOverflowException("Tele van..."));
         }

         t[pointer++] = item;
   }

   public object Pop()
   {
         if(pointer-- >= 0) { return t[pointer]; }

         pointer = 0;
         throw(new InvalidOperationException("Üres..."));
   }
}
```

Ezt most a következőképpen használhatjuk:

```
static public void Main()
{
      Stack s = new Stack(10);

      for(int i = 0;i < 10;++i)
      {
            s.Push(i);
      }

      for(int i = 0;i < 10;++i)
      {
            Console.WriteLine((int)s.Pop());
      }
}
```

Működni működik, de se nem hatékony, se nem kényelmes. A hatékonyság az érték/referenciatípusok miatt csökken jelentősen (ld. boxing/unboxing), a kényelem pedig amiatt, hogy mindig figyelni kell, épp milyen típussal dolgozunk, nehogy olyan kasztolással éljünk ami kivételt dob.
Ezeket a problémákat könnyen kiküszöbölhetjük, ha generikus osztályt készítünk:

```
class Stack<T>
{
   T[] t;
   int pointer;
   readonly int size;

   public Stack(int capacity)
   {
         t = new T[capacity];
         size = capacity;
         pointer = 0;
   }

   public void Push(T item)
   {
         if(pointer >= size)
         {
                  throw(new StackOverflowException("Tele van..."));
         }

         t[pointer++] = item;
   }

   public object Pop()
   {
         if(pointer-- >= 0)
         {
               return t[pointer];
         }

         pointer = 0;
         throw(new InvalidOperationException("Üres..."));
   }
}
```

Ezután akármelyik típuson könnyen használhatjuk:

```
static public void Main()
{
	Stack<int> s = new Stack<int>(10);

	for(int i = 0;i < 10;++i)
	{
		s.Push(i);
	}

	for(int i = 0;i < 10;++i)
	{
		Console.WriteLine(s.Pop());
	}
}
```

26.3 Generikus megszorítások

Alapértelmezetten egy generikus paraméter bármely típust jelképezhet. A deklarációnál azonban kiköthetünk megszorításokat a paraméterre. Ezeket a *where* kulcsszóval vezetjük be:

```
where T : alaposztály
where T : interfész
where T : osztály
where T : struktúra
where T : new()
where T : U
```

Az utolsó két sor magyarázatra szorul. A *new()* megszorítás olyan osztályra utal, amely rendelkezik alapértelmezett konstruktorral. Az U pedig ebben az esetben egy másik generikus paramétert jelöl, vagyis T olyan típusnak felel meg amely vagy U –ból származik, vagy egyenlő vele.

Nézzünk néhány példát:

```
class Test<T>
	where T : class
{

}
```

Ezt az osztályt csak referenciatípusú generikus paraméterrel példányosíthatjuk, minden más esetben fordítási hibát kapunk.

```
class Test<T>
	where T : struct
{

}
```

Ez pedig épp az ellenkezője, értéktípusra van szükség.

```
class Test<T>
      where T : IEnumerable
{

}
```

Most csakis *IEnumerable* interfészt megvalósító típussal példányosíthatjuk az osztályt.

```
class Base { }
class Derived : Base { }

class Test<T>
      where T : Base
{

}
```

Ez már érdekesebb. Ez a megszorítás a generikus paraméter ősosztályára vonatkozik, vagyis példányosíthatunk a *Base* és a *Derived* típussal is.

```
class DefConst
{
      public DefConst() { }
}

class Test<T>
      where T : new()
{

}
```

Itt olyan típusra van szükségünk, amely rendelkezik alapértelmezett konstruktorral, a *DefConst* osztály is ilyen.

```
class Base { }
class Derived : Base { }

class Test<T, U>
      where T : U
{

}
```

Most T típusának olyannak kell lennie, amely implicit módon konvertálható U típusára, vagyis T vagy megegyezik U–val, vagy belőle származik:

```
static public void Main()
{
      Test<Derived, Base> t1 = new Test<Derived, Base>(); // ez jó
      Test<Base, Derived> t2 = new Test<Base, Derived>(); // ez nem jó
}
```

Értelemszerűen írhattunk volna <*Base, Base*>-t, vagy <*Derived, Derived*>-et is.

26.4 Öröklődés

Generikus osztályból származtathatunk is, ekkor vagy az ősosztály egy specializált változatát vesszük alapul, vagy a nyers generikus osztályt:

```
class Base<T>
{

}

class Derived<T> : Base<T>
{

}

//vagy

class IntDerived : Base<int>
{

}
```

26.5 Statikus tagok

Generikus típusok esetében minden típushoz külön statikus tag tartozik:

```
using System;

class Test<T>
{
      static public int Value;
}

class Program
{
      static public void Main()
      {
            Test<int>.Value = 10;
            Test<string>.Value = 20;

            Console.WriteLine(Test<int>.Value); // 10
            Console.WriteLine(Test<string>.Value); // 20
      }
}
```

26.6 Generikus gyűjtemények

A C# 2.0 bevezetett néhány hasznos generikus adatszerkezetet, többek közt listát és vermet. A következőkben megvizsgálunk közülük néhányat. Ezek a szerkezetek a *System.Collections.Generic* névtérben találhatóak.

26.6.1 List<T>

A *List<T>* az *ArrayList* generikus, erősen típusos megfelelője. A legtöbb esetben a *List<T>* hatékonyabb lesz az *ArrayList* –nél, emellett pedig típusbiztos is.
Amennyiben értéktípussal használjuk a *List<T>*-t az alapértelmezetten nem igényel bedobozolást, de a *List<T>* rendelkezik néhány olyan művelettel, amely viszont igen, ezek főleg a kereséssel kapcsolatosak. Azért, hogy az ebből következő teljesítményromlást elkerüljük, a használt értéktípusnak meg kell valósítania az *IComparable* és az *IEquatable* interfészeket (a legtöbb beépített egyszerű (érték)típus ezt meg is teszi) (ezeket az interfészeket az összes többi gyűjtemény is igényli).

```
using System;
using System.Collections.Generic;

class Program
{
	static public void Main()
	{
		List<int> list = new List<int>();

		for(int i = 0;i < 10;++i)
		{
			list.Add(i);
		}

		foreach(int item in list)
		{
			Console.WriteLine(item);
		}
	}
}
```

Az *Add* metódus a lista végéhez adja hozzá a paraméterként megadott elemet, hasonlóan az *ArrayList*–hez. Használhatjuk rajta az indexelő operátort is.

A lista elemeit könnyen rendezhetjük a *Sort* metódussal (ez a metódus igényli, hogy a lista típusa megvalósítsa az *IComparable* interfészt):

```
using System;
using System.Collections.Generic;

class Program
{
	static public void Main()
	{
		List<int> list = new List<int>();

		Random r = new Random();
		for(int i = 0;i < 10;++i)
		{
			list.Add(r.Next(1000));
		}

		list.Sort();
	}
}
```

Kereshetünk is az elemek között a *BinarySearch* metódussal, amely a keresett objektum indexét adja vissza:

```
using System;
using System.Collections.Generic;

class Program
{
	static public void Main()
	{
		List<string> list = new List<string>()
		{
			"alma", "dió", "körte", "barack"
		};

		Console.WriteLine(list[list.BinarySearch("körte")]);
	}
}
```

Megkereshetjük az összes olyan elemet is, amely eleget tesz egy feltételnek a *Find* és *FindAll* metódusokkal. Előbbi az első, utóbbi az összes megfelelő példányt adja vissza egy *List<T>* szerkezetben:

```
using System;
using System.Collections.Generic;

class Program
{
	static public void Main()
	{
		List<int> list = new List<int>();
		Random r = new Random();

		for(int i = 0;i < 100;++i)
		{
			list.Add(r.Next(1000));
		}

		Console.WriteLine("Az elso páros szám a listában: {0}",
			list.Find(delegate(int item)
			{
				return item % 2 == 0;
			}));

		List<int> evenList = list.FindAll(delegate(int item)
			{
				return item % 2 == 0;
			});

		Console.WriteLine("Az összes páros elem a listában:");

		evenList.ForEach(delegate(int item)
			{
				Console.WriteLine(item);
			});
	}
}
```

A feltételek megadásánál és a páros számok listájának kiíratásánál névtelen metódusokat használtunk.

Újdonságot jelent a listán metódusként hívott *foreach* ciklus. Ezt a C# 3.0 vezette be és az összes generikus adatszerkezet rendelkezik vele, lényegében teljesen ugyanúgy működik mint egy „igazi” *foreach*. paramétereként egy „*void Method(T item)* szignatúrájú metódust (vagy névtelen metódust) vár, ahol T a lista elemeinek típusa.

26.6.2 SortedList<T, U> és SortedDictionary<T, U>

A *SortedList<T, U>* kulcs – érték párokat tárol el és a kulcs alapján rendezi is őket:

```
using System;
using System.Collections.Generic;

class Program
{
      static public void Main()
      {
            SortedList<string, int> list = new SortedList<string, int>();

            list.Add("egy", 1);
            list.Add("kettõ", 2);
            list.Add("három", 3);
      }
}
```

A lista elemei tulajdonképpen nem a megadott értékek, hanem a kulcs – érték párokat reprezentáló *KeyValuePair<T, U>* objektumok. A lista elemeinek eléréséhez is használhatjuk ezeket:

```
foreach(KeyValuePair<string, int> item in list)
{
      Console.WriteLine("Kulcs == {0}, Érték == {1}",
            item.Key, item.Value);
}
```

A lista kulcsai csakis olyan típusok lehetnek, amelyek megvalósítják az *IComparable* interfészt, hiszen ez alapján történik a rendezés. Ha ez nem igaz, akkor mi magunk is definiálhatunk ilyet, részletekért ld. az Interfészek fejezetet.
A listában minden kulcsnak egyedinek kell lennie (ellenkező esetben kivételt kapunk), illetve kulcs helyén nem állhat null érték (ugyanez viszont nem igaz az értékekre).

A *SortedDictionary<T, U>* használata gyakorlatilag megegyezik a *SortedList<T, U>*-val, a különbség a teljesítményben és a belső szerkezetben van.
A SD új (rendezetlen) elemek beszúrását gyorsabban végzi, mint a SL (O(Log n) és O(n)). Előre rendezett elemek beszúrásánál pont fordított a helyzet. Az elemek közti keresés mindkét szerkezetben O(Log n). Ezen kívül a SL kevesebb memóriát használ fel.

26.6.3 Dictionary<T, U>

A *SortedDictionary<T, U>* rendezetlen párja a *Dictionary<T, U>:*

```
using System;
using System.Collections.Generic;

class Program
{
	static public void Main()
	{
		Dictionary<string, int> list = new Dictionary<string, int>();

		list.Add("egy", 1);
		list.Add("kettõ", 2);
		list.Add("három", 3);

		foreach(KeyValuePair<string, int> item in list)
		{
			Console.WriteLine("Kulcs == {0}, Érték == {1}",
				item.Key, item.Value);
		}
	}
}
```

Teljesítmény szempontjából a *Dictionary<T, U>* mindig jobb eredményt fog elérni (egy elem keresése kulcs alapján O(1)), ezért ha nem fontos szempont a rendezettség, akkor használjuk ezt.

26.6.4 LinkedList<T>

A *LinkedList<T>* egy kétirányú láncolt lista. Egy elem beillesztése illetve eltávolítása O(1) nagyságrendű művelet. A lista minden tagja különálló objektum, egy-egy *LinkedListNode<T>* példány. A *LLN<T> Next* és *Previous* tulajdonságai a megelőző illetve a következő elemre mutatnak.
A lista *First* tulajdonsága az első, *Last* tulajdonsága pedig az utolsó tagra mutat. Elemeket az *AddFirst* (első helyre szúr be) és *AddLast* (utolsó helyre tesz) metódusokkal tudunk beilleszteni.

```
using System;
using System.Collections.Generic;

class Program
{
	static public void Main()
	{
		LinkedList<string> list = new LinkedList<string>();

		list.AddLast("alma");
		list.AddLast("dió");
		list.AddLast("körte");
		list.AddFirst("narancs");

		LinkedListNode<string> current = list.First;

		while(current != null)
```

```
            {
                  Console.WriteLine(current.Value);
                  current = current.Next;
            }
      }
}
```

Ebben a példában bejárunk egy láncolt listát, a kimeneten a „narancs” elemet látjuk majd első helyen, mivel őt az *AddFirst* metódussal helyeztük be.

26.6.5 ReadOnlyCollection<T>

Ahogy a nevéből látszik ez az adatszerkezet az elemeit csak olvasásra adja oda. A listához nem adhatunk új elemet sem (ezt nem is támogatja), csakis a konstruktorban tölthetjük fel.

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel; // ez is kell

class Program
{
      static public void Main()
      {
            List<string> list = new List<string>()
            {
                  "alma", "körte", "dió"
            };

            ReadOnlyCollection<string> roc = new
ReadOnlyCollection<string>(list);

            foreach(string item in roc)
            {
                  Console.WriteLine(item);
            }
      }
}
```

26.7 Generikus interfészek, delegate –ek és események

A legtöbb hagyományos interfésznek létezik generikus változata is. Például az *IEnumerable* és *IEnumerator* is ilyen:

```
class MyClass<T> : IEnumerable<T>, IEnumerator<T>
{

}
```

Ekkor a megvalósítás teljesen ugyanúgy működik, mint a hagyományos esetben, csak épp használnunk kell a generikus paraméter(eke)t.

A generikus adatszerkezetek (többek között) a generikus *ICollection, IList* és *IDictionary* interfészeken alapulnak, így ezeket megvalósítva akár mi magunk is létrehozhatunk ilyet.

Az interfészekhez hasonlóan a delegate–ek és események is lehetnek generikusak. Ez az ő esetükben egyáltalán nem jár semmilyen „extra” kötelezettséggel.

26.8 Kovariancia és kontravariancia

Nézzük a következő „osztályhierarchiát”:

```
class Person
{

}

class Student : Person
{

}
```

A polimorfizmus elve miatt minden olyan helyen, ahol egy *Person* objektum használható ott egy *Studen*t objektum is megfelel, legalábbis elvileg. Lássuk, a következő kódot:

```
List<Student> studentList = new List<Student>();
List<Person> personList = studentList;
```

A fenti két sor, pontosabban a második nem fordul le, mivel a .NET nem tekinti egyenlőnek a generikus paramétereket, még akkor sem, ha azok kompatibilisak lennének. Azt mondjuk, hogy a generikus paraméterek nem kovariánsak (covariance). A dolog fordítottja is igaz, vagyis nincs kompatibilitás az általánosabb típusról a szűkebbre sem, nem kontravariánsak (contravariance).
Miért van ez így? Képzeljük el azt a helyzetet, amikor a fenti osztályokat kiegészítjük még egy *Teacher* osztállyal, amely szintén a *Person* osztályból származik. Ha a generikus paraméterek kovariánsan viselkednének, akkor lehetséges lenne *Student* és *Teacher* objektumokat is egy listába tenni ez pedig azzal a problémával jár, hogy lehetséges lenne egy elem olyan tulajdonságát módosítani amellyel nem rendelkezik, ez pedig nyílván hibát okoz (persze típusellenőrzéssel ez is áthidalható, de ezzel az egész generikus adatszerkezet értelmét vesztené).
A .NET 4.0 bevezeti a kovariáns és kontravariáns típusparamétereket, úgy oldva meg a fent vázolt problémát, hogy a kérdéses típusok csak olvashatóak illetve csak írhatóak lesznek. A következő példában egy generikus delegate segítségével nézzük meg az új lehetőségeket (új listaszerkezetet írni bonyolultabb lenne) (a példa megértéséhez szükség van a lambda kifejezések ismeretére). Először egészítsük ki a *Person* osztály egy *Name* tulajdonsággal:

```
class Student : Person
{
      public string Name { get; set; }
}
```

Most lássuk a forrást:

```
delegate void Method<T>();

class Program
{
      static public void Main()
      {
            Method<Student> m1 = () => new Student();
            Method<Person> m2 = m1;
      }
}
```

A fenti kód nem fordul le, módosítsuk a delegate deklarációját:

```
delegate void Method<out T>();
```

Most viszont minden működik, hiszen biztosítottuk, hogy minden típust megfelelően kezeljük. Most lássuk a kontravarianciát:

```
delegate void Method<in T>(T t);

class Program
{
      static public void Main()
      {
            Method<Person> m1 = (person) => Console.WriteLine(person.Name);
            Method<Student> m2 = m1;
      }
}
```

A .NET 4.0–tól kezdve az összes fontosabb interfész (pl.: *IEnumerable*, *IComparable*, etc...) képes a kovariáns illetve kontravariáns viselkedésre.

27 Lambda kifejezések

A C# 3.0 bevezeti a lambda kifejezéseket. Egy lambda kifejezés gyakorlatilag megfelel egy névtelen metódus „civilizáltabb”, elegánsabb változatának (ugyanakkor első ránézésre talán ijesztőbb, de ha megszokta az ember sokkal olvashatóbb kódot eredményez).
Minden lambda kifejezés tartalmazza az ún. lambda operátort (=>), ennek jelentése nagyjából annyi, hogy „legyen”. Az operátor bal oldalán a bemenő változók, jobb oldalán pedig a bemenetre alkalmazott kifejezés áll.
Mivel névtelen metódus ezért egy lambda kifejezés állhat egy delegate értékadásában is, elsőként ezt nézzük meg:

```
using System;

class Program
{
      public delegate int IntFunc(int x);

      static public void Main()
      {
            IntFunc func = (x) => (x * x);
            Console.WriteLine(func(10));
      }
}
```

Egy olyan metódusra van tehát szükség amely egy *int* típusú bemenő paramétert vár és ugyanilyen típust ad vissza. A lambda kifejezés bal oldalán a bemenő paraméter (x) jobb oldalán pedig a visszadatott értékről gondoskodó kifejezés (x * x) áll. A bemenő paraméternél nem kell (de lehet) explicit módon jeleznünk a típust, azt a fordító magától „kitalálja” (a legtöbb esetre ez igaz, de néha szükség lesz rá, hogy jelöljük a típust).
Természetesen nem csak egy bemenő paramétert használhatunk, a következő példában összeszorozzuk a lambda kifejezés két paraméterét:

```
using System;

class Program
{
      public delegate int IntFunc2(int x, int y);

      static public void Main()
      {
            IntFunc2 func = (x, y) => (x * y);
            Console.WriteLine(func(10, 2));
      }
}
```

27.1 Generikus kifejezések

Generikus kifejezéseknek (tulajdonképpen ezek generikus delegate–ek) is megadhatunk lambda kifejezéseket, amelyek nem igénylik egy előzőleg definiált delegate jelenlétét, ezzel önálló lambda kifejezéseket hozva létre (ugyanakkor a

generikus kifejezések kaphatnak névtelen metódusokat is). Kétféle generikus kifejezés létezik a *Func,* amely adhat visszatérési értéket és az *Action*, amely nem (*void*) (lásd: függvény és eljárás). Elsőként a *Func*–ot vizsgáljuk meg:

```
using System;

class Program
{
	static public void Main()
	{
		Func<int, int> func = (x) => (x * x);

		Console.WriteLine(func(10));
	}
}
```

A generikus paraméterek között utolsó helyen mindig a visszatérési érték áll, előtte pedig a bemenő paraméterek (maximum négy) kapnak helyet.

```
using System;

class Program
{
	static public void Main()
	{
		Func<int, int, bool> func = (x, y) => (x > y);

		Console.WriteLine(func(10, 5)); // True
	}
}
```

Most megnéztük, hogy az első paraméter nagyobb–e a másodiknál. Értelemszerűen a lambda operátor jobb oldalán lévő kifejezésnek megfelelő típust kell eredményeznie, ezt a fordító ellenőrzi.
A *Func* minden esetben rendelkezik legalább egy paraméterrel, mégpedig a visszatérési érték típusával, ez biztosítja, hogy mindig legyen visszatérési érték.

```
using System;

class Program
{
	static public void Main()
	{
		Func<bool> func = () => true;

		Console.WriteLine(func()); // True
	}
}
```

A *Func* párja az *Action,* amely szintén maximum négy bemenő paramétert kaphat, de nem lehet visszatérési értéke:

```
using System;

class Program
{
	static public void Main()
	{
		Action<int> act = (x) => Console.WriteLine(x);
		act(10);
	}
}
```

27.2 Kifejezésfák

Generikus kifejezések segítségével felépíthetünk kifejezésfákat, amelyek olyan formában tárolják a kifejezésben szereplő adatokat és műveleteket, hogy futási időben a CLR ki tudja azt értékelni. Egy kifejezésfa egy generikus kifejezést kap generikus paraméterként:

```
using System;
using System.Linq.Expressions; // ez kell

class Program
{
	static public void Main()
	{
		Expression<Func<int, int, bool>> expression =
			(x, y) => (x > y);

		Console.WriteLine(expression.Compile().Invoke(10, 2)); // True
	}
}
```

A programban először IL kódra kell fordítani (*Compile*), csak azután hívhatjuk meg.

27.3 Lambda kifejezések változóinak hatóköre

Egy lambda kifejezésben hivatkozhatunk annak a metódusnak a paramétereire és lokális változóira, amelyben definiáltuk. A külső változók akkor értékelődnek ki, amikor a delegate ténylegesen meghívódik, nem pedig a deklaráláskor, vagyis az adott változó legutolsó értékadása számít majd. A felhasznált változókat inicializálni kell, mielőtt használnánk egy lambda kifejezésben. A lambda kifejezés fenntart magának egy másolatot a lokális változóból/paraméterből, még akkor is, ha az időközben kifut a saját hatóköréből:

```
using System;

class Test
{
	public Action<int> act;

	public void Method()
	{
		int local = 11;
		act = (x) => Console.WriteLine(x * local);
```

```
            local = 100;
        }
}

class Program
{
        static public void Main()
        {
            Test t = new Test();
            t.Method();
            t.act(100);
        }
}
```

Ez a program 10000–et fog kiírni, vagyis valóban a legutolsó értékét használta a lambda a lokális változónak.

A lokális változók és paraméterek módosíthatóak egy lambda kifejezésben. A lambda kifejezésben létrehozott változók ugyanúgy viselkednek, mint a hagyományos lokális változók, a delegate minden hívásakor új példány jön létre belőlük.

27.4 Névtelen metódusok kiváltása lambda kifejezésekkel

Lambda kifejezést használhatunk minden olyan helyen, ahol névtelen metódus állhat. Nézzük meg pl., hogy hogyan használhatjuk így a *List<T>* típust:

```
using System;
using System.Collections.Generic;

class Program
{
        static public void Main()
        {
            List<int> list = new List<int>();

            for(int i = 1;i < 10;++i)
            {
                list.Add(i);
            }

            int result = list.Find((item) => (item % 2 == 0));

            Console.WriteLine("Az elsõ páros szám: {0}", result);

            List<int> oddList = list.FindAll((item) => (item % 2 != 0));

            Console.WriteLine("Az összes páratlan szám:");

            oddList.ForEach((item) => Console.WriteLine(item));
        }
}
```

Eseménykezelőt is írhatunk így:

```
class Test
{
	public event EventHandler TestEvent;

	public void OnTestEvent()
	{
		if(TestEvent != null)
		{
			TestEvent(this, null);
		}
	}
}
```

Az *EventHandler* általános delegate–et használtuk az esemény deklarációjánál. Az esemény elindításánál nincs szükségünk most *EventArgs* objektumra, ezért itt nyugodtan használhatunk null értéket.
Most nézzük a programot:

```
class Program
{
	static public void Main()
	{
		Test t = new Test();

		t.TestEvent += (sender, e) =>
			{
				Console.WriteLine("Eseménykezelõ!");
			};

		t.OnTestEvent();
	}
}
```

Lambda kifejezés helyett ún. lambda állítást írtunk, így akár több soros utasításokat is adhatunk.

28 Attribútumok

Már találkoztunk nyelvbe épített módosítókkal, mint amilyen a *static* vagy a *virtual*. Ezek általában be vannak betonozva az adott nyelvbe, mi magunk nem készíthetünk újakat – kivéve, ha a .NET Framework–kel dolgozunk.
Egy attribútum a fordítás alatt beépül a Metadata információkba, amelyeket a futtató környezet (a CLR) felhasznál majd az objektumok kreálása során.

Egy tesztelés során általánosan használt attribútum a *Conditional*, amely egy előfordító által definiált szimbólumhoz köti programrészek végrehajtását. A *Conditional* a *System.Diagnostics* névtérben rejtőzik:

```
#define DEBUG // definiáljuk a DEBUG szimbólumot

using System;
using System.Diagnostics; // ez is kell

class DebugClass
{
        [Conditional("DEBUG")] // ha a DEBUG létezik
        static public void DebugMessage(string message)
        {
                Console.WriteLine("Debugger üzenet: {0}", message);
        }
}

class Program
{
        static public void Main()
        {
                DebugClass.DebugMessage("Main metódus");
        }
}
```

Egy attribútumot mindig szögletes zárójelek közt adunk meg. Ha a programban nem lenne definiálva az adott szimbólum a metódus nem futna le.

Szimbólumok definícióját mindig a forrásfile elején kell megtennünk, ellenkező esetben a program nem fordul le.

Minden olyan osztály, amely bármilyen módon a *System.Attribute* absztrakt osztályból származik felhasználható attribútumként. Konvenció szerint minden attribútum osztály neve a névből és az utána írt „Attribute” szóból áll, így a *Conditional* eredeti neve is *ConditionalAttribute*, de az utótagot tetszés szerint elhagyhatjuk, mivel a fordító így is képes értelmezni a kódot.

Ahogy az már elhangzott, mi magunk is készíthetünk attribútumokat:

```
class TestAttribute : System.Attribute { }

[Test]
class C { }
```

Ez nem volt túl nehéz, persze az attribútum sem nem túl hasznos. Módosítsuk egy kicsit! Egy attribútum-osztályhoz több szabályt is köthetünk a használatára vonatkozóan, pl. megadhatjuk, hogy milyen típusokon használhatjuk. Ezeket a szabályokat az *AttributeUsage* osztállyal deklarálhatjuk. Ennek az osztálynak egy kötelezően megadandó és két opcionális paramétere van: *ValidOn* illetve *AllowMultiple* és *Inherited*. Kezdjük az elsővel: a *ValidOn* azt határozza meg, hogy hol használhatjuk az adott attribútumot, pl. csak referenciatípuson vagy csak metódusoknál (ezeket akár kombinálhatjuk is a bitenkénti vagy operátorral(|)).

```
[AttributeUsage(AttributeTargets.Class)]
class TestAttribute : System.Attribute { }

[Test]
class C { }

[Test] // ez nem lesz jó
struct S { }
```

Így nem használhatjuk ezt az attribútumot. Módosítsunk rajta:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
class TestAttribute : System.Attribute { }
```

Most már működni fog érték- és referenciatípusokkal is. Az *AttributeTargets.All* pedig azt mondja meg, hogy bárhol használhatjuk az attribútumot.

Lépjünk az *AllowMultiple* tulajdonságra! Ő azt fogja jelezni, hogy egy szerkezet használhat-e többet is egy adott attribútumból:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
class TestAttribute : System.Attribute { }

[Test]
[Test] // ez már sok
class C { }
```

Végül az *Inherited* tulajdonság azt jelöli, hogy az attribútummal ellátott osztály leszármazottai is öröklik-e az attribútumot:

```
[AttributeUsage(AttributeTargets.Class, Inherited = true)]
class TestAttribute : System.Attribute { }

[Test]
class C { }

class CD : C { }
```

A CD osztály a C osztály leszármazottja és mivel az attribútum *Inherited* tulajdonsága true értéket kapott ezért ő is örökli az ősosztálya attribútumát.

Attribútumok kétféle paraméterrel rendelkezhetnek: positional és named. Előbbinek mindig kötelező értéket adnunk, tulajdonképpen ez a konstruktor paramétere lesz (ilyen az *AttributeUsage* osztály *ValidOn* tulajdonsága, amelynek mindig kell értéket

adnunk). Utóbbiak az opcionális paraméterek, amelyeknek van alapértelmezett értéke, így nem kell kötelezően megadnunk őket.

