Reiter István

C#

Tartalomjegyzék

1 Bevezetõ 8
1.1 A jegyzet jelölései........ 8
1.2 Jogi feltételek 8
2 M icrosoft .NET Framework 9
2.1 A .NET platfor m 9
2.1.1 MSIL/CIL 9
2.1.2 Fordítás/futtatás 10
2.1.3 BCL 10
2.2 A C# programozási nyelv 10
2.3 Alternatív megoldások 10
2.3.1 SSCLI 10
2.3.2 Mono........ 11
2.3.3 DotGNU 11
3 “Hello C#!”........ 12
3.1 A C# szintaktikája........ 13
3.1.1 Kulcss zavak 13
3.1.2 Megjegyzések 14
3.2 Névterek........ 15
4 Változók 16
4.1 Deklaráció és definíció 16
4.2 Típusok 16
4.3 Lokális és globális változók........ 17
4.4 Referencia- és értéktípusok........ 18
4.5 Referenciák 19
4.6 Boxing és unboxing 20
4.7 Konstansok 21
4.8 A felsorolt típus 22
4.9 Null típusok 24
4.10 A dinamikus típus 24
5 Operátorok 27
5.1 Operátor precedencia 27
5.2 Ér tékadó operátor 28
5.3 Matematikai operátorok 28
5.4 Relációs operátorok 29
5.5 Logikai/feltételes operátorok 30
5.6 Bit operátorok 32

5.7 Rövid for ma 35
5.8 Egyéb operátorok 36
6 Vezér lési szerkezetek 38
6.1 Szekvencia 38
6.2 Elágazás 38
6.3 Ciklus 41
6.3.1 Yield 45
6.3.2 Párhuzamos ciklusok 46
7 Gyakorló feladatok 48
7.1 Szorzótábla 48
7.2 Számológép.. 51
7.3 Kõ – Papír – O lló 52
7.4 Számkitaláló játék.. 54
8 Típuskonver ziók 59
8.1 Ellenõr zött konverziók... 59
8.2 Is és as 60
8.3 Karakterkonverziók 61
9 Tömbök.. 62
9.1 Többdimenziós tömbök 63
10 Str ingek.. 66
10.1 Metódusok 67
10.2 StringBuilder... 68
11 Gyakorló feladatok I I. 70
11.1 Minimum- és maximumkeresés 70
11.2 Szigetek... 70
11.3 Átlaghõmérséklet 71
11.4 Buborékrendezés 72
12 Objektum-orientált programozás - elmélet.. 74
12.1 UML 74
12.2 Osztály 74
12.3 Adattag és metódus 75
12.4 Láthatóság 75
12.5 Egységbezárás 76
12.6 Öröklõdés 76
13 Osztályok 78
13.1 Konstr uktorok 79
13.2 Adattagok... 82
13.3 Láthatósági módosítók 82
13.4 Parciális osztályok 82

13.5 Beágyazott osztályok 84
13.6 Objektum inicializálók........ 85
13.7 Destruktorok 86
13.7.1 IDisposable 92
14 Metódusok 93
14.1 Paraméterek 94
14.1.1 Alapértelmezett paraméterek 99
14.1.2 Nevesített paraméterek 100
14.2 Visszatérési érték 100
14.3 Kiterjes ztett metódusok 102
15 Tulajdonságok 103
16 Indexelõk 105
17 Statikus tagok 107
17.1 Statikus adattagok 107
17.2 Statikus konstr uktor 108
17.3 Statikus metódusok 108
17.4 Statikus tulajdonságok 109
17.5 Statikus osztályok 109
18 Struktúrák 110
18.1 Konstr uktor 110
18.2 Destruktor 111
18.3 Adattagok........ 112
18.4 Hozzárendelés 112
18.5 Öröklõdés 114
19 Gyakorló feladatok I II. 115
19.1 Faktor iális és hatvány 115
19.2 Gyorsrendezés 116
19.3 Láncolt lista 118
19.4 Bináris keresõfa 119
20 Öröklõdés 123
20.1 Virtuális metódusok 125
20.2 Polimor fizmus........ 127
20.3 Lezárt os ztályok és metódusok 128
20.4 Absztrakt osztályok 128
21 Inter fészek 131
21.1 Explicit interfés zimplementáció 133
21.2 Virtuális tagok 134
22 Operátor kiter jesztés 136
22.1 Egyenlõség operátorok 137

22.2 A ++/-- operátorok ... 138
22.3 Relációs operátorok ... 139
22.4 Konverziós operátorok... 139
22.5 Kompatibilitás más nyelvekkel... 140
23 Kivételkezelés ... 141
23.1 Kivétel hierar chia... 143
23.2 Kivétel készítése ... 143
23.3 Kivételek továbbadása ... 144
23.4 Finally blokk ... 145
24 Gyakorló feladatok I V. ... 146
24.1 IEnumerator és IEnumerable ... 146
24.2 IComparable és IComparer ... 148
24.3 Mátr ix típus ... 149
25 Delegate –ek ... 151
25.1 Paraméter és viss zatérési érték ... 154
25.2 Névtelen metódusok ... 155
25. Események... 156
26 Gener ikusok... 160
26.1 Gener ikus metódusok ... 160
26.2 Gener ikus osztályok... 161
26.3 Gener ikus megszorítások ... 163
26.4 Öröklõdés ... 165
26.5 Statikus tagok... 165
26.6 Gener ikus gyûjtemények ... 165
26.6.1 Lis t<T> ... 166
26.6.2 SortedLis t<T, U> és SortedDictionar y<T, U> ... 168
26.6.3 Dictionar y<T, U> ... 168
26.6.4 LinkedList< T>... 169
26.6.5 ReadOnlyCollection<T> ... 170
26.7 Gener ikus inter fészek, delegate –ek és események ... 170
26.8 Kovariancia és kontravariancia... 171
27 Lambda kifejezések ... 173
27.1 Gener ikus kifejezések ... 173
27.2 Kifejezésfák ... 175
27.3 Lambda kifejezések változóinak hatóköre... 175
27.4 Névtelen metódusok kiváltása lambda kifejezésekkel ... 176
28 Attr ibútumok ... 178
29 Unsafe kód ... 181
29.1 Fix objektumok ... 183

30 Többszálú alkalmazások ... 185
30.1 Application Domain -ek ... 187
30.2 Szálak ... 187
30.3 Aszinkron delegate - ek ... 188
30.3.1 Párhuzamos delegate hívás ... 192
30.4 Szálak létrehozása ... 193
30.5 Foreground és background szálak ... 194
30.6 Szinkronizáció ... 195
30.7 ThreadPool ... 199
31 Reflection ... 202
32 Állománykezelés ... 204
32.1 Olvasás/írás fileból/fileba ... 204
32.2 Könyvtárstruktúra kezelése ... 207
32.3 In – memory streamek ... 209
33 Konfigurációs file használata ... 211
33.1 Konfiguráció-szekció készítése ... 212
34 Hálózati programozás ... 215
34.1 Socket ... 215
34.2 Blokk elker ülése ... 221
34.3 Több kliens kezelése ... 223
34.3.1 Select ... 223
34.3.2 Aszinkron socketek ... 225
34.3.3 Szálakkal megvalósított szerver ... 226
34.4 TCP és UDP ... 229
35 LINQ To Objects ... 230
35.1 Nyelvi eszközök ... 230
35.2 Kiválasztás ... 231
35.2.1 Projekció ... 234
35.2.2 Let ... 235
35.3 Szûrés ... 235
35.4 Rendezés ... 237
35.5 Csoportosítás ... 238
35.5.1 Null értékek kezelése ... 240
35.5.2 Összetett kulcsok ... 240
35.6 Listák összekapcsolása ... 241
35.7 Outer join ... 243
35.8 Konverziós operátorok ... 244
35.9 „Element” operátorok ... 246
35.10 Halmaz operátorok ... 247

35.11 Aggregát operátorok 248
35.12 PLINQ – Párhuzamos végrehajtás 249
35.12.1 Többszálúság vs. Párhuzamosság 249
35.12.2 Teljes ítmény 249
35.12.3 PLINQ a gyakorlatban 250
35.12.4 Rendezés 253
35.12.5 AsSequential 254
36 Visual Studio 255
36.1 Az elsõ lépések 255
36.2 Felület 259
36.3 Debug 260
36.4 Debug és Release 262
37 Osztálykönyvtár 263

1 Bevezetõ

Napjainkban egyre nagyobb teret nyer a .NET Framework és egyik fõ nyelve a C#.
Ez a jegyzet abból a célból született, hogy megismertesse az olvasóval ezt a nagyszerû technológiát.
A jegyzet a C# 2.0, 3.0 és 4.0 verziójával is foglalkozik, az utóbbi kettõ által bevezetett új eszközöket a z adott rész külön jelöli.
Néhány fejezet feltétele z olyan tudást amely alapját egy késõbbi fejezet képezi, ezért ne essen kétségbe a kedves olvasó, ha valamit nem ért, egyszerûen olvasson tovább és térjen vissza a kérdéses fejezethez, ha rá talált a válaszra.
A jegyzet megértéséhez nem szükséges programozni tudni, viszont alapvetõ informatikai ismeretek (pl. számrendszerek) jól jönnek.

A jegyzethez tartozó forráskódok letölthetõek a következõ webhelyrõl:

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

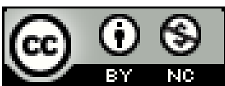
Bármilyen kérést, javaslatot illetve hibajavítást szívesen várok a reiteristvan@gmail.com email címre.

1.1 *A jegyzet jelölései*

Forráskód: szürke alapon, kerettel

Megjegyzés: fehér alap, kerettel

1.2 *Jogi feltételek*



A jegyzet teljes tartalma a **Creative Commons Nevezd meg!-Ne add el! 2.5 Magyarország** liszenszé alá tartozik. Szabadon módosítható és terjeszthetõ, a forrás feltüntetésével.
A jegyzet ingyenes, mindennemû értékesítési kísérlet tiltott és a szerzõ beleegyezése nélkül történik.

2 Microsoft .NET Framework

A kilencvenes évek közepén a Sun MicroSystems kiadta a Java platform elsõ nyilvános változatát. Az addigi programnyelvek/platformok különbözõ okokból nem tudták felvenni a Java –val a versenyt, így számtalan fejlesztõ 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 fedõnevû projektet, amelybõl aztán megszületett a .NET.

2.1 A .NET platform

Maga a .NET platform a Microsoft, a Hewlett Packard, az Intel és mások közremûködésével 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 elképzelésbõl alakult ki, hogy a .NET ugyanaz mint a Java csak Microsoft köntösben. A valóságban nincs .NET virtuális gép, helyette ún. felügyelt kódot használ, vagyis a program teljes mértékben natív módon, közvetlenül a processzoron fut, mellette pedig ott a keretrendszer, amely felelõs pl. a memóriafoglalásért vagy a kivételek kezeléséért.

A .NET nem egy programozási nyelv, hanem egy környezet. Gyakorlatilag bármilyen programozási nyelvnek lehet .NET implementációja. Jelenleg kb. 50 nyelvnek létezik hivatalosan .NET megfelelõje, nem beszélve a számtalan hobbifejlesztésrõl.

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 értelmezhetõek. Ezeknek a nyelveknek az elõnye a hátránya is egyben. Bár gyorsak, de rengeteg hibalehetõség rejtõzik a felügyelet nélküli (unmanaged) végrehajtásban.

A .NET (akárcsak a Java) más úton jár, a fordító elõször egy köztes nyelvre (Intermediate Language) fordítja le a forráskódot. Ez a nyelv a .NET világában az MSIL illetve a szabványosítás után CIL (**M**icrosoft/**C**ommon IL) – különbség csak az elnevezésben van.

2.1.2 *Fordítás/futtatás*

A natív programok ún. gépi kódra fordulnak le, míg a .NET forráskódokból egy CIL nyelvû futtatható állomány keletkezik. Ez a kód a feltelepített .NET Framework – nek szóló utasításokat tartalmaz. Amikor futtatjuk ezeket az állományokat elõször az ún. JIT (just–in–time) fordító veszi kezelésbe és lefordítja õket gépi kódra, amit a processzor már képes kezelni.
Amikor "elõször" lefordítjuk a programunkat akkor egy ún. Assembly (vagy szerelvény) keletkezik. Ez tartalmazza a felhasznált illetve megvalósított típusok adatait (ez az ún. Metadata) amelyeket a futtató környezet fel tud használni a futtatáshoz (az osztályok neve, metódusai, stb…). Egy Assembly egy vagy több file – ból is állhat.

2.1.3 BCL

A .NET Framework telepítésével a számítógépre kerül – többek közt – a BCL (Base Class Library), ami az alapvetõ feladatok (file olvasás/írás, adatbázis kezelés, adatszerkezetek, stb…) elvégzéséhez szükséges eszközöket tartalmazza. Az összes többi könyvtár (ADO.NET, WCF, stb…) ezekre a könyvtárakra épül.

2.2 *A C# programozási nyelv*

A C# (ejtsd: Szí-sárp) a Visual Basic mellett a .NET fõ programozási nyelve. 1999 – ben Anders Hejlsberg vezetésével kezdték meg a fejlesztését.
A C# tisztán objektumorientált, típusbiztos, általános felhasználású nyelv. A tervezésénél a lehetõ legnagyobb produktivitás elérését tartották szem elõtt.
A nyelv elméletileg platformfüggetlen (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 elérhetõ. Ugyanakkor a szabványosítás után a CLI specifikáció nyilvános és bárki számára elérhetõ 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. Ezt a célt nehezíti, hogy a Microsoft idõközben számos a specifikációban nem szereplõ változtatást végzett a keretrendszeren.

2.3.1 SSCLI

Az SSCLI (Shared Source Common Language Infrastructure) 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 liszenszengedélye 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 project jelen pillanatban megszûnni – vagy legalábbis idõlegesen leállni – látszik. Ettõl függetlenül a forráskód és a hozzátartozó dokumentációk rendelkezésre állnak, letölthetõek a következõ webhelyrõl:

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

2.3.2 Mono

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

A Mono Windows, Linux, UNIX, BSD, Mac OSX és Solaris rendszereken elérhetõ. Napjainkban a Mono mutatja a legígéretesebb fejlõdést, mint a Microsoft .NET jövõbeli "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 –el való kompatibilitást helyezi elõtérbe, 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 -hírhedt “ Hello World!” program elsõként Dennis Ritchie és Brian Kerningham “A C programozási nyelv” címû könyvében jelent meg, és azóta szinte hagyomány, hogy egy programozási nyelv bevezetõjeként ezt a programot mutatják be.
Mi itt most nem a világot, hanem a C# nyelvet üdvözöljük, ezért ennek megfelelõen módosítsuk a forráskódot:

```
using System;

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

Mielõtt lefordítjuk tegyünk pár lépést a parancssorból való fordítás elõsegítésére. Ahhoz, hogy így is 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.

Ez utóbbihoz Vezérlõpult/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… vagy a v3.5 .. kezdetû mappát (attól függõen, hogy a C# fordító melyik verziójára van szükségünk kell). Másoljuk ki a címsorból ezt a szép hosszú elérést. Vissza a Path – hoz. A változó értéke sorban navigáljunk el a végére, írjunk egy pontosvesszõt (**;**) és illesszük be az elérési utat. Mindent OK –zunk le és kész. Ha van megnyitva konzol vagy PowerShell azt indítsuk újra, utána írjuk be, hogy csc. Azt kell látnunk, hogy **Microsoft ® Visual C# 2008 Compiler Version 3.5** …
Itt az évszám és verzió változhat, ez a C# 3.0 üzenete.

Most már fordíthatunk a

csc filenev.cs

paranccsal (természetesen a szöveges file kiterjesztése „.txt”, így nevezzük át mivel a C# forráskódot tartalmazó file -ok kiterjesztése „.cs”).

Az elsõ sor megmondja a fordítónak, hogy használja a System névteret. Ezután létrehozunk egy osztályt, mivel a C# teljesen objektumorientált, ezért utasítást csakis osztályon belül adhatunk. A “ HelloWorld” osztályon belül definiálunk egy Main nevû statikus függvényt, ami a programunk belépési pontja lesz. Minden egyes C# program a Main függvénnyel kezdõdik, ezt mindenképpen létre kell hoznunk. Végül meghívjuk a Console osztályban lévõ WriteLine és ReadKey függvényeket. Elõbbi kiírja a képernyõre a paraméterét, utóbbi vár egy billentyûleütést.

Ebben a bekezdésben szerepe l néhá ny (sok) kifejezés ami ismeretlen le het, a jegyze t késõbbi feje zeteiben minde nre fény de rül majd.

3.1 *A C# szintaktikája*

Amikor egy programo zási nyelv szintaktikájáról beszélünk, akkor azo kra a szabályokra gondolunk, amelyek megszabják a fo rráskód felépítését. Ez azér t fontos, mert az egyes fordítóprogramok a z ezekkel a szabályokkal lé tre hozott kódot tudják értelme zni.
A C# ún. C -stílusú szi ntaxissal rende lke zik (a zaz a C programozás i nyelv szi nta xisát veszi alap ul), ez három fo ntos szabályt vo n maga utá n:

- Az egyes utasítások végén pontosvesszõ (**;**) áll
- A kis- és nagybetûk különbözõ jele ntõséggel b írnak, azaz a “ program ” és “ Program ” azo nosítók különbö znek. Ha a fenti kódban Conso le.WriteLi ne helyett console .writeline –t ír nánk a program nem ford ulna le .
- A program eg ységeit (osztá lyok, me tódusok, stb.) ún. blokkokkal jelöljük ki, a kapcsos záró jelek (**{**, **}**) seg ítségé vel.

3. 1.1 Ku lcssz avak

Szinte minde n program nyelv definiál kulcsszavakat, ame lyek speciális jele ntõséggel bírnak a fordító szá mára. Ezeket az a zo nosítóka t a saját meg hatá ro zott jelentésükön kívûl nem lehet másra használni, ellenke zõ esetbe n a ford ító hibát jele z. Vegyünk például egy vá ltozó t, ami nek az “int”nevet akar juk adni. Az “i nt” eg y beép ített típus ne ve is, a zaz kulcsszó, e zért nem fog lefo rdulni a prog ram:

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

A legtöbb fejlesztõeszkö z megszíne zi a kulcsszavakat (is) , e zért könnyû e lkerülni a fenti hibát.
A C# 77 darab kulcsszó t 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 kívûl léte zik még 23 darab azonosító, amelyeke t a nyelv nem tart fe nn speciális használatra, de különleges je lentéssel bír nak. Amennyiben le hetséges kerüljük a használatuka t “hagyományos” változók, metód usok, osztályok létreho zásánál:

add equals group let remove var
ascending from in on select where
by get into orderby set yield
descending global join partial value

Néhá nyuk a környeze ttõl függõe n más -más jelentésse l is bírhat, a megfelelõ fejezet bõvebb információt ad ma jd ezekrõ l az esetekrõl .

3. 1.2 *Megjegyz ések*

A forráskódba megjegyzéseket te hetünk. Ezzel egyrészt üze neteke t hagyhatunk (pl. egy metódus leírása) mag unknak, vagy a többi fejlesztõnek, másrészt a ko mme ntek segítségéve l dokumentációt tudunk generálni, a mi szi nté n a z elsõ célt szolgá lj a csak éppen élve zhetõbb formában.
Megjeg yzéseket a kö vetke zõképpen hag yhatunk:

```
using System ;

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

Az egysoros komme nt a saját sora leg végéig tart, m íg a többsoros a két “/*” –on belül érvényes. Utóbiakat nem lehet egymásba ág yazni:

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

Ez a “kód” nem fo rdul l e.
A kommenteket a fordító nem veszi figyelembe, tula jdonképpen a ford ítóprogram elsõ lépése, hog y a forráskódból eltá volít minden megjeg yzést.

3.2 *Névterek*

A .NET Framework osztálykönyvtárai szerény becslés szerint is legalább tízezer nevet, azonosítót tartalmaznak. Ilyen nagyságrenddel elkerülhetetlen, hogy a nevek ismétlõdjenek. Ekkor egyrészt nehéz eligazodni köztük, másrészt a fordító is megzavarodhat. Ennek a problémának a kiküszöbölésére hozták létre a névterek fogalmát. Egy névtér tulajdonképpen egy virtuális doboz, amelyben a logikailag összefüggõ osztályok, metódusok, stb… vannak. Nyílván könyebb megtalálni az adatbáziskezeléshez szükséges osztályokat, ha valamilyen kifejezõ nevû névtértben vannak (System.Data).

Névteret magunk is definiálhatunk, a namespace kulcsszóval:

```
namespace MyNameSpace
{

}
```

Ezután a névterre vagy a program elején a using –al, vagy az azonosító elé írt teljes eléréssel hivatkozhatunk:

```
using MyNameSpace;

//vagy

MyNameSpace.Valami
```

A jegyzet elsõ felében fõleg a System névteret fogjuk használni, ha másra is szükség van azt a jegyzet majd jelzi.

4 Változók

Amikor programot ír unk, akkor szükség lehet tárolókra, aho vá az adatai nkat ideiglenesen eltároljuk. Ezeket a tárolókat változóknak neve zzük.
A válto zók a memória eg y (vagy több) cellájára hiva tkozó le írók. Egy változót a következõ módon hozhatunk létre C# nyelven:

Típus vá lto zóné v;

A változóné v elsõ karaktere csak betû vagy a lulvonás jel (_) lehet, a többi karakter szám is. Le hetõleg kerüljük az éke zetes ka rakterek haszná latát.

Konve nció sze rint a válto zónevek kisbetûvel kezdõdnek. Amennyiben a változó név több szóból áll, akkor célszer û a zokat a szóhatár nál nagybetûvel “elválaszta ni” (pl.: pirosAlma).

4.1 *Deklaráció és definíció*

Egy válto zó (ille tve lényegében minden objektum) életciklusában megkülönböztetünk deklarációt és defi níciót. A deklaráció nak tartalma znia kell a típ ust és azo nosító t, a definícióban pedig megadjuk az objektum értékét. Értelemsze rûen a deklaráció és a definíció eg yszerre is megtörténhet.

4.2 *Típusok*

A C# erõsen (statikusan) típusos nyelv, ami azt jelenti, hogy minden eg yes vá lto zó típ usának ismertnek ke ll le nnie fordítási idõben. A típ us határozza meg, hog y egy válto zó milyen ér tékeket tarta lmazhat ille tve mekko ra helyet foglal a memóriában.
A következõ táblá za t a C# beépített típusait tartalma zza, mellettük ott a .NET megfele lõjük, a mé retük és egy rö vid leírás:

C# típus	.NET típus	Méret (byte)	Leírás
byte	System .Byte	1	Elõjel nélküli 8 bites egész szám (0..255)
char	System .Char	1	Egy Unicode karakter
bool	System .Boolea n	1	Logikai típus, értéke igaz(1) vag y hamis(0)
sbyte	System .SByte	1	Elõjeles 8 bites egész szá m (-128..127)
shor t	System .Int16	2	Elõjeles 16 bites egész szám (-32768..32767
ushort	System .Ui nt16	2	Elõjel né lküli 16 bites egész szám (0..65535)
int	System .Int32	4	Elõjeles 32 bites egész szám (–2147483647.. 2147483647).
uint	System .Ui nt32	4	Elõjel nélüli 32 bites egész szám (0..4294967295)

float	System .Single	4	Egyszeres po ntosság ú lebegõpontos szám
double	System .Doub le	8	Kétszeres pontosságú lebegõpontos szám
decimal	System .Decimal	8	Fix pontosság ú 28+1 jeg yû szám
long	System .Int64	8	Elõjeles 64 bites egész szám
ulong	System .Ui nt64	8	Elõjel nélküli 64 bites egész szám
string	System .String	NA	Unicode karakterek szekve nciája
object	System .Object	NA	Minde n más típ us õse

A forráskódba n te ljesen mindegy, hog y a “rendes” vagy a .NET né ven hivatko zunk egy típusra.

Alakítsuk á t a “Hello C#” programot úgy, hog y a ki írandó szöveget egy változóba tesszük:

```
using System ;

class HelloWorld
{
    static public void Main ()
    {
        //string típusu v ltozó deklar ciója, benne a kiírandó sz veg
        string message = "Hello C#" ;
        Console . WriteLine ( message );
        Console . ReadKey ();
    }
}
```

A C# 3.0 lehetõ vé teszi, hog y eg y metódus hatókörében deklarált válto zó típ usának meghatá ro zását a fordítóra bízzuk, általában olyankor tesszük ezt, amikor hosszó típ usné vrõl va n szó, vagy nehé z megha táro zni a típ ust. Ezt az akciót **var** szóval kivitele zhetjük. Ez természetesen nem jele nti azt, hogy úg y használhatjuk a nyelvet, mint eg y típustala n környe zetet, mi vel abban a pillana tban, hog y ér téket rendeltünk a válto zóho z (ezt azo nnal meg kell te nnünk), az úgy fog viselkedni mint az ekvi vale ns típ us. A ilyen válto zók típusa nem válto ztatható meg, de a megfelelõ típuskonver ziók végrehajthatóak.

```
int x = 10 ; // int típus v ltozó
var z = 10 ; // int típus v ltozó
z = "string" ; // fordít si hiba
var w; //fordít si hiba
```

4.3 *Lokális és globális változók*

Egy blokko n belül deklarált változó lokális lesz a blokkra né zve , vagyis a program többi részébõl nem látható (úgy is mo ndhatjuk, hog y a változó hatóköre a blokkra terjed ki). A fe nti példában a message egy lokális változó, ha egy másik függ vénybõl vag y osztályból próbálná nk meg elér ni, akkor a progra m nem ford ulna le.
Globális válto zónak azoka t a z objektumokat nevezzük, ame lyek a program bár mely részébõ l elérhetõek. A C# nem rendelke zik “igazi” globális változóva l mivel

deklarációt csak osztályon belül végezhetünk. Áthidalhatjuk a helyzetet statikus változók használatával, erről később lesz szó.

4.4 Referencia- *és értéktípusok*

A .NET minden típusa direkten vagy indirekten a **System.Object** nevű típusból származik, és ezen belül szétoszlik érték- és referenciatípusokra (illetve pointerekre, de erről egy későbbi fejezetben). A kettő közti 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 verem egy ún. **LIFO** (last-in-first-out) adattár, vagyis az az elem amit utoljára berakunk az lesz a tetején, kivenni pedig csak a mindenkori legfelső elemet tudjuk. A halom nem adatszerkezet, hanem a program által lefoglalt nyers memória, amit a CLR tetszés szerint használhat. Minden művelet a vermet használja, pl. ha össze akarunk adni két számot akkor a CLR lerakja mindkettőt a verembe és meghívja a megfelelő utasítást ami kiveszi a verem legfelső két elemét összeadja őket, a végeredményt pedig visszateszi a verembe:

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

A verem:

| 11 |
| 10 |-- összeadás művelet-- | 21 |
```

A referenciatípusok minden esetben a halomban jönnek létre mert ezek összetett adatszerkezetek és így hatékony a kezelésük. Az értéktípusok vagy a verembe n vagy a halomban vannak attól függően, hogy hol deklaráltuk őket:
Egy metóduson belül, lokálisan deklarált értéktípus a stackbe kerül, egy referenciatípuson belül adattagként deklarált értéktípus pedig a halomban foglal 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 eléggé biztosak lehetünk benne, hogy a verembe fog kerülni.

```
class MyClass
{
```

```
        private     int   x  =  10 ;
}
```

Most x egy referenciatípuson (ese ünkbe n eg y osztályo n) belül adattag, e zért a heapben foglal majd helyet.

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

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

Most eg y kicsit bonyolultabb a helyzet. Az y nevû vá ltozó egy refere nciatíp uson belül, de egy me tód usban , lokálisan deklaráltuk , így a stackbe n fog táro lódni, x pedig még mindig adattag ezér t marad a halomban.

Végül nézzük meg, hog y mi lesz ér ték- és mi refere nciatíp us; Értéktípus lesz a z összes olyan objektum, amelyeket a követke zõ típ usokkal deklaráltunk:

- Az összes beép ített numerikus típus (int, b yte, do uble, stb ...)
- A felsorolt típ us (e num)
- Logikai típ us (bool)
- Ka rakter típus (char)

Referenciatípusok lesznek a követke zõk:

- Osztályok (class)
- Interfész típusok
- Delegate típ usok
- Stringek
- Minden olya n típ us, ame ly *kö zvetlenül, direkt m ódon* szárma zik a System.Ob ject –bõl.

4.5 *Referenciák*

Az érték - illetve referenciatípusok kö zötti különbség egy másik aspektusa a z, ahogyan a forráskódban hivatko zunk rá juk. Vegyük a követke zõ "kódot":

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

Az elsõ sorban létreho ztuk a z x ne vû válto zót, a másodikban pedig egy új változónak adtuk értékül x –et. A kérdés az, hogy y hová muta t a memóriában, oda ahol x va n, vag y egy teljesen más helyre?
Amikor egy értéktípusra hiva tkozunk, akkor ténylegesen a z értékét haszná ljuk fel, vagyi s a kérdésünkre a válasz a z, hogy a ké t vá ltozó értéke egyenlõ lesz, de nem ug yana zon a me móriater ülete n helyezked nek e l.

A helyzet más a referenciatípusok esetében. Mivel õk összetett típusok ezért fizikailag lehetetlen lenne az értékeikkel dolgozni, ezért egy referenciatípusként létrehozott változó tulajdonképpen a memóriának arra a szeletére mutat ahol az objektum ténylegesen van. Nézzük meg ezt közelebbrõl:

```
using System;

class MyClass
{
    public int x;
}

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

        MyClass p = s;
        p.x = 14;

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

Vajon mit fog kiírni a program? Kezdjük az elejérõl! Hasonló a felállás mint az elõzõ forráskódnál, viszont amikor a második változónak értékül adjuk az elsõt, akkor az történik, hogy a p nevû referencia ugyanarra a memóriaterületre hivatkozik majd, mint az s, vagyis tulajdonképpen s egy álneve (alias) lesz. Értelemszerûen ha p módosul akkor s is így tesz, ezért a fenti program kimenete 14 lesz.

4.6 Box*ing és unboxing*

Boxing –nak (bedobozolás) azt a folyamatot nevezzük, amely megengedi egy értéktípusnak, hogy úgy viselkedjen, mint egy referenciatípus. Korábban azt mondtuk, hogy minden típus direkten vagy indirekten a System.Object –bõl származik. Az értéktípusok esetében az utóbbi teljesül, ami egy igen speciális helyzetet jelent. Az értéktípusok alapvetõen *nem származnak* az Object –bõl, mivel így hatékony a kezelésük, nem tartozik hozzájuk semmiféle "túlsúly".
Azonban elõfordulnak olyan helyzetek, amikor igenis úgy kell viselkedniük, mint egy referenciatípus (ilyen pl. egy szimpla konzolos kiíratás is). Ekkor az fog történni, hogy a CLR helyet foglal a heapen (akkor is, ha az eredeti értéktípus a stackbõl származik) és létrehoz egy olyan referenciatípust, ami tartalmazni fogja az értéktípus értékét és a System.Object –bõl származik.

Nézzük a következõ példát:

```
int x = 10;
Console.WriteLine("X erteke: {0}", x);
```

A Console.WriteLine me tódus ebbe n a formájában második paraméteréül egy Object típ usú változót vár, vagyis ebben a pilla natba n a CLR automatikusa n bedobozo lja az x vá lto zót.

A követke zõ for ráskód megmutatja, hogyan tudunk “kézze l” dobo zolni:

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

Most nem volt szükség a CLR – re .
Az unbo xing (vag y kidobozo lás) a boxi ng ellentéte, vagyis a bedobozo lt értéktíp usunkból kiva rá zsoljuk az e redeti értéké t:

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

Az object típ uson egy e xplicit típ usko nverziót hajtottunk végre (errõl hamarosan) , íg y visszanye rtük a z eredeti értéket.

Fontos még megjegye zni, hogy a bedobozo lás után teljesen új objektum keletkezik, amelynek semmi köze az erede tihez:

```
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, hog y z – n konver ziót kellett végrehajtanunk a z összeadásho z, de az értékadásho z nem (elõször kidobozoltuk, összeadtuk a két számot, majd a z eredmé nyt visszadobozoltuk).

Értelemszer ûen a z értéktípusok bedobozolása/kidobozo lása nem “olcsó” folyama t, ezért érdemes figye lni rá, hog y lehe tõség szeri nt minél kevesebb ilyen szituációba kerüljünk.

Az értéktíp usok és a System .Object közötti kapcsolatot konverziós kapcsolatnak ne vezzük, vag yis kö zvetlen kapcsolat ugya n ni ncs köztük, de egymás kö zötti konverzió létezik.

4.7 Konstansok

A **const** típusmódosító segítségével egy objektumot konstanssá, megváltoztathatatlanná tehetünk. A konstansoknak egyetlen egyszer adhatunk (és ekkor kell 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 idõben ki kell tudnia értékelni a fordítónak. A következõ forráskód épp ezért nem is fog lefordulni:

```
using System;

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

A Console.ReadLine metódus egy sort olvas be a standard bemenetrõl (ez alapértelmezés szerint a konzol lesz, de megváltoztatható), amelyet termináló karakterrel (Carriage Return, Line Feed, stb...), pl. az Enterrel zárunk.
A metódus egy string típusú értékkel tér vissza, amelybõl ki kell nyernünk a felhasználó által megadott számot. Erre fogjuk használni az int.Parse metódust, ami paraméterként egy stringet vár és számot ad vissza. A paraméterként megadott karaktersor nem tartalmazhat numerikus karakteren kívûl mást ellenkezõ 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 a = Animal.Tiger;

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

Enum típust csakis metóduson kívûl (osztályon belül, vagy "önálló" típusként) deklarálhatunk, ellenkezõ 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 (akkor is lefordulna, ha a Program osztályon kívûl deklaráljuk).

A felsorolás minden tagjának megfeleltethetünk egy egész értéket. Ha mást nem adunk meg, akkor alapérelmezés szerint a számozás nullától kezdõdik és deklaráció szerinti sorrendben (értsd: balról jobbra) eggyel növekszik. Ezen a módon az enum objektumokon explicit konverziót hajthatunk végre a megfelelõ 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
```

Az Enum.TryParse metódussal string értékekbõl "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);
    }
}
enum Animal { Cat = 1, Dog = 3, Tiger, Wolf }
```

Azok az "nevek" amelyekhez nem rendeltünk értéket implicit módon, az õket megelõzõ név értékétõl számítva kapják meg azt. Így a fenti 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);
    }
}
```

4.9 *Null típusok*

A referenciatípusok az inicializálás előtt automatikusan nullértéket vesznek fel, illetve mi magunk is jelölhetjük őket "beállítatlannak":

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

Ugyanez az értéktípusoknál már nem működik:

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

Azt már tudjuk, hogy a referenciatípusokra referenciákkal a nekik megfelelő memóriacímmel mutatunk, ezért lehetséges null értéket megadni nekik. Az értéktípusok pedig az általuk tárolt adatot reprezentálják, ezért ők nem vehetnek fel nullértéket.

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 m r muk dik
```

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
```

4.10 *A 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 minden változó/objektum statikusan típusos volt, vagyis egyrészt a típust fordításkor meg kellett tudnia határozni a fordítónak, másrészt futási idõ 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 idõben, még olyan dolgokat is amelyek futásidejû hibát okoznak. Ezenkívül az összes ilyen „objektum” futásidõben megváltoztathatja a típusát is:

```
using System;

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

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

Vegyük a következõ 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 „átvehetõ” metódusok közé, akár használtuk a deklarációnál a dynamic –ot, akár nem, a paramétereket kötelezõ megadnunk, ellenkezõ esetben a program nem fordul el.

