

**Illés Zoltán**



**Programozás**

**C#**

**nyelven**



**ILLÉS ZOLTÁN**

# **Programozás C# nyelven**

**JEDLIK OKTATÁSI STÚDIÓ**  
**Budapest, 2005**



Minden jog fenntartva. Ezt a könyvet vagy annak részleteit a kiadó engedélye nélkül bármilyen formában vagy eszközzel reprodukálni, tárolni és közölni tilos.

A könyv készítése során a kiadó és a szerző a legnagyobb gondossággal járt el. Az esetleges hibákat és észrevételeket a [jos@jos.hu](mailto:jos@jos.hu) e-mail címen szívesen fogadjuk.

Szakmailag ellenőrizte: Heizlerné Bakonyi Viktória, ELTE

Anyanyelvi lektor: Gulyásné Felkai Ágnes

Borító: Sarkadi Csaba, 2004

© Kiadó: Jedlik Oktatási Stúdió Bt.  
1212 Budapest, Táncsics M. u. 92.  
Internet: <http://www.jos.hu>  
E-mail: [jos@jos.hu](mailto:jos@jos.hu)  
Felelős kiadó: a Jedlik Oktatási Stúdió Bt. cégvezetője

Nyomta: LAGrade Kft.  
Felelős vezető: Szutter Lénárd

**ISBN: 963 86514 1 5**  
**Raktári szám: JO 0331**

Bevezető .....	11
I. Alapismeretek .....	13
I.1. A nyelv története .....	13
I.2. A nyelv jellemzői .....	13
I.3. A .NET környezet áttekintése .....	15
I.4. A program szerkezete .....	16
I.5. Parancssori környezet használata .....	20
I.6. A környezet használata .....	21
I.7. Windows alkalmazások készítése .....	25
I.8. Feladatok .....	27
II. A C# nyelv alaptípusai .....	28
II.1. Változók definiálása .....	28
II.2. Állandók definiálása .....	29
II.3. Változók inicializálása .....	29
II.4. Elemi típusok .....	30
II.5. Felsorolás típus .....	35
II.6. Feladatok .....	37
III. Kifejezések, műveletek .....	38
III.1. Egyoperandusú operátorok .....	38
III.2. Kétooperandusú operátorok .....	39
III.3. Háromoperandusú operátor .....	42
III.4. Kétooperandusú értékadó operátorok .....	43
III.5. Feladatok .....	46
IV. Összetett adattípusok .....	47
IV.1. Tömbök .....	47
IV.2. Struktúra .....	51
IV.3. Feladatok .....	54
V. Utasítások .....	56
V.1. Összetett utasítás .....	56
V.2. Kifejezés utasítás .....	56
V.3. Elágazás utasítás .....	57
V.4. Ciklus utasítás .....	59
V.5. Ugró utasítások .....	63
V.6. Feladatok .....	67
VI. Függvények .....	68
VI.1. A függvények paraméterátadása .....	68
VI.2. A Main függvény paraméterei .....	72
VI.3. Függvények változó számú paraméterrel .....	74

---

VI.4. Függvénynevek átdefiniálása .....	75
VI.5. Delegáltak, események .....	77
VI.6. Feladatok .....	80
VII. Osztályok .....	81
VII.1. Osztályok definiálása .....	82
VII.2. Konstruktor- és destruktor függvények .....	84
VII.3. Konstans, csak olvasható mezők .....	92
VII.4. Tulajdonság, index függvény .....	94
VII.5. Osztályok függvényparaméterként .....	97
VII.6. Operátorok újradefiniálása .....	98
VII.7. Interface definiálása .....	104
VII.8. Osztályok öröklődése .....	107
VII.9. Végleges és absztrakt osztályok .....	110
VII.10. Virtuális tagfüggvények, függvényelfedés .....	112
VII.11. Feladatok .....	117
VIII. Kivételkezelés .....	118
VIII.1. Kivételkezelés használata .....	118
VIII.2. Saját hibatípus definiálása .....	121
VIII.3. Feladatok .....	125
IX. Input-Output .....	126
IX.1. Standard input-output .....	126
IX.2. Fájl input – output .....	129
IX.3. Feladatok .....	133
X. Párhuzamos programvégrehajtás .....	134
X.1. Szálak definiálása .....	134
X.2. Szálak vezérlése .....	135
X.3. Szálak szinkronizálása .....	138
X.4. Feladatok .....	143
XI. Attribútumok .....	144
XI.1. Attribútumok definiálása .....	145
XI.2. Attribútumok használata .....	146
XI.3. Könyvtári attribútumok .....	151
XI.4. Feladatok .....	152
XII. Szabványos adatmentés .....	153
XII.1. Könyvtári típusok szabványos mentése .....	153
XII.2. Saját típusok szabványos mentése .....	155
XII.3. Feladatok .....	158
XIII. Könyvtárak, távoli és web könyvtárak .....	159
XIII.1. Helyi könyvtárak használata .....	159

XIII.2. Távoli könyvtárhívás .....	162
XIII.3. Webkönyvtárak használata .....	168
XIII.4. Feladatok .....	173
XIV. Az előfeldolgozó .....	174
XIV.1. Szimbólumdefiníció használata .....	174
XIV.2. Területi jelölés.....	175
XIV.3. Feltételes fordítás .....	176
XIV.4. Hibaüzenet.....	176
XIV.5. Feladatok .....	176
XV. Nem felügyelt kód használata .....	177
XV.1. Nem felügyelt könyvtár elérése .....	177
XV.2. Mutatók használata .....	178
XV.3. Feladatok .....	179
XVI. Grafikus alkalmazások alapjai .....	180
XVI.1. Windows alkalmazások alapjai .....	180
XVI.2. Webes alkalmazások alapjai .....	185
XVI.3. Feladatok .....	189
Irodalomjegyzék .....	190

# Bevezető

Valamilyen programot megírni nem nehéz, de jó programot írni bizony nem egyszerű feladat.

Nehéz lenne megmondani, hogy melyik az aranyút, amely a jó program-készítés iskolájának tekinthető, de az bizonyosan állítható, hogy jó és hatékony programot csakis megfelelő fejlesztési környezetben tudunk készíteni.

Mai rohanó világunkban a gyors gazdasági, társadalmi változások mellett is szembeötlő, mennyire gyors, milyen dinamikus a változás az informatikában.

Nem telik el úgy hónap, hogy az informatika valamelyik területén be ne jelenjenek valamilyen újdonságot, legyen az hardver, vagy szoftver.

A szoftver fejlesztési környezeteket tekintve az utóbbi idők egyik legnagyobb és legtöbb újdonságot hozó egységcsomagját kapták meg a fejlesztők az új Microsoft .NET rendszerrel. Ebben a környezetben, bármilyen területre gondolunk, új technológiák, új lehetőségek segítik a fejlesztők munkáját (Common Language Runtime (clr), ASP.NET, stb.).

Ismerjük már: „Új műsorhoz új férfi kell...”.

Ezt az elvet alkalmazva a már klasszikusnak tekinthető Basic és C++ nyelvek mellett megjelent az új C# (ejtsd: angolosan „szí sárp”, magyarosan C kettőskereszt, Cisz...) programozási nyelv a .NET rendszer részeként.

A nyelv mottójának talán a következő képletet tekinthetjük:

*A Basic egyszerűsége + a C++ hatékonysága = C#!*

Ebben a könyvben a fejlesztési eszköztárnak ezt az alapját, a C# programozási nyelvet, annak lehetőségeit ismerhetik meg.

Terveim szerint egy következő könyvben a nyelv legfontosabb környezetbeli alkalmazási lehetőségeit részletesen tárgyalni fogjuk.

Remélem, hogy ez a könyv széles olvasótábornak fog hasznos információkat nyújtani. A programozással most ismerkedőknek egy új világ új eszközét mutatja meg, míg a programozásban jártas Olvasók talán ennek a könyvnek a segítségével megérik azt, hogy ez az a nyelv, amit kerestek.

Befejezésül remélem, hogy a magyarázatokhoz mellékelt példaprogramok jól szolgálják a tanulási folyamatot, és az anyag szerkesztése folytán nem kerültek bele „nemkívánatos elemek”.

Végül, de nem utolsósorban meg kell köszönnöm feleségemnek e könyv olvashatóságát segítő munkáját.

*Illés Zoltán*

## *I. Alapismeretek*

---



# *I. Alapismeretek*

## **I.1. A nyelv története**

A C# programozási nyelv a Microsoft új fejlesztési környezetével, a 2002-ben megjelent Visual Studio.NET programcsomaggal, annak részeként jelent meg.

Bár a nyelv hosszú múlttal nem rendelkezik, mindenképpen elődjének tekinthetjük a C++ nyelvet, a nyelv szintaktikáját, szerkezeti felépítését.

A C, C++ nyelvekben készült alkalmazások elkészítéséhez gyakran hosszabb fejlesztési időre volt szükség, mint más nyelvekben, például a MS Visual Basic esetén. A C, C++ nyelv komplexitása, a fejlesztések hosszabb időciklusa azt eredményezte, hogy a C, C++ programozók olyan nyelvet keressenek, amelyik jobb produktivitást eredményez, ugyanakkor megtartja a C, C++ hatékonyságát.

Erre a problémára az ideális megoldás a C# programozási nyelv. A C# egy modern objektumorientált nyelv, kényelmes és gyors lehetőséget biztosítva ahhoz, hogy .NET keretrendszer alá alkalmazásokat készítsünk, legyen az akár számolás, akár kommunikációs alkalmazás. A C# és a .NET keretrendszer alapja a *Common Language Infrastructure (CLI)*.

## **I.2. A nyelv jellemzői**

A C# az új .NET keretrendszer bázisnyelve. Tipikusan ehhez a keretrendszerhez tervezték, nem véletlen, hogy a szabványosítási azonosítójuk is csak egy számmal tér el egymástól. A nyelv teljesen komponens orientált. A fejlesztők számára a C++ hatékonyságát, és a Visual Basic fejlesztés gyorsaságát, egyszerűségét ötvözték ebben az eszközben.

A C# nyelv legfontosabb elemei a következők:

- Professzionális, Neumann-elvű. Nagy programok, akár rendszerprogramok írására alkalmas.
- A program több fordítási egységből – modulból – vagy fájlból áll. Minden egyes modulnak vagy fájlnak azonos a szerkezete.
- Egy sorba több utasítás is írható. Az utasítások lezáró jele a pontosvessző (;). Minden változót deklarálni kell. Változók, függvények elnevezésében az ékezetes karakterek használhatóak, a kis- és nagybetűk különböznek.

- A keretrendszer fordítási parancsa parancssorból is egyszerűen használható. (pl. `csc /out:alma.exe alma.cs`).
- Minden utasítás helyére összetett utasítás (blokk) írható. Az összetett utasítást a kapcsos zárójelek közé `{}` írt utasítások definiálják.
- Érték (alaptípusok, *enum*, *struct*, *value*) illetve referencia (*class*) típusú változók.
- Nincs mutatóhasználat; biztonságos a vektorhasználat.
- Boxing, unboxing. Minden típus öse az *object*, így például egy egész típust (*int*) csomagolhatunk objektumba (boxing) illetve vissza (unboxing).
- Függvények definíciói nem ágyazhatók egymásba, önmagát meghívhatja (rekurzió). Tehát függvénydefiníció esetén nincs blokkstruktúra. Blokkon belül statikus vagy dinamikus élettartamú változók deklarálhatók. Függvénytípuspolimorfizmus megengedett.
- Érték, referencia (*ref*) és output (*out*) függvényparaméterek.
- Kezdő paraméter-értékdadás, változó paraméterszámú függvény deklarálása.
- Delegáltak, események használata.
- Hierarchikus névterekben használt osztályok. Mivel minden osztály, ezért a „program”, a *Main* függvény *public static* hozzáférésű. Több osztály is tartalmazhat *Main* függvényt, de ilyen esetben a fordításkor meg kell mondani, hogy melyik osztálybeli *Main* függvény legyen az aktuális induló (*/main:osztálynév*).
- Új operátorok: *is* operátor egy objektum típusát ellenőrzi (*x is Lambda*), *as* operátor a bal oldali operandust jobb oldali típussá konvertálja (*Lambda l = x as Lambda;*). A hagyományos konverziós operátortól abban különbözik, hogy nem generál kivételt!
- Privát konstruktor (nem akarunk egy példányt se), statikus konstruktor (statikus mezők inicializálása, mindig példány konstruktor előtt hívódik meg, futási időben az osztálybetöltő hívja meg) használata.
- Nincs destruktork, helyette a keretrendszer szemétgyűjtési algoritmus van. Szükség esetén az osztály *Dispose* metódusa újradefiniálható.
- Egyszeres öröklés, interface-ek használata.
- Operátorok definiálásának lehetősége, *property*, *indexer* definiálás.
- Kivételkezelés megvalósítása.
- Párhuzamos végrehajtású szálak definiálhatósága.

### **I.3. A .NET környezet áttekintése**

A Visual Studio 6.0 fejlesztőrendszer átdolgozásaként 2002-ben jelent meg a Microsoft legfrissebb fejlesztőeszköze. Utalva a lényeges változtatásokra, a hálózati munka integrálására, a fejlesztőeszköz a Visual Studio.NET nevet kapta. Jelenleg az eszköz 2003-as frissítése a legutolsó verzió. A könyv szempontjából nincs lényeges különbség a két verzió között, így nem is térünk ki a különbségek bemutatására.

Az új eszköz a korábbi fejlesztőrendszerekkel ellentétben nem a hagyományosnak tekinthető ún. Win32 alapú, hanem a .NET környezet alatt futtatható alkalmazásokat készít. Ez azt jelenti, hogy az új eszközzel készült alkalmazások csak olyan operációs rendszer alatt futtathatók, melyek támogatják a .NET keretrendszert (.NET Framework). Alapértelmezésként a jelenlegi operációs rendszerek egyike sem támogatja ezt, viszont Windows 98 operációs rendszertől felfelé utólag feltelepíthető. A fordító a forráskódot nem natív, hanem egy köztes kódra fordítja le. Ezt a köztes kódot MSIL (Microsoft Intermediate Language) néven szokták említeni.

A Visual Studio.NET telepítése tehát a keretrendszer telepítésével kezdődik.

A .NET keretrendszer legfontosabb erényei:

- Webszabványokon alapuló (XML, HTML, SOAP).
- Univerzális alkalmazási modellt használ, azaz egy .NET osztály, szolgáltatás tetszőleges .NET kompatibilis nyelvből használható. A .NET kompatibilitást a Common Language Specification (CLS) definiálja.
- Minden nyelvből ugyanazok a típusok használhatók (Common Type System)
- Minden .NET osztály a fejlesztők rendelkezésére áll.

A keretrendszer a fejlesztőeszköz számára fordítási idejű szolgáltatásokat végez, míg az így lefordított alkalmazásoknak futási idejű környezetet biztosít (Common Language Runtime). A keretrendszer biztosítja a fentiek mellett az alkalmazások számára a már nem használt objektumok memóriabeli felszabadítását (Garbage Collection).

A keretrendszer osztálykönyvtára az alkalmazások típusa alapján több csoportba osztható:

- ADO.NET, adatbázis alkalmazások új generációs könyvtára.
- ASP.NET, webes alkalmazások (Web Forms) készítésének könyvtára.
- XML Web Service, interneten keresztül elérhető könyvtár.
- Windows alapú felhasználói (Windows Forms, Console Application), alkalmazói könyvtár.

Az osztálykönyvtárak használatához egy fejlesztő nyelvre van szükség. A Visual Studio.NET alapértelmezésben három nyelvet biztosít a fejlesztők számára. A Visual Basic és a Visual C++ nyelveket, mint az előd keretrendszerből megmaradt bázisnyelveket, és egy új nyelvet, amit a .NET keretrendszerhez fejlesztettek ki, a Visual C#-ot. Ma már a Java utódnyelv, a J# is a Visual Studio.NET 2003 fejlesztőrendszer része. Az új C# nyelvet szabványosították, ami a széleskörű elterjedésnek fontos feltétele. A C# nyelv szabványosított dokumentációjához ECMA-334 kód alatt férhetünk hozzá.

A .NET keretrendszer lényege, hogy a közös nyelvi definíciók már szabványosítottak. Ebben a szabványosításban a Microsoft mellett az informatikában érdekelt legnagyobb szervezetek is (HP, Intel, IBM, Sun, Fujitsu, Netscape) részt vettek. Ez ECMA-335-ös azonosítóval, mint a közös nyelvi infrastruktúra vagy eredeti nevén Common Language Infrastructure (CLI) nemzetközi szabványává vált.

### I.4. A program szerkezete

Egy C# program tetszőleges számú fordítási egységből (modulból) állhat. Szokás ezen modulok vagy fájlok összességét projektnek nevezni.

A program ezen modulokban definiált osztályok összessége.

Egy fájl tartalmazza az egyes modulok közti hivatkozásokat (*using*), osztálytípus-, változó- és függvénydeklarációkat.

Egy modul egy tetszőleges névtér része lehet. A névtér (*namespace*) az a logikai egység, amiben az azonosítónknak egyedinek kell lennie. Nem kötelező a névtér definiálása, ebben az esetben az ún. név nélküli névtér része lesz az adott kód.

Névtér szerkezete:

```
namespace új_névtérnév
{
    class új_osztálynév
    {
        Típusdefiníció;
        Függvénydefiníció;
    }
    ...
}
```

Függvények definíciójának formája:

```

visszaadott_típus név(argumentumlista, ha van)
{
    változó definíciók,
    deklarációk
    és utasítások
}

```

Az osztályok egyikében kell egy *Main* nevű függvénynek szerepelnie, amely futtatáskor az operációs rendszertől a vezérlést megkapja. A nyelv megengedi, hogy több típusban (osztályban) is definiáljunk ilyen függvényt. Ebben az esetben fordításkor kell megmondani, hogy melyik típus *Main* függvénye legyen a főprogram.

A programban használt neveknek betűvel vagy `_` jellel kell kezdődniük, majd a második karaktertől tetszőleges betű és szám kombinációja állhat. A nyelvben a kis- és nagybetűk különbözőek, így például a következő két név sem azonos:

```

alma
aLma

```

Az azonosítónevek hossza általában 32 karakter lehet, de sok rendszerben az igényeknek megfelelően lehet ezen változtatni.

A .NET keretrendszer 16 bites unikód karakterábrázolást használ, amiben a magyar ékezetes karakterek is benne vannak. A nyelvi fordító ezeket az ékezetes betűket is megengedi, így az alábbi név is megengedett:

```

körte

```

Azonosítók használatára nem megengedettek a C#-ban a következő kulcsszavak vagy védett azonosítók:

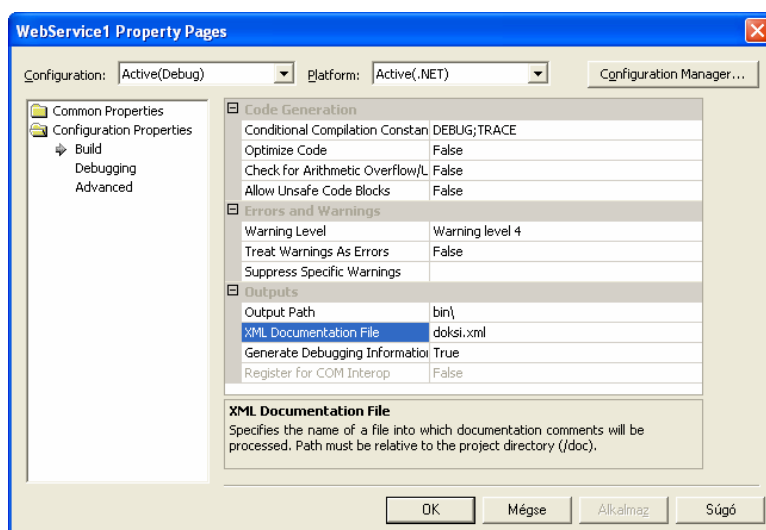
abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public

readonly	ref	return	sbyte	sealed
short	stackalloc	static	string	struct
switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe
ushort	using	virtual	void	while

Egy forrásállomány szerkesztése során magyarázatokat is elhelyezhetünk a `/*` és `*/` jelek között. Az ilyen megjegyzés több soron át is tarthat. Egy soron belül a `//` jel után írhatunk magyarázatot.

A mai fejlesztőkörnyezetekben egyáltalán nem ritka, hogy valamilyen speciális megjegyzést arra használjanak fel, hogy ennek a programszövegből történő kigyűjtésével a forrásállománynak vagy magának a programnak egy dokumentációját kapják. Ezt a kigyűjtést a fordító végzi el.

A C# fordítónak ha a `/doc:fájlnev` formában adunk egy fordítási paramétert, akkor `///` karakterek utáni információkat XML fájlba (a megadott névvel) kigyűjti. Ha nem parancssori fordítót használunk, ami a Visual Studio.NET környezet használatakor gyakran (majdnem mindig) előfordul, akkor ezt a beállítást a projekt Tulajdonságok dialógusablakában az *XML Documentation File* paraméter értékeként állíthatjuk be.



1. ábra

Ha a keretrendszerben egy változó vagy függvény definíciója elé beszurjuk a `///` karaktereket, akkor azt a szövegszerkesztő automatikusan kiegészíti a következő formában:

Példa:

```

    /// <summary>
    /// ez egy változó
    /// </summary>
    int i;
    /// <summary>
    /// ez meg egy függvény
    /// </summary>
    /// <param name="i"></param>
    public void szamol(int i)
    {
        ...
    }

```

A `/** .... */` forma is megengedett, szintén dokumentációs célokra. Mivel az előző módszer nem gyűjti ki az ilyen formájú megjegyzést, nem is használják gyakran.

Ezek alapján nézzük meg a szokásosnak nevezhető bevezető programunk forráskódját.

Példa:

```

using System;
class foci
{
    static void Main()
    {
        Console.WriteLine("Hajrá Fradi!");
    }
}

```

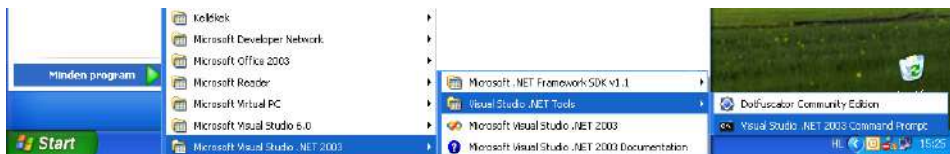
Programunk rövid magyarázataként annyit mondhatunk, hogy a forráskódunk a legegyszerűbb esetben egyetlen állományból áll, amiben nem definiálunk névteret, hanem csak egy *foci* osztályt. (Valamilyen típust kell definiálnunk!) Ebben az osztályban egyetlen függvényt definiálunk, a programot jelentő *Main* függvényt. Mivel egyetlen osztálytípus egyetlen példánya sem létezik a létrehozásig (a definíció még nem létrehozás!), ezért ahhoz, hogy a függvényünk példány nélkül is létezzen, a *static* jelzővel kell a hozzáférési szintet meghatározni. A C stílusú nyelvek hagyományaként a C# nyelvnek sem részei a beolvasó és kiíró utasítások, így könyvtári szolgáltatásokra kell hagyatkoznunk, ami a *System* névtér része, és ennek a névtérnek a *Console* osztálya biztosítja a klasszikus képernyőre írás, *Console.WriteLine()* és billentyűzetolvasás, *Console.ReadLine()* feladatát.

## I.5. Parancssori környezet használata

A tetszőleges szövegszerkesztővel megírt forráskódot egy *alma.cs* (a nyelv a *cs* kiterjesztést szereti...) fájlba menthetjük, majd az alábbi utasítással fordíthatjuk le:

```
csc alma.cs
```

A fordítás eredménye egy *alma.exe* állomány lesz, amit futtatva a képernyő következő sorában láthatjuk kedvenc buzdító mondatunkat. Természetesen akkor kapunk hibamentes fordítási eredményt, ha a fordítónk és a könyvtárak használatához szükséges útvonali, környezeti változó beállítások helyesek. Ezt a *vcvars32.bat* állomány elvégzi. Ilyen parancssori környezetet legkönnyebben a Visual Studio.NET programcsoporthoz tartozó segédeszközök közül tudunk elindítani. (Ekkor lefut a *vcvars32.bat*, nem kell nekünk kézzel indítani!)



2. ábra

A parancssori fordítónak a */?* paraméterrel történő futtatása, mint egy segítség funkció, megadja a fordító fontosabb kapcsolóit.

Ezek közül a legfontosabbak a következők:

- /t:exe*                      alapértelmezés, exe állományt fordít.
- /t:library*                a fordítandó állomány könyvtár (dll) lesz.
- /out:név*                  a fordítás eredményének nevét határozhatjuk meg, alapértelmezésben ez a név megegyezik a fordított fájl nevével.
- /r:valami.dll*            egy könyvtár felhasználása fordításkor, ha több könyvtári állományt kell hozzáfordítani, akkor az állománynevek közé vesszőt kell tenni.
- /main:osztálynév*        a nyelv megengedi, hogy több osztály is tartalmazzon *Main* függvényt, ekkor ezzel a kapcsolóval mondhatjuk meg, hogy a sok *Main* függvény közül melyik legyen a „főprogram”.



A keletkezett exe állományról el kell mondani, hogy annak futtatásához nem elegendő egy hagyományos 32 bites Microsoft operációs rendszer, gyakran ezt úgy fogalmazzák meg, hogy a keletkezett program nem natív kódot tartalmaz, hanem szükséges az operációs rendszerhez hozzátelepíteni a .NET keretrendszert! Ezt természetesen a Visual Studio.NET installálása elvégzi, így a saját gépünkön nem tűnik fel annak hiánya sem. Az irodalom ezt az exe kódot gyakran menedzselt (managed) kódnak nevezi.

Általában elmondható, hogy az új fejlesztőkörnyezet minden egyes nyelvi eszköze olyan kódot fordít, aminek szüksége van erre a keretrendszerre. Természetesen mint az élet minden területén, úgy itt is vannak kivételek. Ilyen kivétel például a C++ nyelv, ahol alapértelmezés szerint meg kell mondani, hogy a programunkat menedzselt vagy natív kódra fordítsa-e a fordítónk. Natív kód alatt értjük azt a fordított eredményt, ami processzor szintű utasításokat tartalmaz. Ebben a fejlesztőkörnyezetben ezt a kódot a menedzselt ellentétéként, nem menedzselt (unmanaged) kódnak is nevezik.

## I.6. A környezet használata

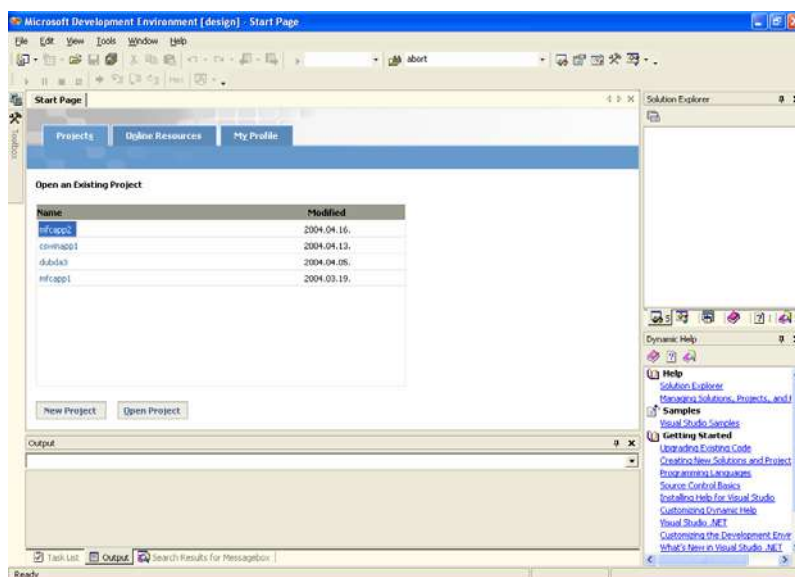
Mielőtt a következő részben a nyelv részletes jellemzőinek ismertetését kezdenénk, nézzük, hogy a felinstallált Visual Studio.NET környezet segítségével miképpen tudjuk az eddigi egy, illetve a későbbi példaprogramokat kipróbálni.

Természetesen nincs szükségünk egy tetszőleges szövegszerkesztő (pl: emacs ) használatára, hiszen a fejlesztőkörnyezet ad egy jól használható belső szövegszerkesztőt, és nincs szükség a parancssori fordítás parancs kiadására sem, hiszen ez a parancs egy menüpontra van rádefiniálva.

A fejlesztőkörnyezet installációja után a Start \ *Programok* menüponton keresztül indíthatjuk el a Visual Studio.NET környezetet, ami a 3. ábrán látható jellemző ablakkal jelenik meg.

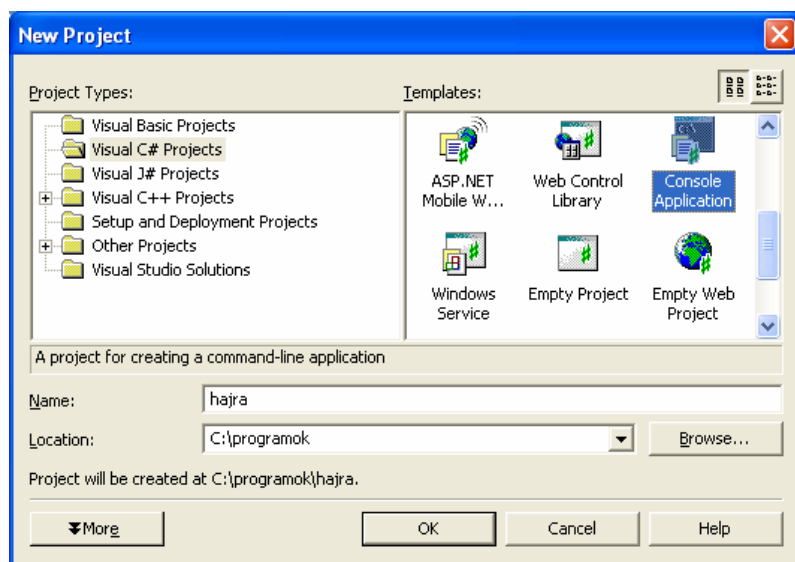
A jelenlegi fejlesztőkörnyezetekben minden program valójában egy projekt-fedőnév alatt készül el. Egy vagy több projekt felügyelete a *Solution Explorer* ablakban végezhető. A fő munkaterületen egy induló *Start Page* látható, amiben a legutóbbi projektek szerepelnek, illetve új vagy korábbi létező projektet nyithatunk meg. Természetesen a *File* menüponton keresztül is elvégezhetők ezek a feladatok.

## I. Alapismeretek



3. ábra

A *New Project* menüpont kiválasztása után, ahogy az az alábbi ablakban is látszik, három fő kérdésre kell válaszolnunk:



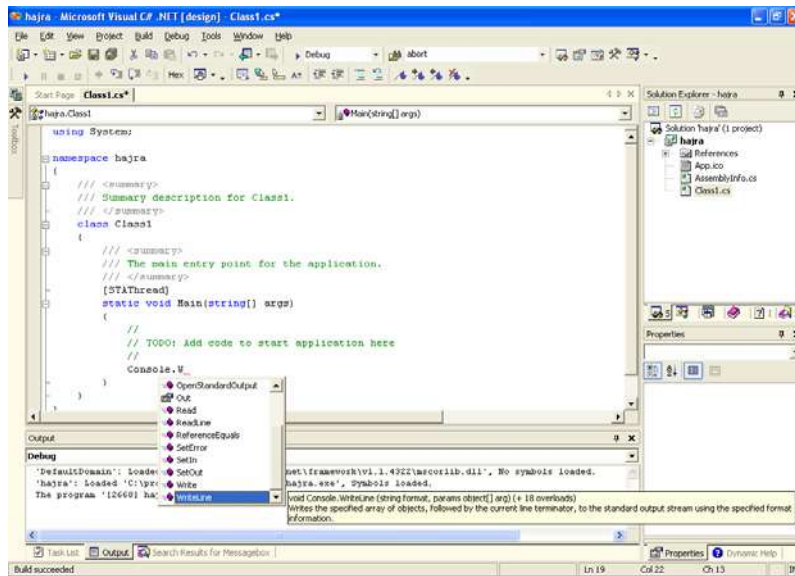
4. ábra

1. Ki kell választani a programozási nyelvet.
2. Az adott nyelvhez meg kell mondani, hogy milyen jellegű alkalmazást szeretnénk készíteni.
3. Meg kell adni a munkakönyvtárat és a projekt nevét.

Ebben a könyvben a nyelvet illetően mindig a *Visual C# Project* lehetőséget választjuk, amit a nyitott mappajel is mutat.

A *Templates* ablakban választhatjuk ki az alkalmazás jellegét. Jelen esetben, illetve a következő rész példáiban az ún. *Console Application* lehetőséget választjuk, ami egy hagyományos karakteres felületű programkörnyezetet jelent. Ezek az alkalmazások parancssori ablakban futnak.