Eljött az ideje, hogy egy használható attribútumot készítsünk, méghozzá egy olyat, amellyel megjegyzéseket fűzhetünk egy osztályhoz vagy struktúrához. Az attribútum-osztály nagyon egyszerű lesz:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct,
                        AllowMultiple = false, Inherited = false)]
class DescriptionAttribute : System.Attribute
{
      public DescriptionAttribute(string description)
      {
            this.Description = description;
      }

      public string Description { get; set; }
}
```

Az attribútumok értékeihez pedig így férünk hozzá:

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct,
                        AllowMultiple = false, Inherited = false)]
class DescriptionAttribute : System.Attribute
{
      public DescriptionAttribute(string description)
      {
            this.Description = description;
      }

      public string Description { get; set; }
}

[Description("Ez egy osztály")]
class C { }

class Program
{
   static public void Main()
   {
            Attribute[] a = Attribute.GetCustomAttributes(typeof(C));

            foreach(Attribute attribute in a)
            {
                  if(attribute is DescriptionAttribute)
                  {
      Console.WriteLine(((DescriptionAttribute)attribute).Description);
                  }
            }
   }
}
```

29 Unsafe kód

A .NET platform legnagyobb eltérése a natív nyelvektől a memória kezelésében rejlik. A menedzselt kód nem enged közvetlen hozzáférést a memóriához, vagyis annyi a dolgunk, hogy megmondjuk, hogy szeretnénk egy ilyen és ilyen típusú objektumot, a rendszer elkészíti nekünk és kapunk hozzá egy referenciát, amelyen elérjük. Nem fogjuk tudni, hogy a memóriában hol van és nem is tudjuk áthelyezni. Épp ezért a menedzselt kód biztonságosabb, mint a natív, mivel a fentiek miatt egy egész sor hibalehetőség egész egyszerűen eltűnik. Nyílván ennek ára van, méghozzá a sebesség, de ezt behozzuk a memória gyorsabb elérésével/kezelésével ezért a két módszer között lényegében nincs teljesítménybeli különbség.
Vannak azonban helyzetek, amikor igenis fontos, hogy közvetlenül elérjük a memóriát:

- A lehető legjobb teljesítményt szeretnénk elérni egy rendkívül számításigényes feladathoz (pl.: számítógépes grafika).
- .NET–en kívüli osztálykönyvtárakat akarunk használni (pl.: Windows API hívások).

A C# a memória direkt elérését mutatókon (pointer) keresztül teszi lehetővé. Ahhoz, hogy mutatókat használhassunk az adott metódust, osztályt, adattagot vagy blokkot az *unsafe* kulcsszóval kell jelölnünk (ez az ún. unsafe context). Egy osztályon belül egy adattagot vagy metódust jelölhetünk *unsafe* módosítóval, de ez nem jelenti azt, hogy maga az osztály is *unsafe* lenne. Nézzük a következő példát:

```
using System;

class Test
{
      public unsafe int* x;
}

class Program
{
   static public void Main()
   {
            unsafe
            {
                  Test t = new Test();
                  int y = 10;
                  t.x = &y;
                  Console.WriteLine(*t.x);
            }
   }
}
```

Először deklaráltunk egy *unsafe* adattagot, méghozzá egy *int* típusra mutató pointert. A pointer típus az érték- és referenciatípusok mellett a harmadik típuskategória. A pointerek nem származnak a *System.Object*–ből és konverziós kapcsolat sincs közöttük (bár az egyszerű numerikus típusokról létezik explicit konverzió). Értelemszerűen boxing/unboxing sem alkalmazható rajtuk. Egy pointer mindig egy memóriacímet hordoz, amely memóriaterületen egy teljesen normális objektum van.

Ebből következően a fenti deklarációban nem adhatok azonnal értéket az unsafe pointernek, mivel numerikus értékadás esetén nem fordul le a program (hiszen nem egy *int* objektumról van szó), más objektum memóriacímét viszont nem tudom. Sebaj, erre való az ún. „címe–operátor" (&) amellyel átadhatom egy hagyományos objektum címét.
A programban ki akarjuk írni a memóriaterületen lévő objektum értékét, ezt a dereference operátorral (*) tehetjük meg. Ez visszaadja a mutatott értéket, míg ha magát a változót használjuk az „csak" a memóriacímet. A memóriacím a memória egy adott byte–jára mutat (vagyis a pointer növelése/csökkentése egy byte–al rakja odébb a mutatót), amely az adott objektum kezdőcíme. A pointer úgy tudja visszaadni az értékét, hogy tudja mekkora méretű az objektum (pl. egy *int* pointer egy 32 bites – 4 byte méretű – területet vesz majd elő).

A programot parancssorból a /unsafe kapcsolóval fordíthatjuk le, Visual Studio esetén jobb klikk a projecten, Properties és ott állítsuk be.

```
csc /unsafe main.cs
```

A megfelelő explicit konverzióval a memóriacímet is lekérhetjük:

```
using System;

class Program
{
   static public void Main()
   {
            unsafe
            {
                  int x = 10;
                  int* y = &x;

                  Console.WriteLine((int)y);
            }
   }
}
```

Pointer csakis a beépített numerikus típusokra (beleértve a *char* is), logikai típusokra, felsorolt típusokra, más pointerekre illetve minden olyan általunk készített struktúrára hivatkozhat, amely nem tartalmaz az eddig felsoroltakon kívül mást. Ezeket a típusokat összefoglaló néven unmanaged típusoknak nevezzük.

Explicit konverzió létezik bármely két pointer típus között, ezért fennállhat a veszélye, hogy ha A és B pointer nem ugyanakkora méretű területre mutat akkor az A–ról B–re való konverzió nem definiált működést okoz:

```
int x = 10;
byte y = 20;

int* p1 = &x; // ez jó
p1 = (int*)&y; // ez nem biztos, hogy jó
```

Implicit konverzió van viszont bármely pointer típusról a *void** univerzális pointer típusra. A *void**-on nem használható a dereference operátor:

```
using System;

class Program
{
   static public void Main()
   {
            unsafe
            {
                  int x = 10;
                  void* p1 = &x;

                  Console.WriteLine( *((int*)p1) );
            }
   }
}
```

Egy struktúrára is hivatkozhatunk pointerrel, ekkor a tagjait kétféleképpen érhetjük el: vagy a mutatón keresztül, vagy a nyíl (->) operátorral amely tulajdonképpen az előbbi rövidítése:

```
using System;

struct Test
{
      public int x;
}

class Program
{
   static public void Main()
   {
            unsafe
            {
                  Test t = new Test();
                  t.x = 10;
                  Test* p = &t;

                  Console.WriteLine((*p).x); // 10
                  Console.WriteLine(p->x); // 10
            }
   }
}
```

29.1 Fix objektumok

Normális esetben a szemétgyűjtő a memória töredezettség mentesítése érdekében mozgatja az objektumokat a memóriában. Egy pointer azonban mindig egy fix helyre mutat, hiszen a mutatott objektumnak a címét kértük le, ami pedig nem fog frissülni. Ez néhány esetben gondot okozhat (pl. amikor hosszabb ideig van a memóriában a mutatott adat, pl. valamilyen erőforrás). Ha szeretnénk, hogy az objektumok a helyükön maradjanak, akkor a *fixed* kulcsszót kell használnunk.
Mielőtt jobban megnéznénk a *fixed*–et vegyük szemügyre a következő forráskódot:

```
using System;

class Program
{
   static public void Main()
   {
            unsafe
            {
                  int[] array = new int[] { 1, 3, 4, 6, 7 };

                  int* p = &array;
            }
   }
}
```

Ez a program nem fordul le. Bár a tömbök referenciatípusok mégis kivételt képeznek, mivel használhatunk rajtuk pointert, igaz nem az eddig látott módon (valamint a tömb elemeinek unmanaged típusnak kell lenniük). Referenciatípuson belül deklarált értéktípusok esetén (ilyenek a tömbelemek is) fixálni kell az objektum helyzetét, hogy a GC ne mozdíthassa el. A következő kód már működni fog:

```
using System;

class Program
{
   static public void Main()
   {
            unsafe
            {
                  int[] array = new int[] { 1, 3, 4, 6, 7 };

                  fixed(int* p = array)

                  for(int i = 0;i < 5;++i)
                  {
                        Console.WriteLine(*(p + i));
                  }
            }
   }
}
```

29.2 Natív DLL kezelés

Előfordu, hogy olyan metódust akarunk hívni, amelyet egy natív (pl. C++ nyelven írt) DLL tartalmaz. A következő példában egy Windows API metódust hívunk meg, amely megjelenít egy MessageBox–ot. Ahhoz, hogy ezt meg tudjuk tenni elsősorban ismernünk kell az eredeti metódus szignatúráját, ehhez szükségünk lesz pl. a Win32 Programmers Reference című dokumentációra amelyet letölthetünk:

http://www.winasm.net/win32hlp.html

Keressük ki a *MessageBox* metódust, és a következőt fogjuk látni:

```
int MessageBox(
    HWND hWnd,    // handle of owner window
    LPCTSTR lpText,     // address of text in message box
    LPCTSTR lpCaption,  // address of title of message box
    UINT uType    // style of message box
   );
```

Ezen kívül még tudnunk kell azt is, hogy melyik DLL tárolja az adott függvényt, ez jelen esetben a user32.dll lesz.

Mindent tudunk, készítsük el a programot! A függvény importálásához a *DllImport* attribútumot fogjuk használni:

```
using System;
using System.Runtime.InteropServices;

public class Program
{
    static public void Main()
      {
        API.MessageBox(0, "Hello!", "Nincs", 0);
    }
}

public class API
{
    [DllImport("user32.dll")]
    public static extern int MessageBox(int hWnd,
        string text, string caption, uint type);
}
```

Az attribútumnak meg kell adnunk a dll nevét, valamint a metódus-deklarációt az *extern* kulcsszóval kell jelölnünk, hogy tudassuk a fordítóval, hogy a függvényt egy küldő forrásból kell előkerítenie.
A forráskódot a szokásos módon fordítjuk, és ha elindítjuk, akkor a következőt kell látnunk:

30 Többszálú alkalmazások

Egy Windows alapú operációs rendszerben minden futtatható állomány indításakor egy ún. process jön létre, amely teljes mértékben elkülönül az összes többitől. Egy process–t az azonosítója (PID – Process ID) alapján különböztetünk meg a többitől.
Minden egyes process rendelkezik egy ún. fő szállal (primary– vagy main thread), amely a belépési pontja a programnak (ld. *Main*).
Azokat az alkalmazásokat, amelyek csak fő szállal rendelkeznek thread-safe alkalmazásoknak nevezzük, mivel csak egy szál fér hozzá az összes erőforráshoz. Ugyanakkor ezek az alkalmazások hajlamosak „elaludni”, ha egy komplexebb feladatot hajtanak végre, hiszen a fő szál ekkor nem tud figyelni a felhasználó interakciójára.
Az ilyen helyzetek elkerülésére a Windows (és a .NET) lehetővé teszi másodlagos szálak (ún. worker thread) hozzáadását a fő szálhoz. Az egyes szálak (a process–ekhez hasonlóan) önállóan működnek a folyamaton belül és „versenyeznek” az erőforrások használatáért (concurrent access).
Jó példa lehet a szálkezelés bemutatására egy szövegszerkesztő használata: amíg kinyomtatunk egy dokumentumot (egy mellékszállal) az alkalmazás fő szála továbbra is figyeli a felhasználótól érkező utasításokat.
A többszálú programozás legnagyobb kihívása a szálak és feladataik megszervezése, az erőforrások elosztása.

Fontos megértenünk, hogy valójában a többszálúság a számítógép által nyújtott illúzió, hiszen a processzor egyszerre csak egy feladatot tud végrehajtani (bár terjednek a többmagos rendszerek, de gondoljunk bele, hogy amikor hozzá sem nyúlunk a számítógéphez is ötven – száz szál fut), így el kell osztania az egyes feladatok közt a processzoridőt (ezt a szálak prioritása alapján teszi) ez az ún. időosztásos (time slicing) rendszer. Amikor egy szálnak kiosztott idő lejár, akkor a futási adatait eltárolja az ún. Thread Local Storage–ben (ebből minden szálnak van egy) és átadja a helyet egy másik szálnak, amely – ha szükséges – betölti a saját adatait a TLS-ből és elvégzi a feladatát.

A .NET számos osztályt és metódust bocsát rendelkezésünkre, amelyekkel az egyes process–eket felügyelhetjük, ezek a *System.Diagnostics* névtérben vannak. Írjunk egy programot, amely kiírja az összes futó folyamatot és azonosítójukat:

```
using System;
using System.Diagnostics;

class Program
{
      static public void Main()
      {
            Process[] processList = Process.GetProcesses(".");

            foreach(Process process in processList)
            {
                  Console.WriteLine("PID: {0}, Process - név: {1}",
                        process.Id, process.ProcessName);
            }
      }
}
```

Amennyiben tudjuk a process azonosítóját, akkor használhatjuk a *Process.GetProcessById(azonosító)* metódust is.

A következő programunk az összes futó process minden szálát és azoknak adatait fogja kilistázni:

```
using System;
using System.Diagnostics;

class Program
{
      static public void Main()
      {
            Process[] processList = Process.GetProcesses(".");

            foreach(Process process in processList)
            {
                  Console.WriteLine("A folyamat ({0}) szálai",
                        process.ProcessName);

                  ProcessThreadCollection ptc = process.Threads;

                  foreach(ProcessThread thread in ptc)
                  {
                        Console.WriteLine("Id: {0}, Állapot: {1}",
                              thread.Id, thread.ThreadState);
                  }
            }
      }
}
```

Elég valószínű, hogy a program futásakor kivételt kapunk, hiszen a szálak listájába olyan szál is bekerülhet, amely a kiíráskor már befejezte futását (ez szinte mindig az Idle process esetében fordul elő, a meggondolás házi feladat, akárcsak a program kivétel-biztossá tétele).
A fenti osztályok segítségével remekül bele lehet látni a rendszer „lelkébe”, az MSDN–en megtaláljuk a fenti osztályok további metódusait, tulajdonságait amelyek az „utazáshoz” szükségesek.

A következő programunk a process–ek irányítását szemlélteti, indítsuk el az Internet Explorert, várjunk öt másodpercet és állítsuk le:

```
using System;
using System.Diagnostics;
using System.Threading;

class Program
{
      static public void Main()
      {
            Process explorer = Process.Start("iexplore.exe");
            Thread.Sleep(5000);
            explorer.Kill();
      }
}
```

Egyúttal felhasználtuk az első igazi szálkezeléshez tartozó metódusunkat is, a *Thread* osztály statikus *Sleep* metódusát (a *Thread* osztály a *System.Threading* névtérben található).

30.1 Application Domain -ek

Egy .NET program nem direkt módon processként fut, hanem be van ágyazva egy ún. application domain–be a processen belül (egy process több AD–t is tartalmazhat egymástól teljesen elszeparálva).
Ezzel a megoldással egyrészt elősegítik a platformfüggetlenséget, hiszen így csak az Application Domaint kell portolni egy másik platformra, a benne futó folyamatoknak nem kell ismerniük az operációs rendszert, másrészt biztosítja a programok stabilitását ugyanis ha egy alkalmazás összeomlik egy AD–ben, attól a többi még tökéletesen működik majd.
Amikor elindítunk egy .NET programot elsőként az alapértelmezett AD *(*default application domain) jön létre, ezután – ha szükséges – a CLR további AD–ket hoz létre.
A következő program kiírja az aktuális AD nevét:

```
using System;

class Program
{
	static public void Main()
	{
		AppDomain currAD = AppDomain.CurrentDomain;
		Console.WriteLine(currAD.FriendlyName);
	}
}
```

Az alkalmazás neve fog megjelenni, hiszen ő az alapértelmezett AD és egyelőre nincs is több.
Hozzunk létre egy második *AppDomaint*:

```
using System;

class Program
{
	static public void Main()
	{
		AppDomain secondAD = AppDomain.CreateDomain("second");
		Console.WriteLine(secondAD.FriendlyName);

		AppDomain.Unload(secondAD); // megszüntetjük
	}
}
```

30.2 Szálak

Elérkeztünk a fejezet eredeti tárgyához, már eleget tudunk ahhoz, hogy megértsük a többszálú alkalmazások elvét. Első programunkban lekérjük az adott programrész szálának az azonosítóját:

```
using System;
using System.Threading;

class Program
{
   static public void Main()
   {
            Console.WriteLine("Szál-Id: {0}",
                  Thread.CurrentThread.ManagedThreadId);
   }
}
```

A kimenetre minden esetben az egyes számot kapjuk, jelezvén, hogy az első, a „fő” szálban vagyunk.

A program utasításainak végrehajtása szerint megkülönböztetünk szinkron- és aszinkron működést. A fenti program szinkron módon működik, az utasításait egymás után hatja végre, ha esetleg egy hosszas algoritmusba ütközik, akkor csak akkor lép a következő utasításra, ha azt befejezte,
Az aszinkron végrehajtás ennek épp az ellentéte az egyes feladatokat el tudjuk küldeni egy másik szálba, a fő szál pedig fut tovább, amíg a mellékszál(ak) vissza nem térnek.

30.3 Aszinkron delegate-ek

A következőkben delegate–ek segítségével fogunk aszinkron programot írni. Minden egyes delegate rendelkezik azzal a képességgel, hogy aszinkron módon hívjuk meg, ezt a *BeginInvoke* és *EndInvoke* metódusokkal fogjuk megtenni.
Vegyük a következő delegate–et:

```
delegate int MyDelegate(int x);
```

Ez valójában így néz ki:

```
public sealed class MyDelegate : System.MulticastDelegate
{
   //...metódusok...

   public IAsyncResult BeginInvoke(int x, AsyncCallback cb, object state);

   public int EndInvoke(IAsyncResult result);
}
```

Egyelőre ne foglalkozzunk az ismeretlen dolgokkal, nézzük meg azt amit ismerünk. A *BeginInvoke*–val fogjuk meghívni a delegate-et, ennek első paramétere megegyezik a delegate paraméterével (vagy paramétereivel). Az *EndInvoke* fogja majd az eredményt szolgáltatni, ennek visszatérési értéke megegyezik a delegate–éval. Az *IAsyncResult* objektum, amit a *BeginInvoke* visszatérít segít elérni az eredményt és az *EndInvoke* is ezt kapja majd paraméteréül.
A *BeginInvoke* másik két paraméterével most nem foglalkozunk, készítsünk egy egyszerű metódust a delegate-hez és hívjuk meg aszinkron:

```
using System;
using System.Threading;

class Program
{
   public delegate int MyDelegate(int x);

   static int Square(int x)
   {
            Console.WriteLine("Szál-ID: {0}",
                  Thread.CurrentThread.ManagedThreadId);
            return (x * x);
   }

   static public void Main()
   {
            MyDelegate d = Square;

            Console.WriteLine("Szál-ID: {0}",
                  Thread.CurrentThread.ManagedThreadId);

            IAsyncResult iar = d.BeginInvoke(12, null, null);

            Console.WriteLine("BlaBla...");

            int result = d.EndInvoke(iar);

            Console.WriteLine(result);
   }
}
```

A kimenet a következő lesz:

```
Szál-ID: 1
BlaBla...
Szál-ID: 3
144
```

Látható, hogy egy új szál jött létre. Amit fontos megértenünk, hogy a *BeginInvoke* azonnal megkezdi a feladata végrehajtását, de az eredményhez csak az *EndInvoke* hívásakor jutunk hozzá, tehát külső szemlélőként úgy látjuk, hogy csak akkor fut le a metódus. A háttérben futó szál üzenete is látszólag csak az eredmény kiértékelésénél jelenik meg, az igazság azonban az, hogy a *Main* üzenete előbb ért a processzorhoz, ezt hamarosan látni fogjuk.

Többszálú program írásánál össze kell tudnunk hangolni a szálak munkavégzését, pl. ha az egyik szálban kiszámolt eredményre van szüksége egy másik, később indult szálnak. Ezt szinkronizálásnak nevezzük.
Szinkronizáljuk az eredeti programunkat, vagyis várjuk meg amíg a delegate befejezi a futását (természetesen a szinkronizálás ennél jóval bonyolultabb, erről a következő fejezetekben olvashatunk):

```
using System;
using System.Threading;

class Program
{
   public delegate int MyDelegate(int x);

   static int Square(int x)
   {
            Console.WriteLine("Szál-ID: {0}",
                  Thread.CurrentThread.ManagedThreadId);
            Thread.Sleep(100);
            return (x * x);
   }

   static public void Main()
   {
            MyDelegate d = Square;
            Console.WriteLine("Szál-ID: {0}",
                  Thread.CurrentThread.ManagedThreadId);

            IAsyncResult iar = d.BeginInvoke(12, null, null);

            while(!iar.IsCompleted)
            {
                  Console.WriteLine("BlaBla...");
            }

            int result = d.EndInvoke(iar);

            Console.WriteLine(result);
   }
}
```

Ezt a feladatot az *IAsyncResult* interfész *IsCompleted* tulajdonságával oldottuk meg. A kimenet:

```
Szál-ID: 1
BlaBla...
BlaBla...
BlaBla...
Szál-ID: 3
BlaBla...
BlaBla...
BlaBla...
BlaBla...
144
```

Itt már tisztán látszik, hogy az aszinkron metódus futása azonnal elkezdődött, igaz a *Main* itt is megelőzte.

A *Square* metódusban azért használtuk a *Sleep* metódust, hogy lássunk is valamit, ellenkező esetben túl gyorsan lefut ez az egyszerű program. Erősebb számítógépeken nem árt módosítani az alvás idejét akár 1000 ms–re is.

Valljuk be, elég macerás mindig meghívogatni az *EndInvoke*–ot, felmerülhet a kérdés, hogy nem lehetne valahogy automatizálni az egészet. Nos, épp ezt a gondot

oldja meg a *BeginInvoke* harmadik *AsyncCallback* típusú paramétere. Ez egy delegate, amely egy olyan metódusra mutathat, amelynek visszatérési értéke *void*, valamint egy darab *IAsyncResult* típusú paraméterrel rendelkezik. Ez a metódus azonnal le fog futni, ha a mellékszál elvégezte a feladatát:

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;

class Program
{
      public delegate int MyDelegate(int x);

      static int Square(int x)
      {
            Console.WriteLine("Szál-ID: {0}",
                  Thread.CurrentThread.ManagedThreadId);
            return (x * x);
      }

      static void AsyncMethodComplete(IAsyncResult iar)
      {
            Console.WriteLine("Aszinkron szál kész...");

            AsyncResult result = (AsyncResult)iar;
            MyDelegate d = (MyDelegate)result.AsyncDelegate;

            Console.WriteLine("Eredmény: {0}", d.EndInvoke(iar));
      }

      static public void Main()
      {
            MyDelegate d = Square;

            Console.WriteLine("Szál-ID {0}",
                  Thread.CurrentThread.ManagedThreadId);

            IAsyncResult iar = d.BeginInvoke(12,
                  new AsyncCallback(AsyncMethodComplete), null);

            Console.WriteLine("BlaBla...");
      }
}
```

Ha futtatjuk ezt a programot, akkor azt fogjuk látni, hogy nem írja ki az eredményt. Ez azért van, mert a „BlaBla” után a program futása megáll, mivel elérte a *Main* végét és nincs több utasítás, valamint ez gyorsabban történik, minthogy az aszinkron metódus kész lenne. Épp ezért érdemes egy *ReadKey* vagy egy *Sleep* metódust használni a program végén.

A kimenet a következő lesz:

```
Szál-ID 1
BlaBla...
Szál-ID: 3
Aszinkron szál kész...
Eredmény: 144
```

Egyetlen dolog van hátra, mégpedig a *BeginInvoke* utolsó paraméterének megismerése. Ez egy *object* típusú változó, azaz bármilyen objektumot átadhatunk. Ezt a paramétert használjuk, ha valamilyen plusz információt akarunk továbbítani. A *BeginInvoke* most így néz ki:

```
IAsyncResult iar = d.BeginInvoke(12,
      new AsyncCallback(AsyncMethodComplete), "Üzenet a jövõbõl :)");
```

Az üzenetet az *IAsyncResult AsyncState* tulajdonságával kérdezhetjük le:

```
static void AsyncMethodComplete(IAsyncResult iar)
{
      Console.WriteLine("Aszinkron szál kész...");

      AsyncResult result = (AsyncResult)iar;
      MyDelegate d = (MyDelegate)result.AsyncDelegate;

      Console.WriteLine("Üzenet: {0}", iar.AsyncState);
      Console.WriteLine("Eredmény: {0}", d.EndInvoke(iar));
}
```

30.3.1 Párhuzamos delegate hívás

A .NET 4.0 lehetővé teszi, hogy delegate-eket illetve önálló metódusokat is egyszerre, párhuzamosan hívjunk meg. Ehhez a feladathoz a korábban már megismert Task Parallel Library lesz a segítségünkre a *Parallel.Invoke* metódussal. Egyetlen szabály azért van, „hagyományos” delegate–et így nem hívhatunk, csakis a C# 3.0–tól használt *Action* megfelelő:

```
using System;
using System.Threading.Tasks;

class Program
{
      static public void Method()
      {
            Console.WriteLine("3");
      }

      static public void Main()
      {
            Action m = () => Console.WriteLine("1");
            m += () => Console.WriteLine("2");
            m += Program.Method;

            Parallel.Invoke(m, () => Console.WriteLine("4"));
      }
}
```

A *Parallel.Invoke Action* típusú elemeket tartalmazó tömböt vár paraméterként. Természetesen ez a metódus leginkább több processzormaggal ellátott számítógépen lesz igazán hatékony.

30.4 Szálak létrehozása

Ahhoz, hogy másodlagos szálakat hozzunk létre nem feltétlenül kell delegate–eket használnunk, mi magunk is elkészíthetjük őket. Vegyük a következő programot:

```
using System;
using System.Threading;

class Test
{
      public void ThreadInfo()
      {
            Console.WriteLine("Szál-név: {0}", Thread.CurrentThread.Name);
      }
}

class Program
{
      static public void Main()
      {
            Thread current = Thread.CurrentThread;
            current.Name = "Current-Thread";

            Test t = new Test();
            t.ThreadInfo();
      }
}
```

Elsőként lekértük és elneveztük az elsődleges szálat, hogy később azonosítani tudjuk, mivel alapértelmezetten nincs neve.

A következő programban a *Test* objektum metódusát egy háttérben futó szálból fogjuk meghívni:

```
using System;
using System.Threading;

class Test
{
      public void ThreadInfo()
      {
            Console.WriteLine("Szál-név: {0}", Thread.CurrentThread.Name);
      }
}

class Program
{
      static public void Main()
      {
            Test t = new Test();

            Thread backgroundThread = new Thread(
                  new ThreadStart(t.ThreadInfo));
            backgroundThread.Name = "Background-Thread";
            backgroundThread.Start();
      }
}
```

A *Thread* konsturktorában szereplő *ThreadStart* delegate-nek kell megadnunk azt a metódust amelyet a másodlagos szál majd futtat.
Ez eddig szép és jó, de mi van akkor, ha a meghívott metódusnak paraméterei is vannak? Ilyenkor a *ThreadStart* parametrizált változatát használhatjuk, ami igen eredeti módon a *ParameterizedThreadStart* névre hallgat. A *ThreadStart*–hoz hasonlóan ez is egy delegate, szintén *void* visszatérési típusa lesz, a paramétere pedig *object* típusú lehet:

```
using System;
using System.Threading;

class Test
{
      public void ThreadInfo(object parameter)
      {
            Console.WriteLine("Szál-név: {0}", Thread.CurrentThread.Name);

            if(parameter is string)
            {
                  Console.WriteLine("Paraméter: {0}", parameter);
            }
      }
}

class Program
{
      static public void Main()
      {
            Test t = new Test();

            Thread backgroundThread = new Thread(
                  new ParameterizedThreadStart(t.ThreadInfo));
            backgroundThread.Name = "Background-Thread";
            backgroundThread.Start("Hello");
      }
}
```

Nyílván a metódusban nem árt ellenőrizni a paraméter típusát mielőtt bármit csinálunk vele.

30.5 Foreground és background szálak

A .NET két különböző száltípust különböztet meg: amelyek előtérben, és amelyek a háttérben futnak. A kettő közti különbség a következő: a CLR addig nem állítja le az alkalmazást, amíg egy előtérbeli szál is dolgozik, ugyanez a háttérbeli szálakra nem vonatkozik (az aszinkron delegate esetében is ezért kellett a program végére a „lassítás”).
Logikus (lenne) a feltételezés, hogy az elsődleges és másodlagos szálak fogalma megegyezik jelen fejezetünk tárgyaival. Az igazság viszont az, hogy ez az állítás nem állja meg a helyét, ugyanis alapértelmezés szerint minden szál (a létrehozás módjától és idejétől függetlenül) előtérben fut. Természetesen van arra is lehetőség, hogy a háttérbe küldjük őket:

```
using System;
using System.Threading;

class Test
{
      public void ThreadInfo()
      {
            Thread.Sleep(5000);
            Console.WriteLine("Szál-név: {0}", Thread.CurrentThread.Name);
      }
}

class Program
{
      static public void Main()
      {
            Test t = new Test();

            Thread backgroundThread = new Thread(
                  new ThreadStart(t.ThreadInfo));
            backgroundThread.IsBackground = true;
            backgroundThread.Name = "Background-Thread";
            backgroundThread.Start();
      }
}
```

Ez a program semmit nem fog kiírni és pont ezt is vártuk tőle, mivel beállítottuk az *IsBackground* tulajdonságot, ezért az általunk készített szál „valódi” háttérben futó szál lett, vagyis a főszálnak nem kell megvárnia.

30.6 Szinkronizáció

A szálak szinkronizációjának egy primitívebb formáját már láttuk a delegate–ek esetében, most egy kicsit komolyabban közelítünk a témához.

Négyféleképpen szinkronizálhatjuk a szálainkat, ezek közül az első a blokkolás. Ennek már ismerjük egy módját, ez a *Thread.Sleep* metódus:

```
using System;
using System.Threading;

class Program
{
   static public void Main()
   {
         Console.WriteLine("Start...");
         Thread.Sleep(2000);
         Console.WriteLine("Stop...");
   }
}
```

Amikor egy szálat leblokkolunk az azonnal „elereszti” a processzort és addig inaktív marad, amíg a blokkolás feltételének a környezet eleget nem tesz, vagy a folyamat valamilyen módon megszakad.

A *Join* metódus addig várakoztatja a hívó szálat, amíg az a szál, amin meghívták nem végezte el a feladatát:

```
using System;
using System.Threading;

class Program
{
   static public void Main()
   {
         Thread t = new Thread(delegate() { Thread.Sleep(2000); } );
         t.Start();
         t.Join();
   }
}
```

A *Join*–nak megadhatunk egy „timeout” paramétert (ezredmásodpercben), amely idő lejárta után – ha a szál nem végzett feladatával – hamis értékkel tér vissza. A következő példa (ezúttal lambda kifejezéssel) ezt mutatja meg:

```
using System;
using System.Threading;

class Program
{
      static public void Main()
      {
            Thread t = new Thread(() => Thread.Sleep(2000));
            t.Start();

            if(t.Join(1000) == false)
            {
                  Console.WriteLine("Az idő lejárt...");
                  t.Abort(); // megszakítjuk a szál futását
            }
      }
}
```

A következő szinkronizációs módszer a lezárás (locking). Ez azt jelenti, hogy erőforrásokhoz, illetve a program bizonyos részeihez egyszerre csak egy szálnak engedünk hozzáférést. Ha egy szál hozzá akar férni az adott dologhoz, amelyet egy másik szál már használ, akkor automatikusan blokkolódik, és várólistára kerül, ahonnan érkezési sorrendben lehet hozzájutni az erőforráshoz (ha az előző szál már végzett).

Nézzük a következő példát:

```
using System;
using System.Threading;

class Test
{
      static int x = 10;
      static int y = 20;

      static public void Divide()
      {
            if(Test.x != 0)
            {
```

```
                    Thread.Sleep(2);
                    Console.WriteLine(Test.y / Test.x);
                    Test.x = 0;
                }
        }
}

class Program
{
        static public void Main()
        {
                Thread t1 = new Thread(new ThreadStart(Test.Divide));
                Thread t2 = new Thread(new ThreadStart(Test.Divide));
                t1.Start();
                t2.Start();
        }
}
```

Tegyük fel, hogy megérkezik egy szál, eljut odáig, hogy kiírja az eredményt, és épp ekkor érkezik egy másik szál is. Megvizsgálja a feltételt, rendben találja és továbblép. Ebben a pillanatban azonban az elsőként érkezett szál lenullázza a változót, és amikor a második szál osztani akar, akkor kap egy szép kis kivételt. A *Divide* metódus feltételében nem véletlenül van ott a *Sleep*, ezzel teszünk róla, hogy tényleg legyen kivétel, mivel ez egy egyszerű program muszáj lelassítani egy kicsit az első szálat (érdemes többször lefuttatni, nem biztos, hogy azonnal kivételt kapunk).
A műveletet lezárhatjuk a következő módon:

```
static object locker = new object();

static public void Divide()
{
        lock(locker)
        {
                if(Test.x != 0)
                {
                        Thread.Sleep(2);
                        Console.WriteLine(Test.y / Test.x);
                        Test.x = 0;
                }
        }
}
```

A *lock* kijelöl egy blokkot, amelyhez egyszerre csak egy szál fér hozzá. Ahhoz azonban, hogy ezt megtehessük ki jelölnünk egy ún. tokent, amelyet lezárhat. A tokennek minden esetben referenciatípusnak kell lennie.

A *lock* helyett bizonyos esetekben egy lehetséges megoldás *volatile* kulcsszóval jelölt mezők használata. A jegyzet ezt a témát nem tárgyalja, mivel a megértéséhez tisztában kell lennünk a fordító sajátosságaival, a következő – angol nyelvű – weboldalon bővebb információt talál a kedves olvasó: http://igoro.com/archive/volatile-keyword-in-c-memory-model-explained/

Természetesen nem csak statikus metódusokból áll a világ, egy „normális” metódust is lezárhatunk, de ilyen esetekben az egész objektumra kell vonatkoztatni a zárolást, hogy ne változzon meg az állapota egy másik szál keze nyomán:

```
class Test
{
	int x = 10;
	int y = 20;

	public void Divide()
	{
		lock(this)
		{
			if(x != 0)
			{
				Thread.Sleep(2);
				Console.WriteLine(y / x);
				x = 0;
			}
		}
	}
}
```

A *lock*–hoz hasonlóan működik a *Mutex* is, a legnagyobb különbség az a kettő közt, hogy utóbbi processz szinten zárol, azaz a számítógépen futó összes folyamat elöl elzárja a használat lehetőségét. Az erőforrás/metódus/stb. használata előtt meg kell hívnunk a *WaitOne* metódust, a használat után pedig el kell engednünk az erőforrást a *ReleaseMutex* metódussal (ha ezt nem tesszük meg a kódból, akkor az alkalmazás futásának végén a CLR automatikusan megteszi helyettünk). A következő példában létrehozunk több szálat és versenyeztetjük őket egy metódus használatáért. Elsőként készítsük el az osztályt, amely tárolja az „erőforrást” és a *Mutex* objektumot:

```
class Test
{
	private Mutex mutex = new Mutex();

	public void ResourceMetod()
	{
		mutex.WaitOne();

		Console.WriteLine("{0} használja az erõforrást...",
			Thread.CurrentThread.Name);

		Thread.Sleep(1000);

		mutex.ReleaseMutex();

		Console.WriteLine("{0} elengedi az erõforrást...",
			Thread.CurrentThread.Name);
	}
}
```

Most pedig jöjjön a főprogram:

```
class Program
{
	static Test t = new Test();

	static public void ResourceUserMethod()
	{
		for(int i = 0;i < 10;++i)
		{
			t.ResourceMetod();
		}
	}
```

```
        static public void Main()
        {
                List<Thread> threadList = new List<Thread>();

                for(int i = 0;i < 10;++i)
                {
                        threadList.Add(
                        new Thread(new ThreadStart(Program.ResourceUserMethod))
                                {
                                        Name = "Thread" + i.ToString()
                                });
                }

                threadList.ForEach((thread) => thread.Start());
        }
}
```

A *Semaphore* hasonlít a *lock*–ra és a *Mutex*–re, azzal a különbséggel, hogy megadhatunk egy számot, amely meghatározza, hogy egy erőforráshoz maximum hány szál férhet hozzá egy időben. A következő program az előző átirata, egy időben maximum három szál férhet hozzá a metódushoz:

```
class Test
{
        private Semaphore semaphore = new Semaphore(3, 3);

        public void ResourceMetod()
        {
                semaphore.WaitOne();

                Console.WriteLine("{0} használja az erõforrást...",
                        Thread.CurrentThread.Name);

                Thread.Sleep(1000);

                semaphore.Release();

                Console.WriteLine("{0} elengedi az erõforrást...",
                        Thread.CurrentThread.Name);
        }
}
```

A főprogram pedig ugyanaz lesz.

30.7 ThreadPool

Képzeljük el a következő szituációt: egy kliens-szerver alkalmazást készítünk, a szerver a főszálban figyeli a bejövő kapcsolatokat, és ha kliens érkezik, akkor készít neki egy szálat majd a háttérben kiszolgálja. Tegyük még hozzá azt is, hogy a kliensek viszonylag rövid ideig tartják a kapcsolatot a szerverrel, viszont sokan vannak.
Ha úgy készítjük el a programot, hogy a bejövő kapcsolatoknak mindig új szálat készítünk, akkor nagyon gyorsan teljesítményproblémákba fogunk ütközni:

- Egy objektum létrehozása költséges.
- Ha már nem használjuk, akkor a memóriában marad, amíg el nem takarítja a GC.
- Mivel sok bejövő kapcsolatunk van, ezért hamarabb lesz tele a memória szeméttel, vagyis gyakrabban fut majd a GC.

A probléma gyökere az egyes pont, vagyis az, hogy minden egyes kapcsolatnak új szálat készítünk, majd eldobjuk azt. Sokkal hatékonyabbak lehetnénk, ha megpróbálnánk valahogy újrahasznosítani a szálakat. Ezt a módszert thread-pooling–nak nevezzük és szerencsénk van, mivel a .NET beépített megoldással rendelkezik (ugyanakkor ha igazán hatékonyak akarunk lenni, írhatunk saját, az adott követelményeknek legjobban megfelelő *ThreadPool* osztályt is).
Egy későbbi fejezetben elkészítünk majd egy, a fenti felvázolt helyzethez hasonló programot, most „csak” egy egyszerű példán keresztül fogjuk megvizsgálni ezt a technikát. Nézzük meg a következő programot:

```
using System;
using System.Threading;

class Program
{
      static public void Do(object inObj)
      {
            Console.WriteLine("A következõ adatot használjuk: {0}",
                  (int)inObj);
            Thread.Sleep(500);
      }

      static public void Main()
      {
            ThreadPool.SetMaxThreads(5, 0);

            for(int i = 0;i < 20;++i)
            {
                  ThreadPool.QueueUserWorkItem(
                        new WaitCallback(Program.Do),
                        i);
            }

            Console.ReadKey();
      }
}
```

Ami elsősorban feltűnhet, az az, hogy a *ThreadPool* egy statikus osztály,vagyis nem tudjuk példányosítani ehelyett a metódusait használhatjuk. A *SetMaxThread* metódus a maximálisan memóriában tartott szálak számát állítja be, az első paraméter a „rendes” a második az aszinkron szálak számát jelzi (utóbbira most nincs szükség ezért kapott nulla értéket).
A *QueueUserWorkItem* metódus lesz a *ThreadPool* lelke, itt indítjuk útjára az egyes szálakat. Ha egy „feladat” bekerül a listára, de nincs az adott pillanatban szabad szál, akkor addig vár, amíg nem kap egyet. A metódus első paramétere egy delegate, amely olyan metódusra mutathat, amelynek visszatérési értéke nincs (*void*) és egyetlen *object* típusú paraméterrel rendelkezik. Ezt a paramétert adjuk meg a második paraméterben.

Fontos tudni, hogy a *ThreadPool* osztály csakis background szálakat indít, vagyis a program nem fog várni amíg minden szál végez hanem kilép. Ennek megakadályozására tettünk a program végére egy *Console.ReadKey* parancsot, így látni is fogjuk, hogy mi történik éppen (erre pl. a fent említett kliens-szerver példában nincs szükség, mivel a szerver a főszálban végtelen ciklusban várja a bejövő kapcsolatokat).

31 Reflection

A „reflection” fogalmát olyan programozási technikára alkalmazzuk, ahol a program (futás közben) képes megváltoztatni saját struktúráját és viselkedését. Az erre a paradigmára épülő programozást reflektív programozásnak nevezzük.
Ebben a fejezetben csak egy rövid példát találhat a kedves olvasó, a Reflection témaköre óriási és a mindennapi programozási feladatok végzése közben viszonylag ritkán szorulunk a használatára (ugyanakkor bizonyos esetekben nagy hatékonysággal járhat a használata).
Vegyük a következő példát:

```
using System;

class Test
{

}

class Program
{
   static public void Main()
   {
            Test t = new Test();
            Type type = t.GetType();
            Console.WriteLine(type); // Test
   }
}
```

Futásidőben lekértük az objektum típusát (a *GetType* metódust minden osztály örökli az *object*–től), persze a reflektív programozás ennél többről szól, lépjünk előre egyet: mi lenne, ha az objektumot úgy készítenénk el, hogy egyáltalán nem írjuk le a *new* operátort:

```
using System;
using System.Reflection; // ez kell

class Test
{

}

class Program
{
   static public void Main()
   {
            Type type = Assembly.GetCallingAssembly().GetType("Test");
            var t = Activator.CreateInstance(type);
            Console.WriteLine(t.GetType()); // Test
   }
}
```

Megjegyzendő, hogy fordítási időben semmilyen ellenőrzés sem történik a példányosítandó osztályra nézve, így ha elgépeltünk valamit az bizony kivételt fog dobni futáskor (*System.ArgumentNullException*).

A következő példában úgy fogunk meghívni egy példánymetódust, hogy nem példányosítottunk hozzá tartozó osztályt:

```
using System;
using System.Reflection; // ez kell

class Test
{
      public void Method()
      {
            Console.WriteLine("Hello!");
      }
}

class Program
{
   static public void Main()
   {
            Type type = Assembly.GetCallingAssembly().GetType("Test");
            type.InvokeMember("Method", BindingFlags.InvokeMethod,
                              null, Activator.CreateInstance(type), null);
   }
}
```

A felhasznált *InvokeMember* metódus első paramétere annak a konstruktornak/metódusnak/tulajdonságnak/… a neve amelyet meghívunk. A második paraméterrel jelezzük, hogy mit és hogyan fogunk hívni (a fenti példában egy metódust). Következő a sorban a binder paraméter, ezzel beállíthatjuk, hogy az öröklött/túlterhelt tagokat hogyan hívjuk meg. Ezután maga a típus, amin a hívást elvégezzük, végül a hívás paraméterei (ha vannak)).

32 Állománykezelés

Ebben a fejezetben megtanulunk fileokat írni/olvasni és a könyvtárstruktúra állapotának lekérdezését illetve módosítását.

32.1 Olvasás/írás fileból/fileba

A .NET számos osztályt biztosít számunkra, amelyekkel fileokat tudunk kezelni, ebben a fejezetben a leggyakrabban használtakkal ismerkedünk meg. Kezdjük egy file megnyitásával és tartalmának a képernyőre írásával. Legyen pl. a szöveges file tartalma a következő:

```
alma
körte
dió
csákány
pénz
könyv
```

A program pedig:

```
using System;
using System.IO;

class Program
{
   static public void Main()
   {
         FileStream fs = new FileStream("Test.txt", FileMode.Open);
         StreamReader sr = new StreamReader(fs);

         string s = sr.ReadLine();
         while(s != null)
         {
               Console.WriteLine(s);
               s = sr.ReadLine();
         }

         sr.Close();
         fs.Close();
   }
}
```

Az IO osztályok a *System.IO* névtérben vannak. A C# ún. stream–eket, adatfolyamokat használ az IO műveletek végzéséhez. Az első sorban megnyitottunk egy ilyen folyamot és azt is megmondtuk, hogy mit akarunk tenni vele.
A *FileStream* konstruktorának első paramétere a file neve (ha nem talál ilyen nevű filet, akkor kivételt dob a program). Amennyiben nem adunk meg teljes elérési utat, akkor automatikusan a saját könyvtárában fog keresni a program. Ha külső könyvtárból szeretnénk megnyitni az állományt, akkor azt a következőképpen tehetjük meg:

```
FileStream fs = new FileStream("C:\\Dir1\\Dir2\\test.txt", FileMode.Open);
```

Azért használunk dupla backslash–t (\), mert ez egy ún. escape karakter, önmagában nem lehetne használni (persze ez nem azt jelenti, hogy minden ilyen karaktert kettőzve kellene írni, minden speciális karakter előtt a backslash–t kell használnunk).

Egy másik lehetőség, hogy az „at” jelet (@) használjuk az elérési út előtt, ekkor nincs szükség dupla karakterekre, mivel mindent normális karakterként fog értelmezni:

```
FileStream fs = new FileStream(@'C:\Dir1\Dir2\test.txt', FileMode.Open);
```

A *FileMode* enum–nak a következő értékei lehetnek:

Create	Létrehoz egy új filet, ha már létezik a tartalmát kitörli
CreateNew	Ugyanaz mint az előző, de ha már létezik a file akkor kivételt dob.
Open	Megnyit egy filet, ha nem létezik kivételt dob.
OpenOrCreate	Ugyanaz mint az előző, de ha nem létezik akkor létrehozza a filet.
Append	Megnyit egy filet, és automatikusan a végére pozícionál. Ha nem létezik, létrehozza.
Truncate	Megnyit egy létező filet és törli a tartalmát. Ebben a módban a file tartalmát nem lehet olvasni (egyébként kivételt dob).

A *FileStream* konstruktorának további két paramétere is lehet, amelyek érdekesek számunkra (tulajdonképpen 15 különböző konstruktora van), mindkettő felsorolt típus. Az első a *FileAccess*, amellyel beállítjuk, hogy pontosan mit akarunk csinálni az állománnyal:

Read	Olvasásra nyitja meg.
Write	Írásra nyitja meg.
ReadWrite	Olvasásra és írásra nyitja meg

A fenti példát így is írhattuk volna:

```
FileStream fs = new FileStream("test.txt", FileMode.Open, FileAccess.Read);
```

Végül a *FileShare*–rel azt állítjuk be, ahogy más folyamatok férnek hozzá a filehoz:

None	Más folyamat nem férhet hozzá a filehoz, amíg azt be nem zárjuk.
Read	Más folyamat olvashatja a filet.
Write	Más folyamat írhatja a filet.
ReadWrite	Más folyamat írhatja és olvashatja is a

	filet.
Delete	Más folyamat törölhet a fileból (de nem magát a filet).
Inheritable	A gyermek processzek is hozzáférhetnek a filehoz.

Ha a fenti programot futtatjuk, akkor előfordulhat, hogy az ékezetes karakterek helyett kérdőjel jelenik meg. Ez azért van, mert az éppen aktuális karaktertábla nem tartalmazza ezeket a karaktereket, ez tipikusan nem magyar nyelvű operációs rendszer esetén fordul elő. A megoldás, hogy kézzel állítjuk be a megfelelő táblát, ezt a *StreamReader* konstruktorában tehetjük meg (a tábla pedig iso-8859-2 néven szerepel):

```
StreamReader rs = new StreamReader(fs, Encoding.GetEncoding("iso-8859-2"),
false);
```

Ehhez szükség lesz még a *System.Text* névtérre is.

Most írjunk is a fileba. Erre a feladatra a *StreamReader* helyett a *StreamWriter* osztályt fogjuk használni:

```
using System;
using System.IO;

class Program
{
   static public void Main()
   {
         FileStream fs = new FileStream("Test.txt", FileMode.Open,
            FileAccess.Write, FileShare.None);
         StreamWriter sw = new StreamWriter(fs);

         Random r = new Random();
         for(int i = 0;i < 10;++i)
         {
               sw.Write(r.Next());
               sw.Write(Environment.NewLine);
         }

         sw.Close();
         fs.Close();
   }
}
```

Házi feladat következik: módosítsuk a programot, hogy ne dobja el az eredeti tartalmát a filenak, és az írás után azonnal írjuk ki a képernyőre az új tartalmát.

Az *Environment.NewLine* egy újsor karaktert ad vissza.

Bináris fileok kezeléséhez a *BinaryReader/BinaryWriter* osztályokat használhatjuk:

```
using System;
using System.IO;

class Program
{
   static public void Main()
   {
         BinaryWriter bw = new BinaryWriter(File.Create("file.bin"));

         for(int i = 0;i < 100;++i)
         {
                  bw.Write(i);
         }

         bw.Close();

         BinaryReader br = new BinaryReader(
            File.Open("file.bin", FileMode.Open));

         while(br.PeekChar() != -1)
         {
                  Console.WriteLine(br.ReadInt32());
         }

         br.Close();
   }
}
```

Készítettünk egy bináris filet, és beleírtuk a számokat egytől százig. Ezután megnyitottuk és elkezdtük kiolvasni a tartalmát. A *PeekChar* metódus a soron következő karaktert (byte–ot) adja vissza, illetve -1–et, ha elértük a file végét. A folyambeli aktuális pozíciót nem változtatja meg.
A cikluson belül van a trükkös rész. A *ReadTípusnév* metódus a megfelelő típusú adatot adja vissza, de vigyázni kell vele, mert ha nem megfelelő méretű a beolvasandó adat, akkor hibázni fog. A fenti példában, ha a *ReadString* metódust használtuk volna, akkor kivétel (*EndOfStreamException*) keletkezik, mivel a kettő nem ugyanakkora mennyiségű adatot olvas be. Az egész számokat kezelő metódus nyílván működni fog, hiszen tudjuk, hogy számokat írtunk ki.

Eddig kézzel zártuk le a stream-eket, de ez nem olyan biztonságos, mivel gyakran elfelejtkezik róla az ember. Használhatjuk ehelyett a *using* blokkokat, amelyek ezt automatikusan megteszik. Fent például írhatnánk ezt is:

```
using(BinaryWriter bw = new BinaryWriter(File.Create("file.bin")))
{
      for(int i = 0;i < 100;++i)
      {
            bw.Write(i);
      }
}
```

32.2 Könyvtárstruktúra kezelése

A fileok mellett a .NET a könyvtárstruktúra kezelését is széleskörűen támogatja. A *System.IO* névtér ebből a szempontból két részre oszlik: információs- és operációs

eszközökre. Előbbiek (ahogyan a nevük is sugallja) információt szolgáltatnak, míg az utóbbiak (többségükben statikus metódusok) bizonyos műveleteket (új könyvtár létrehozása, törlése, stb.) végeznek a filerendszeren. Első példánkban írjuk ki mondjuk a C meghajtó gyökerének könyvtárait:

```
using System;
using System.IO;

class Program
{
   static public void Main()
   {
         foreach(string s in Directory.GetDirectories("C:\\"))
         {
                  Console.WriteLine(s);
         }
   }
}
```

Természetesen nem csak a könyvtárakra, de a fileokra is kíváncsiak lehetünk. A programunk módosított változata némi plusz információval együtt ezeket is kiírja nekünk:

```
using System;
using System.IO;

class Program
{
      static public void PrintFileSystemInfo(FileSystemInfo fsi)
      {
         if((fsi.Attributes & FileAttributes.Directory) != 0)
         {
               DirectoryInfo di = fsi as DirectoryInfo;
               Console.WriteLine("Könyvtár: {0}, Készült: {1}",
                  di.FullName, di.CreationTime);
         }
         else
         {
               FileInfo fi = fsi as FileInfo;
               Console.WriteLine("File: {0}, készült: {1}",
                  fi.FullName, fi.CreationTime);
         }
      }

   static public void Main()
   {
         foreach(string s in Directory.GetDirectories("C:\\"))
         {
                  PrintFileSystemInfo(new DirectoryInfo(s));
         }

         foreach(string s in Directory.GetFiles("C:\\"))
         {
                  PrintFileSystemInfo(new FileInfo(s));
         }
   }
}
```

Először a mappákon, majd a fileokon megyünk végig. Ugyanazzal a metódussal írjuk ki az információkat kihasználva azt, hogy a *DirectoryInfo* és a *FileInfo* is egy közös őstől a *FileSystemInfo*–ból származik (mindkettő konstruktora a vizsgált alany elérési

útját várja paraméterként), így a metódusban csak meg kell vizsgálni, hogy éppen melyikkel van dolgunk és átkonvertálni a megfelelő típusra. A vizsgálatnál egy „bitenkénti és” műveletet hajtottunk végre, hogy ez miért és hogyan működik, annak meggondolása az olvasó feladata.

Eddig csak információkat kértünk le, most megtanuljuk módosítani is a könyvtárstruktúrát:

```
using System;
using System.IO;

class Program
{
   static public void Main()
   {
         string dirPath = "C:\\test";
         string filePath = dirPath + "\\file.txt";

         // ha nem létezik a könyvtár
         if(Directory.Exists(dirPath) == false)
         {
               // akkor elkészítjük
               Directory.CreateDirectory(dirPath);
         }

         FileInfo fi = new FileInfo(filePath);

         // ha nem létezik a file
         if(fi.Exists == false)
         {
               // akkor elkészítjük és írunk bele
               StreamWriter sw = fi.CreateText();
               sw.WriteLine("Dio");
               sw.WriteLine("Alma");
               sw.Close();
         }
   }
}
```

A *FileInfo CreateText* metódusa egy *StreamWriter* objektumot ad vissza amellyel írhatunk egy szöveges fileba.

32.3 In–memory streamek

A .NET a *MemoryStream* osztályt biztosítja számunkra, amellyel memóriabeli adatfolyamokat írhatunk/olvashatunk. Mire is jók ezek a folyamok? Gyakran van szükségünk arra, hogy összegyűjtsünk nagy mennyiségű adatot, amelyeket majd a folyamat végén ki akarunk írni a merevlemezre. Nyílván egy tömb vagy egy lista nem nyújtja a megfelelő szolgáltatásokat, előbbi rugalmatlan, utóbbi memóriaigényes, ezért a legegyszerűbb, ha közvetlenül a memóriában tároljuk el az adatokat. A *MemoryStream* osztály jelentősen megkönnyíti a dolgunkat, mivel lehetőséget ad közvetlenül fileba írni a tartalmát. A következő program erre mutat példát:

```
using System;
using System.IO;

class Program
{
   static public void Main()
   {
         MemoryStream mstream = new MemoryStream();
         StreamWriter sw = new StreamWriter(mstream);

         for(int i = 0;i < 1000;++i)
         {
                  sw.WriteLine(i);
         }

         sw.Flush();

         FileStream fs = File.Create("inmem.txt");
         mstream.WriteTo(fs);

         sw.Close();
         fs.Close();
         mstream.Close();
   }
}
```

A *MemoryStream*–re „ráállítottunk” egy *StreamWriter* objektumot. Miután minden adatot a memóriába írtunk a *Flush* metódussal (amely egyúttal kiüríti a *StreamWriter*–t is) létrehoztunk egy filet és a *MemoryStream WriteTo* metódusával kiírtuk belé az adatokat.

32.4 XML

Az XML általános célú leírónyelv, amelynek elsődleges célja strukturált szöveg és információ megosztása az interneten keresztül. Lássunk egy példát:

```
<?xml version="1.0" encoding="UTF-8" ?>
<list>
      <item>1</item>
      <item>2</item>
      <item>3</item>
</list>
```

Az első sor megmondja, hogy melyik verziót és milyen kódolással akarjuk használni, ezután következnek az adatok. Minden XML dokumentum egyetlen gyökérelemmel rendelkezik (ez a fenti példában a <list>), amelynek minden más elem a gyermekeleme (egyúttal minden nem-gyökérelem egy másik elem gyermeke kell legyen, ez nem feltétlenül jelenti a gyökérelemet). Minden elemnek rendelkeznie kell nyitó (<list>) és záró (</list>) tagekkel. Üres elemeknél ezeket egyszerre is deklarálhatjuk (<üres />). Az egyes elemek tárolhatnak attribútumokat a következő formában:

```
<item value="10" />
<item value="10"></item>
```

Itt mindkét forma legális.

A .NET Framework erősen épít az XML–re, mind offline (konfigurációs fileok, in-memory adatszerkezetek) mind online (SOAP alapú információcsere, etc...) téren.

A számunkra szükséges osztályok a *System.Xml* névtérben vannak, a két legalapvetőbb ilyen osztály az *XmlReader* és az *XmlWriter*. Ezeknek az absztrakt osztályoknak a segítségével hajthatóak végre a szokásos – irás/olvasás – műveletek.

Először nézzük meg, hogyan tudunk beolvasni egy XML filet. A megnyitáshoz az *XmlReader* egy statikus metódusát a *Create*–et fogjuk használni, ez egy streamtől kezdve egy szimpla filenévig mindent elfogad:

```
XmlReader reader = XmlReader.Create("test.xml");
```

Függetlenül attól, hogy az *XmlReader* egy absztrakt osztály tudjuk példányosítani, mégpedig azért, mert ilyenkor valójában egy *XmlTextReaderImpl* típusú objektum jön létre, amely az *XmlReader* egy belső, *internal* elérésű leszármazottja (tehát közvetlenül nem tudnánk példányosítani).

Miután tehát megnyitottuk a filet, végig tudunk iterálni rajta:

```
while (reader.Read())
{
      switch (reader.NodeType)
      {
            case XmlNodeType.Element:
                  Console.WriteLine("<{0}>", reader.Name);
                  break;
            case XmlNodeType.EndElement:
                  Console.WriteLine("</{0}>", reader.Name);
                  break;
            case XmlNodeType.Text:
                  Console.WriteLine(reader.Value);
                  break;
            default:
                  break;
      };
}
```

Az *XmlReader NodeType* tulajdonsága egy *XmlNodeType* felsorolás egy tagját adja vissza, a példában a legalapvetőbb típusokat vizsgáltuk és ezek függvényében írtuk ki a file tartalmát.

A következő példában használni fogjuk az *XmlWriter* osztály egy leszármazottját, az *XmlTextWriter*–t, ugyanis a filet kódból fogjuk létrehozni:

```
XmlTextWriter writer = new XmlTextWriter("newxml.xml", Encoding.UTF8);
writer.Formatting = Formatting.Indented;
```

A *Formatting* tulajdonság, az Xml fileoktól megszokott hierarchikus szerkezetet fogja megteremteni (ennek az értékét is beállíthatjuk).