Bár a fenti tesztek „szórakoztatóak”, de nem túl hasznosak. Valójában a dynamic „hagyományos” objektumokon való használata 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 scriptnyelvekkel – való együttmûködésben rejlik és a dynamic kulcsszó mögött egy

komoly platform a Dynamic Lang uage Runtime (DLR) á ll (természetesen a dynamic melle tt jóné hány osztá ly is helyett kapott a csomagban). A DLR olyan „típusta la n”/gye ngé n típ usos nyelvekkel tud eggyüttmûködni mint a L ua, Java Script, PHP, Pytho n vagy Ruby.

5 Operátorok

Amikor programozunk utasításokat adunk a számítógépnek. Ezek az utasítások kifejezésekbõl 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. x + y –> ezt az értéket jelöljük * -al
2. i = * –> i – nek értékül adjuk a * -ot

Az elsõ esetben x és y operandusok, a „+ jel pedig az összeadás mûvelet operátora. Ugyanígy a második pontban i és * (vagyis x+y) az operandusok az értékadás mûvelet („=) pedig az operátor.
Egy operátornak nem csak két operandusa lehet. A C# nyelv egy - (unáris) és háromoperandusu (ternáris) operátorokkal is rendelkezik.

A következõ néhány fejezetben átvesszünk néhány operátort, de nem az összeset. Ennek oka, hogy bizonyos operátorok önmagukban nem hordoznak jelentést, egy speciális részterület kapcsolódik hozzájuk, ezért ezeket az operátorokat majd a megfelelõ helyen ismerjük meg. (Pl. az indexelõ operátor most kimarad, elsõként a tömböknél találkozhat vele az olvasó.)

5.1 *Operátor precedencia*

Amikor több operátor is szerepel egy kifejezésben a fordítónak muszáj valamilyen sorrendet (precedenciát) fölállítani köztük, hiszen az eredmény ettõl is függ. Pl.:

```
10 * 5 + 1
```

Sorrendtõl függõen az eredmény lehet 51, vagy 60. A jó megoldás az elöbbi, az operátorok végrehajtásának sorrendjében a szorzás és osztás elõnyt élvez. A legelsõ 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 semmilyen hatással nincs (és a forráskód olvashatóságát is javítja). A fenti kifejezés tehát így nézzen ki:

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

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

1. Zárójel, adattaghozzáférés (pont (.) operátor), metódushívás, postfix inkrementáló/dekrementáló operátorok, a new operátor, typeof, sizeof, checked/unchecked.
2. Pozitív/negatív operátorok (x = 5), logikai tagadás, bináris tagadás, prefix inkrementáló/dekrementáló operátorok, explicit típuskonverzió.
3. Szorzás, maradékos - és maradék nélküli osztás.
4. Összeadás, kivonás.
5. Biteltoló (jobbra és balra) operátorok.
6. Kisebb (vagy egyenlõ), nagyobb (vagy egyenlõ), as, is.
7. Egyenlõ és nem egyenlõ 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 mûvelet amit elvégezhetünk az az, hogy egy változónak értéket adunk. A C# nyelvben ezt az egyenlõségjel segítségével tehetjük meg:

```
int x = 10;
```

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

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

Ettõl 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 a fenti két kódrészletbõl).
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 a megfelelõ konverzió (a típuskonverziókkal egy késõbbi fejezet foglalkozik).

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

5.3 *Matematikai operátorok*

A következõ 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 ; // sszead s: z = 10 + 3
                Console . WriteLine ( z ); //Kiírja az eredményt: 13
                z = x - y; //Kivon s: z = 10 - 3
                Console . WriteLine ( z ); // 7
                z = x * y; //Szorz s: z = 10 * 3
                Console . WriteLine ( z ); //30
                z = x / y; //Maradék nélküli oszt s: z = 10 / 3;
                Console . WriteLine ( z ); // 3
                z = x % y; //Maradékos oszt s: z = 10 % 3
                Console . WriteLine ( z ); // Az oszt s maradék t írja ki: 1
                Console . ReadKey (); //V r egy billentyuleütést
        }
}
```

5.4 *Relációs operátorok*

A relációs operátorok seg ítségé vel egy adott értékészlet e lemei közti viszo nyt tudjuk lekérde zni. Relációs operátort használó mûveletek eredménye vagy igaz (true) vagy hamis (false) lesz.
A numerikus típusokon értelme zve van egy re nde zés 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 egyenl y –al: true
            Console . WriteLine ( x <= y); //x kisebb-egyenl mint y: true
    }
}
```

Az elsõ sor eg yértelm û, a másodikban az eg ye nlõséget vizsgáljuk a kettõs egyenlõségje lle l. Ilyen esetekbe n figyelni kell, mert egy elütés is nehe ze n kideríthetõ hibát okoz, amikor egyenlõség helyett a z értékadó operátort használjuk. Az esetek többségében ugya nis íg y is le fog fordulni a program, m ûköd ni viszont valószínûleg rosszul fog.
A relációs operátorok összefogla lása:

x > y x nagyobb mint y

x >= y	x nag yobb vagy egye nlõ mi nt y
x < y	x kisebb mint y
x <= y	x kisebb vagy eg yenlõ mi nt y
x == y	x egye nlõ y -al
x != y	x nem egye nlõ y -al

5.5 *Logikai/feltételes operátorok*

Akárcsak a C++, a C# sem rendelkezik „igazi" logikai típussa l, ehelyett 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" );
        }
    }
}
```

Elõször felvettünk két logikai (bool) változót, az elsõnek „iga z" a másodiknak „ hamis" értéket adtunk. Ezutá n egy elágazás követke zik, errõl bõ vebbe n eg y késõbbi feje zetbe n le het olvasni, a lényege a z, hogy ha a fe ltétel iga z, akkor végre hajt egy utasítás t (vag y utasításokat). A fe nti példában az „ÉS" (**&&**) operátort használtuk, ez két operand ust vá r és akkor ad vissza „igaz" értéket, ha mindké t operandusa „iga z" vag y nullá nál nagyobb értéket képvisel. Ebbõl köve tkezik az is, hogy aká r az elõ zö fejezetben me gismert relációs operá torokból felépített kifeje zések, vagy matematikai formulák is lehetnek operand usok. A program ne m sok minde nt csinál, csak ki ír ja, hogy „Igaz".
Nézzük az „ÉS" igazságtáb lá zatát:

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

A fenti forráskód jó g yakorlás az operátor precede nciához, a z elága zás feltételébe n elõszö r az egyenlõséget fogjuk vizsgálni (a hetes számú kategória) és csak utána a felté teles ÉS –t (tize neg yes 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 elõzõ, a különbség a feltételben van. Látható „k” biztosan nem „igaz” (hiszen épp elõtte 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 kiértékelõdni, mert „l” van az elsõ helyen és mivel õ „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 ))
        {
            Console . WriteLine ( "X nem egyenlo 11 -el!" );
        }
    }
}
```

Ennek az operátornak egy operandusa van, akkor ad vissza igaz értéket, ha az operandusban megfogalmazott feltétel hamis, vagy – ha kifejezésrõl beszélünk - egyenlõ nullával.

A „tagadás” (negáció) igazságtáblája:

A	Eredmény

hamis igaz
igaz hamis

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

```
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 háromoperandusu operátor, a következõképpen mûködik:

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

```
using System;

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

        Console.WriteLine((x == y) ? "Egyenlõ" : "Nem egyenlõ");

    }
}
```

5.6 *Bit operátorok*

Az elõzõ fejezetben említett logikai operátorok bitenkénti mûveletek elvégzésére is alkalmasak.
A számítógép az adatokat kettes számrendszerbeli alakban tárolja, így pl. ha van egy byte típusú változónk (ami egy byte aza z 8 bit hosszú) aminek a „2” értéket adjuk, akkor az a következõképpen jelenik meg a memóriában:

2 00000010

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

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

01101101
00010001 AND
00000001

Elég egyértelmû, hogy az „ÉS” igazságtáblát használtuk az eredmény kiszámolásához.
Példa:

```
using System;

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

Bitenkénti „VAGY”: hasonlóan mûködik mint az „ÉS”, de a végeredményben egy bit értéke akkor lesz 1, ha a két operandus adott bitje közül az egyik is az:

01101101
00010001 OR
01111101

```
using System;

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

Biteltolás balra: a kettes számrendszerbeli alak „felsõ” bitjét eltoljuk és a jobb oldalon keletkezõ „üres” bitet nullára állítjuk. Az operátor: <<:

10001111 LEFT SHIFT
100011110

```
using System;

public class Program
{
```

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

Amikor biteltolást vég zünk figyelnünk kell arra, hogy a mûvele t végeredmé nye minden esetben 32 bites elõjeles szám (int) lesz. Ennek nag yon egyszer û oka va n, mégpedig az, hogy íg y bíztosítja a .NET, hogy a z ered mény elfér jen (a fenti példában használt 143 pl. po ntosan 8 biten felír ható szám, a zaz egy byte típusban már nem férne el eltolás után, hiszen akkor 9 bitre le nne szükségünk) .
Biteltolás jobbra : most az alsó bitet to ljuk el és fe lül pótoljuk a hiányt. Az operátor: >>:

```
using System;

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

A legtöbb beépített típ ust kö nnyen konver tálha tjuk át különbö zõ szá mre ndszerekre a Conve rt.ToString(x, y) metód ussal, aho l x a z ob jektum amit ko nver tálunk y pedig a szám rendszer:

```
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 konver tálás után csak a „ hasznos” részt fogja vissza ad ni, ezé r t fog az i nt típ usú - eg yébként 32 bites - változó csak 4 bite n meg jelenni (hiszen a 10 eg y pontosa n 4 biten felírható szám).
A char típus numerikus értékre konvertá lásakor az Unicode táblába n elfog lalt helyét adja vissza. Ez az a karakter esetében 97 (tízes szám re ndszer), 1100001 (kettes szr .) vagy 0061 (tizenhatos szr.) lesz. Ez utóbbi nál is csak a hasznos részt kapjuk vissza, hiszen a felsõ nyolc bit itt nullákból áll

Vegyük észre, hogy amíg a ba lra va ló eltolás té nylegesen – fi zikailag – ho zzátett az eredeti számunkhoz addig a jobbra tolás elvesz belõle, hiszen a „felülre” érkezõ nulla bitek nem hasznosulnak az eredmény szempontjából. Értelemszerûe n a ba lra tolás ezért mi ndig növelni, a jobbra tolás pedig mindig csökke nteni fogja a z eredmé nyt.

Enné l is tovább mehetünk, fe lfede zve a biteltolások valódi hasznát: egy n bittel ba lra tolás megfelel az alapszám 2 a z n –edik ha tványáva l való szor zás nak:

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

Ugya nígy a jobbra tolás ugyana zzal a ha tvánnya l oszt (nullá ra kerekítéssel):

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

Amikor olyan programot készítünk, amely erõsen épít kettõ vel vagy hatványai val való szor zásra /osztásra akkor a jánlott bitmûvele teket használni, mi ve l e zeket a processzor mindig sokkal gyo rsabban vég zi el, mi nt a hagyományos szor zást (tulajdonképpe n a szorzás a processzor leglassabb mûvelete) .

5.7 *Rövid forma*

Vegyük a követke zõ példát:

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

Az x nevû válto zót megnö veltük tízzel. Csakhogy van eg y kis baj: e z a megoldás nem túl hatékony. Mi törté nik valójában? Elsõké nt értelmezni ke ll a jobb o ldalt, a zaz ki kell értékelni x –et, ho zzá kell ad ni tízet és eltá rolni a veremben. Ezutá n ismét kiértékeljük x –et, e zúttal a bal olda lo n.
Szere ncsére van megoldás, mégpedig az ún. rö vid forma. A fenti so rból ez lesz:

```
x += 10 ;
```

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

Az igazság hoz a zért hozzátar tozik, hogy a fordítóprog ram elvi leg felisme ri a fe nt felvá zolt szituációt és a rövid formá va l egye nér tékü IL –t készít belõle (más kérdés, hogy a forráskód íg y viszont szebb és o lvashatóbb).

A probléma ugya naz, de a megoldás más a kö vetkezõ esetbe n:

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

Szemmel lá thatóan ugyana z a baj, azonban az eggyel való növelésre -csökkentésre va n önä lló operá tor unk:

```
++x/--x;
x++/x--;
```

Ebbõl az operátorból rögtön két verziót is kapunk, prefixes (+/-- elõl) és postfixes formát. A prefixes alak pontosan azt teszi amit elvárunk tõle, azaz megnöveli(csökkenti) az operandusát egyel.
A postfixes forma egy kicsit bonyolultabb, elsõként létrehoz egy átmeneti változót, amiben eltárolja az operandusa értékét, majd megnöveli eggyel az operandust, végül visszaadja az átmeneti változót. Ez elsõre talán nem tûnik hasznosnak, de vannak helyzetek amikor lényegesen megkönnyíti az életünket a használa.
Attól függõen, hogy növeljük vagy csökkentjük az operandust inkrementális 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 -/+: 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 elõjeles típusokon használhatjuk mivel az operátor int típussal tér vissza (akkor is, ha pl. byte típusra alkalmaztuk). A következõ program le sem fordul:

```
using System;

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

typeof: az operandusa típusát adja vissza:

```
using System;

class Program
{
```

```
        static public void Main()
        {
                int x = 143;
                if(typeof(int) == x.GetType())
                {
                        Console.WriteLine("x típusa int");
                }
        }
}
```

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

sizeof: a sizeof operátor a „paraméterként" 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 (4) fog kiírni, hiszen az int típus 32 bites azaz 4 byte méretû típus.

A programot az unsafe kapcsolóval kell fordítanunk:

csc /unsafe main.cs

6 Vezérlési szerkezetek

Vezérlési szerkezetnek a program utasításainak sorrendiségét szabályozó konstrukciókat nevezzük.

6.1 Szekvencia

A legegyszerûbb vezérlési szerkezet a szekvencia. Ez tulajdonképpen egymás után megszabott sorrendben végre hajtott utasításokból áll.

6.2 *Elágazás*

Gyakran elõfordul, hogy meg kell vizsgálnunk egy állítást, és attól függõen, hogy igaz vagy hamis más-más utasítást kell végre hajtanunk. Ilyen esetekben elágazást használunk:

```
using System;

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

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

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

```
using System;

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

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

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

Ez a program pontosan ugyanazt csinálja mint az elõzõ, de van egy nagy különbség a kettõ közt: mindkét feltételt ki kell értékelnie a programnak, hiszen két különbözõ szerkezetrõl beszélünk (ez egyúttal azzal is jár, hogy a feltételtõl függõen mindkét állítás lehet igaz).

Arra is van lehetõségünk, 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
        {
            Console.WriteLine("x értéke 10");
        }
        else if (x == 12) //Vagy ha x == 12
        {
            Console.WriteLine("x értéke 12");
        }
        else //De ha egyik sem
        {
            Console.WriteLine("x értéke nem 10 vagy 12");
        }
    }
}
```

A program az elsõ olyan ágat fogja végrehajtani amelynek a feltétele teljesül (vagy ha egyik feltétel sem bizonyul igaznak, akkor az else ágat – ha van).
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 egyszerûbb 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. „Alapesetben” a break utasítással kilépünk a switchbõl:

```
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 llatot!");
                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++ nyelvtõl eltérõen 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 llat!");
                break;
        }
    }
}
```

A break utasításon kívûl 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 llat!");
                break;
        }
    }
}
```

6.3 Ciklus

Amikor egy adott utasítássorozatot egymás után többször kell végrehajtanunk, akkor cilust használunk. A C# négyféle ciklust biztosít számunkra.
Az elsõ az ún. számlálós ciklus (nevezzük for-ciklusnak). Nézzük a következõ 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? Mielõtt ezt elmondanám elõször 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 elsõ ránézésre eléggé összetett, de ez ne tévesszen meg minket, valójában nem az. Mindössze három kérdésre kell választ adnunk: Honnan? Hová? 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 kezdõértéket adtunk neki.

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 következõ 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 Hová?! 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 ennél 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 amig i kisebb tíznél és nem egyenlõ néggyel. Értelemszerûen 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 adjuk meg, hogy milyen módon növeljük (vagy csökkentjük) a ciklusváltozót. 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 írjuk ki a képernyõre.
Most már megtudjuk válaszolni, hogy az elsõ programunk mit csinál: nullától kilencig kiírja a számokat.

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.
A számlálós ciklust végteleníthetjük, ha nem adunk meg ciklusváltozót és feltételt:

```
using System;

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

Ez a forráskód lefo rdul, de figyelme zte tést kap unk (war ning), hogy „g ya nús” kódot észle lt a fordító.

A „program” futását a Ctr l+C billentyûkombinációval állíthatjuk le , ha parancssorból futta ttuk.

Második kliensünk a z elõlteszte lõs ciklus (mosta ntól hívjuk while-ciklusnak), amely onnan kapta a ne vét, hogy a ciklusmag végrehajtása elõ tt e lle nõr zi a ciklusfelté telt , ezért elõ fo rdulha t az is, hogy a ciklus egyszer sem fut le:

```
using System ;

public class Program
{
    static public void Main ()
    {
        int i = 0; //ciklusv ltozó deklar ció
        while ( i < 10 ) //ciklusfeltétel
        {
            Console . WriteLine ( "i értéke: {0}" , i );
            ++ i ; //ciiklusv ltozó n velése
        }
    }
}
```

A program ug yana zt csi nálja mint az elõzõ , viszo nt itt jól láthatóa n elkülönülnek a ciklusfe lté telér t felelõs utasítások (ke zdõérték, ciklusfe lté tel, nö vel/csökke nt).

Mûködését teki ntve az elö ltesztelõs ciklus haso nlít a számlá lósra (mindkettõ elõször a c iklusfeltételt ellenõrzi), de e löbbi sokkal r ugalmasabb, mivel több le hetõségünk va na ciklusfeltétel meg választására.

A válto zó ér tékének kiíratásánál a Conso le.WriteLi ne egy másik verzióját használtuk, amely ún. for mátumstri nget kap paraméteré ûl. Az elsõ paramé terbe n a kapcsos zárójelek kö zt megadhatjuk, hogy a további paraméterek közül melyiket helyettesítse be a helyére (nullátó l számo zva).

A harmadik verse nyzõ követke zik, õt há tultesztelõs ciklusnak hívják (leg ye n do-while), nem ne héz kitalálni, hogy a zért kapta ezt a ne vet mert a ciklusmag végrehajtása utá n elle nõr zi a ciklus feltételt, íg y legalább eg yszer 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 );
    }
}
```

A ciklusváltozó neve konvenció szerint i (az angol iterate – ismételni szóból). Amennyiben több ciklusváltozót használunk (pl. egymásba ágyazva) akkor ajánlott rendre i, j, k, stb... névre keresztelni õket.

Végül, de nem utolsósorban a foreach (neki nincs külön neve) ciklus következik. Ezzel a ciklussal végigiterálhatunk egy tömbön vagy gyûjteményen, illetve minden olyan objektumon, ami megvalósítja az IEnumerable és IEnumerator interfészeket (interfészekrõl egy késõbbi fejezet fog beszámolni, ott lesz szó errõl a kettõrõl is).
A példánk most nem a már megszokott „számoljunk el kilencig" lesz, helyette végigmegyünk egy stringen:

```
using System;

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

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

A cilusfejben felveszünk egy char típusú változót (egy string karakterekbõl 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 gyûjtemény aktuális elemének értékét. Épp ezért egy foreach ciklus nem módosíthatja egy gyûjtemény elemeit (le sem fordul ebben az esetben).

A foreach ciklus kétféle módban képes mûködni: 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 mûködésével az interfészekrõl szóló fejezet foglalkozik, többek között megvalósítunk egy osztályt, amelyen a foreach képes végigiterálni (azaz megvalósítjuk a fentebb említett két interfészt).

6.3.1 Yield

A yield kifejezés lehetõvé teszi, hogy egy ciklusból olyan osztályt generáljon a fordító amely megvalósítja az IEnumerable interfészt ezáltal pedig használható legyen pl. egy 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);
        }

        Console.ReadKey();
    }
}
```

A yield mûködési elve a következõ : az legelsõ metódushívásnál a ciklus megtesz egy lépést, ezután „kilépünk” a metódusból – de annak állapotát megõrizzük, azaz a következõ 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, ezekrõl egy késõbbi fejezet szól.

A több processzormaggal rendelkezõ számítógépek teljesítményének kihasználása céljából a Microsoft elkészítette a Task Parallel Library –t (illetve a PLINQ –t, errõl egy késõbbi fejezetben), amely a .NET 4.0 verziójában kapott helyet, ezért ehhez a fejezethez a C# 4.0 változata szükséges.
A TPL számunkra érdekes része a párhuzamos ciklusok megjelenése. A NET 4.0 a for és a foreach ciklusok párhuzamosítását támogatja a következõ 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 elsõ paramétere a ciklusváltozó kezdõértéke, második a maximumérték, míg a harmadik helyen a ciklusmagot jelentõ Action<int> generikus delegate áll, amely egyetlen bemenõ paramétere a ciklusváltozó aktuális értéke.
A ForEach két paramétere közül az elsõ az adatforrás, míg a második a ciklusmag.

Mindkét ciklus számos változattal rendelkezik, ezek megtalálhatóak a következõ MSDN oldalon:
http://msdn.microsoft.com/en-us/library/system.threading.tasks.parallel_members.aspx

7 Gyakorló feladatok

7.1 *Szorzótábla*

Készítsünk szor zó táblát! A program vagy a para ncssori paraméterként megadott számot használja, vagy ha ilyet nem adtunk meg, akkor generáljo n egy véle tle nszámo t.

Megoldás (7/Mtable.cs)

Elsõként készítsük le a program vá zá t:

```
using System ;

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

    }
}
```

Vegyük észre, hogy a Main metód us kapo tt egy paraméte rt, mégpedig eg y string típ usú ele mekbõl álló tömböt (tömbökrõl a köve tkezõ fe jezetek adnak több tájékoztatást, most ez nem annyira lesz fo ntos) . Ebben a tömbben lesznek a z ún. parancssori p araméterei nk.
De mi is az a parancssori paraméter? Egy nagyo n egyszer û példát nézzünk meg, azt amikor lefo rd ítunk egy C# fo rráskódot:

csc main.cs

Ebben az esetbe n a csc a fordítóprogram ne ve , míg a forráskódot tar talma zó file neve pedig a paraméter . Ha e zt vesszük alap ul, akkor a z args tömb eg ye tlen e lemet tartalma zna, mégpedig a „mai n.cs” – t.
A következõ lépésben fejlesszük tovább a programot, hog y ír ja ki a paraméterként megadott szám kétszeresét. Ehhez még szükség ünk va n ar ra is, hog y szám típ ussá alak ítsuk a paramétert, hiszen azt stri ngként kapjuk meg. Erre a felada tra az int.Parse metódust használjuk majd, amely szá mmá ko nvertá lja a paraméte rreké nt kapott szö veget (persze csak akkor ha ez lehe tséges).
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 elsõ parancssori paraméter – a megadott szám – a nulladik helyen lesz.
A programot most így tudjuk futtatni:

```
main.exe 12
```

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 utasításokat:

```
using System;

class Program
{
	static public void Main(string[] args)
	{
		if(args.Length == 0)
		{
			Console.WriteLine("Adj 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 õt tartalmazó metódust hívta (ez a metódus jelen esetben a Main, õt pedig mi – vagyis inkább az operációs rendszer – hívta, azaz a 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 következõ lépésben a helyett, hogy kilépünk ha nincs paraméter, inkább generálunk egy véletlenszámot. 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 = new Random();
            number = r.Next(100);
        }
        else
        {
            number = int.Parse(args[0]);
        }

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

Véletle nszámot a Next metódussal ge nerá ltunk, a fe nti formájába n 0 és 100 kö zö tt generál gy számo t, de használhatjuk így is:

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

Ekkor 10 és 100 kö zötti lesz a szám.

Már nincs más dolgunk, mint meg ír ni a feladat lé nyegét, a szor zótáb lát :

```
using System;

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

        if (args.Length == 0)
        {
            Random r = 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);
        }

        Console.ReadKey();
    }
}
```

7.2 *Számológép*

Készítsünk eg y egyszer û számológépet! A program indításakor kérjen be két számot és egy m ûveleti jelet, majd írja ki a z eredményt.
Ezutá n bõvítsük ki a programot, hog y a két számo t illetve a mûveleti jelet parancsso ri paraméterké nt is megadhassuk (ekkor nincs külön m ûveletvá lasztó me nü, ha nem írjuk ki rögtön az eredmé nyt): ./main.e xe 12 23 + (az eredmé ny pedig 35 lesz).

Megoldás (7/Calculator.cs)

Most is két részbõl áll a program unk, elõször ho zzá kell jutnunk a számokhoz és az operátorhoz, ma jd – természe tesen – elvégezzük a meg felelõ mûvelete t és kiírjuk a z eredményt.

Ezútta l is a szükséges változók deklarációjával ke zdjük a program írást, három darab kell, két numerikus (leg yen most int) és eg y karaktertípus. Ezután bekérjük a felhasználótó l a szükséges adato kat, vag y pedig felhaszná ljuk a p ramétereke t. Ez utóbbi esetben végeezünk eg y kis hibaellenõr zést, vizsgáljuk meg, hog y pontosa n három paramétert kaptunk –e? A forráskód eddig így né z ki:

```
using System;

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

		if (args.Length == 0)
		{
			Console.WriteLine("Az els  sz m: ");
			x = int.Parse(Console.ReadLine());

			Console.WriteLine("A m sodik sz m: ");
			y = int.Parse(Console.ReadLine());

			Console.WriteLine("A muvelet(+, -, *, /): ");
			op = Convert.ToChar(Console.Read());
		}
		else
		{
			if (args.Length != 3)
			{
				Console.WriteLine("Nem megfelel  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 egyszerû dolgunk van, mindössze ki kell számolnunk az eredményt. Ezt nyílván a beolvasott operátor alapján fogjuk megtenni, ebben a helyzetben pedig a legkézenfekvõbb, 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 muvelet eredménye: {0}", result);
```

Amire figyelni kell az az, hogy a result változó kapjon kezdõértéket ellenkezõ esetben ugyanis nem fordul le a program (uninitialized variable kezdetû hibaüzenetet kapunk). Ez azért van így, mert a változódeklaráció és az utolsó sorban lévõ kiíratás között nem biztos, hogy megtörténik a változódefiníció. Ugyan az értéktípusok bizonyos helyzetekben automatikusan nullértéket 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 *Kõ – Papír – Olló*

Készítsünk kõ-papír-olló játékot! Ebben az esetben is használjuk a véletlenszámgenerátort. 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.

Megoldás (7/SPS.cs)

A programunk lényegében három részbõl fog állni: elsõként megkérdezzük a felhasználótól, hogy mit választott, majd sorsolunk a „gépnek” is valamit, végül pedig kiértékeljük az eredményt.

Elsõ dolgunk leg yen, hogy deklarálunk öt darab vá ltozót, egy Ra ndom objektumo t, két stringet (ezekben tároljuk, hogy mit választottak a „ versenyzõk”), és két b yte típ ust (e zekbe n pedig az ered ményeket tartjuk számon) (ebben az esetben elég lesz a byt e is, feltesszük, hogy se nki nem fog több szá zszo r játsza ni eg ymás után).
Szükségünk lesz még egy logikai típusra is, e zzel fogjuk jelezni a prog ram nak, hogy akarunk –e még játsza ni.
Készítsük el a program vázá t a változódeklarációkkal, illetve a fõcikluss al (a ciklustö rzsben kérde zzünk rá, hogy aka runk –e még já tszani):

```
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 j tszani? i/n" );
            if ( Console . ReadKey ( true ). KeyChar == 'n' ) { l = false ; }
        } while ( l );
    }
}
```

Most kér jük be az adatokat:

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

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

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

Ez a kódrészlet természetesen a ciklustörzsbe a kérdés elé kerül. Az egyes lehetõségeket a k/p/o billentyûkre dró toztuk.
Már csak az értékelés van hátra:

```
if ((playerChoice == "k" && compChoice == "papír") ||
    (playerChoice == "papír" && compChoice == "olló") ||
    (playerChoice == "olló" && compChoice == "k"))
{
    Console.WriteLine("Vesztettél! Az ll s:\nSz mítógép:
{0}\nJ tékos:{1}", ++compScore, playerScore);
}
else if (playerChoice == compChoice)
{
    Console.WriteLine("D ntetlen! Az ll s:\nSz mítógép:
{0}\nJ tékos:{1}", compScore, playerScore);
}
else
{
    Console.WriteLine("Nyertél! Az ll s:\nSz mítógép: {0}\nJ tékos:{1}",
compScore, ++playerScore);
}
```

7.4 *Számkitaláló játék*

Készítsünk számkitaláló játékot, amely lehetõséget ad kiválasztani, hogy a felhasználó próbálja kitalálni a program által „sorsolt" számot, vagy fordítva. A kitalált szám legyen pl. 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 használjuk véletlenszámgenerátort a szám létrehozására. A gépi játékos úgy találja ki a számot, hogy mindig felezi az intervallumot (pl.elõször 50 –t 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 kiv lasztjuk, hogy ki v laszt sz mot

        if ( /* A j tékos v laszt */ )
        {
                // A sz mítógép megprób lja kital lni a sz mot
        }
        else    // A sz mítógép v laszt
        {
                // A j tékos megprób lja kital lni a sz mot
        }

        // Megkérdezzük a j tékost, hog akar-e még j tszani
}
```

Nem néz ki rosszul, de a probléma az, hogy a két feltétel blokkja nagyon el fog „hízni”, emiatt pedig kevésbé olvasható lesz a forr áskód. Ez persze nem olyan nagy baj, de ahhoz elég, hogy valami mást próbáljunk ki. A procedurális és alacsonyszintû nyelvek az ilyen feladatokat „ugró” utasításokkal oldják meg, vagyis a forráskódban elhelyezett címkék között ugrálnak (tulajdonképpen a magas szintû nyelveknél is ez történik, csak ezt mi nem látjuk mivel az if/switch/stb... elfedi elõlünk). Ez a módszer a magas szintû nyelvek esetén nem igazán ajánlott (fõleg mert megvannak az eszközök az ugrálás kikerülésére), de jelenleg nem is használjuk ki a nyelv adta lehetõségeket, ezért szabad „rosszalkodnunk”. Írjuk át a fenti vázat egy kicsit:

```
using System;

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

            Console.WriteLine("V lassz j tékmódot!");
            Console.WriteLine("1 - Te gondolsz egy sz mra");
            Console.WriteLine("2 - A sz mítógép gondol egy sz mra");

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

            PLAYER:

            goto END;


            COMPUTER:

            goto END;

        END:
            Console.WriteLine("\nAkarsz még j tszani? 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 érthetõvé vált a kód (persze ennél nagyobb terjedelmû 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 billentyût 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 megfelelõ helyre tenni:

```
END:
    Console.WriteLine("\nAkarsz még j tszani? 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. Elsõként azt a szituációt implementáljuk, amikor a játékos próbálja kitalálni a számot mivel ez az egyszerûbb. Szükségünk lesz természetesen változókra is, de érdemes átgondolni, hogy úgy vegyük fel õket, hogy mindkét programrész használhassa õket. 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, nek iadjunk azonnal nulla értéket. Ezt a kettõt deklaráljuk egyelõre a rögtön a START címke elõtt. 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 sz m ennél nagyobb!");
        }
        else if (x > number)
        {
            Console.WriteLine("A sz m ennél kisebb!");
        }
        else
        {
            Console.WriteLine("Nyertél!");
            goto END;
        }

        ++i;
    }

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

Ezt nem okozhat gondot megérteni, lépjünk a következõ állomásra, ami viszont kicsit nehezebb lesz. Ahhoz, hogy megfelelõ 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 uto lsó tippre már e lég szûk lesz az a szám halma z amibõl választhatunk (persze íg y egy kicsit szerencse játék is lesz). Né zzünk meg egy példát: a go ndolt szám legye n a 87 és tudjuk, hogy a szám egy és szár között van. Az elsõ tippünk 50 lesz, amire természetese n a zt a választ kapjuk, hog y a szám ennél nagyobb. Má r csak 50 lehe tésges szám maradt, ismét fele zünk, a következõ tippünk így a 75 lesz. Ismét azt kapjuk vissza, hog y e z nem elég. Ismé t fe lezünk, még hozzá maradék nélkül vag yis tize nkettõt adunk hozzá a hetve nöthö z és íg y ki is találtuk a gondolt számot.