A munkakönyvtár és a projekt nevének megadása azt jelenti, hogy létrehozza a munkakönyvtáron belül a projekt névvel megadott könyvtárat, esetünkben a *c:\programok\hajra* könyvtárat, és ezen belül egy önkényes *class1.cs* állományt, aminek tartalma a következő:



5. ábra

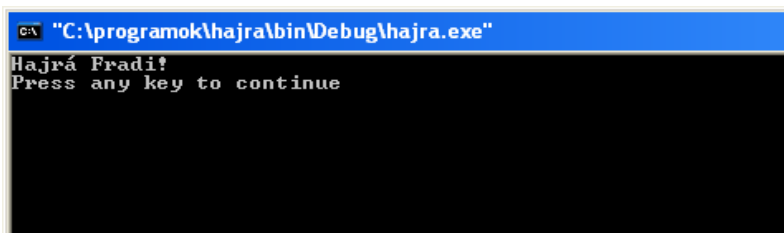
Ahogy látható, a keretrendszer a *class1.cs* állományba mindent előkészít, hogy csak a valódi feladatra kelljen koncentrálnunk. Az állomány tartalma lényegében megegyezik a korábbi első programkóddal. Ahogy már említettem, nem kötelező névteret (*namespace*) definiálni, illetve nem kötelező a *Main* függvény paraméterét jelölni, és a végrehajtási szál attribútum jelölése is opcionális.

Ezeket, illetve a dokumentációs megjegyzéseket figyelembe véve látható, hogy a két kódrészlet azonos.

Amint a 3. ábráról látható, a fejlesztőkörnyezet rendelkezik a már szokásosnak nevezhető beíráskori értelmezéssel (*IntelliSense*), és ha általa ismert típust fedez fel, akkor egy helyi ablakban megmutatja az aktuális objektum elérhető jellemzőit.

A projekt a 3. ábrán látható forráskód mellett még egy *assemblyinfo.cs* állományt is tartalmaz, amiben három jellemző attribútum beállítását végezhetjük el. Beállíthatjuk programunk jellemző adatait, verziószámát, illetve a digitális aláírás kulcsállományra vonatkozó információt is. Ezek a tulajdonságok a .NET keretrendszer szolgáltatásain alapulnak, aminek részletezése meghaladja ennek a könyvnek a kereteit.

A programot a *Debug* menüpont *Start* (F5) vagy *Start without debugging* (CTRL-F5) menüpontjával fordíthatjuk le, illetve futtathatjuk. Ez utóbbi menüpontot használva a parancssori ablak nem tűnik el a program futása után, lehetőséget biztosítva a program eredményének megnézésére.



6. ábra

A fordítás hatására létrejön a *c:\programok\hajra* könyvtárban egy *bin* és egy *obj* könyvtár. Ez utóbbiban a lefordított állományok, míg a *bin* könyvtárban egy *Debug* vagy *Release* könyvtárban létrejön a kívánt exe program is. A két változat közti különbség a nevéből adódik, nevezetesen a *Debug* változatban a nyomkövetés elősegítését biztosítva készül el a futtatható állomány. Ez a lehetőség a programfejlesztések során nagyon hasznos, hiszen logikai vagy egyéb hibák keresésében a lépésenkénti, nyomkövető módszerrel történő végrehajtás nélkülözhetetlen. A *Release*, kiadási végleges változatot a program befejezéseként az 5. ábra fejlécében látható *Debug-Release* választómező állításával készíthetjük el.

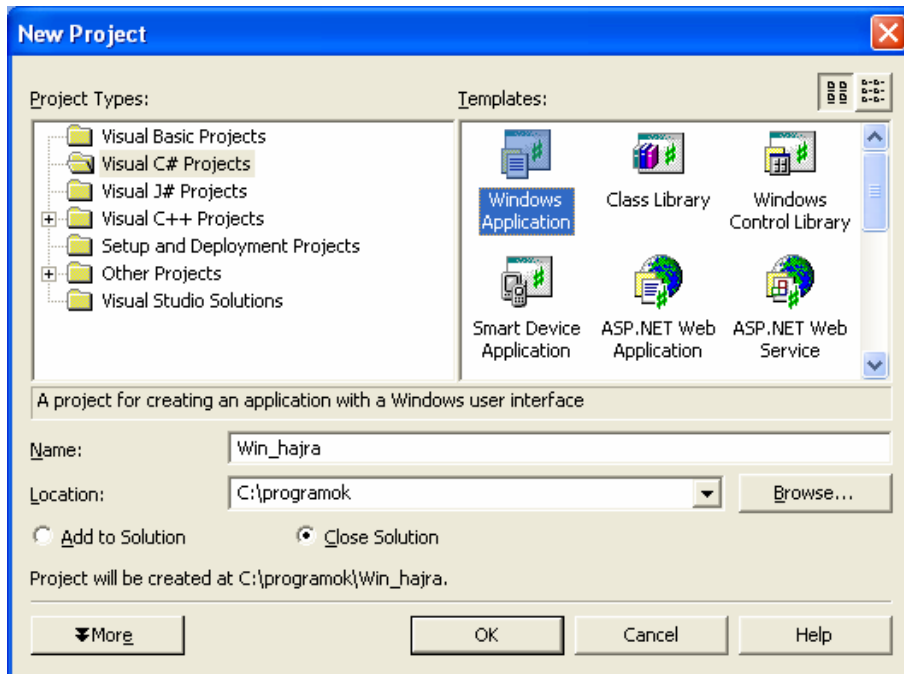
Többfelhasználós környezetben, például Windows XP operációs rendszer alatt, a *Debugger Users* csoportba minden fejlesztőt bele kell rakni.

A könyv második részében a C# nyelv elemeinek megismerésekor jellemző projekt típusként konzolalkalmazást (6. ábra) használunk.

## I.7. Windows alkalmazások készítése

Az alkalmazások másik, és talán ma már szélesebb körben használt csoportja a grafikus alkalmazás készítése. Ezeket a programokat Windows alkalmazásnak is szokták nevezni.

Ahogy a karakteres alkalmazásoknál már láttuk, új feladatot (projektet) készítve a *Windows Application* (Windows alkalmazás) lehetőséget választjuk és megadjuk a nevet, ahogy az a következő képen látható.



7. ábra

Miután a fenti dialógusablakban befejezzük az adatok megadását, azt láthatjuk, hogy a karakteres alkalmazásokhoz képest a munkafelület megváltozik, hiszen ebben a rendszerben az az alap elképzelés, hogy egy üres ablak (*form*) az induló program.

Ez valójában azt jelenti, hogy egy grafikus tervezőablakot kapunk (*Form1*), amibe az eszköztárból (*Toolbox*) a szükséges vezérlőelemeket a grafikus felületre visszük, és eközben a forráskódot (*form1.cs*) a keretrendszer a felületalakításnak megfelelően automatikusan módosítja.

Az első grafikus programunkhoz három lépést végezzünk el!

1. Írjuk át az ablak fejlécszövegét. Ehhez kattintsunk egyet a *form*-ra, majd a jobb alsó *Properties* (Tulajdonságok) ablakban írjuk át a *Text* tulajdonságot. (*Első program!*)
2. A *Toolbox* ablakból helyezünk fel egy *Button*, nyomógomb objektumot. A *Tulajdonság* ablakban állítsuk át a *Text* mezőt a *Vége* szövegre.
3. Kattintsunk kettőt a nyomógomb objektumra, ezzel a nyomógomb alapértelmezett kattintás esemény függvényét definiáltuk. Ebbe írjuk be a *Close()*, ablakbezárás utasítást. Ekkor a *form1.cs* forráskód részlete a következőképpen néz ki:

```
[STAThread]
static void Main()
{
    Application.Run(new Form1());
}

private void button1_Click(object sender, System.EventArgs e)
{
    Close();
}
```

Ezen három lépés után fordítsuk le, majd futtassuk a programot. Ezt, ahogy a korábbi karakteres program esetében is a *Debug* menü *Start* menüpontjával tehetjük meg.

Ahhoz, hogy további grafikus alkalmazásokat tudjunk készíteni, elengedhetetlenül szükséges, hogy ismerjük egyrészt a választott nyelv lehetőségeit, másrészt a rendszer könyvtári szolgáltatásait.

Az előbbi, a nyelvi lehetőségek megismerése általában a legkönnyebb és leggyorsabb feladat. Ez reményeim szerint a következő rész áttanulmányozása után a kedves Olvasónak is sikerül. Viszont az utóbbi, a rendszer könyvtári szolgáltatásainak megismerése hosszabb, több időt, sok egyéni munkát jelentő feladat. Azt remélem, hogy a befejező, a grafikus alkalmazások alapjaival foglalkozó rész után mindenki elég bátorságot érez magában a további önálló munka folytatásához.

## **I.8. Feladatok**

1. Milyen kódú programot fordít a Visual Studio.NET fejlesztőkörnyezet?
2. Mit csinál a Garbage Collection (szemétgyűjtő) algoritmus?
3. Lehet-e ékezetes karaktereket használni a C# programban?
4. Mik a *Main* függvény jellemzői?
5. Egy program hány *Main* függvényt tartalmazhat, hogy tudjuk lefordítani?

## II. A C# nyelv alaptípusai

Egy programozási nyelvben az egyik legfontosabb tulajdonság az, hogy programkészítés során hogyan és milyen típusú adatokat használhatunk. Meg kell jegyezni, hogy van néhány olyan programozási nyelv is (pl. PHP), ahol ez a terület nem igazán fontos, típusok gyakorlatilag nincsenek, adatot szükség szerinti típusban a programozó rendelkezésére bocsát.

A C# szigorúan típusos nyelv, ebben a nyelvben csak ennek figyelembevételével használhatunk saját változókat.

### II.1. Változók definiálása

A nyelvben a változók definiálásának alakja:

típus változónév ;
--------------------

Azonos típusú változókat egymástól vesszővel elválasztva definiálhatunk. A típusok ismerete nélkül nézzünk néhány példát változók definiálására.

Példa:

```
char ch;           // ch változó karakter típusú
int egy, tizenegy; // az egy és tizenegy egész típusú
```

Változók lehetnek külsők, azaz függvényen kívüliek, vagy belsők, azaz függvényen belüliek. A függvényen kívüli változók is természetesen csak osztályon belüliek lehetnek. Gyakran hívják ezeket a változókat adatmezőknek is.

Belső változók, az adott blokkon (függvényen) belüli lokális változók lehetnek dinamikus vagy statikus élettartamúak. Módosító jelző nélküli definiálás esetén dinamikusnak vagy automatikusnak nevezzük, s ezek élettartama a blokkban való tartózkodás idejére korlátozódik. Ha a blokk végrehajtása befejeződött, a dinamikus változók megszűnnek.

Statikus élettartamú belső változót a *static* szó használatával (ahogy külső változó esetén igen) nem definiálhatunk. Láttuk, a függvények lehetnek statikusak (lásd *Main* függvény), melyekre ugyanaz igaz, mint az osztályváltozókra, ezen függvények élettartama is a programéval egyezik meg, más szóval ezen függvények a program indulásakor jönnek létre.



A változókat két kategóriába sorolhatjuk, az osztálytípusú változók referencia típusúak, melyek mindig a dinamikus memóriában helyezkednek el (az irodalomban ezt gyakran *heap*-nek nevezik), míg minden más nem osztálytípusú, az úgynevezett értéktípusú (*value type*) változó. Az értéktípusú változókat szokták *stack változóknak* is nevezni.

Az értéktípusú változók kezdőértékét, az inicializálás hiányában, *0*, *false*, *null* értékre állítja be a fordító.

A változók definiálásakor rögtön elvégezhető azok direkt inicializálása is.

## II.2. Állandók definiálása

Állandók definiálását a típusnév elé írt *const* típusmódosító szócska segítségével tehetjük meg. A definíció alakja:

const típus név = érték;

Példa:

```
const float g=9.81;    // valós konstans
g=2.3;                // !!! HIBA
...
const int min=0;       // egész konstans
```

## II.3. Változók inicializálása

Változók kezdő értékadása, inicializálása azok definiálásakor is elvégezhető a következő formában:

típus változó = érték;

Példa:

```
char s='a'; int []a={1,2,3};
char[] b="Egy";
```

A változók, konstansok és függvények használatára nézzük a következő példát.

## II. A C# nyelv alaptípusai

---

Példa:

```
// A valtozo.cs fájl tartalma:
using System;
// A kiíráshoz szükséges deklarációt tartalmazza.
class változó
{
    static int alma=5;           //alma változó definíciója
                                //és inicializálása
    static float r=5.6F          //statikus valós típusú változó
                                //valós konstans zárhat az
                                //F (float) betű
    const float pi=3.1415;       //konstans definíció

    static void Main()
    {
        Console.WriteLine( "Teszt"); //eredmény Test
        Console.WriteLine( alma );   //eredmény 5
        int alma=6;                  //helyi változó
        Console.WriteLine( alma );   //eredmény 6
        Console.WriteLine( változó.alma ); //a külső takart (5)
                                        //változóra hivatkozás
        Console.WriteLine( terület(r)); //a terület függvény
                                        // meghívása
    }

    static float terület(float sugár) // függvénydefiníció
    {
        return (pi*sugár*sugár);
    }
}
```

Bár még nem definiáltunk függvényeket, így talán korainak tűnhet ez a példa, de inkább felhívnám még egyszer a figyelmet az ékezetes karakterek használatára. A későbbi mintafeladatokban, ha előfordul ékezet nélküli változó, függvénydefiníció, akkor az csak a korábbi környezetek „rossz utóhatásának” köszönhető. Egy osztályon belül a függvények definiálási sorrendje nem lényeges. Sok környezetben furcsa lehet amit a fenti példában látunk, hogy előbb használjuk, és csak ezután definiáljuk a függvényünket. (terület függvény)

### II.4. Elemi típusok

*char* – karakter típus (2 byte hosszú)

A karakter típus 16 bites karakterábrázolást (unikód) használ. Általában igaz, hogy minden alaptípus mérete rögzített.

A karakter típusú változó értékét aposztróf (') jelek között tudjuk megadni.

Példa:

```
char c;  
c='a';
```

A backslash (\) karakter speciális jelentéssel bír. Az utána következő karakter(ek)e)t, mint egy escape szekvenciát dolgozza föl a fordító. Az így használható escape szekvenciák a következők:

\a	-	a 7-es kódú csipogás
\b	-	backspace, előző karakter törlése
\f	-	formfeed, soremelés karakter
\r	-	kocsi vissza karakter
\n	-	új sor karakter (soremelés+kocsi vissza)

Az új sor karakter hatása azonos a formfeed és a kocsi vissza karakterek hatásával.

\t	-	tabulátor karakter
\v	-	függőleges tabulátor
\\	-	backslash karakter
\'	-	aposztróf
\"	-	idézőjel
\?	-	kérdőjel
\uxxyy		xx és yy unikódú karakter

### ***string*** – karaktersorozat

A *System.String* osztálynak megfelelő típus. Szövegek között a + és a += szövegösszefűzést végző operátorok ismertek. A [] operátor a szöveg adott indexű karakterét adja meg. Az indexelés 0-val kezdődik. A @ karakter kiküszöböli a szövegen belüli „érzékeny” karakterek hatását. Ilyen például a backslash (\) karakter.

Példa:

```
char c='\u001b';           // c= a 27-es kódú (Esc) karakterrel  
string s="alma";  
string n="alma"+"barack";  
s+="fa";                   //s= almafa
```

## II. A C# nyelv alaptípusai

---

```
char c1=s[2];           // c1= 'm'
char c2="szia"[1];      // c2=' z'
string f="c:\\programok\\alma.txt";
string file=@"c:\programok\alma.txt";
```

A *String* osztály a fenti lehetőségeken kívül egy sor függvénnyel teszi használhatóbbá ezt a típust. Ezen függvények közül a legfontosabbak:

### **Length**

Csak olvasható tulajdonság, megadja a szöveg karaktereinek a számát:

```
string s="alma";
int i=s.Length;      //i=4
```

### **CompareTo(string)**

Két szöveg összehasonlítását végzi. Ha az eredmény *0*, akkor azonos a két szöveg:

```
string str1="egyik", str2="másik";
int cmpVal = str1.CompareTo(str2);
if (cmpVal == 0)      // az értékek azonosak
    {...}
else if (cmpVal > 0)  // str1 nagyobb mint str2
    {...}
else                  // str2 nagyobb mint str1
```

### **Equals(string)**

Megadja, hogy a paraméterül kapott szöveg azonos-e az eredeti szöveggel:

```
string str1="egyik", str2="másik";
if (str1.Equals(str2){...}    // azonos
else{...}                    // nem azonos
```

### **IndexOf(string)**

Több alakja van, megadja, hogy az eredetiben melyik indexnél található a paraméterül kapott szöveg. A visszaadott érték *-1* lesz, ha nincs benn a keresett szöveg:

```
int i="almafa".IndexOf("fa");    //4
```

### **Insert(int,string)**

A második paraméterül kapott szöveget, az első paraméterrel megadott indextől beszúrja az eredeti szövegbe:

```
string s="almafa alatt";  
string ujs=s.Insert(4," a ");    // alma a fa alatt
```

A szövegtípus további szolgáltatásait, függvényeit a szövegosztály (*System.String*) online dokumentációjában érdemes megnézni. Meg kell említeni még azt, hogy a szövegosztály függvényei nem módosítják az eredeti szöveget, szövegobjektumot, példaként az előző *s* szöveges változó értéke változatlan marad.

Szöveges feldolgozási feladatok során a reguláris kifejezésekkel megadható szövegminták segítségét is igénybe vehetjük. A .NET Framework a Perl5 kompatibilis reguláris kifejezéshasználót támogatja. A *Regex* osztály szolgáltatja a reguláris kifejezést, míg a *Match* a találati eredményt. Az osztályokat a *System.Text.RegularExpressions* névtérben találjuk.

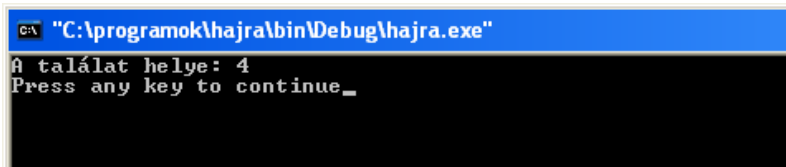
Példa:

```
using System;  
using System.Text.RegularExpressions;  
class reguláris  
{  
    public static void Main()  
    {  
        Regex reg_kif = new Regex("fradi");  
        // a fradi keresése  
        Match találat = reg_kif.Match("A Fradi jó csapat?");  
        if (találat.Success)  
        {  
            // (hol találtuk meg  
            Console.WriteLine("A találat helye: " + találat.Index);  
        }  
    }  
}
```

A fenti példa nem ad eredményt, hiszen alapértelmezésben a kis- és nagybetű különbözik, míg ha a reguláris kifejezést egy kicsit módosítjuk, az alábbiak szerint:

```
Regex reg_kif = new Regex("adi");
```

akkor a futási eredmény az alábbi lesz:



8. ábra

Ha azt szeretnénk elérni, hogy az eredeti szövegünk is változzon, akkor ehhez a *System.Text* névtér *StringBuilder* osztályát tudjuk igénybe venni.

Példa:

```
using System.Text;
...
StringBuilder s1 = new StringBuilder("almafa alatt");
s1.Insert(4, " a ");
Console.WriteLine(s1);      // "alma a fa alatt"
```

### *int* – egész típus(4 byte)

Az egész típusú értékek négy bájtot foglalnak – a leggyakrabban használt nyelvi környezetben –, így értelmezési tartományuk  $-2^{31}$ ,  $2^{31} - 1$  között van.

long	hosszú egész (8 byte)
short	rövid egész (2 byte)
sbyte	előjeles (signed) byte
float	valós (4 byte)
double	dupla pontosságú valós (8 byte)
decimal	„pénzügybeli” típus (16 byte), 28 helyiérték

Mindkét egész típus előjeles, ha erre nincs szükségünk, használhatjuk az előjel nélküli változatukat, melyek: *uint*, *ushort*, *byte*, *ulong*.

Valós számok definiálásakor a 10-es kitevőt jelölő *e* konstans használható.

Példa:

```
float a=1.6e-3;    // 1.6 * 10-3
```

### *void* – típus nélküli típus

Az olyan függvénynek (eljárásnak) a típusa, amelyik nem ad vissza értéket.

Példa:

```
void növel(ref int mit)
{
    mit++;
}
```

Az iménti függvény paraméterének jelölése referencia szerinti paraméterátadást jelent, amiről a *Függvények* c. fejezetben részletesen szólnunk.

### ***bool*** – logikai típus (1 byte)

A logikai típus értéke a *true* (igaz) vagy *false* (hamis) értéket veheti fel. Ahogy a nyelv foglalt alapszavainál láthattuk, ezeket az értékeket a *true* és *false* nyelvi kulcsszó adja meg.

Általában elmondható, hogy míg a nyelv nem definiál sok alaptípust, addig az egyes implementációk, főleg azok grafikus felület alatti könyvtárai ezt bőségesen pótolják, és gyakran több időt kell az 'új típusok' megfejtésének szentelni, mint a függvények tanulmányozásának.

## **II.5. Felsorolás típus**

A felsorolás típus gyakorlatilag nem más, mint egész konstans(ok), szinonimák definiálása. Felsorolás típust névtéren vagy osztályon belül definiálhatunk.

Ennek a kulcsszava az *enum*. A kulcsszó után a felsorolás típus azonosítóját meg kell adni. A *System.Enum* osztály szinonimáját adja az ezen kulcsszó segítségével definiált típus.

Példa:

```
enum szinek {piros,kék,zöld,sárga};
enum betuk {a='a', b='b'};
betuk sajátbetű=betuk.a;           // változó kezdőértéke a
enum valami { x="alma"};           // hiba
```

Ekkor a 0,1, ... értékek rendelődnek hozzá a felsorolt nevekhez, és a nevek kiírásakor ezen számokat is látjuk. A kezdőértékadás lehetőségével élve nem kell ezeket feltétlenül elfogadnunk, hanem mi is megadhatjuk az értéküket.

Példa:

```
class Szinek
{
    enum újszinek {piros=4, kék=7, zöld=8, sárga=12};

    public static void Main()
```

## II. A C# nyelv alaptípusai

---

```
{
    Console.WriteLine(újszinek.zöld); // kiírja a színt
    //ciklus a színeken történő végighaladásra
    for(újszinek i=újszinek.kék; i<újszinek.sárga; i++)
    {
        //kiírja a színeket
        Console.WriteLine(i);
    }
}
```

A fenti ciklus eredményeként azon egészek esetén, ahol az egész értékéhez egy „nevet” rendeltünk hozzá, a név kerül kiírásra, egyébként a szám. Az eredmény a következő lesz:

```
kék
zöld
9
10
11
```

Mivel ez a típus a *System.Enum* megfelelője, ezért rendelkezik annak jellemzőivel is. Ezek közül a két legjellemzőbb a *GetHashCode()* és a *ToString()* függvény. Az előbbi a belső tárolási formát, a megfelelő egész számot adja meg, míg a *ToString()*, mint alapértelmezett reprezentáció, a szöveges alakot (kék,zöld, stb) adja meg.

Példa:

```
újszinek s=újszinek.piros;
Console.WriteLine(s.GetHashCode()); // eredmény: 4
```

A felsorolás típus alapértelmezésben egész típusú értékeket vesz fel. Ha ez nem megfelelő, akár *byte* (vagy tetszőleges egész szinonima) típusú értékeket is felvehet úgy, hogy a típusnév után kettősponttal elválasztva megadom a típusnevet. (Nem adhatom meg a valós vagy karakter típust!)

Példa:

```
enum színek:byte {piros,kék,zöld,sárga};
...
enum hibás:float {rossz1, rossz2}; // ez fordítási hiba
```

Előfordulhat, hogy olyan felsorolásadatokra van szükségem, melyek nem egy egész konstanshoz, hanem egy bithez kötődnek, hiszen ekkor a logikai



és/vagy műveletekkel a felsorolásadatok kombinációját határozhatjuk meg. Ebben az esetben a *[Flags]* attribútumot kell az *enum* szócska elé szúrni.

Példa:

```
[Flags] enum alma
{piros=1,jonatán=2,zöld=4,golden=8};
...
alma a=alma.piros | alma.jonatán;
Console.WriteLine(a);           // piros, jonatán
a=(alma) System.Enum.Parse(typeof(alma),"zöld,golden");
// a felsorolás típus egyszerű értékadása helyett
// a könyvtári hívás lehetőségét is használhatjuk
//
Console.WriteLine(a.GetHashCode());
...
// eredmény 12 lesz
```

Bithez kötött értékek esetén kötelező minden egyes konstans névhez megadni a neki megfelelő bites ábrázolást!

Az alaptípusokat a .NET keretrendszer biztosítja, így ennek a fejlesztési környezetnek egy másik nyelvben pontosan ezek a típusok állnak rendelkezésre. Az természetesen előfordulhat, hogy nem ezekkel a nevekkal kell rájuk hivatkozni, de a nyelvi fordító biztosan ezen értelmezés szerinti kódot (köztes kód) fordítja le a keretrendszer, mint futtató környezet számára.

Az alaptípusok zárásaként meg kell jegyezni, hogy a mutató típus is értelmezett, ahogy a C++ világában, de ennek a használata csak olyan C# programrészben megengedett, amely nem biztonságos környezetben helyezkedik el (*unsafe context*). Erről röviden a könyv XIV. fejezetében olvashat.

## II.6. Feladatok

1. Milyen alaptípusokat ismer a C# nyelv?
2. Mi a változó és az állandó között a különbség?
3. Mi a különbség egy statikus és egy dinamikus élettartamú változó között?
4. Definiáljon két szöveg típusú változót, adja meg a hosszukat!
5. Ábrázolja felsorolás típussal az NB 1 bajnokság csapatait!

### *III. Kifejezések, műveletek*

A nyelv jellemzője a korábban (például C++ nyelvben) megismert, megszokott kifejezésfogalom, amit tömören úgy is fogalmazhatunk, hogy a vezérlési szerkezeteken kívül a nyelvben minden kifejezés.

Egy kifejezés vagy elsődleges kifejezés, vagy operátorokkal kapcsolt kifejezés.

Elsődleges kifejezések:

- azonosító
- (kifejezés)
- elsődleges kifejezés[]
- függvényhívás
- elsődleges kifejezés.azonosító

A kifejezéseket egy-, két- vagy háromoperandusú operátorokkal kapcsolhatjuk össze.

Egy kifejezésen belül először az elsődleges kifejezések, majd utána az operátorokkal kapcsolt kifejezések kerülnek kiértékelésre.

A nyelvben használható operátorok prioritási sorrendben a következők.

#### **III.1. Egyoperandusú operátorok**

**new**

A dinamikus helyfoglalás operátora. A helyfoglalás operátorának egyetlen operandusa az, hogy milyen típusú objektumnak foglalunk helyet. Sikeres helyfoglalás esetén a művelet eredménye a lefoglalt memória címe. Sikertelen helyfoglalás esetén a *null* mutató a visszatérési eredmény.

Példa:

```
int[] a;  
a = new int;           //egy egész tárolásához  
                        //elegendő hely foglalása  
a = new int[5]         //öt egész számára foglal helyet
```

A *new* utasítás gyakran szerepel a vektorokkal összefüggésben, amiről később részletesen szólunk.

A .NET keretrendszer egyik legfontosabb tulajdonsága az, hogy a dinamikusan foglalt memóriaterületeket automatikusan visszacsatolja a felhasználható memóriatartományba, ha arra a memóriára a programnak már nincs szüksége. Sok nyelvi környezetben erre a *delete* operátor szolgál.

-	aritmetikai negálás
!	logikai negálás (not)
~	bitenkénti negálás,
++,--	inc, dec.
(típus) kifejezés	típuskényszerítés

Példa:

```
double d=2.3;  
int i=(int) d;    // i=2
```

A ++ és -- operátor szerepelhet mind pre-, mind postfix formában. Prefix esetben a változó az operátorral változtatott értékét adja egy kifejezés kiértékelésekor, míg postfix esetben az eredetit.

Példa:

```
a= ++b;           // hatása: b=b+1; a=b;  
a= b++;           // hatása: a=b; b=b+1;
```

Az egyoperandusú operátorok és az értékadás operátora esetén a végrehajtás jobbról balra haladva történik, minden más operátor esetén azonos precedenciánál balról jobbra.

## III.2. Kétooperandusú operátorok

*	szorzás
/	osztás
%	maradékképzés

E három operátorral mindig igaz: az  $(a/b)*b+a\%b$  kifejezés az  $a$  értékét adja.

(% operátor esetén az operandusok nem lehetnek valósak, és a prioritásuk azonos.)

### III. Kifejezések, műveletek

---

+	összeadás, string esetén azok összefűzése
-	kivonás
<<	bitenkénti balra léptetés
>>	bitenkénti jobbra léptetés

A jobbra léptetés operátorához példának tekintsük az  $a \gg b$ ; utasítást. Ekkor ha az  $a$  unsigned, akkor balról nullával, különben pedig az előjelbittel tölti a bitenkénti jobbra léptetés operátora.

Példa:

```
int a=-1, b;  
b= a >> 2;           // b értéke -1 marad
```

<, >	kisebb, nagyobb relációs operátorok
<=, >=	kisebb vagy egyenlő, ill. a nagyobb vagy egyenlő operátorok
==, !=	egyenlő, nem egyenlő relációs operátorok
&	bitenkénti és
^	bitenkénti kizáró vagy
	bitenkénti vagy
&&	logikai és
	logikai vagy

#### **is**

Logikai operátor, egy objektumról megmondja, hogy a bal oldali operandus a jobb oldali típusnak egy változója-e.

Példa:

```
...  
int i=3;  
if (i is int) Console.WriteLine("Bizony az i egész!");  
    else Console.WriteLine("Bizony az i nem egész!");
```

#### **as**

Kétooperandusú típuskényszerítés. A bal oldali változót a jobb oldali referencia típusra alakítja, ha tudja. Ha sikertelen az átalakítás, akkor eredményül a *null* értéket adja.

Példa:

```
...  
double d=2.5;  
Object o= d as Object;  
Console.WriteLine(o);           // eredmény: 2.5  
// Object-re mindent lehet alakítani, aminek van toString kiíró  
//függvénye. Erről bővebben később esik szó.
```

### **typeof**

Egy *System.Type* típusú objektumot készít. Ennek az objektumnak a mezőfüggvényeivel, tulajdonságaival (*FullName*, *GetType()*) tudunk típusinformációhoz jutni.

Példa:

```
using System;
class Teszt
{
    static void Main() {
        Type[] t = {
            typeof(int),
            typeof(string),
            typeof(double),
            typeof(void)
        };
        for (int i = 0; i < t.Length; i++)
        {
            Console.WriteLine(t[i].FullName);
        }
    }
}
```

A program futása a következő eredményt produkálja:

```
System.Int32
System.String
System.Double
System.Void
```

A típuskonverzióval kapcsolatban elmondható, hogy minden értéktípusból létrehozhatunk egy neki megfelelő *Object* típusú objektumot. Ezt gyakran *boxing*-nak, csomagolásnak, míg az ellenkező kicsomagoló műveletet, amihez a zárójeles típuskonverzió operátort kell használni, *unboxing*-nak nevezi a szakirodalom.

Példa:

```
int i=5;
Object o=i;           // boxing, becsomagolás
int j=(int) o;         // unboxing, kicsomagolás
```

Az *Object* típus, ami a *System.Object* típusnak felel meg, minden típus őseként tekinthető. Ennek a típusnak a függvényei ily módon minden általunk használt típushoz rendelkezésre állnak.

### III. Kifejezések, műveletek

---

Az *Object* típus tagfüggvényei a következők:

#### **ToString()**

Megadja az objektum szöveges reprezentációját. Ha erre van szükség, például kiírásnál, akkor ezt automatikusan meghívja. Ha ezt egy új típushoz újradefiniáljuk, akkor ez hívódik meg kiírás esetén.

```
Console.WriteLine(o);          // indirekt ToString hívás
Console.WriteLine(o.ToString());
```

#### **GetType()**

Megadja az objektum típusát, hasonló eredményt ad, mint a korábban látott *typeof* operátor.

#### **Equals(object)**

Megadja, hogy két objektum egyenlő-e, logikai értéket ad eredményül.

```
int i=5;
object o=i;
Console.WriteLine(o.Equals(5));
```

Az *Equals* függvénynek létezik egy statikus változata is, aminek a formája: *Object.Equals(object, object)*

#### **GetHashCode()**

Megadja az objektum *hash* kódját, ami egy egész szám. Tetszőleges saját utódtípusban újradefiniálhatjuk, amivel a saját típusunk *hash* kódját tudjuk számolni.

## III.3. Háromoperandusú operátor

Az operátorral képezhető kifejezés alakja a következő:  $e1 ? e2 : e3$ , ahol a ? és a : az operátor jelei, míg  $e1, e2, e3$  kifejezések.

A kifejezés értéke  $e2$  ha  $e1$  igaz (nem 0), különben  $e3$ .

Példa:

```
char c;
int a;
...
a = ((c >= '0' && c <= '9') ? c - '0' : -1);
```

Az  $a$  változó vagy a 0-9 vagy  $-1$  értéket kapja.

A háromoperandusú operátor valójában egy logikai elágazás utasítás. A hatékony kifejezőkészítés, tömörebb írásmód kiváló eszköze.