```
writer.WriteStartDocument();
writer.WriteComment(DateTime.Now.ToString());
```

Elkezdtük az adatok feltöltését, és beszúrtunk egy kommentet is. A file tartalma most a következő:

```
<?xml version="1.0" encoding="utf-8"?>
<!--2010.08.20. 12:04:21-->
```

A file személyek adatait fogja tartalmazni:

```
writer.WriteStartElement("PersonsList");
writer.WriteStartElement("Person");
writer.WriteElementString("Name", "Reiter Istvan");
writer.WriteElementString("Age", "22");
writer.WriteEndElement();
```

A *StartElement* egy új „tagcsoportot” kezd a paramétereként megadott névvel, a *WriteElementString* pedig feltölti azt értékekkel, kulcs-érték párokkal.
A file:

```
<?xml version="1.0" encoding="utf-8"?>
<!--2008.10.17. 20:02:05-->
<PersonsList>
  <Person>
    <Name>Reiter Istvan</Name>
    <Age>24</Age>
  </Person>
</PersonsList>
```

Az egyes tagekhez attribútumokat is rendelhetünk:

```
writer.WriteAttributeString("Note", "List of persons");
```

Első paraméter az attribútum neve, utána pedig a hozzá rendelt érték jön. Ekkor a file így alakul:

```
<?xml version="1.0" encoding="utf-8"?>
<!--2008.10.17. 20:05:24-->
<PersonsList Note="List of persons">
  <Person>
    <Name>Reiter Istvan</Name>
    <Age>22</Age>
  </Person>
</PersonsList>
```

Az *XmlReader* rendelkezik néhány *MoveTo* előtaggal rendelkező metódussal, ezekkel a file egy olyan pontjára tudunk navigálni, amely megfelel egy feltételnek. Néhány példa:

```
bool l = reader.MoveToAttribute("Note");
```

Ez a metódus a paramétereként megadott attribútumra állítja a reader–t, és igaz értékkel tér vissza, ha létezik az attribútum. Ellenkező esetben a pozíció nem változik, és a visszatérési érték hamis lesz.

```
reader.MoveToContent();
```

Ez a metódus a legközelebbi olyan csomópontra ugrik, amely tartalmaz adatot is, ekkor az *XmlReader* (vagy leszármazottainak) *Value* tulajdonsága ezt az értéket fogja tartalmazni.

A *MoveToElement* metódus visszalép arra a csomópontra, amely azt az attribútumot tartalmazza, amelyen a reader áll (értelemszerűen következik, hogy az előző kettő metódust gyakran használjuk együtt):

```
while (reader.Read())
{
    if (reader.HasAttributes)
    {
        for (int i = 0; i < reader.AttributeCount; ++i)
        {
            reader.MoveToAttribute(i);
                  Console.WriteLine("{0} {1}", reader.Name, reader.Value);
        }

        reader.MoveToElement();
    }
}
```

A *HasAttribute* metódussal megtudakoljuk, hogy van–e a csomóponton attribútum, utána pedig az *AttributeCount* tulajdonság segítségével (amely azoknak a számát adja vissza) végigiterálunk rajtuk. Miután végeztünk, visszatérünk a kiindulási pontra (hogy a *Read* metódus tudja végezni a dolgát).

Említést érdemelnek még a *MoveToFirstAttribute* és a *MovetoNextAttribute* metódusok, amelyek nevükhöz méltóan az első illetve a „következő” attribútumra pozícionálnak. Módosítsuk egy kicsit az előző példát:

```
for (int i = 0; i < reader.AttributeCount; ++i)
{
    //reader.MoveToAttribute(i);
    reader.MoveToNextAttribute();
    Console.WriteLine("{0} {1}", reader.Name, reader.Value);
}
```

Végül, de nem utolsósorban a *Skip* metódus maradt, amely átugorja egy csomópont gyermekeit és a következő azonos szinten lévő csomópontra ugrik.

32.5 XML DOM

Eddig az XML állományokat hagyományos fileként kezeltük, ennél azonban van egy némileg kényelmesebb lehetőségünk is. Az XML DOM (Document Object Model) a fileokat a hierarchikus felépítésük szerint kezeli. Az osztály amelyen keresztül elérjük a DOM –ot az az *XmlDocument* lesz. Egy másik fontos osztály, amelyből egyébként maga az *XmlDocument* is származik, az az *XmlNode*. Egy XML file egyes csomópontjait egy-egy *XmlNode* objektum fogja jelképezni.
Ahogy azt már megszokhattuk az *XmlDocument* számos forrásból táplálkozhat, pl. stream vagy egyszerű filenév. A forrás betöltését az *XmlDocument Load* illetve *LoadXml* metódusaival végezzük:

```
XmlDocument xdoc = new XmlDocument();
xdoc.Load("test.xml");
```

Egy *XmlDocument* bejárható *foreach* ciklussal a *ChildNodes* tulajdonságon keresztül, amely egy *XmlNodeList* objektumot ad vissza. Persze nem biztos, hogy léteznek gyermekei, ezt a *HasChildNodes* tulajdonsággal tudjuk ellenőrizni. A következő forráskódban egy *XmlDocument* első szinten lévő gyermekeit járjuk be:

```
foreach (XmlNode node in xdoc.ChildNodes)
{
    Console.WriteLine(node.Name);
}
```

Az *XmlDocument* a *ChildNodes* tulajdonságát az *XmlNode* osztálytól örökli (ugyanígy a *HasChildNodes*–t is, amely igaz értéket tartalmaz, ha léteznek gyermek-elemek), így az egyes *XmlNode*–okat bejárva a teljes „fát' megkaphatjuk.

Most nézzük meg, hogyan tudjuk manipulálni a filet. Hozzunk létre kódból egy teljesen új dokumentumot:

```
XmlDocument xdoc = new XmlDocument();

XmlElement element = xdoc.CreateElement("Test");
XmlText text = xdoc.CreateTextNode("Hello XML DOM!");

XmlNode node = xdoc.AppendChild(element);
node.AppendChild(text);

xdoc.Save("domtest.xml");
```

Az *XmlText* illetve *XmlElement* osztályok is az *XmlNode* leszármazottai. A *Save* metódussal el tudjuk menteni a dokumentumot, akár filenevet, akár egy streamet megadva. A fenti kód eredménye a következő lesz:

```
<Test>Hello XML DOM!</Test>
```

A *CloneNode* metódussal már létező csúcsokat „klónozhatunk":

```
XmlNode source = xdoc.CreateNode(XmlNodeType.Element, "test", "test");
XmlNode destination = source.CloneNode(false);
```

A metódus egyetlen paramétert vár, amely jelzi, hogy a csúcs gyermekeit is átmásoljuk–e.

Két másik metódusról is megemlékezünk, első a *RemoveChild*, amely egy létező csúcsot távolít el a csúcsok listájából, a másik pedig a *ReplaceChild*, amely felülír egy csúcsot.

Az *XmlDocument* eseményeket is szolgáltat, amelyek segítségével teljes körű felügyeletet nyerhetünk. Kezeljük le például azt az eseményt, amikor egy új csúcsot adunk hozzá:

```
XmlNodeChangedEventHandler handler = null;
handler = (sender, e) =>
    {
        Console.WriteLine(e.Node.Value);
    };
xdoc.NodeInserting += handler;
```

32.6 XML szerializáció

Mi is az a szérializálás? A legegyszerűbben egy példán keresztül lehet megérteni a fogalmat, képzeljük el, hogy írtunk egy játékot, és a játékosok pontszámát szeretnénk eltárolni. Az elért pontszám mellé jön a játékos neve és a teljesítés ideje is. Ez elsőre semmi különös, hiszen simán kiírhatjuk az adatokat egy fileba, majd visszaolvashatjuk onnan. Egy hátulütő mégis van, mégpedig az, hogy ez egy elég bonyolult feladat: figyelni kell a beolvasott adatok típusára, formátumára, stb…
Nem lenne egyszerűbb, ha a kiírt adatokat meg tudnánk feleltetni egy osztálynak? Ez megtehetjük, ha készítünk egy megfelelő osztályt, amelyet szérializálva kiírhatunk XML formátumba és onnan vissza is olvashatjuk (ezt deszérializálásnak hívják). Természetesen a szérializálás jelentősége ennél sokkal nagyobb és nemcsak XML segítségével tehetjük meg. Tulajdonképpen a szérializálás annyit tesz, hogy a memóriabeli objektumainkat egy olyan szekvenciális formátumba (pl. bináris, vagy most XML) konvertáljuk, amelyből vissza tudjuk alakítani az adatainkat.

Már említettük, hogy több lehetőségünk is van a szerializációra, ezek közül az XML a legáltalánosabb, hiszen ezt más környezetben is felhasználhatjuk.

Kezdjük egy egyszerű példával (hamarosan megvalósítjuk a pontszámos példát is), szérializáljunk egy hagyományos beépített objektumot:

```
FileStream fstream = new FileStream("serxml.xml", FileMode.Create);
XmlSerializer ser = new XmlSerializer(typeof(DateTime));
ser.Serialize(fstream, DateTime.Now);
```

Ezután létrejön egy XML file, benne a szérializált objektummal:

```
<?xml version="1.0"?>
<dateTime>2008-10-23T16:19:44.53125+02:00</dateTime>
```

Az *XmlSerializer* a *System.Xml.Serialization* névtérben található.
A deszérializálás hasonlóképpen működik:

```
FileStream fstream = new FileStream("serxml.xml", FileMode.Open);
XmlSerializer ser = new XmlSerializer(typeof(DateTime));
DateTime deser = (DateTime)ser.Deserialize(fstream);
```

Most már eleget tudunk ahhoz, hogy készítsünk egy szérializálható osztályt. Egy nagyon fontos dolgot kell megjegyeznünk, az osztálynak csakis a publikus tagjai szérializálhatóak a *private* vagy *protected* elérésűek automatikusan kimaradnak (emellett az osztálynak magának is publikus elérésűnek kell lennie). Ezeken kívül még szükség lesz egy alapértelmezett konstruktorra is (a deszérializáláshoz, hiszen ott még nem tudja, hogy milyen objektumról van szó).

```
public class ScoreObject
{
    public ScoreObject()
    {

    }

    private string playername;

    [XmlElement("PlayerName")]
    public string PlayerName
    {
        get { return playername; }
        set { playername = value; }
    }

    private int score;

    [XmlElement("Score")]
    public int Score
    {
        get { return score; }
        set { score = value; }
    }

    private DateTime date;

    [XmlElement("Date")]
    public DateTime Date
    {
        get { return date; }
        set { date = value; }
    }
}
```

Az egyes tulajdonságoknál beállítottuk, hogy az miként jelenjen meg a fileban (sima element helyett lehet pl. attribútum is). Enélkül is lefordulna és működne, de így jobban kezelhető. Rendben, most szérializáljuk:

```
ScoreObject so = new ScoreObject();
so.PlayerName = "Player1";
so.Score = 1000;
so.Date = DateTime.Now;

FileStream fstream = new FileStream("scores.xml", FileMode.Create);
XmlSerializer ser = new XmlSerializer(typeof(ScoreObject));
ser.Serialize(fstream, so);
```

És az eredmény:

```
<?xml version="1.0"?>
<ScoreObject xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <PlayerName>Player1</PlayerName>
  <Score>1000</Score>
  <Date>2008-10-23T16:34:32.734375+02:00</Date>
</ScoreObject>
```

33 Konfigurációs file használata

Eddig, amikor egy programot írtunk minden apró változtatásnál újra le kellett fordítani, még akkor is, ha maga a program nem, csak a használt adatok változtak. Sokkal kényelmesebb lenne a fejlesztés és a kész program használata is, ha a „külső” adatokat egy külön fileban tárolnánk. Természetesen ezt megtehetjük úgy is, hogy egy sima szöveges filet készítünk, és azt olvassuk/írjuk, de a .NET ezt is megoldja helyettünk a konfigurációs fileok bevezetésével. Ezek tulajdonképpen XML dokumentumok amelyek a következőképpen épülnek fel:

```
<?xml version="1.0" encoding="utf-8"?>

<configuration>
      <appSettings>
            <add key="1" value="alma" />
            <add key="2" value="korte" />
      </appSettings>
</configuration>
```

Az első sor egy szabványos XML file fejléce, vele nem kell foglalkoznunk. Sokkal érdekesebb viszont az *appSettings* szekció, ahová már be is raktunk néhány adatot. Az *appSettings* az általános adatok tárolására szolgál, vannak speciális szekciók (pl. adatbázisok eléréséhez) és mi magunk is készíthetünk ilyet (erről hamarosan).

A file ugyan megvan, de még nem tudjuk, hogy hogyan használjuk fel azt a programunkban. A konfig file neve mindig *alkalmazásneve.exe.config* formában használandó (vagyis a main.cs forráshoz – ha nem adunk meg mást – a main.exe.config elnevezést kell használnuk), ekkor a fordításnál ezt nem kell külön jelölni, automatikusan felismeri a fordítóprogram, hogy van ilyenünk is. Írjunk is egy egyszerű programot:

```
using System;
using System.Configuration; // ez kell

class Program
{
   static public void Main()
   {
            string s = ConfigurationManager.AppSettings["1"];

            Console.WriteLine(s); // alma
   }
}
```

Látható, hogy az *AppSettings* rendelkezik indexelővel, amely visszaadja a megfelelő értéket a megadott kulcshoz.

Ugyanezt elérhetjük a *ConfigurationSettings.AppSettings* –szel is, de az már elavult – erre a fordító is figyelmeztet – csak a korábbi verziók kompatibilitása miatt van még benne a frameworkben.

Az *AppSettings* indexelője minden esetben stringet ad vissza, tehát ha más típussal akarunk dolgozni, akkor konvertálni kell. Vezessünk be a konfig-fileba egy új kulcs-érték párost:

```
<add key="3" value="42" />
```

És használjuk is fel a forrásban:

```
using System;
using System.Configuration;

class Program
{
   static public void Main()
   {
            int x = int.Parse(ConfigurationManager.AppSettings["3"]);

            Console.WriteLine(x);
   }
}
```

33.1 Konfiguráció-szekció készítése

Egyszerűbb feladatokhoz megteszi a fenti módszer is, de hosszabb programok esetében nem kényelmes mindig figyelni a konverziókra. Épp ezért készíthetünk olyan osztályt is, amely egyrészt típusbiztos másrészt a konfigurációs fileban is megjelenik.
A következő példában elkészítünk egy programot, amely konfigurációs fileból kiolvas egy filenevet és a kódtáblát és ezek alapján kiírja a file tartalmát. Első lépésként készítsünk egy új forrásfilet AppDataSection.cs néven, amely a következőt tartlamazza:

```
using System;
using System.Configuration;

public class AppDataSection : ConfigurationSection
{
      private static AppDataSection settings =
            ConfigurationManager.GetSection("AppDataSettings")
                  as AppDataSection;

      public static AppDataSection Settings
      {
            get { return AppDataSection.settings; }
      }

      [ConfigurationProperty("fileName", IsRequired=true)]
      public string FileName
      {
            get
            {
                  return this["fileName"] as string;
            }
            set
            {
                  this["fileName"] = value;
            }
      }

      [ConfigurationProperty("encodeType", IsRequired=true)]
      public string EncodeType
```

```
        {
            get
            {
                return this["encodeType"] as string;
            }
        }
}
```

Az osztályt a *ConfigurationSection* osztályból származtattuk, ennek az indexelőjét használjuk. Itt arra kell figyelni, hogy az *AppSettings*–szel ellentétben ő *object* típussal tér vissza, vagyis muszáj konverziót végrehajtani.

A *Settings* property segítségével az osztályon keresztül egyúttal hozzáférünk az osztály egy példányához is, így soha nem kell példányosítanunk azt.

A tulajdonságok attribútumával beállítottuk az konfig-fileban szereplő nevet illetve, hogy muszáj megadnunk ezeket az értékeket (a *DefaultValue* segítségével kezdőértéket is megadhatunk). Most jön a konfig-file itt regisztrálnunk kell az új szekciót:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
      <configSections>
            <section name="AppDataSettings" type="AppDataSection, main" />
      </configSections>

      <AppDataSettings fileName="file.txt" encodeType="iso-8859-2" />
</configuration>
```

A *type* tulajdonságnál meg kell adnunk a típus teljes elérési útját névtérrel együtt (ha van), illetve azt az assemblyt amelyben a típus szerepel, ez a mi esetünkben a *main* lesz.
Most jöjjön az osztály, amely használja az új szekciót, készítsünk egy DataHandlerClass.cs nevű filet is:

```
using System;
using System.IO;
using System.Text;
using System.Configuration;

public class DataHandlerClass
{
      public void PrintData()
      {
            Encoding enc = Encoding.GetEncoding(
                  AppDataSection.Settings.EncodeType);
            string filename = AppDataSection.Settings.FileName;

            using(FileStream fs = new FileStream(filename, FileMode.Open))
            {
                  using(StreamReader sr = new StreamReader(fs, enc, false))
                  {
                        string s = sr.ReadLine();
                        while(s != null)
                        {
                              Console.WriteLine(s);
                              s = sr.ReadLine();
                        }
                  }
```

```
            }
        }
}
```

Most már könnyen használhatjuk az osztályt:

```
using System;
using System.Configuration;

class Program
{
   static public void Main()
   {
            DataHandlerClass handler = new DataHandlerClass();
            handler.PrintData();
   }
}
```

A fileokat a fordítóprogram után egymás után írva fordíthatjuk le:

```
csc main.cs AppDataSection.cs DataHandlerClass.cs
```

34 Hálózati programozás

A .NET meglehetősen széles eszköztárral rendelkezik hálózati kommunikáció kialakításához. A lista a legegyszerűbb hobbiszintű programokban használható szerver-kliens osztályoktól egészen a legalacsonyabb szintű osztályokig terjed. Ebben a fejezetben ebből a listából szemezgetünk.
A fejezethez tartozó forráskódok megtalálhatóak a jegyzethez tartozó Sources\Network könyvtárban.

34.1 Socket

A socketek számítógépes hálózatok (pl. az Internet) közötti kommunikációs végpontok. Minden socket rendelkezik két adattal, amelyek által egyértelműen azonosíthatóak és elérhetőek: ezek az IP cím és a port szám.

Mi az az IP cím? Az Internet az ún. Internet Protocol (IP) szabványa szerint működik. Eszerint a hálózaton lévő minden számítógép egyedi azonosítóval – IP címmel rendelkezik (ugyanakkor egy számítógép több címmel is rendelkezhet, ha több hálózati hardvert használ). Egy IP cím egy 32 bites egész szám, amelyet 8 bites részekre osztunk (pl.: 123.255.0.45), ezáltal az egyes szekciók legmagasabb értéke 255 lesz (ez a jellemző az IP negyedik generációjára vonatkozik, az új hatos generáció már 128 bites címeket használ, igaz ez egyelőre kevésbé elterjedt (a .NET támogatja az IPv4 és IPv6 verziókat is)).

Az IP címet tekinthetjük az ország/város/utca/házszám négyesnek, míg a portszámmal egy szobaszámra hivatkozunk. A portszám egy 16 bites előjel nélküli szám 1 és 65535 között. A portszámokat ki lehet „sajátítani”, vagyis ezáltal biztosítják a nagyobb szoftvergyártók, hogy ne legyen semmilyen ütközés a termék használatakor. A portszámok hivatalos regisztrációját az Internet Assigned Numbers Authority (IANA) végzi.
Az 1 és 1023 portokat ún. well-knowed portként ismerjük, ezeken olyan széleskörben elterjedt szolgáltatások futnak amelyek a legtöbb rendszeren megtaláhatóak (operációs rendszertől függetlenül). Pl. a böngészők a HTTP protokollt a 80–as porton érik el, a 23 a Telnet, míg a 25 az SMTP szerver portszáma (ezektől el lehet – és biztonsági okokból a nagyobb cégek el is szoktak – térni, de a legtöbb számítógépen ezekkel az értékekkel találkozunk).
Az 1024 és 49151 közötti portokat regisztrált portoknak nevezik, ezeken már olyan szolgáltatásokat is felfedezhetünk amelyek operációs rendszerhez (is) kötöttek pl. az 1433 a Microsoft SQL Server portja, ami értelemszerűen Windows rendszer alatt fut. Ugyanitt megtalálunk szoftverhez kötött portot is, pl a World of Warcraft a 3724 portot használja. Ez az a tartomány amit az IANA kezel.
Az efelettieket dinamikus vagy privát portoknak nevezik, ezt a tartományt nem lehet lefoglalni, programfejlesztés alatt célszerű ezeket használni.

Mielőtt nekiállunk a *Socket* osztály megismerésének, játsszunk egy kicsit az IP címekkel! Azt tudjuk már, hogy a hálózaton minden számítógép saját címmel rendelkezik, de számokat viszonylag nehéz megjegyezni, ezért feltalálták a domain-név vagy tartománynév intézményét, amely „ráül” egy adott IP címre, vagyis ahelyett,

hogy 65.55.21.250 írhatjuk azt, hogy www.microsoft.com. A következő programunkban lekérdezzük egy domain-név IP címét és fordítva. Ehhez a *System.Net* névtér osztályai lesznek segítségünkre:

```
using System;
using System.Net; // ez kell

class Program
{
	static public void Main()
	{
		IPHostEntry host1 = Dns.GetHostEntry("www.microsoft.com");

		foreach(IPAddress ip in host1.AddressList)
		{
			Console.WriteLine(ip.ToString());
		}

		IPHostEntry host2 = Dns.GetHostEntry("91.120.22.150");
		Console.WriteLine(host2.HostName);
	}
}
```

Most már ideje mélyebb vizek felé venni az irányt, elkészítjük az első szerverünket. A legegyszerűbb módon fogjuk csinálni a beépített *TcpListener* osztállyal, amely a TCP/IP protokollt használja (erről hamarosan részletesebben). Nézzük meg a forrást:

```
using System;
using System.Net;
using System.Net.Sockets;

public class Server
{
	static public void Main(string[] args)
	{
		IPAddress ip = IPAddress.Parse(args[0]);
		int port = int.Parse(args[1]);
		IPEndPoint endPoint = new IPEndPoint(ip, port);

		TcpListener server = new TcpListener(endPoint);
		server.Start();

		Console.WriteLine("A szerver elindult!");
	}
}
```

Ezt a programot így futtathatjuk:

```
.\Server.exe 127.0.0.1 50000
```

A 127.0.0.1 egy speciális cím, ez az ún. localhost vagyis a „helyi” cím, amely jelen esetben a saját számítógépünk (ehhez Internet kapcsolat sem szükséges mindig rendelkezésre áll). Ez az ip cím lesz az, ahol a szerver bejövő kapcsolatokra fog várni (érelemszerűen ehhez az IP címhez csak a saját számítógépünkről tudunk kapcsolódni, ha egy távoli gépet is szeretnénk bevonni, akkor szükség lesz a „valódi” IP –re, ezt pl. a parancssorba beírt ipconfig paranccsal tudhatjuk meg).

```
using System;
using System.Net;
using System.Net.Sockets;

public class Server
{
	static public void Main(string[] args)
	{
		TcpListener server = null;

		try
		{
			IPAddress ipAddr = IPAddress.Parse(args[0]);
			int portNum = int.Parse(args[1]);
			IPEndPoint endPoint = new IPEndPoint(ipAddr, portNum);

			server = new TcpListener(endPoint);
			server.Start();

			Console.WriteLine("A szerver elindult!");
		}
		catch(Exception e)
		{
			Console.WriteLine(e.Message);
		}
		finally
		{
			server.Stop();

			Console.WriteLine("A szerver leállt!");
		}
	}
}
```

A *TcpListener* konstruktorában meg kell adnunk egy végpontot a szervernek, ahol figyelheti a bejövő kapcsolatokat, ehhez szükségünk lesz egy IP címre és egy portszámra. Előbbit az *IPAddress.Parse* statikus metódussal egy karaktersorozatból nyertük ki.

A következő lépésben bejövő kapcsolatot fogadunk, ehhez viszont szükségünk lesz egy kliensre is, őt a *TcpClient* osztállyal készítjük el, amely hasonlóan működik, mint a párja, szintén egy cím-port kettősre lesz szükségünk (de most nem kell végpontot definiálnunk):

```
using System;
using System.Net;
using System.Net.Sockets;

class Program
{
	static public void Main(string[] args)
	{
		TcpClient client = null;

		try
		{
			client = new TcpClient(args[0], int.Parse(args[1]));
		}
		catch(Exception e)
		{
			Console.WriteLine(e.Message);
		}
```

```
            finally
            {
                  client.Close();
            }
      }
}
```

A *TcpClient* –nek vagy a konstruktorban rögtön megadjuk az elérni kívánt szerver nevét és portját (és ekkor azonnal csatlakozik is), vagy meghívjuk az alapértelmezett konstruktort és a *Connect* metódussal elhalasztjuk a kapcsolódást egy későbbi időpontra (természetesen ekkor neki is meg kell adni a szerver adatait).
Ahhoz, hogy a szerver fogadni tudja a kliensünket meg kell mondanunk neki, hogy várjon amíg bejövő kapcsolat érkezik, ezt a *TcpListener* osztály *AcceptTcpClient* metódusával tehetjük meg:

```
TcpClient client = server.AcceptTcpClient();
```

Persze szerveroldalon egy *TcpClient* objektumra lesz szükségünk, ezt adja vissza a metódus.
A programunk jól működik, de nem csinál túl sok mindent. A következő lépésben adatokat küldünk oda-vissza. Nézzük a módosított klienst:

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class Program
{
      static public void Main(string[] args)
      {
            TcpClient client = null;
            NetworkStream stream = null;

            try
            {
                  client = new TcpClient(args[0], int.Parse(args[1]));
                  byte[] data = Encoding.ASCII.GetBytes("Hello szerver!");
                  stream = client.GetStream();
                  stream.Write(data, 0, data.Length);
                  data = new byte[256];
                  int length = stream.Read(data, 0, data.Length);

                  Console.WriteLine("A szerver üzenete: {0}",
                        Encoding.ASCII.GetString(data, 0, length));
            }
            catch(Exception e)
            {
                  Console.WriteLine(e.Message);
            }
            finally
            {
                  stream.Close();
                  client.Close();
            }
      }
}
```

Az elküldeni kívánt üzenetet byte–tömb formájában kell továbbítanunk, mivel a TCP/IP byet-ok sorozatát továbbítja, ezért számára emészthető formába kell hoznunk az üzenetet. A fenti példában a legegyszerűbb, ASCII kódolást választottuk, de mást is használhatunk a lényeg, hogy tudjuk byte-onként küldeni (értelemszerűen mind a kliens, mind a szerver ugyanazt a kódolást kell használja).
A *NetworkStream* ugyanolyan adatfolyam, amit a filkezeléssel foglalkozó fejezetben megismertünk, csak éppen az adatok most a hálózaton keresztül „folynak” át.

Már ismerjük az alapokat, eljött az ideje, hogy egy szinttel lejjebb lépjünk a socketek világába.

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.IO;

class Program
{
	static public void Main(string[] args)
	{
		Socket server = null;
		Socket client = null;

		try
		{
			server = new Socket(
				AddressFamily.InterNetwork,
				SocketType.Stream,
				ProtocolType.Tcp);

			IPEndPoint endPoint = new IPEndPoint(
				IPAddress.Parse(args[0]),
				int.Parse(args[1]));

			server.Bind(endPoint);
			server.Listen(2);

			client = server.Accept();

			byte[] data = new byte[256];
			int length = client.Receive(data);

			Console.WriteLine("A kliens üzenete: {0}",
				Encoding.ASCII.GetString(data, 0, length));

			data = new byte[256];
			data = Encoding.ASCII.GetBytes("Hello kliens!");
			client.Send(data, data.Length, SocketFlags.None);
		}
		catch(Exception e)
		{
			Console.WriteLine(e.Message);
		}
		finally
		{
			client.Close();
			server.Close();
		}
	}
}
```

A *Socket* osztály konstruktorában megadtuk a címzési módot (*InterNetwork*, ez az IPv4), a socket típusát (*Stream*, ez egy oda-vissza kommunikációt biztosító kapcsolat lesz) és a használt protokollt, ami jelen esetben TCP. A három paraméter nem független egymástól, pl. *Stream* típusú socketet csak TCP portokoll felett használhatunk.
Ezután készítettünk egy *IPEndPoint* objektumot, ahogyan azt az egyszerűbb változatban is tettük. Ezt a végpontot a *Bind* metódussal kötöttük a sockethez, majd a *Listen* metódussal megmondtuk, hogy figyelje a bejövő kapcsolatokat. Ez utóbbi paramétere azt jelzi, hogy maximum hány bejövő kapcsolat várakozhat.
Innentől kezdve nagyon ismerős a forráskód, lényegében ugyanazt tesszük, mint eddig, csak épp más a metódus neve.
A kliens osztály sem okozhat nagy meglepetést:

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.IO;

class Program
{
	static public void Main(string[] args)
	{
		Socket client = null;

		try
		{
			client = new Socket(
				AddressFamily.InterNetwork,
				SocketType.Stream,
				ProtocolType.Tcp);

			IPEndPoint endPoint = new IPEndPoint(
				IPAddress.Parse(args[0]),
				int.Parse(args[1]));

			client.Connect(endPoint);

			byte[] data = new byte[256];
			data = Encoding.ASCII.GetBytes("Hello szerver!");

			client.Send(data, data.Length, SocketFlags.None);

			data = new byte[256];
			int length = client.Receive(data);

			Console.WriteLine("A szerver üzenete: {0}",
				Encoding.ASCII.GetString(data, 0, length));
		}
		catch(Exception e)
		{
			Console.WriteLine(e.Message);
		}
		finally
		{
			client.Close();
		}
	}
}
```

A különbséget a *Connect* metódus jelenti, mivel most kapcsolódni akarunk, nem hallgatózni.

34.2 Blokk elkerülése

Az eddigi programjaink mind megegyeztek abban, hogy bizonyos műveletek blokkolták a főszálat és így várakozni kényszerültünk. Ilyen művelet volt pl. az *Accept/AcceptTcpClient*, de a *Read/Receive* is.
A blokk elkerülésére ún. előre-ellenőrzést (prechecking) fogunk alkalmazni, vagyis megvizsgáljuk, hogy adott időpillanatban van-e bejövő adat, vagy ráérünk később újraellenőrizni, addig pedig csinálhatunk mást.
A *TcpListerner/TcpClient* osztályok a rájuk csatlakoztatott *NetworkStream* objektum *DataAvailable* tulajdonságán keresztül tudják vizsgálni, hogy jön-e adat vagy sem. A következő példában a kliens rendszeres időközönként ellenőrzi, hogy érkezett-e válasz a szervertől és ha az eredmény negatív akkor foglalkozhat a dolgával:

```
bool l = false;
while(!l)
{
      if(stream.DataAvailable)
      {
            data = new byte[256];
            int length = stream.Read(data, 0, data.Length);

            Console.WriteLine("A szerver üzenete: {0}",
                  Encoding.ASCII.GetString(data, 0, length));

            l = true;
      }
      else
      {
            Console.WriteLine("A szerver még nem küldött választ!");
            System.Threading.Thread.Sleep(200);
      }
}
```

Ugyanezt a hatást socketek esetében a *Socket* osztály *Available* tulajdonságával érhetjük el, amely jelzi, hogy van-e még várakozó adat a csatornán (egészen pontosan azoknak a byte-oknak a számát adja vissza amelyeket még nem olvastunk be):

```
while(true)
{
      if(client.Available > 0)
      {
            int length = client.Receive(data);
            Console.WriteLine("A szerver üzenete: {0}",
                  Encoding.ASCII.GetString(data, 0, length));
            break;
      }
      else
      {
            Console.WriteLine("Várunk a szerver válaszára...");
            System.Threading.Thread.Sleep(200);
      }
}
```

Itt egy másik módszert választottunk a ciklus kezelésére.
Most nézzük meg, hogy mi a helyzet szerveroldalon. Itt a tipikus probléma az, hogy az *AcceptTcpClient/Accept* teljesen blokkol, amíg bejövő kapcsolatra várakozunk. Erre is van persze megoldás, a *TcpListener* esetében ezt a *Pending*- , míg a *Socket* osztálynál a *Poll* metódus jelenti.
Nézzük elsőként a *TcpListener* –t:

```
while(true)
{
      if(server.Pending())
      {
            client = server.AcceptTcpClient();

            Console.WriteLine("Kliens kapcsolódott...");

            stream = client.GetStream();
            byte[] data = new byte[256];
            int length = stream.Read(data, 0, data.Length);

            Console.WriteLine("A kliens üzenete: {0}",
                  Encoding.ASCII.GetString(data, 0, length));

            data = Encoding.ASCII.GetBytes("Hello kliens!");
            stream.Write(data, 0, data.Length);
      }
      else
      {
            Console.WriteLine("Most valami mást csinálunk");
            System.Threading.Thread.Sleep(500);
      }
}
```

A *Pending* azt az információt osztja meg velünk, hogy várakozik-e bejövő kapcsolat. Tulajdonképpen ez a metódus a következő forrásban szereplő (a *Socket* osztályhoz tartozó) *Poll* metódust használja:

```
while(true)
{
      if(server.Poll(0, SelectMode.SelectRead))
      {
            client = server.Accept();

            /* itt pedig kommunikálunk a klienssel */
      }
      else
      {
            Console.WriteLine("A szerver bejövõ kapcsolatra vár!");
            System.Threading.Thread.Sleep(500);
      }
}
```

A *Poll* metódus első paramétere egy egész szám, amely mikromásodpercben (nem milli-, itt valóban a másodperc milliomod részéről van szó, vagyis ha egy másodpercig akarunk várni akkor 1000000–ot kell megadnunk) adja meg azt az időt amíg várunk bejövő kapcsolatra/adatra. Amennyiben az első paraméter negatív szám, akkor addig várunk, amíg nincs kapcsolat (vagyis blokkoljuk a programot), ha pedig nullát adunk meg akkor használhatjuk pre-checking–re is a metódust.
A második paraméterrel azt mondjuk meg, hogy mire várunk. A *SelectMode* felsorolt típus három taggal rendelkezik:

1. *SelectRead*: a metódus igaz értékkel tér vissza, ha meghívtuk a *Listen* metódust, és várakozik bejövő kapcsolat vagy van bejövő adat illetve ha a kapcsolat megszűnt, minden más esetben a visszatérési érték *false* lesz.
2. *SelectWrite*: igaz értéket kapunk vissza, ha a *Connect* metódus hívása sikeres volt, azaz csatlakoztunk a távoli állomáshoz, illetve ha lehetséges adat küldése.
3. *SelectError*: *true* értéket ad vissza, ha a *Connect* metódus hívása sikertelen volt.

Egy másik lehetőség a blokk feloldására, ha a *Socket* objektum *Blocking* tulajdonságát false értékre állítjuk. Ekkor a *Socket* osztály *Receive* és *Send* metódusainak megadhatunk egy *SocketError* típusú (out) paramétert, amely *WouldBlock* értékkel tér vissza, ha a metódushívás blokkot okozna (vagyis így újra próbálhatjuk küldeni/fogadni később az adatokat). Azt azonban nem árt tudni, hogy ez és a fenti *Poll* metódust használó megoldások nem hatékonyak, mivel folyamatos metódushívásokat kell végrehajtanunk (vagy a háttérben a rendszernek). A következő fejezetben több kliens egyidejű kezelésével egy sokkal hatékonyabb módszer(eke)t fogunk megvizsgálni.

34.3 Több kliens kezelése

Az eddigi programjainkban csak egy klienst kezeltünk, ami nagyon kényelmes volt, mivel egy sor dolgot figyelmen kívül hagyhattunk:

1. A kliensekre mutató „referencia” tárolása
2. A szerver akkor is tudjon kapcsolatot fogadni, amíg a bejelentkezett kliensekkel foglalkozunk.
3. Minden kliens zavartalanul (lehetőleg blokk nélkül) tudjon kommunikálni a szerverrel és viszont.

A következőkben háromféleképpen fogjuk körüljárni a problémát.

34.3.1 Select

Az első versenyzőnk a *Socket* osztály *Select* metódusa lesz, amelynek segítségével meghatározhatjuk egy vagy több *Socket* példány állapotát. Lényegében arról van szó, hogy egy listából kiválaszthatjuk azokat az elemeket, amelyek megfelelnek bizonyos követelményeknek (írhatóak, olvashatóak). A *Select* (statikus) metódus szignatúrája a következőképpen néz ki:

```
public static void Select(
      IList checkRead,
      IList checkWrite,
      IList checkError,
      int microSeconds
)
```

Az első három paraméter olyan *IList* interfészt implementáló gyűjtemények (lényegében az összes beépített gyűjtemény ilyen beleértve a sima tömböket is) amelyek *Socket* példányokat tartalmaznak. Az első paraméternek megadott listát olvashatóságra, a másodikat írhatóságra, míg a harmadikat hibákra ellenőrzi a *Select*, majd a feltételnek megfelelő listaelemeket megtartja a listában a többit eltávolítja (vagyis ha szükségünk van a többi elemre is akkor célszerű egy másolatot használni az eredeti lista helyett). Az utolsó paraméterrel azt adjuk meg, hogy mennyi ideig várjunk a kliensek válaszára (mikroszekundum).
Készítsünk egy *Select*–et használó kliens-szerver alkalmazást! A kliens oldalon lényegében semmit nem változtatunk azt leszámítva, hogy folyamatosan üzenetet küldünk a szervernek (most csak egyirányú lesz a kommunikáció):

```
Random r = new Random();
while(true)
{
	if(r.Next(1000) % 2 == 0)
	{
		byte[] data = new byte[256];
		data = Encoding.ASCII.GetBytes("Hello szerver!");

		client.Send(data, data.Length, SocketFlags.None);
	}

	System.Threading.Thread.Sleep(500);
}
```

Most nézzük a szervert:

```
int i = 0;
while(i < MAXCLIENT)
{
	Socket client = server.Accept();
	clientList.Add(client);
	++i;
}

while(true)
{
	ArrayList copyList = new ArrayList(clientList);
	Socket.Select(copyList, null, null, 1000);

	foreach(Socket client in clientList)
	{
		Program.CommunicateWithClient(client);
	}
}
```

A MAXCLIENT változó egy egyszerű egész szám, meghatározzuk vele, hogy maximum hány klienst fogunk kezelni. Miután megfelelő számú kapcsolatot hoztunk létre elkezdünk „beszélgetni” a kliensekkel. A ciklus minden iterációjában meghívja a *Select* metódust, vagyis kiválasztjuk, hogy melyik kliensnek van mondandója. A *CommunicateWithClient* statikus metódus a már ismert módon olvassa a kliens üzenetét:

```
static public void CommunicateWithClient(Socket client)
{
      byte[] data = new byte[256];
      int length = client.Receive(data);

      Console.WriteLine("A kliens üzenete: {0}",
            Encoding.ASCII.GetString(data, 0, length));
}
```

34.3.2 Aszinkron socketek

Kliensek szinkron kezelésére is felhasználhatjuk a *Socket* osztályt, lényegében az aszinkron delegate -ekre épül ez a megoldás. Az aszinkron hívható metódusok Begin és End előtagot kaptak, pl. kapcsolat elfogadására most a *BeginAccept* metódust fogjuk használni.
A kliens ezúttal is változatlan, nézzük a szerver oldalt:

```
while(true)
{
      done.Reset();

      Console.WriteLine("A szerver kapcsolatra vár...");

      server.BeginAccept(Program.AcceptCallback, server);

      done.WaitOne();
}
```

A *done* változó a *ManualResetEvent* osztály egy példánya, segítségével szabályozni tudjuk a szálakat. A *Reset* metódus alaphelyzetbe állítja az objektumot, míg a *WaitOne* megállítja (blokkolja) az aktuális szálat, amíg egy jelzést (A *Set* metódussal) nem kap. A *BeginAccept* aszinkron metódus első paramétere az a metódus lesz, amely a kapcsolat fogadását végzi, második paraméternek pedig átadjuk a szervert reprezentáló *Socket* objektumot. Tehát: meghívjuk a *BeginAccept*–et, ezután pedig várunk, hogy az *AcceptCallback* metódus visszajelezzen a főszálnak, hogy a kliens csatlakozott és folytathatja a figyelést. Nézzük az *AcceptCallback*–ot:

```
static public void AcceptCallback(IAsyncResult ar)
{
      Socket client = ((Socket)ar.AsyncState).EndAccept(ar);
      done.Set();

      Console.WriteLine("Kliens kapcsolódott...");

      StringState state = new StringState();
      state.Client = client;

      client.BeginReceive(state.Buffer, 0, State.BufferSize,
            SocketFlags.None, Program.ReadCallback, state);
}
```

A kliens fogadása után meghívtuk a *ManualResetEvent Set* metódusát, ezzel jeleztük, hogy kész vagyunk, várhatjuk a következő klienst (látható, hogy csak addig kell blokkolnunk, amíg hozzá nem jutunk a klienshez, az adatcsere nyugodtan elfut a háttérben).

A *StringState* osztályt kényelmi okokból mi magunk készítettük el, ezt fogjuk átadni a *BeginReceive* metódusnak:

```
class State
{
      public const int BufferSize = 256;

      public State()
      {
            Buffer = new byte[BufferSize];
      }

      public Socket Client { get; set; }
      public byte[] Buffer { get; set; }
}

class StringState : State
{
      public StringState() : base()
      {
            Data = new StringBuilder();
      }

      public StringBuilder Data { get; set; }
}
```

Végül nézzük a *BeginReceive* callback metódusát:

```
static public void ReadCallback(IAsyncResult ar)
{
      StringState state = (StringState)ar.AsyncState;

      int length = state.Client.EndReceive(ar);

      state.Data.Append(Encoding.ASCII.GetString(state.Buffer, 0, length));

      Console.WriteLine("A kliens üzenete: {0}", state.Data);

      state.Client.Close();
}
```

A fordított irányú adatcsere ugyanígy zajlik, a megfelelő metódusok Begin/End előtagú változataival.

34.3.3 Szálakkal megvalósított szerver

Ez a módszer nagyon hasonló az aszinkron változathoz, azzal a különbséggel, hogy most manuálisan hozzuk létre a szálakat, illetve nem csak a beépített aszinkron metódusokra támaszkodhatunk, hanem tetszés szerinti műveletet hajthatunk végre a háttérben.
A következő programunk egy számkitalálós játékot fog megvalósítani úgy, hogy a szerver gondol egy számra és a kliensek ezt megpróbálják kitalálni. Minden kliens ötször találgathat, ha nem sikerül kitalálnia, akkor elveszíti a játékot. Mindent tipp után a szerver visszaküld egy számot a kliensnek: -1–et, ha a gondolt szám nagyobb, 1–et ha a szám kissebb, 0–át ha eltalálta a számot és 2–t, ha valaki már

kitalálta a számot. Ha egy kliens kitalálta a számot, akkor a szerver megvárja, míg minden kliens kilép, és új számot sorsol.

Készítettünk egy osztályt, amely megkönnyíti a dolgunkat. A *Listen* metódussal indítjuk el a szervert:

```
public void Listen()
{
      if(EndPoint == null)
      {
            throw new Exception("IPEndPoint missing");
      }

      server = new Socket(AddressFamily.InterNetwork,
                                    SocketType.Stream,
                                    ProtocolType.Tcp);
      server.Bind(EndPoint);
      server.Listen(5);

      ThreadPool.SetMaxThreads(MaxClient, 0);
      NewTurnEvent += NewTurnHandler;

      Console.WriteLine("A szerver elindult, a szám: {0}", NUMBER);

      Socket client = null;
      while(true)
      {
            client = server.Accept();

            Console.WriteLine("Kliens bejelentkezett");

            ThreadPool.QueueUserWorkItem(ClientHandler, client);
      }
}
```

A kliensek kezeléséhez a *ThreadPool* osztályt fogjuk használni, amelynek segítségével a kliens objektumokat átadjuk a *ClientHandler* metódusnak:

```
private void ClientHandler(object socket)
{
      Socket client = null;
      string name = String.Empty;
      int result = 0;

      try
      {
            client = socket as Socket;

            if(client == null)
            {
                  throw new ArgumentException();
            }

            ++ClientNumber;

            byte[] data = new byte[7];
            int length = client.Receive(data);
            name = Encoding.ASCII.GetString(data, 0, length);

            Console.WriteLine("Új játékos: {0}", name);

            int i = 0;
```

```
            bool win = false;
            while(i < 5 && win == false)
            {
                  data = new byte[128];
                  length = client.Receive(data);

                  GuessNumber(name,
                        int.Parse(Encoding.ASCII.GetString(
                                    data, 0, length)),
                                    out result);

                  data = Encoding.ASCII.GetBytes(result.ToString());
                  client.Send(data, data.Length, SocketFlags.None);

                  if(result == 0) { win = true; }

                  ++i;
            }
      }
      catch(Exception e)
      {
            Console.WriteLine(e.Message);
      }
      finally
      {
            client.Close();

            if(--ClientNumber == 0 && result == 0)
            {
                  NewTurnEvent(this, null);
            }
      }
}
```

Minden kliens rendelkezik névvel is, amely 7 karakter hosszú (7 byte) lehet, elsőként ezt olvassuk be. A *GuessNumber* metódusnak adjuk át a tippünket és a *result* változót out paraméterként. Végül a *finally* blokkban ellenőrizzük, hogy van-e bejelentkezett kliens, ha pedig nincs akkor új számot sorsolunk.
Nézzük a *GuessNumber* metódust:

```
private void GuessNumber(string name, int number, out int result)
{
      lock(locker)
      {
            if(NUMBER != -1)
            {
                  Console.WriteLine("{0} szerint a szám: {1}",
                        name, number);

                  if(NUMBER == number)
                  {
                        Console.WriteLine("{0} kitalálta a számot!", name);
                        result = 0;
                        NUMBER = -1;
                  }
                  else if(NUMBER < number)
                  {
                        result = 1;
                  }
                  else
                  {
                        result = -1;
                  }
```

```
            }
            else result = 2;
      }

      Thread.Sleep(300);
}
```

A metódus törzsét le kell zárnunk, hogy egyszerre csak egy szál (egy kliens) tudjon hozzáférni. Ha valamelyik kliens kitalálta számot, akkor annak -1–et adunk vissza, így gyorsan ellenőrizhetjük, hogy a feltétel melyik ágába kell bemennünk.
Kliens oldalon sokkal egyszerűbb dolgunk van, ezt a forráskódot itt nem részletezzük, de megtalálható a jegyzethez tartozó források között.

34.4 TCP és UDP

Ez a fejezet elsődlegesen a TCP protokollt használta, de említést kell tennünk az Internet másik alapprotokolljáról az UDP–ről is.
A TCP egy megbízható, de egyúttal egy kicsit lassabb módszer. A csomagokat sorszámmal látja el, ez alapján pedig a fogadó fél nyugtát küld, hogy az adott csomag rendben megérkezett. Amennyiben adott időn belül a nyugta nem érkezik meg, akkor a csomagot újra elküldi. Ezen kívül ellenőrzi, hogy a csomagok sérülésmentesek legyenek, illetve kiszűri a duplán elküldött (redundáns) adatokat is.
Az UDP épp az ellenkezője, nem biztosítja, hogy minden csomag megérkezik, cserében gyors lesz. Jellemzően olyan helyeken használják, ahol a gyorsaság számít és nem nagy probléma, ha egy-két csomag elveszik, pl. valós idejű média lejátszásnál illetve játékoknál.

A .NET a TCP–hez hasonlóan biztosítja számunkra az *UdpListener/Client* osztályokat, ezek kezelése gyakorlatilag megegyezik TCP –t használó párjaikkal, ezért itt most nem részletezzük.
Hagyományos *Socket*–ek esetén is elérhető ez a protokoll, ekkor a *Socket* osztály konstruktora így fog kinézni:

```
Socket server = new Socket(
      AddressFamily.InterNetwork,
      SocketType.Dgram,
      ProtocolType.Udp);
```

Ezután ugyanúgy használhatjuk ezt az objektumot, mint a TCP–t használó társát.

35 LINQ To Objects

A C# 3.0 bevezette a LINQ –t (Language Integrated Query), amely lehetővé teszi, hogy könnyen, uniformizált úton kezeljük a különböző adatforrásokat, vagyis pontosan ugyanúgy fogunk kezelni egy adatbázist, mint egy memóriában lévő gyűjteményt. Miért jó ez nekünk? Napjainkban rengeteg adatforrás áll rendelkezésünkre, ezek kezeléséhez pedig új eszközök használatát illetve új nyelveket kell megtanulnunk (SQL a relációs adatbázisokhoz, XQuery az XML–hez, stb...).A LINQ lehetővé teszi, hogy egy plusz réteg (a LINQ „felület") bevezetésével mindezeket áthidaljuk méghozzá teljes mértékben függetlenül az adatforrástól. Egy másik előnye pedig, hogy a LINQ lekérdezések erősen típusosak, vagyis a legtöbb hibát még fordítási időben el tudjuk kapni és kijavítani.
A LINQ család jelenleg három főcsapást jelölt ki, ezek a következőek:

- LINQ To XML: XML dokumentumok lekérdezését és szerkesztését teszi lehetővé.
- LINQ To SQL (vagy DLINQ) és LINQ To Entities (Entity Framework): relációs adatbázisokon (elsősorban MS SQL-Server) végezhetünk műveleteket velük. A kettő közül a LINQ To Entites a „főnök", a LINQ To SQL inkább csak technológiai demónak készült, a Microsoft nem fejleszti tovább (de továbbra is elérhető marad, mivel kisebb projectekre illetve hobbifejlesztésekhez kiváló). Az Entity Framework használatához a Visual Studio 2008 első szervizcsomagjára van szükség.
- LINQ To Objects: ennek a fejezetnek a tárgya, memóriában lévő gyűjtemények, listák, tömbök feldolgozását teszi lehetővé (lényegében minden olyan osztállyal működik amely megvalósítja az *IEnumerable<T>* interfészt).

A fentieken kívül számos third-party/hobbi project létezik, mivel a LINQ „framework" viszonylag könnyen kiegészíthető tetszés szerinti adatforrás használatához.
A fejezethez tartozó forráskódok megtalálhatóak a Sources\LINQ könyvtárban.

35.1 Nyelvi eszközök

A C# 3.0–ban megjelent néhány újdonság részben a LINQ miatt került a nyelvbe, jelenlétük jelentősen megkönnyíti a dolgunkat. Ezek a következők:

Kiterjesztett metódusok (extension method): velük már korábban megismerkedtünk, a LINQ To Objects teljes funkcionalitása ezekre épül, lényegében az összes művelet az *IEnumerable<T>/IEnumerable* interfészeket egészíti ki.

Objektum és gyűjtemény inicializálók: vagyis a lehetőség, hogy az objektum deklarálásával egyidőben beállíthassuk a tulajdonságaikat, illetve gyűjtemények esetében az elemeket:

```
using System;
using System.Collections.Generic;

class Program
{
      static public void Main()
      {
            /*Objektum inicializáló*/
            MyObject mo = new MyObject()
            {
                  Property1 = "value1";
                  Property2 = "value2";
            };

            /*Gyűjtemény inicializáló*/
            List<string> list = new List<string>()
            {
                  "alma", "körte", "dió", "kakukktojás"
            };
      }
}
```

Lambda kifejezések: a lekérdezések többségénél nagyon kényelmes lesz a lambdák használata, ugyanakkor lehetőség van a „hagyományos” névtelen metódusok felhasználására is.

A **„var”**: a lekérdezések egy részénél egészen egyszerűen lehetetlen előre megadni az eredmény-lista típusát, ezért ilyenkor a *var* –t fogjuk használni.

Névtelen típusok: sok esetben nincs szükségünk egy objektum minden adatára, ilyenkor feleslegesen foglalná egy lekérdezés eredménye a memóriát. A névtelen típusok beveztése lehetővé teszi, hogy helyben deklaráljunk egy névtelen osztályt:

```
using System;

class Program
{
      static public void Main()
      {
            var type1 = new { Value1 = "alma", Value2 = "körte" };
            Console.WriteLine(type1.Value1); //alma
      }
}
```

35.2 Kiválasztás

A legegyszerűbb dolog, amit egy listával tehetünk, hogy egy vagy több elemét valamilyen kritérium alapján kiválasztjuk. A következő forráskódban épp ezt fogjuk tenni, egy egész számokat tartalmazó *List<T>* adatszerkezetből az összes számot lekérdezzük. Természetesen ennek így sok értelme nincsen, de szűrésről majd csak a következő fejezetben fogunk tanulni.

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

class Program
{
	static public void Main()
	{
		List<int> list = new List<int>()
		{
			10, 2, 4, 55, 22, 75, 30, 11, 12, 89
		};

		var result = from number in list select number;
		foreach(var item in result)
		{
			Console.WriteLine("{0}", item);
		}
	}
}
```

Elsősorban vegyük észre az új névteret a *System.Linq*–t. Rá lesz szükségünk mostantól az összes lekérdezést tartalmazó programban, tehát ne felejtsük le.
Most pedig jöjjön a lényeg, nézzük a következő sort:

```
var result = from number in list select number;
```

Egy LINQ lekérdezés a legegyszerűbb formájában a következő sablonnal írható le:

eredmény = *from* azonosító *in* kifejezés *select* kifejezés

A lekérdezés első fejében meghatározzuk az adatforrást:

from azonosító *in* kifejezés

Egészen pontosan a „kifejezés” jelöli a forrást, míg az „azonosító” a forrás egyes tagjait jelöli a kiválasztás minden iterációjában, lényegében pontosan ugyanúgy működik, mint a *foreach* ciklus:

foreach(*var* azonosító *in* kifejezés)

Vagyis a lekérdezés alatt a forrás minden egyes elemével tehetünk, amit jólesik. Ezt a valamit a lekérdezés második fele tartalmazza:

select kifejezés

A fenti példában egyszerűen vissza akarjuk kapni a számokat eredeti formájukban, de akár ezt is írhattuk volna:

```
var result = from number in list select (number + 10);
```

Azaz minden számhoz hozzáadunk tízet.

Az SQL –t ismerők számára furcsa lehet, hogy először az adatforrást határozzuk meg, de ennek megvan az oka, mégpedig az, hogy ilyen módon a fejlesztőeszköz (a Visual Studio) támogatást illetve típusellenőrzést adhat a *select* utáni kifejezéshez (illetve a lekérdezés többi részéhez).

A fenti kódban SQL szerű lekérdezést készítettünk (ún. Query Expression Format), de máshogyan is megfogalmazhattuk volna a mondanivalónkat. Emlékezzünk, minden LINQ To Objects művelet egyben kiterjesztett metódus, vagyis ezt is írhattuk volna:

```
var result = list.Select(number => number);
```

Pontosan ugyanazt értük el, és a fordítás után pontosan ugyanazt a kifejezést is fogja használni a program, mindössze a szintaxis más (ez pedig az Extension Method Format) (a fordító is ezt a formátumot tudja értelmezni, tehát a Query Syntax is erre alakul át). A *Select* az adatforrás elemeinek típusát használó *Func<T, V>* generikus delegate–et kap paraméteréül, jelen esetben ezt egy lambda kifejezéssel helyettesítettük, de írhattuk volna a következőket is:

```
var result1 = list.Select(
      delegate (int number)
      {
            return number;
      });

Func<int, int> selector = (x) => x;
var result2 = list.Select(selector);
```

A két szintaxist keverhetjük is (Query Dot Format), de ez az olvashatóság rovására mehet, ezért nem ajánlott (leszámítva olyan eseteket, amikor egy művelet csak az egyik formával használható), ha csak lehet, ragaszkodjunk csak az egyikhez.
A következő példában a *Distinct* metódust használjuk, amely a lekérdezés eredményéből eltvolítja a duplikált elemeket. Őt csakis Extension Method formában hívhatjuk meg:

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

class Program
{
      static public void Main()
      {
            List<int> list = new List<int>()
            {
                  1, 1, 3, 5, 6, 6, 10, 11, 1
            };

            var result = (from number in list select number).Distinct();

            foreach(var item in result)
            {
                  Console.WriteLine("{0}", item);
            }
      }
}
```

A jegyzet ezután mindkét változatot bemutatja a forráskódokban.

35.2.1 Projekció

Vegyük a következő egyszerű „osztály-hieraarchiát”:

```
class Address
{
      public string Country { get; set; }
      public int PostalCode { get; set; }
      public int State { get; set; }
      public string City { get; set; }
      public string Street { get; set; }
}

class Customer
{
      public int ID { get; set; }
      public string FirstName { get; set; }
      public string LastName { get; set;}
      public string Email { get; set; }
      public string PhoneNumber { get; set; }
      public Address Address { get; set; }
}
```

Minden vásárlóhoz tartozik egy *Address* objektum, amely a vevő címét tárolja. Tegyük fel, hogy egy olyan lekérdezést akarok írni, amely visszaadja az összes vevő nevét, email címét és telefonszámát. Ez nem egy bonyolult dolog, a kód a következő lesz:

```
var result = from customer in custList select customer;

foreach(var customer in result)
{
      Console.WriteLine("Név: {0}, Email: {1}, Telefon: {2}",
            customer.FirstName + " " + customer.LastName,
            customer.Email, Customer.PhoneNumber);
}
```

A probléma a fenti kóddal, hogy a szükséges adatokon kívül megkaptuk az egész objektumot beleértve a cím példányt is amelyre pedig semmi szükségünk nincsen. A megoldás, hogy az eredeti eredményhalmazt leszűkítjük, úgy, hogy a lekérdezésben készítünk egy névtelen osztályt, amely csak a kért adatokat tartalmazza. Ezt projekciónak nevezzük. Az új kód:

```
var result = from customer in custList select new
{
      Name = customer.FirstName + Customer.LastName,
      Email = customer.Email,
      Phone = customer.PhoneNumber
};
```

Természetesen nem csak névtelen osztályokkal dolgozhatunk ilyen esetekben, hanem készíthetünk specializált direkt erre a célra szolgáló osztályokat is.

A lekérdezés eredményének típusát eddig nem jelöltük, helyette a *var* kulcsszót használtuk, mivel ez rövidebb. Minden olyan lekérdezés, amelytől egynél több eredményt várunk (tehát nem azok az operátorok, amelyek pontosan egy elemmel térnek vissza – erről később részletesebben) *IEnumerable<T>* típusú eredményt (vagy annak leszármazott, specializált változatát) ad vissza.

35.2.2 Let

A *let* segítségével – a lekérdezés hatókörén belüli – változókat hozhatunk létre, amelyek segítségével elkerülhetjük egy kifejezés ismételt felhasználását. Nézzünk egy példát:

```
string[] poem = new string[]
{
      "Ej mi a kõ! tyúkanyó, kend",
      "A szobában lakik itt bent?",
      "Lám, csak jó az isten, jót ád,",
      "Hogy fölvitte a kend dolgát!"
};

var result = from line in poem
                        let words = line.Split(' ')
                              from word in words
                                    select word;
```

A *let* segítségével minden sorból egy újabb string-tömböt készítettünk, amelyeken egy belső lekérdezést futattunk le.

35.3 Szűrés

Nyílván nincs szükségünk mindig az összes elemre, ezért képesnek kell lennünk szűrni az eredménylistát. A legalapvetőbb ilyen művelet a *where*, amelyet a következő sablonnal írhatunk le:

from azonosító *in* kifejezés *where kifejezés select* azonosító

Nézzünk egy egyszerű példát:

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

class Program
{
      static public void Main()
      {
            List<int> list = new List<int>()
            {
                  12, 4, 56, 72, 34, 0, 89, 22
            };

            var result1 = from number in list
                                    where number > 30
```

```
                                        select number;

            var result2 = list.Where(number => number > 30);

            var result3 = (from number in list select number)
                                    .Where(number => number > 30);

      }
}
```

A forrásban mindhárom lekérdezés szintaxist láthatjuk, mind pontosan ugyanazt fogja visszaadni és teljesen szabályosak.