Most már kö nnyen fel tudjuk ír ni, hogy mit kell tennie a számítógépnek: a z elsõ négy kísé rletné l fele zzük a z inter vallumo t az utolsó körben pedig tippelük. Nézzük a kész kódot:

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

        x = 50 ;
        int min = 0;
        int max = 100 ;
        while ( i < 5)
        {
                Console . WriteLine ( "A sz mítógép szerint a sz m {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 sz mítógép kital lta a sz mot
:)" );
                                goto END;
                }

                ++ i ;
        }

        Console . WriteLine ( "A sz mítógép nem tudta kital lni a sz mot :(" );
goto END;
```

A min illetve ma x változókka l tartjuk szá mon az i nterva llum a lsó illetve felsõ határá t . Az x válto zóban táro ljuk az aktuális tippet neki meg is adtuk a kezdõértéket.
Egy példán keresztûl nézzük meg , hog y hog yan mûködik a kódunk. Legyen a gondolt szám ismét 87. A számítógép elsõ tippje 50 mi erre azt mondjuk, hog y a szám e nnél nagyobb a zért a switch n ága fog beindulni. Az intervallum alsó határa

ezután x (vag yis 50) lesz, mi vel tudjuk, hogy a szám ennél biztosan nag yobb. A felsõ határ nyílván nem válto zik már csak a z új x –et kell kiszámolni, vagyis x – hez ho zzá kell adni a felsõ és alsó ha tárok különbségének a felé t: (100 – 50) / 2 = 25 (ellenkezõ esetben pedig nyílván le kell vonni ugyane zt).
Amirõl még nem beszéltünk a z az a feltéte l, amelyben x eg yenlõ ségét vizsgáljuk három mal. Ez az elágazás fogja visszaadni az utolsó tippet a véle tle nszámgenerátorra l a megfelelõ i nte rvallumok között.

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 castolást) kell elvégeznünk. Kétféleképpen konvertálhatunk: implicit és explicit módon. Az elõbbi esetben nem kell semmit tennünk, a fordító elvégzi helyettünk. Implicit konverzió általában „hasonló” típusokon mûködik, szinte minden esetben a szûkebb 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 a long a tágabb, ezért a konverzió gond nélkül megy. Egy implicit konverzió minden esetben sikeres és nem jár adatvesztéssel.

Egy explicit konverzió nem feltétlenül fog mûködni és adatvesztés is felléphet. Vegyük a következõ példát:

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

A byte szûkebb típus mint az int (8 illetve 32 bitesek), ezért explicit konverzitó hajtottunk végre, ezt a változó elõtti zárójelbe írt típussal jelöltük. Ha lehetséges a konverzió, akkor végbemegy, egyébként a fordító figyelmeztetni fog. Vajon mennyi most az y változó értéke? A válasz elsõre meglepõ 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 elsõ 8 bitjét adhatjuk át az y –nak, ami pont 44.

8.1 *Ellenõrzött konverziók*

A programfejlesztés alatt hasznos lehet tudnunk, hogy minden konverzió gond nélkül lezajlott vagy sem. Ennek ellenõrzésére ún. ellenõrzött konverziót fogunk használni, amely kivételt dob (errõl 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. Figyeljünk arra, hogy ilyen esetekben csak a blokkon belül deklarált, statikus és tag változókat vizsgálhatjuk.

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

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

Az ellenõrzés kikapcsolását is meg tehetjük az unchecked használatával:

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

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

8.2 *Is és as*

Az is operátort típusok futásidejû lekérdezésére 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 legnagyobb haszna az, hogy le tudjuk kérdezni, hogy egy adott osztály megvalósít–e egy bizonyos interfészt (errõl késõbb).

Párja az as az ellenõrzés mellett egy explicit típuskonverziót is végrehajt. Az as operátorral csakis referenciatípusra 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";

		int x = 10;
		object c = x;

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

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

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

Amennyiben ez a konverzió nem hajtható végre a célváltozóhoz null érték rendelõdik (ezért is van a referenciatípusokhoz 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 kimenet a következõ lesz:

97
98
99
100
…

A kis a betû 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 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 egyértelmûen mutat egy index (egész szám). A tömbök referenciatípusok. A C# mindig folytonos memóriablokkokban helyezi el egy tömb elemeit.
Tömböt a következõképpen deklarálhatunk:

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

Ez a tömb tíz darab int típusú elem tárolására alkalmas. A tömb deklarációja után az egyes indexeken lévõ elemek automatikusan a megfelelõ nullértékre inicializálódnak (ebben az esetben 10 darab nullát fog tartalmazni a tömbünk). Ez a szabály referenciatípusoknál kissé máshogy mûködik, 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 mûvelet), míg referenciatípusoknál ugyanez NullReferenceException típusú kivételt fog generálni.
Az egyes elemekre az indexelõ operátorral (szögeletes zárójel -> [ ]) és az elem indexével (sorszámával) hivatkozunk. A számozás mindig nullától kezdõdik, így a legutolsó elem indexe az elemek száma mínusz egy. A következõ 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 nevû tulajdonságát használtuk, amely visszaadja a tömb hosszát. Látható az indexelõoperátor használata is, az array[i] a tömb i –edik elemét jelenti. Az indexeléssel vigyázni kell, ugyanis a fordító nem ellenõrzi fordítási idõben az indexek helyességét, viszont helytelen indexelés esetén futás idõben IndexOutOfRangeException kivételt fog dobni a program.

Egy tömböt akár a deklaráció pillanatában is feltölthetünk a nekünk megfelelõ 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 késõbbiekben nem lehet változtatni. Dinamikusan bõvíthetõ adatszerkezetekrõl a **Gyûjtemények** címû fejezet szól.

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

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

9.1 *Többdimenziós tömbök*

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

12, 23, 2
A = [13, 67, 52]
45, 55, 1

Ez egy kétdimenziós tömbnek (mátrix a neve – ki hinné?) felel meg, az egyes elemekre két indexxel hivatkozunk, elsõ helyen a sor áll utána az oszlop. Így a 45 indexe: [2, 0] (ne feledjük, még mindig nullától indexelünk).
Multidimenziós tömböt a következõ módon hozunk létre C# nyelven:

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

Ez itt egy 3x3 –as mátrix, olyan mint a fent látható. Itt is összekötheljük 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 már pontosan olyan mint amit már láttunk. Az elemszám most is meghatározott, nem változtatható.
Nyílván nem akarjuk mindig kézzel feltölteni a tömböket, viszont ezúttal nem olyan egyszerû a dolgunk, hiszen egy ciklus biztosan nem lesz elég ehhez, vagyis gondolkodnunk kell: az index elsõ tagja a sort a második az oszlopot adja meg, pontosabban az adott sorban elfoglalt indexét. Ez a lapján pedig jó ötletnek tûnik, 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 külsõ ciklus a sorokon megy át a belsõ 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, ezt nem okozhat gondot megírni. A tömbök GetLength() metódusa a paraméterként megadott dimenzió hosszát adja vissza (nullától számozva), tehát a példában az elsõ 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 belsõ tömbök hossza tetszés szerint megadható:

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

Készítettünk egy három sorral rendelkezõ tömböt, azonban a sorok hosszát (az egyes sorok nyílván önálló vektorok) ráérünk késõbb 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;
            }
        }
    }
}
```

Véletle nszámge nerátorral adjuk meg a belsõ tömbök hosszát, persze érte lmes kereteken be lül. A belsõ ciklusba n jól lá tható, hogy a tömb elemei valóban tömbök, hisze n használtuk a Length tulajdonságot (persze a hagyományos többdimenziós tömbök esetében is e z a helyzet, de ott nem lenne ér telme külön elér hetõ vé tenni az egyes soroka t).

Az inicializálás a követke zõképpen alakul ebbe n az esetben:

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

10 Stringek

A C# beépített karaktertípusa (char) egy Unicode karaktert képes eltárolnimelynek mérete két byte. A szintén beépített string típus ilyen karakterekbõl áll (tehát char –ként hivatkozhatunk az egyes betûire).

```
using System;

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

A látszat ellenére a string referenciatípus, viszont nem kötelezõ használnunk a new operátort.
Egy string egyes betûire az indexelõ operátorral hivatkozhatunk (vagyis minden stringet kezelhetünk tömbként is):

```
using System;

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

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

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

Az indexelõ 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 létezõ 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 fõleg akkor okozhat (teljesítmény)problémát, ha sokszor van szükségünk erre a mûveletre.

10.1 *Metódusok*

A :NET számos hasznos metódust biztosít a stringek 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 = "m sik";

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

        if (x == 0)
        {
            Console.WriteLine("A két string egyenl ");
        }
        else if (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 egyenlõ 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', '0' };

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

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

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 elsõ paraméterének megfelelõ karaktereket lecseréli a második paraméterre. A Trim() metódus a string elején és végén lévõ karaktereket vágja le, a Substring() kivág egy karaktersorozatot, paraméterei a kezdõ és végindexek (van egyparaméteres változata is, ekkor a csak a kezdõindexet adjuk meg és a végéig megy). Az Insert()/Remove() metódusok hozzáadnak illetve elvesznek a stringbõl. Végül a ToLower() és ToUpper() metódusok pedig kis- illetve nagybetûssé alakítják az eredeti stringet.
Fontos megjegyezni, hogy ezek a metódusok soha nem az eredeti stringen végzik a módosításokat, hanem egy új példányt hoznak létre és azt adják vissza.

10.2 StringBuilder

Azt már tudjuk, hogy amikor módosítunk egy stringet akkor automatikusan egy új példány jön létre a memóriában, ez pedig nem feltétlenül „olcsó” mûvelet.
Ha sokszor (10+ legalább) van szükségünk erre, akkor használjuk inkább a StringBuilder típust, ez automatikusan lefoglal egy nagyobb darab memóriát és ha ez sem elég, akkor allokál egy nagyobb területet és átmásolja magát oda. A StringBuilder a System.Text névtérben található:

```
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 az alapértelmezett tizenhat karakternek foglal helyet). Az Append() metódussal tudunk karaktereket (vagy egész stringeket) hozzáfûzni.

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 elsõ elõfordulás).

Megoldás (11/MinMax.cs)

A mininum/maximumkeresés a legalapvetõbb algoritmusok egyike. Az alapelv rendkívûl egyszerû, végigmegyünk a tömb elemein és minden elemet összehasonlítunk az aktuális legkissebbel/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 kezdõértéket is adtunk, ezeket úgy kell megválasztani, hogy biztosan kissebbek illetve nagyobbak legyenek a tömb elemei.

Értelemszerûen mivel minden elemet kötlezõen meg kell vizsgálnunk ez az algoritmus nem túl gyors nagy elemszám esetében.

11.2 Szigetek

Egy szigetcsoport fölött elrepülve bizonyos idõközönként 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 billentyûzetrõl vagy – ha van – a parancssori paraméterbõl kap meg. A feladatok:

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

Megoldás (11/Islands.cs)

A megoldásban csak az adatok feldolgozását nézzük meg, a neolvasásuk nem okozhat gondot. A szigeteken végzett méréseket a data ne vû string típusú válto zóban tároltuk el.
A két fe ladatot egyszer re fogjuk megoldani, mi vel a programunk a követke zõ elve n alapul: a data stringe n fogunk keresztûlme nni egy ciklus segítségével. Ha a string adott inde xén egyest találunk, akkor eli nd ítunk egy másik ciklust, amely attó l a z indextõl megy egésze n addig, am íg nulláho z nem ér . Ekkor egyrészt megnö veljük eggyel a szigetek számát számontartó válto zót, másrészt tudni fogjuk, hogy milye n hosszú volt a sziget és össze haso nlíthatjuk az eddigi eredményekkel. Nézzük is meg ezt 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, a z elsõ a belsõ ciklus fe ltétele: ellenõrizzük, hogy még a szigetet mérjük és azt is, hogy nem -e értünk a string végére. Ami fontos, az a felté telek sorrendje : mivel tudjuk, hogy a feltételek kiértéke lése balról jobbra halad, ezért elõször azt kell vi zsgálnunk, hogy helyes inde xe t használunk ellenke zõ esetben ug yanis kivételt kapnánk.
A másik érdekesség a két ciklusválto zó. Amikor befeje zzük a belsõ ciklust a külsõ ciklusválto zót új pozívióba kell helyeznünk, mégho zzá oda ahol a belsõ ciklus abbahagyta a vi zsgálatot.

11.3 *Átlaghõmérséklet*

Az é v minde n napján megmér tük a z átlag hõmérsékletet az eredményeket pedig egy mátri xban tároljuk (az egyszer ûség kedvéér t tegyük fel, hog y mi nden hó nap har minc

napos, az eredmé nyeket pedig véletlenszámgenerátorral (ésszer û kereteke n belül) sorsoljuk ki).

- Keressük meg az é v legme legebb/leghidegebb nap ját.
- Adjuk meg a z é v legme legebb/leghidegebb hónapját.
- Volt –e egymást köve tõ öt nap (egy hónapon belül) amikor m ínusz fokot mértünk?

Megoldás (11/Tempe rature.cs)

Ehhez a fe ladatho z ne m tarto zik írásos megoldás, tulajdonképpen egy minimum/maximumkiválasztásról van szó csak éppen kétdimenziós tö mbre (viszont a jegyzethe z csatolva van egy lehe tséges megoldás) .

11.4 *Buborékrendezés*

Valósítsuk meg egy tömbö n a buborékrende zést.

Megoldás (11/BubbleSort.cs)

A buborékos rendezés egy alapvetõ rendezõ algoritm us, amelynek alapelve, hog y a kissebb elemek – buborék módjára – felszi várog nak, míg a nagyobb elemek lesüllyed nek. Ennek a z algoritm usnak többféle impleme ntációja is léte zik, mi most két változatát is megvizsgáljuk. Az elsõ íg y 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;
        }
    }
}
```

Kezdjü ka belsõ cikl ussal. Ez a tömb végérõ l fog visszafelé menni és cserélgeti a z elemeket, hog y a legkissebbet vigye to vább magá val. Legye n pl. a tömb utolsó né hány ele me:

10 34 5

Fogjuk az 5 –õt (arra y[j]) és összehasonlítjuk a z elõtte lé võ elemmel ami a 34 (array[j-1]). Mi vel nagyobb nála, ezért megcseréljük a kettõ t:

10 5 34

Ezutá n csökke ntjük a ciklusválto zót ami most megint a z eddigi legkissebb elemre az 5 –re fog m utatni és cseré lhetjük tovább. Természetesen ha kissebb elem t ta lálunk

akkor ezután õt fogjuk tovább vinni egészen addig am íg a legkissebb elem elfogla lja
a tömb elsõ i ndexé t. Itt jön képbe a külsõ ciklus, ami a zt biztosítja, hog y a rende ze tt
elemeket már ne vi zsgáljuk, hiszen a belsõ ciklus minde n futásakor a tömb elejére
tesszük az aktuális legkissebb el e met.

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 lé nyegében ugyanarró l van szó , csak most elõ lrõl vizsgá ljuk az ele meket.
Nem ár t tudni, hog y a buborékos rendezés csak kis elemszám esetében haték ony,
nagyjábó l O(n^2) nag yságre ndû.

Az O() (ún. nagy ordó) je lölést használjuk eg y algoritmus futásidejének
megbecsülésére (illetve haszná lják a ma tematika más területei n is).

12 Objektum-orientált programozás - elmélet

A korai programozási nyelvek nem az adatokra, hanem a mûveletekre helyezték a hangsúlyt. Ekkoriban még fõleg 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 õket.
Az elsõ objektum-orientált programozási nyelv a Simula 67 volt. Tervezõi Ole-Johan Dahl és Kristen Nygaard hajók viselkedését szimulálták, ekkor jött az ötlet, hogy a különbözõ hajótípusok adatait egy egységként kezeljék, így egyszerûsítve a munkát.

Az OOP már nem a mûveleteket helyezi a középpontba, hanem az egyes adatokat (adatszerkezeteket) és köztük lévõ kapcsolatot (hierarchiát).
Ebben a fejezetben az OOP elméleti oldalával foglalkozunk, a cél a paradigma megértése, gyakorlati példákkal a következõ részekben találkozhatunk (szintén a következõ részekben található meg néhány elméleti fogalom amely gyakorlati példákon keresztül érthetõbben megfogalmazható, ezért ezek csak késõbb lesznek tárgyalva, pl.: polimorfizmus).

12.1 UML

Az OOP tervezés elõsegítésére hozták létre az UML – t (Unified Modelling Language). Ez egy általános tervezõeszköz, a célja egy minden fejlesztõ által ismert közös jelrendszer megvalósítása. A következõkben az UML eszközeit fogjuk felhasználni az adatok közti relációk grafikus ábrázolásához.

12.2 *Osztály*

Az OOP világában egy osztály olyan adatok és mûveletek összessége, amellyel leírhatjuk egy modell (vagy entitás) tulajdonságait és mûködését. 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 következõképpen jelöl egy osztályt:

Kutya

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

12.3 *Adattag és metódus*

Egy objektumnak a z életciklusa sorá n megválto zhat az állapota , tulajdo nságai. Ezt az állapo tot valahog y el kell tud nunk tárolni illetve biztosítani kell a szükséges mûveleteket a tulajdo nságok megvá lto ztatásá hoz (pl. a kutya eszik(ez eg y m ûvelet), ekkor megválto zik a „jóllakottság” tulajdo nsága) .
A tulajdonságokat tároló változóka t adattag nak (vagy me zõnek), a m ûveleteket metódusnak neve zzük. A m ûveletek összességét felüle tnek is hívjuk .
Módosítsuk a diagramunka t:

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

Az adattagokat *név : típus* alakba n ábrázo ljuk, a me tódusokat pedig *név(param éterlista) : visszatérési_érték* formába n. Ezekkel a fogalmakkal eg y késõbbi fejeze t foglalkozik.

12.4 *Láthatóság*

Az eg yes tulajdonságokat, me tódusoka t nem bi ztos, hog y jó kö zszemlére bocsátani.
Az OOP egyik alapelve, hogy a fe lhasználó csak a nnyi adatot kapjon ame nnyi felté tlenül szükséges. A kutyás példába n az eszik() mûvele t magába foglalja a rágást, nyelést, emésztést is, de errõl ne m fo ntos tud nunk, csak az e vés té nye szám ít.
Ugya nígy egy tulajdo nság (adattag) esetébe n sem jó , ha mi ndenki ho zzájuk fér (az elfogadha tó, ha a közvetle n család ho zzáfér a számlám hoz, de idegenekkel nem akarom megoszta ni).
Az õs -OOP szabályai háromféle lá thatóságot fogalma znak meg, ez nyelvtõ l függõen bõvülhet, a C# látha tóságairól a követke zõ részekben lesz szó.
A háromféle láthatóság:
Public : mi ndenki láthatja (UML jelölés: +).
Private : csakis az osztályo n be lül e lérhetõ, illetve a leszárma zo tt osztályok is láthatják, de nem módosíthatják (a származtatás/öröklõdés ha marosan jön) (UML jelölés: **-**).
Protected : ug yanaz mi nt a **private** , de a leszá rmazott osztályok módosítha tják is (UML jelölés: #).
A Kutya osztály most így né z ki:

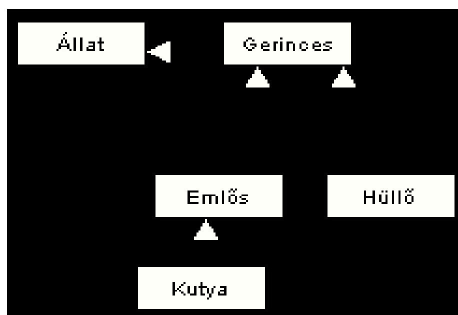
Kutya
-jollak : **int**
+eszik() : **void**

12.5 *Egységbezárás*

A „hagyományos”, nem OO programnyelvek (pl. a C) az adatokat és a rajtuk végezhetõ mûveleteket a program külön részeként kezelik. Bevett szokás ezeket elkülöníteni egy önálló forrásfile –ba, de ez még mindig nem elég biztonságos. A kettõ között nincs összerendelés, ezért más programozók gond nélkül átírhatják egyiket vagy másikat, illetve hozzáférnek a struktúrákhoz és nem megfelelõen használják 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 elõnye, hogy egy adott osztály belsõ 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 *Öröklõdés*

Az öröklõdés vagy származtatás az új osztályok létrehozásának egy módja. Egy (vagy több) már létezõ osztályból hozunk létre egy újat, úgy, hogy az minden szülõjének tulajdonságát örökli vagy átfogalmazza azokat. A legegyszerûbben egy példán keresztül érthetõ meg. Legyen egy „Állat” osztályunk. Ez egy eléggé tág fogalom, ezért szûkíthetjük a kört, mondjuk a „Gerinces Állatok” –ra. Ezen belül megkülönböztethetünk „Emlõs” –t vagy „Hüllõ” – t. Az „Emlõs” osztály egy leszármazottja lehet a „Kutya” és így tovább. Az öröklõdést 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 jelentõsége.



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

Elõfordul, hogy nem fontos számunkra a belsõ 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,

ha nem egy i nter fészrõl – fe lületrõl – beszélünk, amelynek nincsenek adattag jai csakis a mûveleteket deklarálja.
A C# rend e lkezik ö nálló interfészekkel, de ez nem minden programnye lvre iga z, ezért õk egy haso nló szerkezete t ún. absztrakt osztályokat használnak. Ezekbe n elõford ulnak adattagok is, de leginkább a felület defi niálására koncentrálnak. A C# nyelvi szinte n támogat absztrakt osztályokat is, a kettõ kö zött ugyanis lényegi külö nbség van. Az inter fészek az osztálytól függetlenek, csakis felületet biztosíta nak (például az IEnumerable és IEnumerator a foreach – nek (is) biztosítanak felületet, az nem lé nyeges, hogy milyen osztályró l va n szó). Az absztrakt osztályok viszont eg y õshöz kötik a z utódokat (erre az esetre példa az „Állat” osztály, ame lyben mo ndjuk megadunk egy absztrakt „evés” metód ust, amit az utódok meg valósítanak - egy krokodil nyílván máshogy eszik mint egy hangya, de az evés az állatokho z köthetõ vala mi, ezér t közös).

13 Osztályok

Osztályt a **class** kulcsszó segítségével deklarálha tunk:

```
using System ;

class Dog
{
}

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

    }
}
```

Látható, hog y a főprogram és a saját osztályunk e lkülönül.

Konve nció szeri nt osztá lynév mi ndig nag ybetûve l kezdõdik.

Felmer ülhet a kérdés, hogy a ford ító, honnan tudja , hogy melyik a „fõ ” osztá ly? A helyzet az, hogy a Main eg y speciális metódus, e zt a ford ító a utomatikusan felisme r i és megjelöli, mint a program belépési pontjá t. Igazából le hetséges több Main ne vû metódust is létre hozni, ekkor a ford ítóprogram nak meg kell adni (a/m ain kapcsoló segítségéve l), hogy melyik az igazi belépési pont. Ettõl függetlenül ezt a megoldást ha cs ak lehet ker üljük el.
Nézzünk eg y példát:

```
using System ;

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

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

Ezt a fo rráskódot így ford íthajuk:

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

Ekkor a „Program1 ” szö veget fogja ki ír ni a p rogram.
A fordító to vábbi kapcsolóiról tud hatunk meg többet, ha a prancssorba be ír juk, hogy:

```
csc -help
```

A fenti példában a lehetõ legegyszerûbb osztályt hoztuk létre. A C++ nyelvet ismerõk figyeljenek arra, hogy az osztálydeklaráció végén nincs pontosvesszõ.
Az osztályunkból a new operátor segítségével tudunk készíteni egy példányt.

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

A new hívásakor lefut a konstruktor, megfelelõ nagyságú hely foglalódik a memóriában, ezután 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, ettõl függetlenül rendelkezik alapértelmezett (azaz paraméter nélküli) konstruktorral. Ez igaz minden olyan osztályra, amelynek nincs konstruktora (amennyiben bármilyen konstruktort létrehoztunk akkor ez a lehetõség megszûnik).
Az alapértelmezett konstruktor legelõször meghívja a saját õsosztálya alapértelmezett konstruktorát. Ha nem származtattunk direkt módon (mint a fenti programban), akkor ez a System.Object konstruktora lesz (tulajdonképpen ez elõbb vagy utóbb mindenképpen meghívódik, hiszen az õsosztály konstruktora is meghívja a saját õsét és így tovább...
Abban az esetben, ha az õsosztály 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 nem fordul le, mivel a Base osztály csakis paraméteres konstruktorral rendelkezik.

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

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

Most viszont mûködni fog.

A base nem összekeverendõ a Base nevû osztállyal, ez egy önálló metódus, amely minden esetben az õsosztály valamely konstruktorát hívja.

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

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

Az adattagok – ha vannak – automatikusan a nekik megfelelõ nullértékre inicializálódnak (pl.: int -> 0, bool -> false, referencia- és nullable típusok -> null).
A C++ nyelvet ismerõk vigyázzanak, mivel itt csak alapértelmezett konstruktort kapunk automatikusan, értékadó operátort illetve másoló konstruktort nem. Ugyanakkor minden osztály a System.Object –bõl 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 sem lehet. A mi konstruktorunk két paramétert vár, a nevet és a kort (metódusokkal és paramétereikkel a következõ rész foglalkozik bõvebben). Ezeket a példányosításnál muszáj megadni, egyébként nem fordul le a program. Egy osztálynak paraméterlistától függõen bármennyi konstruktora lehet és egy konstruktorból hívhatunk egy másikat a this –el:

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

Ilyen esetekben a paraméter típusához leginkább illeszkedõ konstruktor fut le.
A példában a konstruktor törzsében értéket adtunk a mezõknek a this hivatkozással, amely mindig arra a példányra mutat, amelyen meghívták (a this kifejezés így minden olyan helyen használható, ahol az osztály típusára van szükség). Nem kötelezõ használni, ugyanakkor hasznos lehet, ha sok adattag/metódus van, illetve ha a paraméterek neve megegyezik az adattagokéval (ez persze nem ajánlott). A fordítóprogram automatikusan „odaképzeli" magának a fordítás során, így mindig tudja mivel dolgozik.
Az adattagok private elérésûek (ld. elméleti rész), azaz most csakis az osztályon belül használhatjuk/módosíthatjuk õket, például a konstruktorban, ami viszont publikus.
Nem csak a konstruktorban adhatunk értéket a mezõknek, 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 elõtt fut le, ez egyben azt is jelenti, hogy az 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 (fölülrõl lefelé halad).

A konstruktorok egy speciális változata az ún. másoló- vagy copy-konstruktor. Ez paramétereként egy saját magával megegyezõ típusú objektumot kap és annak értékeivel inicializálja magát. Másoló konstruktort általában az értékadó operátorral szoktak implementálni, de az operátortúlterhelés egy másik fejezet témája így most egyszerûbben 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 elsõ ráné zésre furcsa le het, mive l pri vát elérhe tõségû tagokat haszná lunk, de ezt minden go nd né lkül megtehetjük, mivel a C# a p rivát elé rhetõséget csak osztá lyon kívûl é rvényesíti, ugyano lya n típ usú o bjektumok látják egymást.
Most má r használhatjuk is az új ko nstruktort:

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

13.2 Adattagok

Az adattagok vag y me zõk olya n válto zók, amelyeket egy osztályon (vagy struktúrá n) belül deklará ltunk. Az eddigi példáinkban is használtunk már adattagokat, ilye nek voltak a Dog osztályon belüli _ name és _age vá ltozók.
Az adatta gokon haszná lhatjuk a const típusmódosító t is, ekkor a deklarációnál értéket ke ll adnunk a mezõ nek, hasonlóa n az e lõzõ feje ze tben említe tt inicializáláshoz. Ezek a me zõk pontosa n ugya núgy viselkednek mint a hagyomá nyos konsta nsok.

Egy konstans mezõt nem l e het statikusnak (statikus tagokról hamarosan) je lölni, mivel a ford ító eg yébként is úgy fogj ke zelni (ha egy adat mi nden objektumba n válto zatlan, felesleges mi nden alka lomma l külön pé ldányt készíteni belõ le), vag yis minden ko nstans adattagból globálisa n egy da rab van.

A mezõkön alka lmazható a readonly módosító is, ez két dologban külö nbözik a konstansoktó l: az értékadás elhalasztható a konstruktorig és az ér tékül ado tt kifejezés értékének nem szükséges ismertnek lennie ford ítási idõben.

13.3 *Láthatósági módosítók*

A C# nyelv ötféle módosítót ismer:

- **public** : a z osztályon/struktúrán kívûl és belül teljes mértékben hozzáfér hetõ.
- **private** : csakis a tarta lmazó osztályon belül látható, a leszárma zo ttak sem láthatják, osztályok/str uktúrák ese tében az alapértelmezés .
- **protected** : csakis a tartalma zó osztályo n és leszármazottai n belül látható.
- **internal** : csakis a tartalmazó (és a barát) assembly(ke) –n belül látható .
- **protected internal** : a protected és internal keveréke.

Ezek kö zül leggyakrabban a z elsõ há rmat fogjuk használni.

13.4 *Parciális osztályok*

C# nyelven létrehozha tunk ún. parciális osztá lyokat (par tial class) , ha egy osztá lydeklarációban haszná ljuk a partial kulcsszót (ezt minde n darb nál meg kell tennünk) . Egy parciális osztály definiciója több részbõl (tipikusan több forrásfilebó l)

áll(hat). Egy parciális osztály minden töredékének ugyanazzal a láthatósági módosítóval kell rendelkeznie, valamint az egyik résznél alkalmazott egyéb módosítók (pl. abstract) illetve õsosztály deklaráció a teljes osztályra (értsd: minden töredékre) érvényesek lesznek (ebbõl következik, hogy ezeket nem kötelezõ 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 egyidõben).
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 C# a parciális osztályokat fõként olyan esetekben használja, amikor az osztály egy részét a fordító generálja (pl. a grafikus felületû alkalmazásoknál a kezdeti beállításokat az InitializeComponent() metódus végzi, ezt teljes egészében a fordító készíti el). Ennek a megoldásnak az a nagy elõnye, hogy könnyen ki tudjuk egészíteni ezeket a generált osztályokat.

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

Egy parciális osztály darabjai ug yanabban a z assemblyben kell legyenek.

A C# 3.0 már engedélye zi parciális metódusok használatát is, ekkor a metódus deklarációja és defi níciója szétoszlik:

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

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

Parciális metódusnak ne m le het e lérhetõségi módosítója (épp e zért minde n esetben private e lérésû lesz) valamint void „ér tékkel" kell visszatérnie.

A partial kulcsszót ilyenkor is ki kell tenni minden elõ ford ulásnál. Csakis parciális osztá ly tarta lmazhat parciális me tódust.

13.5 *Beágyazott osztályok*

Egy osztály tarta lmazhat me tód usokat, adattagokat és más osztályoka t is. Ezeket a „belsõ" osztályoka t beágya zott (nested) osztálynak ne vezzük. Egy ilyen osztá lyt általában e lrejtünk, de ha mégis pub likus elérésûnek deklaráljuk, akkor a „külsõ" osztá lyon keresztûl ér hetjük el. A beág yazott osztályok alapértelme zés sze rint privát elérésû ek.

```
//a be gyazott oszt ly nem l tható
class Outer
{
    class Inner
    {

    }
}

//de most m r igen
class Outer
{
    public class Inner
    {

    }
}

//péld nyosít s
Outer . Inner x = new Outer . Inner ();
```

Egy beágyazott osztály hozzáfér az õt tartalma zó osztálypéldány mi nden tagjá hoz (beleértve a private elérésû tagokat és más beágyazo tt osztályokat is), de csakis akkor, ha a beágyazo tt osztály táro l egy , a külsõ osztályra hivatkozó refe renciát:

```
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 példá nyosításá nak eg y 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 = "Istv n"
        };

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

Ilyen esetekben vagy egy nyílvános 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 erõforrások felszabadításáért felelnek.
A .NET ún. automatikus szemétgyûjtõ (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 mr nem mutat semmi, felszabadítható
```

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

A szemétgyûjtõ mûködése nem determinisztikus, azaz elõre nem tudjuk megmondani, hogy mikor fut le, ugyanakkor kézzel is meghívható, de ez nem ajánlott. A következõ példában foglalunk némi memóriát, majd megvizsgáljuk, hogy mi történik felszabadítás elõtt 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 szemétgyûjtõt

        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 szemétgyûjtõt.
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 eltüntethetõek, mivel nincs többé rájuk hivatkozó referencia (hiszen a lokális objektumok hatóköre ott véget ér).

De hogyan is mûködik a szemétgyûjtõ? A .NET ún. generációs garbage collectort használ, amely abból a feltevésbõl 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 következõ történik: minden friss objektum nulladik – legfiatalabb – generációba kerül. Amikor eljön a szemétgyûjtés ideje a GC elõször ezt a generációt vizsgálja meg, ha talál hivatkozás nélküli objektumot azt törli (pontosabban az elfoglalt memóriaterületet szabadnak jelöli), a maradékot átrakja az elsõ generációba. Ezután sorban átvizsgálja a többi generációt (még kettõ van) és elvégzi a megfelelõ módosításokat. Értelemszerûen a második generációs objektumok akkor törlõdnek, ha a program megáll (illetve elfogyhat a memória is, ekkor OutOfMemoryException kivétel keletkezik). Az egyes generációk összefüggõ memóriaterületen vannak, így kissebb területen kell dolgozni a gyûjtõnek.

Az is érdekes kérdés, hogy honnan tudja a GC, hogy melyik objektumok feleslegesek. Ehhez be kell vezetnünk két fogalmat, az elsõ a gyenge- illetve erõs referenciák intézménye:
Minden olyan objektumra, amelyet a new operátorral hozunk létre erõs referenciával hivatkozunk, ezek „normális" objektumok, amelyek akkor és csakis akkor kerülnek hatókörön kívûlre (és takaríthatóak el a GC által), ha nincs rájuk hivatkozó érvényes referencia.
A gyenge referenciák ennek épp ellenkezõjét nyújtják, bármikor törölhetõ az általuk mutatott objektum ha nincs elég memória – akkor is ha létezik rá mutató érvényes 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 eltakarítható.

Egy WeakReference objektumból „visszanyerhetõ" az eredeti erõs referencia a Target tulajdonság segítségével, viszont ne felejtsük el ellenõrizni 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, amelyekrõl feltételezhetjük, hogy elérhetõek (ilyen objektumok lesznek pl. az összes lokális és globális változó). A GC mindig a root objektumokat vizsgálja meg elõször 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 idõre, vagy kevesebb erõforráshoz jut (ugyanakkor azt is számításba kell venni, hogy a GC a lehetõ legjobb idõben – értsd: akkor amikor a program a legkevesebb erõforrást 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 fenáll a hatékonyság problémája.
Épp ezért, a helyett, hogy minden alkalommal teljes vizsgálatot végezne a GC bevezették a részleges takarítás fogalmát, amely a következõ feltevésre épül: az egyes illetve kettes generációkban lévõ objektumok nagy valószínûséggel 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 életû 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 csinál, valójában az õ dolga az objektumok teljes életciklusának a kezelése és a memória megfelelõ szervezése is.
Amikor elindítunk egy .NET programot akkor a GC elsõként szabad memóriát kér az operációs rendszertõl (a .NET ún. szegmensekre osztja a memóriát, minden szegmens 16 MB méretû), mégpedig két szegmensnyit egyet a hagyományos objektumoknak (GC Heap) és egyet a nagyméretû (100 kilobyte+) objektumoknak (LOH – Large Object Heap) (ez utóbbit csakis teljes takarításnál vizsgálja a GC). Ezután nyugodtan készíthetünk objektumokat, 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 teremtõjének, nevezetesen a GC – nek. Ilyenkor a rá hárul az a rendkívûl fontos feladat is, hogy rendbe rakja a memóriát. Mit is értünk ezalatt? Hatékonyság szempontjából az a legjobb, ha az osztálypéldányok

egymásho z kö zel – lehetõ leg egymás mellett – vannak a memóriában. Épp ezé rt a GC minde n (fõ)gyûjtõciklus (tehát teljes takarítás) alka lmával átmo zgatja az objektumokat, hogy a le hetõ legoptimálisabban ér jük el õket.

Ennek a megoldásnak egy hátulütõje, hogy ilyen módon nem haszná lhatunk unmanaged kódot, mive l ez te ljes mértékbe n megakadályo zza a poi nterm ûveleteke t.
A megoldást az ob jektumok rögzítése (ún. pi nni ng) jelenti, errõl egy késõ bbi fejezet számol be.

A managelt kód épp a fenti té nyek miatt tudja felve nni a versenyt a na tív programokkal. Sõt, olya n alkalmazások esetébe n a hol sokszor foglalunk/felszabad ítunk memóriát a natív kód hátrányba is kerül(het).
Összességébe n azt mondhatjuk, hogy natív és ma nagelt program között ni ncs nagy külö nbég sebesség tekintetébe n.

A GC há romfé le módban tud mûködni:

- A GC párhuzamosan fut a z alkalma zással
- A GC felfüggesztheti az alkalma zást
- Szer ver mód

Nézzük az e lsõt: a z objektumok allokálását egy különálló szá l vég zi, ha szükség va n tisztításra, akkor a program többi szálát csak nag yo n rövid ideig függeszti fel és a takarítássa l eg yidejûleg a program to vábbra is fogla lhat helye t a memóriában, kivé ve ha a z átlépte a maximálisa n kiszabo tt keretet. Ez a limitálás a nulladik generációra vo natkozik, tehát azt szabja meg , hog y me nnyi memóriát használhat fel eg ysze rre a G0 (ame nnyiben ezt az ér téket e lérjük a GC beind ul).
Ezt a módszert o lyan alkalma zásoknál használjuk, a mikor fontos, hogy felhasználói felüle t reszponzív maradjon. Ez a z alapértelme zett mód.

Hasonlóan mûködik a második is, viszont õ teljes mértékben „ leállítja” az alkalmazást (ún. Stop - The-World) a tisztítás idejére. Ez a mód sokkal kissebb G0 kerettel rendelkezik.

A szer ve r módba n mi nden eg yes processzor külön heap –pel és GC – vel rendelkezik. Ha egy alkalmazás kifut a memóriából a z szól a GC – nek, amely felfüggeszti a program futását a tisztítás idejé re.

Ha meg akarjuk változta tni eg y program GC mdjá t szükségünk le sz eg y konfigurációs file – ra (errõl egy késöbbi feje zetben), a mely a kö vetkezõket tarta lmazza (a példában kikapcsoljuk a párhuzamos futást) :

```
<conf iguratio n>
<runtime >
<gcConcurre nt enabled="fa lse"/>
</runt ime >
</configurat io n>
```

Vagy:

```xml
<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 referenciatípushoz, ez pedig a destruktor.