### III.4. Kétoperandusú értékadó operátorok

= értékadás operátora

A nyelvben az értékadás sajátos, önmaga is kifejezés. Az egyenlőségjel bal oldalán olyan kifejezés, változó állhat, amely által képviselt adat új értéket vehet fel. Gyakran hívja a szakirodalom ezt balértéknek is (*lvalue*). Az egyenlőségjel jobb oldalán egy értéket adó kifejezés kell, hogy álljon (jobbérték, *rvalue*). Ez gyakorlatilag azt jelenti, hogy a jobbértékre semmilyen korlátozás nincs.

Az értékadás során a bal oldal megkapja a jobb oldali kifejezés értékét, és egyben ez lesz a kifejezés értéke is.

Példa:

```
...
char []d=new char[80]; // másoljuk az s forrásszöveget d-be
while (i<s.Length)
{
    d[i]=s[i]; i++;
}
char []c={'L','a','l','i'};
d=c;      // hiba!! mivel d már egy 80 karakteres
          //területet jelöl, ezért új területet már nem
          // jelölhet, d nem lehet balérték
```

Az értékadás operátora jobbról balra csoportosít, ezért pl. az  $a=b=2$ ; utasítás hatására  $a$  és  $b$  is 2 lesz.

Az egyenlőségjel mellett még néhány, az aritmetikai operátorokkal 'kombinált' értékadó operátor is használható.

Ezek az operátorok a következők:

$+=$ ,  $-=$ ,  $*=$ ,  $/=$ ,  $\%=$ ,  $>>=$ ,  $<<=$ ,  $\&=$ ,  $\^{}=$ ,  $|=$

Jelentésük: az  $e1$  operátor  $e2$  kifejezés, ahol  $e1$  és  $e2$  is kifejezések, az  $e1 = e1 \text{ op } e2$  kifejezésnek felel meg, ahol  $op$  az egyenlőségjel előtti operátor.

### III. Kifejezések, műveletek

---

Példa:

```
int a=2;
a *= 3;      // a= a*3, azaz a értéke 6 lesz
string b="alma";
b+="fa";     // b=almafa
```

A típusok egymás közti konverziójával kapcsolatban azt lehet mondani, hogy a C vagy C++-ban ismert automatikus konverziók nem léteznek. Egy egész típusú változót egy valósba gond nélkül beírhatunk, míg fordítva már hibát jelez a fordító. A konverziós lehetőségek széles skáláját nyújtja a fejlesztőeszközünk.

Konverziókkal kapcsolatban két esetet szoktak megkülönböztetni. Ezek közül az első szám szöveggé alakítása. Ez gyakorlatilag nem igényel semmilyen segítséget, a számtípushoz tartozó osztály *ToString* függvényét hívja meg a fordító. Ehhez nem kell semmit sem tenni.

Példa:

```
int a=2;
string s="Az eredmény:" + a;
```

A másik eset, mikor szövegből szeretnénk számot kapni, már nem ilyen automatikus, de semmiképpen nem lehet bonyolultnak nevezni. Többféle módszer lehet az alakításra, de a legáltalánosabb talán a *Convert* osztály használata. Az osztály konvertáló függvényei azonos névkonvencióval az alábbi formájúak:

```
Convert.ToCTStípusnév
```

ahol a CTS (Common Type System) típusnév az alábbi lehet: *Boolean*, *Int16* (short int), *Int32* (rendes int), *Int64* (hosszú int), *Float*, *Double*, *String*, *Decimal*, *Byte*, *Char*.

Példa:

```
string s="25";
int i=Convert.ToInt32(s); // i=25 lesz
```

Érdekességgént megemlítem, hogy létezik a *Convert.ToDateTime(object)* függvény is, ami egy könyvtári dátumtípust készít a paraméterül kapott objektumból.

Ha bármelyik konvertáló függvény hibás paramétert kap, nem tud eredményt produkálni, akkor *InvalidCastException* kivételt vagy más, a hiba okára utaló kivételt generál. A kivételkezelésről később részletesen is fogunk beszélni.



Általában igaz, hogy grafikus alkalmazások során a beírt adataink szövegesek, így minden esetben az azokkal történő számolás előtt konvertálnunk kell, ezért a konvertáló függvények használata elég gyakori.

Hasonló konvertáló szolgáltatásokat kapunk, ha az alaptípusok *Parse* függvényét használjuk (*int.Parse(szöveg)*, *double.Parse(szöveg)*).

Gyakran előfordul, hogy az alpműveletek nem elégítik ki a számolási igényeinket. A *System* névtér *Math* osztálya a leggyakrabban használt számolási műveleteket biztosítja. A leggyakrabban használt függvények a következők:

<code>Math.Sin(x)</code>	<code>sin(x)</code> , ahol az <code>x</code> szög értékét radiánban kell megadni
<code>Math.Cos(x)</code>	<code>cos(x)</code>
<code>Math.Tan(x)</code>	<code>tg(x)</code>
<code>Math.Exp(x)</code>	<code>ex</code>
<code>Math.Log(x)</code>	<code>ln(x)</code>
<code>Math.Sqrt(x)</code>	<code>x</code> négyzetgyöke
<code>Math.Abs(x)</code>	<code>x</code> abszolút értéke
<code>Math.Round(x)</code>	kerekítés a matematikai szabályok szerint
<code>Math.Ceiling(x)</code>	felfelé kerekítés
<code>Math.Floor(x)</code>	lefelé kerekítés
<code>Math.Pow(x,y)</code>	hatványozás, <code>xy</code>
<code>Math.PI</code>	a $\pi$ konstans (3.14159265358979323846)
<code>Math.E</code>	az $e$ konstans (2.7182818284590452354)

Példa:

```
...
double dd=Math.Sin(Math.PI/2);
Console.WriteLine(dd);      // értéke 1.
dd=Math.Pow(2,3);
Console.WriteLine(dd);      // 8
```

Matematikai, statisztikai feladatoknál a véletlenszámok használata gyakori igény. A *System* névtér *Random* osztálya nyújtja a pseudo-véletlenszámok generálásának lehetőségét. Egy véletlenszám-objektum létrehozását a rendszer-időhöz (paraméter nélküli konstruktor) vagy egy adott egész számhoz köthetjük. A véletlenobjektum *Next* függvénye a következő véletlen egész számot, a *NextDouble* a következő valós véletlenszámot adja. A *Next* függvénynek három, míg a *NextDouble* függvénynek egy változata van, amit a következő példa is bemutat.

### III. Kifejezések, műveletek

---

Példa:

```
Random r=new Random();  
//r véletlenszám objektum rendszeridő alapú létrehozása  
Random r1=new Random(10);  
// r1 véletlenobjektum generátor a 10 értékből indul ki  
//  
int v= r.Next();  
// véletlen egész szám 0 és MaxInt (legnagyobb egész) között  
//0 lehet az érték, MaxInt nem  
//  
int v1=r.Next(10);  
// véletlen egész szám 0 és 10 között, 0<=v1<10  
//  
int v2=r.Next(10,100);  
// véletlen egész szám 10 és 100 között, 10<=v2<100  
//  
double d=r.NextDouble();  
// véletlen valós szám 0 és 1 között, 0<=d<1
```

### III.5. Feladatok

1. Mit nevezünk kétoperandusú operátornak?
2. Melyik operátor három operandusú?
3. Milyen szám-szöveg konverziós lehetőségeket ismer?
4. Fogalmazza meg két elem közül a nagyobb kiválasztását *operator* segítségével!
5. Ismerjük egy háromszög két oldalát és közbezárt szögét. Számoljuk ki a szöggel szemközti oldal hosszát!

## IV. Összetett adattípusok

### IV.1. Tömbök

A feladataink során gyakori igény az, hogy valamilyen típusú elemből többet szeretnénk használni. Amíg ez a „több” kettő vagy három, addig megoldás lehet, hogy két vagy három változót használunk. Amikor viszont tíz, húsz adatra van szükségünk, akkor ez a fajta megoldás nem igazán kényelmes, és sok esetben nem is kivitelezhető. Lehetőség van azonos típusú adatok egy közös névvel való összekapcsolására, és az egyes elemekre index segítségével hivatkozhatunk. Ezt az adatszerkezetet tömbnek nevezzük. A tömbök elemeinek elhelyezkedése a memóriában sorfolytonos. A tömb helyett gyakran használjuk a vektor szót, annak szinonímjaként.

A C# nyelv minden tömb- vagy vektortípus definíciót a *System.Array* osztályból származtat, így annak tulajdonságai a meghatározóak.

A tömbök definíciójának formája a következő:

típus[] név;
--------------

Az egyes tömbelemekre való hivatkozás, a tömbök indexelése mindig 0-val kezdődik. Az index egész típusú kifejezés lehet, a típus pedig tetszőleges.

Egy tömb általános definíciója nem tartalmazza az elemszám értékét. Bár a nyelv környezetében nem lehet mutatókat definiálni, de ez gyakorlatilag azt jelenti. A vektor definíciója egy referencia típus létrehozása, és a nyelv minden referencia típusát a *new* operátorral kell konkrét értékekkel inicializálni (példányosítani). Ekkor kell megmondani, hogy az adott vektor hány elemű lesz. Minden vektornak, ellentétben a C++ nyelvvel, a létrehozása után lekérdezhető az elemszáma a *Length* tulajdonsággal.

Példa:

```
int[] numbers;           // egész vektordefiníció
numbers = new int[10];    // 10 elemű lesz az egész vektorunk
...
numbers = new int[20];    // most meg 20 elemű lett
int db = 15;
numbers = new int[db];    // egy változó adja a hosszát
```

#### IV. Összetett adattípusok

---

Mivel a nyelv minden dinamikus referencia típusának memóriabeli felszabadítását a rendszer automatikusan végzi – ezt az irodalom gyakran szeméthyűjtési algoritmusnak (*garbage collection*) nevezi –, ezért a példabeli 10 elemű vektor memóriahelyét automatikusan visszakapja az operációs rendszer.

A vektorelemek a fenti dinamikus létrehozás után 0 kezdőértéket kapnak. Ha egyéb elemekkel szeretnénk az inicializációt elvégezni, kapcsos zárójelek közé írva adhatjuk meg azokat.

típus[] név={érték, érték,...};

Példa:

```
int[] n={3,4,5,6};  
int hossz= n.Length;           // vektorhossz meghatározás
```

Ha logikai vektort definiálunk, akkor a vektorelemek kezdőértékkadás hiányában logikai hamis (*false*), míg ha referencia vektort definiálunk, akkor a referenciaelemek a *null* kezdőértéket kapják meg.

Példa:

```
int[] numbers = {1, 2, 3, 4, 5};  
int[] numbers = new int[5] {1, 2, 3, 4, 5};  
string[] nevek = new string[3] {"Ali", "Pali", "Robi"};  
int[] numbers = new int[] {1, 2, 3, 4, 5};  
string[] nevek = new string[] {"Ali", "Pali", "Robi"};
```

A C# nyelvben nem csak a fenti egydimenziós vektor definiálható. Ezenkívül még definiálható két- vagy többindexű, multidimenziós vektor, illetve vektorok vektora.

#### *Multidimenziós vektor:*

Példa:

```
string[,] nevek;  
...  
nevek= new string[2,4];
```

A vektor dimenzióját a *Rank* tulajdonsággal kérdezhetjük le.

```
Console.WriteLine(nevek.Rank);           // 2  
// az egyes dimenziókbeli méretet a GetLength adja  
Console.WriteLine(nevek.GetLength(0));   // 2
```

```
Console.WriteLine(nevek.GetLength(1)); // 4
```

Természetesen a normál vektornak (*int[] v*) is lekérdezhetjük a dimenzió értékét.

Példa:

```
int[,] numbers = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
string[,] kapocs = new string[2, 2] { {"Miki","Ani"}, {"Mari","Albert"} };
int[,] numbers = new int[, ] { {1, 2}, {3, 4}, {5, 6} };
string[,] kapocs = new string[, ] { {"Miki","Ani"}, {"Mari","Albert"} };
int[,] számok = { {1, 2}, {3, 4}, {5, 6} };
string[,] kapocs = { {"Miki", "Ani"}, {"Mari", "Albert"} };
```

Használat:

```
numbers[1,1] = 667;
```

### *Vektorok vektora:*

A más nyelvekbeli analógiát tekintve, a multidimenziós vektorban minden sorban azonos darabszámú elem van, ez tehát a szabályos, négyzetes mátrix stb. definíciónak felel meg. A vektorok vektora pedig valójában egy mutató vektor-definíció, ahol először megmondjuk, hogy hány elemű a mutató vektor, majd ezután egyenként meghatározzuk, hogy az egyes elemek milyen vektorokat jelentenek.

Példa:

```
byte[][] meres = new byte[5][];
for (int x = 0; x < meres.Length; x++)
{
    meres[x] = new byte[4];
}

int[][] számok= new int[2][]
{
    new int[] {3,2,4},
    // ez a három elemű vektor lesz a számok első vektora
    new int[] {5,6}
    // ez a két elemű vektor lesz a számok második vektora
};

int[][] numbers =
new int[][] { new int[] {2,3,4}, new int[] {5,6,7,8,9} };
int[][] numbers = { new int[] {2,3,4}, new int[] {5,6,7,8,9} };
```

#### IV. Összetett adattípusok

---

Használat:

```
numbers[1][1] = 5;
```

Természetesen mindkét esetben – multidimenziós, vektorok vektora – nemcsak kétdimenziós esetről beszélhetünk, hanem tetszőleges méretű vektorról.

Példa:

```
int[, ,] harom = new int[4,5,3];
```

Lehetőségünk van a kétféle vektordefiníció kevert használatára is. A következő példa egy olyan egyszerű vektort definiál, aminek minden eleme  $3 \times 2$  dimenziós szabályos mátrix.

Példa:

```
int[,] [, ,] mix;
```

Ezek után írjunk egy példaprogramot, amely bemutatja a korábban megbeszélt vektorjellemzők használatát:

Példa:

```
//vektor.cs
using System;

class vektorpelda
{
    public static void Main()
    {
        // Sima vektordefiníció, a definíció pillanatában
        // 5 elemű vektor lesz
        // A vektorelemeknek külön nem adtunk kezdőértéket,
        // ezért azok 0 értéket kapnak.
        int[] numbers = new int[5];

        // Kétdimenziós vektor, sima 5x4-es szöveges,
        //ezen elemek inicializálás hiányában null értéket kapnak.
        string[,] names = new string[5,4];

        // Vektorok vektora, ezt az irodalom gyakran
        // "jagged array", (csipkés, szaggatott vektor) néven említi
        byte[,] scores = new byte[5][];

        // Az egyes vektorok definiálása
        for (int i = 0; i < scores.Length; i++)
        {
            scores[i] = new byte[i+3]; // elemről elemre nő az elemszám.
        }
        // Egyes sorok hosszának kiírása
```

```
        for (int i = 0; i < scores.Length; i++)
        {
            Console.WriteLine("Az {0} sor hossza: {1}", i, scores[i].Length);
            // A {} jelek közötti számmal hivatkozhatunk az első paraméter
            // utáni további értékekre. {0} jelenti az i változót.
            // Használata nagyon hasonlít a C printf használatához.
        }
    }
}
```

A program futása után a következő eredményt kapjuk:

```
Az 0 sor hossza: 3
Az 1 sor hossza: 4
Az 2 sor hossza: 5
Az 3 sor hossza: 6
Az 4 sor hossza: 7
```

Példa:

```
string honapnev(int n)
{
    string[]név={"Nem létezik","Január",
                "Február","Március",
                "Április","Május","Június",
                "Július","Augusztus",
                "Szeptember","Október",
                "November","December"};
    return((n<1)|| (n>12)) ? név[0] : név[n];
}
```

Az iménti példa meghatározza egy tetszőleges sorszámú hónaphoz annak nevét.

## **IV.2. Struktúra**

Gyakran szükségünk van az eddigiektől eltérő adatszerkezetekre, amikor a leírni kívánt adatunkat nem tudjuk egy egyszerű változóval jellemezni. Elég, ha a talán legkézenfekvőbb feladatot tekintjük: hogyan tudjuk a sík egy pontját megadni? Egy lehetséges megadási mód, ha a pontot mint a sík egy derékszögű koordináta-rendszerének pontját tekintem, és megadom az X és Y koordinátákat.

A struktúra-definiálás az ilyen feladatok megoldása esetén lehetővé teszi, hogy az összetartozó adatokat egy egységként tudjuk kezelni.

A struktúra-definíció formája a következő:

#### IV. Összetett adattípusok

---

```
struct név {  
    hozzáférés típus mezőnevek;  
    ...  
    hozzáférés függvénydefiníció ;  
};
```

Hivatkozás egy mezőre: *változónév.mezőnév*

A struktúrákon azonos típus esetén az értékadás elvégezhető, így ha például *a* és *b* azonos típusú struktúra, akkor az *a=b* értékadás szabályos, és az *a* minden mezője felveszi *b* megfelelő mezőértékeit.

Példa:

```
struct pont  
{  
    int x;  
    int y;  
};  
...  
pont a;  
...  
struct egy  
{  
    string name;  
    int kor;  
};  
egy[] c=new egy[10]; // 10 elemű struktúra vektor
```

A fenti definíció teljesen jó, csak a hozzáférési szint megadásának hiányában minden mező privát, azaz a struktúra mindkét adata csak belülről látható. A struktúra alapértelmezett mezőhozzáférése privát (*private*). Minden taghoz külön meg kell adni a hozzáférési szintet.

Struktúra esetén a megengedett hozzáférési szintek:

private	csak struktúrán belülről érhető el
public	bárki elérheti ezt a mezőt
internal	programon belülről (assembly) elérhető

Példa:

```
struct személy  
{  
    public string név;
```



```
        public int kor;  
    };
```

Az egyes mezőkre hivatkozás formája:

Példa:

```
    személy st=new személy();  
    System.WriteLine( st.kor); // kor kiírása
```

Struktúradefiníció esetén a nyelv nem csak adatmező, hanem függvénymező definiálást is megenged. Azt a függvénymezőt, aminek ugyanaz a neve, mint a struktúrának, konstruktornak nevezzük. Paraméter nélküli konstruktor nem definiálható, azt mindig a környezet biztosítja. A struktúra értéktípusú adat, így a vermen és nem a dinamikus memóriában jön létre. Általában elmondható, hogy a struktúra majdnem úgy viselkedik, mint egy osztály, de funkcionalitását tekintve megmarad a különböző típusú adatok tárolásánál.

A struktúrákkal kapcsolatosan érdemes megjegyezni három alapvető tulajdonságot:

- Egy struktúra adatmezőt a definiálás pillanatában nem inicializálhatunk.

Példa:

```
    struct alma  
    {  
        string nev="jonatán"; // fordítási hiba  
        ...  
    }
```

- Struktúra adatmezőket a *default* konstruktor nem inicializál. A kezdőérték beállításáról, ha szükséges, saját konstruktorral vagy egyéb beállítási móddal kell gondoskodni.

Példa:

```
    struct pont  
    {  
        public int x,y;  
    }  
    pont p=new pont();  
    p.x=2; p.y=4;
```

- Bár a struktúra érték típusú, azért a *new* operátorral kell biztosítani a saját konstruktorral történő inicializációt. Ez ebben az esetben a vermen fog elhelyezkedni.

#### IV. Összetett adattípusok

---

Befejezésül a struktúra definiálására, használatára, struktúra vektorra nézzünk egy teljesebb példát:

Példa:

```
using System;
struct struktúra_példa
{
    public int kor;
    public string név;
}
class struktúra_használ
{
    public static void Main()
    {
        struktúra_példa sp=new struktúra_példa();
        sp.kor=5;
        sp.név="Éva";
        // struktúra_példa vektor
        struktúra_példa [] spv=new struktúra_példa[5];
        int i=0;
        // beolvasás
        while(i<5)
        {
            spv[i]=new struktúra_példa();
            Console.WriteLine("Kérem az {0}. elem nevét!",i);
            string n=Console.ReadLine();
            spv[i].név=n;
            Console.WriteLine("Kérem az {0}. elem korát!",i);
            n=Console.ReadLine();
            spv[i].kor=Convert.ToInt32(n);
            i++;
        }
        // kiírás
        for(i=0;i<5;i++)
        {
            Console.WriteLine(spv[i].név);
            Console.WriteLine(spv[i].kor);
        }
    }
}
```

### IV.3. Feladatok

1. Mit nevezünk tömbnek?

2. Milyen tömbök létrehozását támogatja a C# nyelv?
3. Mi a különbség a tömb és a struktúra között?
4. Határozzuk meg egy tömb legnagyobb elemét!
5. Ábrázoljuk struktúra típussal az alma legjellemzőbb adatait (név, szín, méret)! Készítsünk 5 elemű *alma* vektort!

# V. Utasítások

A programvezérlés menetét az utasítások szekvenciája szabja meg. Ahogy korábban is láttuk, az operációs rendszer a *Main* függvénynek adja át a vezérlést, majd a függvénytörzs egymás után következő utasításait (szekvencia) hajtja végre, ezután tér vissza a vezérlés az operációs rendszerhez.

Az utasítások fajtái:

- összetett utasítás
- kifejezés utasítás
- elágazás utasítás
- ciklus utasítás
- ugró utasítás

## V.1. Összetett utasítás

Ahol utasítás elhelyezhető, ott szerepelhet összetett utasítás is.

Az összetett utasítás vagy blokk a `{}` zárójelek között felsorolt utasítások listája. Blokkon belül változók is deklarálhatók, amelyek a blokkban lokálisak lesznek. Blokkok tetszőlegesen egymásba ágyazhatók.

## V.2. Kifejezés utasítás

Az utasítás formája:

kifejezés ;
-------------

Az utasítás speciális formája az, amikor a kifejezés elmarad. Az ilyen utasítást `(;)` üres vagy nulla utasításnak nevezzük. Ennek természetesen általában nagyon sok értelme nincs. Az alábbi példa egy változó értékét növeli egyesével egy ciklusban.

Példa:

```
int k=5;
for(i=0; i<10; i++)
    k++;
```

### V.3. Elágazás utasítás

Az elágazások két típusa használható, a kétfelé és a sokfelé elágazó utasítás. A kétfelé elágazó utasítás formája:

```
if (kifejezés) utasítás;  
if (kifejezés) utasítás; else utasítás;
```

Először a kifejezés kiértékelődik, majd annak igaz értéke esetén a kifejezés utáni utasítás, hamis értéke esetén az *else* – ha van – utáni utasítás hajtódik végre.

Egymásba ágyazás esetén az *else* ágat a legutóbbi *else* nélküli *if*hez tartozónak tekintjük.

Példa:

```
if (c=='a')  
    Console.WriteLine("Ez az a betű");  
else  
    Console.WriteLine("Nem az a betű");
```

Ha az igaz ágon összetett utasítást használunk, akkor utána pontosvessző nem kell, illetve ha mégis van, az az *if* lezárását jelentené, és hibás lenne az utasításunk az *if* nélküli *else* használata miatt.

Példa:

```
if (c=='a')  
    { Console.WriteLine("Ez az a betű");}  
else  
    Console.WriteLine("Nem az a betű");
```

A többirányú elágazás utasítása, a *switch* utasítás formája:

```
switch (kifejezés) {  
    case érték1: utasítás1 ;  
    case érték2: utasítás2 ;  
    ...  
    default: utasítás ;  
};
```

## V. Utasítások

---

A *switch* utáni kifejezés nem csak egész értékű lehet, hanem szöveg (*string*) is. Ha a kifejezés értéke a *case* után megadott értékkel nem egyezik meg, akkor a következő *case* utáni érték vizsgálata következik. Ha egy *case* utáni érték azonos a kifejezés értékével, akkor az ez utáni utasítás végrehajtódik. Ha egy *case* ágban nincs egyértelmű utasítás arra vonatkozóan, hogy hol folytatódjon a vezérlés, akkor fordítási hibát kapunk.

A C++ nyelvet ismerők tudhatják, hogy abban a nyelvben egy *case* ág végét nem kellett vezérlésátadással befejezni, és ebből gyakran adódtak hibák.

Minden elágazáságot, még a *default* ágot is, kötelező valamilyen vezérlésátadással befejezni! (*break*, *goto*, *return*...)

Példa:

```
switch (parancs)
{
    case "run":
        Run();
        break;
    case "save":
        Save();
        break;
    case "quit":
        Quit();
        break;
    default:
        Rossz(parancs);
        break;
}
```

A *case* belépési pont után csak egy érték adható meg, intervallum vagy több érték megadása nem megengedett, hiszen ezek az 'értékek' gyakorlatilag egy címke szerepét töltik be. Ha két értékhez rendeljük ugyanazt a tevékenységet, akkor két címkét kell definiálni.

Példa:

```
...
switch (a)
{
    case 1:
    case 2:
        Console.WriteLine("Egy és kettő esetén...");
        break;
    default: Console.WriteLine("Egyébként...");
        break;
};
...
```

## V.4. Ciklus utasítás

A ciklus utasításnak négy formája alkalmazható a C# nyelvben, ezek a *while*, a *do*, a *for* és a *foreach* ciklus utasítások.

### V.4.1. „while” utasítás

A *while* ciklus utasítás formája:

*while* (kifejezés) utasítás;

Először a zárójelben lévő kifejezés kiértékelődik, ha értéke igaz, akkor a zárójeles kifejezést követő utasítás – amit szokás ciklusmagnak nevezni –, végrehajtódik. Ezután újból a kifejezés kiértékelése következik, igaz érték esetén pedig a ciklusmag végrehajtása. Ez az ismétlés addig folytatódik, amíg a kifejezés hamis értéket nem ad.

Mivel az utasítás először kiértékeli a kifejezést és csak utána hajtja végre a ciklusmagot (ha a kifejezés igaz értéket adott), ezért a *while* ciklus utasítást előtesztelő ciklusnak is szokás nevezni.

Példa:

```
while(true)
{
    Console.WriteLine("Végtelen ciklus!");
    Console.WriteLine("Ilyet ne nagyon használj!");
}
...
static void Main()                // betűnkénti kiírás
{
    string szöveg[]="Szöveg!";
    int i=0;
    while(i<szöveg.Length)        // a string végéig
    {
        Console.WriteLine(szöveg[i]);
        i++;
    }
}
```

A második példában a ciklusmagban használhattam volna egy utasítást is, kihasználva a ++ operátor postfix alakját az alábbi módon:

```
while(i<szöveg.Length)
    Console.WriteLine(szöveg[i++]);
```

## V. Utasítások

---

### V.4.2. „do” utasítás

Az utasítás formája:

do utasítás; while (kifejezés) ;

A ciklusmag – a *do* és a *while* közötti rész – végrehajtása után kiértékeli a kifejezést, és amíg a kifejezés igaz értéket ad, megismétli a ciklusmag végrehajtását. A *do* ciklust szokás hátultesztelő ciklusnak is nevezni.

Példa:

```
string név;  
do  
{  
    Console.WriteLine("Ki vagy?");  
    név=Console.ReadLine();  
}  
while (név!="Zoli");  
Console.WriteLine("Szia {0}!",név);  
  
do  
    Console.WriteLine("Biztos egyszer lefut!");  
while(false);
```

### V.4.3. „for” utasítás

Az utasítás formája:

for (kifejezés1; kifejezés2; kifejezés3) utasítás;

Ez az utasítás egyenértékű a következő alakkal:

```
kifejezés1;  
while (kifejezés2) {  
    utasítás;  
    kifejezés3;  
};
```

Mindegyik kifejezés elmaradhat. Ha *kifejezés2* is elmarad, akkor *while* (*true*)-ként tekinti a ciklust. A kifejezések közötti pontosvessző nem maradhat el.

A *kifejezés1* típusdefiníciós kifejezés utasítás is lehet. Nagyon gyakran látható forráskódban az alábbi forma:



Példa:

```
for (int i=0; i<10; i++)
    Console.WriteLine("Hajrá Fradi!");
```

Ez a fajta ciklus használat felel meg például a Basic *For...Next* ciklusának.

#### V.4.4. „foreach” utasítás

Az utasítás formája:

foreach (azonosító in vektor) utasítás;

Ez az utasítás egyenértékű a vektor-elemeken történő végiglépdeléssel. Az azonosító, mint ciklusváltozó felveszi egyenként e vektor elemeit.

Példa:

```
public static void Main()
{
    int paros = 0, paratlan = 0;
    int[] v = new int [] {0,1,2,5,7,8,11};
    foreach (int i in v)
    {
        if (i%2 == 0)
            paros++;
        else
            paratlan++;
    }
    Console.WriteLine("Találtam {0} páros, és {1} páratlan
                                                                számot.",paros, paratlan);
}
```

A *foreach* ciklus az általunk definiált vektorokon kívül az ehhez hasonló könyvtári adatszerkezeteken is (*ArrayList*, *Queue* stb.) jól használható. Ezzel kapcsolatos példát a *Helyi könyvtárak használata* fejezetben láthatunk.

A *foreach* utasítás nemcsak közvetlen vektortípuson, mint például a fenti *v* vektoron alkalmazható, hanem minden olyan „végigjárható” típuson, amely az alábbi feltételeknek megfelel:

- A típus rendelkezzen egy *GetEnumerator()* függvénnyel, aminek eredményül kell adni azt a típust, amelyen típusú elemeken lépdelünk végig.
- Definiáljunk egy *MoveNext()* függvényt, amelyik az indexet aktuálisra állítja, és logikai visszatéréssel megadja, hogy van-e még maradék elem.
- Definiáljunk egy *Current* tulajdonságot, ami az aktuális indexű elemét adja eredményül.

## V. Utasítások

---

Példa:

```
using System;
public class adatsor
{
    int[] elemek;

    public adatsor()
    {
        elemek = new int[5] {12, 44, 33, 2, 50};
    }

    public vektor GetEnumerator()
    {
        return new vektor(this);
    }

    // Az "enumerator", class definiálás:
    public class vektor
    {
        int Index;
        adatsor a;
        public vektor(adatsor ad)
        {
            a = ad;
            Index = -1;
        }

        public bool MoveNext()
        {
            Index++;
            return (Index < a.elemek.GetLength(0));
        }
        public int Current
        {
            get
            {
                return (a.elemek[Index]);
            }
        }
    }
}

public class MainClass
{
    public static void Main()
    {
        adatsor adatok = new adatsor();
    }
}
```

```
        Console.WriteLine("Az adatsor elemei:");

        // elemek kiírása
        foreach (int i in adatok)
        {
            Console.WriteLine(i);
        }
    }
}

/*Eredmény:
12
44
33
2
50*/
```

## V.5. Ugró utasítások

### V.5.1. „break” utasítás

Az utasítás formája:

<code>break;</code>
---------------------

Hatására befejeződik a legbelső *while*, *do*, *for* vagy *switch* utasítás végrehajtása. A vezérlés a következő utasításra adódik.

Példa:

```
int i=1;
while(true)                // látszólag végtelen ciklus
{
    i++;
    if (i==11) break;       // Ciklus vége
    Console.WriteLine( i);
}
```

A *break* a többirányú elágazás (*switch*) utasításban is gyakran használt, így kerülhetjük el, hogy a nem kívánt *case* ágak végrehajtódjanak.

### V.5.2. „continue” utasítás

Az utasítás formája:

<code>continue;</code>
------------------------

## V. Utasítások

---

Hatására a legbelső *while*, *for*, *do* ciklus utasításokat vezérlő kifejezések kerülnek kiértékelésre. (A ciklus a következő ciklusmag végrehajtásához készül.)

Példa:

```
int i=1;
while(true)                // 10 elemű ciklus
{   i++;
    if (i<=10) continue;   // következő ciklusmag

    if (i==11) break;       // Ciklus vége
    Console.WriteLine( i);
}
```

A fenti példában a *continue* utasítás miatt a *Console.WriteLine(i)* utasítás egyszer sem hajtódik végre.

### V.5.3. „return” utasítás

Az utasítás formája:

<pre>return ; return kifejezés; return (kifejezés);</pre>
---

A vezérlés visszatér a függvényből, a kifejezés értéke a visszaadott érték.

### V.5.4. „goto” utasítás

Az utasítás formája:

<pre>goto címke;</pre>
------------------------

A vezérlés arra a pontra adódik, ahol a *címke*: található.

Példa:

```
goto tovább;
Console.WriteLine("Ezt a szöveget sohase írja ki!");
tovább;;
...
int i=1;
switch (i)
{
    case 0:
        nulla();
```

```
        goto case 1;
case 1:
    egy();
    goto default;
default:
    valami();
    break;
}
```

A *goto* utasításról zárasképpen meg kell jegyezni, hogy a strukturált programkészítésnek nem feltétlenül része ez az utasítás, így használata sem javasolt. A könyv későbbi példaprogramjaiban sem fordul elő egyszer sem ez az utasítás.

#### V.5.5. „using” utasítás

A *using* kulcsszó kétféle nyelvi környezetben szerepelhet. Egyrészt ún. direktíva, könyvtári elemek, adott névtér, osztály használataként. Ennek formája:

`using névtér_vagy_osztály;`

Példa:

```
using System; // a keretrendszer fő névtérének használata
              // a C++ #include-hoz hasonlóan megmondja a
              // fordítónak, hogy ennek a névtérnek a típusait is
              // használja. Ezen típusokat azért enélkül is teljes
              // névvel (System.Console...) használhatjuk
```

A másik eset a *using* utasítás. Egy ideiglenesen használt objektum esetén a *using* utasítás után a keretrendszer automatikusan meghívja az objektum *Dispose* metódusát. A *using* utasítás alakja a következő:

`using (objektum) { utasítások;}`

Példa:

```
using (Objektumom o = new Objektumom())
{
    o.ezt_csinald();
}
```

## V. Utasítások

---

Ez a használat az alábbi kódrészletnek felel meg, a nem ismert nyelvi elemeket később ismertetjük:

```
...
Objektumom o = new Objektumom();
try
    {o.ezt_csinald();}
finally
{
    if (o != null) ((IDisposable)o).Dispose();
}
```

A *Dispose* utasítást és környezetét bővebben a destruktorokkal kapcsolatban részletezzük.