A *where* egy paraméterrel rendelkező, *bool* visszatérési értékű metódust (anonim metódust, lambda kifejezést, stb...) vár paramétereként:

```
Func<int, bool> predicate = (x) => x > 30;

var result1 = from number in list
                        where predicate(number)
                              select number;

var result2 = list.Where(predicate);
```

A *where* feltételeinek megfelelő elemek nem a lekérdezés hívásakor kerülnek az eredménylistába, hanem akkor, amikor ténylegesen felhasználjuk őket, ezt elhalasztott végrehajtásnak (deferred execution) nevezzük (ez alól kivételt jelent, ha a lekérdezés eredményén azonnal meghívjuk pl. a *ToList* metódust, ekkor az elemek szűrése azonnal megtörténik).
A következő példában teszteljük a fenti állítást:

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

class Program
{
      static public void Main()
      {
            List<int> list = new List<int>()
            {
                  12, 4, 56, 72, 34, 0, 89, 22
            };

            Func<int, bool> predicate = (x) =>
                  {
                        Console.WriteLine("Szűrés...");
                        return x > 30;
                  };

            Console.WriteLine("Lekérdezés előtt...");

            var result = from number in list
                                    where predicate(number)
                                          select number;

            Console.WriteLine("Lekérdezés után...");

            foreach(var item in result)
            {
```

```
                Console.WriteLine("{0}", item);
            }
        }
}
```

A kimenet pedig ez lesz:

```
Lekérdezés előtt...
Lekérdezés után...
Szűrés...
Szűrés...
Szűrés...
56
Szűrés...
72
Szűrés...
34
Szűrés...
Szűrés...
89
Szűrés...
```

Jól látható, hogy a *foreach* ciklus váltotta ki a szűrő elindulását.

A *where* két alakban létezik, az elsőt már láttuk, most nézzük meg a másodikat is:

```
var result = list.Where((number, index) => index % 2 == 0);
```

Ez a változat két paramétert kezel, ahol index az elem indexét jelöli, természetesen nullától számozva. A fenti lekérdezés a páros indexű elemeket választja ki.

35.4 Rendezés

A lekérdezések eredményét könnyen rendezhetjük az *orderby* utasítással, a lekérdezés sablonja ebben az esetben így alakul:

from azonosító *in* kifejezés *where* kifejezés *orderby tulajdonság ascending/descending select* kifejezés

Az első példában egyszerűen rendezzük egy számokból álló lista elemeit:

```
var result1 = list.OrderBy(x => x); // növekvő sorrend

var result2 = from number in list
                        orderby number ascending
                              select number; // szintén növekvő

var result3 = from number in list
                        orderby number descending
                              select number; // csökkenő sorrend
```

A rendezés a gyorsrendezés algoritmusát használja.

Az elemeket több szinten is rendezhetjük, a következő példában neveket fogunk sorrendbe rakni, mégpedig úgy, hogy az azonos kezdőbetűvel rendelkezőket tovább rendezzük a név második karaktere alapján:

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

class Program
{
      static public void Main()
      {
            List<string> names = new List<string>()
            {
                  "István", "Iván", "Judit", "Jolán", "Jenő", "Béla",
                  "Balázs", "Viktória", "Vazul", "Töhötöm", "Tamás"
            };

            var result1 = names.OrderBy(name => name[0])
                               .ThenBy(name => name[1]);

            var result2 = from name in names
                               orderby name[0], name[1]
                                     select name;

            foreach(var item in result2) { Console.WriteLine(item); }
      }
}
```

Az Extension Method formában a *ThenBy* volt segítségünkre, míg a Query szintaxissal csak annyi a dolgunk, hogy az alapszabály mögé írjuk a további kritériumokat. Egy lekérdezés pontosan egy *orderby/OrderBy*–t és bármennyi *ThenBy*–t tartalmazhat.

Az *OrderBy* metódus egy másik változata két paramétert fogad, az első a rendezés alapszabálya, míg második paraméterként megadhatunk egy tetszőleges *IComparer<T>* interfészt megvalósító osztályt.

35.5 Csoportosítás

Lehetőségünk van egy lekérdezés eredményét csoportokba rendezni a *group by/GroupBy* metódus segítségével. A sablon ebben az esetben így alakul:

from azonosító *in* kifejezés *where* kifejezés *orderby* tulajdonság *ascending/descending group kifejezés by kifejezés into azonosító select* kifejezés

Használjuk fel az előző fejezetben elkészített program neveit és rendezzük őket csoportokba a név első betűje szerint:

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

class Program
{
      static public void Main()
      {
            List<string> names = new List<string>()
            {
                  "István", "Iván", "Judit", "Jolán", "Jenõ", "Béla",
                  "Balázs", "Viktória", "Vazul", "Töhötöm", "Tamás"
            };

            var result1 = names.OrderBy(name => name[0])
                               .GroupBy(name => name[0]);

            var result2 = from name in names
                               orderby name[0]
                                     group name by name[0]
                                           into namegroup
                                                 select namegroup;

            foreach(var group in result1)
            {
                  Console.WriteLine(group.Key);

                  foreach(var name in group)
                  {
                        Console.WriteLine("-- {0}", name);
                  }
            }
      }
}
```

A kimenet a következő lesz:

```
B
-- Béla
-- Balázs
I
-- István
-- Iván
J
-- Judit
-- Jolán
-- Jenő
T
-- Töhötöm
-- Tamás
V
-- Viktória
-- Vazul
```

A csoportosításhoz meg kell adnunk egy kulcsot, ez lesz az *OrderBy* paramétere. Az eredmény típusa a következő lesz:

IEnumerable<IGrouping<TKey, TElement>>

Az *IGrouping* interfész tulajdonképpen maga is egy *IEnumerable<T>* leszármazott kiegészítve a rendezéshez használt kulccsal, vagyis lényegében egy lista a listában típusú „adatszerkezetről” van szó.

35.5.1 Null értékek kezelése

Előfordul, hogy olyan listán akarunk lekérdezést végrehajtani, amelynek bizonyos indexein null érték van. A *Select* képes kezelni ezeket az eseteket, egyszerűen null értéket tesz az eredménylistába, de amikor rendezünk, akkor szükségünk van az objektum adataira és ha null értéket akarunk vizsgálni akkor gyorsan kivételt kaphatunk. Használjuk fel az előző programok listáját, egészítsük ki néhány null értékkel és írjuk át a lekérdezést, hogy kezelje őket:

```
var result1 = names.GroupBy(name =>
      {
            return name == null ? '0' : name[0];
      });

var result2 = from name in names
                              group name by
                                    name == null ? '0' : name[0]
                                          into namegroup
                                                select namegroup;

foreach(var group in result2)
{
      Console.WriteLine(group.Key);
      foreach(var name in group)
      {
            Console.WriteLine("-- {0}", name == null ? "null" : name);
      }
}
```

Többféle megoldás is létezik erre a problémára, használhattuk volna a ?? operátort is, vagy akár készíthetünk egy metódust, amely a megfelelő alapértelmezett értéket adja vissza, a lényeg, hogy amikor a listában előfordulhatnak null értékek akkor azt figyelembe kell venni a lekérdezés megírásakor. Ez akkor is számít, ha egyébként a lekérdezés nem végez műveleteket az elemekkel, hanem „csak” kiválasztást végzünk, hiszen az eredménylistában attól még ott lesznek a null elemek amelyek később fejfájást okozhatnak.

35.5.2 Összetett kulcsok

Kulcs meghatározásánál lehetőségünk van egynél több értéket kulcsként megadni, ekkor névtelen osztályként kell azt definiálni. Használjuk fel a korábban megismert *Customer* illetve *Address* osztályokat, ezeket, illetve a hozzájuk tartozó listákat a jegyzet mellé csatolt forráskódok közül a Data.cs file-ban találja az olvasó.

```
class Address
{
	public string Country { get; set; }
	public int PostalCode { get; set; }
	public int State { get; set; }
	public string City { get; set; }
	public string Street { get; set; }
}

class Customer
{
	public int ID { get; set; }
	public string FirstName { get; set; }
	public string LastName { get; set;}
	public string Email { get; set; }
	public string PhoneNumber { get; set; }
	public Address Address { get; set; }
}
```

A lekérdezést pedig a következőképpen írjuk meg:

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

class Program
{
	static public void Main()
	{
		var result = from customer in DataClass.GetCustomerList()
				group customer by
					new
					{
						customer.Address.Country,
						customer.Address.State
					}
						into customergroup
							select customergroup;

		foreach(var group in result)
		{
			Console.WriteLine("{0}, {1}",
				group.Key.Country, group.Key.State);

			foreach(var customer in group)
			{
				Console.WriteLine("-- {0}",
					customer.FirstName + " " + customer.LastName);
			}
		}
	}
}
```

Fordítani így tudunk:

```
csc main.cs Data.cs
```

35.6 Listák összekapcsolása

A relációs adatbázisok egyik alapköve, hogy egy lekérdezésben több táblát összekapcsolhatunk (join) egy lekérdezéssel, pl. egy webáruházban a vevő-árú-rendelés adatokat. Memóriában lévő gyűjtemények esetében ez viszonlag ritkán szükséges, de a LINQ To Objects támogatja ezt is.
A „join"–műveletek azon az egyszerű feltételen alapulnak, hogy az összekapcsolandó objektum-listák rendelkeznek közös adattagokkal (relációs adatbázisoknál ezt elsődleges kulcs (primary key) – idegen kulcs (foreign key) kapcsolatként kezeljük). Nézzünk egy példát:

```
class Customer
{
      public int ID { get; set; }
      public string FirstName { get; set; }
      public string LastName { get; set;}
      public string Email { get; set; }
      public string PhoneNumber { get; set; }
      public Address Address { get; set; }
}

class Order
{
      public int CustomerID { get; set; }
      public int ProductID { get; set; }
      public DateTime OrderDate { get; set; }
      public DateTime? DeliverDate { get; set; }
      public string Note { get; set; }
}

class Product
{
      public int ID { get; set; }
      public string ProductName { get; set; }
      public int Quantity { get; set; }
}
```

Ez a fent említett webáruház egy egyszerű megvalósítása. Mind a *Customer* mind a *Product* osztály rendelkezik egy *ID* nevű tulajdonsággal, amely segítségével egyértelműen meg tudjuk különböztetni őket, ez lesz az elsődleges kulcs (tegyük fel, hogy egy listában egy példányból – és így kulcsból – csak egy lehet). Az *Order* osztály mindkét példányra tartalmaz referenciát (hiszen minden rendelés egy vevő-termék párost igényel), ezek lesznek az idegen kulcsok. Írjunk egy lekérdezést, amely visszaadja minden vásárló rendelését.
Elsőként írjuk fel a join–nal felszerelt lekérdezések sablonját:

from azonosító *in* kifejezés *where* kifejezés *join azonosító in kifejezés on kifejezés equals kifejezés into azonosító orderby* tulajdonság *ascending/descending group* kifejezés *by* kifejezés *into* azonosító *select* kifejezés

Most pedig jöjjön a lekérdezés (az adatokat most is a Data.cs tárolja):

```
var result = from order in DataClass.GetOrderList()
             join customer in DataClass.GetCustomerList()
             on order.CustomerID equals customer.ID
             select new
             {
                   Name = customer.FirstName + " " + customer.LastName,
                   Products = DataClass.GetProductList()
                                       .Where(order.ProductID == product.ID)
             };

foreach(var orders in result)
{
      Console.WriteLine(orders.Name);

      foreach(var product in orders.Products)
      {
            Console.WriteLine("-- {0}", product.ProductName);
      }
}
```

A join tipikusan az a lekérdezés típus, ahol az SQL-szerű szintaxis olvashatóbb, ezért itt csak ezt írtuk meg. A lekérdezés nagyon egyszerű, elsőként csatoltuk az elsődleges kulcsokat tartalmazó listát az idegen kulccsal rendelkező listához, majd megmondtuk, hogy milyen feltételek szerint párosítsa az elemeket (itt is használhatunk összetett kulcsokat ugyanúgy névtelen osztály készítésével). Az eredménylistát persze ki kell egészítenünk a vásárolt termékek listájával, ezért egy belső lekérdezést is írtunk. A kimenet a következő lesz:

```
Istvan Reiter
-- Elosztó
Istvan Reiter
-- Papír zsebkendő
József Fekete
-- Elektromos vízforraló
```

35.7 Outer join

Az előző fejezetben azokat az elemeket választottuk ki, amelyek kapcsolódtak egymáshoz, de gyakran van szükségünk azokra is, amelyek éppen hogy nem kerülnének bele az eredménylistába, pl. azokat a „vásárlókat” keressük, akik eddig még nem rendeltek semmit. Ez a feladat a *join* egy speciális outer join–nak nevezett változata. A LINQ To Objects bár közvetlenül nem támogatja ezt (pl. az SQL tartalmaz OUTER JOIN „utasítást”), de könnyen szimulálhatjuk a *DefaultIfEmpty* metódus használatával, amely egyszerűen egy null elemet helyez el az eredménylistában az összes olyan elem helyén, amely nem szerepelne a „hagyományos” *join* által kapott lekérdezésben.

A következő példában a lekérdezés visszaadja a vásárlók rendelésének a sorszámát (most nincs szükségünk a *Products* listára), vagy ha még nem rendelt, akkor megjelenítünk egy „nincs rendelés” feliratot.:

```
var result = from customer in DataClass.GetCustomerList()
             join order in DataClass.GetOrderList()
             on customer.ID equals order.CustomerID into tmpresult
             from o in tmpresult.DefaultIfEmpty()
             select new
             {
                   Name = customer.FirstName + " " + customer.LastName,
                   Product = o == null ?
                        "nincs rendelés" : o.ProductID.ToString()
             };

foreach(var order in result)
{
      Console.WriteLine("{0}: {1}",
            order.Name, order.Product);
}
```

35.8 Konverziós operátorok

A LINQ To Objects lehetőséget ad listák konverziójára a következő operátorok segítségével:

OfType és **Cast**: ők a „sima” *IEnumerable* interfészről konvertálnak generikus megfelelőikre, elsődlegesen a .NET 1.0–val való kompatibilitás miatt léteznek, hiszen a régi *ArrayList* osztály nem valósítja meg a generikus *IEnumerable*–t, ezért kasztolni kell, ha LTO –t akarunk használni.
A kettő közti különbséget a hibakezelés jelenti: az *OfType* egyszerűen figyelmen kívül hagyja a konverziós hibákat és a nem megfelelő elemeket kihagyja az eredménylistából, míg a *Cast* kivételt (*System.InvalidCastException*) dob:

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

class Program
{
      static public void Main()
      {
            ArrayList list = new ArrayList();
            list.Add("alma");
            list.Add("dió");
            list.Add(12345);

            var result1 = from item in list.Cast<string>()
                          select item;

            var result2 = from item in list.OfType<string>()
                          select item;

            foreach(var item in result1)
            {
                  Console.WriteLine(item); // kivétel
            }
      }
}
```

A program ugyan kiírja a lista első két elemét, de a harmadiknál már kivételt dob, tehát a konverzió elemenként történik, mégpedig akkor, amikor az eredménylistát ténylegesen felhasználjuk nem pedig a lekérdezésnél.

ToArray, **ToList**, **ToLookup**, **ToDictionary**: ezek a metódusok, ahogyan a nevükből is látszik az *IEnumerable<T>* eredménylistát tömbbé vagy generikus gyűjteménnyé alakítják. A következő példában a *ToList* metódus látható:

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

class Program
{
      static public void Main()
      {
            List<int> list = new List<int>()
            {
                  10, 32, 45, 2, 55, 32, 21
            };

            var result = (from number in list
                                    where number > 20
                                          select number).ToList<int>();

            result.ForEach((number) => Console.WriteLine(number));
      }
}
```

Amikor ezeket az operátorokat használjuk akkor a *Where* végrehajtása nem tolódik el, hanem azonnal kiszűri a megfelelő elemeket, vagyis itt érdemes figyelni a teljesítményre.
A *ToArray* metódus értelemszerűen tömbbé konvertálja a bemenő adatokat. A *ToDictionary* és *ToLookup* metódusok hasonló feladatot látnak el abban az értelemben, hogy mindkettő kulcs-érték párokkal operáló adatszerkezetet hoz létre. A különbség a duplikált kulcsok kezelésében van, míg a *Dictionary<T, V>* szerkezet ezeket nem engedélyezi (sőt kivételt dob), addig a *ToLookup ILookup<T, V>* szerkezetet ad vissza, amelyben az azonos kulccsal rendelkező adatok listát alkotnak a listán belül, ezért őt kiválóan alkalmazhatjuk a join műveletek során. A következő példában ezt a metódust használjuk, hogy a vásárlókat megye szerint rendezze:

```
var result = (from customer in DataClass.GetCustomerList()
                        select customer)
                  .ToLookup((customer) => customer.Address.State + " megye");

foreach(var item in result)
{
      Console.WriteLine(item.Key);

      foreach(var customer in item)
      {
            Console.WriteLine("-- {0} {1}",
                  customer.FirstName, customer.LastName);
      }
}
```

A *ToLookUp* paramétereként a lista kulcsértékét várja.

AsEnumerable: Ez az operátor a megfelelő *IEnumerable<T>* típusra konvertálja vissza a megadott *IEnumerable<T>* interfészt megvalósító adatszerkezetet.

35.9 „Element" operátorok

A következő operátorok megegyeznek abban, hogy egyetlen elemmel (vagy egy előre meghatározott alapértékkel) térnek vissza az eredménylistából.

First/Last és **FirstOrDefault/LastOrDefault**: ezeknek a metódusoknak két változata van: a paraméter nélküli az első/utolsó elemmel tér vissza a listából, míg a másik egy *Func<T, bool>* típusú metódust kap paraméternek és e szerint a szűrő szerint választja ki az első elemet. Amennyiben nincs a feltételnek megfelelő elem a listában (vagy üres listáról beszélünk) akkor az első két operátor kivételt dob, míg a másik kettő az elemek típusának megfelelő alapértelmezett null értékkel tér vissza (pl. *int* típusú elemek listájánál ez nulla míg *string*ek esetén null lesz):

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

class Program
{
	static public void Main()
	{
		List<int> list = new List<int>()
		{
			10, 3, 56, 67, 4, 6, 78, 44
		};

		var result1 = list.First(); // 10
		var result2 = list.Last(); // 44

		var result3 = list.First((item) => item > 10); // 56

		try
		{
			var result4 = list.Last((item) => item < 3); // kivétel
		}
		catch(Exception e)
		{
			Console.WriteLine(e.Message);
		}

		var result5 = list.FirstOrDefault((item) => item < 3); // 0

		Console.WriteLine("{0}, {1}, {2}, {3}",
			result1, result2, result3, result5);
	}
}
```

Single/SingleOrDefault: ugyanaz, mint a *First/FirstOrDefault* páros, de mindkettő kivételt dob, ha a feltételnek több elem is megfelel.

ElementAt/ElementAtOrDefault: visszaadja a paraméterként átadott indexen található elemet. Az első kivételt dob, ha az index nem megfelelő a másik pedig a megfelelő alapértelmezett értéket adja vissza.

DefaultIfEmpty: a megfelelő alapértelmezett értékkel tér vissza, ha egy lista nem tartalmaz elemeket, őt használtuk korábban a join műveleteknél.

35.10 Halmaz operátorok

Nézzük a halmaz operátorokat, amelyek két lista közötti halmazműveleteket tesznek lehetővé.

Concat és **Union**: mindkettő képes összefűzni két lista elemeit, de az utóbbi egy elemet csak egyszer tesz át az új listába:

```
List<int> list1 = new List<int>()
{
      10, 3, 56, 67, 4, 6, 78, 44
};

List<int> list2 = new List<int>()
{
      10, 5, 67, 89, 3, 22, 99
};

var result1 = list1.Concat(list2);
/*
      10, 3, 56, 67, 4, 6, 78, 44, 10, 5, 67, 89, 3, 22, 99
*/

var result2 = list1.Union(list2);
/*
      10, 3, 56, 67, 4, 6, 78, 44, 5, 89, 22, 99
*/
```

Intersect és **Except**: az első azokat az elemeket adja vissza, amelyek mindkét listában szerepelnek, míg a második azokat amelyek csak az elsőben:

```
List<int> list1 = new List<int>()
{
      10, 3, 56, 67, 4, 6, 78, 44
};

List<int> list2 = new List<int>()
{
      10, 5, 67, 89, 3, 22, 99
};

var result1 = list1.Intersect(list2);
/*
      10, 3, 67
*/

var result2 = list1.Except(list2);
/*
      56, 4, 6, 78, 44
*/
```

35.11 Aggregát operátorok

Ezek az operátorok végigárnak egy listát, elvégeznek egy műveletet minden elemen, és végeredményként egyetlen értéket adnak vissza (pl. elemek összege vagy átlagszámítás).

Count és **LongCount**: visszaadják az elemek számát egy listában. Alapértelmezés szerint az összes elemet számolják, de megadhatunk feltételt is. A két operátor közötti különbség az eredmény nagyságában van, a *Count* 32 bites egész számmal (*int*), míg a *LongCount* 64 bites egész számmal (*int64*) tér vissza:

```
List<int> list = new List<int>()
{
      10, 3, 56, 67, 4, 6, 78, 44
};

var result1 = list.Count(); // 8
var result2 = list.Count((item) => item > 10); // 4
```

Min és **Max**: a lista legkisebb illetve legnagyobb elemét adják vissza. Mindkét operátornak megadhatunk egy szelektor kifejezést, amelyet az összes elemre alkalmaznak, és e szerint választják ki a megfelelő elemet:

```
List<int> list = new List<int>()
{
      10, 3, 56, 67, 4, 6, 78, 44
};

var result1 = list.Max(); // 78
var result2 = list.Max((item) => item % 3); // 2
```

Természetesen a második eredmény maximum kettő lehet, hiszen hárommal való osztás után ez lehet a legnagyobb maradék.
Mindkét metódus az *IComparable<T>* interfészt használja, így minden ezt megvalósító típuson használhatóak.

Average és **Sum**: a lista elemeinek átlagát illetve összegét adják vissza. Ők is rendelkeznek szelektor kifejezést használó változattal:

```
List<int> list = new List<int>()
{
      10, 3, 56, 67, 4, 6, 78, 44
};

var result1 = list.Sum(); // 268
var result2 = list.Average(); // 33.5
var result3 = list.Sum((item) => item * 2); // 536
```

Aggregate: ez az operátor lehetővé teszi tetszőleges művelet elvégzését és a részeredmények „felhalmozását”. Az *Aggregate* három formában létezik, tetszés szerint megadhatunk neki egy kezdőértéket, illetve egy szelektor kifejezést is:

```
List<int> list = new List<int>()
{
      1, 2, 3, 4
};

var sum = list.Aggregate((result, item) => result + item);
var max = list.Aggregate(-1, (result, item) => item > result ? item :
result);
var percent = list.Aggregate(0.0, (result, item) => result + item,
result => result / 100);
```

Az első esetben a *Sum* operátort szimuláltuk, ezt nem kell magyarázni. A második változatban maximumkeresést végeztünk, itt megadtunk egy kezdőértéket, amelynél biztosan van nagyobb elem a listában. Végül a harmadik műveletnél kiszámoltuk a számok összegének egy százalékát (itt figyelni kell arra, hogy *double* típusként lássa a fordító a kezdőértéket, hiszen tizedes törtet akarunk visszakapni).
A végeredményt tároló „változó” bármilyen típus lehet, még tömb is.

35.12 PLINQ – Párhuzamos végrehajtás

Napjainkban már egyáltalán nem jelentenek újdonságot a több maggal rendelkező processzorok, így teljesen jogosan jelentkezett az igény, hogy minél jobban ki tudjuk ezt használni. A .NET 4.0 bevezeti nekünk a Parallel Task Library–t és a jelen fejezet tárgyát a Parallel–LINQ–t, vagyis a lekérdezések párhuzamosításának lehetőségét.

35.12.1 Többszálúság vs. Párhuzamosság

Amikor többszálú programokat készítünk alapvetően nem törődünk a hardver lehetőségeivel, létrehozunk szálakat amelyek versengnek a processzoridőért. Ilyenkor értelemszerűen nem fogunk különösebb teljesítménynövekedést kapni, hiszen minden művelet ugyanannyi ideig tart, nem tudjuk őket közvetlenül szétosztani az – esetleges – több processzor között. Ezt a módszert pl. olyankor használjuk, amikor nem akarjuk, hogy a háttérben futó szálak megzavarják a „főszál” kezelhetőségét (pl. ha egy böngészőben több oldalt is megnyitunk, nem akarjuk megvárni amíg mindegyik letöltődik), vagy egyszerre több „kérést” kell kezelnünk (pl. egy kliens-szerver kapcsolat).
A párhuzamos programozás ugyanezt képes nyújtani, de képes a processzorok számának függvényében szétosztani a munkát a CPU –k között, ezzel pedig teljesítménynövekedést ér el. Ennek a módszernek a nagy hátránya, hogy olyan algoritmust kell találjunk, amely minden helyzetben a lehető leghatékonyabban tudja elosztani a munkát, anékül, hogy bármely processzor üresjáratban álljon.

35.12.2 Teljesítmény

Nagyon könnyen azt gondolhatjuk, hogy a processzorok számának növelésével egyenes arányban nő a teljesítmény, magyarul két processzor kétszer gyorsabb, mint egy. Ez az állítás nem teljesen igaz (ezt később a saját szemünkkel is látni fogjuk), ezt pedig Gene Amdahl bizonyította (Amdahl’s Law), miszerint:

Egy párhuzamos program maximum olyan gyors lehet, mint a leghosszabb szekvenciális (tovább már nem párhuzamosítható) részegysége.

Vegyünk például egy programot, amely 10 órán keresztül fut nem párhuzamosan. Ennek a programnak 9 órányi „része” párhuzamosítható, míg a maradék egy óra nem. Ha ezt a 9 órát párhuzamosítjuk, akkor a tétel alapján a programnak így is minimum egy órás futásideje lesz.
Amdahl a következő képletet találta ki:

$$\frac{1}{(1-P) + P/N}$$

Ahol P a program azon része, amely párhuzamosítható, míg (1 – P) az amelyik nem, N pedig a processzorok száma.
Nézzük meg, hogy mekkora a maximum teljesítmény, amit a fenti esetből ki tudunk préselni. P ekkor 0,9 lesz (9 óra = 90% = 0,9), és a képlet (két processzort használva):

$$\frac{1}{(1-0{,}9) + 0{,}9/2}$$

Könnyen kiszámolható, hogy az eredmény 1/0,55 (1,81) lesz vagyis 81% -os teljesítménynövekedést érhetünk el két processzor bevezetésével. Vegyük észre, hogy a processzorok számának növelésével P/N a nullához tart, vagyis kimondhatjuk, hogy minden párhuzamosítható feladat maximum 1/(1 – P) nagyságrendű teljesítménynövekedést eredményezhet (feltételezve, hogy mindig annyi processzor áll rendelkezésünkre, hogy P/N a lehető legközelebb legyen nullához: ez nem feltétlenül jelent nagyon sokat, a példa esetében 5 processzor már 550%-os növekedést jelent, innen pedig egyre lassabban nő az eredmény, mivel ekkor P/N értéke már 0,18, hat processzornál 0,15 és így tovább), tehát a fenti konkrét esetben a maximális teljesítmény a hagyományos futásidő tízszerese lehet (1 / (1 – 0,9), vagyis pontosan az az egy óra, amelyet a nem párhuzamosítható programrész használ fel.

35.12.3 PLINQ a gyakorlatban

Ebben a fejezetben összehasonlítjuk a hagyományos és párhuzamos LINQ lekérdezések teljesítményét. Elsősorban szükségünk lesz valamilyen, megfelelően nagy adatforrásra, amely jelen esetben egy szöveges file lesz. A jegyzethez mellékelt forráskódok között megtaláljuk a DataGen.cs nevűt, amely az első paraméterként megadott fileba a második paraméterként megadott mennyiségű vezetéknév-keresztnév-kor-foglalkozás-megye adatokat ír. A következő példában tízmillió személlyel fogunk dolgozni, tehát így futassuk a programot:

```
DataGen.exe E:\Data.txt 10000000
```

Az elérési út persze tetszőleges. Most készítsük el a lekérdezést. Felhasználjuk, hogy a .NET 4.0–ban a *File.ReadLines* metódus *IEnumerable<string>*-gel tér vissza, vagyis közvetlenül hajthatunk rajta végre lekérdezést:

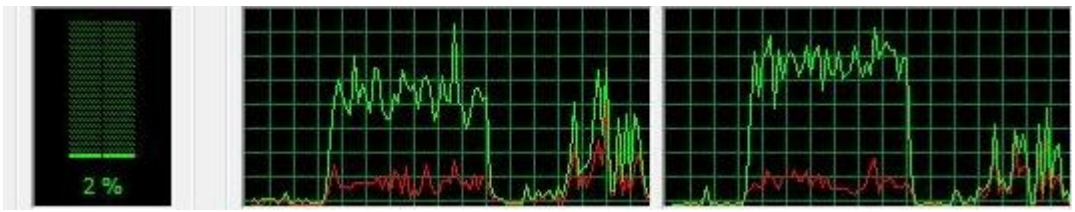
```
var lines = File.ReadLines(@"E:\Data.txt"); // System.IO kell

var result = from line in lines
                        let data = line.Split( new char[] { ' ' })
                        let name = data[1]
                        let age = int.Parse(data[2])
                        let job = data[3]
                              where name == "István" &&
                                          (age > 24 && age < 55) &&
                                          job == "börtönõr"
                                    select line;
```

Keressük az összes 24 és 55 év közötti István nevű börtönőrt. Elsőként szétvágunk minden egyes sort, majd a megfelelő indexekről (az adatok sorrendjéért látogassuk meg a DataGen.cs–t) összeszedjük a szükséges adatokat. Az eredmény írassuk ki egy egyszerű *foreach* ciklussal:

```
foreach(var line in result)
{
      Console.WriteLine(line);
}
```

A fenti lekérdezés egyelőre nem párhuzamosított, nézzük meg, hogy mi történik:



Ez a kép a processzorhasználatot mutatja, két dolog világosan látszik: 1. a két processzormag nem ugyanazt a teljesítményt adja le, 2. egyik sem teljesít maximumon.
Most írjuk át a lekérdezést párhuzamosra, ezt rendkívül egyszerűen tehetjük meg, mindössze az *AsParallel* metódust kell meghívnunk az adatforráson:

```
var result = from line in lines.AsParallel()
                        let data = line.Split( new char[] { ' ' })
                        let name = data[1]
                        let age = int.Parse(data[2])
                        let job = data[3]
                              where name == "István" &&
                                          (age > 24 && age < 55) &&
                                          job == "börtönõr"
                                    select line;
```

Lássuk az eredményt:

A kép önmagáért beszél, de a teljesítménykülönbség is, a tesztgépen átlagosan 25% volt a párhuzamos lekérdezés előnye a hagyományoshoz képest (ez természetesen mindenkinél más és más lehet).

Az *AsParallel* kiterjesztett metódus egy *ParalellQuery* objektumot ad vissza, amely megvalósítja az *IEnumerable* interfészt.

A kétoperandusú LINQ operátorokat (pl. *join*, *Concat*) csakis úgy használhatjuk párhuzamosan, ha mindkét adatforrást *AsParallel* –lel jelüljük, ellenkező esetben nem fordul le:

```
var result = list1.AsParallel().Concat(list2.AsParallel());
```

Bár úgy tűnhet, hogy nagyon egyszerűen felgyorsíthatjuk a lekérdezéseinket a valóság ennél kegyetlenebb. Ahhoz, hogy megértsük, hogy miért csak bizonyos esetekben kell párhuzamosítanunk, ismerni kell a párhuzamos lekérdezések munkamódszerét:

1. Analízis: a keretrendszer megvizsgálja, hogy egyáltalán megéri-e párhuzamosan végrehajtani a lekérdezést. Minden olyan lekérdezés, amely nem tartalmaz legalább viszonylag összetett szűrést vagy egyéb „drága” műveletet szinte biztos, hogy szekvenciálisan fog végrehajtódni (és egyúttal a lekérdezés futásidejéhez hozzáadódik a vizsgálat ideje is). Természetesen az ellenőrzést végrehajtó algoritmus sem tévedhetetlen, ezért lehetőségünk van manuálisan kikényszeríteni a párhuzamosságot:

```
var result = select x from data.AsParallel()
             .WithExecutionMode(ParallelExecutionMode.ForceParallelism);
```

2. Ha az ítélet a párhuzamosságra nézve kedvező, akkor a következő lépésben a feldolgozandó adatot a rendszer a rendelkezésre álló processzorok száma alapján elosztja. Ez egy meglehetősen bonyolult téma, az adatforrástól függően több stratégia is létezik, ezt itt nem részletezzük.
3. Végrehajtás.
4. A részeredmények összeillesztése. Erre a részre is lehet ráhatásunk, miszerint egyszerre szeretnénk a végeredményt megkapni, vagy megfelelnek a részeredmények is:

```
var result = from x in data.AsParallel()
      .WithMergeOptions(ParallelMergeOptions.NotBuffered);
```

A *ParallelMergeOptions* három taggal rendelkezik: a *NotBuffered* hatására azonnal visszakapunk minden egyes elemet a lekérdezésből, az *AutoBuffered* periódikusan

ad vissza több elemet, míg a *FullyBuffered* csak akkor küldi vissza az eredményt, ha a lekérdezés teljesen kész van.

Láthatjuk, hogy komoly számítási kapacitást igényel a rendszertől egy párhuzamos lekérdezés végrehajtása, épp ezért a nem megfelelő lekérdezések parallel módon való indítása nem fogja az elvárt teljesítményt hozni, tulajdonképpen még teljesítményromlás is lehet az eredménye. Tipikusan olyan helyzetekben akarunk ilyen lekérdezéseket használni, ahol nagy mennyiségű elemen sok vagy drága műveletet végzünk el, mivel itt nyerhetünk a legtöbbet.

35.12.4 Rendezés

Nézzük a következő kódot:

```
List<int> list = new List<int>()
{
      0, 1, 2, 3, 4, 5, 6, 7, 8, 9
};

var result1 = from x in list select x;
var result2 = from x in list.AsParallel() select x;
```

Vajon mit kapunk, ha kiíratjuk mindkét eredményt? A válasz meglepő lehet:

```
result1: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
result2: 0, 6, 3, 4, 8, 2, 5, 1, 9, 7
```

Hogyan kaphattunk egy rendezett listából rendezetlen eredményt? A válasz nagyon egyszerű, hiszen tudjuk, hogy a PLINQ részegységekre bontja az adatforrást, vagyis nem fogjuk sorban visszakapni az eredményeket (a fent mutatott eredmény nem lesz mindenkinél ugyanaz minden futásnál más sorrendet kaphatunk vissza).
Amennyiben számít az eredmény rendezettsége, akkor vagy használnunk kell az *orderby*–t, vagy az *AsParallel* mellett meg kell hívnunk az *AsOrdered* kiterjesztett metódust:

```
var result2 = from x in list.AsParallel().AsOrdered() select x;
var result3 = from x in list.AsParallel() orderby x select x;
```

A két lekérdezés hasonlónak tűnik, de nagy különbség van közöttük. Ha lemérjük a végrehajtási időt, akkor látni fogjuk, hogy az első valamivel gyorsabban végzett, mint a második, ennek pedig az az oka, hogy amíg az *orderby* mindig végrehajtja a rendezést addig az *AsOrdered* egyszerűen megőrzi az eredeti sorrendet, és aszerint osztja szét az adatokat (nagy adatmennyiséggel a kettő közötti sebességkülönbség is jelentősen megnő, érdemes néhányezer elemre is tesztelni).

Gyakran előfordul, hogy az eredeti állapot fenntartása nem előnyös számunkra, pl. kiválasztunk néhány elemet egy listából és ezek alapján akarunk egy join műveletet végrehajtani. Ekkor nem célszerű fenntartani a sorrendet, használjuk az *AsUnordered* metódust, amely minden érvényes rendezést érvénytelenít.

35.12.5 AsSequential

Az *AsSequential* metódus épp ellenkezője az *AsParallel*–nek, vagyis szabályozhatjuk, hogy egy lekérdezés mely része legyen szekvenciális és melyik párhuzamos. A következő példában megnézzük, hogy ez miért jó nekünk. A PLINQ egyelőre még nem teljesen kiforrott, annak a szabályai, hogy mely operátorok lesznek mindig szekvenciálisan kezelve a jövőben valószínűleg változni fognak, ezért a példaprogramot illik fenntartással kezelni:

```
var result = (from x in list.AsParallel() select x).Take(10);
```

Kiválasztjuk az eredménylistából az első tíz elemet. A probléma az, hogy a *Take* operátor szekvenciálisan fut majd le függetlenül attól, hogy mennyire bonyolult a „belső” lekérdezés, épp ezért nem kapunk semmiféle teljesítménynövekedést. Írjuk át egy kicsit:

```
var result2 = (from x in list.AsParallel() select x)
.AsSequential().Take(10);
```

Most pontosan azt kapjuk majd, amire várunk, a belső párhuzamos műveletsor végeztével visszaváltottunk szekvenciális módba, vagyis a *Take* nem fogja vissza a sebességet.

36 Visual Studio

A .NET Framework programozásához az első számú fejlesztőeszköz a Microsoft Visual Studio termékcsaládja. A „nagy” fizetős Professional/Ultimate/stb... változatok mellett az Express változatok ingyenes és teljes körű szolgáltatást nyújtanak számunkra.
Ebben a fejezetben a Visual Studio 2008 verzióval fogunk megismerkedni (a jegyzet írásának pillanatában már létezik a 2010 változat is, de ez egyrészt még nem annyira elterjedt, másrészt nagy különbség nincs közöttük – aki az egyiket tudja használni annak a másikkal sem lehet problémája.

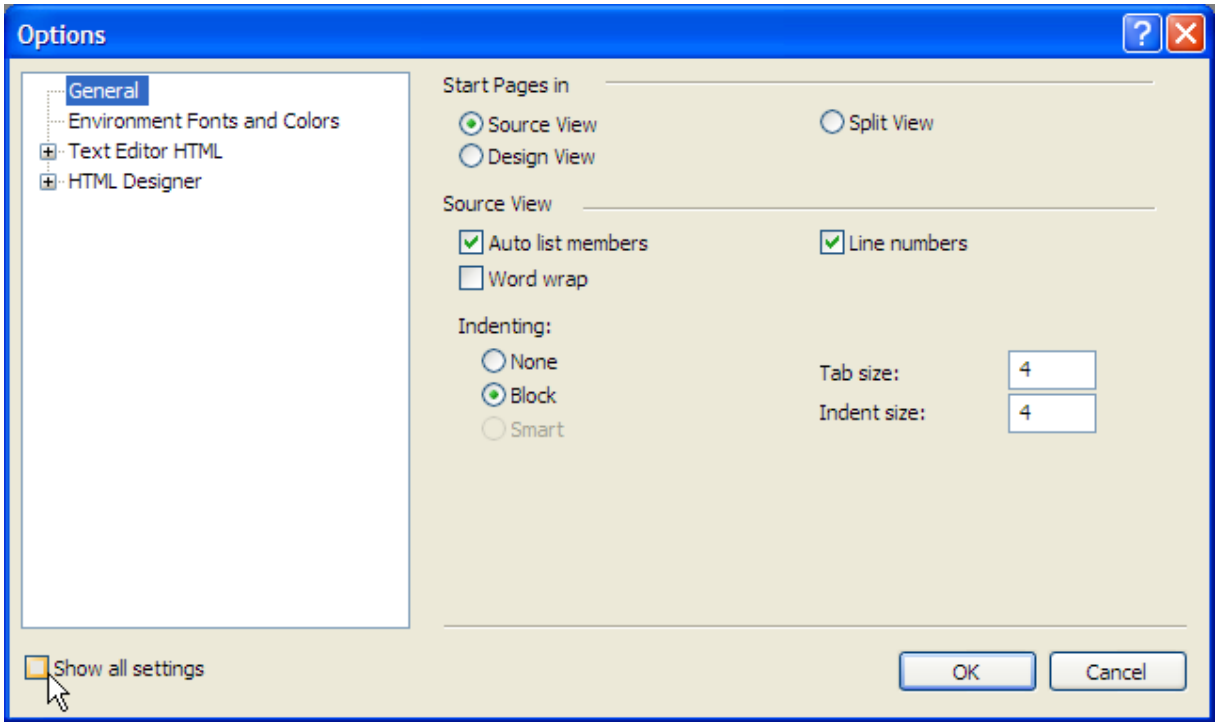
Érdemes telepíteni a Visual Studio 2008 szervízcsomagját is, ez letölthető a Microsoft oldaláról.

36.1 Az első lépések

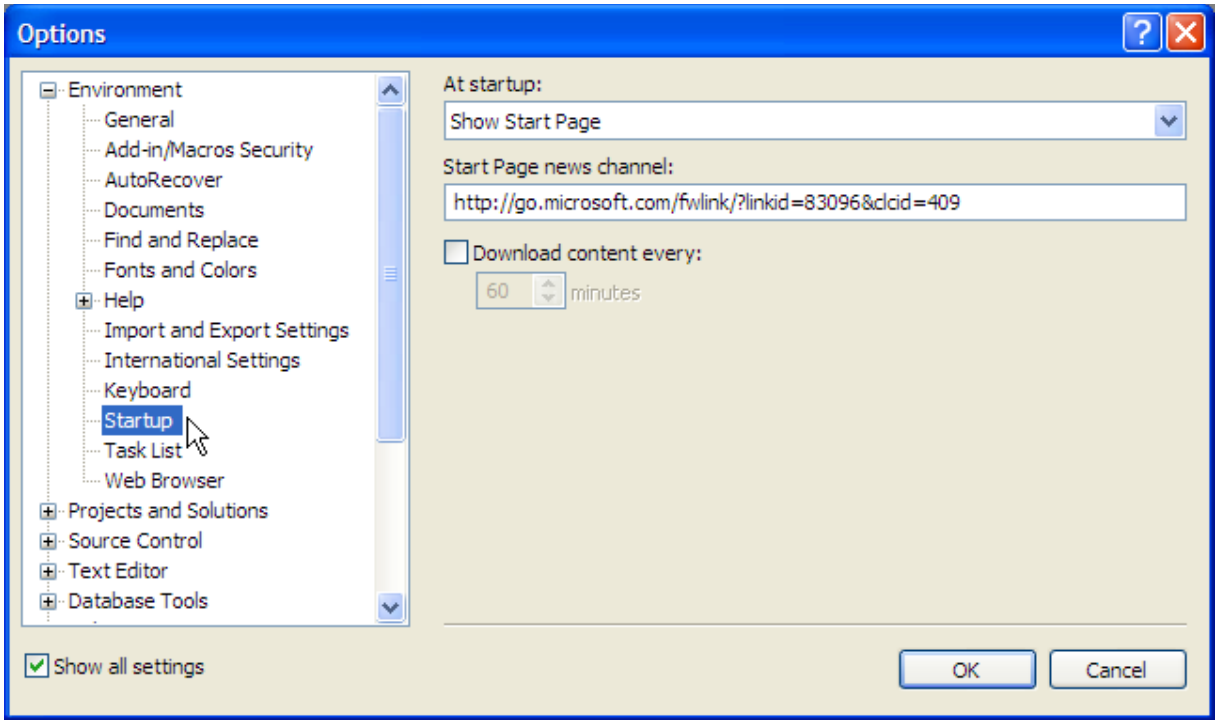
A VS 2008 telepítéséhez legalább Windows XP SP2 szükséges.

A VS első indításakor megkérdezi, hogy melyik nyelvhez tartozó beállításokkal működjön, ez a mi esetünkben a C# lesz. Ezután – alapbeállítás szerint – a kezdőlap jelenik meg amely a korábban megnyitott projectek mellett élő internetes kapcsolat esetén a fejlesztői blogok illetve hírcsatornák legfrissebb bejegyzéseit is megjeleníti.

Beállíthatjuk, hogy mi jelenjen meg indításkor, ehhez válasszuk ki a Tools menü Options pontját. Ekkor megjelenik a következő ablak:

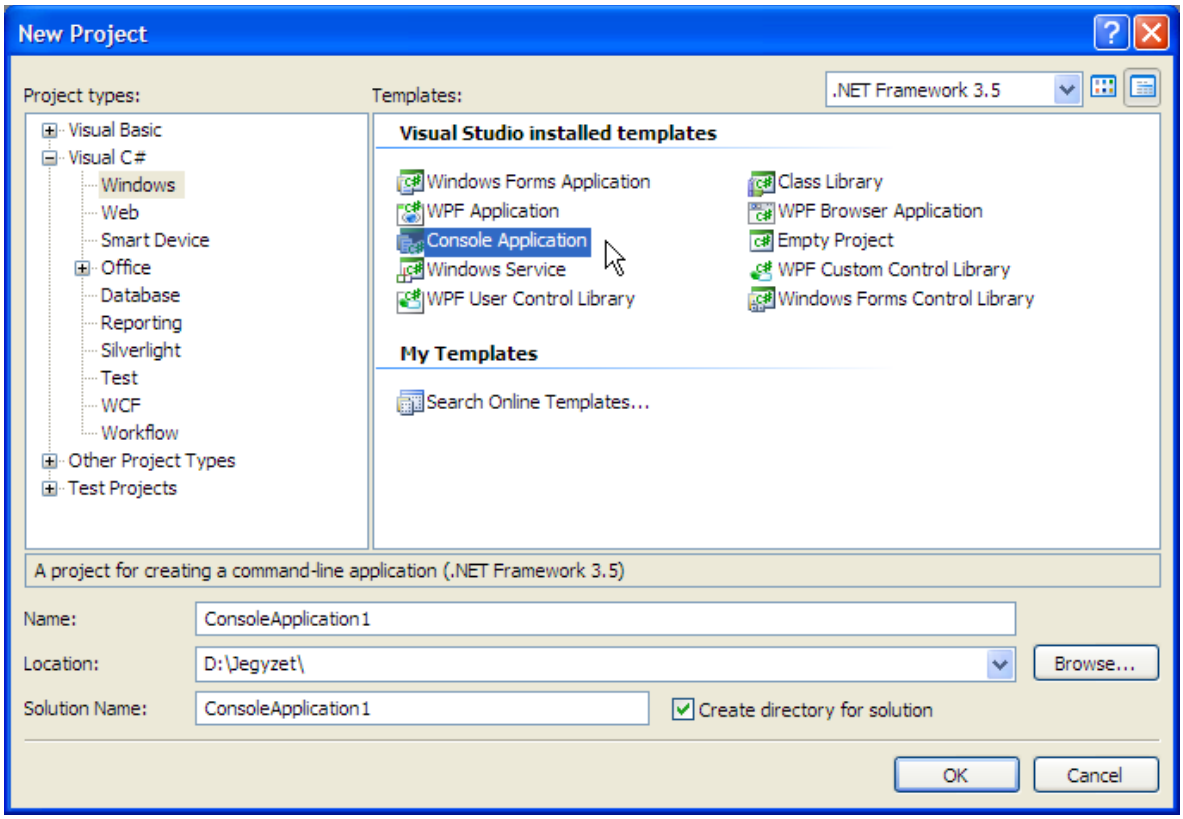


Amennyiben a Show all settings jelölőnégyzet üres, akkor jelöljük be, majd a megfelenő listából válasszuk ki a Startup–ot:



Itt beállíthatjuk, hogy mit szeretnénk látni indításkor.

Most készítsük el az első programunkat, ehhez a File menü New Project pontját válasszuk ki:



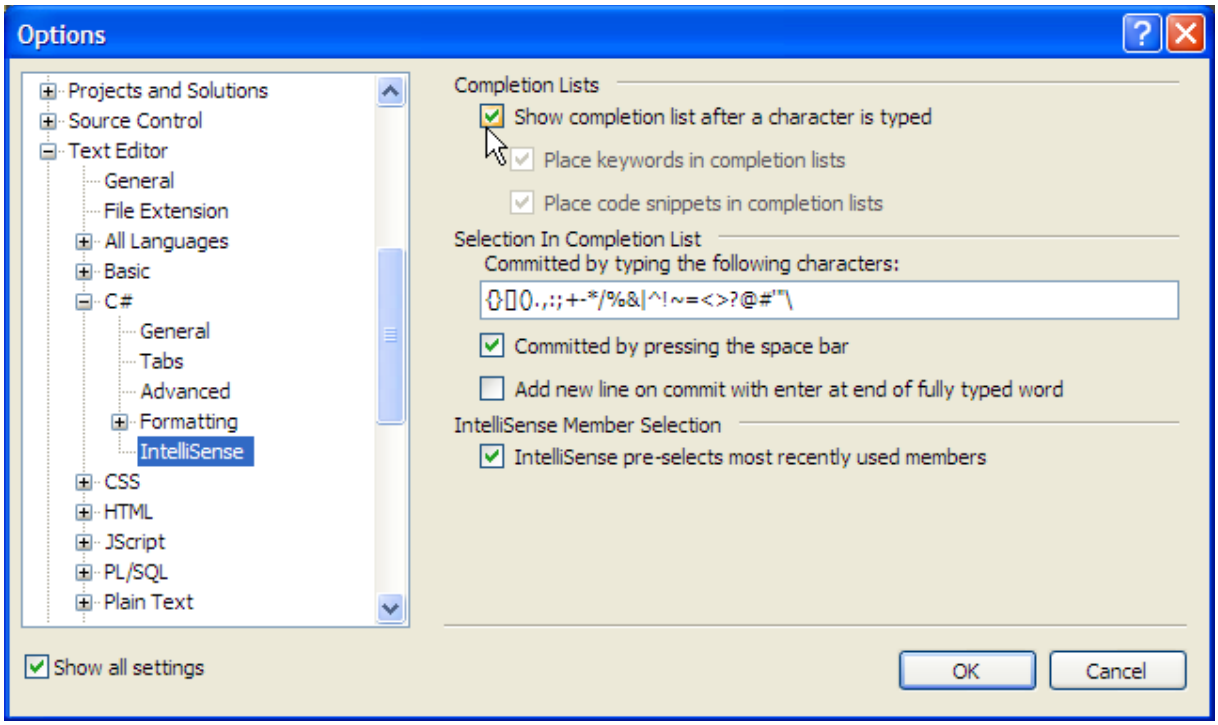
Itt a Visual C# fül alatt a Windows elem következik, ahol kiválaszthatjuk az előre elkészített sablonok közül, hogy milyen típusú projectet akarunk készíteni, ez most egy Console Application lesz, a jegyzetben eddig is ilyeneket készítettünk.

A jobb fölső sarokban beállíthatjuk, hogy a Framework melyik verziójával akarunk dolgozni. Alul pedig a project nevét és könyvtárát adhatjuk meg.

Az OK gombra kattintva elkészíthetjük a projectet. A most megjelenő forráskód valószínűleg ismerős lesz, bár néhány (eddig ismeretlen) névteret már előre beállított a VS.
Írjunk egy egyszerű Hello World! programot, ezt az F5 gomb segítségével fordíthatjuk és futtathatjuk (ne felejtsünk el egy Console.ReadKey utasítást tenni a végére, különben nem látunk majd semmit).

A Ctrl+Shift+B kombinációval csak fordítunk, míg az F5 egyszerre fordít és futtat. A nem mentett változásokat a VS mindkét esetben automatikusan menti.

Újdonságot jelenthet, hogy a Visual Studio kiegészíti a forráskódot ezzel rengeteg gépeléstől megkímélve minket, ezt IntelliSense–nek hívják. Amennyiben nem működik az (valószínűleg) azt jelenti, hogy nincs bekapcsolva. A Tools menüből válasszuk ki ismét az Options–t és azon belül pedig a Text Editor elemet. Ekkor a listából kiválaszthatjuk a C# nyelvet és azon belül pedig az IntelliSense menüt:



Ahogy a képen is látható a „Show completion list...” jelölőnégyzetet kell megjelölni.

Nézzük meg, hogy hogyan néz ki egy Visual Studio project. Nyissuk meg a könyvtárat ahová mentettük, ekkor egy .sln és egy .suo filet illetve egy mappát kell látnunk.

Az sln file a projectet tartalamzó ún. Solution–t írja le, ez lesz minden project gyökere (egy Solution tartalmazhat több projectet is). Ez a file egy egyszerű szöveges file, valami ilyesmit kell látnunk:

```
Microsoft Visual Studio Solution File, Format Version 10.00
# Visual Studio 2008
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "ConsoleApplication1",
"ConsoleApplication1\ConsoleApplication1.csproj", "{4909A576-5BF0-48AA-AB70-4B96835C00FF}"
EndProject
Global
	GlobalSection(SolutionConfigurationPlatforms) = preSolution
		Debug|Any CPU = Debug|Any CPU
		Release|Any CPU = Release|Any CPU
	EndGlobalSection
	GlobalSection(ProjectConfigurationPlatforms) = postSolution
		{4909A576-5BF0-48AA-AB70-4B96835C00FF}.Debug|Any CPU.ActiveCfg = Debug|Any CPU
		{4909A576-5BF0-48AA-AB70-4B96835C00FF}.Debug|Any CPU.Build.0 = Debug|Any CPU
		{4909A576-5BF0-48AA-AB70-4B96835C00FF}.Release|Any CPU.ActiveCfg = Release|Any CPU
		{4909A576-5BF0-48AA-AB70-4B96835C00FF}.Release|Any CPU.Build.0 = Release|Any CPU
	EndGlobalSection
	GlobalSection(SolutionProperties) = preSolution
		HideSolutionNode = FALSE
	EndGlobalSection
EndGlobal
```

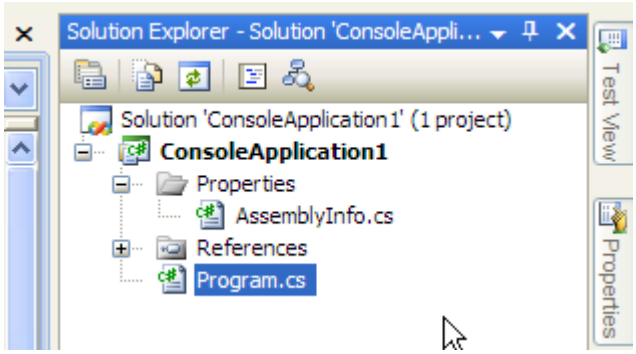
A projectek mellett a fordításra vonatkozó információk is megtalálhatóak itt. A .suo file pedig a Solution tényleges fordítási adatait kapcsolóit tartalmazza bináris formában (hexa editorral többé-kevésbé olvashatóvá válik).

Most nyissuk meg a mappát is, megtalálhatjuk benne a projectet leíró .csproj filet, amely a project fordításának információit tárolja. A Properties mappában található AssemblyInfo.cs file amelyben pl. a program verziószáma, tulajdonosa, a gyártója illetve a COM interoperabilitásra vonatkozó információk vannak. Az obj mappa a fordítás során keletkezett „melléktermékeket” tárolja, ezek lényegében átmeneti fileok a végleges program működéséhez nem szükségesek.
A bin mappában találjuk a lefordított végleges programunkat vagy a debug vagy a release mappában attól függően, hogy hogyan fordítottuk (erről később bővebben).
Ha megnézzük a mappa tartalmát, láthatjuk, hogy bár egyetlen exe kiterjesztésű filera számítottunk találunk még három ismeretlen filet is. Ezek a Visual Studio számára hordoznak információt, segítik pl. a debuggolást (erről is később).

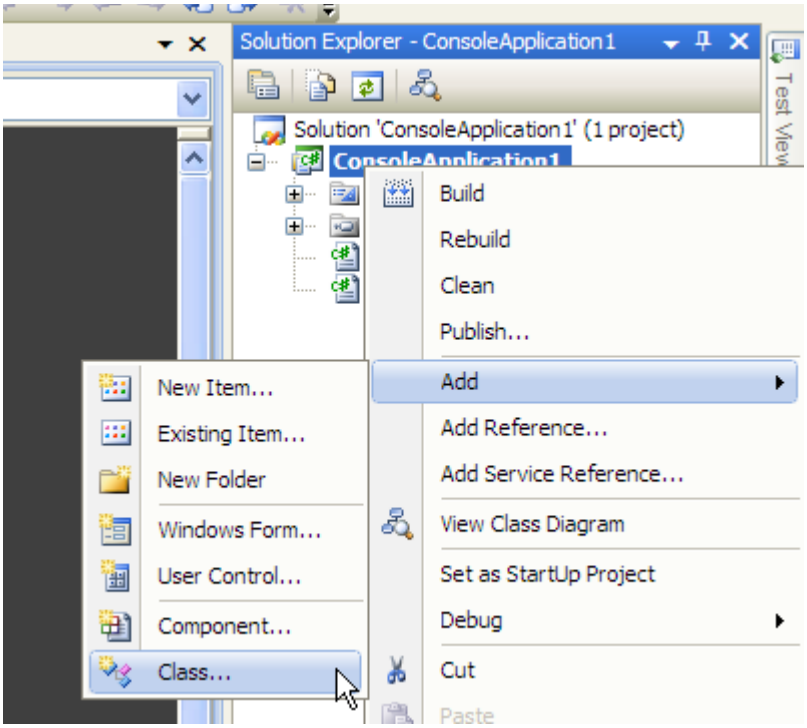
A Visual Studio 2008 hajlamos „eldobni” fileokat a projectekből, vagyis bár fizikailag jelen vannak, de a Solution Explorerben (és így fordításkor sem) nem szerepelnek. Ilyenkor két megoldás van: vagy kézzel újra fel kell venni (jobb klikk a projecten, majd Add/Existing Item), vagy a csproj filet kell szerkeszteni. Ezt a problémát megelőzhetjük rendszeres biztonsági mentéssel (nagyobb programok esetén érdemes valamilyen verziókezelő eszközt használni).

36.2 Felület

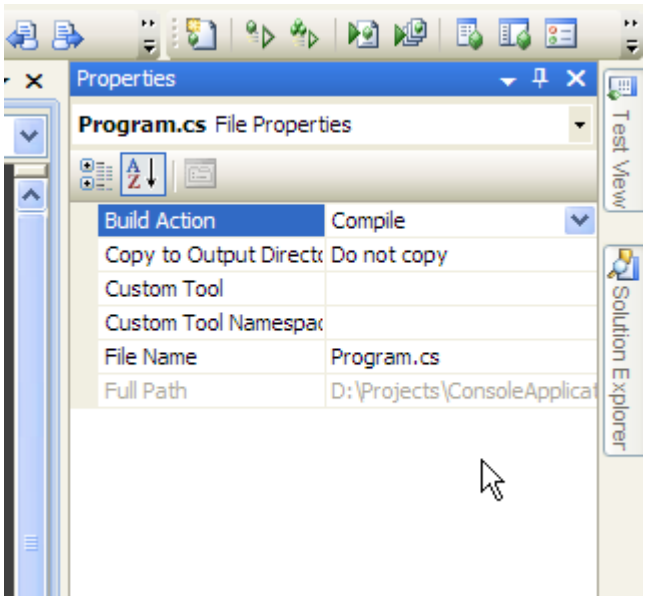
Térjünk most vissza a VS–hez és nézzük meg, hogy miként épül fel a kezelőfelület. Kezdjük a Solution Explorer–rel, amelyben az aktuális Solution elemeit kezelhetjük. Amennyiben nem látjuk ezt az ablakot (ha az Auto Hide be van kapcsolva, akkor csak egy „fülecske” jelenik meg az ablak szélén), akkor a View menüben keressük meg.



A „rajzszögre” kattintva az ablakot rögzíthetjük is. A Solution–ön vagy a projecten jobb egérgombbal kattintva új forrásfileokat, projecteket, stb. adhatunk a programunkhoz.

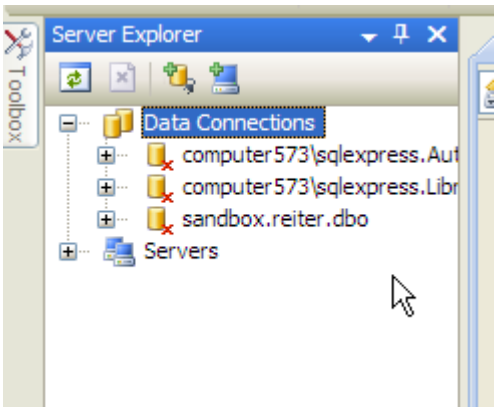


A Properties ablakban a kiválasztott elem tulajdonságait állíthatjuk. Többféle területen is felhasználjuk majd, többek között a grafikus felületű alkalmazások alkotóelemeinek beállításakor.



A képen a Program.cs file fordítási beállításait láthatjuk.

A Server Explorer segítségével (távoli) adatbázisokhoz kapcsolódhatunk, azokat szerkeszthetjük, illetve lekérdezéseket hajthatunk végre.

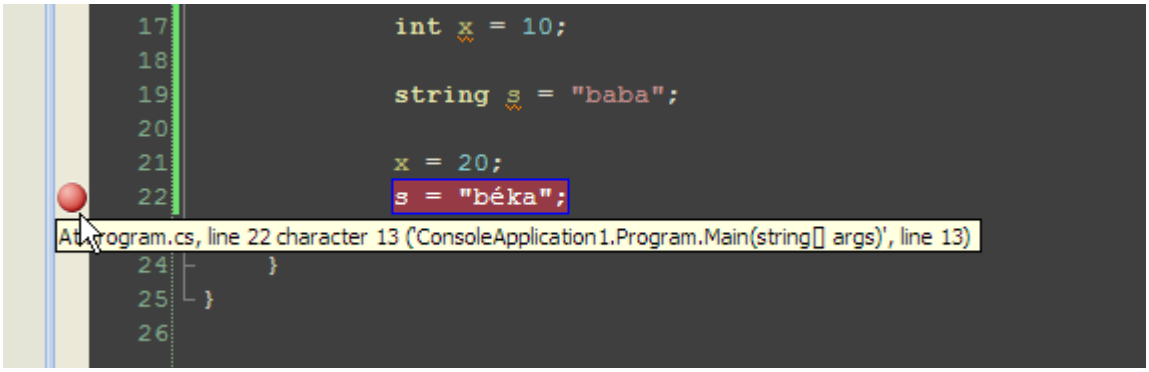


Végül a ToolBox ablak maradt, neki főként grafikus felületű alkalmazások fejlesztésekor vesszük hasznát, ekkor innen tudjuk kiválasztani a komponenseket/vezérlőket.

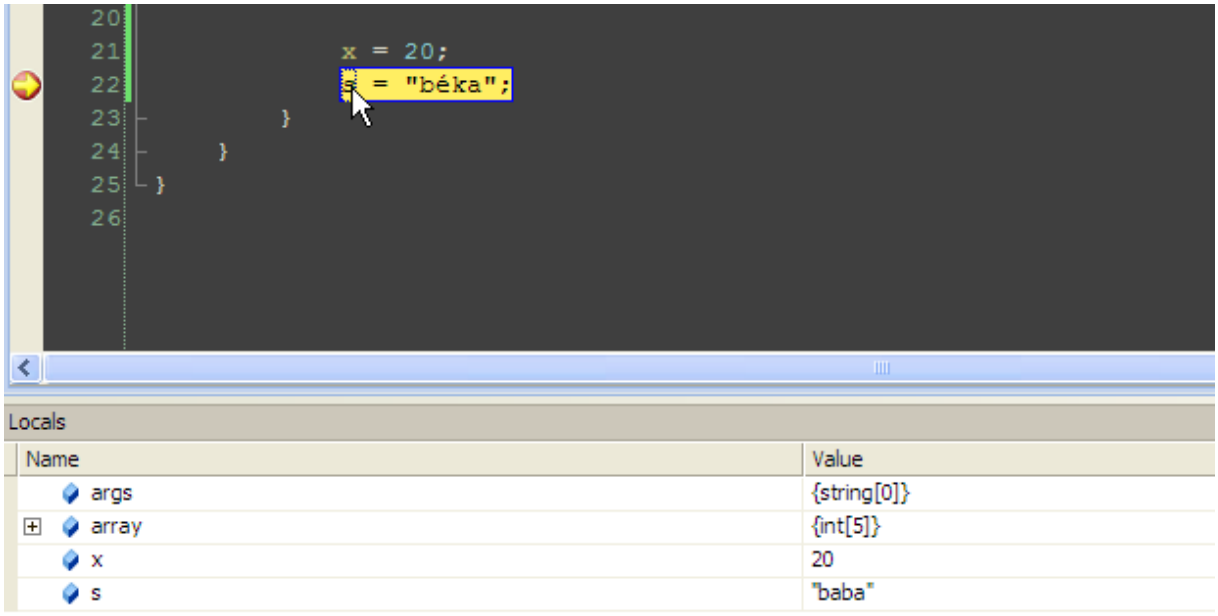
36.3 Debug

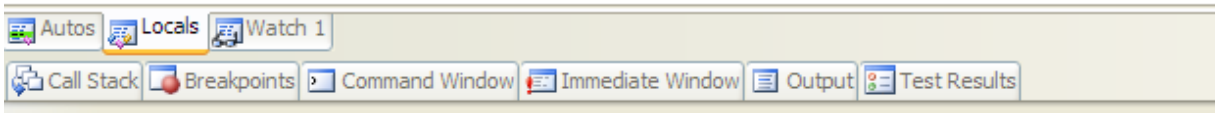
A Visual Studio segítségével a programok javítása, illetve a hibák okának felderítése sem okoz nagy nehézséget. Ebben a részben elsajátítjuk a hibakeresés alapjait: elsőként ismerkedjünk meg a breakpoint (vagy töréspont) fogalmával. Nyílván úgy tudjuk a legjobban felderíteni a programjaink hibáit, ha működés közben láthatjuk az objektumok állapotát. Erre olyan „fapados” módszereket is használhatunk, mint, hogy kiírjuk a konzolra ezeket az értékeket. Ennek persze megvan a maga hátránya, hiszen egyrészt be kell gépelni a parancsokat, másrészt beszennyezzük a forráskódot ami így nehezebben olvashatóvá válik. A breakpointok segítségével a

program egy adott pontján lehetőségünk van megállítani annak futását, így kényelmesen megvizsgálhatjuk az objektumokat.
A Visual Studio–ban a forráskód ablak bal oldalán lévő üres sáv szolgál töréspontok elhelyezésére:

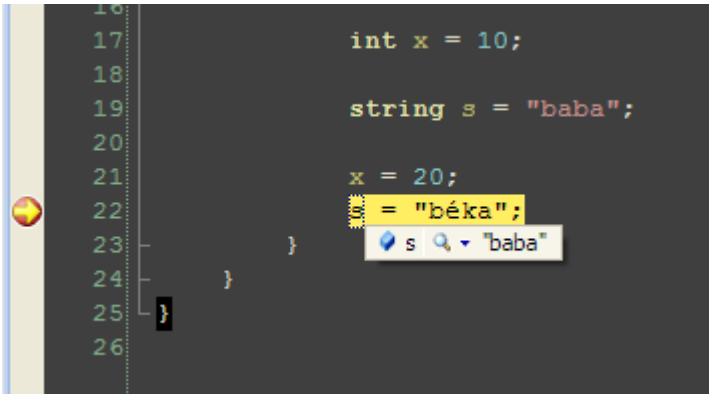


Ezután, ha futtatjuk a programot, akkor ezen a ponton a futása megáll. Fontos, hogy a törést tartalmazó sor már nem fog lefutni, vagyis a fenti képen s értéke még „baba” marad.





A képen látható a Locals ablak, amely a változók állapotát/értékét mutatja. Ugyanígy elérhetjük ezeket az értékeket, ha az egérmutatót közvetlenül a változó fölé visszük:

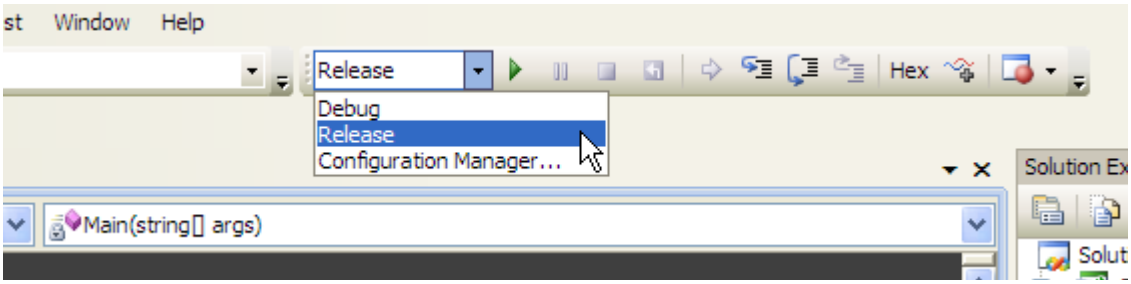


Mind itt, mind a Locals ablakban módosíthatjuk az objektumokat.

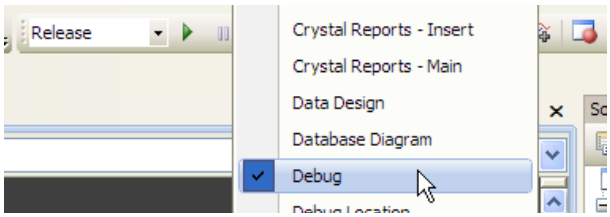
Az F11 gombbal utasításonként lépkedhetünk debug módban.

36.4 Debug és Release

A Visual Studio kétféle módban – Debug és Release – tud fordítani. Az előbbit tipikusan a fejlesztés során használjuk, mivel – ahogyan a nevében is benne van – a debuggolást segíti, mivel a lefordított programot kiegészíti a szükséges információkkal. Ebből kifolyólag ez a mód a lassabb, akár 100% -os sebességveszteséget is kaphatunk, ha így próbálunk számításigényes műveleteket végezni (épp ezért a Debug módot csakis a program általános funkcióinak fejlesztésekor használjuk). Amikor a sebességet is tesztelni akarjuk, akkor a Release módot kell bevetnünk.
Alapértelmezés szerint a Visual Studio Toolbar –ján ki kell tudnunk választani, hogy mit szereténk:



Ha mégsincs ott, akkor a ToolBar –on jobb gombbal kattintva jelöljük be a Debug elemet:



37 Osztálykönyvtár

Eddig mindig egyetlen futtatható állományt készítettünk, ennek viszont megvan az a hátránya, hogy a nagyobb osztályokat minden egyes alkalommal, amikor használni akarjuk újra be kell másolni a forráskódba. Sokkal egyszerűbb lenne a dolgunk, ha az újrahasznosítandó forráskódot külön tudnánk választani a tényleges programtól, ezzel időt és helyet takarítva meg. Erre a célra találták ki a shared library (~megosztott könyvtár) fogalmát, amely a Windows alapú rendszereken a DLL -t (Dynamic Link Library) jelenti.
A DLL könyvtárak felépítése a használt fejlesztői platformtól függ, tehát egy COM DLL nem használható közvetlenül egy .NET programban (de megoldható).

Készítsük el az első osztálykönyvtárunkat először parancssort használva, utána a Visual Studio segítségével. Az osztályunk (TestLib.cs a file neve) nagyon egyszerű lesz:

```
using System;

namespace TestNamespace
{
      public class TestClass
      {
            public string Text = "Hello DLL!";
      }
}
```

Ezúttal névteret is megadtunk, ez célszerű, mivel nem akarjuk, hogy más osztállyal ütközzön, vagyis egyértelműen meg tudjuk mondani, hogy mire gondoltunk. Amire még figyelni kell, az az osztály elérhetősége, hiszen most két különböző assembly–ről van szó, vagyis ha kívűlről is el akarjuk érni ezt az osztályt, akkor publikusként kell deklarálnunk. Fordítani a következőképpen tudjuk:

```
csc /target:library TestLib.cs
```

Ezután egy TestLib.dll nevű filet kapunk.
A „főprogramot” pedig így készítjük el:

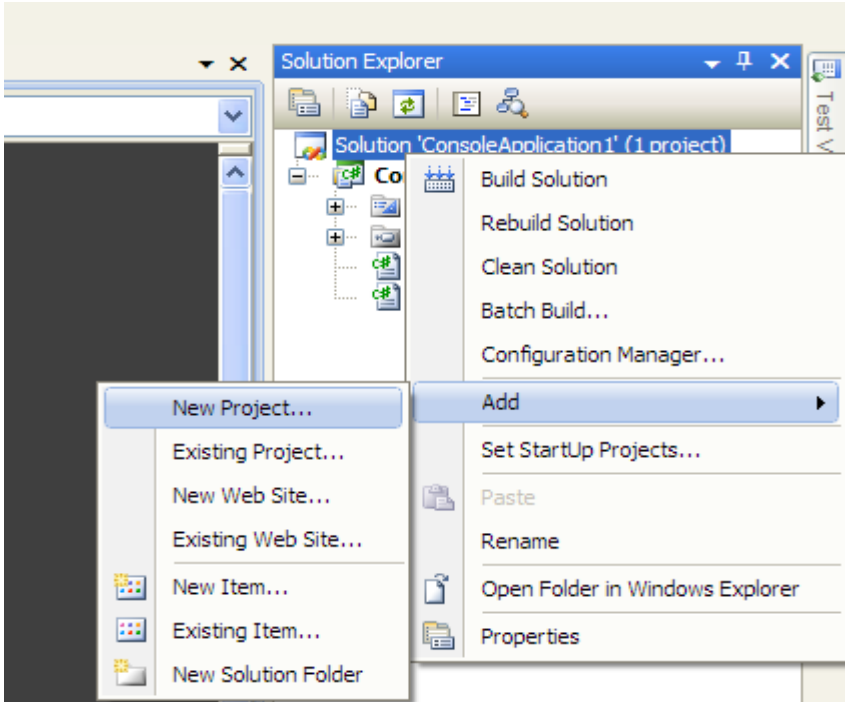
```
using System;
using TestNamespace;

class Program
{
      static public void Main()
      {
            TestClass tc = new TestClass();
            Console.WriteLine(tc.Text);
      }
}
```

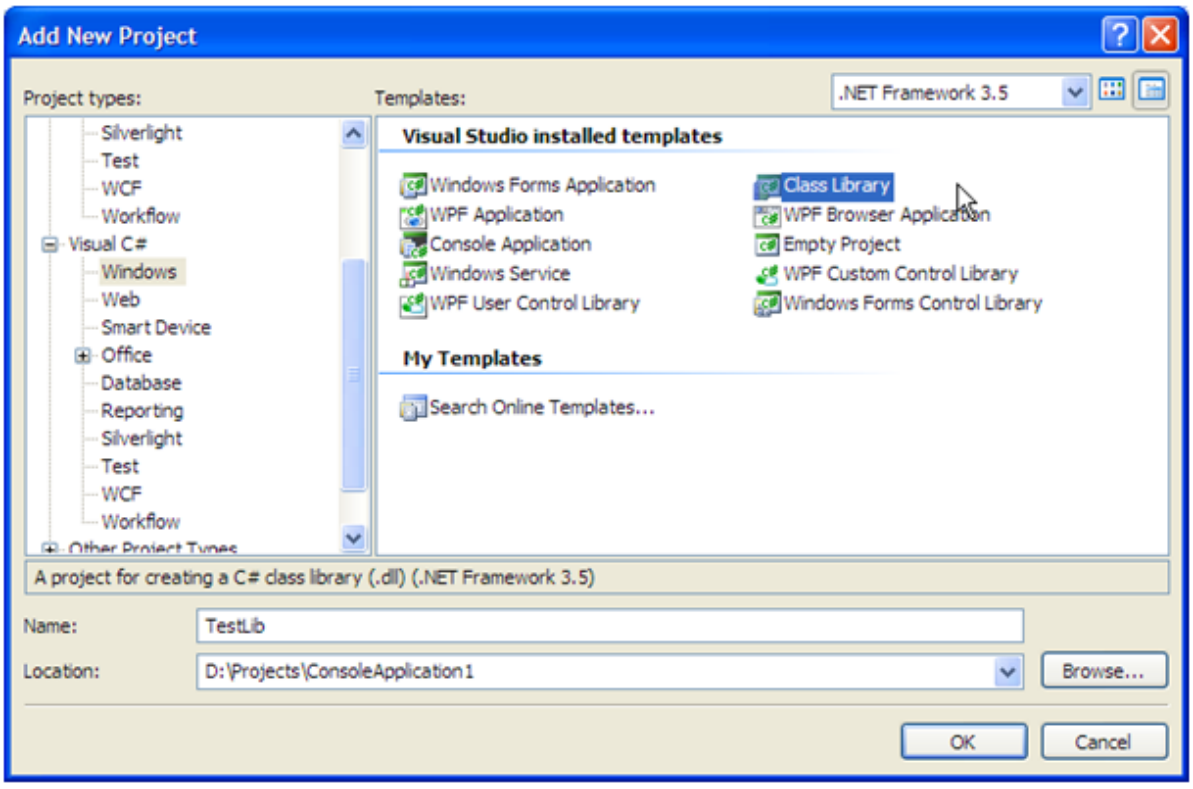
És így fordítjuk:

```
csc /reference:TestLib.dll main.cs
```

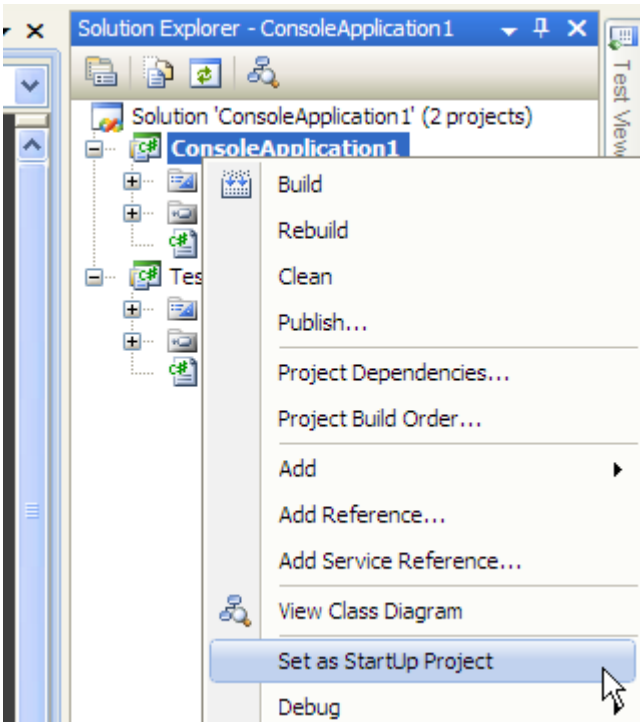
Most készítsük el ugyanezt Visual Studio–val. Csináljunk elsőként egy Console Application–t mint az előző fejezetben, majd kattintsunk jobb egérgombbal a Solution–ön (a Solution Explorerben) és az Add menüből válasszuk ki a New Projectet:



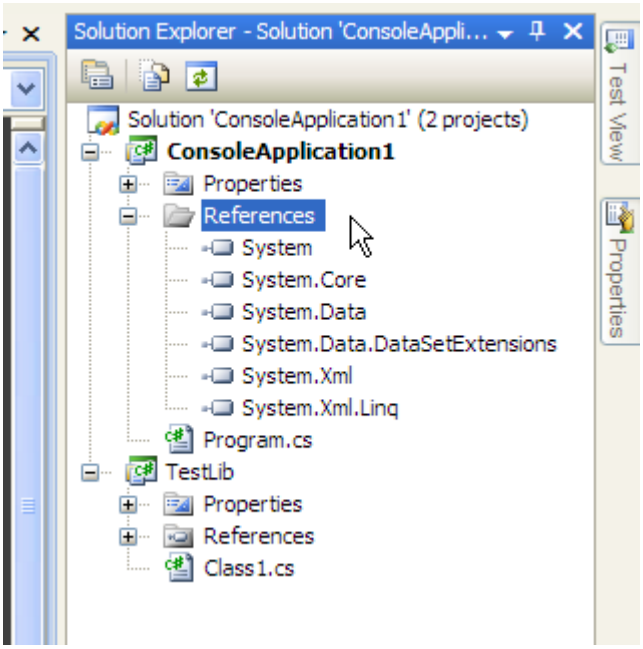
Majd a Class Library sablont



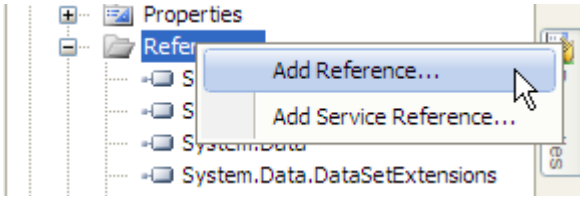
Az OK gombra kattintva a Solution–höz hozzáadódik az osztálykönyvtár. Előfordulhat, hogy ezután nem tudjuk futtatni a programot, mivel egy „Project with an Output Type...” kezdetű hibaüzenetet kapunk. Ez azt jelenti, hogy a frissen hozzáadott osztálykönyvtár lett a Solution „főprojectje”. Ezt megváltoztathatjuk, ha a valóban futtatni kívánt projecten jobb egérgombbal kattintva kiválasztjuk a „Set as StartUp Project” pontot:



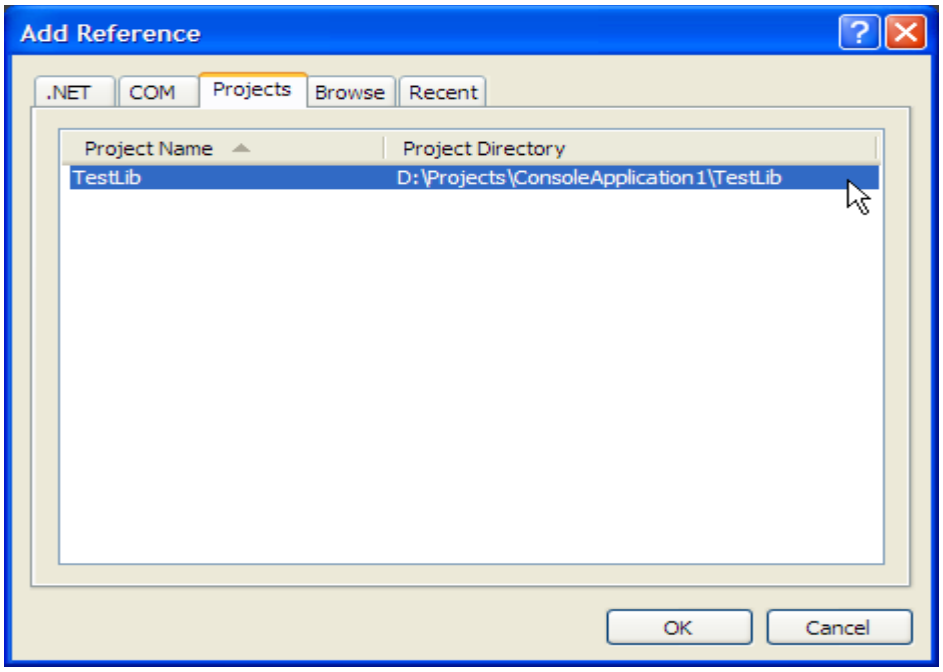
Készítsük el az osztálykönyvtárat és fordítsuk le (jobb egérgomb a projecten és Build). Most térjünk vissza a főprojecthez és nyissuk le a References fület, ez a már hozzáadott osztálykönyvtárakat tartalmazza:



Kattintsunk jobb egérgombbal rajta és válasszuk ki az Add Reference pontot:



A megjelenő ablakban kiválaszthatjuk, hogy a már beépített osztálykönyvtárakból (.NET) akarunk válogatni vagy megkeressük a nekünk kellő filet (Browse), esetleg az egyik aktuális projectre van szükségünk (Projects). Nekünk természetesen ez utóbbi kell:



Az Ok gombra kattintva a References alatt is megjelenik az új könyvtár. Ezután a szerkesztőablakban is láthatjuk az osztálykönyvtárunk névterét (ha mégsem akkor fordítsuk le az egész Solution–t):