A GC mielõtt megsemmísiti az objektumokat meghívja a hozzájuk tartozó destruktort, másnéven Finalizert. Ennek a metódusnak a feladata, hogy felszabadítsa az osztály által használt erõforrásokat (pl., hogy lezárja a hálózati kapcsolatokat, bezárjon minden megnyitott filet, stb...).
Vegyük a következõ kódot:

```csharp
using System;

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

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

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

A destruktor neve tilde jellel (~) kezdõdik, neve megegyezik az osztályéval és nem lehet semmilyen módosítója vagy paramétere (értelemszerûen egy destruktor mindig privát elérhetõségû lesz, vagyis közvetlenül soha nem hívhatjuk ez csakis a GC elõjoga).

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 elõször a „Konstruktor” szót fogjuk látni, majd egy gomb lenyomása után megjelenik a párja is (ha látni is akarjuk nem árt parancssorból futtatni).
A fenti kód valójában a következõ formában létezik:

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

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

}
```

Ez a forráskód nem fordul le, a Finalize metódust nem definiálhatjuk felül, erre való a destruktor.
A Finalize metódust minden referenciatípus örökli a System.Object –tõl. A Finalize elõször felszabadítja az osztály erõforrásait (a destruktorban általunk megszabott módon), azután meghívja az õsosztály 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 következõek:

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

13.7.1 IDisposable

Az IDisposable interfész segítségével egy osztály által használt erõforrások felszabadítása kézzel 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();
    }
}
```

A Dispose metódusban meg kell hívnunk a GC.SuppressFinalize metódust, hogy jelezzük a GC –nek, hogy ez az osztály már felszabadította az erõforrásait nem kell destruktort hívnia.

Az IDisposable interfészt megvalósító osztályok használhatóak ún. using blokkban, ami azt jelenti, hogy a blokk hatókörén kívûlre 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 mûvelettel (file és hálózatkezelés) kapcsolatos osztály megvalósítja az IDisposable –t, ezeket ajánlott mindig using blokkban használni.

14 Metódusok

Az objektum- orientált programo zásban egy metódus olya n programrész, amely vag y egy objektumho z, vagy egy osztályhoz köthetõ. Elõbbi az ún. osztá ly metódus, utóbbi pedig a statikus metódus. Ebben a feje zetben a z osztá ly me tódusokról lesz szó .
Egy metódussal megválto ztathatjuk eg y objektum állapotá t, vagy információt kaphatunk annak adatairól. Op timális esetben egy adattaghoz csakis metódusoko n keresztûl férhetünk hozzá (ez akkor is igaz, ha látszólag nem íg y tör ténik, pl. minde n operátor va lójában metódus fo rmájában léte zik, ezt majd látni fogjuk).

Bizonyára szeretnénk, ha a ko rábban elkészített kutya osztályunk nem csak lógna a semmiben, hanem csinálna is valamit. Készítsünk néhány metód ust a legfontosabb mûveletekhez: a z evéshe z és 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 ép ül fel egy metódus: elsõként megadjuk a láthatóságot, itt is érvényes a szabá ly, hog y e nnek hiányában az alapér telme ze tt privát elérés lesz érvénybe n. Ezutá n a visszaté rési érték típusa áll, je len esetbe n a void –dal jele ztük, hogy nem vár unk ilyesmit. Köve tkezik a metódus neve , ez konvenció szeri nt nagybetûvel kezdõdik, a sort a paraméte rlista zárja , er rõl hamarosan.

A függvé nyeket az objektum neve után ír t po nt operá torral hívhatjuk meg (ugya nez érvényes a publikus adattag okra, tula jdonságokra, 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 köthetõek, annak életciklusában játszanak szerepet. Nézzünk egy példát:

```
using System;

class NewString1
{
    private string _string;

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

    public void PrintUpper()
    {
        Console.WriteLine(this._string.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 elsõ osztály „valódi” osztály adattaggal, konstruktorral, etc... Van állapota, végezhetünk rajta mûveleteket. A második osztály nem igazi osztály, csak egy doboz, amelyben egy teljesen önálló egyedül is életképes szerkezet van, mindössze azért kell az osztálydefiníció mert egyébként nem fordulna le a program (ebben az esetben egy statikus „metódust” kellett volna készítenünk, errõl hamarosan).
Az elsõ 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 kívûlrõl megadni adatokat, vagyis paramétereket. A paraméterek számát és típusait a metódus deklarációjában, vesszõvel elválasztva adjuk meg. Egy metódusnak

gyakorlatilag bármennyi paramétere lehet. A metódus nevét és paraméterlistáját aláírásnak, szignatúrának vagy prototípusnak nevezzük. Egy osztály bármennyi azonos nevû metódust tartalmazhat, ha 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. Elõbbi 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 referenciatípusok különbözõen viselkednek az átadás szempontjából. Az értéktípusok alapértelmezetten érték szerint adódnak át, míg a referenciatípusoknál a cím szerinti átadás az elõre meghatározott viselkedés. Utóbbi esetben van azonban egy kivétel, mégpedig az, hogy míg a referenciatípus 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 észlelhetõ. Erre a magyarázat nagyon egyszerû, már említettük, hogy egy metódusparaméter lokális változóként viselkedik, vagyis ebben az esetben egyszerûen 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 referenciatípus referenciáját, akkor külön jeleznünk kell azt, hogy „valódi” referenciaként szerint akarjuk átadni.
Kétféleképpen adhatunk át paramétert referencia szerint. Az elsõ esetben az átadott objektum inicializálva kell legyen (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 „utasítással”).
Referenciatípust gyakorlatilag soha nem kell ilyen módon átadnunk (persze nincs megtiltva, de gondos tervezéssel elkerülhetõ), 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 következõ kódot:

```
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 a mit szeretnénk: x és y értéke megcserélõdött.

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 elõször 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 elsõ 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 következõ 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 ellenõriznünk kell, hogy x és y ne legyen egyenlõ)

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

A címszerinti 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 –el, 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();
		t.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. Elõfordul viszont, hogy ezt nem tudjuk egyértelmûen megmondani, ekkor ún.

paramétertömböket kell használnunk. Ha ezt tesszük, akkor az adott metódus paraméterlistájában a praméterrtömbnek kell az utolsó helyen állnia, illetve egy paraméterlistában 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", "k rte", 4, 1, "dió");
        t.PrintElements(); //ez is muk dik
    }
}
```

A paramétertömböt a params kulcsszóval vezetjük be, ezután a metódus belsejében pontosan úgy viselkedik, mint egy normális tömb. Paramétertömbként átadhatunk megfelelõ típusú tömböket is.

14.1.1 *Alapértelmezett paraméterek*

A C# 4.0 bevezeti az alapértelmezett paramétereket, amelyek lehetõvé teszik, hogy egy paraméternek alapértelmezett értéket adjunk, ezáltal nem kell kötelezõen megadnunk minden paramétert a metódus hívásakor. Nézzük a következõ 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 bizotsan minden emberhez munkahelyet rendelni, ezért két konstruktort kellett készítenünk. Ez alapvetõen nem nagy probléma, viszont az

gondot okozhat, ha valaki csak a z elsõ konstruktort haszná lja, majd megpróbál ho zzáfér ni a m unka tulajdo nság hoz. Nyílván ezt sem nagy gond megoldani, de miért fáradná nk, ha re nde lkezésünkre állnak az alapérte lmezett 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 alapér telmeze tt értéket rendeltünk, íg y bizo tsak lehetünk benne, hogy minden adattag megfele lõen inicializált. Az osztályt most így tudjuk használni:

```
Person p1 = new Person ( "Istv n" , "Reiter" );

Person p2 = new Person ( "Istv n" , "Reiter" , "b rt n r" );
```

14.1. 2 *Nevesített par améter ek*

A C# 4.0 az alapérte lmezett paramé terek me lle tt be vezeti a nevesíte tt para méterek (named pa rameter) foga lmát, amely seg ítségével explicit megadhatjuk, hogy melyik paraméter nek adunk ér téket. Né zzük az elõ zõ feje zet Person osztályá nak konstruktorát:

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

Mivel tudatjuk a fordító val, hog y po ntosan me lyik paraméte rre go ndolunk, ezér t nem kell be tarta nunk a z eredeti metód usdeklarációban elõírt sorrendet:

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

14.2 *Visszatérési érték*

Az objektumainko n nem csak m ûveleteket vég zünk, de szeretnénk lekérde zni az állapotukat is és felhasználni e zeket az ér tékeket. Ezen kívûl szere tnénk olyan függvényeket is készíteni, amelyek nem kapcsolódnak kö zvetle nül egy osztályho z, de hasznosak lehetnek (pl. a z Int.Parse () függvény ilye n).
Készítsünk eg y egyszerü függvényt, ame ly összead két számo t, az ered ményt pedig visszatérési ér tékként kap juk 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);
        }
}
```

Az eredményt a return utasítással adhatjuk vissza. A metódusdeklarációnál meg kell adnunk a visszatérési érték típusát is. Amennyiben ezt megtettük, a metódusnak mindenképpen tartalmaznia kell egy return utasítást a megfelelõ 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 kettõ között létezni kell implicit típuskonverziónak:

```
public int Add(int x, int y)
{
        return (byte)(x + y); //ez muk dik b r nincs sok értelme
}

public int Add(int x, int y)
{
        return (long)(x + y); // ez le sem fordul
}
```

Visszatérési értékkel rendelkezõ metódust használhatunk minden olyan helyen, ahol a program valamilyen típust vár (értékadás, logikai kifejezések, metódus paraméterei, ciklusfeltétel, etc..).

Valójában a Main metódus két formában létezik: visszatérési értékkel és anélkül. 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 külsõ

program/scipt hívta meg a programunkat és kíváncsiak vag yunk, hogy sikeres volt – e a futás a.

```
static public int Main ()
{
    return 0 ;
}
```

14.3 *Kiterjesztett metódusok*

A C# 3.0 le hetõséget ad arra, hogy egy már létezõ típusho z új metódusokat adjunk, anélkül, hogy a zt közvetle nül módosítaná nk vag y szá rma ztatnánk belõle . Egy kiterjesztett metód us (exte nsion method) minden esetben eg y statikus osztály statikus metódusa ke ll legye n (errõ l a kö vetkezõ fejezetbe n) .
Egészítsük ki a string típ ust egy metód ussal, ami kiírja a képernyõre az adott karaktersorozato t:

```
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 ); //igy is haszn lhatjuk
    }
}
```

A this módosító utá n a paramé ter típ usa kö vetke zik, amely meghatározza a kiterjesztett osztály típ usát. A fenti példában látható, hogy hag yomá nyos statikus metóduské nt is használha tó eg y extensio n me thod.

Ha két kiterjesztett metódus ug ya nazzal a szigna túrá val rendelkezik, akkor a hagyomá nyos statikus úton kell hívnunk õket. Ha ne m így teszünk akkor a speciálisabb (szûkebb típusú) para méterû metód us fog meghívódni.

Kiterjesztett me tódust nem def iniálhatunk beág ya zott osztá lyban.

15 Tulajdonságok

A tulajdonságokat (property) a mezõk közvetlen módosítására használjuk, a nélkül, hogy megsértenénk az egységbe zárás elvét. A tulajdonságok kívûlrõl nézve pontosan ugyanolyanok mint a hagyományos változók, de valójában ezek speciális metódusok.
Minden tulajdonság rendelkezhet getter és setter blokkal, elõbbi a property mögött lévõ mezõ 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("Istv n");
        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 nevû változót használtunk. Ez egy speciális objektum, az értéke mindig megegyezik azzal a mit a tulajdonság értékéûl adtunk:

```
Person p = new Person("Istv n");
p.Name = "Béla"; //value = "Béla"
```

A getter és setter elérhetõsége nem kell egyezzen, de a getternek minden esetben publikusnak kell lennie:

```
public string Name
{
    private get { return this._name; } //ez nem muk dik
    set { this._name = value; }
}

public string Name
{
    get { return this._name; }
    private set { this._name = value; } //ez viszont jó
}
```

Arra is van le hetõség, hogy csak az egyiket használjuk, ekkor csak írható /olvasható tulajdonságokró l beszélünk:

```
public string Name
{
    get { return this . _name ; }
}
```

Egyik esetben sem vag yunk rákénysze rítve , ho g y a zonnal visszadjuk/beolvassuk az adttag értékét, tetszés szeri nt végezhetünk mûvele teket is rajtuk:

```
public string Name
{
    get { return "Mr. " + this . _name ; }
}
```

A C# 3.0 rendelkezik eg y nagyon érdekes újítással a z ún. automa tikus tulajdonságokkal. Nem kell létre hoznunk sem az adattagot, sem a teljes tulajdonságot, a ford ító mindkettõt lege nerálja nekünk:

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

A ford ító a utoma tikusan létre hoz egy pri vate elérésû, stri ng típusú na me ne vû adattagot és elkészíti hozzá a ge ttert/se tter t is. Van azonban egy prob léma, mégho zzá az, hogy a ford ítás pillanatába n ez a válto zó még nem léte zik, vag yis közvetlenül nem hivatkozhatunk rá pl. a konstr uktorban . Ile nkor a settere n keresztûl kell é rtéket adnunk.

```
class Person
{
    public Person ( string name )
    {
        this . Name = name ;
    }

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

16 Indexelõk

Az indexelõk hasonlóak a tulajdonságokhoz, azzal a különbséggel, hogy nem névvel, ha nem egy indexxel férünk hozzá az adott információhoz. Általában olyan esetekben használják, amikor az osztály/struktúra tartalmaz egy tömböt vagy valamilyen gyûjteményt (vagy olyan objektumot, amely maga is megvalósít egy indexelõt).
Egy indexelõt így implementálhatunk:

```
using System;
using System.Collections;

class Names
{
    private ArrayList nameList;

    public Names()
    {
        nameList = new ArrayList();
        nameList.Add("Istvan");
        nameList.Add("Judit");
        nameList.Add("Bela");
        nameList.Add("Eszter");
    }

    public int Count
    {
        get
        {
            return nameList.Count;
        }
    }

    public string this[int idx]
    {
        get
        {
            if (idx >= 0 && idx < nameList.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 „ névtele n tula jdonság ”, a this m utat a z aktuális objektum ra, amin az inde xe lõt defi niáltuk. Több indexet is megad hatunk, amelyek különbözõ típusúak is lehetnek.

17 Statikus tagok

A hagyományos adattagok és metódusok objektumszinten léteznek, azaz minden objektum minden adattagjából saját példánnyal rendelkezik. Gyakran van azonban szükségünk objektumtól független mezõkre/metódusokra, pl. ha meg szeretnénk számolni, hogy hány objektumot hoztunk létre. Erre a célra szolgálnak az ún. statikus tagok, amelyekbõl osztályszinten összesen egy darab létezik. Statikus tagokat akkor is használhatunk, ha az osztályból nem készült példány.
A statikus tagok jelentõsége a C# tisztán objektum orientáltságában rejlik, ugyanis nem definiálhatunk globális mindenki számára egyformán elérhetõ tagokat. Ezt (is) váltják ki a statikus adattagok és metódusok.

17.1 Statikus adattagok

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 elõször használjuk az adott osztály egy példányát.

Konve nció szerint minde n statikus tag (adattagok is) ne ve nagybetûvel kezdõdik. A statikus és láthatósági módosító megadásának sorrendje mindeg y.

17.2 Statikus konstruktor

A statikus konstruktor a statikus tagok beállításáér t fele l. A statikus konstruktor közvetlenül aze lõtt fut le, hogy eg y példány keletkezik az adott osztályból .
A statikus ko nstruktornak nem lehe t láthatóságot adni, illetve ni ncsenek paraméte rei sem. Nem férhet ho zzá a pé ldánytagokhoz 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 eli ndítjuk a programot a követke zõ kimenetet kapjuk:

Start...
Var = 10
Statikus konstrukto r
Konstruktor

17.3 *Statikus metódusok*

Statikus metód ust a hag yomá nyos metódusokhoz hasonlóan készítünk, mi ndössze a static kulcsszóra van szükség ünk. Ilye n me tódus volt a z elõzõ példában a z Init

metódus is. A staikus konstruktortól eltérõen rá nem vonatkozik, hogy nem lehetnek paraméterei. Statikus metódusok nem férnek hozzá az osztály „normális” tagjaihoz, legalábbis direkt módon nem (az minden további nélkül mûködik, 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 mûvelet elvégzése. Ilyen metódus például az Int osztályhoz tartozó Parse statikus metódus is.

A „leghíresebb” statikus metódus a Main.

17.4 *Statikus tulajdonságok*

A statikus tulajdonságok a C# egy viszonylag ritkán használt lehetõsége. Általában osztályokhoz kapcsolódó konstans értékek lekérdezésére használjuk (lényegében a statikus metódusok egy olvashatóbb verziója)

```
class Math
{
    static public double PI
    {
        get { return 3.14; }
    }
}
```

17.5 *Statikus osztályok*

Egy osztályt statikusnak jelölhetünk, ha csak és kizárólag statikus tagjai vannak. Egy statikus osztályból nem hozható létre példány, nem lehet példánykonstruktora (de statikus igen) és mindig lezárt (ld. Öröklõdés). A fordító minden esetben ellenõrzi ezeknek a feltételeknek a teljesülését.

```
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 eltérõen nem referencia- hanem értéktípusok.
Minden struktúra indirekt módon a System.ValueType osztályból származik, amely pedig a System.Object –bõl. A System.ValueType egy speciális típus, amely lehetõséget biztosít értéktípusok számára, hogy referenciatípusként viselkedjenek. A struktúrák alapvetõen értéktípusok, de viselkedhetnek referenciatípusként mivel *konvertálhatóak* System.ValueType típusra (lásd: boxing).
A struktúrák közvetlenül tartalmazzák a saját értékeiket míg osztályok „csak" referenciákat tárolnak. Épp ezért struktúrát általában akkor használunk, ha egyszerû adatokkal kell dolgoznunk, de nincs szükségünk egy osztály minden szolgáltatására.

18.1 Konstruktor

Minden struktúra alapértelmezetten rendelkezik paraméternélküli konstruktorral, amelyet nem rejthetünk el. Ez a konstruktor a struktúra minden mezõjét a megfelelõ nullértékkel tölti fel:

```
using System;

struct Test
{
    public int x;
}

class Program
{
    static public void Main()
    {
        Test t = new Test();
        Console.WriteLine(t.x); //x == 0
    }
}
```

Nem kötelezõ használni a new operátort, de ha így teszünk akkor a struktúra tagjainak használata elõtt definiálni kell az értéküket, ellenkezõ 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 mezõ értékadásáról gondoskodnunk kell. Egy struktúra mezõit nem inicializálhatjuk:

```
struct Test
{
    int _x = 10; // ez hiba
    int _y;

    public Test(int x, int y)
    {
        _y = y; // hiba x nem kap értéket
    }
}
```

Struktúrának csakis paraméteres konstruktort definiálhatunk magunk, 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 mukdik
    }
}
```

18.2 Destruktor

Struktúrák nem rendelkezhetnek destruktorral. Egy struktúra két helyen lehet a memóriában: a stackben és a heapben (ha egy referenciatípus tagja).
Ahhoz, hogy megértsük, hogy miért nincs destruktor szükségünk van a következõre: egy struktúrában lévõ referenciatípusnak csak a referenciáját tároljuk. Ha a veremben van a struktúra, akkor elõbb vagy utóbb kikerül onnan és mivel így a benne lévõ 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 megfelelõ nulla értékre inicializálódik.

Struktúra nem tartalmazhat saját magával megegyezõ típusú adattagot. Ugyanígy, egy struktúra nem tartalmazhat olyan típusú tagot a mely típus hivatkozik az eredeti struktúrára:

```
struct Test
{
    Test t;
}

struct Test1
{
    Test2 t;
}

struct Test2
{
    Test1 t;
}
```

Mindhárom struktúra hibás. Az ok nagyon egyszerû: mivel a struktúrák direkt módon – nem referenciákon keresztûl – 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 leford ítjuk ezt a kódot azt fogjuk lá tni, hogy t1 .x értéke nem változott, tehát nem refere nciát adtunk át t2 – nek.

Most né zzünk meg eg y nag yon g yakori hibát amibe belefuthatunk. Adott a következõ 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);
    }
}
```

Teljese n szabályos forrás, le is fordul. Látható, hogy a Point str uktúra pub likus tulajdonságokkal bír, vag yis jogosnak tûnik, hog y a Line str uktúrá n keresztûl módosíta ni tud juk a koordi nátákat. 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 gettert 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 elõ grafikus felületû alkalmazások készítése közben, mivel a .NET beépített Point típusa szintén struktúra.

18.5 *Öröklõdés*

Struktúrák számára az öröklõdés tiltott, minden struktúra automatikusan sealed módosítót kap. Ilyen módon egy struktúra nem lehet absztrakt, tagjainak elérhetõsége nem lehet protected/protected internal, metódusai nem lehetnek virtuálisak illetve csak a System.ValueType (és így a System.Object) 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? Olyan függvény, amely önmagát hívja. Rengeteg olyan probléma van, amelyeket több lényegében azonos feladatot végrehajtó részre lehet osztani. Vegyük pl. a hatványozást: semmi mást nem csinálunk, mint meghatározott számú szorzást végzünk, méghozzá ugyanazzal a számmal. Írhatunk persze egy egyszerû ciklust is, de ez egy kicsit atombombával egérre típusú megoldás lenne. Nézzük meg a hatványozás rekurzív megfelelõjét:

```
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 kitevõt (y) a függvény minden hívásakor eggyel csökkentjük, egészen addig, amíg értéke nulla nem lesz, ekkor befejezzük az „ördögi” kõrt 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 gyorsre ndezést!

Megoldás (19/QuickSort.cs)

A gyorsrende zés a leggyorsabb rendezés nag y e lemszám eseté n O(n* log n) nagyságrendû átlaggal. Az algoritmus lényege, hogy a rende zendõ elemek kö zül kiválaszt egy ún. pivo t elemet, amely elé a ná la nag yobb , mögé pedig a nála kissebb elemeket teszi, majd az így kapott két csoportra ismét meg hívja a gyorsrendezést (tehát egy rekur zív algoritmusról beszélünk). Lássuk, hogy hogya n is néz ez ki a gyakor latba n!

```
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 );
    }
}
```

Készíte ttünk eg y osztályt, amely reprezentálja a rendeze ndõ tömböt. A Sort metódussa l fogjuk meghívni a té nyleges rendezõ me tódust ame ly a következõképpen 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 kissebb/nagyobb elemeket majd továbbhívjuk a rendezést. Nézzünk egy példát! A rendezendõ számsorozat legyen:

3, 9, 4, 6, 8 , 11

Left és right 0 és 5, ezeket az értékeket eltároljuk mivel az értékük módosul, de a metódus végén is szükség van az eredetiekre. Az elsõ ciklus – mivel nem fog a left indexen lévõ számnál (3) kissebbet találni – úgy végzõdik, hogy right értéke 0 lesz. Az elágazásba – mivel right és left egyenlõ – nem megyünk bele ugyanúgy ahogyan a második ciklusba sem, mivel a hármasnál kissebb elem nincs és a nála nagyobbak pedig utána helyezkednek el. A következõ 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 elsõ elemet – mivel õ 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 elsõ ciklus egyszer fog lefutni hiszen a tömb negyedik indexén ülõ nyolcas szám már kissebb mint a pivot elem. Left nem egyenlõ right –tal ezért a következõ elágazásba is bemegyünk és a left indexre helyezzük a right indexen lévõ nyolcast (a pivot elem pedig pont az itt „már nem” lévõ kilencest tárolja). Left elõrelép 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 egyenlõ: 4. Éppen ezért a második elágazást kihagyjuk és tovább lépünk. A left által mutatott indexre be is helyezzük a pivotban lévõ 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 ezutáni 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 következõ elemre hivatkozó referenciát tartalmaznak. A láncolt listát az elsõ fej- vagy gyökérelemen keresztûl érjük el. Ha az elemek csak a következõ tagra mutatnak akkor egyszeresen-, ha a megelõzõ elemre is akkor kétszeresen láncolt listáról beszélünk:

Elsõként valósítsuk meg az elemeket jelképezõ osztályt:

```
class Node
{
    public Node ( int value )
    {
        this . Value = value ;
    }

    public int Value { get ; set ; }

    public Node Next { get ; set ; }
    public Node Previous { 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 Node Root { get ; private set ; }
}
```

Az Add metódus az a melyik érdekes. Elsõként megvizsgáljuk, hogy léte zik -e gyökére lem, ha nem akkor lé tre hozzuk és ni ncs más dolgunk (ugye i lyenkor még nincs se elõ zõ se rákövetkezõ elem). Más a helyzet, ha van már néhá ny elem a listában, ekkor meg kell keresnünk a leg utolsó ele met és utánafûzni a z újat (megva lósíthattuk volna úg y is a listát, hogy tárolunk egy referenciát az uto lsó elemre is ez lé nyegesen g yorsabb lenne – de kev ésbé é rdekes).

Ahho z, hog y megkeressük az utolsó eleme t szükségünk lesz egy átmene ti refere nciára amely mindig a z aktuális e lemet mutatja ma jd. A ciklust addig kell futtatni , ameddig az aktuális elem ráköve tkezõje null értékre nem muta t, ekkor beállítjuk a Next és Pre vious értékeket is.

19.4 *Bináris keresõfa*

Készítsünk bináris keresõfát!

Megoldás (19/Bi naryTree.cs)

A fa típus olyan adatszerkeze t amelynek e lemei nulla vag y több gyer mekelemmel és maximum egy szülõelemme l re ndelke znek:

A fa típus egy speciális esete a bináris fa, ame ly mi nden elemének pontosan eg y szülõ és ma xim um kettõ g yermekeleme lehet. A bináris fa speciális esete pedig abináris keresõfa, amelynek jellem zõje, hog y eg y szülõ elem ba l olda li részfájában a szülõe lemnél kissebb jobboldali részfájába n pedig a szülõelemnél nagyobb elemek

vannak, ezáltal egyértelmûen meghatározható, hogy egy elem benne van-e a fában vagy nincs (értelemszerûen a részfák is keresõfák, vagyis rájuk is ugyanez vonatkoziki). A bináris keresõfa minden elemének egyedi kulccsal kell rendelkeznie, vagyis ugyanaz az elem kétszer nem szerepelhet a fában. A keresés mûvelet O(logn) nagyságrendû.

Ahogyan az elõzõ feladatban is most is készítsük el elsõként a fa csúcsainak osztályát:

```
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 jöjjön a fa osztály:

```
class BinaryTree
{
    public BinaryTree()
    {

    }

    public BinaryTree(int[] values)
    {
        foreach (int value in values)
        {
            this.Insert(value);
        }
    }

    public void Insert(int value)
    {
        if (Root == null)
        {
            Root = new TreeNode(value);
        }
        else
        {
            TreeNode current = Root;

            while (true)
            {
                if (current.Value > value)
                {
                    if (current.Left != null)
                    {
                        current = current.Left;
                    }
                    else
                    {
                        current.Left = new TreeNode(value);
                        current.Left.Parent = current;
```

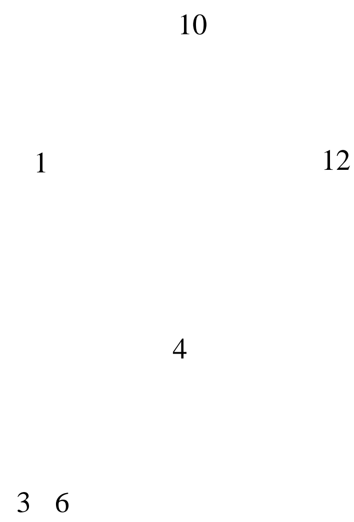
```
                    }
                }
                else if (current.Value < value)
                {
                    if (current.Right != null)
                    {
                        current = current.Right;
                    }
                    else
                    {
                        current.Right = new TreeNode(value);
                        current.Right.Parent = current;
                    }
                }
                else
                    break;
            }
        }
    }

    public TreeNode Root { get; private set; }
}
```

Az Insert metódus ami érdekel minket. Ha a gyökérelem null értéken áll akkor elkészítjük egyébként megkeressük az új elem helyét. Tegyük fel, hogy az elemek a következõ sorrendben érkeznek:

10, 1, 4, 6 , 6, 3, 9, 12

Ekkor a bináris keresõfa így fog kinézni:



Látható, hogy a hatos csak egyszer szerepel a duplikált elemeket a beszúrásnál kiszûrjük.

A következõ fe ladatunk, hogy ki írjuk a fa ele meit a ko nzolra. Persze ez nem is olya n egyszerû, hiszen megfelelõ algoritm usra lesz szükségünk, hog y a fa elemeit bejárhassuk. Egy rekurzív algo ritm ust fogunk használni, amely stratégiátó l függõen az egyes csúcsok részfáit ma jd magát a csúcsot látoga tja meg. Három féle stratégiát ismerünk: preorder, inorder és postorder. A preorder elsõké nt a csúcsot majd a bal és jobb oldali részfát veszi kezelésbe. Inorde r módon a bal o ldali részfa a csúcs és a jobb oldali részfa lesz a sorrend vég ül pedig a postorder bejárás sorrend je a va l oldal irészfa a jobb olda li részfa vég ül pedig a csúcs. Né zzük meg, hogyan is mûkdöik mindez a g yakor latba n! A fent felépített fán fogunk i norder módo n végigmenni.

Az algo ritmust a gyökérele mre (10) fogjuk meghívni, amely elsõké nt a bal oldali részfa csúcsá t (1) fogja meglétogatni. Mivel neki nincsen bal oldali részfá ja ezért kiír juk az egyes számot és lép ünk a jobb oldali részfára (4). Neki már van bal oldali ága ezért õt vizsgáljuk a továbbiakban. Itt ni ncs gyer mekelem , ezér t a következõ szám amit ki ír hatunk a három. Visszalépünk a szülõelemre és ki írjuk õt (4) majd lépünk jobbra. A hatos csúcsnak sincs bal oldali fája ezért ki írjuk majd jö n a jobb fa a kilences amit megi ntcsak kiír unk hiszen nem rende lkezik gyermeke lemme l.
Ezen a po nton végeztünk a gyökérelem bal oldali fájáva l ezért õt fogjuk ki ír ni, e zután pedig már csak egyetle n elem marad. A végsõ sorrend tehát:

1, 3, 4, 6, 9 , 10, 12

A forráskódban e z az algoritm us meg lehetõsen eg yszerûe n jelenik meg, kö vetke zze n az i norder 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<> osztá lyról a Lambda kifeje zések c. fejezetben o lvashat többet a z olvasó.

Öröklõdéssel egy már létezõ típust terjeszthetünk ki vagy bõvíthetjük tetszõleges szolgáltatással. A C# csakis egyszeres öröklõdést engedélyez, ugyanakkor megengedi több interfész impementálását (interfészekrõl hamarosan).

Készítsük el az elméleti rész példáját (Állat-Kutya-Krokodil) C# nyelven. Az egyszerûség kedvéért hagyjuk ki az Állat és Kutya közti speciálisabb osztályokat:

```
class Animal
{

}

class Dog : Animal
{

}

class Crocodile : Animal
{

}
```

Az õsosztályt az osztálydeklaráció után írt kettõspont mögé kell tenni, szintén itt lesznek majd az osztály által megvalósított interfészek is.

A Kutya és Krokodil osztályok egyaránt megvalósítják az õsosztály (egyelõre szegényes) funkcionalitását. Bõvítsük 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 õsosztálynak, vagyis átt kell gondolnunk a példányosítást. Az elsõ változatot (az alapértelmezett konstruktorral) így használhattuk:

```
static public void Main()
{
    Dog d = new Dog();
    Crocodile c = new Crocodile();
}
```

Ha ezt az új Animal osztállyal probálnánk meg akkor meglepetés fog érni, mivel nem fordul le a program. Ahhoz, hogy ki tudjuk javítani a hibát tudnunk kell, hogy a

leszármazott osztályok elõször mindig a közvetlen õsosztály konstruktorát hívják meg, vagyis – ha nem adunk meg mást – az alapértelmezett konstruktort. A probléma az, hogy az õsosztálynak már nincs ilyenje, ezért a leszármazott osztályokban explicit módon hívni kell a megfelelõ 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("Bund s");
    Crocodile c = new Crocodile("Alad r");

    Console.WriteLine("{0} és {1}", d.Name, c.Name);
}
```

Ugyanígy használhatjuk az õsosztály metódusát is:

```
static public void Main()
{
    Dog d = new Dog("Bund s");
    Crocodile c = new Crocodile("Alad r");

    d.Eat();
    c.Eat();
}
```

Honnan tudja vajon a fordító, hogy egy õsosztálybeli metódust kell meghívnia? A referenciatípusok speciális módon jelennek meg a memóriában, rendelkeznek többek közt egy ún. metódustáblával, ami mutatja, hogy az egyes

metódushívásokná l me lyik metódust kell meg hívni. Persze e zt is meg kell határozni vala hogy, ez nagy vo nalakba n úgy tör ténik, hogy a fordító a ford ítás pillanatában megkapja a metódus nevé t és elind ul „ visszafelé” az osztályhierarchia mentén. A fenti példában a hívó osztály nem rendelkezik Eat() ne vû metódussal és nem is definiálja át annak a viselkedését (errõl hamarosan) , ezért a z eggyel feljebbi õst kell megné znünk. Ez egészen a lehetõ „legújabb” metód usdefi nícióig megy és amikor megtalá lja a megfelelõ impleme ntációt bejeg yzi a zt a me tódustáblába.

20.1 *Virtuális metódusok*

Az õsosztályban deklarált virtuális (vag y polimorfikus) metódusok viselkedését a leszá rma zottak átdefiniálhatják. Virtuális metód ust a szig natúra elé ír tvir tual kulcsszó segítségéve l deklarálha tunk:

```
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 leszárma zott osztályba n a z o verride kulcsszóval mond juk meg a fordító nak, hogy szá ndékosan hoztunk létre az õsosztályé val a zonos ne vû metódust és a leszá rma zott osztályon ezt kívánjuk haszná lni mosta ntól. Egy override –a l jelölt

metódus automatikusan virtuális is lesz, így az õ leszármazottai is átdefiniálhatják a mûködését:

```
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 õsosztály 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 belõle virtuális metódust, amelyet az utódjai 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 –al jelölt tagokat (az utolsó kettõ egyébként virtuális is lesz, de ezt nem kell külön jelölni).

20.2 Polimorfizmus

Korábban már beszéltünk arról, hogy az õs és leszármazottak közt az-egy (is-a) reláció áll fent. Ez a gyakorlatban azt jelenti, hogy minden olyan helyen, ahol egy õstípust használunk ott használhatunk leszármazottat is (pl. egy állatkertben állatok vannak, de az állatok helyére (nyílván) behelyettesíthetek egy speciális fajt).
Például gond nélkül írhatom a következõt:

```
Animal d = new Dog("Bunds");
```

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 mûködik, a fordító hibát jelezne.

Abban az esetben ugyanis, ha a fordító engedné a visszafelé konverziót az ún. leszeletelõdés (slicing) effektus lépne fel, azaz az adott objektum elveszítené a speciálisabb osztályra jellemzõ karakterisztikáját. A C++ nyelvben sokszor jelent gondot ez a probléma, mivel ott egy pointeren keresztülmegtehetõ a „lebutítás". Szerencsére a C# nyelvben ezt megoldották, így nem kell aggódnunk miatta.

Mi történik vajon a következõ 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 elemekbõl 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 futásidejû típust, és ezáltal ütemezi a metódushívásokat. Ez az ún. késõi 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 elsõ ilyen

implementáció egy virtuális metódus lesz, a za z a keresés legkésõbb az elsõ virtuá lis válto zatná l megá ll.

20.3 *Lezárt osztályok és metódusok*

Egy osztá lyt lezár hatunk, aza z megtilthatjuk, hogy új osztályt szár maztassunk be lõle:

```
sealed class Dobermann : Dog
{

}

class MyDobermann : Dobermann // ez nem jó
{

}
```

Egy metódust is deklarálhatunk lezá rtké nt, ekkor a leszá rma zottak már nem definiálhatják át a m ûködését:

```
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 le het példányosítani. A létre hozásának 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 következõek:

- absztrakt osztályt nem lehet példányosítani
- abszrakt 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 függvények. 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 mûködik a következõ példában a polimorfizmus elve? :

```
Animal[] animalArray = new Animal[2];

animalArray[0] = new Dog("Bund s");
animalArray[1] = new Crocodile("Alad r");
```

Ennek a kódnak hiba nélkül kell fordulnia, hiszen ténylegesen egyszer sem példányosítottuk az absztrakt õsosztályt. A fordító csak azt fogja megvizsgálni, hogy mi van a new operátor jobb oldalán, az alaposztály nem érdekli. Természetesen a következõ esetben nem fordulna le:

```
animalArray[0] = new Animal("Animal");
```

21 Interfészek

Az interfészek hasonlóak az absztrakt osztályokhoz, abban az értelemben, hogy meghatározzák egy soztály viselkedését, felületét. A nagy különbség a kettõ közt az, hogy míg elõbbi eleve meghatároz egy osztályhierarchiát, egy interfész nem köthetõ közvetlenül egy osztályhoz, mindössze elõír egy mintát, amit meg kell valósítania az osztálynak. Egy másik elõnye az interfészek használatának, hogy míg egy osztálynak csak egy õse lehet, addig bármennyi intefészt megvalósíthat. Ezen felül interfészt használhatunk struktúrák esetében is.A következõ példában átírjuk az Animal õsosztályt 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 betûvel kezdjük. Látható, hogy a metódusokhoz nem tartozik definíció, csak deklaráció. A megvalósító osztály dolga lesz majd implementálni a tagjait. Egy interfész a következõket tartalmazhatja: metódusok, tulajdonságok, indexelõk és események (errõl hamarosan). A tagoknak nincs külön megadott láthatóságuk, mindannyiuk elérhetõsége publikus. Az interfész elérhetõsége alapesetben publikus, illetve jelölhetjük internal –ként, másféle láthatóságot nem adhatunk meg (illetve osztályon belül deklarált (beágyazott) interfész elérhetõsége 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 õsosztály nevét kell elõre venni, utána jönnek az interfészek:

```
class Base { }

interface IFace { }

class Derived : IFace, Base { } // ez nem fordul le
```

Egy interfészt származtathatunk más interfészekbõl:

```
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 is 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 mégis működik a program. 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 ér vé nyes ezekre a tagokra.
Újabb problémánk van, még hozzá az, hogy hogyan fogjuk meg hívni a metód usokat?
Most fogjuk kihaszná lni, hogy egy osztály konvertálható a megvalósított interfészek típ usára:

```
static public void Main ()
{
    Test t = new Test ();

    (( IOne ) t ). Method (); // ez muk dik

    ITwo it = t ;
    it . Method (); // ez is muk dik
}
```

21.2 *Virtuális tagok*

Egy interfész tag jai alapértelmezés szerint lezártak, de a meg valósításnál jelölhe tjük õket virtuálisnak. Ezutá n az osztály leszármazottjai tetszés szerint módosíthatják a definíciót, a már ismert override kulcsszó val:

```
class Dog : IDog
{
    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 újraimplementá lha tja a z adott i nter fészt, ame nnyiben nemcsak az õsnél, de a z utód nál is je löljük 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 használnuk kell a new kulcsszót annak jelölésére, hogy eltakarjuk az õs megvalósítását.