### V.5.6. „lock” utasítás

Ha egy kritikus utasítás blokk végrehajtásakor egy referencia blokkolására van szükség a biztonságos végrehajtáshoz, akkor ezt a *lock* utasítással meglehetjük.

A *lock* kulcsszó egy referenciát vár, ez jellemzően a *this*, majd utána következik a kritikus blokk. Ennek formája:

lock(ref) utasítás;
---------------------

Példa:

```
lock(this)
{
    a=5; // biztonságos
}
```

A *lock* utasításnak, ahogy láttuk, a leggyakrabban használt paramétere a *this*, ha a védett változó vagy függvényutasítás nem statikus. (Osztálpéldányok.)

Ha statikus változókat vagy függvényutasításokat szeretnénk biztosítani (lockolni), hogy egyszerre csak egy végrehajtási szál tudjon hozzáférni, akkor a *lock* referenciának az osztály típusát kell megadni.

Példa:

```
class adatok
{
    static int szamlalo=0;
    public void mentes(string s)
```

```
{
    lock(typeof(adatok))
    {
        szamlalo++;
        Console.WriteLine("Adatmentés elindul!");
        for(int i=0;i<50;i++)
        {
            Thread.Sleep(1);
            Console.Write(s);
        }
        Console.WriteLine("");
        Console.WriteLine("Adatmentés {0}.alkalommal
                                                                    befejeződött!",szamlalo);
    }
}
```

A példa bővebb magyarázata, teljes környezetbeli alkalmazása a *IX. rész Párhuzamos programvégrehajtás* fejezetében található.

## V.6. Feladatok

1. Milyen típusú adat szerint készíthetünk többirányú (*switch*) elágazást?
2. Milyen előtesztelő és hátulatesztelő ciklusokat használhatunk?
3. Mire használható a *break* utasítás?
4. Írjon rövid programot, amely beolvassa egy focicsapat, adott fordulóban szerzett pontszámát (0,1,3) egy egész típusú változóba, majd felhasználva a *switch* utasítást a beolvasott érték alapján kiírja a következő szövegeket: *győzelem, döntetlen, vereség, hibás adat*.
5. Írjon programot, amelyik beolvas egy kettővel osztható, 10 és 100 közé eső egész számot! (Ha rossz értéket adnak meg, akkor addig folytassa a beolvasást, amíg a feltételeknek megfelelő számot nem sikerül megadni!)

# VI. Függvények

## VI.1. A függvények paraméterátadása

Ha egy függvénynek adatot adunk át paraméterként, akkor alapvetően két különböző esetről beszélhetünk. Egy adat érték szerint és hivatkozás szerint kerülhet átadásra. Az első esetben valójában az eredeti adatunk értékének egy másolata kerül a függvényhez, míg a második esetben az adat címe kerül átadásra.

### VI.1.1 Érték szerinti paraméterátadás

Általánosan elmondhatjuk, hogyha nem jelölünk semmilyen paraméterátadási módszert, akkor érték szerinti paraméterátadással dolgozunk. Ekkor a függvény paraméterében, mint formális paraméterben, a függvény meghívásakor a hívóérték másolata helyezkedik el.

Tekintsük meg a következő maximum nevű függvényt, aminek az a feladata, hogy a kapott két paramétere közül a nagyobbbat adja vissza eredményként.

Példa:

```
class maxfv
{
    static int maximum(int x,int y)
    {
        return (x>y?x:y);
    }

    static void Main()
    {
        Console.WriteLine(maximum(4,3));
    }
}
```

Az így megvalósított paraméterátadást érték szerinti paraméterátadásnak nevezzük, a *maximum* függvény két (*x* és *y*) paramétere a híváskor megadott két paramétert kapja értékül. Érték szerinti paraméterátadásnál, híváskor konstans érték is megadható.

A függvény hívásának egy sajátos esete az, mikor egy függvényt saját maga hív meg. Szokás ezt rekurzív függvényhívásnak is nevezni. Erre példaként nézzük meg a klasszikus faktoriálisszámoló függvényt.



Példa:

```
class faktor
{
    static int faktorialis(int x)
    {
        return (x<=1?1: (x* faktorialis(x-1)));
    }
    static void Main()
    {
        Console.WriteLine(faktorialis(4));
    }
}
```

### VI.1.2. Referencia (cím) szerinti paraméterátadás

Amikor egy függvény paramétere nem a változó értékét, hanem a változó tárolási helyének címét, hivatkozási helyét kapja meg a függvény meghívásakor, akkor a paraméterátadás módját cím szerinti paraméterátadásnak nevezzük.

A cím szerinti paraméterátadást gyakran hivatkozás (*reference*) szerinti paraméterátadásnak is szokás nevezni.

Ha a paraméternév elé a *ref* (hivatkozás) kulcsszót beszurjuk, akkor a hivatkozás szerinti paraméterátadást definiáljuk. Ezt a kulcsszót be kell szúrunk a függvénydefinícióba és a konkrét hívás helyére is!

Példaként írjunk olyan függvényt, ami a kapott két paraméterének értékét felcseréli.

Példa:

```
...
// a ref kulcsszó jelzi, hogy referencia
// paramétereket vár a függvény
void csere(ref int x, ref int y)
{
    int segéd=x;
    x=y; y=segéd;
}
static void Main()
{
    int a=5, b=6;
    Console.WriteLine("Csere előtt: a={0}, b={1}.", a , b);
    // függvényhívás, a ref kulcsszó itt is kötelező
    csere(ref a,ref b);
    Console.WriteLine("Csere után: a={0}, b={1}.", a , b);
}
```

A *csere(a,b)* hívás megcseréli a két paraméter értékét, így az *a* változó értéke 6, míg *b* értéke 5 lesz.

## VI. Függvények

---

### VI.1.3. Függvényeredmény paramétere

A referencia szerinti paraméterátadáshoz hasonlóan, a függvénybeli változó értéke kerül ki a hívó változóba. A különbség csak az, hogy ebben a változóban a függvény nem kap értéket.

Az eredmény paramétert az *out* kulcsszóval definiálhatjuk. A híváskor is a paraméter elé ki kell írni az *out* jelzőt. A használata akkor lehet hasznos, amikor egy függvénynek egynél több eredményt kell adnia.

Példa:

```
using System;
public class MyClass
{
    public static int TestOut(out char i)
    {
        i = 'b';
        return -1;
    }

    public static void Main()
    {
        char i;          // nem kell inicializálni a változót
        Console.WriteLine(TestOut(out i));
        Console.WriteLine(i);
    }
}
```

Képernyő eredmény:

-1  
B

### VI.1.4. Tömbök paraméterátadása

A nyelv a tömböt annak nevével, mint referenciával azonosítja, ezért egy tömb paraméterátadása nem jelent mást, mint a tömb referenciájának átadását. Egy függvény természetesen tömböt is adhat eredményül, ami azt jelenti, hogy az eredmény egy tömbreferencia lesz. Egy tömb esetében sincs másról szó, mint egyszerű változók esetében, a tömbreferencia – egy referenciaváltozó –, érték szerinti paraméterátadásáról.

Az elmondottak illusztrálására lássuk a következő szemantikus példát.

Példa:

```
void módosít(int[] vektor)
{
    // a vektort, mint egy referenciát kapjuk paraméterül
    // ez érték szerint kerül átadásra, azaz ez a referencia
    //nem változik, változik viszont a 0. tömbelem, és ez a
    // változás a hívó oldalon is látszik
    vektor[0]=5;
}
```

Ahogy a fenti példán is látható, a vektor paraméterként a referenciájával kerül átadásra. Ez érték szerinti paraméterátadást jelent. Ha egy függvényben a mutatott értéket (vektorelemet) megváltoztatom, akkor a változás a külső, paraméterként átadott vektorban is látható lesz!

A nyelvben a vektor tudja magáról, hogy hány eleme van (*Length*), ezért – ellentétben a C++ nyelvvel –, az elemszámot nem kell átadni.

Tömb esetében is alkalmazhatjuk a referencia vagy *out* paraméter átadásának lehetőségét. Ennek illusztrálására nézzük a következő példákat:

1. példa:

```
using System;
class teszt
{
    static public void feltölt(out int[] vektor)
    {
        // a vektor létrehozása, és inicializálása
        vektor = new int[5] {1, 2, 3, 4, 5};
    }

    static public void Main()
    {
        int[] vektor; // nem inicializáltuk
        // meghívjuk a feltölt függvényt:
        feltölt(out vektor);
        // A vektorelemek kiírása:
        Console.WriteLine("Az elemek:");
        for (int i=0; i < vektor.Length; i++)
            Console.WriteLine(vektor[i]);
    }
}
```

### 2. példa:

```
using System;
class Refteszt
{
    public static void feltölt(ref int[] arr)
    {
        // Ha hívó fél még még nem készítette el,
        //akkor megteesszük itt.

        if (arr == null)
            arr = new int[10];
        // néhány elem módosítás
        arr[0] = 123;
        arr[4] = 1024;
        // a többi elem értéke marad az eredeti
    }

    static public void Main ()
    {
        // Vektor inicializálása
        int[] vektor = {1,2,3,4,5};

        // Vektor átadása ref, paraméterként:
        feltölt(ref vektor);

        // Kiírás:
        Console.WriteLine("Az elemek:");
        for (int i = 0; i < vektor.Length; i++)
            Console.WriteLine(vektor[i]);
    }
}
```

## VI.2. A Main függvény paraméterei

A parancsok, programok indításakor gyakori, hogy a parancsot valamilyen paraméter(ek)rel indítjuk. Ilyen parancs például a DOS *echo* parancsa, amely paramétereit a képernyőre „visszhangozza”, vagy a *type* parancs, mely a paraméterül kapott fájl tartalmát írja a képernyőre.

Ezeket a paramétereket parancssor-argumentumoknak is szokás nevezni. Amikor elindítunk egy programot (a *Main* függvény az operációs rendszertől átveszi a vezérlést), akkor híváskor a *Main* függvénynek egy paramétere van. Ez a paraméter egy szöveges (*string*) tömb, amelynek elemei az egyes paraméterek. A paramétertömböt gyakran *args* névre keresztelik.

A parancssor-argumentumok használatára nézzük az imént már említett *echo* parancs egy lehetséges megvalósítását.

Példa:

```
static void Main(string[] args)
{
    int i=0;
    while (i<args.Length)
    {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

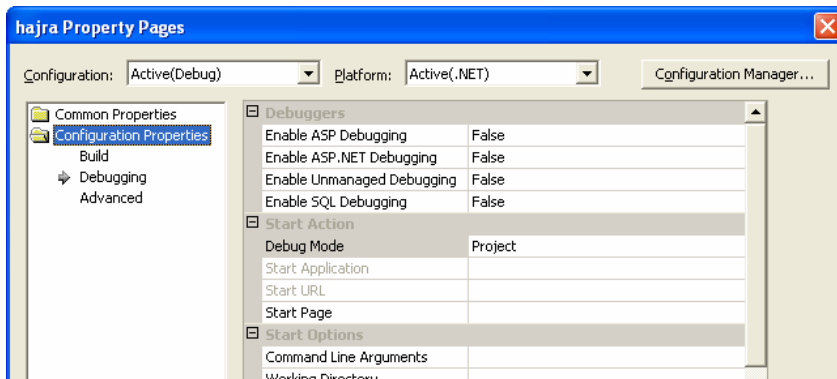
Ha a fenti programot lefordítjuk, és a neve *parancs.exe* lesz, akkor a következőképpen próbálhatjuk ki:

```
c:\parancs.exe fradi vasas újpest
```

A program futtatása után az alábbi eredményt kapjuk:

```
fradi
vasas
újpest
```

A parancssori paraméterek Visual Studio.NET környezetet használva a projekt *Tulajdonság* ablakában is megadhatók: *Configuration Properties \ Debugging \ Command Line Arguments*, ahogy az alábbi képen is látható.



9. ábra

## VI. Függvények

---

A *Main* függvényt, abban az esetben, ha a parancssori paramétereket nem akarjuk feldolgozni, a paraméterei nélkül is deklarálhatjuk:

```
static void Main()  
{...}
```

### VI.3. Függvények változó számú paraméterrel

A feladatmegoldások során gyakran előfordulhat, hogy előre nem tudjuk eldönteni, hány elemet kell a függvénynek feldolgoznia. Tehát nem tudjuk, hogy hány paraméterrel készítsük el a kívánt függvényünket. A *params* kulcsszó segítségével egy vektorparamétert definiálhatunk a függvénynek, ami ezeket a paramétereket tartalmazza majd.

Nézzünk néhány példát ilyen függvény definiálására, majd a használatára.

Példa:

```
using System;  
public class váltparaméter  
{  
  
    public static void intParaméterek(params int[] list)  
    {  
        for ( int i = 0 ; i < list.Length ; i++ )  
            Console.WriteLine(list[i]);  
        Console.WriteLine();  
    }  
  
    public static void objParaméterek(params object[] list)  
    {  
        for ( int i = 0 ; i < list.Length ; i++ )  
            Console.WriteLine(list[i]);  
        Console.WriteLine();  
    }  
  
    public static void Main()  
    {  
        intParaméterek(1, 2, 3);  
        objParaméterek(1, "fradi", "teszt", 3.5);  
        int[] myarray = new int[3] {10,11,12};  
        intParaméterek(myarray);  
    }  
}
```

Az eredmény a következő lesz:

```
1
2
3

1
fradi
teszt
3.5

10
11
12
```

Egy változó paraméterszámú függvénynek lehetnek fixen megadott paraméterei is. A fixen megadott paramétereknek kötelezően meg kell előzniük a változó paramétereket. A következő példa egy ilyen függvénydefiníciót mutat.

Példa:

```
using System
class parameters
{
    public static void valtozo(int x, params object[] list)
    {
        Console.WriteLine(x);
        foreach (object o in list)
            Console.WriteLine(o);
    }

    public static void Main()
    {
        valtozo(25, "alma", "barack", "szilva");
    }
}
```

Eredményül kapjuk a függvény paramétereinek kiírását egymás alá.

## VI.4. Függvénynevek átdefinálása

Egy-egy feladat esetén gyakorta előfordul, hogy a megoldást adó függvényt különböző típusú vagy különböző számú paraméterrel jellemezhetjük. Természetesen az ezt megoldó függvényt azonos névvel szeretnénk elkészíteni minden esetben, hisz ez lenne a legkifejezőbb. A lehetőséget gyakran függvény „overloading” névvel is megtaláljuk az irodalomban, vagy a polimorfizmus kapcsán kerül megemlítésre.

## VI. Függvények

---

Határozzuk meg két szám maximumát! Ha el szeretnénk kerülni a típuskonvertálást mondjuk egész típusról valósba és fordítva, akkor külön az egész és külön a valós számok esetére is meg kell írunk a *maximum* függvényt.

Kényelmetlen dolog lenne, ha mondjuk *egészmax* és *valósmax* néven kellene megírni a két függvényt. Mindkét esetben szeretnénk a *maximum* függvénynevet használni, hogy ne nekünk kelljen eldönteni azt, hogy az egészek vagy valósak maximumát akarjuk-e meghatározni, hanem döntse el a fordító a paraméterlista alapján automatikusan, hogy melyik függvényt is kell az adott esetben meghívni.

Példa:

```
// valós maximum függvény
double maximum(double x,double y)
{
    if (x>y)
        return x;
    else
        return y;
}

// egész maximum függvény
int maximum(int x,int y)
{
    if (x>y)
        return x;
    else
        return y;
}

...
int a=2,b=4;
double c=3.123, d=2;
Console.WriteLine(maximum(4.1,2)); // valós hívás
Console.WriteLine(maximum(4,2)); // egész hívás
Console.WriteLine(maximum(a,b)); // egész hívás
Console.WriteLine(maximum(c,d));
...
```

Meg kell jegyezni, hogy ezt a tulajdonságot sok nyelvben függványsablon (*template*) tulajdonság segítségével elegánsabban oldhatnánk meg, de a jelenlegi C# ezt nem támogatja.



## VI.5. Delegáltak, események

Ahogy korábban is szó volt róla, a biztonságosabb programkód készítésének érdekében a mutató aritmetika a C# nyelvben nem engedélyezett. Ebbe beletartozik az is, hogy a függvénymutatók sem lehetnek kivételek.

Ez utóbbi esetben viszont olyan nyelvi tulajdonságok nem implementálhatók, mint például egy menüponthoz hozzárendelt függvény, amit a menüpont választása esetén kell végrehajtani. Ezen utóbbi lehetőséget hivatott a delegált típus biztosítani. Ez már csak azért is fontos, mert többek között a Windows programozás „*callback*” jellemző paramétere is ezt használja.

A C++ környezetben ezt a lehetőséget a függvénymutató biztosítja.

A delegált valójában egy függvénytípus-definíció, aminek alakja a következő:

delegate típus delegáltnév(típus paraméternév,...);

A delegált referencia típus. Ha paramétert is megadunk, akkor a paraméter nevét is meg kell adni.

Példa:

```
delegate int pelda(int x, string s);
```

Az előbbi példában tehát a *pelda* delegált típust definiáltuk. Ez olyan függvénytípus, amelyiknek egy egész és egy szöveg paramétere van, és eredményül egy egész számot ad.

Egy delegált objektumnak vagy egy osztály statikus függvényét, vagy egy osztály példányfüggvényét adjuk értékül. Delegált meghívásánál a hagyományos függvényhívási formát használjuk. Ezek után nézzünk egy konkrét példát:

Példa:

```
using System;
class proba
{
    public static int negyzet(int i)
    {
        return i*i;
    }
    public int dupla(int i)
    {
        return 2*i;
    }
}
```

## VI. Függvények

---

```
class foprogram
{
    delegate int emel(int k); // az emel delegált definiálása

    public static void Main()
    {
        emel f=new emel(proba.negyzet);
                                // statikus függvény lesz a delegált
        Console.WriteLine(f(5)); // eredmény: 25
        proba p=new proba();
        emel g=new emel(p.dupla);
                                // normál függvény lesz a delegált
        Console.WriteLine(g(5)); // eredmény: 10
    }
}
```

A delegáltakat a környezet két csoportba sorolja. Ha a delegált visszatérési típusa *void*, akkor egy delegált több végrehajtandó függvényt tartalmazhat (*multicast*, összetett delegált), ha nem *void* a visszatérési típus, mint a fenti példában, akkor egy delegált csak egy végrehajtandó függvényt tud meghívni (*single cast*, egyszerű delegált).

Az egyszerű és összetett delegáltakra nézzük a következő példát:

Példa:

```
using System;
class proba
{
    public static int negyzet(int i)
    {
        return i*i;
    }
    public int dupla(int i)
    {
        return 2*i;
    }
    public int tripla(int i)
    {
        return 3*i;
    }
    public void egy(int i)
    {
        Console.WriteLine(i);
    }
    public void ketto(int i)
    {
        Console.WriteLine(2*i);
    }
}
```

```

class foprogram
{
    delegate int emel(int k);
    delegate void meghiv(int j);

    public static void Main()
    {
        emel f=new emel(proba.negyzet);
        Console.WriteLine(f(5));
        proba p=new proba();
        emel g=new emel(p.dupla);
        Console.WriteLine(g(5));
        emel h=new emel(p.tripla);
        g+=h;           //g single cast, g a tripla lesz
        Console.WriteLine(g(5));           // eredmény: 15
        meghiv m=new meghiv(p.egy);       // multicast delegált
        m+=new meghiv(p.ketto);
        m(5);           // delegált hívás, eredmény 5,10
    }
}

```

Ha függvénytípus (delegált) deklarálunk, akkor értékadás esetén a mutató típusa határozza meg, hogy a fordítónak melyik aktuális függvényt is kell az azonos nevek közül választania.

```

delegate double vmut(double,double);
// vmut olyan delegált amelyik egy valós értéket visszaadó,
// és két valós paramétert váró függvényt jelent

delegate int emut(int,int);
// emut olyan delegált, amelyik egy egész értéket visszaadó
// , és két egész paramétert váró függvényt jelent
vmut mut=new vmut(maximum); // valós hívás
Console.WriteLine(mut(3,5));

```

A *multicast* delegált (típus) definiálási lehetőséget, igazítva az igényeket az eseményvezérelt programozási környezethez (mind a Windows, mind az X11 ilyen), az egyes objektumokhoz, típushoz úgy kapcsolhatjuk, hogy esemény típusú mezőt adunk hozzájuk. Ezt az alábbi formában tehetjük meg:

public event meghiv esemeny;
------------------------------

ahol a *meghiv* összetett delegált. Az esemény objektum kezdő értéke *null* lesz, így az *automatikus meghívás (esemeny(...))* nem ajánlott!

Az eseményhez záráskeppen meg kell jegyezni, hogy az egész keretrendszer a felhasználói tevékenységet ehhez a lehetőséghez rendeli akkor, amikor klasszikus Windows alkalmazást akarunk készíteni.

Példaként nézzük meg egy Windows alkalmazás esetén a form tervező által generált kódot. A form (*this*) objektuma egy választómező (*ComboBox*), *ampl* névre hallgat. Ennek könyvtári eseménye, a *SelectedIndexChanged* végrehajtódik (a keretrendszer hajtja végre), ha módosítunk a választáson.

Mikor ehhez az eseményhez egy eseménykezelő függvényt definiálunk *ampl\_SelectedIndexChanged* néven, akkor az alábbi sort adja a programhoz a tervező:

Példa:

```
this.ampl.SelectedIndexChanged += new  
    System.EventHandler(this.ampl_SelectedIndexChanged);
```

## VI.6. Feladatok

1. Milyen paraméterátadási módokat ismer?
2. Mikor definiálhatunk azonos névvel függvényeket egy osztályban?
3. Hogyan kell változó paraméterszámú függvényt használni?
4. Írjon egy függvényt, amely a paraméterként megadott néhány név közül a leghosszabb hosszat adja meg!
5. Definiáljon kupaforduló eseményt, melyekre a szurkolók jelentkezhetnek. Kupaforduló esetén hívjuk meg a feliratkozott szurkolókat a mérkőzésre!

## VII. Osztályok

C#-ban az osztályok a már ismert összetett adatszerkezetek (struktúrák) egy természetes kiterjesztése. Az osztályok nemcsak adattagokat, hanem operátorokat, függvényeket is tartalmazhatnak. Az osztályok használata esetén általában az adatmezők az osztály által éppen leírni kívánt esemény „állapotjelzői”, míg a függvénymezők az állapotváltozásokat írják le, felhasználói felületet biztosítanak. A függvénymezőket gyakran metódusoknak is nevezi a szakirodalom. Osztálymezőket lehetőségünk van zárttá nyilvánítani (*encapsulation*), ezzel biztosítva az osztály belső „harmóniáját”. Ezek az osztálymezők általában az „állapotjelző” adatmezők.

Új igények, módosítások esetén lehetőségünk van az eredeti osztály megtartása mellett, abból a tulajdonságok megőrzésével egy új osztály származtatására, amely *öröklí* (*inheritance*) az ősoosztály tulajdonságait (az osztályok elemeit). A C++ nyelvben lehetőségünk van arra, hogy több osztályból származtassunk utódosztályt (*multiple inheritance*), esetünkben ezt a jellemzőt a CLR nem támogatja, így ennek a fejlesztőkörnyezetnek egyik nyelve sem támogatja a többszörös öröklés lehetőségét. Lehetőségünk van *interface* definícióra, amiket egy osztály implementálhat. Egy osztály több *interface*-t is implementálhat.

Az öröklés lehetőségének a gyakorlatban legelőször jelentkező haszna talán az, hogy a programunk forráskódja jelentősen csökkenhet.

Egy osztálytípusú változót, az osztály megjelenési alakját gyakran objektumnak is szokás nevezni.

Az osztályok használatához néhány jótanácsot, a nyelv „jobbkez-szabályát” a következő pontokban foglalhatjuk össze. Ezek a tanácsok természetesen nem a nyelv tanulási időszakára, hanem a későbbi, a nyelvben történő programozási feladatokra vonatkoznak elsősorban.

- Ha egy programelem önálló értelmezéssel, feladattal, tulajdonságokkal rendelkezik, akkor definiáljuk ezt az elemet önálló osztályként.
- Ha egy programrész adata önálló objektumként értelmezhető, akkor definiáljuk őt a kívánt osztálytípus objektumaként.
- Ha két osztály közös tulajdonságokkal rendelkezik, akkor használjuk az öröklés lehetőségét.
- Általában is elmondható, hogy ha a programokban az osztályok közös vonásokkal rendelkeznek, akkor törekedni kell univerzális bázisosztály létrehozására. Gyakran ezt *absztrakt bázisosztálynak* is nevezzük.
- Az osztályok definiálásakor kerüljük a „nyitott” (publikus) adatmezők használatát.

## VII.1. Osztályok definiálása

Az osztályok deklarációjának alakja a következőképpen néz ki:

```
osztálykulcsszó osztálynév [: szülő, ...]  
{  
    osztálytaglista  
};
```

Az osztálykulcsszó a *class*, *struct* szó lehet. Az osztálynév az adott osztály azonosító neve. Ha az osztály valamely bázisosztályból származik, akkor az osztálynév, majd kettőspont után az ősoosztály felsorolása történik.

Az osztályok tagjainak öt elérési szintje lehet:

- private
- public
- protected
- internal
- protected internal.

A *private* tagokat csak az adott osztályon belülről érhetjük el.

Az osztályok *publikus* mezőit bárhonnán elérhetjük, módosíthatjuk.

A *protected* mezők az osztályon kívüliek számára nem elérhetőek, míg az utódosztályból igen.

Az *internal* mezőket a készülő program osztályaiból érhetjük el.

A *protected internal* elérés valójában egy egyszerű vagy kapcsolattal megadott hozzáférési engedély. A mező elérhető a programon belülről, vagy az osztály utódosztályából! (Egy osztályból természetesen tudunk úgy utódosztályt származtatni, hogy ez nem tartozik az eredeti programhoz.)

Ha az „osztály” definiálásakor a *struct* szót használjuk, akkor abban elsősorban adatok csoportját szeretnénk összefogni, ezért gyakran tanácsként is olvashatjuk, hogy ha hagyományos értelemben vett struktúrát, mint összetett adatszerkezetet szeretnénk használni, akkor azt „*struct típusú osztályként*” definiáljuk.

Az egyes mezőnevekre való hivatkozás ugyanúgy történik, ahogy korábban a struktúrák esetében.

Az osztálydefiníció alakjának ismeretében nézzünk egy konkrét példát!

Példa:

```
...
class első {
    private int x;                // az x mező privát
    public void beállít(int mennyi)
        { x=mennyi;}            // függvénymező
    public int kiolvas()
        { return x;}
}
```

Mivel az *x* az osztály privát mezője, ezért definiáltunk az osztályba két függvénymezőt, amelyek segítségével be tudjuk állítani, illetve ki tudjuk olvasni az *x* értékét. Ahhoz, hogy ezt a két függvényt az osztályon kívülről el tudjuk érni, publikusnak kellett őket deklarálni.

```
static void Main() {
    első a = new első(); // legyen egy a nevű első típusú
                          // osztályunk
    a.x=4;                // hiba!!! x privát mező
    a.beállít(7);         // az x mező értékét 7-re állítjuk
    Console.WriteLine(a.kiolvas());
}
```

Osztályt egy osztályon vagy egy függvényen belül is definiálhatunk, ekkor azt belső vagy lokális osztálynak nevezzük. Természetesen ennek a lokális osztálynak a láthatósága hasonló, mint a lokális változóké.

Mielőtt a statikus mezőkről szólnánk, szót kell ejteni a *this* mutató szerepéről. Ez a mutató minden osztályhoz, struktúrához automatikusan létrejön. Tételezzük fel, hogy definiáltunk egy osztálytípust. A programunk során van több ilyen típusú objektumunk. Felvetődik a kérdés, hogy amikor az egyes osztályfüggvényeket meghívjuk, akkor a függvény végrehajtása során honnan tudja meg a függvényünk, hogy ő most éppen melyik aktuális osztályobjektumra is kell, hogy hasson? Minden osztályhoz automatikusan létrejön egy mutató, aminek a neve *this*, és az éppen aktuális osztályra mutat. Így, ha egy osztályfüggvényt meghívunk, amely valamilyen módon például a privát változókra hat, akkor az a függvény a *this* mutatón keresztül tudja, hogy mely aktuális privát mezők is tartoznak az objektumhoz. A *this* mutató konkrétabb használatára a későbbiek során láthatunk példát.

Tételezzük fel, hogy egy mérőeszközt, egy adatgyűjtőrendszert egy osztállyal akarunk jellemezni. Ennek az osztálynak az egyes objektumai a mérőeszközünk meghatározott tulajdonságait képviselik. Viszont az eszköz jelerősítési együtthatói azonosak. Ezért szeretnénk, hogy az osztály erősítésmezője közös legyen, ne objektumhoz, hanem az osztályhoz kötődjön. Ezt a lehetőséget statikus mezők segítségével valósíthatjuk meg.

## VII. Osztályok

---

Statikus mezőket a *static* jelző kiírásával definiálhatunk. Ekkor az adott mező minden objektum esetében közös lesz. A statikus mező kezdőértéke inicializálás hiányában *0*, *null*, *false* lesz.

Példa:

```
class teszt {
    public static int a;          // értéke még 0
    public static string s="Katalin";
    ...
};

...
teszt.a=5;                      // statikus mező értékadása, 5 az érték
```

Statikus mezők a *this* mutatóra vonatkozó utalást nem tartalmazhatnak, hiszen a statikus mezők minden egyes objektum esetén (azonos osztálybelire vonatkozóan) közősek. Tehát például statikus függvények nem statikus mezőket nem tudnak elérni! Fordítva természetesen problémamentes az elérés, hiszen egy normál függvényből bármely statikus mező, függvény elérhető.

## VII.2. Konstruktor- és destruktor függvények

Adatok, adatszerkezetek használata esetén gyakori igény, hogy bizonyos kezdeti értékadási műveleteket, kezdőérték-állításokat el kell végezni. A korábban tárgyalt típusoknál láttuk, hogy a definícióval „egybekötve” a kezdőértékadás elvégezhető. Osztályok esetén ez a kezdőértékadás nem biztos, hogy olyan egyszerű, mint volt elemi típusok esetén, ezért ebben az esetben egy függvény kapja meg az osztály inicializálásával járó feladatot. Ez a függvény az osztály „születésének” pillanatában automatikusan végrehajtódik, és konstruktornak vagy konstruktor függvénynek nevezzük. A konstruktor neve mindig az osztály nevével azonos. Ha ilyet nem definiálunk, a keretrendszer egy paraméter nélküli automatikus konstruktort definiál az osztály számára.

A konstruktor egy szabályos függvény, így mint minden függvényből, ebből is több lehet, ha mások a paraméterei.

Az osztály referencia típusú változó, egy osztálypéldány létrehozásához kötelező a *new* operátort használni, ami egyúttal a konstruktor függvény meghívását végzi el. Ha a konstruktornak vannak paraméterei, akkor azt a típusnév után zárójelek között kell megadni.

A bináris fa feladat egyfajta megoldását tekintsük meg példaként a konstruktor használatára.



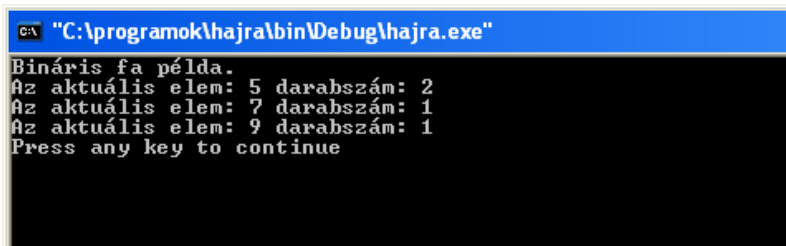
Példa:

```
using System;
class binfa
{
    int x; // elem
    int db; // elemszám
    public binfa(int i) // konstruktor
    {
        x=i; db=1;
        bal=jobb=null;
    }
    public binfa bal, jobb;
    public void illeszt(int i)
    {
        if (x==i) db++;
        else if (i<x) if (bal!=null) bal.illeszt(i);
                       else bal= new binfa(i);
                       // bal oldalra illesztettük az elemet
        else if (jobb!=null) jobb.illeszt(i);
                       else jobb= new binfa(i);
                       // jobb oldalra illesztünk
    }
    public void bejar()
    {
        // először bejárjuk a bal oldali fát
        if (bal!=null) bal.bejar();
        // ezután jön az aktuális elem
        Console.WriteLine("Az aktuális elem: {0} darabszám:
                                                                    {1}",x,db);

        // végül a jobb oldali fa következik
        if (jobb!=null) jobb.bejar();
    }
}

class program
{
    public static void Main()
    {
        binfa bf=new binfa(7);
        Console.WriteLine("Bináris fa példa.");
        bf.illeszt(5);
        bf.illeszt(9);
        bf.illeszt(5);
        bf.bejar();
    }
}
```

Futási eredményül az alábbiakat kapjuk:



```
C:\programok\hajra\bin\Debug\hajra.exe
Bináris fa példa.
Az aktuális elem: 5 darabszám: 2
Az aktuális elem: 7 darabszám: 1
Az aktuális elem: 9 darabszám: 1
Press any key to continue
```

10. ábra

### VII.2.1. Statikus konstruktor

Egy osztály, ahogy azt korábban is említettük, tartalmazhat statikus adat- és függvénymezőket is. Ezen osztálymezők a dinamikusan létrejövő objektum-példányoktól függetlenül jönnek létre. Ezeknek a mezőknek az inicializálását végezheti a statikus konstruktor. Definiálása opcionális, ha nem definiáljuk, a keretrendszer nem hozza létre.

Ha definiálunk egy statikus konstruktort, akkor annak meghívása a program indulásakor megtörténik, még azelőtt, hogy egyetlen osztálypéldányt is definiálnánk.

Statikus konstruktor elé nem kell hozzáférési módosítót, visszatérési típust írni. Ennek a konstruktornak nem lehet paramétere sem.

Példa:

```
using System;
class statikus_osztaly
{
    public static int x;
    static statikus_osztaly()
    {
        x=2;
    }
    public int getx()
    {
        return x
    }
}
```

---

```

class program
{
    public static void Main()
    {
        // valahol itt a program elején kerül meghívásra a
        // statikus_osztaly statikus konstruktora, az x értéke 2 lesz!!!
        Console.WriteLine(statikus_osztaly.x);    // 2
        statikus_osztaly s=new statikus_osztaly();
        // dinamikus példány
        Console.WriteLine(s.getx()); // természetesen ez is 2 lesz
    }
}

```

### VII.2.2. Privát konstruktor

Szükség lehet arra is – ha nem is gyakran –, hogy egy osztályból ne tudjunk egyetlen példányt se létrehozni. Természetesen ebben az esetben nem léteznek a dinamikus mezők sem, így azok definiálásának nincs is értelme. Ha nem definiálunk konstruktort, akkor a keretrendszer definiál egy alapértelmezettet, ekkor pedig lehet példányt készíteni. Így ez nem járható út.

A probléma úgy oldható meg, hogy privát konstruktort definiálunk, ami semmit nem csinál (de azt jól), illetve ha formálisan csinál is valamit, nem sok értelme van, hiszen nem tudja senki kívülről meghívni! Ebben az esetben természetesen minden tagja az osztálynak statikus.

Példa:

```

using System;

class nincs_peldanya
{
    public static int x=4;

    public static void fv1()
    {
        Console.WriteLine("halihó..");
    }

    private nincs_peldanya()
    {
        // a konstruktortörzs üres
    }
}

```

```
class program
{
    public static void Main()
    {
        // nem lehet nincs_peldanya típusú változót definiálni
        Console.WriteLine(nincs_peldanya.x);        // 4
        // csak a statikus mezőket használhatjuk
        nincs_peldanya.fv1();
    }
}
```

### VII.2.3. Saját destruktork

Ahogy egy osztály definiálásakor szükségünk lehet bizonyos inicializáló kezdeti lépésekre, úgy az osztály vagy objektum „elmúlásakor” is sok esetben bizonyos lépéseket kell tennünk. Például, ha az osztályunk dinamikusan foglal magának memóriát, akkor azt használat után célszerű felszabadítani. Azt a függvényt, amelyik az osztály megszűnésekor szükséges feladatokat elvégzi destruktornak vagy destruktork függvénynek nevezzük. Már tudjuk, hogy az osztály konstruktorának neve az osztály nevével egyezik meg. A destruktork neve is az osztály nevével azonos, csak az elején ki kell egészíteni a ~ karakterrel. A destruktork meghívása automatikusan történik, és miután az objektumot nem használjuk, a keretrendszer automatikusan lefuttatja, és a felszabaduló memóriát visszaadja az operációs rendszernek.

Konstruktornak és destruktorknak nem lehet visszaadott értéke. A destruktorknak mindig publikus osztálymezőnek kell lennie.

Ha egy osztályhoz nem definiálunk konstruktor vagy destruktort, akkor a rendszer automatikusan egy alapértelmezett, paraméter nélküli konstruktor, illetve destruktort definiál hozzá. Ha viszont van saját konstruktorunk, akkor nem ‘készül’ automatikus.

A destruktork automatikus meghívásának a folyamatát szemétgyűjtési algoritmusnak nevezzük (garbage collection, GC). Ez az algoritmus nem azonnal, az objektum blokkjának, élettartamának a végén hívja meg a destruktort, hanem akkor, amikor az algoritmus „begyűjti” ezt a szabad memóriaterületet. Ha nem írunk saját destruktork függvényt, akkor is a garbage collector minden, az objektum által már nem használt memóriát felszabadít az automatikusan definiált destruktork függvény segítségével. Ez azt jelenti, hogy nincs túlzottan nagy kényszer saját destruktork definiálására. A garbage collector valójában az osztály *Finalize* függvényét hívja meg.

Ez azért lehetséges, mert amikor a programozó formálisan destruktork függvényt definiál, akkor azt a fordító egy *Finalize* függvényre alakítja az alábbiak szerint:

Példa:

```
class osztály
{
    ~osztály()
    { ...
        // destruktork függvénytörzs
    }
}
```

A fenti osztálydestruktorból a fordító az alábbi kódot generálja:

```
protected override void Finalize()
{
    try
    { ...
        // destruktork függvénytörzs
    }
    finally
    {
        base.Finalize();
    }
}
```

A fenti destruktork konstrukciónak egyetlen bizonytalan pontja van, ez pedig a végrehajtás időpontja. Ha szükségünk van olyan megoldásra, ami determinált destrukciós folyamatot eredményez, akkor ezt az ún. *Dispose* metódus implementálásával (ez az *IDisposable* interface része) tehetjük meg.

Mielőtt ezzel a klasszikus használati formával foglalkoznánk, meg kell ismerkednünk a *System* névtér GC osztályának a szemétygyűjtési algoritmus befolyásolására leggyakrabban használt metódusaival:

```
void System.GC.Collect();
```

Kezdeményezzük a keretrendszer szemétygyűjtő algoritmusának indítását:

```
void System.GC.WaitForPendingFinalizers();
```

A hívó végrehajtási szál addig várakozik, amíg a destruktorkok hívásának sora (*Finalize* függvények hívási sora) üres nem lesz. Semmi garancia nincs arra, hogy ez a függvényhívás visszatér, hiszen ettől a száltól függetlenül más objektumok is életciklusuk végére érhetnek.

```
void System.GC.SuppressFinalize(Object o);
```

## VII. Osztályok

---

Ha egy objektumnak nincs szüksége már semmilyen destrukcióra, *Finalize* függvényhívásra, akkor a paraméterül adott objektum ilyen lesz:

```
void System.GC.ReRegisterForFinalize(Object o);
```

Máris feliratkozunk a destrukcióra várakozók sorába. Természetesen, ha ezt többször hívjuk meg, akkor az objektum többször a várakozók sorába kerül.

Ahogy korábban említettük a destruktorkal kapcsolatosan, mi nem tudjuk meghívni, a destruktort a keretrendszer szemétgyűjtő algoritmus hívja meg. Ha mégis szükségünk lenne olyan hívásra, amit közvetlenül is végre tudunk hajtani, akkor a *Dispose* függvény definiálására van szükségünk. Ekkor jellemzően egy logikai változó mutatja, hogy az aktuális objektum élő, és még nem hívták meg a *Dispose* metódusát. Ezen függvény klasszikus használata a következő:

```
bool disposed=false;
...
protected void Dispose( bool disposing )
{
    if( !disposed )
    {
        if (disposing)
        {
            // ide jön az erőforrás felszabadító kód
        }
        this.disposed=true;    // nem kell több hívás
        // ha van ősosztály, akkor annak dispose hívása
        base.Dispose( disposing );
        // erre az objektumra már nem kell finalize függvényt
        //hívni a keretrendszernek
        GC.SuppressFinalize(this);
    }
}
```

Itt kell szót ejtenünk arról, hogy a nyelv *using* utasításának segítségével (lásd a *Nyelvi utasítások* fejezetet) tömör kód írható.

A destrukció folyamata lényegében a hatékony erőforrás-gazdálkodáshoz nyújt segítséget. Ennek egyik leglényegesebb eleme a memóriagazdálkodás. Bár a mai számítógépek világában a memória nagysága nem a kritikus paraméterek között szerepel, azért előfordulhat, a fejlesztések során az alkalmazásunk memóriahiányban szenved. Ekkor segítséget adhatunk a memória-visszanyerő szemétgyűjtési algoritmusnak ahhoz, mely területeket lehet visszaadni a

rendszer részére. A kritikus esetben visszaadható objektumok hivatkozásait „gyenge referenciának” (*weak reference*) nevezzük. Ezeket az objektumokat erőforrás hiányában a keretrendszer felszabadítja. Ilyen objektumot a *WeakReference* típus segítségével tudunk létrehozni.

Példa:

```
...
Object obj = new Object(); // erős referencia
WeakReference wr = new WeakReference(obj);
obj = null; // az eredeti erős objektumot megszüntetjük
// ...
obj = (Object) wr.Target;
if (obj != null) { //garbage collection még nem volt
// ...
}
else { // objektum törölve
// ...
}
```

A destruktorkal kapcsolatban zárasképpen a következő (a rendszer szolgáltatásaiból eredő) tanácsokat adhatjuk:

- Az esetek nagy részében, ellentétben a C++ nyelvbeli használattal, nincs szükség destruktor definiálására.
- Ha mégis valamilyen rendszererőforrást kézi kóddal kell lezárni, akkor definiáljunk destruktort. Ennek hátránya az, hogy végrehajtása nem determinisztikus.
- Ha programkódból kell destruktor jellegű hívást kezdeményezni, a C++ belüli *delete* híváshoz hasonlóan, akkor a *Dispose* függvény definiálásával, majd annak hívásával élhetünk.

A feladataink során gyakorta előfordul, hogy egy verem-adatszerkezetet kell létrehoznunk. Definiáljuk most a veremosztályt úgy, hogy a rendszerkönyvtár *Object* típusát tudjuk benne tárolni. Ennél a feladatnál is, mint általában az igaz, hogy nincs szükségünk destruktor definiálására.

Példa:

```
using System;
class verem
{
    Object[] x;           // elemek tárolási helye
    int mut;              // veremmutató
    public verem(int db)  // konstruktor
    {
        x=new object[db]; // helyet foglalunk a vektornak
        mut=0;             // az első szabad helyre mutat
    }
}
```

```
// NEM DEFINIÁLUNK DESTRUKTORT
// MERT AZ AUTOMATIKUS SZEMÉTTYŰJTÉSI
// ALGORITMUS FELSZABADÍTVÁ A MEMÓRIÁT
public void push(Object i)
{
    if (mut<x.Length)
    {
        x[mut++]=i;
    }
    // beillesztettük az elemet
}
public Object pop()
{
    if (mut>0) return x[--mut];
    // ha van elem, akkor visszaadjuk a tetejéről
    else return null;
}
}
class program
{
    public static void Main()
    {
        verem v=new verem(6);
        v.push(5);
        v.push("alma");
        Console.WriteLine(v.pop()); // alma kivétele a veremből
        // az 5 még bent marad
    }
}
```

A képernyőn futási eredményként az *alma* szót látjuk.

Egy osztály természetes módon nemcsak „egyszerű” adatmezőket tartalmazhat, hanem osztálymezőket is. A tagosztályok inicializálása az osztálydefiniációban vagy a konstruktor függvényben explicit kijelölhető.

Az explicit kijelöléstől függetlenül egy objektum inicializálásakor az objektum konstruktorának végrehajtása előtt, a tagosztálykonstruktorok is meghívásra kerülnek a deklaráció sorrendjében.

### VII.3. Konstans, csak olvasható mezők

Az adatmezők hozzáférhetősége az egyik legfontosabb kérdés a típusaink tervezésekor. Ahogy korábban már láttuk, az osztályon belüli láthatósági hozzáférés állításával (*private*, *public*, ...) a megfelelő adathozzáférési igények kialakíthatók. Természetes ezek mellett az az igény is, hogy a változók módosíthatóságát is szabályozni tudjuk.



### VII.3.1 Konstans mezők

Ahogy korábban említettük, egy változó módosíthatóságát a *const* kulsszóval is befolyásolhatjuk. A konstans mező olyan adatot tartalmaz, amelyik értéke fordítási időben kerül beállításra. Ez azt jelenti, hogy ilyen mezők inicializáló értékadását kötelező a definíció során jelölni.

Példa:

```
using System;
class konstansok
{
    public const double pi=3.14159265; // inicializáltuk
    public const double e=2.71828183;
}
class konstansapp
{
    public static void Main()
    {
        Console.WriteLine(konstansok.pi);
    }
}
```

Egy konstans mező eléréséhez nincs szükség arra, hogy a típusból egy változót készítsünk, ugyanis a konstans mezők egyben statikus mezők is. Ahogy látható, a konstans mezőt helyben inicializálni kell, ebből pedig az következik, hogy minden később definiálandó osztályváltozóban ugyanezen kezdőértékadás hajtódik végre, tehát ez a mező minden változóban ugyanaz az érték lehet. Ez pedig a statikus mező tulajdonsága.

### VII.3.2. Csak olvasható mezők

Ha egy adatmezőt a *readonly* jelzővel látunk el, akkor ezt a mezőt csak olvasható mezőnek nevezzük. Ezen értékeket a program során csak olvasni, használni tudjuk. Ezen értékek beállítását egyetlen helyen, a konstruktorban végezhetjük el. Látható, hogy a konstans és a csak olvasható mezők közti leglényegesebb különbség az, hogy míg a konstans mező minden példányban ugyanaz, addig a csak olvasható mező konstansként viselkedik miután létrehoztuk a változót, de minden változóban más és más értékű lehet!

Példa:

```
using System;
class csak_olvashato
{
    public readonly double x;
    public csak_olvashato(double kezd)
    { x=kezd; }
}
```

```
class olvashatosapp
{
    public static void Main()
    {
        csak_olvashato o=new csak_olvashato(5);
        Console.WriteLine(o.x);          // értéke 5
        csak_olvashato p=new csak_olvashato(6);
        Console.WriteLine(p.x);          // értéke 6
        p.x=8;                          // fordítási hiba, x csak olvasható!!!!
    }
}
```

Természetesen definiálható egy csak olvasható mező statikusként is, ebben az esetben a mező inicializálását az osztály statikus konstruktorában végezhethjük el.

### VII.4. Tulajdonság, index függvény

Egy osztály tervezésekor nagyon fontos szempont az, hogy az osztály adataihoz milyen módosítási lehetőségeket engedélyezzünk, biztosítsunk. Hagyományos objektumorientált nyelvekben ehhez semmilyen segítséget nem kaptunk, maradtak az általános függvénydefiniálási lehetőségeink. Ezeket a függvényneveket jellemzően a set illetve a get előtagokkal módosították.

#### VII.4.1. Tulajdonság, property függvény

A C# nyelv a tulajdonság (*property*) függvénydefiniálási lehetőségével kínál egyszerű és kényelmes adathozzáférési és módosítási eszközt. Ez egy olyan speciális függvény, amelynek nem jelölhetünk paramétereket, még a zárójeleket sem *()*, és a függvény törzsében egy *get* és *set* blokkot definiálunk. Használata egyszerű értékadásként jelenik meg. A *set* blokkban egy „*value*” névvel hivatkozhatunk a beállítási értékadás jobb oldali kifejezés értékére.

Példa:

```
using System;
class Ember
{
    private string nev;
    private int kor;
    // a nev adatmező módosításához definiált Nev tulajdonság
    public string Nev // kis- és nagybetű miatt nev!=Nev
    {
        get          // ez a blokk hajtódik végre akkor,
                    // amikor a tulajdonság értéket kiolvassuk
        {
```

---

```

        return nev;
    }
    set
    {
        nev=value;
    }
}
public Ember(string n, int k)
{
    nev=n; kor=k;
}
}
class program
{
    public static void Main()
    {
        Ember e=new Ember("Zoli", 16);
        //a Nev tulajdonság hívása
        Console.WriteLine(e.Nev); // a Nev property get blokk hívása
        e.Nev="Pali";             // a Nev property set blokkjának hívá-
                                // sa a value változóba kerül a "Pali"
    }
}

```

Ha a tulajdonság függvénynek csak *get* blokkja van, akkor azt csak olvasható tulajdonságnak (*readonly*) nevezzük.

Ha a tulajdonság függvénynek csak *set* blokkja van, akkor azt csak írható tulajdonságnak (*writeonly*) nevezzük.

#### VII.4.2. Index függvény (*indexer*)

Az *indexer* függvény definiálása valójában a vektorhasználat és a tulajdonság függvény kombinációja. Gyakran előfordul, hogy egy osztálynak olyan adatához szeretnénk hozzáférni, aminél egy indexérték segítségével tudjuk megmondani, hogy melyik is a keresett érték.

Az *indexer* függvény esetében lényeges különbség, hogy van egy *index* paraméter, amit szögletes zárójelek között kell jelölni, és nincs neve, pontosabban a *this* kulcsszó a neve, ugyanis az aktuális típust, mint vektort indexeli.

Mivel az *indexer* az aktuális típust, a létező példányt indexeli, ezért az *indexer* függvény nem lehet statikus.

Lássuk az alábbi példát, ami a jellemző használatot mutatja.

Példa:

```

class valami
{
    int [] v=new int[10];
}

```

```
...
public int this[int i]
{
    get
    {
        return v[i];
    }
    set
    {
        v[i]=value;        // mint a property value értéke
    }
}
}
class program
{
    static void Main()
    {
        valami a=new valami();
        a[2]=5;              // az indexer set blokk hívása
        Console.WriteLine(a[2]);    // 5, indexer get hívása
    }
}
```

Az *indexer* esetében is, hasonlóan a tulajdonságdefiníció használatához, nem feltétlenül kell mindkét (*get*, *set*) ágat definiálni. Ha csak a *get* blokk definiált, akkor csak olvasható, ha csak a *set* blokk definiált, akkor csak írható *indexer* függvényről beszélünk.

Az *indexer*használat kísértetiesen hasonlít a vektorhasználatához, de van néhány olyan különbség, amit érdemes megemlíteni.

- Az *indexer* függvény, egy függvénynek pedig nem csak egész (*index*) paramétere lehet.

Példa:

```
public int this[string a, int b]
{
    get
    {
        return x;
    }
    set
    {
        x=value;
    }
}
```

- az *indexer* függvény az előzőekből következően újradefiniálható (*overloaded*). A fenti *indexer* mellett a következő „hagyományos” is megfér:

Példa:

```
public string this[int x]
{
    get
    {
        return s[x];
    }
    set
    {
        s[x]=value;
    }
}
```

- az *indexert ref* és *out* paraméterként nem használhatjuk.

## VII.5. Osztályok függvényparaméterként

Egy függvény paramétereit más egyszerű adattípushoz hasonlóan lehetnek osztálytípusok is. Alapértelmezés szerint az osztálytípusú változó is érték szerint adódik át.

Tekintsük a következő példaosztályt. Legyen az osztálynak destruktora is.

Példa:

```
using System;
class példa
{
    private int x;

    public példa (int a)
    {
        Console.WriteLine( "Konstruktorhívás!");
        x=a;
    };
    public int X
    {
        get
        {
            return x;
        }
        set
        {
            x=value;
        }
    }
}
```

```
class program
{
    static int négyzet(példa a)
    {
        a.X=5;
        return (a.X*a.X);
    }

    static void Main()
    {
        példa b=new példa(3);
        Console.WriteLine(négyzet(b));
    }
}
```

A program futtatása a következőket írja a képernyőre:

```
Konstruktorhívás!
25
```

Mikor egy függvény paraméterként (érték szerint) egy osztályt kap, akkor a függvényparaméter egy értékreferenciát kap, és nem egy másolata készül el az eredeti osztályváltozónak.

Teljesen hasonló akkor a helyzet, mikor egy függvény osztálytípust ad visszatérési értéként.

### VII.6. Operátorok újradefiniálása

A már korábban tárgyalt operátoraink az ismert alaptípusok esetében használhatók. Osztályok (új típusok) megjelenésekor az értékadás operátora automatikusan használható, mert a nyelv létrehoz egy számára alapértelmezett értékadást az új osztályra is. Ezen értékadás operátort felülbírálni, azaz egy újat definiálni nem lehet a C# nyelvben (ellentétben pl. a C++ nyelvvel).

Hasonlóan nem lehet az index operátort [] sem újradefiniálni, bár az *indexer* definiálási lehetőség lényegében ezzel egyenértékű.

A legtöbb operátor újradefiniálható, melyek egy- vagy kétoperandusúak. Az alábbi operátorok definiálhatók újra:

Egyoperandusú operátorok: +, -, !, ~, ++, --, true, false

Kétoperandusú operátorok: +, -, \*, /, %, &, |, ^, <<, >>, <=, >=, ==, !=, <, >

A fenti hagyományosnak mondható operátorkészlet mellett még típuskonverziós operátor függvény is definiálható.

Az operátor függvény definíciójának formája:

```
static visszatérési_érték operator?(argumentum)
{
    // függvénytörzs
}
```

Az *operator* kulcsszó utáni *?* jel helyére az újradefiniálni kívánt operátor neve kerül. Tehát ha például az összeadás (+) operátort szeretnénk felülbírálni, akkor a *?* helyére a + jel kerül.

Az irodalomban az operátor újradefiniálást gyakran operátor overloading-nak hívják.

Az operátorok precedenciája újradefiniálás esetén nem változik, és az operandusok számát nem tudjuk megváltoztatni, azaz például nem tudunk olyan / (osztás) operátort definiálni, amelynek csak egy operandusa van.

Az operátor függvények öröklődnek, bár a származtatott osztályban az ősoztály operátor függvényei igény szerint újradefiniálhatóak.

Az operátor függvény argumentumával kapcsolatosan elmondhatjuk, hogy egyoperandusú operátor esetén egy paramétere van, míg kétoperandusú operátor esetén két paramétere van a függvénynek.

Meg kell említeni, hogy a C++ nyelvhez képest nem olyan általános az operátor újradefiniálás lehetősége, de az is igaz, hogy sok, ma népszerű programozási nyelv (Java, Delphi, ...) még ennyit se nyújt.

Tekintsük első példaként a komplex számokat megvalósító osztályt, amelyben az összeadás operátort szeretnénk definiálni oly módon, hogy egy komplex számhoz hozzá tudjunk adni egy egész számot. Az egész számot a komplex szám valós részéhez adjuk hozzá, a képzetes rész változatlan marad.

Példa:

```
using System;
class komplex
{
    private float re, im;
    public komplex(float x, float y) // konstruktor
    {
        re=x; im=y;
    }
    float Re
    {
        get
        {
            return re;
        }
    }
}
```

## VII. Osztályok

---

```
        set
        {
            re=value;
        }
    }
    float Im
    {
        get
        {
            return im;
        }
        set
        {
            im=value;
        }
    }

    public static komplex operator+(komplex k,int a)
    {
        komplex y=new komplex(0,0);
        y.Re=a+k.Re;
        y.Im=k.Im;
        return y;
    }

    override public string ToString()
    {
        string s="A keresett szám:"+re+": "+im;
        return s;
    }
}

class program
{
    public static void Main()
    {
        komplex k=new komplex(3,2);
        Console.WriteLine("Komplex szám példa.");
        Console.WriteLine("Összeadás eredménye: {0}",k+4);
    }
}
```

Az eredmény a következő lesz:

Komplex szám példa.

Összeadás eredménye: A keresett szám:7:2



A  $k+4$  összeadás úgy értelmezendő, hogy a  $k$  objektum + függvényét hívtuk meg a  $k$  komplex szám és a 4 egész szám paraméterrel, azaz  $k+(k,4)$  függvényhívás történt.

Abban az esetben, ha a *komplex* + *komplex* típusú összeadást szeretnénk definiálni, akkor egy újabb operátor függvénnyel kell a *komplex* osztályt bővíteni. Ez a függvény a következőképpen nézhet ki:

```
public static komplex operator+(komplex a, komplex b)
{
    komplex temp=new komplex(0,0);
    temp.re= b.re+a.re;
    temp.im= b.im+a.im;
    return temp;
}
```

Ahhoz, hogy az összeadás operátorát a korábban (az egyszerű típusoknál) megszokott módon tudjuk itt is használni, már csak az kell, hogy az *egész* + *komplex* típusú összeadást is el tudjuk végezni. (Az összeadás kommutatív!) Erre az eddigi két összeadás operátor nem ad lehetőséget, hiszen ebben az esetben, a bal oldali operandus mindig maga az aktuális osztály. A mostani esetben viszont a bal oldali operandus egy egész szám.

Ekkor a legkézenfekvőbb lehetőség az, hogy az *egész* + *komplex* operátorfüggvényt is definiáljuk. Figyelembe véve a Visual Studio szövegszerkesztőjének szolgáltatásait, ez gyorsan megy, így elkészítjük ezt a változatot is:

```
...
public static komplex operator+(int a, komplex k)
{
    komplex y=new komplex(0,0);
    y.Re=a+k.Re;
    y.Im=k.Im;
    return y;
}
...
```

Ekkor a *Console.WriteLine("Összeadás eredménye: {0}",4+k);* utasítás nem okoz fordítási hibát.

Erre a problémára még egy megoldást adhatunk. Ez pedig a konverziós operátor definiálásának lehetősége. Ugyanis, ha definiálunk olyan konverziós operátort, amely a 4 egész számot komplex típusúra konvertálja, akkor két komplex szám összeadására vezettük vissza ezt az összeadást.

A konverziós operátoroknak készíthetünk implicit vagy explicit változatát is. Implicitnek nevezzük azt a konverziós operátorhívást, mikor nem jelöljük a forrásszövegben a konverzió műveletét, explicitnek pedig azt, amikor jelöljük.

## VII. Osztályok

---

Konverziós operátornál az operátor jel helyett azt a típust írjuk le, amire konvertálunk, míg paraméterül azt a típust adjuk meg, amiről konvertálni akarunk.

Visszatérve a fenti komplex szám kérdésre, az *egész* + *komplex* operátor helyett az alábbi operátort is definiálhattuk volna:

```
public static implicit operator komplex(int a)
{
    komplex y=new komplex(0,0);
    y.Re=a;
    return y;
}
```

Ha az operátor szó elé az *implicit* kulcsszót írjuk, akkor az implicit operátor függvényt definiáljuk, tehát *4* + *komplex* jellegű utasításnál a *4* számot implicit (nem jelölve külön) konvertáljuk komplex számmá.

Előfordulhat, hogy az implicit és az explicit konverzió mást csinál, ekkor, ha akarjuk, az explicit verziót is definiálhatjuk.

```
public static explicit operator komplex(int a)
{
    komplex y=new komplex(0,0);
    y.Re=a+1;    // mást csinál, mint az előző
                // nem biztos, hogy matematikailag is helyes!!!
    return y;
}
```

Az explicit verzió meghívása a következőképpen történik:

```
...
komplex k=(komplex) 5;
Console.WriteLine(k.Re);    // 6
...
```

Természetesen egy komplex számhoz értékadás útján rendelhetünk egy valós számot is, mondjuk oly módon, hogy a komplex szám valós részét adja a valós szám, míg a képzetes rész legyen 0.

Két egyoperandusú operátor, a ++ és a -- esetében, ahogy az operátorok tárgyalásánál is láttuk, létezik az operátorok postfix illetve prefix formájú használata is:

```
komplex k=new komplex(1,2);
k++;    // postfix ++
++k;    // prefix forma
```

Ha definiálunk ++ operátor függvényt, akkor ezen két operátor esetében mindkét forma használatánál ugyanaz az operátor függvény kerül meghívásra.

```
public static komplex operator++(komplex a)
{
    a.Re=a.Re+1;
    a.Im=a.Im+1;
    return a;
}
```

Befejezésül a *true*, *false* egyoperandusú operátor definiálásának lehetőségéről kell röviden szólni. A C++ nyelvvel ellentétben, ahol egy adott típust logikainak is tekinthetünk (igaz, ha nem 0, hamis, ha 0), a C# nyelvben a már korábról ismert

```
if (a) utasítás;
```

alakú utasítások akkor fordulnak le, ha az *a* változó logikai. A *true* és *false* nyelvi kulcsszavak nemcsak logikai konstansok, hanem olyan logikai egyoperandusú operátorok, melyeket újra lehet definiálni.

A *true*, *false* operátor logikai. Megkötés, hogy mindkét operátort egyszerre kell újradefiniálnunk. A jelentése pedig az, hogy az adott típust mikor tekinthetjük logikai igaznak vagy hamisnak.

Definiáljuk újra ezeket az operátorokat a már megismert komplex osztályunkhoz:

```
...
public static bool operator true(komplex x)
{
    return x.Re > 0;
}
public static bool operator false(komplex x)
{
    return x.Re <= 0;
}
```

Ez a definíció ebben az esetben azt jelenti, hogy egy komplex szám akkor tekinthető logikai igaz értékűnek, ha a szám valós része pozitív.

Példa:

```
komplex k=new komplex(2,0);
if (k) Console.WriteLine("Igaz");
```

Ekkor a képernyőre kerülő eredmény az *igaz* szó lesz!

Végül azt kell megemlíteni, hogy a logikai *true*, *false* operátorok mintájára, azokhoz hasonlóan csak párban lehet újradefiniálni néhány operátort. Ezek az operátorok a következők:

<code>==, !=</code>	azonosság, különbözőség megadása
<code>&lt;, &gt;</code>	kisebb, nagyobb
<code>&lt;=, &gt;=</code>	kisebb vagy egyenlő, nagyobb vagy egyenlő

### VII.7. Interface definiálása

Egy osztály definiálása során a legfontosabb feladat az, hogy a készítendő típus adatait, metódusait megadjuk. Gyakran felmerül az az igény, hogy egy további fejlesztés, általánosabb leírás érdekében ne egy osztály keretében fogalmazzuk meg a legjellemzőbb tulajdonságokat, hanem kisebb tulajdonságcsoportok alapján definiáljunk. A keretrendszer viszont csak egy osztályból enged meg egy új típust, egy utódosztályt definiálni. Ez viszont ellentmond annak az elvárásnak, hogy minél részletesebben fogalmazzuk meg a típusainkat.

#### VII.7.1. Interface fogalma

A fenti ellentmondás feloldására alakult ki az a megoldás, hogy engedjünk meg olyan típust, interface-t definiálni, ami nem tartalmaz semmilyen konkrétumot, de rendelkezik a kívánt előírásokkal.

Az interfacedefiníció formája:

```
interface név
{
    // deklarációs fejlécek
}
```

Példa:

```
...
interface IAlma
{
    bool nyári();
    double termésátlag();
}
```

A fenti példában definiáltuk az *Ialma* interface-t, ami még nem jelent közvetlenül használható típust, hanem csak azt írja elő, hogy annak a típusnak, amelyik majd ennek az *Ialma* típusnak a tulajdonságait is magán viseli, kötelezően definiálnia kell a *nyári()*, és a *termésátlag()* függvényeket. Tehát erről a típusról azt tudhatjuk, hogy az interface által előírt tulajdonságokkal biztosan rendelkezni fog. Ezen kötelező tulajdonságok mellett természetesen tetszőleges egyéb jellemzőkkel is felruházhatjuk majd az osztályunkat.

Amikor könyvtári előírásokat, interface-eket implementálunk, akkor általában az elnevezések az *I* betűvel kezdődnek, utalva ezzel a név által jelzett tartalomra.

Egy interface előírásai közé nem csak függvények, hanem tulajdonságok és indexer előírás megadása és esemény megadása is beletartozhat.

Példa:

```
interface IPozíció
{
    int X { get; set; }
    int Y { get; }           // readonly tulajdonság
    int Z { set; }           // csak írható tulajdonság
    int this[int i] { get; set; } // read-write indexer
    int this[int i] { get; } // read-only indexer
    int this[int i] { set; } // write-only indexer
}
```

### VII.7.2. Interface használata

Az *Ialma* előírások figyelembevétele a következőképpen történik. Az *osztálynév (típusnév)* után kettőspontot kell tenni, majd utána következik az implementálandó név.

Példa:

```
using System;
class jonatán: IAlma
{
    private int kor;
    public jonatán(int k)
    {
        kor=k;
    }
    public bool nyári()
    {
        return false;
    }
    public double termésátlag()
    {
        if ((kor>5) && (kor<30))
```

## VII. Osztályok

---

```
        return 230;
        else return 0;
    }
}

class program
{
    public static void Main()
    {
        jonatán j=new jonatán(8);
        Console.WriteLine(j.termésátlag());
        // 230 kg az átlagtermés
    }
}
```

### VII.7.3. Interface-ek kompozíciója

Az interface egységek tervezésekor lehetőségünk van egy vagy több már meglévő interface felhasználásával ezekből egy újabbat definiálni.

Példa:

```
using System;
public interface IAlma
{
    bool nyári();
    double termésátlag();
}

public interface szállítható
{
    bool szállít();
}

public interface szállítható_alma:IAlma,szállítható
{
    string csomagolás_módja();
}

public class jonatán: szállítható_alma
{
    public bool nyári()
    {
        return false;
    }
}
```

```

    public double termésátlag()
    {
        return 250;
    }
    public string csomagolás_módja()
    {
        return "konténer";
    }
    public bool szállít()
    {
        return false;
    }
}
class program
{
    public static void Main()
    {
        jonatán j=new jonatán();
        Console.WriteLine(j.termésátlag()); // 250 kg az átlagtermés
        Console.WriteLine(j.csomagolás_módja()); // konténer
    }
}

```

A definiált új típusra vonatkozóan használhatjuk mind az *is* mind az *as* operátort. Az iménti példát tekintve értelmes, és igaz eredményt ad az alábbi elágazás:

Példa:

```

...
if (j is IAlma) Console.WriteLine("Ez bizony alma utód!");

IAlma a=j as IAlma;           // IAlma típusra konvertálás
a.termésátlag();               // függvényvégrehajtás

```

## VII.8. Osztályok öröklődése

Az öröklődés az objektumorientált programozás elsődleges jellemzője. Egy osztályt számozhatunk egy őszosztályból, és ekkor az utódosztály az őszosztály tulajdonságait (függvényeit, ...) is sajátjának tudhatja. Az örökölt függvények közül a változtatásra szorulókat újradefiniálhatjuk. Öröklés esetén az osztály definíciójának formája a következő:

<pre> class utódnév: ősnév {     // ... } </pre>
--

## VII. Osztályok

---

Hasonlóan az osztályok mezőhozzáférési rendszeréhez, több nyelvben is lehetőség van arra, hogy öröklés esetén az utódosztály az őszosztály egyes mezőit többféle (*private*, *protected*, *public*) módon örökölhessen. A C# nyelvben a .NET Frameworknek (Common Type System) köszönhetően erre nincs mód. Minden mező automatikusan, mintha publikus öröklés lenne, megtartja őszosztálybeli jelentését.

Ekkor az őszosztály publikus mezői az utódosztályban is publikus mezők, és a *protected* mezők az utódosztályban is *protected* mezők lesznek.

Egy őstípusú referencia bármely utódtípusra hivatkozhat.

Az őszosztály privát mezői az utódosztályban is privát mezők maradnak az őszosztályra nézve is, így az őszosztály privát mezői közvetlenül az utódosztályból sem érhetők el. Az elérésük például publikus, ún. „közvetítő” függvény segítségével valósítható meg.

Az elérési módok gyakorlati jelentését nézzük meg egy szematikus példán keresztül:

```
class ős
{
    private int i;           // privát mező
    protected int j;        // protected mezőtag
    public int k;            // publikus mezők
    public void f(int j)
    { i=j; };
}

class utód: ős
{
    ...
};
```

Ekkor az utódosztálynak „helyből” lesz egy *protected* mezője, a *j* egész változó, és lesz két publikus mezője, a *k* egész változó és az *f* függvény.

Osztálykönyvtárak használata mellett (pl. MFC for Windows, Windows NT) gyakran találkozunk azzal az esettel, mikor a könyvtárosztály *protected* mezőket tartalmaz. Ez azt jelenti, hogy a függvényt, mezőt olyan használatra szánták, hogy csak származtatott osztályból tudjuk használni.

A C# nyelvben nincs lehetőségünk többszörös öröklés segítségével egyszerre több őszosztályból egy utódosztályt származtatni. Helyette viszont tetszőleges számú interface-t implementálhat minden osztály.

```
class utód: ős, interfacs1, ... {
    //
};
```



Konstruktorok és destruktorok használata öröklés esetén is megengedett. Egy típus definiálásakor a konstruktor függvény kerül meghívásra, és ekkor először az ősosztály konstruktora, majd utána az utódosztály konstruktora kerül meghívásra, míg destruktor esetén fordítva, először az utódosztály majd utána az ősosztály destruktorát hívja a rendszer. Természetesen a destruktor hívására az igaz, hogy a keretrendszer hívja meg valamikor azután, hogy az objektumok élettartama megszűnik.

Paraméteres konstruktorok esetén az utódkonstruktor alakja:

```
utód(paraméterek) : base(paraméterek)
{
    // ...
}
```

Többszörös öröklés esetén először az őskonstruktorok kerülnek a definíció sorrendjében végrehajtásra, majd az utódbeli tagosztályok konstruktora és legvégül az utódkonstruktor következik. A destruktorok hívásának sorrendje a konstruktorsorrendhez képest fordított. Ha nincs az utódban direkt őshívás, akkor a rendszer a paraméter nélküli őskonstruktorát hívja meg.

Ezek után nézzük a fentieket egy példán keresztül.

Példa:

```
using System;

class a
{
    public a()
    { Console.WriteLine( "A konstruktor");}
    ~a()
    { Console.WriteLine("A destruktor");}
}
class b: a
{
    public b()
    { Console.WriteLine("B konstruktor");}
    ~b()
    { Console.WriteLine("B destruktor");}
}
class program
{
    public static void Main()
    {
        b x=new b(); // b típusú objektum keletkezik, majd 'kimúlik'
        // Először az a majd utána a b konstruktort hívja meg
        // a fordító
    }
}
```

```
        // Kimúláskor fordítva, először a b majd az a  
        // destruktora kerül meghívásra  
    }  
}
```

### VII.9. Végleges és absztrakt osztályok

A típusdefiníciós lépéseink során előfordulhat, hogy olyan osztályt definiálunk, amelyekre azt szeretnénk kikötni, hogy az adott típusból, mint ősből ne tudjunk egy másik típust, utódot származtatni.

Ahhoz, hogy egy adott osztályt véglegesnek definiáljunk, a *sealed* jelzővel kell ellátni.

Példa:

```
sealed class végleges  
{  
    public végleges()  
    {  
        Console.WriteLine( "A konstruktor" );  
    }  
    class utód:végleges          // fordítási hiba  
    {  
  
    }  
}
```

Ha *protected* mezőt definiálunk egy *sealed* osztályban, akkor a fordító figyelmeztető üzenetet ad, mivel nincs sok értelme az utódosztályban látható mezőt definiálni.

Interface definiálás esetében csak előírásokat – függvény, tulajdonság formában – fogalmazhatunk meg az implementáló osztály számára. Gyakran előfordul, hogy olyan típust szeretnénk létrehozni, mikor a definiált típusból még nem tudunk példányt készíteni, de nemcsak előírásokat, hanem adatmezőket, implementált függvényeket is tartalmaz.

Ez a lehetőség az *abstract* osztály definiálásával valósítható meg. Ehhez az *abstract* kulcsszót kell használnunk az osztály neve előtt. Egy absztrakt osztály egy vagy több absztrakt függvénymezőt tartalmazhat (nem kötelező!!!). Ilyen függvénynek nincs törzse, mint az interface függvényeknek. Egy absztrakt osztály utódosztályában az absztrakt függvényt kötelező implementálni. Az utódosztályban ekkor az *override* kulcsszót kell a függvény fejlécbe írni.

**1. példa:**

```
abstract class os
{
    private int e;
    public os()
    {
        e=5;
    }
    // az osztálynak nincs absztrakt mezője, de
    // ettől az osztály még lehet absztrakt
}
os x= new os(); // hiba, mert absztrakt osztályból nem
                // készíthetünk változót
```

**2. példa:**

```
using System;
abstract class os
{
    private int e;
    public os(int i)
    {
        e=i;
    }
    public abstract int szamol();
    public int E
    {
        get
        {
            return e;
        }
    }
}
class szamolo:os
{
    public szamolo():base(3)
    {
        ...
    }
    public override int szamol()
    {
        return base.E*base.E;
    }
}
```

```
class program
{
    public static void Main()
    {
        szamolo f=new szamolo();
        Console.WriteLine(f.szamol()); // eredmény: 9
    }
}
```

### VII.10. Virtuális tagfüggvények, függvényelfedés

A programkészítés során, ahogy láttuk, az egyik hatékony fejlesztési eszköz az osztályok öröklési lehetőségének kihasználása. Ekkor a függvénypolimorfizmus alapján természetesen lehetőségünk van ugyanazon névvel mind az ősosztályban, mind az utódosztályban függvényt készíteni. Ha ezen függvényeknek különbözők a paraméterei, akkor gyakorlatilag nincs is kérdés, hiszen függvényhíváskor a paraméterekből teljesen egyértelmű, hogy melyik kerül meghívásra. Nem ilyen egyértelmű a helyzet, amikor a paraméterek is azonosak.

#### VII.10.1. Virtuális függvények

A helyzet illusztrálására nézzük a következő példát. Definiáltuk a *pont* ősosztályt, majd ebből származtattuk a *kor* osztályunkat. Mindkét osztályban definiáltunk egy *kiir* függvényt, amely paraméter nélküli és az osztály adattagjait írja ki.

Példa:

```
using System;

class pont
{
    private int x,y;
    public pont()
    {
        x=2;y=1;
    }
    public void kiir()
    {
        Console.WriteLine(x);
        Console.WriteLine(y);
    }
}
```

```

class kor:pont
{
    private int r;
    public kor()
    {
        r=5;
    }
    public void kiir()
    {
        Console.WriteLine(r);
    }
}
class program
{
    public static void Main()
    {
        kor k=new kor();
        pont p=new pont();
        p.kiir();           //2,1
        k.kiir();           //5
    }
}

```

A program futásának eredményeként először a *p* pont adatai (2,1), majd a *kor* adata (5) kerül kiírásra. Bővítsük a programunkat az alábbi két sorral:

```

    public static void Main()
    {
        kor k=new kor();
        pont p=new pont();
        p.kiir();
        k.kiir();
        p=k;           //Egy őstípus egy utódra hivatkozik
        p.kiir();       //Mit ír ki?
    }

```

A  $p=k$  értékadás helyes, hiszen *p* (*pont* típus) típusa a *k* típusának (*kor*) az őse. Azt is szoktuk mondani, hogy egy őstípusú referencia tetszőleges utódtípusra hivatkozhat. Ekkor a második *p.kiir()* utasítás is, a  $p=k$  értékadástól függetlenül a 2,1 értékeket írja ki! Miért? Mert az osztályreferenciák alapértelmezésben statikus hivatkozásokat tartalmaznak a saját függvényeikre. Mivel a *pont*-ban van *kiir*, ezért attól függetlenül, hogy időközben a program során (futás közbeni – dinamikus – módosítás után) a *p* már egy *kor* objektumot azonosít, azaz a *kor.kiir* függvényét kellene meghívni, még mindig a fixen, fordítási időben hozzákapcsolt *pont.kiir* függvényt hajtja végre.

## VII. Osztályok

---

Ha azt szeretnénk elérni, hogy mindig a dinamikusan hozzátartozó függvényt hívja meg, akkor az őszosztály függvényét virtuálisnak (*virtual*), míg az utódosztály függvényét felüldefiniáltnak (*override*) kell nevezni, ahogy az alábbi példa is mutatja.

Példa:

```
using System;
class pont
{
    private int x,y;
    public pont()
    {
        x=2;y=1;
    }
    virtual public void kiir()
    {
        Console.WriteLine(x);
        Console.WriteLine(y);
    }
}
class kor:pont
{
    private int r;
    public kor()
    {
        r=5;
    }
    override public void kiir()
    {
        Console.WriteLine(r);
    }
}
public class MainClass
{
    public static void Main()
    {
        kor k=new kor();
        pont p=new pont();
        p.kiir();    // pont kiir 2,1
        k.kiir();    // kor kiir 5
        p=k;        // a pont a korre hivatkozik
        p.kiir();    // kor kiir 5 - a kor kiir kerül
                    // meghívásra, mert a kiir függvény virtuális, így a
                    // kiir hívásakor mindig a változó aktuális, és nem
                    // pedig az eredeti típusát kell figyelembe venni.
    }
}
```

Ez a tulajdonság az osztályszerkezet rugalmas bővítését teszi lehetővé, míg a programkód bonyolultsága jelentősen csökken.

Gyakran előfordul, hogy az őosztályban nincs szükségünk egy függvényre, míg a származtatott osztályokban már igen, és szeretnénk, ha virtuálisként tudnánk definiálni. Ebben az esetben az őosztályban egy absztrakt függvénydefiniációt kell használnunk, ahogy az alábbi példában olvasható.

Példa:

```
abstract class pont          // absztrakt osztály, ebből nem
{                             // lehet példányt készíteni
    protected int x,y;
    public pont()
    {
        x=20;y=10;
    }
    abstract public void rajzol();
    public void mozgat(int dx, int dy)
    {
        x+=dx;
        y+=dy;
        // rajzol hívás
        rajzol();
    }
}
class kor:pont
{
    private int r;
    public kor()
    {
        r=5;
    }
    override public void rajzol()
    {
        Console.WriteLine("Megrajzoljuk a kört az {0}, {1}
                           pontba, {2} sugárral.",x,y,r);
    }
}
public class MainClass
{
    public static void Main()
    {
        // pont p=new pont(); utasításhibát eredményezne
        kor k=new kor();
        k.mozgat(3,4);
    }
}
```

### VII.10.2. Függvényeltakarás, sealed függvény

Az öröklés kapcsán az is előfordulhat, hogy a bázisosztály egy adott függvényére egyáltalán nincs szükség az utódosztályban. Az alábbi példában a focicsapat osztálynak van *nevkiir* függvénye. Az utód *kedvenc\_focicsapat* osztálynak is van ilyen függvénye. A *new* hatására a *kedvenc\_focicsapat* osztály *nevkiir* függvénye eltakarja az ősoosztály hasonló nevű függvényét.

Példa:

```
class focicsapat
{
    protected string csapat;
    public focicsapat()
    {
        csapat="UTE";
    }
    public void nevkiir()
    {
        Console.WriteLine("Kedves csapat a lila-fehér {0}",csapat);
    }
}
class kedvenc_csapat:focicsapat
{
    public kedvenc_csapat()
    {
        csapat="Fradí";
    }
    new public void nevkiir()
    {
        Console.WriteLine("Kedvenc csapatunk a: {0}",csapat);
    }
}
public class MainClass
{
    public static void Main()
    {
        kedvenc_csapat cs=new kedvenc_csapat();
        cs.nevkiir();
    }
}
```

A *new* utasítás hatására egy öröklési sor új bázisfüggvénye lesz az így definiált *nevkiir* függvény.

Ennek az ellenkezőjére is igény lehet, mikor azt akarjuk elérni, hogy az ősoosztály függvényét semmilyen más formában ne lehessen megjeleníteni. Ekkor a függvényt, az osztály mintájára, véglegesíteni kell, azaz a *sealed* jelzővel kell megjelölni.



### VII.11. Feladatok

1. Mi a különbség egy struktúra és egy osztály között?
2. Milyen szerepe van a konstruktoroknak, destruktorknak? Milyen konstruktorok definiálhatók?
3. Mi a különbség az osztály, az absztrakt osztály is az interface között?
4. Definiáljon *bútor* osztályt a jellemző tulajdonságokkal! (Bútor neve, alapanyaga, rendeltetése, ára) A megadott tulajdonságokhoz készítse el a kezelő függvényeket!

Készítsen *lapraszerelt* névvel interface-t, amiben az összeszerelési utasítást írjuk elő! Módosítsa az előző *bútor* osztályt *lapraszerelt bútor*-ra, amelyik implementálja a *lapraszerelt* interface-t, biztosítva azt, hogy ennek a típusnak biztosan legyen összeszerelési utasítása.

## VIII. Kivételkezelés

Egy program, programrész vagy függvény végrehajtása formális eredményesség tekintetében három kategóriába sorolható.

- A függvény vagy programrész végrehajtása során semmilyen „rendellenesség” nem lép fel.
- A függvény vagy programrész aktuális hívása nem teljesíti a paraméterekre vonatkozó előírásainkat, így ekkor a „saját hibavédelmünk” eredményekénti hibával fejeződik be a végrehajtás.
- A függvény vagy programrész végrehajtása során előre nem látható hibajelenség lép fel.

Ezen harmadik esetben fellépő hibajelenségek programban történő kezelésére nyújt lehetőséget a kivételkezelés (*Exception handling*) megvalósítása. Ha a második esetet tekintjük, akkor a „saját hibavédelmünk” segítségével, mondjuk valamilyen „nem használt” visszatérési értékkel tudjuk a hívó fél tudomására hozni, hogy hiba történt. Ez néha komoly problémákat tud okozni, hiszen például egy egész értékkel visszatérő függvény esetében néha elég körülményes olyan egész értéket találni, amelyik nem egy lehetséges valódi visszatérési érték. Így ebben az esetben is, bár a fellépő hibajelenség valahogy kezelhető, a kivételkezelés lehetősége nyújt elegáns megoldást.

A kivételkezelés lehetősége hasonló, mint a fordítási időben történő hibakeresés, hibavédelem azzal a különbséggel, hogy mindezt futási időben lehet biztosítani. A C# kivételkezelés a hibakezelést támogatja. Nem támogatja az ún. aszinkron kivételek kezelését, mint például billentyűzet- vagy egyéb hardvermegszakítás (*interrupt*) kezelése.

Ehhez hasonló lehetőséggel már a *BASIC* nyelvben is találkozhattunk (*ON ERROR GOTO*, *ON ERROR GOSUB*). Ehhez hasonló forma jelenik meg az *Object Pascal* nyelvben is (*ON ... DO*).

### VIII.1. Kivételkezelés használata

A kivételkezelés implementálásához a következő új nyelvi alapszavak kerülnek felhasználásra:

---

<code>try</code>	mely kritikus programrész következik
<code>catch</code>	ha probléma van, mit kell csinálni
<code>throw</code>	kifejezés, kivételkezelés átadása, kivétel(ek) deklarálása
<code>finally</code>	kritikus blokk végén biztos végrehajtódik

A kivételkezelés használata a következő formában adható meg :

```
try      {
    // azon utasítások kerülnek ide, melyek
    // hibát okozhatnak, kivételkezelést igényelnek
}
catch( típus [név])
{
    // Adott kivételtípus esetén a vezérlés ide kerül
    // ha nemcsak a hiba típusa az érdekes, hanem az
    // is, hogy például egy indexhiba esetén milyen
    // index okozott 'galibát', akkor megadhatjuk a
    // típus nevét is, amin keresztül a hibát okozó
    // értéket is ismerhetjük. A név megadása opcionális.
}
finally {
    // ide jön az a kód, ami mindenképpen végrehajtódik
}
```

A *try* blokkot követheti legalább egy *catch* blokk, de több is következhet. Ha a *try* blokk után nincs elkapó (*catch*) blokk, vagy a meglévő *catch* blokk típusa nem egyezik a keletkezett kivétellel, akkor a keretrendszer kivételkezelő felügyelete veszi át a vezérlést.

A C++ nyelv kivételkezelő lehetősége megengedi azt, hogy a *catch* blokk tetszőleges hibatípust kapjon el, míg a C# ezt kicsit korlátozza. A C# nyelvben a *catch* blokk típusa csak a keretrendszer által biztosított *Exception* osztálytípus, vagy ennek egy utód típusa lehet. Természetesen mi is készíthetünk saját kivétel-típust, mint az *Exception* osztály utódosztályát.

Ezek után nézzünk egy egyszerű példát a kivételkezelés gyakorlati használatára.

Példa:

```
...
int i=4;
int j=0;
try
{
    i=i/j;    // 0-val osztunk
}
```

```
catch (Exception )
{
    Console.WriteLine("Hiba!");
}
finally
{
    Console.WriteLine("Végül ez a Finally blokk is lefut!");
}
...
```

A fenti példában azt láthatjuk, hogy a rendszerkönyvtár használatával, például a nullával való osztás próbálkozásakor, a keretrendszer hibakivételt generál, amit elkapunk a *catch* blokkal, majd a *finally* blokkot is végrehajtjuk.

A példa egész számokhoz kapcsolódik, így meg kell jegyezni, hogy gyakran használják az egész aritmetikához kapcsolódóan a *checked* és az *unchecked* módosítókat is. Ezek a jelzők egy blokkra vagy egy függvényre vonatkozhatnak.

Ha az egész aritmetikai művelet nem ábrázolható, vagy hibás jelentést tartalmaz, akkor a *checked* blokkban ez a művelet nem kerül végrehajtásra.

Példa:

```
int i=System.Int32.MaxValue;
checked
{
    i++;    // OverflowException kivétel keletkezik
}
...
int j=System.Int32.MaxValue;
unchecked
{
    j++;    // OverflowException kivétel nem keletkezik
}
Console.WriteLine(i); // -2147483648 lesz a kiírt érték
                      // ez azonos a System.Int32.MinValue
                      // értékével
```

Természetesen nemcsak a keretrendszer láthatja azt, hogy a normális utasításvégrehajtást nem tudja folytatni, ezért kivételt generál, és ezzel adja át a vezérlést, hanem maga a programozó is. Természetesen az, hogy a programozó mikor látja szükségesnek a kivétel generálását, az rá van bízva.

Példa:

```
...
int i=4;
try
```

```

{
    if (i>3) throw new Exception(); // ha i>3, kivétel indul
}
catch (Exception )
// mivel az i értéke 4, itt folytatódik a végrehajtás
{
    Console.WriteLine("Hibát dobtál!");
}
finally
{
    Console.WriteLine("Végül ez is lefut!");
}
...

```

A kivételkezelések egymásba ágyazhatók.

Többféle abnormális jelenség miatt lehet szükség kivételkezelésre, ekkor az egyik „kivételkezelő” a másiknak adhatja át a kezelés lehetőségét, mondván „ez már nem az én asztalom, menjen a következő szobába, hátha ott a címzett” (*throw*). Ekkor nincs semmilyen paramétere a *throw*-nak. Ez az eredeti hiba-jelenség újragenerálását, továbbadását jelenti.

Példa:

```

...
int i=4;
try
{
    if (i>3) throw new Exception(); // ha i>3, kivétel indul
}
catch (Exception )
{
    Console.WriteLine("Hibát dobtál!");
    throw;           //a hiba továbbítása
    // ennek hatására , ha a program nem kezel további kivétel-
    // elkapást, a .NET keretrendszer lesz a kivétel elkapója.
    // így szabvány hibaüzenetet kapunk, majd a
    // programunk leáll
}
...

```

## VIII.2. Saját hibatípus definiálása

Gyakori megoldás, hogy az öröklődés lehetőségét használjuk ki az egyes hibák szétválasztására, saját hibatípust. Például készítünk egy *Hiba* osztályt, majd ebből származtatjuk az *Indexhiba* osztályt. Ekkor természetesen egy hibakezelő lekezel az *Indexhibát* is, de ha a kezelő formális paraméterezése érték szerinti,

### VIII. Kivételkezelés

---

akkor az *Indexhibára* jellemző plusz információk nem lesznek elérhetők! Mivel egy őstípus egyben dinamikus utódtípusként is megjelenhet, ezért a hibakezelő blokkokat az öröklés fordított sorrendjében kell definiálni.

Példa:

```
using System;
public class Hiba:Exception
{
    public Hiba(): base()
    {
    }
    public Hiba(string s): base(s)
    {
    }
}
public class IndexHiba:Hiba
{
    public IndexHiba(): base()
    {
    }
    public IndexHiba(string s): base(s)
    {
    }
}
...
int i=4;
int j=0;
try
{
    if (i>3) throw new Hiba("Nagy a szám!");
}
catch (IndexHiba h )
{
    Console.WriteLine(h);
}
catch (Hiba h )    // csak ez a blokk hajtódik végre
{
    Console.WriteLine(h);
}
catch (Exception )
{
    Console.WriteLine("Hiba történt, nem tudom milyen!");
}
finally    // és természetesen a finally is
{
    Console.WriteLine("Finally blokk!");
}
```

Ahogy korábban láttuk, minden objektum a keretrendszer *Object* utódjának tekinthető, ennek a típusnak pedig a *ToString* függvény a része, így egy tetszőleges objektum kiírása nem jelent mást, mint ezen *ToString* függvény meghívását.

A fenti példa így meghívja az *Exception* osztály *ToString* függvényét, ami szövegparaméter mellett kiírja még az osztály nevét és a hiba helyét is.

Befejezésül nézzük meg a korábban látott verem példa megvalósítását kivételkezeléssel kiegészítve.

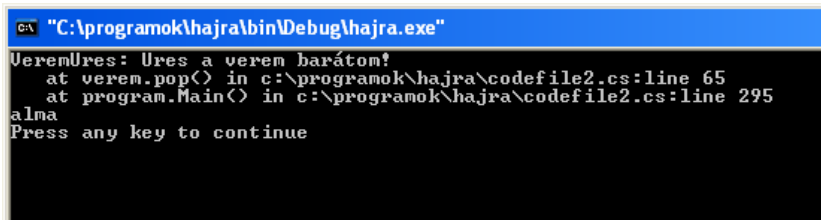
Példa:

```
using System;
class verem
{
    Object[] x;           // elemek tárolási helye
    int mut;              // veremmutató
    public verem(int db)   // konstruktor
    {
        x=new object[db]; // helyet foglalunk a vektornak
        mut=0;             // az első szabad helyre mutat
    }
    // NEM DEFINIÁLUNK DESTRUKTORT,
    // MERT AZ AUTOMATIKUS SZEMÉTKYŰJTÉSI
    // ALGORITMUS FELSZABADÍTVÁ A MEMÓRIÁT
    public void push(Object i)
    {
        if (mut<x.Length)
        {
            x[mut++]=i; // beillesztettük az elemet
        }
        else throw new VeremTele("Ez bizony tele van!");
    }
    public Object pop()
    {
        if (mut>0) return x[--mut];
        // ha van elem, akkor visszaadjuk a tetejéről
        else throw new VeremÜres("Üres a verem barátom!");
    }
}

public class VeremTele:Exception
{
    public VeremTele(): base("Tele a verem!")
    {
    }
}
```

```
        public VeremTele(string s): base(s)
        {
        }
    }
    public class VeremUres:Exception
    {
        public VeremUres(): base("Üres a verem!")
        {
        }
        public VeremUres(string s): base(s)
        {
        }
    }
    class program
    {
        public static void Main()
        {
            verem v=new verem(6);
            try
            {
                Console.WriteLine(v.pop());
                // mivel a verem üres, kivételt fog dobni
            }
            catch (VeremUres ve)
            // amit itt elkapunk
            {
                Console.WriteLine(ve);
            }
            v.push(5);
            v.push("alma");
            Console.WriteLine(v.pop());
        }
    }
}
```

A program futása az alábbi eredményt adja:



```
C:\programok\hajra\bin\Debug\hajra.exe
VeremUres: Üres a verem barátom!
   at verem.pop() in c:\programok\hajra\codefile2.cs:line 65
   at program.Main() in c:\programok\hajra\codefile2.cs:line 295
alma
Press any key to continue
```

11. ábra



### VIII.3. Feladatok

1. Mikor is hogyan használjuk a kivételkezeléseket?
2. Mit jelent a *checked*, *unchecked* kulcsszó, hogyan tudjuk használni?
3. Hogyan tudunk saját kivételt (típust) definiálni?
4. Olvassa be a másodfokú egyenlet paramétereit. Számolja ki a gyököket, ha a diszkrimináns negatív szám, használja a rendszerkönyvtár kivételkezelési lehetőségét!
5. Készítsen *Diszkrimináns\_negatív* kivételt. Oldja meg az előző feladatot ennek segítségével!

## IX. Input-Output

A be- és kiviteli szolgáltatások nem részei a nyelvnek. A programok a szabványos könyvtárban lévő függvények segítségével tartják a környezetükkel a kapcsolatot. Ez a fajta kapcsolattartás nem csak a C# nyelvben írt programok sajátossága. A C++-ból származtatható nyelvek hasonlóan használják az input-output műveleteket, mert így azok egyszerűbben és rugalmasabban kezelhetők.

### IX.1. Standard input-output

Minden klasszikus konzolprogram végrehajtása esetén automatikusan használhatjuk a *System* névtér *Console* osztályát, amely a beolvasás (standard input, billentyűzet), kiírás (standard output, képernyő) és hibairási (standard error, képernyő) műveleteket nyújtja. Ezek a *Console* osztály *In*, *Out* és *Error* tulajdonságaiban vannak hozzárendelve a be- és kiviteli eszközeinkhez. Az *In* egy *TextReader*, míg az *Out* és az *Error* *TextWriter* típus lesz. (A *TextReader*, *TextWriter* a karakteres adatfolyam osztályai.) Természetesen lehetőségünk van ezen tulajdonságok újradefiniálásával az alapértelmezés megváltoztatására is.

Mielőtt röviden áttekintjük a legfontosabb lehetőségeket, meg kell jegyezni, hogy Windows alkalmazás készítésekor ezek a függvények nem használhatók. A grafikus felületű eszközöket a könyv második részében nézzük át.

A *Console* osztály végleges, nem lehet belőle új típust származtatni.

Kiírás:

```
Write(...); // a paramétereket kiírja
WriteLine(...); // kiírás, majd soremelés
```

Mindkét utasítás újradefiniált formájú, tehát léteznek a *Write(int)*, *Write(double)* stb. könyvtári utasítások. A kiíró utasításoknak létezik egy másik változata is:

```
Write( string, adat, ...);
WriteLine( string, adat, ...);
```

Ez a változat a C világából ismert megoldást valósítja meg (*printf*). Az első paraméter, mint eredményszöveg, kapcsos zárójelek { } között a második stb. paraméterek behelyettesítését végzi. A kapcsos zárójelek között a szövegek formázási lehetőségeit használhatjuk.

Ezen formázásért felelős karaktersorozat három részből állhat, ahol a formázás alakja a következő:

{sorszám[, szélesség ][:kijelzési_forma]}
---

Az első rész 0-tól kezdődően egy sorszám, azt mondja meg, hogy a szöveg utáni kiírandó adatok közül hányadikat kell behelyettesíteni.

Ha van második rész, akkor az vesszővel elválasztva következik, és a teljes adatszélességet határozza meg. Ha a szélesség pozitív, akkor az jobbra, ha negatív, akkor balra igazítást jelent az adott szélességen belül. A hiányzó karaktereket ún. *white space* (helyköz) karakterekkel tölti fel.

Ha van harmadik rész, akkor az a kettőspont után következik, és meghatározza az adat típusát, kijelzési formáját. Az adattípus jelzésére az alábbi karakterek használtak:

c	pénznembeli (currency) kijelzés
e	tudományos, tízes alapú hatványkijelzés
x	hexadecimális formátum

Ezek mellett a 0 és a # helyiérték karakterek használhatóak. A 0 biztosan megjelenítendő helyiértéket, míg a # opcionálisan – ha van oda érték, – megjelenítendő karaktert jelent.

Példa:

```
int i=5;
Console.WriteLine("Az {0}. művelet eredménye: {1}",i,4*i);
Console.WriteLine(" A szám: {0:c}",25); // pénznem : 25,00 Ft.
Console.WriteLine("A kapott összeg: {0,10:c}",25);
    // 10 karakter széles pénznem formájú kijelzés jobbra igazítva
Console.WriteLine(" A szám: {0: 0.###e+0}",25); // 2.5E+1
Console.WriteLine("A kapott szám: {0,10:00.#0}",25.1);
    // 10 széles jobbra igazított 25.10 lesz a kijelzés
Console.WriteLine("A kapott szám: {0,10:x}",25);
    // 10 széles jobbra igazított hexa alakú (19) kijelzés
```

A *System.String* osztály *Format* függvénye az iménti forma alapján tud egy eredményszöveget készíteni (formázni).

Beolvasás:

```
int Read(); // egy karaktert olvas be
```

Ha nem sikerül az olvasás, akkor -1 az eredmény. Ha egyéb hiba jelentkezik, akkor *IOException* kivételt dob a *Read* utasítás.

Példa:

```
char c=(char)Console.Read();

string ReadLine();// egy egész sort olvas
```

## IX. Input-Output

---

Példa:

```
string s=Console.ReadLine();
```

Ha nem sikerül az olvasás, akkor *OutOfMemoryException* vagy *IOException* kivételt dob a *ReadLine* utasítás.

A könyvtár nem tartalmaz típusos beolvasási lehetőséget, viszont rendelkezésre állnak a *Convert* osztály konvertálási mezőfüggvényei.

Példa:

```
string s=Console.ReadLine();
int j=Convert.ToInt32(s); // egész konvertálás
int i=Convert.ToInt32(s,16);
//konvertálás 16-os számrendszerből
Console.WriteLine(i);
```

Ha nem sikerül a konvertálás, akkor a konvertálási hibának megfelelő kivételt dob a *Convert* osztályfüggvénye. Ezt a kivételkezelést használva számok beolvasására, az alábbi példát, mint egy lehetséges megvalósítást tekinthetjük.

Nézzük meg a *Struktúrák* fejezet végén használt példánkat azzal a kiegészítéssel, hogy a *kor* mező olvasását a konverzió kivételfigyelésével egészítjük ki.

Példa:

```
using System;
struct struktúra_pelda
{
    public int kor;
    public string név;
}

class struktúra_használ
{
    public static void Main()
    {
        // struktúra_pelda vektor
        struktúra_pelda [] spv=new struktúra_pelda[5];
        int i=0;
        // beolvasás
        while(i<5)
        {
            spv[i]=new struktúra_pelda();
            Console.WriteLine("Kérem az {0}. elem nevét!",i);
            string n=Console.ReadLine();
            spv[i].név=n;
            Console.WriteLine("Kérem az {0}. elem korát!",i);
```