22 Operátor kiterjesztés

Nyílván szere tnénk, hogy a z álta lunk készített típusok haso nló funkcionalitással rendelkezze nek mint a beép ített típ uso k (int, stri ng, stb …).
Vegyük p l. a zt a példát, a mikor egy má tri x típust valósítunk meg. Jó lenne , ha az összeadás, ki vonás, szor zás, stb. mûve leteket úg y tudnánk végreha jtani, mi nt egy egész szám esetében, nem pedig metódushívásokkal. Szerencsére a C# ezt is lehe tõvé teszi szám unkra, ugyanis e ngedi az operátorok ki terjesztését (operator over loading) , vagyis egy adott operátort tetszés szerinti funkció val ruházhatunk fel a z osztá lyunkra vonatko ztatva .

```
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 kiterjesz hetõ operá torok listája:

+(unáris) - (unáris) ! ~ ++
-- + - * /
% & | ^ <<
>> == != > <
>= <=

A C# nye lvben a z operátorok valójában statikus metód usok, paraméte reik az operandusok, visszatérési értékük pedig az e redmény. Egy egyszerû 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 mûködését fogalmaztuk át. A paramé terek (opera ndusok) ne vei konvenció szeri nt lhs (left- hand-side) és rhs (right-ha nd-side), uta lva a jobb és baloldali opera ndusra.
Tehát most már nyugodta n írhatom a kö vetkezõt:

```
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 mûveletet:

```
MyInt result = MyInt.operator+(x, y);
```

22.1 *Egyenlõség operátorok*

A C# nyelv megfogalmaz néhány szabályt az operátorkiterjesztéssel kapcsolatban. Ezek egyike az, hogy ha túlterheljük az egyenlõség operátort (==) akkor definiálnunk kell a nem-egyenlõ (!=) 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-egyenlõ operátor esetében a saját egyenlõség operátort használtuk fel (a megvalósítás elve nem feltétlenül világos, elsõként megvizsgáljuk, hogy a két elem egyenlõ – e, de mi a nem-egyenlõségre 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 megõrizni, errõl késõbb még lesz szó. A fenti esetben ezt a metódust is illik megvalósítani. Az Equals azonban egy kicsit különbözik, õ egyetlen object típusú paramétert vár, ezért meg kell majd gyõzõdnünk 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 –tõl örökölt GetHashCode metódussal is, így az osztály használható lesz gyûjteményekkel és a HashTable típussal is. A legegyszerûbb implementáció visszad 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(pár) elég nagy fejfájást tud okozni, elsõként nézzük meg a következõ 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 egyszerûen nem készíthetünk (ezt egyébként maga a Microsoft sem ajánlja) (illetve készíthetünk, de nem fog úgy mûködni a hogyan 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 különbözõ gyûjteményekkel való együttmûködés érdekében. Ezekkel hamarosan többet is foglalkozunk.

22.4 *Konverziós operátorok*

A C# a szûkebbrõl tágabbra konverziókat implicit módon (azaz külön jelölés nélkül), míg a tágabbról szûkebbre 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, hog y a konverziós operá torok mindig sta tikusak.

22.5 *Kompatibilitás más nyelvekkel*

Mivel nem minde n nyelv teszi le hetõvé a z ope rátorok kiterjesztését ezér t a CLS javaso lja , hog y 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 ne m kö telezõ, de bizo nyos he lyzetekbe n jól jön.

23 Kivételkezelés

Nyílván vannak olyan esetek, amikor az alkalmazásunk bár gond nélkül lefordul, mmégsem úgy fog mûködni, ahogy elképzeltük. Az ilyen „abnormális” mûködés 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;
    }
}
```

tt 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 mûvelethez 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 megfelelõ kivételt (arra figyeljünk, hogy ezek is blokkok, azaz a blokkon belül deklarált változók a blokkon kívûl nem láthatóak). A fenti programra a következõ lesz a kimenet:

```
A hiba: Index was outside the bounds of the array.
```

Látható, hogy a kivétel egy objektum formájában létezik. Minden kivétel õse 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 kettõt kell megemlítenünk:

- Message: ez egy olvashatóbb formája a kivétel okának.
- InnerException: ez alapértelmezetten null értékkel rendelkezik, akkor kap értéket, ha több kivétel is történik. Értelemszerûen ekkor a legújabb kivételt kaphatjuk el és az InnerException –ön keresztül követhetjük vissza az eredeti kivételig.

Nézzük meg, hogyan mûködnek a kivételek. Kivétel két módon keletkezhet: vagy a throw utasítással szándékosan mi magunk idézzük elõ, vagy az alkalmazás hibás mûködése miatt.
Abban a pillanatban amikor a kivétel megszületik a rendszer azonnal elkezdi keresni a legközelebbi megfelelõ catch blokkot, elsõként 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 lehetõség, hogy nem talál megfelelõ catch blokkot, ekkor a program futása leáll.
Ha mégis talált használható catch blokkot, akkor a kivétel helyérõl a vezérlés a talált catch –re kerül. Ha több egymásba ágyazott kivételrõl van szó, akkor a megelõzõ 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. Hurr!");
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

A C++ -tól eltérõen itt példányosítanunk kell a kivételt.

A catch –nek nem kötelezõ megadni a kivétel típusát, ekkor minden kivételt elkap:

```
try
{
    throw new System.Exception();
}
catch
{
    Console.WriteLine("Kivétel. Hurr!");
}
```

Ilyenkor viszont nem használhatjuk a kivételobjektumot.

23.1 *Kivétel hierarchia*

Amikor kivétel dobódik, akkor a vezérlést az elsõ alkalmas catch blokk veszi át. Az összes kivétel ugyanis 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 õs 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 sem használhatjuk a kivételobjektumot.

23.2 K*ivétel 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. Hurr !");
        }
        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 kivételkezelõnek akarjuk átadni a kivételt:

```
try
{
}
catch (System.ArgumentException e)
{
    throw; //tov bbadjuk
    throw (new System.ArgumentNullException()); //vagy egy jat dobunk
}
```

Természetesen a fenti példában csak az egyik throw szerepelhetne „legálisan”, ebben a formában nem fog leforduln.
Ilyenkor beállíthatjuk az Exception InnerException tulajdonságát is:

```
try
{
    throw new Exception();
}
catch (System.ArgumentException e)
{
    throw (new System.ArgumentNullException("Tov bb", 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 el tt: {0}", x);
			throw new Exception();
		}
		catch(Exception)
		{
			Console.WriteLine("Kivétel. Hurr!");
		}
		finally
		{
			Console.WriteLine("Finally blokk");
			x = 11;
		}

		Console.WriteLine("x értéke a kivétel ut n {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. Mindkettõ a System.Collections névtérben található.

Elsõként nézzük az IEnumerable interfészt:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Ez a foreach –nek fogja szolgáltatni a megfelelõ 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 következõ elemre mozgatja a mutatót, ha tudja, ellenkezõ 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 lévõ elemet adja vissza. Ennek object típussal kell visszatérnie, hiszen minden típusra mûködnie kell (létezik generikus változata is, de errõl késõbb).

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("Bund s"));
        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 kezdõértékéül -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 kívûl akarjuk használni az osztályt pl. ha készítettünk indexelõt is, akkor gondoskodnunk kell a megfelelõ konverzióról is (a foreach kivételt képez, mivel õ 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 rendezõ metódusa, amely ezzel a metódusal dolgozik. Az IComparable egyetlen metódussal a CompareTo –val rendelkezik, amely egy object típust kap paraméteréûl:

```
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 megfelel objektum..."));
    }
}
```

Az osztályban a beépített típusok CompareTo metódusát használtuk, hiszen õk mind megvalósítják ezt az interfészt. Ez a metódus -1 –et ad vissza ha a hívó fél kisebb, 0 –át, ha egyenlõ 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<> típusról bõvebben a Generikusok címû 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 is, amely megvalósítja az IComparer interfészt. 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 megfelel paraméter..."));
        }
}
```

A metódus elkészítésénél a zegyszerûség miatt feltételeztük, hogy a z összehasonlított osztály megvalósítja az IComparable interfészt, természetesen mi magunk is megírhatjuk az eredményt elõállító programrészt. Ezután a következõképpen rendezhetjük a listát:

```
list.Sort(new ComparableClassComparer());
```

Az IComparer elõnye, hogy nem kötõdik szorosan az osztályhoz (akár anélkül is megírhatjuk, hogy ismernénk a belsõ szerkezetét), így többféle megvalósítás is lehetséges.

24.3 *Mátrix típus*

Készítsük el a mátrix típust és valósítsuk meg rajta az összeadás mûveletet! Az egyszerûség 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;
    }
}
```

Mátri xokat úgy adunk össze, hogy az a zonos indexeke n lévõ ér tékeket öszeadjuk:

123 145 1+1 2+4 3+5
456 + 532 = (stb…)
789 211

Az összeadás m ûveletet csakis a zo nos nagyságú dimenziók mellet le het e lvége zni (3x3 –as mátrixhoz nem lehet hozzáad ni egy 4x4 –eset). Ezt elle nõriztük is az operátor megvalósításána l.

Most csak az összeadás m ûveletet valósítottuk meg, a többi a kedves olvasóra vá r. Plusz feladatként i nde xellenõr zést is lehet végezni a z inde xelõ nél.

25 Delegate –ek

A delegate –ek olya n típusok amelyek egy vag y több metódusra hivatko znak. Minden delegate különá lló objektum, amely egy listát táro l a meghívandó me tóudsokról (értele mszer ûen ez eg yúttal e rõs refere ncia is lesz a metódust szolgáltató osztályra).
Nemcsak példá ny - , hanem statikus metód usokra is m utathat. Egy delegate deklarációjánál megadjuk, hogy milyen szig natúráva l re nde lkezõ metód usok megfele lõek :

```
delegate int TestDelegate ( int x );
```

Ez a delegate o lyan metódusra m utathat amelynek visszatérési értéke int típ usú és egyetle n int típusú paramétere van:

```
public int Pow ( int x )
{
    return ( x * x);
}
```

A használata:

```
TestDelegate dlgt = Pow ;
int result = Pow ( 10 );
```

A delegate hívásako r az összes a listájá n lévõ metódust neghívja . A delegate – ekhez a += illetve + operátorokkal hozzáadni a -= és – operátorokkal elve nni tud unk 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 paraméterül, 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 egyenlõ, ha mindkettõ 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 kovariáns (covariant) viselkedésnek nevezzük.

25.2 *Névtelen metódusok*

Egy delegate–nek nem kötelezõ létezõ metódust megadnunk, lehetõségünk van helyben kifejteni egyet. Természetesen ez a névtelen metódus a program többi részébõl közvetlenül nem hívható, csak a delegate–en keresztûl.

```
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 õt tároló blokk lokális változóit és módosíthatja is õket:

```
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 külsõ változóra a delegate is erõs 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 használhat, hogy a saját állapota megváltozásakor értesítsen más osztályokat. Ehhez a „megfigyelõ” osztályoknak fel kell iratkozni a „megfigyelt” osztály eseményére azáltal, hogy az elõbbiek rendelkeznek egy, az eseménynek megfelelõ szignatúrájú metódussal, ún. eseménykezelõkkel. Az esemény megtörténtekor ezek a metódusok fognak lefutni. Eddig ez nagyon úgy hangzik, mintha a delegate –ekrõl beszéltünk volna, és valóban egy esemény tulajdonképpen egy speciális delegate. Egy esemény három dologban különbözik egy delegate –tõl, ezek a következõek:

- 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 egyszerû 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 oszt ly llapota megv ltozott!");
        }
    }
}
```

Nem feltétlenül kell delegate –t deklarálnunk, mivel rendelkezésünkre áll a beépített általános EventHandler delegate amely két paraméterrel (errõl hamarosan) rendelkezik és void visszatérési típussal bír.

Az esemény akkor fog beindulni, amikor a data mezõ értéke megváltozik. Ekkor meghívjuk az OnStatusChanged metódust, amely elsõként megvizsgálja, hogy az eseményre feliratkoztak – e vagy sem, mivel utóbbi esetben a hívás kivételt váltana ki. Ezt az osztályt 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 eseményekhe z rendelt eseményke zelõknek konvenció sze rint (ettõ l elté rhe tünk, de a Framework eseményei mind ilye nek) két paramétere van, az elsõ az az objektum, ame ly kiváltotta az eseményt, a második pedig az eseményhe z kapcsolódó információk. A második paraméter ekkor olyan típus le het, amely az EventArgs osztályból származik. Módosítsuk ennek megfelelõen a fenti programot. Elsõként készítünk egy Eve ntArgs osztályból szárma zó új osztályt, amely képes tárolni az eseményhez kapcsolódó üzene tet (az EventArgs alapértelme zette n nem redelekezik i lyesmivel csak egy alaposztály a specializált ese ményekhe z) :

```
class TestEventArgs : EventArgs
{
    public TestEventArgs(string message)
        : base()
    {
        this.Message = message;
    }

    public string Message { get; set; }
}
```

Ezutá n már csak módosítani kell a delegate –et és a z esemény kiváltásá t:

```
public delegate void EventHandlerDelegate(object sender, TestEventArgs e);

private void OnStatusChange()
{
    if (TestStatusChanged != null)
    {
        TestStatusChanged(this, new TestEventArgs("Az oszt ly llapota
megv ltozott"));
    }
}
```

A sender paraméter nek a this –szel adjuk meg az értékét, ezután exp licit konverzió val visszakap hatjuk belõle a küldõ példányt (az elsõ paramé ter szi nté n konvenció szeri nt minden esetben object típusú lesz, mi vel ug ya nazt az eseményt használhatjuk különbözõ osztályokkal).

Még módosítsuk az eseményke zelõt is:

```
static public void Handler(object sender, TestEventArgs e)
{
    Console.WriteLine(e.Message);
}
```

A következõ példában az események valódi hasznát fogjuk látni. Készítsünk egy egyszerû kliens-szerver alkalmazást (persze nem a hálózaton, csak szimuláljuk). A kliensek csatlakozhatnak a szerverhez (ezután pedig kiléphetnek). A feladat, hogy minden ilyen eseményrõl 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 egyszerû, hiszen gondoskodni kell arról, hogy a kilépett klienseket töröljük a listából illetve a lista mérete is gondot jelenthet. Események alkalmazásával viszont nagyon egyszerû lesz a dolgunk.
Elsõként készítsünk egy EventArgs osztályt:

```
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 ServerEventHandler(object sender,
ServerEventArgs e);

    public event ServerEventHandler ServerChange;

    public Server() { }

    public void Connect(Client client)
    {
        this.ServerChange += client.ServerMessageHandler;
        OnServerChange(string.Format("Felhaszn ló <{0}> csatlakozott!",
client.Name));
    }

    public void Disconnect(Client client)
    {
        OnServerChange(string.Format("Felhaszn ló <{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 felirakozásért/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 kódújrafelhasználás, vagyis, hogy egy adott kódrészletet elég általánosra írjunk meg ahhoz, hogy minél többször felhasználhassuk. Ennek megvalósítására két eszköz áll rendelkezésünkre, az egyik az öröklõdés, a másik pedig jelen fejezet tárgya a generikusok.

26.1 Generikus *metódusok*

Vegyük a következõ 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 mûködjön, 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 ebbõl 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, cserébe sokkal biztonságosabb a használatuk. Két fontos különbség van a kettõ

közt, míg a C++ fordítási idõben készíti el a specializált metódusokat/osztályokat, addig a C# ezt a mûveletet futási idõben végzi el. A másik eltérés az elsõbõl következik, mivel a C++ fordításkor ki tudja szûrni azokat az eseteket, amelyek hibásak, pl. összeadunk két típust a sablonmetódusban, amelyeken nincs értelmezve összeadás. A C# ezzel szemben kénytelen az ilyen problémákat megelõzni, a fordító csakis olyan mûveletek elvégzését fogja engedélyezni, amelyek mindenképpen mûködni fognak. A következõ példa nem fog lefordulni:

```
static public T Sum<T>(T x, T y)
{
    return x + y;
}
```

Fordításkor csak azt tudja ellenõrizni, hogy létezõ 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 legkézenfekvõbb 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("Ures..."));
    }
}
```

Ezt most a következõképpen 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());
    }
}
```

Mûködni mûködik, de se nem hatékony se nem kényelmes. A hatékonyság az érték/referenciatípusok miatt csökken jelentõsen (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("Ures..."));
    }
}
```

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 megs*zorítások*

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 egyenlõ vele.

Nézzünk néhány példát:

```
class Test<T>
    where T : class
{

}
```

Ezt az osztályt csak referenciatípusú generikus praméterrel példányosíthatjuk, minden más esetben fordítási hibát kapunk.

```
class Test<T>
    where T : struct
{

}
```

Ez pedig épp az ellenkezõje, 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 õsosztályára 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 alapérelmezett 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 belõle 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
}
```

Értelemszerûen írhattunk volna <Base, Base> - t, vagy <Derived, Derived> -et is.

26.4 *Öröklõdés*

Generikus osztályból származtathatunk is, ekkor vagy az õsosztály 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 gyûjtemények*

A C# 2.0 bevezetett néhány hasznos generikus adatszerkezetet, többek közt listát és vermet. Ezeket a típusokat a tömbökhöz hasonlóan használhatjuk. A következõkben megvizsgálunk ezek közül néhányat. Ezek a szerkezetek a System.Collections.Generic névtérben találhatóak.

26.6.1 List<T>

A List<T> az ArrayList generikus, erõsen típusos megfelelõje. 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 mûvelettel, amely viszont igen, ezek fõleg a kereséssel kapcsolatosak. Azért, hogy az ebbõl következõ teljesítményromlást elkerüljük a használt értéktípusnak meg kell valósítania az IComparable és az IEquatable interfészeket (a legtöbb beépített egyszerû (érték) típus ezt meg is teszi) (ezeket az interfészeket az összes többi gyûjtemény 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 indexelõ operátort is.

A lista elemeit könnyen redezhetjük 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 a z elemek kö zött a BinarySearch metódussal, ame ly a kerese tt objektum i ndexét adja vissza:

```
using System;
using System.Collections.Generic;

class Program
{
    static public void Main()
    {
        List<string> list = new List<string>()
        {
            "alma", "dió", "k rte", "barack"
        };

        Console.WriteLine(list[list.BinarySearch("k rte")]);
    }
}
```

Megkereshetjük az összes olyan elemet is, amely eleget tesz eg y fe ltételnek a Find és FindAll metódusokkal. Elõbbi a z elsõ, utóbbi az összes meg felelõ példá nyt adja vissza eg y 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 p ros sz m a list ban: {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 sszes p ros elem a list ban:");

        evenList.ForEach(delegate(int item)
            {
                Console.WriteLine(item);
            });
    }
}
```

A feltételek megadásánál és a páros számok listájá nak kiíra tásánál névtele n metódusokat használtunk.
Újdo nságot je lent a listán metód usként hívott foreach szerkezet. Ezt a C# 3.0 veze tte be, az összes generikus adatszerke zet rendelkezik vele, lé nyegében te ljese n ug yanúgy m ûködik mint egy „igazi” foreach. A paramétereként eg y „ 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 õket:

```
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("kett ", 2);
        list.Add("h rom", 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 (ellenkezõ 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 belsõ szerkezetben van.
A SD új (rendezetlen) elemek beszúrását gyorsabban végzi mint a SL (O(Log n) és O(n)). Elõre rendezett elemek beszúrásánál pont fordított a helyzet. Az elemek közti keresés mindkét szerkezetben O(Log n). Ezen kívûl 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("kett ", 2);
        list.Add("h rom", 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) nagyságrendû mûvelet. A lista minden tagja különálló objektum egy-egy LinkedListNode<T> példány. A LLN<T> Next és Previous tulajdonságai az elõzõ illetve a következõ elemre mutatnak.
A lista First tulajdonsága az elsõ, Last tulajdonsága pedig az utolsó tagra mutat. Elemeket az AddFirst (elsõ helyre) és AddLast (utolsó helyre) 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("k rte");
        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 elsõ helyen, mivel õt az AddFirst metódussal helyeztük be.

26.6.5 ReadOnlyCollection<T>

Ahogy a nevébõl 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", "k rte", "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 mûködik 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 õ esetükben egyáltalán nem jár semmilyen „extra” kötelezettséggel.

26.8 *Kovariancia é*s kontravariancia

Nézzük a következõ „osztályhierarchiát”:

```
class Person
{

}

class Student : Person
{

}
```

A polimorfizmus elve miatt minden olyan helyen, ahol egy Person objketum használható ott egy Student objektum is megfelel, legalábbis elvileg. Lássuk, a következõ 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 egyenlõnek a generikus praméthereket, még akkor sem, ha azok kompatibilisak lennének. Azt mondjuk, hogy a generikus paraméterek nem kovariánsak (covariance). A dolog fordítottja is igaz, vagyis nincs kompatibilitás a záltalánosabb típusról a szûkebbre sem, vagyis 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 típusellenõrzéssel 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 következõ példában egy generikus delegate segítségével nézzük meg az új lehetõségeket (új listaszerkezetet írni bonyolultabb lenne) (a példa megértéséhez szükség van a lambda kifejezések ismeretére). Elsõként 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 dekla rációját:

```
delegate void Method<out T>();
```

Most viszont minden mûködik, hiszen biztosítottuk, hogy mi nde n típ ust megfe lelõen kezeljük. Most lássuk a ko ntra varianciát:

```
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 –ban az összes fontosabb i nterfész (pl.: IEnumerable , IComparab le, etc...) képes a ko variáns illetve ko ntra variáns viselkedésre.

27 Lambda kifejezések

A C# 3.0 bevezeti a lambda kifejezéseket. Egy lambda kifejezés gyakorlatilag megfelel egy névtelen metódus „civilizáltabb”, elegánsabb változatának (ugyanakkor elsõ ránézésre talán ijesztõbb, 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 bemenõ változók, jobb oldalán pedig a bemenetre alkalmazott kifejezés áll.
Mivel névtelen metódus ezért egy lambda kifejezés állhat egy delegate értékadásában is, elsõként 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 bemenõ paramétert vár és ugyanilyen típust ad vissza. A lambda kifejezés bal oldalán a bemenõ paraméter (x) jobb oldalán pedig a visszadatott értékrõl gondoskodó kifejezés (x * x) áll. A bemenõ 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 bemenõ paramétert használhatunk, a következõ 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 elõzõleg 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). Elsõként 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, elõtte pedig a bemenõ 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 elsõ paraméter nagyobb –e a másodiknál. Értelemszerûen a lambda operátor bal oldalán lévõ kifejezésnek megfelelõ típust kell eredményeznie, ezt a fordító ellenõrzi.
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 bemenõ paramétert kaphat, és 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 szereplõ adatokat és mûveleteket, hogy futási idõben 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 elõször IL kódra kell fordítani (Compile), csak azután hívhatjuk meg. Kifejezésfákról még olvashatunk a LINQ –rõl szóló fejezetekben.

27.3 *Lambda kifejezések változóinak hatóköre*

Egy lambda kifejezésben hivatkozhatunk annak a metódusnak a paramétereire és lokális változóira amelyben definiáltuk. A külsõ változók akkor értékelõdnek ki amikor a delegate ténylegesen meghívódik, nem pedig a deklaráláskor, vagyis az adott változó legutolsó értékadása számít majd. A felhasznált változókat inicializálni kell mielõtt használnánk egy lambda kifejezésben. A lambda kifejezés fenntart magának egy másolatot a lokális változóból/paraméterbõl, még akkor is, ha az idõközben kifut a saját hatókörébõl:

```
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 –t fog kiír ni, vag yis valóban a legutolsó értékét használta a lambda a lokális válto zónak.

A lokális vá lto zók és paraméte rek módosíthatóak egy lambda kifeje zésbe n. A lambda kifejezésben létre hozo tt változók ugya núgy viselkednek, mint a hagyományos lokális válto zók, a delegate mi nden hívásakor új példány jön lé tre belõlük.

27.4 *Névtelen metódusok kiváltása lambda kifejezésekkel*

Lambda kifeje zést használha tunk mi nden olyan helyen, a hol névte len metód us állhat. Nézzük meg pl., hog y hog ya n használhatjuk így a List<T> típ ust:

```
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 els p ros sz m: {0}", result);

            List<int> oddList = list.FindAll((item) => (item % 2 != 0));

            Console.WriteLine("Az sszes p ratlan sz m:");

            oddList.ForEach((item) => Console.WriteLine(item));
        }
}
```

Eseménykezelõt 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("Eseménykezel !");
            };

        t.OnTestEvent();
    }
}
```

Lambda kifejezés helyett ún. lambda állítást írtunk, így akár több soros utasításokat is adhatunk.

28 Attribútumok

Már találkoztunk nyelvbe épített módosítokkal, 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 elõfordító (ld. következõ fejezet) által definiált szimbólumhoz köti programrészek végrehajtását. A Conditional a System.Diagnostics névtérben rejtõzik:

```
#define DEBUG // definiljuk 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, ellenkezõ esetben a program nem fordul le.

Minden olyan osztály, amely bármilyen módon a System.Attribute absztrakt osztályból származik felhasználható attribútumként. Konvenció szerint minden attribútum oszály neve a névbõl 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 ez az attribútum sem nem túl hasznos. Módosítsuk is egy kicsit! Egy attribútumosztályhoz több szabályt is köthetünk a használatára vonatkozóan, pl. megadhatjuk, hogy milyen típusokon használhatjuk. Ezeket a szabályokat az AttributeUsage osztállyal deklarálhatjuk. Ennek az osztálynak egy kötelezõen megadandó és két opcionális paramétere van: ValidOn illetve AllowMultiple és Inherited. Kezdjük az elsõvel: 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 mûködni fog érték- és referenciatípusokkal is. Az AttributeTargets.All pedig azt mondja meg, hogy bárhol használhatjuk az attributumot.

Lépjünk az AllowMultiple tulajdonságra! Õ 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 m r sok
class C { }
```

Végül az Inerited 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 leszármazotta és mivel az attribútum Inherited tulajdonsága tr ue ér téket kapott ezért õ is örökli az õsosztálya attribútumát.

Attribútumok kétféle paraméterrel rendelkezhetnek: positional és named. Elõbbinek mindig kötelezõ értéket adnunk, tulajdonképpen eza konstruktor paramétere lesz (ilyen az AttributeUsage osztály ValidOn tulajdonsága amelynek mindig kell értéket

adnunk). Utóbbiak az opcióná lis paraméterek, amelyeknek van alapértelmeze tt értéke, így nem kell kötelezõe n megadnunk õket.

Eljött az ideje, hogy egy használható a ttribútumot készítsünk, még hozzá egy o lyat amellyel meg jegyzéseket fûzhetünk egy osztályhoz vag y struktúrá hoz. Az attribútumosztály nagyo n eg yszerû 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 értékei hez 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 oszt ly" )]
class C { }

class Program
{
    static public void Main ()
    {
        Attribute [] attributes =
Attribute . GetCustomAttributes ( typeof ( C));

        foreach ( Attribute attribute in attributes )
        {
            if ( attribute is DescriptionAttribute )
            {

    Console . WriteLine ((( DescriptionAttribute ) attribute ). Description );
            }
        }
    }
}
```

29 Unsafe kód

A .NET platform legnagyobb eltérése a natív nyelvektõl 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 hibalehetõség egész egyszerûen eltûnik. 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 lehetõ legjobb teljesítményt szeretnénk elérni egy rendkívûl számításigényes feladathoz (pl.: számítógépes grafika).
- .NET –en kívûli osztálykönyvtárakat akarunk használni (pl.: Windows API hívások).

A C# a memória direkt elérését mutatókon keresztûl teszi lehetõvé. Ahhoz, hogy használhassunk mutatókat (pointereket) 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 következõ 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);
        }
    }
}
```

Elõször 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 –bõl és konverziós kapcsolat sincs közöttük (bár az egyszerû numerikus típusokról létezik explicit konverzió). Értelemszerûen boxing/unboxing sem alkalmazható rajtuk. Egy pointer mindig egy memóriacímet hordoz, amely memóriaterületen egy teljesen normális objektum van.

Ebbõl következõen 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 lévõ objektum értékét, ezt a dereference operátorral (*) tehetjük meg. Ez visszaadja a mutatott értéket, míg ha csak 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 kezdõcíme. A pointer úgy tudja visszaadni az értékét, hogy tudja mekkora méretû az objektum (pl. egy int pointer egy 32 bites – 4 byte – területet vesz majd elõ).

A programot parancssorból az /unsafe kapcsolóval fordíthatjuk le, Visual Studio esetén jobb klikk a projecten, Properties és ott állítsuk be.

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

A megfelelõ 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 amely nem tartalmaz az eddig felsoroltakon kívül mást hívatkozhat. Ezeket a típusokat összefoglaló néven unmanaged típusoknak nevezzük.

Explicit konverzió létezik bármely két pointer típus között, ezért fennálhat a veszélye, hogy ha A és B pointer nem ugyanakkora méretû területre mutat akkor az A –ról B –re való konverzió nem definiált mûködést 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 str uktúrára is hiva tkozhatunk pointerrel, ek kor a tagjait kétféleképpen érhetjük e l: vag y a mutató n keresztûl, vag y a nyíl (->) operátorral amely tulajdonképpen a z elõbbi 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 szemétgyûjtõ a memória töredeze ttségme ntesítése érdekében mozgatja a z objektumokat a m emóriában. Egy pointer azo nban mindig egy fix helyre mutat, hisze n a mutatott o bjektumnak a címét kértük le, ami pedig nem fog frissülni. Ez néhány esetben go ndot oko zhat (pl. amikor hoszabb ideig van a me móriában a mutatott adat, p l. va lamilyen erõforrás) . Ha szere tnénk, hogy az ob jektumok a helyükön maradja nak a fixed kulcsszót kell használnunk.
Mielõtt jobban megné znénk a fixed –et vegyük szemüg yre a követke zõ 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 mégiscsak használhatunk rajtuk pointert, igaz nem az eddig látott módon (valamint a tömb elemeinek unmanaged típusnak kell lenniük). Referenciatípuson belûl deklarált értéktípusok esetén (ilyenek a tömbök is) fixálni kell az objektum helyzetét, hogy a GC ne mozdíthassa el. A következõ kód már mûködni 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));
            }
        }
    }
}
```

Egy Windows alapú operációs rendszerben minden futtatható állomány indításakor egy ún. process jön létre, amely teljes mértékben elkülönül az összes többitõl. Egy process –t az azonosítója (PID – Process ID) alapján különböztetünk meg a többitõl. Minden egyes process rendelkezik egy ún. fõ szállal (primary– vagy main thread), amely a belépési pontja a programnak (ld. Main).
Azokat az alkalmazásokat, amelyek csak a fõ szállal rendelkeznek thread-safe alkalmazásoknak nevezzük, mivel csak egy szál fér hozzá az összes erõforráshoz.
Ugyanakkor ezek az alkalmazások hajlamosak „elaludni”, ha egy komplexebb feladatot hajtanak végre, hiszen a fõ szál ekkor nem tud figyelni a felhasználó interakciójára.
Az ilyen helyzetek elkerülésére a Windows (és a .NET) lehetõvé teszi másodlagos szálak (ún. worker thread) hozzáadását a fõ szálhoz. Az egyes szálak (a process–ekhez hasonlóan) önállóan mûködnek a folyamaton belül és „versenyeznek” az erõforrások használatáért (concurrent access).
Jó példa lehet a szálkezelés bemutatására egy szövegszerkesztõ használata: amíg kinyomtatunk egy dokumentumot (egy mellékszállal) az alkalmazás fõ szála továbbra is figyeli a felhasználótól érkezõ utasításokat.
A többszálú programozás legnagyobb kihívása a szálak és feladataik megszervezése, az erõforrások 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 között a processzoridõt (ezt a a szálak prioritása alapján teszi) ez az ún. idõosztásos (time slicing) rendszer. Amikor egy szálnak kiosztott idõ lejár, akkor a futási adatait eltárolja az ún. Thread Local Storage –ben (ebbõl minden szálnak van egy) és átadja a helyet egy másik szálnak, amely – ha szükséges – betölti a saját adatait a TLS-bõl és elvégzi a feladatát.

A .NET számos osztályt és metódust bocsájt 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 azo nosítóját, akkor haszná l hatjuk a Process.GetProcessById(*azonosító*) metód ust is.

A következõ programunk a z összes futó process minde n szálá t és a zoknak 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}) sz lai",
                process.ProcessName);

            ProcessThreadCollection ptc = process.Threads;

            foreach (ProcessThread thread in ptc)
            {
                Console.WriteLine("Id: {0}, Állapot: {1}",
                    thread.Id, thread.ThreadState);
            }
        }
    }
}
```

Elég valószínû, hogy a program futásakor kivételt kapunk, hiszen a szálak listájába olya n szál is bekerülhet amely a kiíráskor már befeje zte futásá t (ez szi nte mindig a z Id le process esetében fordul elõ, a meggondolás házi feladat, akárcsak a program kivételbi ztossá tétele).
A fenti osztályok segítségével remekül bele le het lá tni a rendszer „ lelkébe” , az MSDN –en megtaláljuk a fe nti osztályok to vábbi metódusait, tulajdo nságait amelyek a z „utazásho z” szükségesek.

A következõ programunk a process –ek írá nyítását szemlélteti, ind ítsuk el a z Internet Explore rt, 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();
    }
}
```

Együttal felhasználtuk az elsõ 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 elõsegítik 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, a többi még tökéletesen mûködik majd.
Amikor elindítunk egy .NET programot elsõként az alapértelmezett AD (default application domain) jön létre, ezután –ha szükséges – a CLR további AD –ket hoz létre.
A következõ 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 õ az alapértelmezett AD és egyelõre 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. Elsõ programunkban lekérjük az adott programrész szálának 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 elsõ, a „fõ” szálban vagyunk.