```
        while(true)
        {
            try
            {
                n=Console.ReadLine();
                spv[i].kor=Convert.ToInt32(n);
            }
            catch(Exception e)
            {
                Console.WriteLine("Te kis csibész, a kor az          szám!");
                Console.WriteLine("Azt add meg még egyszer! ");
                continue;
            }
            break;
        }
        i++;
    }
    // kiírás
    for(i=0;i<5;i++)
    {
        Console.WriteLine(spv[i].név);
        Console.WriteLine(spv[i].kor);
    }
}
```

## IX.2. Fájl input – output

A fájl input-output szolgáltatásokat a *System.IO* névtér osztályai nyújtják. Kétféle típusát különböztethetjük meg a fájlok írásának, olvasásának, ezek a bináris, illetve a szöveges fájl műveletek.

A könyvtári osztályok jellemző műveletei, tulajdonságai:

- Az osztályok a *System.IO.Stream*-ből származnak
- Jellemző utasítások:
  - read (olvasás)
  - write (írás)
  - seek (pozíció állítás).
- Flush, a belső puffereket üríti
- Close , zárás

### IX.2.1. Bináris fájl input-output

A bináris műveletek két alaposztálya:

- BinaryReader olvasásra,
- BinaryWriter írásra.

Mindkét osztály konstruktora – ha nem akarunk egy általános, semmihez nem kötött objektumot kapni, – egy *Stream* utódot, jellemzően *FileStream* típust vár paraméterül. A *FileStream* osztály teremti meg a program objektuma és egy fájl között a kapcsolatot.

Egy *FileStream* objektum létrehozásánál leggyakrabban négy paramétert szoktak megadni (a fájl nevét és a módot kötelező):

- a fájl nevét
- fájl mód paramétert:
  - FileMode.Open (létező fájl),
  - FileMode.Append (létező végére),
  - FileMode.Create (új fájl)
- fájllelérést,
  - FileAccess.Read,
  - FileAccess.ReadWrite,
  - FileAccess.Write.
- megosztás módja:
  - FileShare.None,
  - FileShare.Read,
  - FileShare.ReadWrite,
  - FileShare.Write.

Természetesen a *FileStream* konstruktornak sok változata van, ezek részletezését az online dokumentációban lehet olvasni.

Bináris állományoknál lehetőségünk van még a fájlmutató állítására is (*Seek*), ezzel egy állomány különböző pozícióiba végezhetünk fájl műveleteket.

Egy fájlrendszerbeli állomány eléréséhez a rendszerkönyvtár *File* és *Fileinfo* osztályai is lehetőséget nyújtanak. A *File* osztály függvényei statikusak, míg a *Fileinfo* osztályból példányt kell készíteni.

A legfontosabb *File* statikus függvények:

- |  |                               |
|--|-------------------------------|
| • FileStream f=File.Create(fájlnév),       | //új fájl létrehozása         |
| • FileStream f=File.Open(fájlnév),         | //fájl megnyitása             |
| • StreamWriter f=File.CreateText(fájlnév), | //fájl létrehozása            |
| • StreamReader f=File.OpenText(fájlnév),   | //fájl nyitás                 |
| • File.Exists(fájlnév)                     | //létezik-e az adott állomány |

Könyvtárakkal kapcsolatosan a *Directory* osztály statikus függvényei állnak rendelkezésre.

Ha megnyitottunk egy bináris állományt, akkor írásra a legfontosabb *BinaryWriter* függvények a következők:

- Write(adat) // több példányban létezik, bármely  
// alaptípust kiír.
- Flush() // pufferek ürítése
- Close() // fájlzáras
- Seek(mennyit, honnan) // fájlmutató pozicionálása

A legfontosabb *BinaryReader* függvények:

- Read(...) // több példányban létezik, bármely  
// alaptípust beolvas.
- ReadInt32() // egész szám beolvasása  
// más alaptípusokra is létezik
- Close() // fájlzáras

A fájl olvasási műveletek fájl vége esetén *EndOfFileException* kivételt dob-  
nak, így ha nem tudjuk, mennyi adat van egy állományban, akkor ennek  
megfelelően *try* blokkban kell az olvasást végezni!

Ezek után nézzünk egy rövid példát:

```
using System;
using System.IO;
class fileteszt
{
    private static string nev = "Test.data";
    public static void Main(String[] args)
    {
        // új állomány létrehozása
        if (File.Exists(nev))
        {
            Console.WriteLine("{0} már létezik!", nev);
            return;
        }
        // FileStream létrehozása
        FileStream fs = new FileStream(nev, FileMode.CreateNew);
        // a filestream hozzárendelése bináris íráshoz.
        BinaryWriter w = new BinaryWriter(fs);
        // adatkiírás.
```

```
        for (int i = 0; i < 5; i++)
        {
            w.Write( i);
        }
        w.Close();
        fs.Close(); // fájlzárás
        // Create the reader for data.
        fs = new FileStream(nev, FileMode.Open, FileAccess.Read);
        BinaryReader r = new BinaryReader(fs);
        // olvasás.
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine(r.ReadInt32());
        }
        w.Close();
    }
}
```

### IX.2.2. Szöveges fájl input-output

Szöveges fájl input-output műveletek alapsztályai, ahogy már korábban is volt róla szó, a *TextReader* és *TextWriter* osztályok. Ezek az osztályok absztrak-tak, az ezekből származó *StreamReader* és *StreamWriter* osztályok jelentik a gyakorlatban a szöveges állományokkal kapcsolatos lehetőségeket.

Ha megnyitottunk egy szöveges állományt, akkor írásra a legfontosabb *StreamWriter* függvények a következők:

- Write(adat) // több példányban létezik, bármely  
// alaptípust kiír.
- WriteLine(s) // egy sort ír ki
- Flush() // pufferek ürítése
- Close() // fájlzárás

A legfontosabb *StreamReader* függvények:

- int Read() // karaktert beolvas.
- ReadLine() // egész sort beolvas  
// más alaptípusokra is létezik
- Close() // fájlzárás
- Peek() // a következő karaktert kapjuk a fájlból  
// de nem módosul a fájlpozíció



Példa:

```
using System;
using System.IO;
class Test
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter("Teszt.txt");
        //kiírás.
        sw.Write("Szia");
        sw.WriteLine("EZ A FOLYTATÁS.");
        sw.WriteLine("Újabb sor.");
        sw.Close();
        // fájlolvasás.
        StreamReader sr = new StreamReader("Teszt.txt");
        string sor;
        // olvasás soronként, amíg van adat
        while ((sor = sr.ReadLine()) != null)
        {
            Console.WriteLine(sor);
        }
    }
}
```

A .NET keretrendszer könyvtára a bináris és a szöveges fájlok mellett sok egyéb típusú fájl i/o műveleteit is támogatja, ilyenek például az *XmlTextReader* és az *XmlTextWriter*, amelyek XML állományok kezelését segítik elő, vagy a *HtmlTextWriter*, és a *HtmlTextReader*, amelyek HTML állományok írását, olvasását végzik.

A fájl input-output szolgáltatásokra is, mint általában minden könyvtári szolgáltatásra igaz az, hogy a megfelelő megoldási elképzelés kialakításához érdemes az online, MSDN segítség szolgáltatását is igénybe venni.

### IX.3. Feladatok

1. Mit jelent a standard input-output lehetősége?
2. Milyen formázási lehetőségeket ismer?
3. Mi a különbség a bináris és szöveges fájl között?
4. Írja ki a számot decimális, hexadecimális formában balra, majd jobbra igazítva 20 szélességben!
5. Tárolunk egy nevet és egy számot, írjuk ki ezeket adatok *.bin* névvel bináris, majd *adatok.txt* névvel szöveges formában!

## ***X. Párhuzamos programvégrehajtás***

A feladatok gyorsabb, hatékonyabb elvégzésének érdekében az utóbbi években speciális lehetőségként jelent meg a párhuzamos programvégrehajtás. Gondolhatunk a többfeladatos operációs rendszerekre, vagyis arra például, hogy miközben a programunkat fejlesztjük, és valamilyen szövegszerkesztőt használunk, a produktívabb munkavégzés érdekében, gondolkodásunkat serkentendő, kedvenc zenénket hallgathatjuk számítógépünk segítségével.

A párhuzamos programvégrehajtás programjaink szintjén azt jelenti, hogy az operációs rendszer felé több végrehajtási feladatot tudunk definiálni.

Például egy adatgyűjtési feladatban az egyik szál az adatok gyűjtését végzi, a másik szál pedig a korábban begyűjtött adatokat dolgozza fel.

A .NET keretrendszer, ahogy több más fejlesztőeszköz is, lehetőséget ad arra, hogy egy program több végrehajtási szálát definiáljon. Ezekről a végrehajtási szálakról az irodalomban, online dokumentációban *threads*, *threading* néven olvashatunk.

### **X.1. Szálak definiálása**

Általában elmondhatjuk, hogy a párhuzamos végrehajtás során az alábbi lépéseket kell megtenni (ezek a lépések nem csak ebben a C# környezetben jellemzők):

- Definiálunk egy függvényt, aminek nincsenek paraméterei és a visszatérési típusa *void*. Ez több rendszerben kötelezően *run* névre hallgat.
- Ennek a függvénynek a segítségével egy függvénytípust, delegátat definiálunk. Könyvtári szolgáltatásként a *ThreadStart* ilyen, használhatjuk ez is, ahogy a következő példa mutatja.
- Az így kapott delegátat felhasználva készítünk egy *Thread* objektumot.
- Meghívjuk az így kapott objektum *Start* függvényét.

A párhuzamos végrehajtás támogatását biztosító könyvtári osztályok a *System.Threading* névtérben találhatók.

Ezek után az első példaprogram a következőképpen nézhet ki:

Példa:

```
using System;
using System.Threading;

class program
{
    public static void szal()
    {
        Console.WriteLine("Ez a szálprogram!");
    }
    public static void Main()
    {
        Console.WriteLine("A főprogram itt indul!");
        ThreadStart szalmutato=new ThreadStart(program.szal);
        // létrehoztuk a fonal függvénymutatót, ami a
        // szal() függvényre mutat
        Thread fonal=new Thread(szalmutato);
        // létrehoztuk a párhuzamos végrehajtást végző objektumot
        // paraméterül kapta azt a delegáltat (függvényt) amit
        // majd végre kell hajtani
        fonal.Start();
        // elindítottuk a fonalat, valójában a szal() függvényt
        // a fonal végrehajtása akkor fejeződik be amikor
        // a szal függvényhívás befejeződik
        Console.WriteLine("Program vége!");
    }
}
```

## **X.2. Szálak vezérlése**

Ahogy az előző példában is láhattuk, egy szál végrehajtását a *Start* függvény hívásával indíthatjuk el, és amikor a függvény végrehajtása befejeződik, a szál végrehajtása is véget ér. Természetesen a szál végrehajtása független a *Main* főprogramtól.

A természetes végrehajtás mellett szükség lehet a szál végrehajtásának befolyásolására is. Ezek közül a legfontosabbak a következők:

- *Sleep* (millisec): statikus függvény, az aktuális szál végrehajtása várakozik a paraméterben megadott ideig.
- *Join*(): az aktuális szál befejezését várjuk meg

## *X. Párhuzamos programvégrehajtás*

---

- Interrupt(): az aktuális szál megszakítása. A szál objektum interrupt hívása ThreadInterruptedException eseményt okoz a szál végrehajtásában.
- Abort(): az aktuális szál befejezése, valójában a szálban egy AbortThreadException kivétel dobását okozza, és ezzel befejeződik a szál végrehajtása. Ha egy szálát abortáltunk, nem tudjuk a Start függvénnyel újraindítani, a start utasítás ThreadStateException kivételt dob. Ezt a kivételt akár mi is elkaphatjuk, és ekkor, ha úgy látjuk, hogy a szálát mégsem kell „abortálni”, akkor a Thread.ResetAbort() függvényhívással hatályon kívül helyezhetjük az Abort() hívását.
- IsAlive: tulajdonság, megmondja, hogy a szál élő-e
- Suspend(): a szál végrehajtásának felfüggesztése
- Resume(): a felfüggesztés befejezése, a szál továbbindul

Ezek után nézzük meg a fenti programunk módosítását, illusztrálva ezen függvények használatát.

Példa:

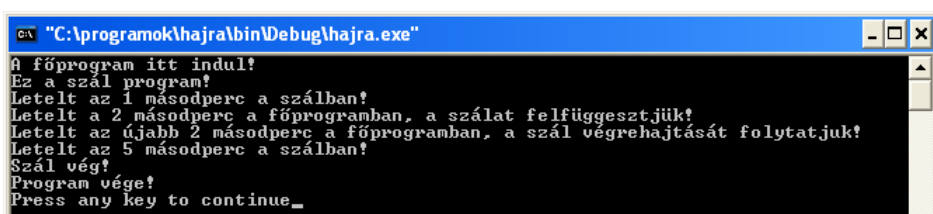
```
using System;
using System.Threading;

class program
{
    public static void szal()
    {
        Console.WriteLine("Ez a szálprogram!");
        Thread.Sleep(1000);
        // egy másodpercet várunk
        Console.WriteLine("Letelt az 1 másodperc a szálban!");
        Thread.Sleep(5000);
        Console.WriteLine("Letelt az 5 másodperc a szálban!");
        Console.WriteLine("Szál vég!");
    }

    public static void Main()
    {
        Console.WriteLine("A főprogram itt indul!");
        ThreadStart szalmutato=new ThreadStart(program.szal);
        // létrehoztuk a fonál függvényt, ami a
        // szal() függvényre mutat
        Thread fonal=new Thread(szalmutato);
        // létrehoztuk a párhuzamos végrehajtást végző objektumot
        // paraméterül kapta azt a delegáltat (függvényt), amit
        // majd végre kell hajtani
        fonal.Start();
        // elindítottuk a fonalat, valójában a szal() függvényt
```

```
Thread.Sleep(2000);  
// várunk 2 másodpercet  
Console.WriteLine("Letelt a 2 másodperc a főprogramban,  
a szálát felfüggesztjük!");  
  
fonal.Suspend();  
// fonal megáll  
Thread.Sleep(2000);  
Console.WriteLine("Letelt az újabb 2 másodperc a  
főprogramban, a szál végrehajtását folytatjuk!");  
fonal.Resume();  
// fonal újraindul  
fonal.Join();  
// megvárjuk a fonalbefejeződést  
Console.WriteLine("Program vége!");  
}  
}
```

A program futása az alábbi eredményt adja:



12. ábra

Ha több szálát is definiálunk, vagy akár csak egyet, mint a fenti példában, szükségünk lehet a végrehajtási szál prioritásának állítására. Ezt a szálobjektum *Priority* tulajdonság állításával tudjuk megtenni az alábbi utasítások valamelyikével:

```
fonal.Priority=ThreadPriority.Highest      // legmagasabb  
fonal.Priority=ThreadPriority.AboveNormal  // alap fölött  
fonal.Priority=ThreadPriority.Normal       // alapértelmezett  
fonal.Priority=ThreadPriority.BelowNormal  // alap alatt  
fonal.Priority=ThreadPriority.Lowest       // legalacsonyabb
```

A *Thread* osztály további tulajdonságait az online dokumentációban találhatjuk meg.

### **X.3. Szálak szinkronizálása**

Amíg a szálak végrehajtásánál adatokkal, egyéb függvényhívással kapcsolatos feladataink nincsenek, a szálak egymástól nem zavartatva rendben elvégzik feladataikat. Ez az ideális helyzet viszont ritkán fordul elő.

Tekintsük azt a példát, mikor egy osztály az adatmentés feladatát végzi. (Ezt a példánkban egyszerűen a képernyőre írással valósítjuk meg.)

Definiáljunk két szálát, amelyek természetesen a programosztály adatmentését használják. Az előző forráskódot kicsit átalakítva az alábbi programot kapjuk:

Példa:

```
using System;
using System.Threading;

class adatok
{
    public void mentes(string s)
    {
        Console.WriteLine("Adatmentés elindul!");
        for(int i=0;i<50;i++)
        {
            Thread.Sleep(1);
            Console.Write(s);
        }
        Console.WriteLine("");
        Console.WriteLine("Adatmentés befejeződött!");
    }
}

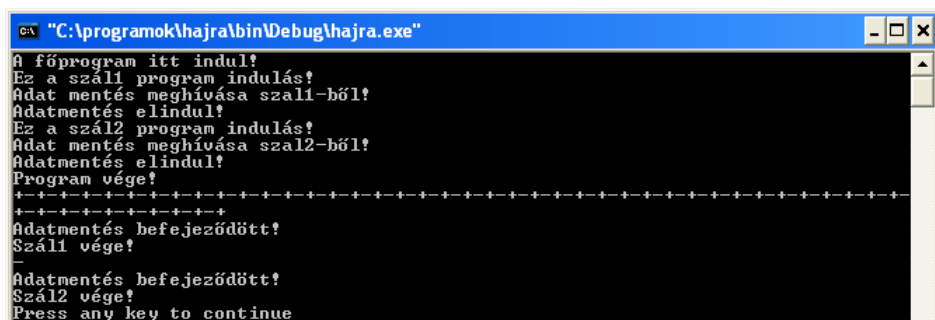
class program
{
    public static adatok a=new adatok();
    // adatmentésmező a programban

    // szál definiálás
    public static void szall()
    {
        Console.WriteLine("Ez a szál program indulás!");
        // egy másodpercet várunk
        Console.WriteLine("Adatmentés meghívása szál-ból!");
        a.mentes("+");
        Console.WriteLine("Szál vége!");
    }
}
```

```
// szál1 definiálás
public static void szal2()
{
    Console.WriteLine("Ez a szál2 programindulás!");
    // egy másodpercet várunk
    Console.WriteLine("Adatmentés meghívása szal2-ből!");
    a.mentes("-");
    Console.WriteLine("Szál2 vége!");
}

public static void Main()
{
    Console.WriteLine("A főprogram itt indul!");
    ThreadStart szalmutato1=new ThreadStart(program.szal1);
    ThreadStart szalmutato2=new ThreadStart(program.szal2);
    // létrehoztuk a fonal függvénymutatót, ami a
    //
    Thread fonal1=new Thread(szalmutato1);
    Thread fonal2=new Thread(szalmutato2);
    fonal1.Start();
    fonal2.Start();
    //
    //
    Console.WriteLine("Program vége!");
}
}
```

A programot futtatva az alábbi eredményt kapjuk:



```
C:\programok\hajra\bin\Debug\hajra.exe
A főprogram itt indul!
Ez a szál1 program indulás!
Adat mentés meghívása szal1-ből!
Adatmentés elindul!
Ez a szál2 program indulás!
Adat mentés meghívása szal2-ből!
Adatmentés elindul!
Program vége!
+-----+
+-----+
Adatmentés befejeződött!
Szál1 vége!
-
Adatmentés befejeződött!
Szál2 vége!
Press any key to continue
```

13. ábra

## *X. Párhuzamos programvégrehajtás*

---

Ebből a futási eredményből az látszik, hogy a *fonal1* és a *fonal2* mentése, azaz ugyanannak a függvénynek a végrehajtása párhuzamosan történik!

(Csak azért került egy kis várakozás a ciklusba, hogy szemléletesebb legyen a párhuzamos függvényfutás, a + és – karakterek váltott kiírása.)

A + és a – karakterek váltakozó kiírása, azaz a két fonal törzsének váltakozó végrehajtása nem jár különösebb gonddal. De gondoljunk például egy olyan esetre, amikor az adatmentő függvény soros portra írja a mentési adatokat archiválási céllal, vagy vezérel valamilyen eszközt. Ekkor lényeges az adatok „sorrendje”, és ez a fajta futási eredmény nem kielégítő. Tehát alapvető igény a többszálú végrehajtás esetén, hogy biztosítani tudjuk egy függvény, egy utasítás megszakításmentes (ezt gyakran *thread safe* lehetőségnek nevezik), ún. szinkronizált végrehajtását.

Ha egy adat módosítását, egy függvény hívását egy időpontban csak egy végrehajtási szálnak szeretnénk engedélyezni, akkor erre több lehetőségünk is kínálkozik. Csak a leggyakrabban használt lehetőségeket említjük, hiszen nem tudjuk, és nem is célunk az összes lehetőség referenciaszerű felsorolása.

Automatikus szinkronizációnak nevezi a szakirodalom azt a lehetőséget, amikor egy egész osztályra „beállítjuk” ezt a szolgáltatást. Ehhez két dolgot kell megtennünk:

1. A *Synchronization()* attribútummal kell jelölni az osztályt. Az attribútumokkal a következő fejezet foglalkozik, így most egyszerűen fogadjuk el ezt az osztályra, függvényre, változóra állítható információt.
2. Az osztályt a *ContextBoundObject* osztályból kell származtatni.

Példa:

```
[Synchronization()]
class szinkronosztaly: ContextBoundObject
{
    int i=5;                // egy időbencsak egy szál fér hozzá
    public void Novel()     // ezt a függvényt is csak egy szál
                           // tudja egyszerre végrehajtani
    {
        i++;
    }
    ...
}
```

Az automatikusan szinkronizált osztályoknál a statikus mezőket többen is elérhetik. Ezt orvosolja a függvények, blokkok szinkronizációja, amit gyakran manuális szinkronizációnak is neveznek.



Egy függvény szinkronizált végrehajtását a legegyszerűbben a *MethodImplAttribute* attribútum beállításával érhetjük el. Ezt mind példány, mind pedig statikus függvényre alkalmazhatjuk.

```
[MethodImplAttribute(MethodImplOptions.Synchronized)]
public void safe_fv()
{
    ...
}
```

Egy függvény szinkronizált végrehajtására kínál egy másik megoldást a *Monitor* osztály *enter* és *exit* függvénye. Amíg egy monitor a belépéssel véd egy végrehajtási blokkot, addig egy másik szál azt nem tudja meghívni. Ekkor a következőképpen módosul az adatmentés függvénye:

```
public void mentes(string s)
{
    Monitor.Enter(this);

    Console.WriteLine("Adatmentés elindul!");
    for(int i=0;i<50;i++)
    {
        Thread.Sleep(1);
        Console.Write(s);
    }
    Console.WriteLine("");
    Console.WriteLine("Adatmentés befejeződött!");
    Monitor.Exit(this);
}
```

Az eredményben azt láthatjuk, hogy az a blokk, amit a *Monitor* véd, megszakítás nélkül fut le.



14. ábra

## *X. Párhuzamos programvégrehajtás*

---

Hasonló eredményt kaphatunk, ha a C# nyelv *lock* utasításával védjük a kívánt blokkot. Ha a fenti függvényt a *lock* nyelvi utasítással védjük, azt a fordító a következő alakra alakítja:

```
Monitor.Enter(this);  
try  
{  
    utasítások;  
}  
finally  
{  
    Monitor.Exit(this);  
}
```

A teljesség igénye nélkül a *Monitor* osztály két függvényéről még szót kell ejtenünk. Az egyik a *Wait*, ami a nevéből sejthetően leállítja a szál végrehajtását, és a paraméterobjektum zárását befejezi.

```
Monitor.Wait(objektum);
```

A függvény használathoz a *Pulse* függvény is hozzátartozik, ami az objektumra várakozó szálakat továbbbengedi.

```
Monitor.Pulse(objektum);
```

Hasonló szolgáltatást ad a Windows API *mutex* (*mutual exclusive*, kölcsönös kizárás) lehetőségének megfelelő *Mutex* osztály. Lényeges különbség a monitorhoz képest, hogy a kizárólagos végrehajtáshoz megadhatjuk azt is, hogy ez mennyi ideig álljon fenn. (*WaitOne* függvény)

```
class adatok  
{  
    Mutex m=new Mutex();  
    public void mentes(string s)  
    {  
        m.WaitOne();  
        Console.WriteLine("Adatmentés elindul!");  
        for(int i=0;i<50;i++)  
        {  
            Thread.Sleep(1);  
            Console.Write(s);  
        }  
        Console.WriteLine("");  
        Console.WriteLine("Adatmentés befejeződött!");  
        m.Release();  
    }  
}
```

Még manuálisabb adat vagy utasítás szinkronizációt biztosít a *ReadWriterLock*, valamint az *Interlocked* osztály is. Ezek és a további szinkronizációs lehetőségek ismertetése túlmutat e könyv keretein.

Ha könyvtári szolgáltatásokat veszünk igénybe, például MSDN Help, az egyes hívások, osztályok mellett olvashatjuk, hogy *thread safe*-e vagy nem a használata, attól függően, hogy a szinkronizált hozzáférés biztosított-e vagy sem.

A száakkal kapcsolatban befejezésül meg kell említeni a *[STAThread]* illetve az *[MTAThread]* attribútumot, ami azt mondja meg, hogy *Single* (egy) vagy *Multi* (több) szál tartalmazó alkalmazásként definiáljuk a programunkat COM komponensként való együttműködésnél. Alapértelmezés a *[STAThread]* használata, ami például a Windows alkalmazások „Drag and Drop” jellemzők használata esetén követelmény is.

## X.4. Feladatok

1. Mit értünk párhuzamos programvégrehajtás alatt?
2. Hogyan tudunk szálakat definiálni?
3. Mit jelent a *lock* utasítás? Milyen könyvtári szolgáltatás felel meg ennek az utasításnak?
4. Készítsen programot, amely két süket ember beszélgetését szimulálja! Mindketten egy-egy fájlban tárolják a mondókájukat!
5. Használjunk szinkronizálást, módosítsuk az előző feladatot úgy, hogy a mondatvégeknél lehet a másik szereplőnek átvenni a beszéd fonalát!

## ***XI. Attribútumok***

Képzeliük el, hogy definiáltunk egy új típust (osztályt), minden adatnak, függvénymezőnek elkészítettük a megfelelő kezelését, jelentését, használatát, azaz a típusunk használatra kész. Mégis, egyes mezők vagy esetleg az egész típus mellé bizonyos plusz információkat szeretnénk hozzárendelni.

Példaként nézzünk két ilyen, a mindennapi programozási feladataink során is előforduló esetet!

Első példa:

Egy program általában több típussal, adattal dolgozik. Ezek között az adatok között lehet néhány, amelyek értékére a program következő indulásakor is szükségünk lehet. Ezeket a program végén elmentjük a registry-be. (A registry a korábbi Windows ini állományokat váltja fel, programhoz kötődő információkat tudunk ebben az operációs rendszerbeli adatbázisban tárolni.) Természetesen innen be is olvashatjuk ezeket az adatokat. A programunk elvégzi ezt a mentést is, a visszaolvasást is. Ha viszont felteszi valaki a kérdést a programban, hogy jelenleg kik azok, akiket a registry-ben is tárolunk, akkor erre nincs általános válasz! Minden programba bele kell kódolni azt, hogy melyek a mentett adataink! Hasznos lenne, ha nem kellene ezt megtenni, csak jelölhetnénk ezeket a mezőket, és ezt a jelölést később kikereshetnénk.

Második példa:

Ez a példa talán kicsit távolabb áll az informatikától, de nem kevésbé életszerű. Definiáljuk – vagy nem is kell definiálnunk, mert egyébként is léteznek – az emberek különböző csoportjait! Ebbe a definícióba értsük bele a normális használatához szükséges összes tulajdonságot (nem, foglalkozás, életkor, végzettség, ...)! Emellett szükségünk lehet mondjuk egy olyan információra, hogy például az illető fradidrukker-e! Természetesen az alaptulajdonságok közé is felvehetnénk ezt a jellemzőt, de mivel ennek az adatnak a típus tényleges működéséhez semmi köze nincs – nem fradidrukker is végezheti rendesen a dolgát –, ezért ez nem lenne szerencsés.

Az ilyen plusz információk elhelyezésére nyújt lehetőséget a C# nyelvben az attribútum.

Az attribútumok olyan nyelvi elemek, melyekkel fordítási időben osztályokhoz, függvényekhez vagy adatmezőkhöz információkat rendelhetünk. Ezek az információk aztán futási időben lekérdezhetők.

## XI.1. Attribútumok definiálása

Mivel a nyelvben gyakorlatilag minden osztály, így ha egy új attribútumot szeretnénk definiálni, akkor valójában egy új osztályt kell definiálnunk. Annyi megkötés van csak, hogy ezen attribútumot leíró osztálynak az *Attribute* bázis-osztályból kell származnia.

Ezek alapján a *fradi\_szurkoló* attribútum definiálása a következőképpen történhet:

Példa:

```
class fradi_szurkoló:Attribute
{
    ...
}
```

Egy attribútum, ahogyan egy kivétel is, már a nevével információt hordoz. Természetesen lehetőségünk van ehhez az osztályhoz is adatmezőket definiálni, ha úgy ítéljük meg, hogy a típusnév, mint attribútum még kevés információval bír. Ekkor – mint egy rendes osztályhoz – adatmezőket tudunk definiálni. Az adattípusokra annyi megkötés van, hogy felhasználói osztálytípus nem lehet. Ellenben lehetnek a rendszerbeli alaptípusok (*bool*, *int*, *float*, *char*, *string*, *double*, *long short*, *object*, *Type*, *publikus enum*). Adatok inicializálására konstruktort definiálhatunk.

Egy attribútum osztályban kétféle adatot, paramétert különböztethetünk meg. Pozicionális és nevesített paramétert. Pozicionális paraméter a teljesen normális konstruktorparaméter. Nevesített paraméternek nevezzük a publikus elérésű adat- vagy tulajdonságmezőket. A tulajdonságmezőnek rendelkezni kell mind a *get*, mind a *set* taggal. A nevesített paramétereknek úgy adhatunk értéket, mintha normál pozicionális konstruktorparaméter lenne, csak a konstruktorban nincs semmilyen jelölésre szükség. A konstrukció kicsit hasonlít a C++ nyelvben használatos alapértelmezett (*default*) paraméterek használatához.

Ezek után nézzük meg a *fradi\_szurkoló* attribútumunkat két adattal kiegészítve. Az egyik legyen az az információ, hogy hány éve áll fenn ez a viszony, míg a másik az, hogy az illető törzsszurkoló-e?

Példa:

```
class fradi_szurkoló:Attribute
{
    private int év;           // hány éve szurkoló
    private bool törzsgárda;  // törzsszurkoló-e
    public fradi_szurkoló(int e)
    {
        év=e;               // konstruktor beállítja a normal paramétert
    }
}
```

```
public bool Törzsgárda
{
    get
    {
        return törzsgárda;
    }
    set
    {
        törzsgárda=value;
    }
}
```

Ekkor a *Törzsgárda* tulajdonságmezőt nevesített paraméternek hívjuk. Az elnevezést az mutatja, hogy erre a tulajdonságra a konstruktorhívás kifejezésében tudunk hivatkozni, mintha ez is konstruktorparaméter lenne, pedig nem is az!

Példa:

```
[fradi_szurkoló(5,Törzsgárda=true)]
```

### XI.2. Attribútumok használata

Az eddig megismert attribútumjellemzők definiálását, használatát, majd a beillesztett információ visszanyerését nézzük meg egy példán keresztül! Mielőtt ezt tennénk, meg kell jegyezni, hogy az információvisszanyerési lehetőségek a típusinformáció lekérdezés lehetőségéhez illeszkednek. Ennek bázisosztálya a *Type*. Ezt a könyvtári szolgáltatást az irodalom gyakran reflekciónak nevezi. Ezeket a lehetőségeket csak olyan mértékben nézzük, amennyire a példaprogram megkívánja.

A típusinformáció szolgáltatás a *Reflection* névtérben található, tehát a használatához a

```
using System.Reflection;
```

utasítást kell a program elejére beírni.

A típusinformáció visszanyeréséhez jellemzően a *Type* osztály *GetType()* statikus függvényét kell meghívni, ami egy paramétert vár tőlünk, azt az osztálytípust, amit éppen fel szeretnénk dolgozni.

```
Type t=Type.GetType("program");
```

Ekkor a programosztályomat szeretném a *t* nevű típusleíró objektumon keresztül elemezni. A kapott *t* objektumra megkérdezhetem például, hogy osztály-e:

```
if (t.IsClass()) Console.WriteLine("Igen");
```

További lehetőségekhez a *Type* osztály online dokumentációjánál nincs jobb forrás.

A típushoz tartozó attribútumok listáját a *GetCustomAttributes()* függvényhívás adja meg. Ez eredményül egy *Attribute* vektort ad. Jellemző módon, ezen egy *foreach* ciklussal lépdélünk végig:

```
foreach (Attribute at in t.GetCustomAttributes())
{
    // feldolgozzuk az at attribútumot...
}
```

Ez a feldolgozás osztályokra (pl. program, stb.) vonatkozó attribútumokra megfelelő. Előfordulhat viszont olyan is, mikor függvényekhez vagy adatmezőkhöz rendelünk attribútumot. Ekkor először az adott típus függvényeit vagy az adatmezőit kell megkapnunk, majd ezen információk attribútumait kell lekérdeznünk.

Egy típus függvényeit a *GetMethods()* hívás szolgáltatja, eredményül egy *MethodInfo* típust ad.

```
foreach (MethodInfo m in t.GetMethods())
{
    foreach (Attribute at in m.GetCustomAttributes())
    {
        // feldolgozzuk az m függvény at attribútumát...
    }
}
```

Egy típus adatmezőit a *GetFields()* adja, eredménye *FieldInfo* típusú.