A program utasításainak végrehajtása szerint megkülönböztetünk szinkron- és aszinkron mûködést. A fenti program szinkron módon mûködik, az utasításait egymás után hatja végre, ha esetleg egy hosszas algoritmusba ütközik akkor csak akkor lép a következõ utasításra ha azt befejezte,
Az aszinkron végrehajtás ennek épp az ellentéte az egyes feladatokat el tudjuk küldeni egy másik szálba a fõ szál pedig fut tovább, amíg a mellékszál(ak) vissza nem térnek.

30.3 Aszinkron delegate-ek

A következõkben 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 következõ 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);
}
```

Egyelõre ne foglalkozzunk az ismeretlen dolgokkal, nézzük meg azt amit ismerünk. A BeginInvoke –kal fogjuk meghívni a delegate –et, ennek elsõ 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 egyszerû delegate–et és hívjuk meg aszinkron:

```
using System;
using System.Threading;

class Program
{
    public delegate int MyDelegate(int x);

    static int Square(int x)
    {
        Console.WriteLine("Sz l-ID: {0}",
            Thread.CurrentThread.ManagedThreadId);
        return (x * x);
    }

    static public void Main()
    {
        MyDelegate d = Square;

        Console.WriteLine("Sz l-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 következõ 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 külsõ szemlélõként 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 elõbb 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, késõbb indult szálnak. Ezt szinronizálásnak nevezzük.
Szinkronizáljuk az eredeti programunkat, vagyis várjuk meg amíg a delegate befejezi a futását (természetesen a szinkronizálás ennél jóval bonyolultabb, errõl a következõ fejezetekben olvashatunk):

```
using System;
using System.Threading;

class Program
{
    public delegate int MyDelegate(int x);

    static int Square(int x)
    {
        Console.WriteLine("Sz l-ID: {0}",
            Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(100);
        return (x * x);
    }

    static public void Main()
    {
        MyDelegate d = Square;
        Console.WriteLine("Sz l-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 elkezdõdött, igaz a Main itt is megelõzte.

A Square metódusban azért használtuk a Sleep metódust, hogy lássunk is valamit, ellenkezõ esetben túl gyorsan lefut ez az egyszerû program. Erõsebb 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("Sz l-ID: {0}",
            Thread.CurrentThread.ManagedThreadId);
        return (x * x);
    }

    static void AsyncMethodComplete(IAsyncResult iar)
    {
        Console.WriteLine("Aszinkron sz l 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("Sz l-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), " zenet a j v b l :)");
```

Az üzenetet az IAsyncResult AsyncState tulajdonságával kérdezhetjük le:

```
static void AsyncMethodComplete(IAsyncResult iar)
{
    Console.WriteLine("Aszinkron sz l kész...");

    AsyncResult result = (AsyncResult)iar;
    MyDelegate d = (MyDelegate)result.AsyncDelegate;

    Console.WriteLine(" zenet: {0}", iar.AsyncState);
    Console.WriteLine("Eredmény: {0}", d.EndInvoke(iar));
}
```

30.3.1 *Párhuzamos delegate hívás*

A .NET 4.0 lehetõvé teszi, hogy delegateket 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 megfelelõ:

```
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"));
    }
}
```

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 õket. Vegyük a következõ programot:

```
using System;
using System.Threading;

class Test
{
    public void ThreadInfo()
    {
        Console.WriteLine("Sz l-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();
    }
}
```

Elsõként lekértük és elneveztük az elsõdleges szálat, hogy késõbb azonosítani tudjuk, mivel alapértelmezetten nincs neve.

A következõ programban a Test objektum metódusát egy háttérben futó szálból fogjuk meghívni:

```
using System;
using System.Threading;

class Test
{
    public void ThreadInfo()
    {
        Console.WriteLine("Sz l-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 szereplõ 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("Sz l-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 ellenõrizni a praméter típusát mielõtt bármit csinálunk vele.

30.5 *Foreground és background szálak*

A .NET két különbözõ száltípust különböztet meg: amelyek elõtérben és amelyek a háttérben futnak. A kettõ közti különbség a következõ: a CLR addig nem állítja le az alkalmazást, amíg egy elõtérbeli 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 elsõdleges és másodlagos szálak fogalma megegyezik jelen fejezetünk tárgyaival. Az igazság viszont az, hogy ez az állítás nem igaz, ugyanis alapértelmezés szerint minden szál (a létrehozás módjától és idejétõl függetlenül) elõtérben fut. Természetesen van arra is lehetõség, hogy a háttérbe küldjük õket:

```
using System;
using System.Threading;

class Test
{
    public void ThreadInfo()
    {
        Thread.Sleep(5000);
        Console.WriteLine("Sz l-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 tõle, mivel beállítottuk az IsBackground tulajdonságot, ezért az általunk készített szál „valódi” háttérben futó szál lett, vagyis a fõszálnak 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 elsõ 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 környezete leget 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 idõ lejárta után – ha a szál nem végzett feladatával – hamis értékkel tér vissza. A következõ 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 id lej rt...");
            t.Abort(); // megszakítjuk a sz l fut s t
        }
    }
}
```

A következõ szinkronizációs módszer a lezárás (locking). Ez azt jelenti, hogy erõforrásokhoz, illetve a program bizonyos részeihez egyszerre csak egy szálnak engedünk hozzáférést. Ha egy szál hozzá akar férni a zadott dologhoz, amelyet egy másik szál már használ, akkor automatikusan blokkolódik és várólistára kerül, ahonnan érkezési sorrendben lehet hozzájutni az erõforráshoz (ha az elõzõ szál már végzett).
Nézzük a következõ 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 elsõként 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 egyszerû program muszáj lelassítani egy kicsit az elsõ szálat (érdemes többször lefuttatni, nem biztos, hogy azonnal kivételt kapunk).
A mûveletet lezárhatjuk a következõ 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 mezõk használata. A jegyzet ezt a témát nem tárgyalja, mivel a megértéséhez tisztában kell lennünk a fordító sajátosságaival, a következõ – angol nyelvû – weboldalon bõvebb információt találakedves 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 hasonlóa n m ûködik a Mute x is, a legnagyobb különbség a z a kettõ kö zt, hogy utóbbi processz szinten zárol, a za z a szám ítógépen futó összes folya mat elöl elzá rja a használat lehe tõségét. Az erõ forrás/metódus/stb. használa ta elõtt meg kell hívnunk a WaitOne metódust, a használat után pedig el kell engednünk a z erõforrást a ReleaseMutex metód ussal (ha ezt nem tesszük meg a kódból, akkor az alkalma zás futásának végé n a CLR automatikusan megteszi he lyettünk). A követke zõ példában létreho zunk több szálat és versenye ztetjük õket eg y metód us haszná latáér t. Elsõként készítsük el a z osztályt, amely tárolja a z „erõfor rást” és a m utex ob jektumo t:

```
class Test
{
    private Mutex mutex = new Mutex();

    public void ResourceMetod()
    {
        mutex.WaitOne();

        Console.WriteLine("{0} haszn lja az er forr st...",
            Thread.CurrentThread.Name);

        Thread.Sleep(1000);

        mutex.ReleaseMutex();

        Console.WriteLine("{0} elengedi az er forr st...",
            Thread.CurrentThread.Name);
    }
}
```

Most pedig jöjjön a fõp rogram:

```
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 erõforráshoz maximum hány szál férhet hozzá egy idõben. A következõ program az elõzõ átirata, egy idõben maximum három szál férhet hozzá a metódushoz:

```
class Test
{
    private Semaphore semaphore = new Semaphore(3, 3);

    public void ResourceMetod()
    {
        semaphore.WaitOne();

        Console.WriteLine("{0} haszn lja az er forr st...",
            Thread.CurrentThread.Name);

        Thread.Sleep(1000);

        semaphore.Release();

        Console.WriteLine("{0} elengedi az er forr st...",
            Thread.CurrentThread.Name);
    }
}
```

A fõprogram pedig ugyanaz lesz.

30.7 ThreadPool

Képzeljük el a következõ szituációt: egy kliens-szerver alkalmazást készítünk, a szerver a fõszálban figyeli a bejövõ kapcsolatokat, ha kliens érkezik akkor készít neki egy szálat és a háttérben kiszolgálja. Tegyük még hozzá azt is, hogy a kliensek viszonylag rövid ideig vannak kapcsolatban a szerverrel, viszont sokan vannak.
Ha úgy készítjük el a programot, hogy a bejövõ kapcsolatoknak mindig új szálat készítünk, akkor nagyon gyorsan teljesítményproblémákba fogunk ütközni:

1. Egy objektum létrehozása költséges.

2. Ha már nem használjuk, akkor a memóriában marad amíg el nem takarítja a GC.
3. Mivel sok bejövõ 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 megfelelõ ThreadPool osztályt is).
Egy késõbbi fejezetben elkészítünk majd egy, a fenti felvázolt helyzethez hasonló programot, most „csak" egy egyszerû példán keresztûl fogjuk megvizsgálni ezt a technikát. Nézzük meg a következõ programot:

```
using System;
using System.Threading;

class Program
{
    static public void Do(object inObj)
    {
        Console.WriteLine("A k vetkez adatot haszn ljuk: {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 elsõsorban feltûnhet, 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 elsõ 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 elsõ 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 fõszálban végtelen ciklusban várja a bejövõ kapcsoaltokat).

31 Reflection

A programozástechnikában a „reflection" fogalmát olyan programozási technikára alkalmazzuk, ahol a program (futás közben) képes megváltoztatni saját struktúráját és viselkedését. Az erre a paradigmára épülõ 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 következõ példát:

```
using System;

class Test
{

}

class Program
{
    static public void Main()
    {
        Test t = new Test();
        Type type = t.GetType();
        Console.WriteLine(type); // Test
    }
}
```

Futásidõben lekértük az objektum típusát (a GetType metódust minden osztály örökli az object–tõl), persze a reflektív programozás ennél többrõl szól, lépjünk elõre egyet: mi lenne, ha az objektumot úgy készítenénk el, ha 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
    }
}
```

Megjegyzendõ, hogy fordítási idõben semmilyen ellenõrzés 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);
    }
}
```

Az InvokeMember 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/file*ba

A .NET számos osztályt biztosít számunkra, amelyekkel fileokat udunk kezelni, ebben a fejezetben a leggyakrabban használtakkal ismerkedünk meg. Kezdjük egy file megnyitásával és tartalmának képernyõre írásával. Legyen pl. a szöveges file tartalma a következõ:

```
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 mûveletek végzéséhez. Az elsõ sorban megnyitottunk egy ilyen folyamot és azt is megmondtuk, hogy mit akarunk csinálni vele.
A FileStream konstruktorának elsõ paramétere a file neve (ha nem talál filet, akkor kivételt dob a program). Ha nem adunk meg teljes elérési utat, akkor automatikusan a saját könyvtárában fogja keresni a program. Ha külsõ könyvtárból szeretnénk megnyitni az állományt, akkor azt a következõképpen tehetjük meg:

```
FileStream fs = new FileStream("C:\\Dir1\\Dir2\\test.txt", FileMode.Open);
```

Azért használunk dupla backslash –t (\), mert az egy ún. escape karakter, magában nem lehetne használni (persze ez nem azt jelenti, hogy minden ilyen karaktert kettõzve kellene írni, minden speciális karakter elõtt a backslash –t kell használnunk).

Egy másik lehetõség, hogy az „at” jelet (@) használjuk az elérési út elõtt, 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 következõ értékei lehetnek:

Create	Létrehoz egy új file –t, ha már létezik a tartalmát kitörli
CreateNew	Ugyanaz mint az elõzõ, de ha már létezik a file, akkor kivételt dob.
Open	Megnyit egy file –t, ha nem létezik kivételt dob.
OpenOrCreate	Ugyanaz mint az elõzõ, de ha nem létezik akkor létrehozza a file –t.
Append	Megnyit egy file –t és automatikusan a végére pozícionál. Ha nem létezik létrehozza.
Truncate	Megnyit egy létezõ file –t 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 féle konstruktora van), mindkettõ enum típusú. Az elsõ 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 file –hoz:

None	Más folyamat nem férhet hozzá a file –hoz, amíg azt be nem zárjuk.
Read	Más folyamat olvashatja a file –t.
Write	Más folyamat írhatja a file –t.
ReadWrite	Más folyamat írhatja és olvashatja is a file –t.
Delete	Más folyamat törölhet a file –ból (de nem magát a file –t).

Inheritable	A gyermek processzek is hozzá férhetnek a file – hoz.

Ha a fenti programot futtatjuk, akkor elõfordulhat, hogy az ékezetes karakterek helyett kérdõjel jelenik meg. Ez azért van, mert a az éppen aktuális karaktertábla nem tartalmazza ezeket a karaktereket, ez tipikusan nem magyar nyelvû operációs rednszer esetén fordul elõ. A megoldás, hogy kézzel állítjuk be a megfelelõ táblát, ezt a StreamReader konstruktorában tehetjük meg (a tábla pedig az 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 file-ba. 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 file – nak, és az írás után azonnal írjuk ki a képernyõre az új tartalmát.

Az Environment.NewLine egy újsor karaktert ad vissza.

Bináris file –ok 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 file –t, és beleírtuk a számokat egytõl százig. Ezután megnyitottuk a már létezõ file –t és elkezdjük kiovasni a tartalmát. A PeekChar metódus a soron következõ 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 ReadValami metódus a megfelelõ típusú adatot adja vissza, de vigyázni kell vele, mert ha nem megfelelõ méretû a beolvasandó adat, akkor hibázni fog. A fenti példában, ha a ReadString metódust használtuk volna akkor kivétel (EndOfStreamException) keletkezik, mivel a kettõ nem ugyanakkor mennyiségû adatot olvas be. Az integer –eket beolvasó metódus nyílván mûködni fog, hiszen tudjuk, hogy számokat írtunk ki.

Eddig kézzel zártuk le a streameket, 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 file–ok kezelése mellett a .NET a könyvtárstruktúra kezelését is széleskörûen támogatja. A System.IO névtér ebbõl a szempontból két részre oszlik: információs - és operációs eszközökre. Elõbbiek (ahogyan a nevük is sugallja) információt

szolgá ltatnak, m íg a z utóbbiak (többségükben statikus metód usok) bizonyos mûveleteket (új könyvtár létreho zása, törlése, stb.) vége znek a file rendszeren. Elsõ példánkba n ír juk ki mondjuk a C megha jtó gyöke rének 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 file –okra is kívá ncsiak lehetünk. A programunk módosított változa ta némi plusz információval eg yütt e zeket 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("K nyvt r: {0}, Készült: {1}",
di.FullName, di.CreationTime);
            }
            else
            {
                FileInfo fi = fsi as FileInfo;
                Console.WriteLine("File: {0}, készül t: {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));
            }
    }
}
```

Elsõként a mappákon, majd a fi le –okon meg yünk végig. Ugyana zzal a metódussal írjuk ki az információkat kihasználva azt, hog y DirectoryInfo és a FileInfo is egy közös õstõl a FileSystem Info –ból származik (mindke ttõ konstr uktora a vizsgált alany elérési útját várja paraméterként), íg y a metód usban csak meg kell vizsgá lni, hogy

éppen melyikkel va n dolg unk és átko nvertá lni a megfelelõ típusra. A vizsgá latnál egy „bitenké nti és" mûvelete t ha jtottunk végre, hog y e z miért és hogya n m ûködik, annak meggondolása a z olvasó felada ta.

Eddig csak információkat kértünk le, most megtanuljuk módosítani is a könyvtárstr uktúrát:

```
using System;
using System.IO;

class Program
{
    static public void Main()
    {
        string dirPath = "C:\\test";
        string filePath = dirPath + "\\file.txt";

        //ha nem létezik a k nyvt r
        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 CreateTe xt metódusa egy StreamWriter objektumo t ad vissza, amellyel ír hatunk egy szöveges file – ba.

32.3 In – memory streamek

A .NET a Memor yStream osztályt biztosítja számunkra, amellyel memóriabeli adatfolyamokat írhatunk/olvashatunk. Mire is jók ezek a folyamok? Gyakran va n szükségünk arra, hog y összegyûjtsünk nagy mennyiségû adatot, amelyeket majd a folyamat végén ki akarunk ír ni a mere vlemezre. Nyílván egy tömb vagy eg y lista nem nyújtja a megfe lelõ szo lgáltatásokat, elõbbi ruga lmatla n, utóbbi memóriaigényes, ezért a legeg yszer ûbb, ha kö zvetlenül a memóriában tároljuk el a z adatokat. A Memor yStream osztály je le ntõse n megkö nnyíti a dolgunkat, mi vel lehetõséget ad közvetlenül file –ba ír ni a tartalmát. A követke zõ program erre mutat példá t:

```
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 –t. Miután minden adatot a memóriába írtunk a Flush metódussal (amely egyúttal kiûríti a StreamWriter –t is) létrehoztunk egy filet és a MemoryStream WriteTo metódusával kiírtuk belé az adatokat.

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 „külsõ” 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 a fileok tulajdonképpen XML dokumentumok amelyek a következõképpen épülnek fel:

```
<?xml version="1.0" encoding="utf-8" ?>

<configuration>
    <appSettings>
        <add key="1" value="alma" />
        <add key="2" value="korte" />
    </appSettings>
</configuration>
```

Az elsõ 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 (errõl 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 egyszerû 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 indexelõvel, amely visszaadja a megfelelõ 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.

Figyelni kell arra, hogy a konfigfile ne tartalmazzon ékezetes karaktert, egyébként hibaüzenetet kapunk a program futtatásakor.

Az AppSettings indexelõje minden esetben stringet ad vissza, tehát ha más típussal akarunk dolgozni akkor konvertálni kell. Vezessünk be a konfigfileba 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*

Egyszerûbb 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 következõ példában elkészítünk egy programot, amely konfigurációs fileból kiolvas egy filenevet és a kódtáblát és ezek alapján kiírja a file tartalmát. Elsõ lépésként készítsünk egy új forrásfilet AppDataSection.cs néven, amely a következõt 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 indexelõjét használjuk. Itt arra kell figyelni, hogy az AppSettings –szel ellentétben õ object típussal tér vissza, vagyis muszáj konverziót végrehajtani.

A Settings property segítségével az osztályon keresztûl egyúttal hozzáférünk az osztály egy példányához is, így soha nem kell példányosítanunk azt.

A tulajdonságok attribútumával beállítottuk a konfigfileban szereplõ nevet illetve, hogy muszáj megadnunk ezeket az értékeket (a DefaultValue segítségével kezdõértéket is megadhatunk). Most jön a konfigfile 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 assembly –t amelyben a típus szerepel, ez a mi esetünkben a main lesz.
Most jöjjön az osztály amely használja az új szekciót, készítsünk egy DataHandlerClass.cs nevû 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 meglehetõse n széles eszkö ztárral re ndelkezik hálózati kommunikáció kialakításához. A lista a legeg yszerûbb hobbiszintû programokban használható szerver - klie ns osztályoktól egészen a legalacsonyabb szintû osztályokig terjed. Ebben a feje zetbe n ezt a listát fog juk végigjárni.
A fejeze thez tarto zó for ráskódok megtalá lha tóak jegyzethe z tar tozó Sources \Network könyvtárba n.

34.1 Socket

A socketek számítógépes hálóza tok (pl. az Inter net) közötti komm unikációs végpontok. Minden socket rendelkezik két adattal amelyek álta l eg yértelm ûen azo nosítha tóak és elér hetõek, ezek a z IP cím és a por t szám .

Mi az az IP cím? Az Interne t az ú n. Internet Protoco l (IP) szab vá nya szeri nt m ûködik. Eszerint a hálózaton lé võ minden számítógép egyedi azonosítóval – IP cím mel rendelkezik (ug yanakkor egy szám ítógép több címme l is rendelkezhe t ha több háló zati hardvert haszná l) . Eg y IP cím egy 32 bites egész szám, amelye t 8 bites részekre osztunk (pl.: 123.255.0.45), e záltal a z egyes szekciók legmagasabb értéke 255 lesz (e z a je lle mzõ az IP neg yedik generációjára vo natkozik, az új hatos generáció már 128 bites címeket használ, igaz ez egyelõre kevésbé elterjedt (a .NET támogatja az IPv4 és IPv6 ver ziókat is)).

Az IP címet tekinthetjük az ország/város/utca /há zszám nég yesnek, m íg a portszámmal egy szo ba számra hi vatkozunk, esetünkben egy szám ítógépre. A portszám egy 16 bites elõjeles szám 1 és 65535 között. A portszámoka t ki lehet „sajátíta ni”, vag yis ezá lta l biztosítják a nagyobb szoftvergyártók, hogy ne leg ye n semmilyen ütkö zés a termék haszná latakor . A portszámok hivatalos regisztrációját az Inter net Assigned Numbers Authority (IANA) végzi.
Az 1 és 1023 portokat ún. we ll -knowed portké nt ismer jük, e zeken olya n széleskörben elterjedt szolgá ltatások futnak amelyek a leg töb b rendszere n meg taláhatóak (operációs rendszer tõl függetlenül). Pl. a böngészõk a HTTP pro tokollt a 80 – as porton érik el, a 23 a Telnet, míg a 25 az SMTP szer ver portszá ma (ezektõl el lehe t – és bizto nsági okokból a nag yobb cégek el is szoktak – tér ni, de a legtöbb szám ítógépen e zekkel a z értékekkel talá lko zunk).
Az 1024 és 49151 kö zötti portokat regisztrált portoknak neve zik, ezeke n már olya n szolgá ltatásokat is felfede zhetünk amelyek operációs rendszerhe z (is) kötö ttek p l. az 1433 a Microsoft SQL Ser ver por tja , ami érte lemszer ûen Wi ndows re ndszer alatt fut. Ugya nitt megtalálunk szoftverhe z kötött portot is, pl a World of Warcraft a 3724 portot használja . Ez a z a tarto má ny amit az IANA kezel.
Az efelettieket dinamikus vagy pri vát p ortoknak ne ve zik, ezt a t artomá nyt ne m le het lefoglalni, programfe jlesztés ala tt célszer û ezeket használni.

Mielõtt nekiállunk a Socket osztá ly megismerésének játsszunk egy kicsit az IP címekkel! Azt tud juk már, hogy a hálózato n minde n számítógép saját címmel rendelkezik, de számokat viszo nylag nehé z megjegyezni, ezér t feltalálták a domain - né v vagy ta rtománynév inté zményé t, ame ly „rá ül” egy adott IP címre, vagyis ahe lyett,

hogy 65.55.21.250 írhatjuk azt, hogy www.microsoft.com. A következõ 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 vízek felé venni az irányt, elkészítjük az elsõ szerverünket. A legegyszerûbb módon fogjuk csinálni a beépített TcpListener osztállyal amely a TCP/IP protokollt használja (errõl 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 rendelekezésre áll). Ez az ip cím lesz az, ahol a szerver bejövõ kapcsolatokra fog várni (érelemszerûen ehhez az IP címhez csak a saját számítógépünkrõl 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).

A továbbiak megkönnyítésére készítsünk eg y statikus metód ust, amely visszaad egy TcpListene r példá nyt, e ze nkívûl teg yünk a progra munkba kivételkezelést i s (a statikus metódust e zentúl létezõ nek teki ntjük nem lesz benne a fo rráskódokban) :

```
using System;
using System.Net;
using System.Net.Sockets;

public class Server
{
	static public TcpListener CreateTcpListener(string ipaddr, string portnum)
	{
		IPAddress ip = IPAddress.Parse(ipaddr);
		int port = int.Parse(portnum);
		IPEndPoint endPoint = new IPEndPoint(ip, port);

		return new TcpListener(endPoint);
	}

	static public void Main(string[] args)
	{
		TcpListener server = null;

		try
		{
			server = CreateTcpListener(args[0], args[1]);
			server.Start();

			Console.WriteLine("A szerver elindult!");
		}
		catch(Exception e)
		{
			Console.WriteLine(e.Message);
		}
		finally
		{
			server.Stop();

			Console.WriteLine("A szerver le llt!");
		}
	}
}
```

A következõ lépésben bejövõ kapcsola tot fogad unk, ehhe z viszont szükségünk lesz egy kliensre is, õt a TcpClie nt osztállyal készítjük el, ame ly haso nlóan m ûkö dik mint a párja , szi nté n egy cím -port kettõsre lesz szükség ünk:

```
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 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();
```

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 byte-tok sorozatát továbbítja, ezért számára emészthetõ formába kell hoznunk az üzenetet. A fenit példában a legegyzserûbb ASCII kódolást választottuk, de mást is használhatunk a lényeg, hogy tudjuk bytonként küldeni (értelemszerûen mind a klien, 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 keresztûl „folynak” át.

Már ismerjük az alapokat, eljött az ideje, hogy egy szinttel lejjebb lépjünk a socketek világába. Írjuk át a szerverünket úgy, hogy socketeket használjunk:

```
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 egyszerûbb változatban is tettük. Ezt a végpontot a Bind metódussal kötöttük a sockethez, majd a Listen metódussal megmondtuk, hogy figyelje a bejövõ kapcsolatokat. Ez utóbbi paramétere azt jelzi, hogy maximum hány bejövõ kapcsolat várakozhat.
Innentõl kezdve nagyon ismerõs 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 jele nti, mi vel most kapcsolódni akar unk, nem hallgatózni.

34.2 Blokk *elkerülése*

Az eddigi programjaink mind megegyeztek abban, hog y bizo nyos mûve letek blokkolták a fõszá lat és így vá rako zni kényszerültünk. Ilye n mûve let volt pl. az Accept/AcceptTcpClient, de a Read/Receive is.
A blokk elker ülésére ún. elõre -ellenõrzést (prechecking) fogunk a lkalmazni, vag yis megvizsgáljuk, hogy adott idõpillanatba n van -e bejövõ adat, vagy ráérünk késõbb újraellenõrizni, addig pedig csinálhatunk mást.
A TcpListerner /TcpClie nt osztályok a rájuk csatlako ztatott NetworkStream objektum DataAvailable tulajdonságán keresztûl tudják vizsgá lni, hogy jön -e adat vag y sem. A következõ példában a kliens rendszeres idõközönké nt ellenõr zi, hog y érke zett -e válasz a szer ve rtõ l és ha nem, akkor tesz valami mást:

```
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 küld tt v laszt!");
        System.Threading.Thread.Sleep(200);
    }
}
```

Ugya nezt a hatást Socket –ek esetébe n a Socket osztály Available tulajdo nságá val érhetjük el, amely jelzi, hogy van -e még várakozó adat a csato rnán (egészen pontosa n azoknak a byteoknak a szá mát adja vissza ame lyeket 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("V runk a szerver v lasz ra...");
        System.Threading.Thread.Sleep(200);
    }
}
```

Itt egy másik módsze rt választottunk a ciklus ke ze lésére.
Most né zzük meg, hog y mi a helyzet szervero ldalon. Itt a tipikus probléma az, hogy az AcceptTcpClient/Accept te ljese n blokko l am íg bejö võ kapcsola tra várako zunk. Erre is va n pe rsze megoldás, a TcpListener esetében ezt a Pendi ng-, m íg a Socke t –eknél a Poll me tód us jelenti.
Nézzük elsõként 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 m st csin lunk" );
		System . Threading . Thread . Sleep ( 500 );
	}
}
```

A Pending a zt az i nformációt osztja meg velünk, hog y várakozik -e bejö võ kapcsolat. Tulajdonképpe n ez a metód us a következõ forrásban szerep lõ (a Socket osztályhoz tartozó) Poll metódust használja:

```
while ( true )
{
	if ( server . Poll ( 0, SelectMode . SelectRead ))
	{
		client = server . Accept ();

		/* itt pedig kommunik lunk a klienssel */
	}
	else
	{
		Console . WriteLine ( "A szerver bej v kapcsolatra v r!" );
		System . Threading . Thread . Sleep ( 500 );
	}
}
```

A Poll metódus elsõ paramé tere eg y egész szá m amely mikromásodpercben (nem milli-, itt valóban a másodperc milliomod részérõl van szó , vagyis ha eg y másodpercig akarunk vár ni akkor 1000000 – ot kell megadnunk) adja meg azt a z idõt amíg vá runk bejö võ kapcsolatra/adatra. Amennyiben az elsõ paramé ter nega tív szám , akkor addig vár unk amíg ni ncs kapcsolat (vagyis blokkoljuk a programo t), ha pedig nullát adunk meg akkor haszná lhatjuk pre -checking –re is a me tódust.
A második paraméterrel a zt mo ndjuk meg, hog mire várunk. A SelectMode felsorolt típ us három tagga l rendelkezik:

1. SelectRead: a metódus igaz értékkel tér vissza, ha meghívtuk a Listen metódust és várakozik bejövõ kapcsolat, ha van bejövõ adat illetve ha a kapcsolat megszûnt, minden más esetben a visszatérési érték false lesz.
2. SelectWrite: igaz értéket kapunk vissza ha a Connect metódus híváa 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 lehetõség 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) pramétert 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 következõ fejezetben több kliens egyidejû 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 kívûl hagyhattunk:

1. A kliensekre mutató „referencia" tárolása
2. A szerver akkor is tudjon kapcsolatot fogadni amíg a bejelentkezett kliensekkel foglalkozunk.
3. Minden kliens zavartalanul (lehetõleg blokk nélkül) tudjon kommunikálni a szerverrel és viszont.

A következõkben háromféleképpen fogjuk körüljárni a problémát.

34.3.1 Select

Az elsõ versenyzõnk 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 következõképpen néz ki:

```
public static void Select(
    IList checkRead,
    IList checkWrite,
    IList checkError,
    int microSeconds
)
```

Az elsõ három paraméter olyan IList interfészt implementáló gyûjtemények (lényegében az összes beépített gyûjtemény ilyen beleértve a sima tömböket is) amelyek Socket példányokat tartalmaznak. Az elsõ paraméternek megadott listát olvashatóságra, a másodikat írhatóságra míg a harmadikat hibákra ellenõrzi a Select, majd a feltételnek megfelelõ listaelemeket megtartja a listában a többit eltávolítja (vagyis ha szükségünk van a többi elemre is akkor célszerû 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 gyirányú 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 egyszerû egész szám, meghatározzuk vele, hogy amximum hány klienst fogunk kezelni. Miután megfelelõ számú kapcsoatot 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

Korábban már megismerkedtünk az aszinkron delegate–ekkel, ez a lehetõség a Socket osztály esetében is rendelkezésünkre áll. Az aszinkron hívható metódusok Begin és End elõtagot 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 v r...");

    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 elsõ 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 fõszálnak, 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 következõ klienst.
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 megfelelõ metódusok Begin/End elõtagú 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 mûveletet hajthatunk végre a háttérben.
A következõ 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 sz m: {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 j tékos: {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 klie ns rendelkezik né vvel is, ame ly maximum 7 karakter hosszú (7 byte) lehet , elsõké nt ezt olvassuk be. A GuessNumber metód usnak adjuk át a tippünket és a result változót out paraméterként. Végül a fi nally blokkba n ellenõrizzük, hog y van -e bejelentke zett kliens, ha pedig nincs akkor új számo t sorsolunk.
Nézzük a GuesNumber me tódust:

```
private void GuessNumber ( string name , int number , out int result )
{
    lock ( locker )
    {
        if ( NUMBER != - 1)
        {
            Console . WriteLine ( "{0} szerint a sz m: {1}" , name ,
number );

            if ( NUMBER == number )
            {
                Console . WriteLine ( "{0} kital lta a sz mot!" , 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 valaki kitalálta számot, akkor annak -1 –et adunk értékül, így gyorsan ellenőrizhetjük, hogy a feltétel melyik ágába kell bemennünk.
Kliens oldalon sokkal egyszerûbb dolgunk van, ezt a forráskódot itt nem részletezzük, de megtalálható a jegyzethez tartozó források között.

34.4 *TCP és UDP*

Ez a fejezet elsõdlegesen a TCP protokollt használta, de említést kell tennünk az Internet másik alapprotokolljáról az UDP –rõl 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 idõn belül a nyugta nem érkezik meg, akkor a csomagot újra elküldi. Ezenkívûl ellenõrzi, hogy a csomagok sérülésmentesek legyenek, illetve kicsûri a duplán elküldött (redundáns) adatokat is.
Az UDP épp ellenkezõleg nem biztosítja, hogy minden csomag megérkezik, cserében gyors lesz. Jellemzõen olyan helyeken használják, ahol a gyorsaság számít, pl. valós idejû 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 elérhetõ 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 lehetõvé teszi, hogy könnyen, uniformizált úton kezeljük a különbözõ adatforrásokat, vagyis pontosan ugyanúgy fogunk kezelni egy adatbázist, mint egy memóriában lévõ gyûjteményt. 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 lehetõvé teszi, hogy egy plusz réteg (a LINQ „felület") bevezetésével mindezeket áthidaljuk teljes mértékben függetlenül az adatforrástól. Egy másik elõnye pedig, hogy a LINQ lekérdezések erõsen típusosak, vagyis a legtöbb hibát még fordítási idõben el tudjuk kapni és kijavítani.
A LINQ család jelenleg három fõcsapást jelölt ki, ezek a következõek:

- LINQ To XML: XML dokumentumok lekérdezését és szerkesztését teszi lehetõvé.
- LINQ To SQL (vagy DLINQ) és LINQ To Entities (Entity Framework): relációs adatbázisokon (elsõsorban MS SQL-Server) végezhetünk mûveleteket velük. A kettõ közül a LINQ To Entites a „fõnök", a LINQ To SQL inkább csak technológiai demónak készült, a Microsoft nem fejleszti tovább (de továbbra is elérhetõ marad, mivel kissebb projectekre illetve hobbifejlesztésekhez kiváló). Az Entity Framework használatához a Visual Studio 2008 elsõ szervízcsomagjára van szükség.
- LINQ To Objects: ennek a fejezetnek a tárgya, memóriában lévõ gyûjtemények, listák, tömbök feldolgozását teszi lehetõvé (lényegében minden olyan osztállyal mûködik amely megvalósítja az IEnumerable<T> interfészt).

A fentieken kívûl számos third-party /hobbi project létezik, mivel a LINQ „framework" viszonylag könnyen kiegészíthetõ 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 jelentõsen megkönnyíti a dolgunkat. Ezek a következõek:

Kiterjesztett metódusok (extension method): velük már korábban megismerkedtünk, a LINQ To Objects teljes funkcionalitása ezekre épül, lényegében az összes LTO mûvelet a z IEnumerable<T>/IEnumerable interfészeket egészíti ki.

Objektum és gyûjtemény inicializálók: vagyis a lehetõség, hogy az objektum deklarálásával egyidõben beállíthassuk a tulajdonságaikat, illetve gyûjtemények esetében az elemeket:

```
using System;
using System.Collections.Generic;

class Program
{
	static public void Main()
	{
		/*Objektum inicializ ló*/
		MyObject mo = new MyObject()
		{
			Property1 = "value1";
			Property2 = "value2";
		};

		/*Gyujtemény inicializ ló*/
		List<string> list = new List<string>()
		{
			"alma", "k rte", "dió", "kakukktoj s"
		};
	}
}
```

Lambda kifejezések: a lekérdezések többségénél nagyon kényelmes lesz a lambdák használata, ugyanakkor lehetõség van a „hagyományos” névtelen metódusok felhasználására is.

A „var”: a lekérdezések egy részénél egészen egyszerûen lehetetlen elõre 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 bevezetése lehetõvé teszi, hogy helyben deklaráljunk egy névtelen osztályt:

```
using System;

class Program
{
	static public void Main()
	{
		var type1 = new { Value1 = "alma", Value2 = "k rte" };
		Console.WriteLine(type1.Value1); //alma
	}
}
```

35.2 *Kiválasztás*

A legegyszerûbb dolog amit egy listával tehetünk, hogy egy vagy több elemét valamilyen kritérium alapján kiválasztjuk. A következõ forráskódban épp ezt fogjuk tenni, egy egész számokat tartalmazó List<T> adatszerkezetbõl az összes számot lekérdezzük. Természetesen ennek így sok értelme nincsen, de szûrésrõl majd csak a következõ 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);
		}
	}
}
```

Elsõsorban 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 következõ sort:

```
var result = from number in list select number;
```

Egy LINQ lekérdezés a legegyszerûbb formájában a következõ sablonnal írható le:

eredmény = from azonosító in kifejezés select kifejezés

A lekérdezés elsõ 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 mûködik 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 egyszerûen 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 ismerõk számára furcsa lehet, hogy elsõként az adatforrást határozzuk meg, de ennek megvan az oka, mégpedig az, hogy ilyen módon a fejlesztõeszköz (a Visual Studio) támogatást illetve típusellenõrzést adhat a select utáni kifejezéshez (illetve a lekérdezés többi részéhez, de errõl késõbb).

A fenti kódban SQL szerû lekérdezést készítettünk (ún. Query Expression Format), de máshogyan is megfogalmazhattuk volna a mondanivalónkat. Emlékezzünk, minden LINQ To Objects mûvelet egy 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 következõket is:

```
var result1 = list.Select(
    delegate(int number)
    {
        return number;
    });

Func<int, int> selector = (x) => x;
var result2 = list.Select(selector);
```

A visszatérési érték típusa nem kell egyezzen a bemenõ paraméter típusával.

A két szintaxist keverhetjük is (Query Dot Format), de ez az olvashatóság rovására mehet, ezért nem ajánlott (leszámítva olyan eseteket amikor egy mûvelet csak az egyik formával használható), ha csak lehet ragaszkodjunk csak az egyikhez.
A következõ példában a Distinct metódust használjuk, amely a lekérdezés eredményébõl eltvolítja a duplikált elemeket. Õt 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 jegyze t ezután f mindkét válto zatot bemuta tja a for ráskódokban.

35.2. 1 *Pr ojekció*

Vegyük a követke zõ eg yszerû osztá lyhieraarchiát:

```
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 vásár ló hoz tarto zik egy Address objektum, amely a vevõ címét tárolja. Tegyük fel, hog y eg y olyan lekérde zést akarok írni, amely visszaadja a z összes ve võ ne vét, email címét és telefo nszámát. Ez nem egy bonyolult dolog , a kód a követke zõ 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, hog y a szükséges adatokon kívûl megkaptuk a z egész objektumot be leértve a cím pé ldányt is amelyre pedig semmi szükségünk nincse n. A megoldás, hog y az eredeti eredmé nyha lmazt leszûkítjük, úgy, hog y a lekérdezésben készítünk egy né vtelen osztá lyt, amely csak a kér t adatokat ta rtalmazza. Ezt projekciónak neve zzük. 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, ha nem 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, amelytõl egynél több eredményt várunk (tehát nem azok az operátorok amelyek pontosan egy elemmel térnek vissza – errõl késõbb 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 k ! ty kanyó, kend",
    "A szob ban lakik itt bent?",
    "L m, csak jó az isten, jót d,",
    "Hogy f lvitte a kend dolg t!"
};

var result = from line in poem
             let words = line.Split(' ')
                 from word in words
                     select word;
```

A let segítségével minden sorból egy újabb stringtömböt készítettünk, amelyeken egy belsõ lekérdezést futattunk le.

35.3 *Szûrés*

Nyílván nincs szükségünk mindig az összes elemre, ezért képesnek kell lennünk szûrni az eredménylistát. A legalapvetõbb ilyen mûvelet a where, amelyet a következõ sablonnal írhatunk le:

from azonosító in kifejezés *where kifejezés* select azonosító

Nézzünk egy egyszerû 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 rendelkezõ, bool visszatérési értékû 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 megfelelõ elemek nem a lekérdezés hívásakor kerülnek az eredménylistába, hanem akkor amikor ténylegesen felhasználjuk õket, ezt elhalasztott végrehajtásnak (deferred execution) nevezzük (ez alól kivételt jelent, ha a lekérdezés eredményén azonnal meghívjuk pl. a ToList metódust, ekkor az elemek szûrése azonnal megtörténik). A következõ 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("Szurés...");
                return x > 30;
            };

        Console.WriteLine("Lekérdezés el tt...");

        var result = from number in list
                     where predicate(number)
                     select number;
```

```
            Console.WriteLine("Lekérdezés ut n...");

            foreach (var item in result)
            {
                Console.WriteLine("{0}", item);
            }
        }
}
```

A kimenet pedig ez lesz:

Lekérdezés elõtt...
Lekérdezés után...
Szûrés...
Szûrés...
Szûrés...
56
Szûrés...
72
Szûrés...
34
Szûrés...
Szûrés...
89
Szûrés...

Jól lá tható, hog y a foreach ciklus váltotta ki a szûrõ elindulását.

A where két alakban létezik, a z elsõt már láttuk, most nézzük emg a másodikat is:

```
var result = list.Where((number, index) => index % 2 == 0);
```

Ez a vá lto zat két paramé tert kezel, ahol inde x a z ele m indexé t jelö li, természtese n nullától számo zva. A fe nti lekérdezés a páros inde xû elemeket választja ki.

35.4 *Rendezés*

A lekérdezések eredmé nyét kö nnyen rendezhetjük az orderby utasítással, a lekérde zés sablonja ebben a z esetben így a lakul:

from azonosító in kifejezés where kifejezés *orderby tulajdonság* ascending/descending select kifejezés

Az elsõ példába n eg yszerûe n re ndezzük egy s zámokból álló lista e lemeit:

```
var result1 = list.OrderBy(x => x); // n vekv sorrend

var result2 = from number in list
              orderby number ascending
                  select number; // szintén n vekv
```

```
var result3 = from number in list
              orderby number descending
              select number; // cs kken sorrend
```

A rendezés a gyorsrendezés algoritmusát használja.

Az elemeket több szinten is rendezhetjük, a következõ példában neveket fogunk sorrendbe rakni, mégpedig úgy, hogy az azonos kezdõbetûvel rendelkezõeket 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>()
        {
            "Istv n", "Iv n", "Judit", "Jol n", "Jen ", "Béla",
            "Bal zs", "Viktória", "Vazul", "T h t m", "Tam s"
        };

        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 elsõ a rendezés alapszabálya, míg második paraméterként megadhatunk egy tetszõleges ICompare r<T> interfészt megvalósító osztályt.

35.5 *Csoportosítás*

Lehetõségünk 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 elõzõ fejezetben elkészített program neveit és rendezzük õket csoportokba a név elsõ betûje szerint:

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

class Program
{
        static public void Main()
        {
                List<string> names = new List<string>()
                {
                        "Istv n", "Iv n", "Judit", "Jol n", "Jen ", "Béla",
                        "Bal zs", "Viktória", "Vazul", "T h t m", "Tam s"
                };

                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 következõ lesz:

```
B
-- Béla
-- Balázs
I
-- István
-- Iván
J
-- Judit
-- Jolán
-- Jenõ
T
-- Töhötöm
-- Tamás
V
-- Viktória
-- Vazul
```

A csoportosításho z meg kell adnunk egy kulcsot, ez lesz az OrderBy para métere. Az eredmény típ usa a kö vetkezõ lesz:

IEnumerable<IGro uping<TKey, TElement>>

Az IGro uping interfész tulajdonképpen maga is eg y IEnumerable<T> leszárma zo tt kiegészítve a rendezéshe z haszná lt kulcssal, vag yis lényegében egy lista a listában típ usú „adatszerkezetrõl" van szó.

35.5. 1 *Nu ll ér téke k kez elése*

Elõfordul, hogy olyan listán aka runk lekérde zést végrehajta ni, amelynek bizo nyos indexein null érték van. A Select képes kezelni ezeket az eseteket, egyszer ûen null értéket tesz az eredménylistába, de amikor rendezünk, akkor szükségünk va n az objektum adataira és ha null értéket akarunk vi zsgálni akkor gyo rsan ki véte lt kaphatunk. Haszná ljuk fel az e lõ zõ progra mok listáját, egészítsük ki né hány null értékkel és írjuk át a lekérde zést, hog y kezelje õket:

```
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 );
    }
}
```

35.5. 2 *Össz etett kulcso k*

Kulcs meg határo zásánál le hetõségünk van egynél több értéket kulcsként megadni, ekkor névte len osztályké nt kell azt definiálni. Használjuk fel a korábban megismert Custo mer illetve Address osztályokat, ezeket, ille tve a ho zzájuk tar tozó listákat a jegyze t mellé csatolt forráskódok közül a Data.cs file -ba n találja a z 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 köve tkezõképpen ír juk 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 íg y 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 vevõ -áru-rendelés adatokat. Memóriában lévõ gyûjtemények esetében ez viszonlag ritkán szükséges, de a LINQ To Objects támogatja ezt is.
A „join" – mûveletek azon az egyszerû feltételen alapulnak, hogy az összekapcsolandó objektum -listák rendelkeznek közös adattagokkal (relációs adatbázisoknál ezt elsõdleges 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 egyszerû megvalósítása. Mind a Customer mind a Product osztály rendelkezik egy ID nevû tulajdonsággal, amely segítségével egyértelmûen meg tudjuk különböztetni õket, ez lesz az elsõdleges 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 vevõ -termék párost igényel), ezek lesznek az idegen kulcsok. Írjunk egy lekérdezést, amely visszaadja minden vásárló rendelését.
Elsõként 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-szerû szintaxis olvashatóbb, ezért itt csak ezt írtuk meg. A lekérdezés nagyon egyszerû, elsõként csatoltuk az elsõdleges kulcsokat tartalmazó listát az idegen kulccsal rendelkezõ 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 belsõ lekérdezést is írtunk. A kiemenet a következõ lesz:

Istvan Reiter
-- Elosztó
Istvan Reiter
-- Papír zsebkendõ
József Fekete
-- Elektromos vízforraló

35.7 Outer join

Az elõzõ fejezetben azokat az elemeket választottuk ki, amelyek kapcsolódtak egymáshoz, de gyakran van szükségünk azokra is, amelyek éppenhogy 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 egyszerûen egy null elemet helyez el az eredménylistában az összes olyan elem helyén, amely nem szerepelne a „hagyományos" join által kapott lekérdezésben.
A következõ 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 lehetõséget ad listák konverziójára a következõ operátorok segítségével:

OfType és **Cast**: õk a „sima” IEnumerable interfészrõl konvertálnak generikus megfelelõikre, elsõdlegesen 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 kettõ közti különbséget a hibakezelés jelenti: az OfType egyszerûen figyelmen kívûl hagyja a konverziós hibákat és a nem megfelelõ 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 elsõ két elemét, de a harmadiknál 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 nevükbõl is látszik az IEnumerable<T> eredménylistát tömbbé vagy generikus gyûjteménnyé alakítják. A következõ példában a ToList metódust használjuk:

```
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 kiszûri a megfelelõ elemeket, vagyis itt érdemes figyelni a teljesítményre.
A ToArray metódus értelemszerûen tömbbé konvertálja a bemenõ adatokat. A ToDictionary és ToLookup metódusok hasonló feladatot látnak el abban az értelemben, hogy mindkettõ kulcs-érték párokkal operáló adatszerkezetet hoz létre. A különbség a duplikált kulcsok kezelésében van, míg a Dictionary<T, V> szerkezet ezeket nem engedélyezi (sõt kivételt dob), addig a ToLookup ILookup<T, V> szerkezetet ad vissza amelyben az azonos kulccsal rendelkezõ adatok listát alkotnak a listán belül, ezért õt kiválóan alkalmazhatjuk a join mûveletek során. A következõ példában ezt a metódust használjuk, hogy a vásárlókat megyeszerint 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 megfelelõ IEnumerable<T> típusra konvertálja vissza a megadott IEnumerable<T> interfészt megvalósító adatszerkezetet.

35.9 *„Element" operátorok*

A következõ operátorok megegyeznek abban, hogy egyetlen elemmel (vagy egy elõre meghatározott alapértékkel) térnek vissza az eredménylistából.

First/Last és **FirstOrDefault/LastOrDefault**: ezeknek a metódusoknak két változata van: a paraméter nélküli az elsõ/utolsó elemmel tér vissza a listából, míg a másik egy Func<T, bool> típusú metódust kap paraméternek és e szerint a szûrõ szerint választja ki az elsõ elemet. Amennyiben nincs a feltételnek megfelelõ elem a listában (vagy üres listáról beszélünk) akkor az elsõ két operátor kivételt dob, míg a másik kettõ az elemek típusának megfelelõ alapértelmezett nullértékkel tér vissza (pl. int típusú elemek listájánál ez nulla míg stringek 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 mindkettõ kivételt dob, ha a feltételnek több elem is megfelel.

ElementAt/ElementAtOrDefault: visszaadja a paraméterként átadott indexen található elemet. Az elsõ kivételt dob, ha az index nem megfelelõ a másik pedig a megfelelõ alapértelmezett értékét adja vissza.

DefaultIfEmpty : a megfelelõ alapér telme ze tté rtékkel tér vissza, ha egy lista nem tartalma z elemeket, õt használtuk korábba n a join mûve leteknél.

35.10 Halmaz *operátorok*

Elsõként né zzük a halma z operátorokat amelyek két lista közö tti ha lmazm ûveleteket tesznek lehetõvé.

Concat és **Union** : mindkettõ képes összefûzni két lista e lemeit, de a z utóbbi egy elemet csak egyszer tesz á t 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 e lsõ a zokat a z elemeket adja vissza amelyek mindkét listában szerepelnek, míg a második azoka t ame lyek csak a z elsõbe n:

```
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 mûveletet minden elemen és végeredményként egyetlen értéket adnak vissza (pl. elemek összege vagy átlagszámítás).

Count és **LongCount**: visszaadják az elemek számát egy listában. Alapérelmezés szerint az összes elemet számolják, de megadhatunk feltételt is. A két operátor közötti különbség az eredmény nagyságában van, a Count 32 bites egyész 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 legkissebb illetve legnagyobb elemét adják vissza. Mindkét operátornak megadhatunk egy szelektor kifejezést amelyet az összes elemre alkalmaznak és e szerint választják ki a megfelelõ 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 kettõ 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. Õk 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 lehetõvé teszi tetszõleges mûvelet elvégzését és a részeredmények „felhalmozását”. Az Aggregate három formában létezik, tetszés szerint megadhatunk neki egy kezdõértéket 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 elsõ esetben a Sum operátort szimuláltuk, ezt nem kell magyarázni. A második változatban maximumkeresést végeztünk, itt megadtunk egy kezdõértéket, amelynél biztosan van nagyobb elem a listában. Végül a harmadik mûveletnél kiszámoltuk a számok összegének egy százalékát (itt figyelni kell arra, hogy double típusként lássa a fordító a kezdõértéket, hiszen tizedestörtet 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 rendelkezõ 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 lehetõségét.

35.12.1 *Több szálúság vs. Párhuzamosság*

Amikor többszálú programokat készítünk alapvetõen nem törödünk a hardver lehetõségeivel, létrehozunk szálakat amelyek versengnek a processzoridõért. Ilyenkor értelemszerûen nem fogunk különösebb teljesítménynövekedést kapni, hiszen minden mûvelet ugyanannyi ideig tart, nem tudjuk õket közvetlenül szétosztani az - esetleges – több processzor között. Ezt a módszert pl. olyankor használjuk, amikor nem akarjuk, hogy a háttérben futó szálak megzavarják a „fõszál" kezelhetõségét (pl. ha egy böngészõben több oldalt is megnyitunk, nem akarjuk megvárni amíg mindegyik letöltõdik), 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 lehetõ leghatékonyabban tudja elosztani a munkát, anélkül, hogy bármely processzor üresjáratban álljon.

35.12.2 *Teljesítmény*

Nagyon könnyen azt gondolhatjuk, hogy a processzorok számának növelésével egyenes arányban nõ a teljesítmény, magyarul két processzor kétszer gyorsabb mint

egy. Ez a z állítás nem teljesen igaz (ezt késõbb a saját szem ünkkel is látni fogjuk), ezt pedig Gene Amdahl bizo nyította (Amda hl s Law) , miszeri nt:

Egy *párhuzamos program m aximum olyan gyors lehet mint a leghosszabb szekvenciális (tovább m ár nem párhuzamosítható) részegysége.*

Vegyünk példá ul egy programot amely 10 órán keresztûl fut nem párhuzamosa n. Ennek a program nak 9 órányi „része" párhuzamosítható, míg a maradék egy óra nem. Ha ezt a 9 órát párhuzamosítjuk akkor a tétel alapjá n a program nak íg y is minimum egy órás futásideje lesz.
Amdahl a kö vetkezõ képlete t találta ki:

Ahol P a progra m a zon része amely párhuzamosítható, míg (1 − P) a z ame lyik nem, N pedig a processzorok száma.
Nézzük meg, hogy mekkora a ma xim um teljesítmény amit a fenti esetbõl ki tud unk préselni. P ekkor 0,9 lesz (9 óra = 90% = 0,9), és a képlet (ké t processzo rt használva):

Könnye n kiszá molható, hog y a z eredmé ny 1/0,55 (1,81) lesz vagyis 81% -os teljesítménynövekedést érhetünk el két processzor bevezetésével. Vegyük észre, hogy a processzorok számá nak növe lésével P/N a nulláho z tart, vagyis kimondhatjuk, hog y mi nden pá rhuza mosítható fe ladat ma x im um 1/(1 −P) nagyságrendû teljesítmé nynövekedést eredmé nyezhet (feltéte le zve, hog y mi ndig annyi processzor áll rende lkezésünkre, hogy P/N a lehetõ legkö zelebb legyen nullá hoz: e z ne m fe ltétlenül jelent nag yon sokat, a példa esetében 5 processzor már 550% -os növekedést jelent, i nnen pedig egyre lassabban nõ az eredmény, mi vel ekkor P/N értéke már 0,18, hat processzor nál 0,15 és így to vább), te hát a fenti konkrét esetben a ma ximális teljesítmé ny a hag yomá nyos futásidõ tízszerese le het (1 / (1−0,9), vag yi s pontosa n az az egy ó ra amelyet a nem pár huzamosítható programrész használ fel.

35.12.3 PLINQ a gya ko r latban

Ebben a fejezetben összehasonlítjuk a hagyomá nyos és pár huzamos LINQ lekérde zések telejsítményé t. Elsõsorban szükségünk lesz valamilye n, megelelõe n nag y adatforrásra, amely je len esetbe n egy szö veges file lesz. A jegyzethez me llékelt forráskódok közö tt megtaláljuk a DataGen.cs ne vût, ame ly az elsõ paraméterként megadott fileba a második paraméterként megadott mennyiségû ve zetékné v - keresztné v -kor-foglalkoz ás -megye adatokat ír. A következõ példába n tízmillió személlyel fogunk dolgozni, tehát így futassuk a p rogramot:

DataGen.exe E:\Data .txt 10000000

Az elérési út persze tetszõleges. Most készítsük el a lekérdezést. Felhasználjuk, hogy a .NET 4.0 –ban a File.ReadLines metódus IEnumerable<string> -gel tér vissza, vagyis közvetlenül hathatunk rajta végre lekérdezést:

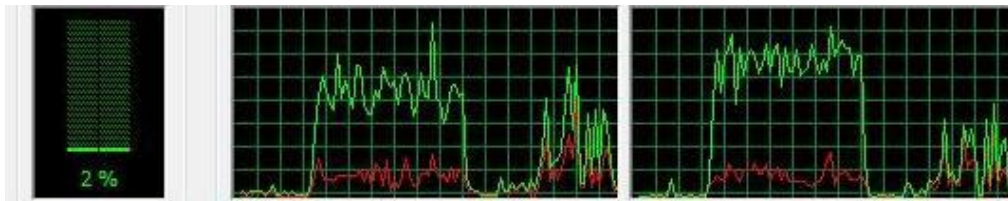
```
var lines = File.ReadLines(@"E:\Data.txt"); // System.IO kell

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 == "Istv n" &&
                       (age > 24 && age < 55) &&
                       job == "b rt n r"
                 select line;
```

Keressük az összes 24 és 55 év közötti István nevû börtönõrt. Elsõként szétvágunk minden egyes sort, majd a megfelelõ indexekrõl (az adatok sorrendjéért látogassuk meg a DataGen.cs –t) összeszedjük a szükséges adatokat. Az eredmény irassuk ki egy egyszerû foreach ciklussal:

```
foreach (var line in result)
{
    Console.WriteLine(line);
}
```

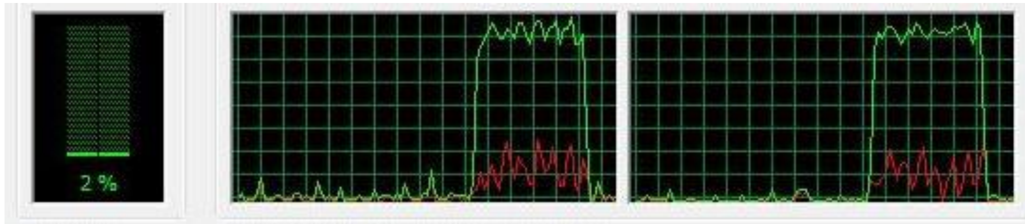
A fenti lekérdezés egyelõre nem párhuzamosított, nézzük meg, hogy mi történik:



Ez a kép a processzorhasználatot mutatja, két dolog világosan látszik: 1. a két processzor mag nem ugyanazt a teljesítményt adja le, 2. egyik sem teljesít maximumon.
Most írjuk át a lekérdezést párhuzamosra, ezt rendkívûl egyszerûen 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 == "Istv n" &&
                       (age > 24 && age < 55) &&
                       job == "b rt n r"
                 select line;
```

Lássuk az eredményt:



A kép önmagáért beszél, de a teljesítménykülönbség is, a tesztgépen átlagosan 25% volt a párhuzamos lekérdezés elõnye 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 jelöljük, ellenkezõ esetben nem fordul le:

```
var result = list1.AsParallel().Concat(list2.AsParallel());
```

Bár úgy tûnhet, hogy nagyon egyszerûen 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 szûrést vagy egyéb „drága” mûveletet szinte biztos, hogy szekvenciálisan fog végrehajtódni (és egyúttal a lekérdezés futásidejéhez hozzáadódik a vizsgálat ideje is). Természetesen az ellenõrzést végrehajtó algoritmus sem tévedhetetlen, ezért lehetõségünk van manuálisan kikényszeríteni a párhuzamosságot:

```
var result = select x from data.AsParallel()
                .WithExecutionMode(ParallelExecutionMode.ForceParallelism);
```

2. Ha az ítélet a párhuzamosságra nézve kedvezõ, akkor a következõ lépésben a feldolgozandó adatot a rendszer a rendelkezésre álló processzorok száma alapján elosztja. Ez egy meglehetõsen bonyolult téma, az adatforrástól függõen 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 lekérdezésbõl, az AutoBuffered periódikusan

advissza több elemet, míg a FullyBuffered csak akkor küldi vissza az eredményt, ha a lekérdezés teljesen kész van.

Láthatjuk, hogy komoly számítási kapacitást igényel a rendszertõl egy pár huzamos lekérdezés végrehajtása, épp ezért a nem megfelelõ 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 mennyiségû elemen sok vagy drága mûveletet végzünk el, mivel itt nyerhetünk a legtöbbet.

35.12.4 *Rendezés*

Nézzük a következõ 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 meglepõ 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 egyszerû, hiszen tudjuk, hogy a PLINQ részegységekre bontja az adatforrást, vagyis nem fogjuk sorban visszakapni az eredményeket (a fent mutatott eredmény nem lesz mindenkinél ugyanaz elvileg minden futásnál más sorrendet kapunk vissza).
Amennyiben számít az eredmény rendezettsége akkor vagy használnunk kell az orderby –t vagy az AsParallel mellettmeg 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 tûnik, de nagy különbség van közöttük. Ha lemérjük a végrehajtási idõt, akkor látni fogjuk, hogy az elsõ valamivel gyorsabban végzett mint a második, ennek pedig az az oka, hogy amíg az orderby mindig végrehajtja a rendezést addig az AsOrdered egyszerûen megõrzi az eredeti sorrendet és aszerint osztja szét az adatokat (nagy adatmennyiséggel a kettõ közötti sebességkülönbség is jelentõsen megnõ, érdemes néhányezer elemre is tesztelni).

Gyakran elõfordul, hogy az eredeti állapot fenntartása nem elõnyös számunkra, pl. kiválasztunk néhány elemet egy listából és ezek alapján akarunk egy join mûveletet végrehajtani. Ekkor nem célszerû fenntartani a sorrendet, használjuk az AsUnordered metódust amely minden érvényes rendezést érvénytelenít.

35.12.5 AsSequential

Az AsSequential metódus épp ellenkezõje az AsParallel – nek, vagyis szabályozhatjuk, hogy egy lekérdezés mely része legyen szekvenciális és melyik párhuzamos. A következõ példában megnézzük, hogy ez miért jó nekünk. A PLINQ egyelõre még nem teljesen kiforrott, annak a szabályai, hogy mely operátorok lesznek mindig szekvenciálisan kezelve a jövõben valószínûleg változni fognak, ezért a példaprogramot illik fenntartással kezelni:

```
var result = (from x in list.AsParallel() select x).Take(10);
```

Kiválasztjuk az eredménylistából az elsõ tíz elemet. A probléma az, hogy a Take operátor szekvenciálisan fut majd le függetlenül attól, hogy mennyire bonyolult a „belsõ" 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 belsõ párhuzamos mûveletsor végeztével visszaváltottunk szekvenciális módba, vagyis a Take nem fogja vissza a sebességet.

36 Visual Studio

A .NET Framewo rk programo zásáho z a z elsõ szám ú fejlesztõeszköz a Microso ft Visual Studio termékcsaládja. A „nagy” fizetõs Professional/Ultimate /stb... válto zatok melle tt a z Express változatok ingye nes és teljeskör û szolgáltatást nyújta nak szám unkra .

Ebben a f ejezetben a Visual Studio 2008 ve rzióval fogunk megismerked ni (a jeg yzet írásának pilla natában már lé tezik a 2010 változat is, de ez egyrészt még nem annyira elterjedt, másrészt nagy különbség nincs közöttük – aki az egyiket tud ja használni annak a másikkal sem lehe t prob lémája .

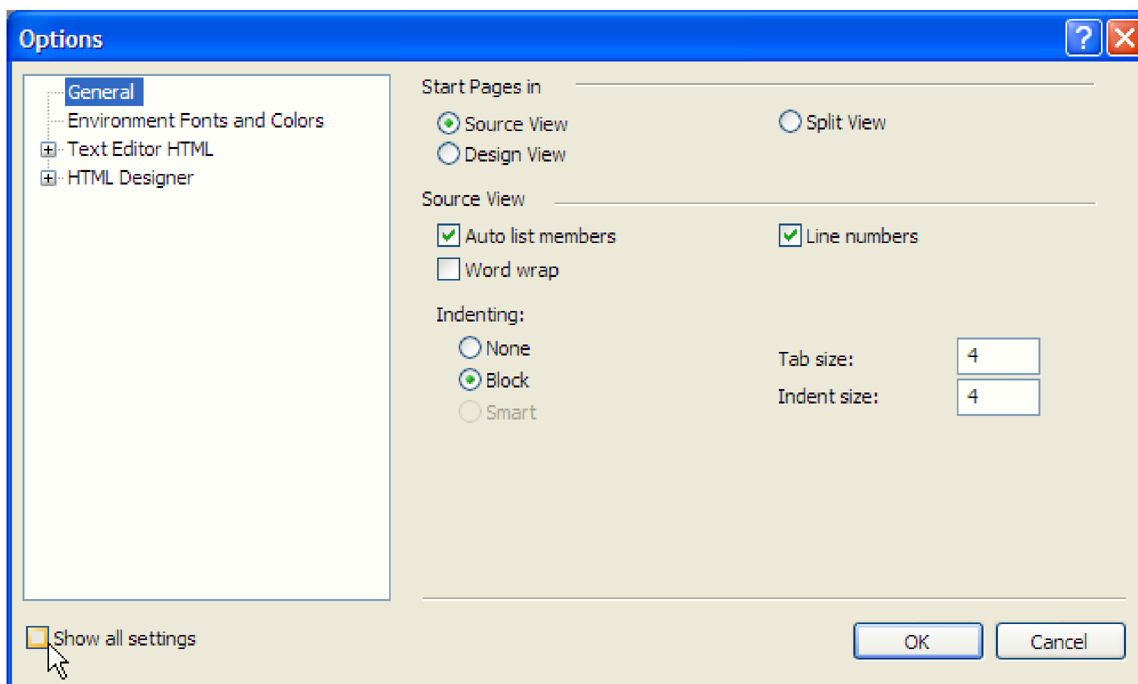
Érdemes telep íteni a Visual Studio 2008 szer vízcsomagját is, e z letölthetõ a Microsoft o ldaláró l.

36.1 *Az elsõ lépések*

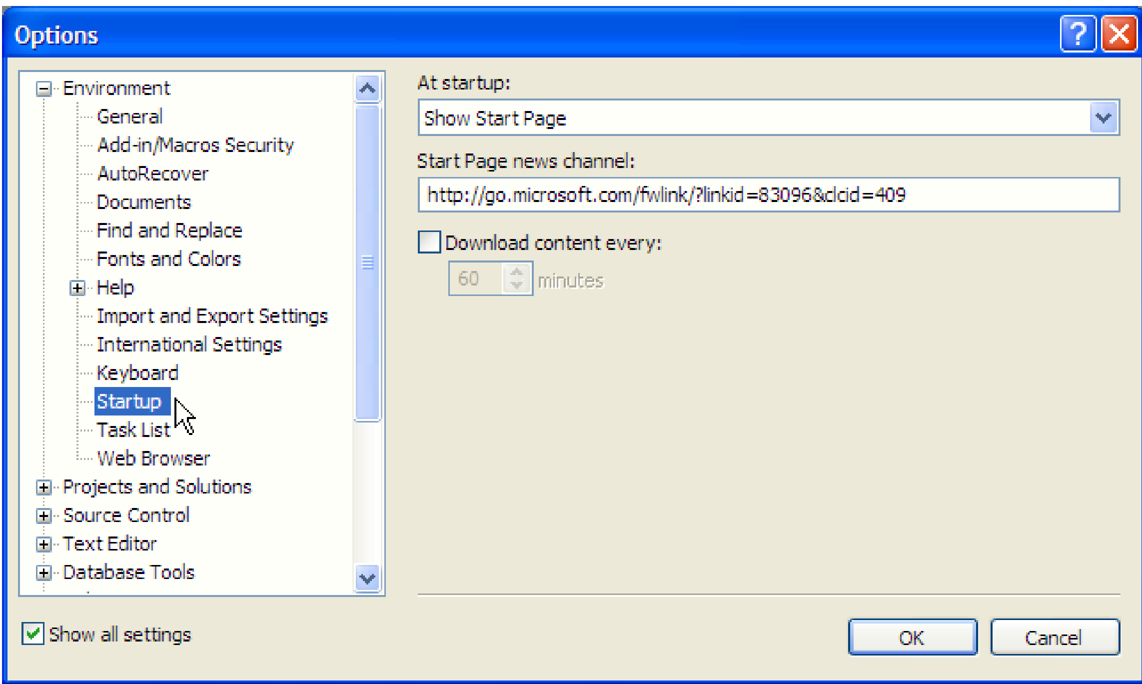
A VS 2008 telepítéséhez legalább Windows XP SP2 szükséges.

A VS elsõ indításakor megkérdezi, hogy melyik nyelvhez tartozó beállításokkal mûködjön, ez a mi esetünkba n a C# lesz. Ezutá n – alapbeállítás szerint – a kezdõlap jele nik meg amely a korábba n megnyito tt projectek me llett élõ inter netes kapcso lat esetén a fejlesztõi blogok illetve hírcsa tor nák legfrissebb bejeg yzéseit is megjeleníti.

Beállítha tjuk, hogy mi jele njen meg indításkor, e hhez vá lasszuk ki a Tools me nü Options pontját. Ekkor meg jelenik a kö vetkezõ ablak:

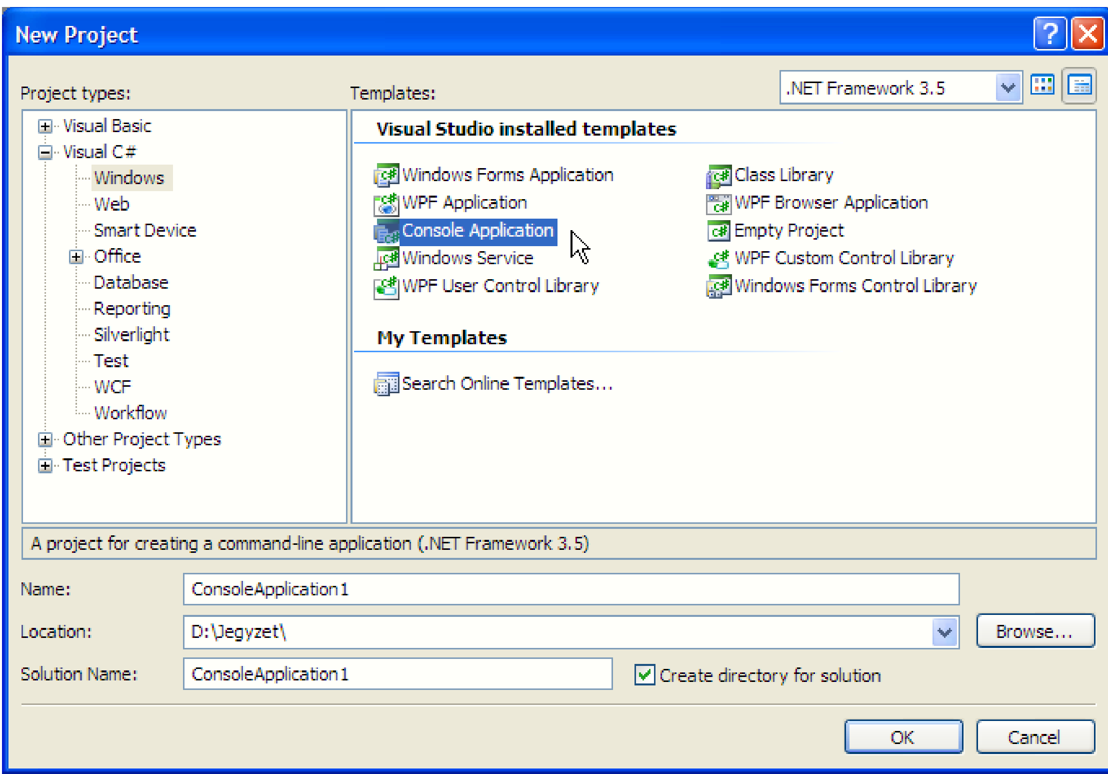


Amennyiben a Show all settings je lölõnég yzet üres, akkor jelö ljük be, majd a megfele nõ listából vá lasszuk ki a Startup – ot:



Itt beállítha tjuk, hogy mit szere tnénk lá tni indításkor.

Most készítsük el a z elsõ progra munkat, ehhez a File menü Ne w Project po ntját válasszu k ki:



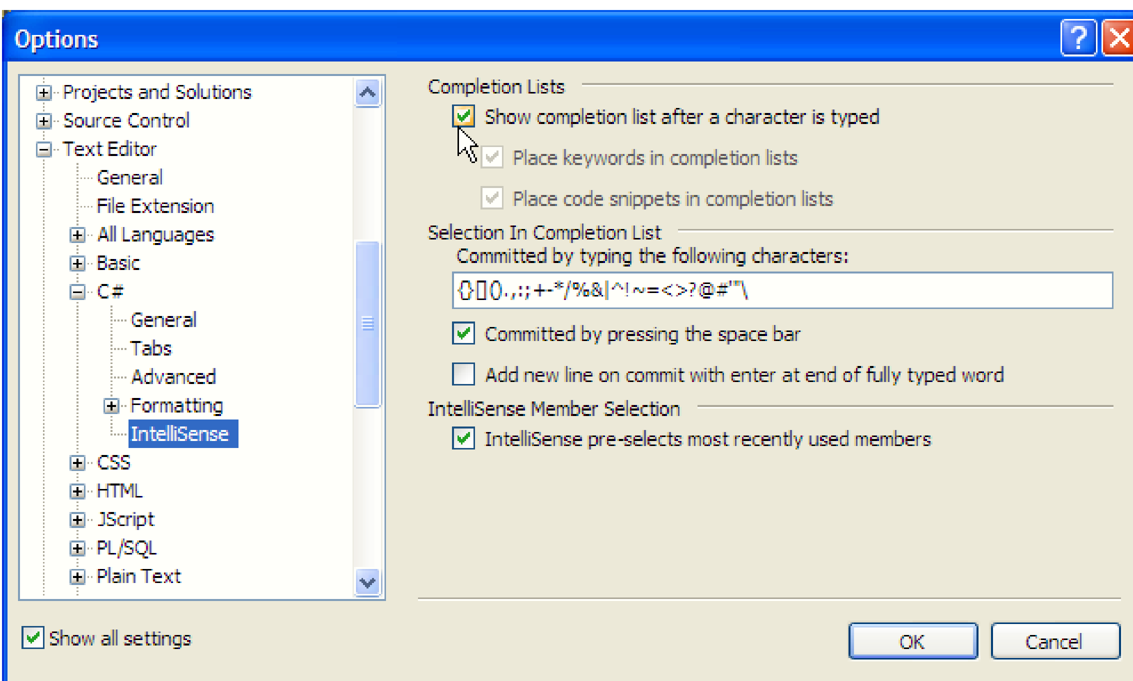
Itt a Visual C# fül alatt a Windows elem következik, ahol kiválaszthatjuk az elõre elkészített sablonok közül, hogy milyen típusú projectet akarunk készíteni, ez most egy Console Application lesz, a jegyzetben eddig is ilyeneket készítettünk.

A jobb fölsõ 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 megjelenõ forráskód valószínüleg ismerõs lesz, bár néhány (eddig ismeretlen) névteret már elõre beállított a VS.
Írjunk egy egyszerû 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 gépeléstõl megkímélve minket, ezt IntelliSense –nek hívják. Amennyiben nem mûködik az (valószínüleg) azt jelenti, hogy nincs bekapcsolva. A Tools menübõl válasszuk ki ismét az Options –t és azon belül pedig a Text Editor elemet. Ekkor a listából ki választhatjuk a C# nyelvet és azon belül pedig az IntelliSense menüt:



Ahogy a képen is látható a „Show completion list...” jelölõ négyzetet 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 tartalmazó ún. Solution –t írja le, ez lesz minden project gyökere (egy Solution tartalmazhat több projectet is). Ez a file egy egyszerû 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 (hex a editorral többé-kevésbé olvasható válik).

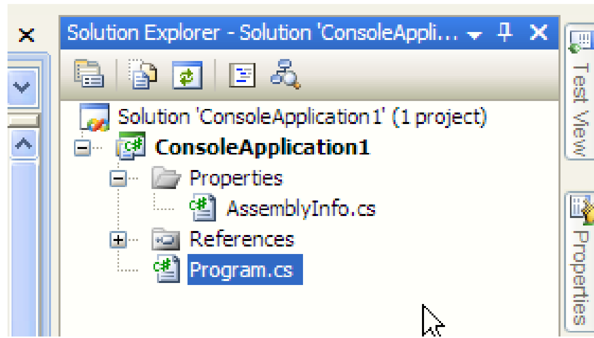
Most nyissuk meg a mappát is, megtalálhatjuk benne a projectet leíró .csproj file t, amely a project fordításának információit tárolja. A Properties mappában található AssemblyInfo.cs file amelyben pl. a program verziószáma, tulajdonosa, a gyártója illetve a COM interoperabilitásra vonatkozó információk vannak. Az obj mappa a fordítás során keletkezett „melléktermékét" tárolja, ezek lényegében átmeneti fileok a végleges program mûködéséhez nem szükségesek.
A bin mappában találjuk a lefordított végleges programunkat vagy a debug vagy a release mappában attól függõen, hogy hogyan fordítottuk (errõl késõbb bõvebben).
Ha megnézzük a mappa tartalmát láthatjuk, hogy bár egyetlen exe kiterjesztésû filera számítottunk találunk még három ismeretlen filet is. Ezek a Visual Studio számára hordoznak információt, segítik pl. a debuggolást (errõl is késõbb).

A Visual Studio 2008 hajlamos „eldobni" fileokat a projectekbõl, vagyis bár fizikailag jelen vannak, de a Solution Explorerben (és így fordításkor sem) nem szerepelnek. Ilyenkor két megoldás van: vagy kézzel újra felkell venni (jobb klikk a projecten, majd Add/Existing Item), vagy a csproj filet kell szerkeszteni. Ezt a problémát megelõzhetjü

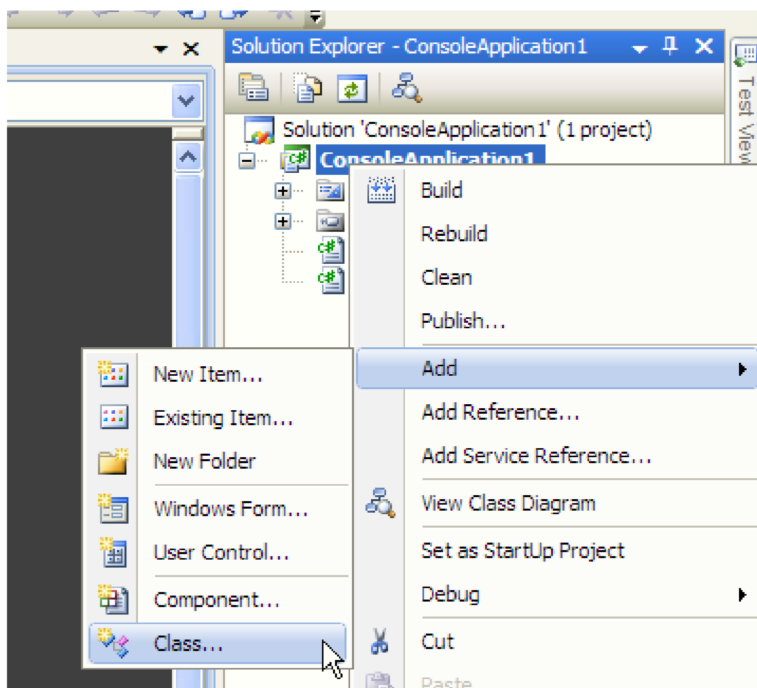
rendszeres biztonsági mentéssel (nagyobb programok esetén érdemes valamilyen verziókezelõ eszközt használni).

36.2 *Felület*

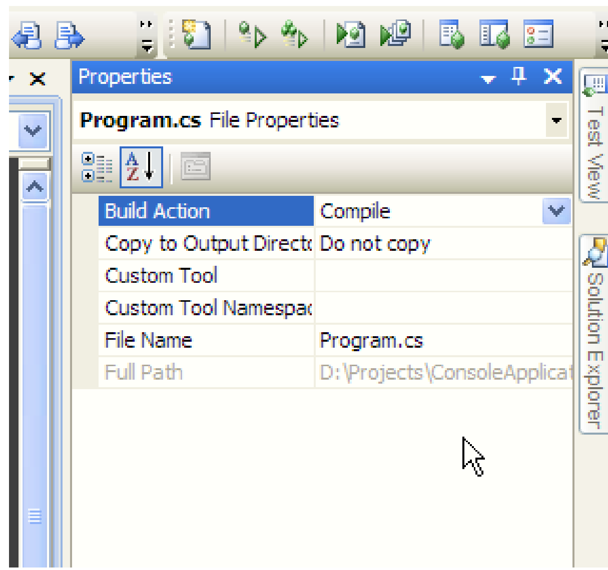
Térjünk most vissza a VS – hez és nézzük meg, hogy miként épül fel a kezelõfelület. Kezdjük a Solution Explorer –rel, amelyben az aktuális Solution elemeit kezelhetjük. Amennyiben nem látjuk ezt az ablakot (ha az Auto Hide be van kapcsolva, akkor csak egy „fülecske" jelenik meg az ablak szélén), akkor 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.

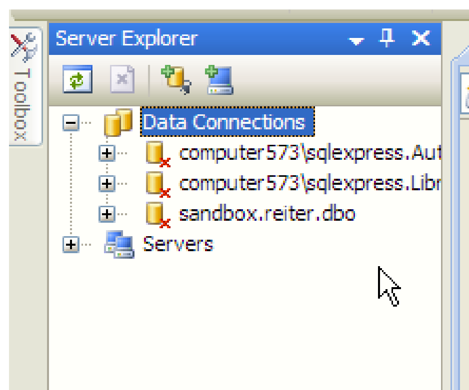


A Properties ablakban a kiválasztott elem tulajdonságait állíthatjuk. Többféle területen is felhasználjuk majd, többek között a grafikus felületû alkalmazások alkotóelemeinek beállításakor.



A képen a Program.cs file fordítási beállításait láthatjuk.

A Server Explorer segítségével (távoli) adatbázisokhoz kapcsolódhatunk, azokat szerkeszthetjük, illetve lekérdezéseket hajthatunk végre.



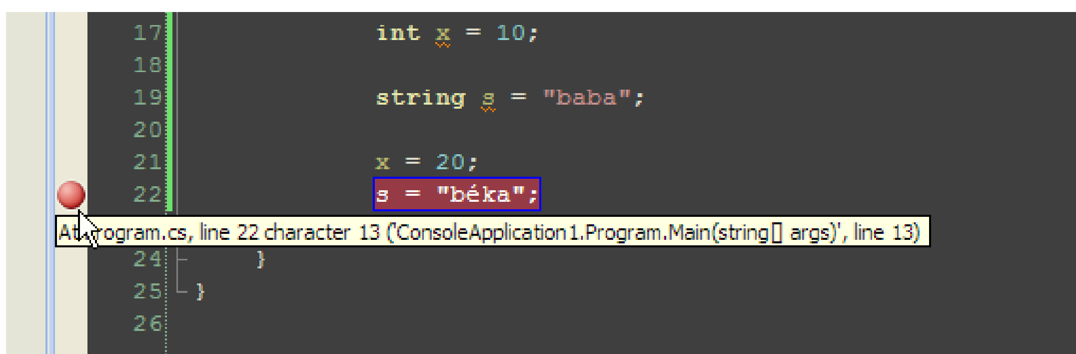
Ennek az ablaknak majd egy késõbbi fejezetben vesszük hasznát.

Végül a ToolBox ablak maradt, neki fõként grafikus felületû alkalmazások fejlesztésekor vesszük hasznát, ekkor innen tudjuk kiválasztani a komponenseket/vezérlõket.

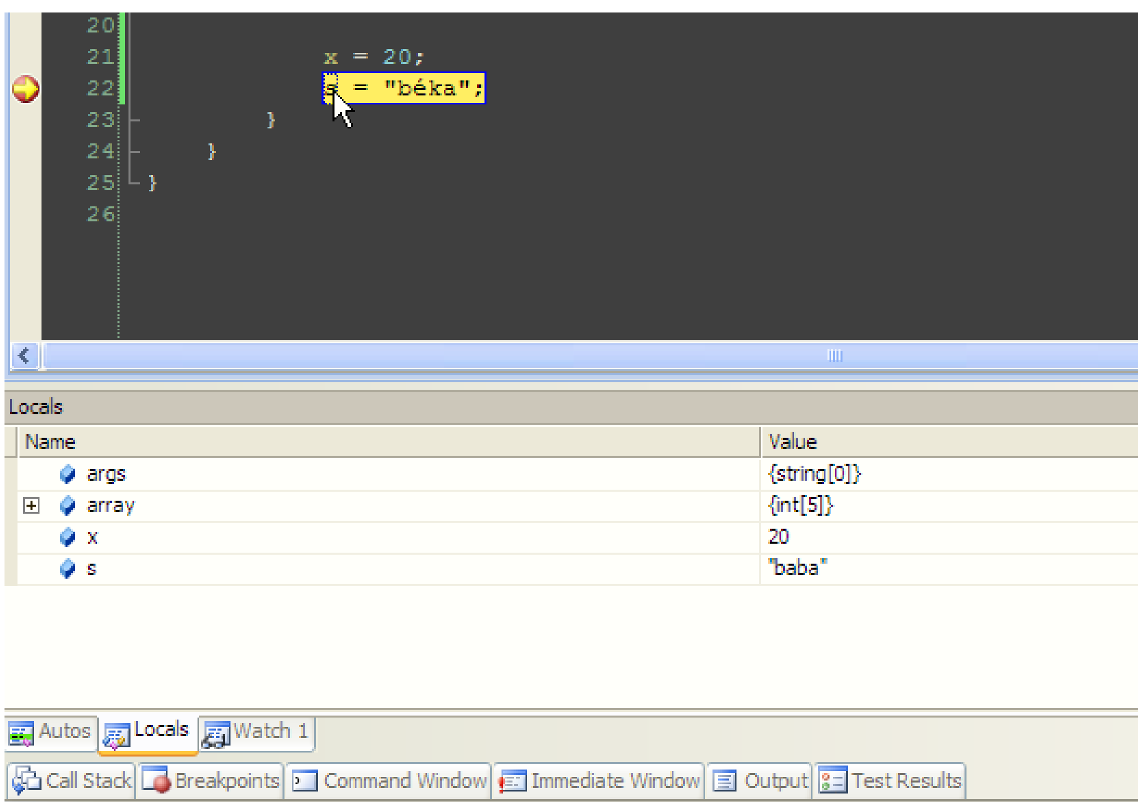
36.3 Debug

A Visual Studio segítségével a programok ja vítása, illetve a hibák okának felderítése sem okoz nagy nehézséget. Ebben a részben elsajátítjuk a hibakeresés alapjait: elsõként ismerkedjünk meg a breakpoint (vagy töréspont) fogalmával. Nyílván úgy

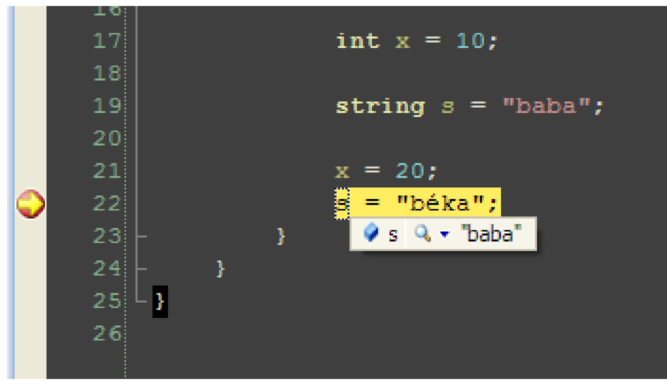
tud juk a legjobban felder íteni a progra mjaink hibái t, ha mûködés kö zben láthatjuk az objektumok állapotát. Erre olya n „fapados” módszereket is használhatunk, mi nt, hogy kiír juk a ko nzolra ezeket a z értékeket. Ennek pe rsze megva n a maga há trá nya, hisze n egyrészt be kell gépelni ezeket a pa rancsokat, másrészt beszennyezzük a forráskódot ami íg y nehe zebbe n olvasha tóvá válik. A breakpointok segítségével a program egy adott pontján le hetõségünk van megállíta ni annak futását, íg y kényelmesen meg vizsgálhatjuk az ob jektumokat.
A Visual Studio –ban a forráskód ablak bal oldalán lé võ üres sá v szolgá l töréspontok elhe lyezésére:



Ezutá n, ha futtatjuk a programot, akkor e zen a po nton a futása megáll. Fontos, hogy a törést tar talma zó sor már nem fog le futni, vagyis a fenti képen s értéke még „baba” marad.



A képen látható a Locals ablak, ame ly a változók á llapotát/értékét m utatja. Ugya nígy elérhetjük ezeket a z ér tékeket, ha a z egér mutatót kö zvetlenül a változó fölé visszük:

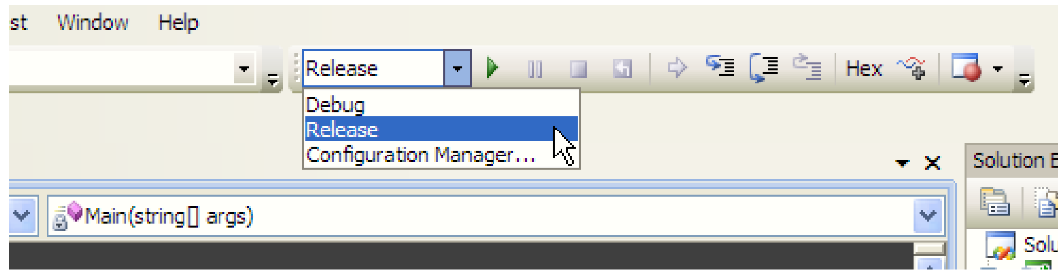


Mind itt, mind a Locals ablakba n módosíthatjuk az objektumokat.

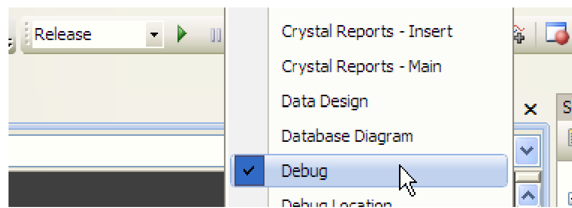
Az F11 gombba l utasításonként lépked hetünk debug módban.

36.4 *Debug és Release*

A Visual Studio kétféle módban – Debug és Release – tud fordítani. Az elõbbit tipikusan a fejlesztés sorá n használjuk, mive l – ahog yan a nevében is benne va n – a debuggolást seg íti, mive l a leford ított programot kiegészíti a szükséges információkkal. Ebbõl kifolyólag ez a mód a lassabb, akár 100% -os sebességveszteséget is kaphatunk, ha így próbá lunk számításigényes m ûveleteket végezni (épp e zért a Debug módot csakis a p rogram általá nos funkcióinak fejlesz tésekor haszná ljuk). Amikor a sebességet is tesztelni akatjuk, akkor a Release módot kell bevetnünk.
Alapértelmezés szerint a Visual Studio Toolbar –ján ki kell tudnunk vá lasztani, hog y mit szereté nk:



Ha mégsincs ott, akkor a ToolBar -on jobb gombbal kattintva je löljük 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 egyszerûbb lenne a dolgunk, ha az újrahasznosítandó forráskódot külön tudnánk választani a tényleges programtól, ezzel idõt 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 fejlesztõi platformtól függ, tehát egy COM DLL nem használható közvetlenûl egy .NET programban (de megoldható). Az viszont mindegyikre igaz, hogy a DLL állományok lényegében ugyanolyan szerkezettel rendelkeznek mint a futtatható állományok, leszámítva, hogy õket nem lehet futtatni.

Készítsük el az elsõ osztálykönyvtárunkat elõször parancssort használva, utána a Visual Studio segítségével. Az osztályunk (TestLib.cs a file neve) nagyon egyszerû lesz:

```
using System;

namespace TestNamespace
{
    public class TestClass
    {
        public string Text = "Hello DLL!";
    }
}
```

Ezúttal névteret is megadtunk, ez célszerû, mivel nem akarjuk, hogy más osztállyal ütközzön, vagyis egyértelmûen meg tudjuk mondani, hogy mire gondoltunk. Amire még figyelni kell, az az osztály elérhetõsége, hiszen most két különbözõ assembly–rõl van szó, vagyis ha kívûlrõl is el akarjuk érni ezt az osztályt akkor publikusként kell deklarálnunk. Fordítani a következõképpen tudjuk:

csc /target:library TestLib.cs

Ezután egy TestLib.dll nevû file–t kapunk.
A „fõprogramot” pedig így készítjük el:

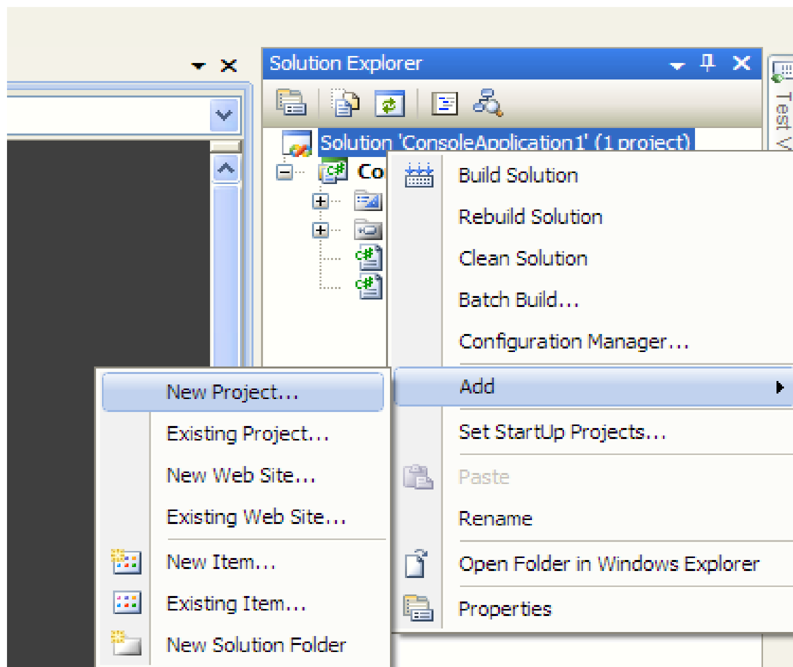
```
using System;
using TestNamespace;

class Program
{
    static public void Main()
    {
        TestClass tc = new TestClass();
        Console.WriteLine(tc.Text);
    }
}
```

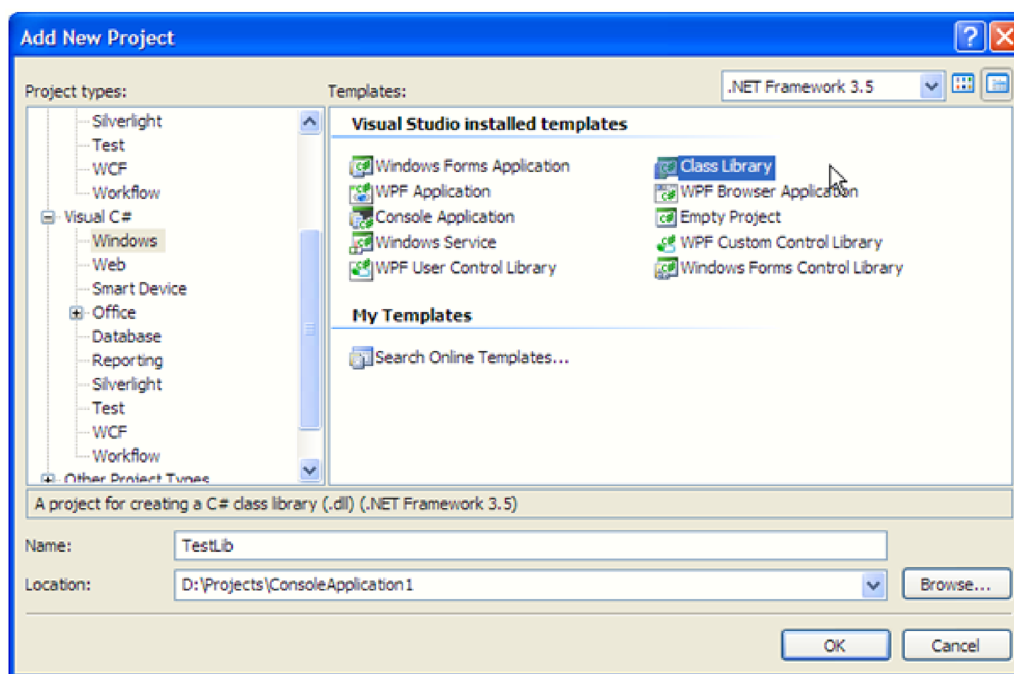
És így fordítjuk:

csc /reference:TestLib.dll main.cs

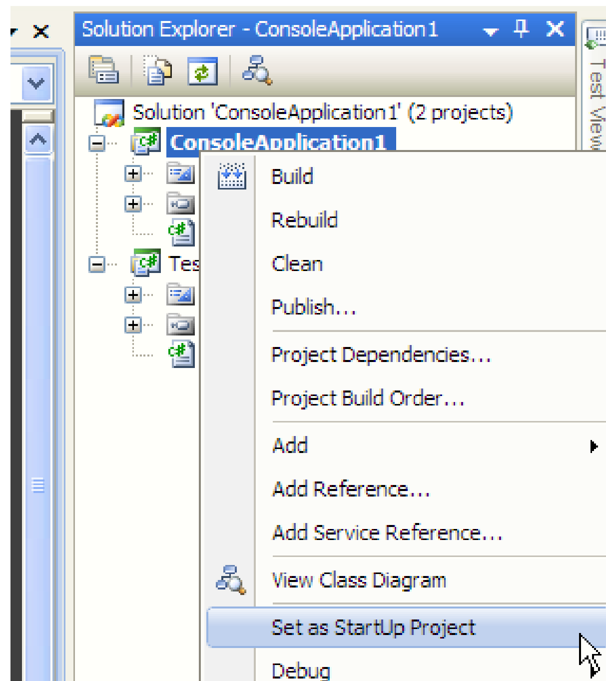
Most készítsük el ugyanezt a Visual Studio –val. Csináljunk elsüként egy Console Application –t mint az elõzõ fejezetben. majd kattintsunk jobb egérgombbal a Solution –ön (a Solution Explorerben) és az Add menübõl válasszuk ki a New Porjectet:



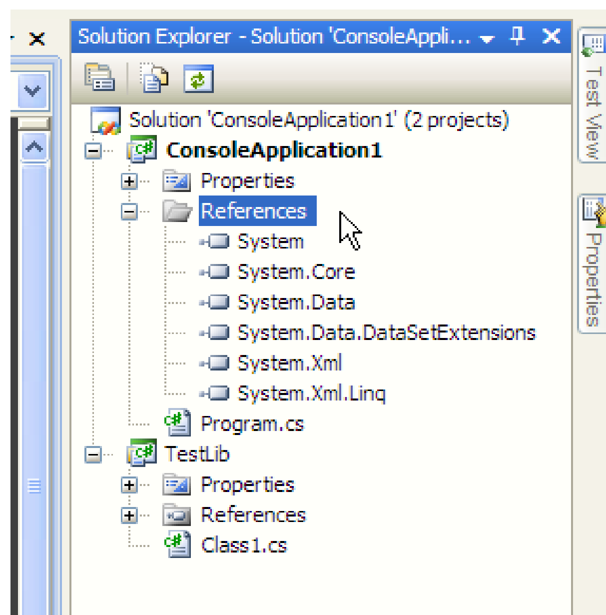
Majd a Class Library sablont



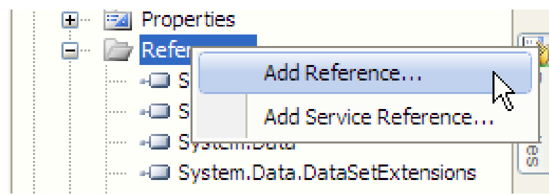
Az OK gombra ka ttintva a Solutio n –höz hozzáadódik az osztálykönyvtá r. Elõfordulhat, hog y ezután ne m tudjuk futtatni a programot, mivel egy „Pro ject with an Output Type...” kezdetû hibaüzenetet kap unk. Ez azt je lenti, hogy a frissen ho zzáadott osztálykönyvtár lett a Solution „fõprojectje” . ezt meg vá ltoztaha tjuk, ha a valóban futtatni kívá nt p rojecten jobb egérgombbal katti ntva kiválasztjuk a „Set as StartUp Project” pontot:



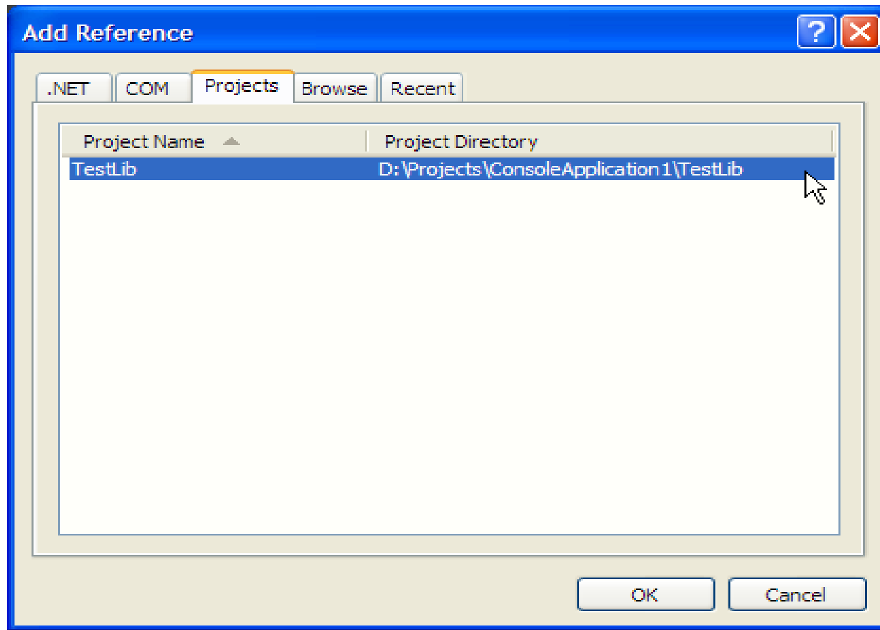
Készítsük el az osztálykönyvtárat és ford ítsuk le (jobb egérgomb a projecte n és Build). Most tér jünk vissza a fõprojecthez és nyissuk le a References fület, ez a már ho zzáadott osztá lykönyvtárakat tartalmazza:



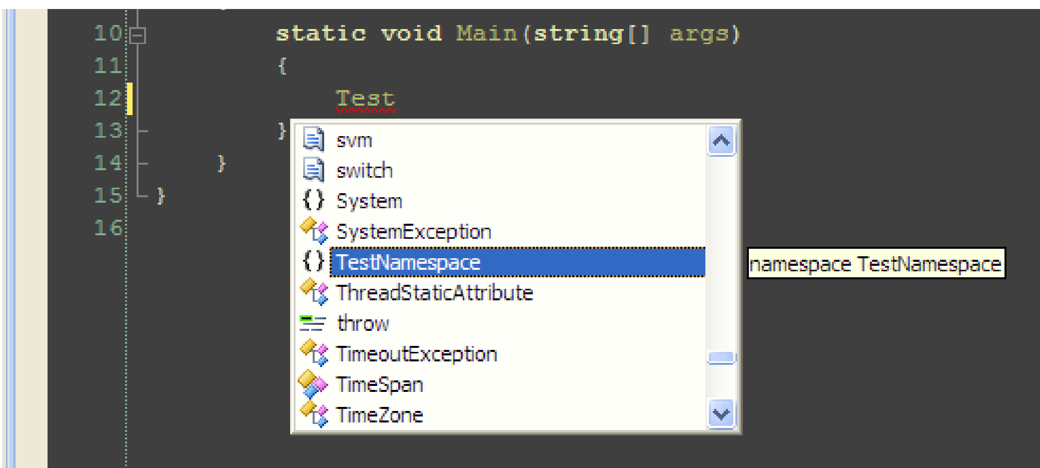
Kattintsunk jobb egérgombbal rajta és válasszuk a z Add Reference pontot:



A megjelenõ ablakba n kiválasztha tjuk, hog y a már beépített osztálykö nyvtárakból (.NET) akarunk válogatni vag y megkeressük a nekünk ke llõ file t (Browse), esetleg az egyik aktuális projectre van szükségünk (Pro jects). Nekünk te rmészetese n e z utóbbi kell:



Az Ok gombra kattintva a References alatt is megjele nik az új könyvtár . Ezutá n a szerkesztõablakban is láthatjuk az osztálykö nyvtár unk né vterét (ha mégsem akkor fordítsuk le az egész S olution –



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

namespace zh
{
    class Kör
    {
        protected double sugar;
        protected const double PI = 3.14;
        public Kör(double sugar1)
        {
            if (sugar1 > 0)
                sugar = sugar1;
            if (sugar < 1)
                sugar = 1;
        }
        public virtual double méret1()
        {
            return (2*sugar*PI);
        }
        public virtual double méret2()
        {
            return (sugar*sugar*PI);
        }
    }
    class Gömb: Kör
    {
        public Gömb(double sugar2) : base(sugar2)
        {

        }
        public override double méret1()
        {
            return (4 * sugar * sugar * PI);
        }                                            t) :
        public override double méret2()
        {
            return ((4 / 3) * sugar * sugar * sugar * PI);
        }

    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Kérem a sugár értékét");
            double    s=double.Parse(Console.ReadLine());
            Kör k = new Kör(s);
            Console.WriteLine("A kör kerülete: "+k.méret1());
            Console.WriteLine("A kör Terület: "+k.méret2());
            Gömb g = new Gömb(s);
            Console.WriteLine("A Gömb kerülete: "+g.méret1());
            Console.WriteLine("A Gömb Terület: "+g.méret2());
        }
    }
}
```

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

namespace BusaGaborZH
{

    class Számlista
    {

        LinkedList<int> számok = new LinkedList<int>();


        public Számlista()
        {
            for (int i = 1; i <= 10; i++)
                számok.AddLast(i);

        }

        public void kiír()
        {

            foreach (int be in számok)
            {


                Console.WriteLine("A szám: " + be + '\n');

            }
```

```
        }

        public bool töröl(int törl)
        {


            return számok.Remove(törl);
        }

        public void elemszám()
        {
          Console.WriteLine(számok.Count);

        }


    }


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

            Számlista sz = new Számlista();
            sz.kiír();
            Console.WriteLine("Melyik számot szeretné törölni?");
            int törl = int.Parse(Console.ReadLine());

            if (sz.töröl(törl) == false)
            {
                Console.WriteLine("Nincs ilyen szám a listában.");
            }
            else Console.WriteLine("A szám törölve.");
            sz.kiír();
            sz.elemszám();
        }
    }
}
```

kocka

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

namespace _11gyak
{
    class Kockadobás
    {
        private int[] dobasok;
        private int kockak_szama;
        Random r;
        public Kockadobás()
        {
            r=new Random();
            string s;
            Console.WriteLine("Mennyivel akarsz dobni he??????");
            try
            {
                s = Console.ReadLine();
                kockak_szama = int.Parse(s);
            }
            catch(FormatException e)
                {
                    Console.WriteLine("EZ nem szám!!!!" + e.Message);
                    Console.WriteLine("Mennyivel akarsz dobni he??????");
                    s = Console.ReadLine();
                    kockak_szama = int.Parse(s);

                }
            if (kockak_szama < 0)
            {
                kockak_szama = Math.Abs(kockak_szama);
            }
            else if (kockak_szama==0)
            {
                kockak_szama = 1;

            }
            dobasok = new int[10];
            r=new Random();
        }
        public void feltolt()
        {
            for (int i = 0; i < 10; i++)
            {
                dobasok[i] = r.Next(kockak_szama, 6 * kockak_szama + 1);
            }
        }
        public void kiir()
        {
```

```
            Console.WriteLine("Ennyi kockával dobott: ", kockak_szama);
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine("A(z) {0}.dobás:{1} ", i + 1, dobasok[i]);
            }
        }
        public double átlag()
        {
            double x=0;
            for (int i = 0; i < 10; i++)
            {
                x += dobasok[i];
            }
            double átlag = x / 10;
            return átlag;
        }
    }
}
```

program

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

namespace _11gyak
{
    class Program
    {
        static void Main(string[] args)
        {
            Kockadobás a = new Kockadobás();
            a.feltolt();
            a.kiir();
            Console.WriteLine("Az átlag:" + a.átlag());
        }
    }
}
```