```
foreach (FieldInfo f in t.GetFields())
{
    foreach (Attribute at in f.GetCustomAttributes())
    {
        // feldolgozzuk az f adatmező at attribútumát...
    }
}
```

Ha egy attribútumosztályt definiálunk, írjuk az osztály neve után az *Attribute* szót, ahogyan azt gyakran tanácsként is megfogalmazzák a szakirodalomban,

## *XI. Attribútumok*

---

hiszen így könnyen meg lehet különböztetni a rendes osztályoktól. Természetesen ebben az esetben is elhagyhatjuk az *attribute* szócskát a használat során, mert azt a fordító odaérti.

Tehát ekkor a *fradi\_szurkoló* attribútumosztályt a következőképpen definiáljuk:

```
class fradi_szurkolóAttribute:Attribute
{
    ...
}
```

Ezen definíció esetében is használhatjuk a következő alakot:

```
[fradi_szurkoló]
...
```

Természetesen ekkor a teljes nevű hivatkozás is helyes:

```
[fradi_szurkolóAttribute]
...
```

Ezek után megnézzük a *fradi\_szurkoló* attribútumunk definiálását és használatát egy kerek példán keresztül. A példa nem használja az iménti *Attribute* kiegészítéses definíciót.

Példa:

```
using System;
using System.Reflection;

class fradi_szurkoló:Attribute
{
    private int ev;           // hány éve szurkoló
    private bool torzsszurkolo; // vajon törzsszurkoló-e
    public fradi_szurkoló(int e) // az e rendes, pozicionális
    {                           // paraméter
        ev=e;
    }
    public override string ToString()
    {
        string s="Ez az ember fradiszurkoló!
                 Évek száma:"+ev+"Törzsszurkoló:"+torzsszurkolo;
        return s;
    }
    // az alábbi Torzsszurkoló tulajdonság nevesített
    // fradi_szurkoló paraméter, mert publikus és van get és set
```



```
// része, amivel az adott torzsszurkolo logikai mezőt állíthatjuk
public bool Törzsszurkoló
{
    get
    {
        return torzsszurkolo;
    }
    set
    {
        torzsszurkolo=value;
    }
}

}

class ember
{
    string nev;
    public ember(string n)
    {
        nev=n;
    }
}

class adatok{} //nem fontos, hogy érdemi információt tartalmazzon

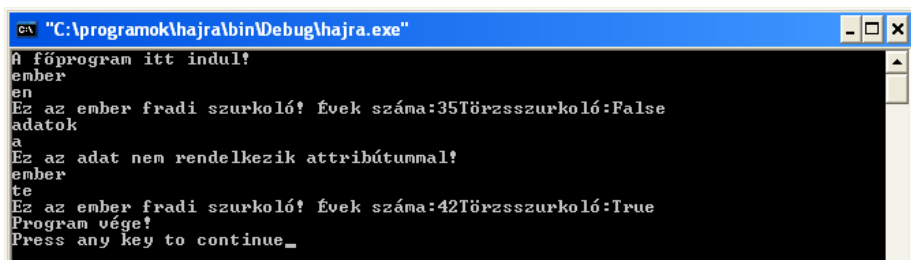
class program
{
    // attribútum beállítása
    [fradi_szurkoló(35)]
    public ember en= new ember("Zoli");
    // a normál konstruktort hívtuk meg, azok a mezők, amiket nem
    // inicializál, alapértelműs szerint kerülnek beállításra
    // 0, ha szám, false ha bool, illetve null, ha referencia
    // Egy attribútum csak a következő adat vagy függvény vagy
    // osztályra hat!!!
    // Így a következő adatok típusú változónak nincs köze
    // a fradi_szurkoló attribútumhoz
    //
    public adatok a=new adatok();
    //
    [fradi_szurkoló(42,Törzsszurkoló=true)]
    public ember te= new ember("Pali");
    // Pali már törzsszurkoló
    public static void Main()
    {
```

## XI. Attribútumok

---

```
program p=new program();
Console.WriteLine("A főprogram itt indul!");
Type t=Type.GetType("program");
foreach( FieldInfo mezo in t.GetFields())
{
    Console.WriteLine(mezo.FieldType);
    // mezőtípus kiírása
    Console.WriteLine(mezo.Name);
    // mezőnév kirása
    if ((mezo.GetCustomAttributes(true)).Length>0)
    {
        foreach (Attribute at in mezo.GetCustomAttributes(true))
        {
            // megnézzük fradiszurkoló-e
            fradi_szurkoló f=at as fradi_szurkoló;
            if (f!=null) // igen ez a mező fradiszurkoló
                Console.WriteLine(at.ToString());
        }
    }
    else
        Console.WriteLine("Ez az adat nem rendelkezik
                                attribútummal!");
}
Console.WriteLine("Program vége!");
}
```

Eredményül az alábbi kép jelenik meg:



```
GA "C:\programok\hajra\bin\Debug\hajra.exe"
A főprogram itt indul!
ember
en
Ez az ember fradi szurkoló! Évek száma:35Törzsszurkoló:False
adatok
a
Ez az adat nem rendelkezik attribútummal!
ember
te
Ez az ember fradi szurkoló! Évek száma:42Törzsszurkoló:True
Program vége!
Press any key to continue_
```

15. ábra

### XI.3. Könyvtári attribútumok

Három könyvtári attribútum áll rendelkezésünkre:

- `System.AttributeUsageAttribute`: Segítségével megmondhatjuk, hogy a definiálandó új attribútumunkat milyen típusra engedjük használni. Ha nem állítjuk be, alapértelmezés szerint mindenre lehet használni.

Az *AttributeTargets* felsorolás tartalmazza a választási lehetőségeinket:

```
[AttributeUsage(AttributeTargets.Class)]
```

```
osztályattribútum:Attribute
{
}
}
```

Ekkor az osztályattribútumok csak osztályok jelölésére használhatók. Előfordulhat, hogy egy attribútumot többször is hozzá szeretnénk rendelni egy adathoz, akkor ennek az osztálynak az *AllowMultiple* nevesített paraméterét igazra kell állítani.

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
```

- `System.ConditionalAttribute`: Egy szöveg a paramétere. Csak függvényhez kapcsolhatjuk, és az a függvény, amihez kapcsoltuk csak akkor hajtódik végre, ha a paraméterül adott szöveg definiált.

Példa:

```
...
[Conditional("alma")]
void almafuggvény()
{
    Console.WriteLine("Alma volt!");
}

#define alma
almafuggvény();           // hívás rendben

#undefine alma
almafuggvény();           // hívás elmarad
```

A feltételes végrehajtású függvények kötelezően *void* visszatérésűek, és nem lehetnek virtuálisak, vagy *override* jelzővel ellátottak! Ezeket a kiértékeléseket az „előfordító” nézi végig.

- *System.ObsoleteAttribute*: egy üzenet(szöveg) a paramétere, ha egy függvény elavult, és javasolt a mellőzése, akkor használhatjuk.

### **XI.4. Feladatok**

1. Mi az attribútum, hogy tudjuk használni?
2. Hogyan definiálhatunk saját attribútumot?
3. Mi a különbség a rendes és nevesített attribútum paraméter között?
4. Készítsünk *Hisz\_e\_a\_mesében* attribútumot! Definiáljuk az *ember* osztályt, majd alkalmazzuk az attribútumot egyes 'példányaira'!

Módosítsuk az előző attribútumot úgy, hogy meg tudjuk adni azt az időt amilyen régóta hisz valaki a mesében!

## ***XII. Szabványos adatmentés***

Az természetes, ahogy korábban már láttuk is, hogy a rendszerkönyvtárak bőséges támogatást nyújtanak az adatok fájlba mentéséhez illetve visszaolvasásához. Az alkalmazások tekintetében ez teljesen természetes igény. Az alapértelmezett lehetőségek mellett általában a környezetek olyan szabványos eszközzel is rendelkeznek, amelyek minden rendszertípusra rendelkezésre állnak, illetve a felhasználói típusokra „egyszerűen” implementálhatók. Ezzel egy olyan szolgáltatást kapnak az alkalmazások, amelyekkel szabványos ki- és bemeneti adatmozgatást végezhetnek minden erre felkészített típusra.

Ennek a gyakorlati következménye az, hogy az így felkészített rendszerben a programozónak nem kell semmilyen extra mentési stratégián gondolkodnia, hanem egyszerűen csak azokat az objektumokat kell kiválasztania, amelyek értékeit menteni akarja.

Természetesen ez a szolgáltatás is megtalálható majd minden mai környezetben. Az első klasszikus megvalósítása ennek a szolgáltatásnak talán a Visual C++ környezetében jelent meg a Microsoft Foundation Classes (MFC) szolgáltatásaként, ahol a dokumentumosztály (az adatok helye) rendelkezésre bocsát egy *Serialize* függvényt, amely ezt a szabványos alkalmazás adatmentést biztosítja. Az MFC-ben minden könyvtári típus a *Serialize* függvényben használható, míg a felhasználói típusok egy egyszerű lépéssorozat eredményeként képessé tehetők a szerializációra.

Talán innen eredeztethető a névadás is, az irodalomban ezt a szolgáltatást szerializáció (*Serialization*), szabványos mentés névvel találhatjuk meg.

### **XII.1. Könyvtári típusok szabványos mentése**

A szerializációs szolgáltatás a C# nyelvben, ahogy a standard input-output is, a rendszer könyvtári szolgáltatásának része, és ez a *System.Runtime.Serialization* névtérben található.

Ha könyvtári típusok mentéséről van szó, akkor közvetlenül ennek a névtérnek a használatára nincs is szükségünk.

Mielőtt a közvetlen lehetőségéről beszélnénk, a ki- és bemeneti adatfolyam formázásáról kell néhány szót ejtenünk.

A jelenlegi rendszerkönyvtár kétféle típusú adatfolyam formázását támogatja. Bináris, illetve Soap, XML alapú (szöveges) formázást. Ezek az adatformázó osztályok végzik a valódi adatfolyam elkészítését. A bináris adatfolyam bináris eredményfájlt, míg a Soap XML formátumú szöveges állományt hoz

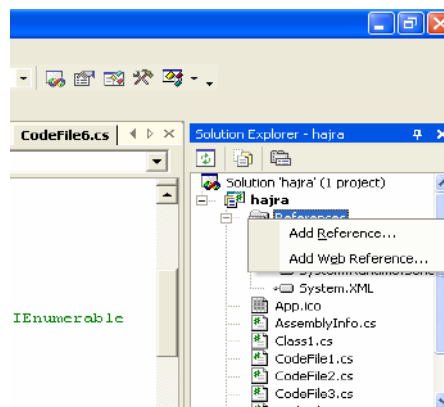
## XII. Szabványos adatmentés

létre. Ha ez a kétféle lehetőség nem elég, akkor egyéni formázó osztályok definiálhatók.

A formázó osztályok a *System.Runtime.Serialization.Formatters.Binary* és a *System.Runtime.Serialization.Formatters.Soap* névterekben találhatók.

A szerializáció folyamata az alábbi lépésekből áll:

1. Ki kell jelölni egy *FileStream* objektumot, ami a program kapcsolatát jelenti az állománnyal.
2. Definiálni kell egy formázó objektumot, ez jelenleg bináris vagy szöveges lehet. A szöveges formázó objektum létrehozásához a *System.Runtime.Serialization.Formatters.soap.dll* referenciát a projekthez kell csatolni (*Project nézet \ References \ jobb egérgomb \ Add reference*).



16. ábra

3. A formázó objektum *Serialize* függvényével mentjük az adatokat.
4. Bezárjuk a fájlkapcsolatot.

Ezek után nézzük a lehetőséget bemutató példaprogramot!

Példa:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;
```

```
class serprog
{
    // program adatai
    int i=2;
    double d=3.5;
    string s="fradi";

    public static void Main()
    {
        serprog p=new serprog();
        //1.filestream létrehozása
        FileStream fb=File.Create("ser.bin");
        // 2.Binaryformatter készítés
        FileStream fs=File.Create("ser.txt");
        BinaryFormatter b=new BinaryFormatter();
        SoapFormatter so=new SoapFormatter();
        //3.bináris mentés
        b.Serialize(fb,p.i);
        b.Serialize(fb,p.d);
        b.Serialize(fb,p.s);
        // 3.Soap mentés
        so.Serialize(fs,p.i);
        so.Serialize(fs,p.d);
        so.Serialize(fs,p.s);

        //4. fájlzárás
        fb.Close();
        fs.Close();
    }
}
```

A program magyarázatául csak annyit, hogy miután lefuttatjuk, a projekt *debug* könyvtárában (ebbe a könyvtárba kerül a lefordított program) létrejön a *ser.bin*, illetve a *ser.txt* állomány. Ha ezt megnézzük, láthatjuk, hogy míg az előbbi bináris, az utóbbi XML formátumú.

Visszaolvasás hasonló módon, csak a *formatter* objektum *Deserialize* függvényhívással végezhető el.

## XII.2. Saját típusok szabványos mentése

Az előző példa során láttuk, mit jelent az alaptípusok mentési lehetősége. A valós alkalmazásaink során azonban biztosan készíteni kell a fő programosztályon kívül egyéb felhasználói osztályokat is.

## *XII. Szabványos adatmentés*

---

Ebben az esetben a mentési folyamat visszavezeti az adatok mentését a rendszertípusok mentésére. Ahhoz, hogy ezt megtegye nekünk, egyetlen dolgot kell tennünk, a saját típusunkat a menthető, *Serializable* attribútummal kell jelölnünk.

Módosítsuk a példaprogramunkat úgy, hogy definiálunk egy *ember* típust, amit még használnia kell a programunknak. A módosított forrásszöveg a következőképpen néz ki:

Példa:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;

[Serializable]
// az attribútum hatására a formázó lekezeli a típus szerializációját
class ember
{
    string nev;
    int kor;
    public ember(string n, int k)
    {
        nev=n; kor=k;
    }
}

class serprog
{
    // program adatai
    int i=2;
    double d=3.5;
    string s="fradi";
    // a saját típus használata
    ember e=new ember("Zoli", 34);
    public static void Main()
    {
        serprog p=new serprog();
        //fájlstream létrehozása
        FileStream fb=File.Create("ser.bin");
        // Binaryformatter készítés
        FileStream fs=File.Create("ser.txt");
        BinaryFormatter b=new BinaryFormatter();
        SoapFormatter so=new SoapFormatter();
        //Bináris mentés
        b.Serialize(fb,p.i);
        b.Serialize(fb,p.d);
```



```

        b.Serialize(fb,p.s);
        b.Serialize(fb,p.e);
        //Soap mentés
        so.Serialize(fs,p.i);
        so.Serialize(fs,p.d);
        so.Serialize(fs,p.s);
        so.Serialize(fs,p.e);

        //fájl lezárása
        fb.Close();
        fs.Close();
    }
}

```

A gyakorlati munka során előfordulhat – például az osztály belső szerkezete nem engedi meg –, hogy nem bízhatjuk rá a formázóra a típusunk végigelemzését és az elemek mentését. Ebben az esetben a valódi szerializációs szolgáltatást végző függvényeket nekünk kell az osztályunkba implementálni.

Azon túl, hogy a *[Serializable]* attribútumot definiálni kell, az osztályunknak még implementálnia kell az *ISerializable* interface-t is. Ebben a deszerializáció számára egy speciális konstruktort és egy *GetObjectData* függvényt kell definiálni.

Az interface és a függvények hosszabb elemzése nélkül nézzük meg az ennek megfelelően módosított *ember* osztályunkat. A program kódja változatlan, ezért azt nem listázzuk ide.

Példa:

```

...
[Serializable]
// az attribútum hatására a formázó lekezeli a típus szerializációját
class ember:ISerializable
// az interface két plusz függvénydefiníciót ír elő
{
    string nev;
    int kor;
    public ember(string n, int k)
    {
        nev=n; kor=k;
    }
    internal ember(SerializationInfo si, StreamingContext st)
    {
        // elemek visszaállítása
        nev=si.GetString("nev");
        kor=si.GetInt32("kor");
    }
}

```

## *XII. Szabványos adatmentés*

---

```
public void GetObjectData(SerializationInfo si, StreamingContext st)
{
    // a szerializálás adatait beállítja a SerializationInfo si
    // objektumba
    si.AddValue("nev",nev);
    si.AddValue("kor",kor);
    // típusinformáció beállítása
    Type t=this.GetType();
    si.AddValue("Típusinfo",t);
}
}
```

### **XII.3. Feladatok**

1. Mit nevezünk szabványos mentésnek?
2. Milyen hasonló megvalósításokat ismer más fejlesztő rendszerben?
3. Milyen szabványos mentéseket támogat a fejlesztő rendszer?
4. A másodfokú egyenlet (VIII.3. fejezet 4. feladat) adatait mentsük el bináris formában!
5. Definiáljunk egy *szurkoló* osztályt (*név*, *kedvenc\_csapat*), és biztosítsuk ennek a típusnak szabványos menthetőségét!

## ***XIII. Könyvtárak, távoli és web könyvtárak***

Az alkalmazások készítésének egyik lényeges eleme a fejlesztők rendelkezésére álló könyvtári szolgáltatások mennyisége, minősége, illetve a használt keretrendszernek az a tulajdonsága, hogy mennyire ad lehetőséget osztott alkalmazás készítésére.

Egy alkalmazás kiszolgáló függvényei vagy a helyi gépen helyezkednek el, vagy valamelyik távoli kiszolgáló gépen.

### **XIII.1. Helyi könyvtárak használata**

#### ***XIII.1.1. Rendszerkönyvtári elemek használata***

A helyi fejlesztőrendszer könyvtárainról nyugodtan kijelenthetjük, hogy azok használata nélkül, ahogy a legelső részben is láttuk, még a legegyszerűbb program sem készíthető el. (*System* névtér, *Console* osztály, *WriteLine* függvényhívás.)

A .NET keretrendszer egyik, talán leggyakrabban használt könyvtára a *System.Collections* névtér. A névtér sok hasznos osztálydefiníciót tartalmaz, melyek közül az alábbiak a legfontosabbak:

- *ArrayList*: a méretét dinamikusan növelni képes objektumvektor. Az *ICollection* interface-t implementálja. Az *ArrayList* három jellemző tulajdonságot ír elő, ezek:

<i>IsFixedSize</i> :	megvizsgálja, fix méretű-e a vektor,
<i>IsReadOnly</i> :	olvasható-e a vektor,
<i>Item</i> :	a vektor adott indexű elemét adja, az <i>ArrayList</i> objektum indexereként használható.

- *HashTable*: olyan vektoriális adatsorozat, ahol egy kulcsindexhez tartozik egy érték. A kulcs „hash” értékéhez rendeli az adatot. Gyakran asszociatív vektornak hívják ezt a szerkezetet.

- Queue: a sor adatszerkezet megvalósítása. Az az elem megy elsőnek ki a sorból, amely elsőnek beérkezett. (*First in, first out.*)
- SortedList: rendezett lista. Hasonló szerkezet, mint a *HashTable*, melyek elemei a kulcs alapján sorba vannak rendezve. Kulcs és index alapján is az elemekhez lehet férni.
- Stack: verem adatszerkezet. Az utoljára betett elemet tudjuk mindig ki-venni. (*Last in, first out*)

Példaként tekintsük a következő *ArrayList* és *Queue* használatát bemutató mintaprogramot:

Példa:

```
using System;
using System.Collections;
public class mintacol
{
    public static void Main()
    {
        // Új ArrayList objektum létrehozása.
        ArrayList elemek = new ArrayList();
        elemek.Add( "Hajrá" );
        // alapértelmezésben a méret növelhető (IsFixedSize==false)
        elemek.Add( "Fradi" );
        elemek.Add(25);
        for (int i=0;i<elemek.Count;i++)
            Console.WriteLine(elemek[i]);

        // Új sor (Queue) objektum létrehozása.
        Queue sor = new Queue();
        // adat sorba írása
        sor.Enqueue( "mézes" );
        sor.Enqueue( "maci" );
        sor.Enqueue( 3.1415 );
        // mind az ArrayList, mind a sor implementálja az IEnumerable
        // interface-t, így a foreach használható
        foreach(object o in sor)
            Console.WriteLine(o);

        // ArrayList objektumhoz hozzáadjuk a sort.
        elemek.AddRange( sor );
        // elem kivétele a sorból
        Console.WriteLine(sor.Dequeue());
        Console.WriteLine(sor.Dequeue());
        Console.WriteLine(sor.Dequeue());
    }
}
```

```
// Kiírjuk az elemeket
Console.WriteLine( "A bővített ArrayList a következő
                    elemeket tartalmazza:" );

PrintValues( elemek );

}

// az IEnumerable alapján végiglépdelünk az elemeken
public static void PrintValues( IEnumerable adatok)
{
    IEnumerator adat = adatok.GetEnumerator();
    while ( adat.MoveNext() )
        Console.WriteLine( adat.Current );
}
}
```

### XIII.1.2. Saját könyvtári elemek használata

A keretrendszer közös nyelvi specifikációjának köszönhetően tetszőleges nyelvi könyvtárat használhatunk azonos módon, tetszőleges alkalmazásban. A példában VB függvényt definiálunk, amit egy másik alkalmazás használ:

```
Public Class Demo
    Shared Function legjobb_csapat() As String
        legjobb_csapat = "Fradi!"
    End Function
End Class
```

A fenti forráskódot tetszőleges szövegszerkesztőbe beírva, az alábbi parancssor segítségével lefordítható. (A keretrendszer mindegyik fordítója indítható parancssorból.)

```
vbc /target: library /out:DllDemo Demo.vb
```

Természetesen új projektet készítve, az osztálykönyvtár típust választva a Visual Studio.NET környezetbe is beírhatjuk ezt a pár sort, majd a *Build* menüpont segítségével lefordíthatjuk.

Az elkészült *DllDemo.dll* állomány egy tipikus könyvtári állomány lesz, amit bármely másik alkalmazás használhat, pontosan úgy, ahogy a rendszerkönyvtárakat. Készítsünk most egy C# nyelvű alkalmazást, amely szeretné használni a *DllDemo* könyvtári szolgáltatást.

A használatához az alkalmazáshoz kell csatolni (*add reference*) ezt a könyvtárállományt, majd a *using DllDemo* utasítással a *DllDemo* névtér elérhetőségét is biztosítani kell.

Példa:

```
using System;
using DllDemo;

public class minta
{
    public static void Main()
    {
        // Új demo objektum létrehozása.
        // ezt készítettük Visual Basic nyelvben
        Demo d = new Demo();
        //kíváncsiak vagyunk arra, hogy ki a legjobb csapat
        // meghívjuk a dll függvényt
        Console.WriteLine(d.legjobb_csapat());
        Console.WriteLine("Program vége");
    }
}
```

## XIII.2. Távoli könyvtárhívás

A távoli könyvtárhívás szerkezete, lehetőségei önmagában egy teljes könyvet is megérdemelnének, de a jelen tárgyalási menetbe, a könyvtári szolgáltatások típusai közé is beletartozik, ezért egy rövid bevezetést meg kell említeni erről a területről.

Egy alkalmazás operációs rendszerbeli környezetét *application domain*nek nevezzük. Az előző DLL készítési lehetőség egy *application domain*t alkot.

A független alkalmazások közti kommunikációt, távoli alkalmazáshívás lehetőségét a .NET környezetben *REMOTING* névvel említi az irodalom. Valójában hasonló lehetőségről van szó, amit a korábbi fejlesztési környezetek, *COM (Component Object Model)* néven említenek.

Az alapprobléma valójában ugyanaz, mint a DLL könyvtár esetében, egy valahol meglévő szolgáltatást szeretnénk igénybe venni. Ez DLL formában most nincs jelen, viszont az a típusú objektum, aminek ez a szolgáltatása van, egy másik gép önálló alkalmazásaként van jelen.

Így a legfontosabb kérdés az, hogy önálló alkalmazási környezetek között, melyek természetesen vagy azonos számítógépen helyezkednek el, vagy nem, a legfontosabb kérdés az, hogyan tudunk információt átadni.

Ennek a kommunikációnak legfontosabb elemei a következők:

- Kommunikációs csatorna kialakítása, regisztrálása.
- Az adatok szabványos formázása a kommunikációs csatornába írás előtt.

- Átmeneti, proxy objektum létrehozása, mely az adatcserét elvégzi a távoli objektum (pontosabban annak proxy objektuma) és a helyi alkalmazás között.
- Távoli objektum aktiválása, élettartama

Kommunikációs csatornát *Tcp* vagy *Http* alapon alakíthatunk ki. Használat előtt regisztrálni kell egy csatornát, ami egy kommunikációs porthoz kötött. Egy alkalmazás nem használhat más alkalmazás által lefoglalt csatornát. A kétféle kommunikáció használata között a legfontosabb különbség az adatok továbbításában van. A *Http* protokoll a SOAP szabványt használja az adatok továbbítására XML formában. A *Tcp* csatorna bináris formában továbbítja az adatokat.

A proxy objektum reprezentálja a távoli objektumot, továbbítja a hívásokat, visszaadja a hívási eredményt. A távoli objektumok a szerver oldalon automatikusan, vagy kliens oldali aktiválással jöhetnek létre. Az automatikus objektumok esetében megkülönböztetünk egy kérést kiszolgáló objektumot (*Single call*) vagy több kérést kiszolgáló (*Singleton*) objektumot. Meg kell természetesen jegyezni, hogy a szolgáltatást, a programot magát el kell indítani, hiszen a klienshívás hatására csak az alkalmazás egy kiszolgáló típusa jön automatikusan létre! Ezt vagy úgy érzük el, hogy az alkalmazásunkat futtatjuk egy konzolsori parancs kiadásával, vagy mint regisztrált szervízt az operációs rendszer futtatja!

Mielőtt egy konkrét példát néznénk, beszélni kell a paraméter átadás lehetőségéről. Egy alkalmazáson belül a keretrendszer biztosítja az adatok paraméterkénti átadását, átvételét. Esetünkben nem egy alkalmazásról van szó, hanem egy kliensről és egy (vagy több) szerverről, melyek különböző környezetben (*app. domainben*) futnak. Emiatt az adatok átadása sem lehet ugyanaz, mint egy alkalmazáson belül. A különböző alkalmazási környezetben futó programok közötti adatcsere folyamatát *Marshaling* kifejezéssel illet az irodalom, utalva arra, hogy ez a fajta alkalmazási határon átvezető adatforgalom mást jelent, mint egy alkalmazási környezet esetében.

Egy adatot három kategóriába sorolhatunk a .NET keretrendszerben, a távoli adatátadás (*Marshaling*) szempontjából:

1. Érték szerint átadott adatok. Ezen típusok a szabványos szerializációt (mentés) támogató objektumok. (*Marshal-by-value*). Ekkor a típus a *[Serializable]* attribútummal jelölt. A környezet alaptípusai (*int*, *stb.*) menthetőek, így érték szerint átadható adatok.
2. Referencia szerint átadott adatok. Ezek a típusok kötelezően a *MarshalByRefObject* típusból származnak.

3. Alkalmazásdomainek között nem átadható típusok. Ebbe a kategóriába esik minden olyan típus, amelyik nem *MarshalByRefObject* utód, vagy rendelkezik a *[Serializable]* attribútummal.

A legfontosabb tulajdonságok ismertetése után nézzünk egy egyszerű példát. Manuálisan fogjuk a szerverkiszolgálókat is futtatni az egyszerűség kedvéért.

A példánkban két szerverszolgáltatást készítünk, az egyik *http*, míg a másik *tcp* hálózati kommunikációt folytat. A forráskódot mind parancssorból, mind a környezetből tudjuk fordítani. Ez utóbbi esetben a *Project* referencia ablakban a projekthez kell adni a *System.Runtime.Remoting* névteret.

Az alábbi példa egy *bajnokszerver* típust definiál, regisztrálja magát, és más feladata nincs. Enterre a *Main* program azért várakozik, mert ha engedjük befejeződni, mivel nem rendszerrész-szolgáltatás, nem lehet elérni.

Példa:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
public class BajnokSzerver : MarshalByRefObject {

    public string foci;

    public static int Main(string [] args) {

        TcpChannel chan1 = new TcpChannel(8085);
        // 8085 tcp port lefoglalva
        ChannelServices.RegisterChannel(chan1);
        // regisztráció rendben
        RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(BajnokSzerver),
            "bajnok",      // kliens oldalon elérhető
                        // szolgáltatás neve
            WellKnownObjectMode.Singleton);

        System.Console.WriteLine("Program vége, nyomjon entert");
        System.Console.ReadLine();
        return 0;
    }

    public BajnokSzerver() {
        foci="Magyar bajnokság";
        Console.WriteLine("Remote szerver aktiválva!");
    }
}
```



```
public string Ki_a_bajnok(int ev)
{
    string nev="Nem tudom!";
    switch (ev)
    {
        case 2002: nev="Dunaferri";
            break;
        case 2003: nev="MTK";
            break;
        case 2004: nev="Ferencváros";
            break;
    }
    Console.WriteLine("A kért bajnocsapat: {0} ",nev);
    return nev;
}
}
```

A másik szerver lényegében abban különbözik, hogy *http* csatornát használ:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
public class golkiraly : MarshalByRefObject {

    public string sportag ;

    public static int Main(string [] args) {

        HttpChannel chan1 = new HttpChannel(8086);
        // http csatorna foglalása
        ChannelServices.RegisterChannel(chan1);
        // regisztráció
        RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(golkiraly), // típus nevének regisztrálása
            "golkiraly",       // kliens oldali http szolgáltatásnév
            WellKnownObjectMode.Singleton); // szervermód

        System.Console.WriteLine("Program vége!");
        System.Console.ReadLine();
        return 0;
    }

    public golkiraly() {
        sportag = "foci";
        Console.WriteLine("Gólkirály szerver aktiválva");
    }
}
```

### *XIII. Könyvtárak, távoli és webkönyvtárak*

---

```
public string ki_a_golkiraly(int ev) {
    Console.WriteLine("Gólkirály szolgáltatás az alábbi
                                sportágban: {0}",
                                sportag);
    return "Sajnos még nem tudom!";
}
}
```

A kliens programunk, amelyik mindkét kiszolgálóprogramot használja a következő formájú:

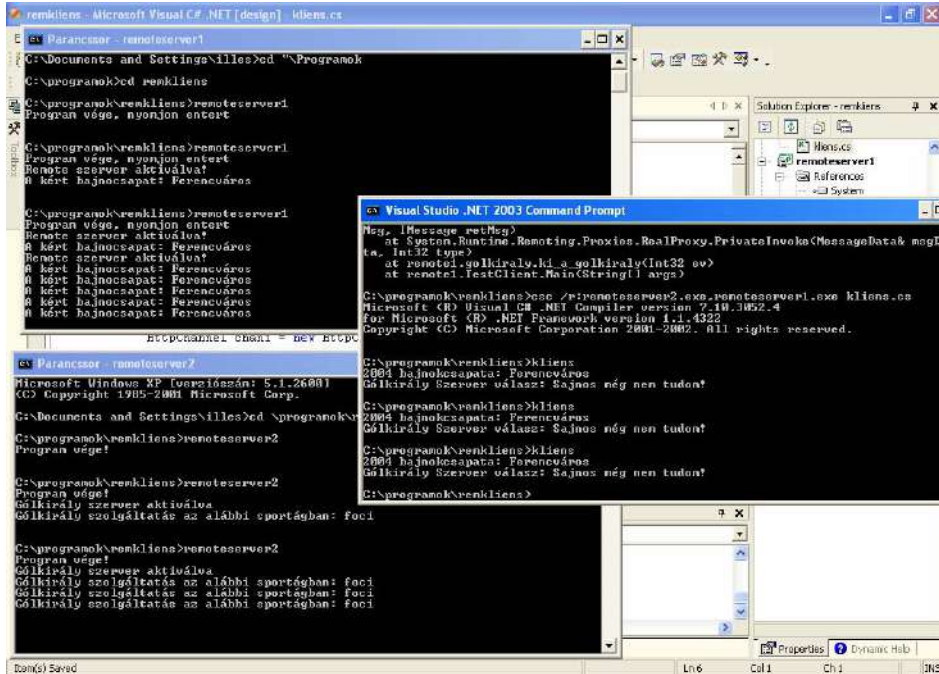
```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels.Http;
using System.IO;
public class kliens
{
    public static int Main(string [] args) {
        HttpChannel chan1 = new HttpChannel();
        // kliens csatorna portot nem adunk meg
        ChannelServices.RegisterChannel(chan1);
        golkiraly g =(golkiraly)Activator.GetObject(
            typeof(golkiraly),
            "http://localhost:8086/golkiraly");
        TcpChannel chan2 = new TcpChannel();
        ChannelServices.RegisterChannel(chan2);
        BajnokSzerver b = (BajnokSzerver)Activator.GetObject(
            typeof(BajnokSzerver),
            "tcp://localhost:8085/bajnok");
        try {
            string bajnok = b.Ki_a_bajnok(2004);
            Console.WriteLine("2004 bajnokcsapata: {0}",bajnok );
            string golkiraly= g.ki_a_golkiraly(2004);
            Console.WriteLine(
                "Gólkirály Szerver válasz: {0}",golkiraly );
        }

        catch (Exception ioExcep) {
            Console.WriteLine("Remote IO Error" +
                "\nException:\n" + ioExcep.ToString());
            return 1;
        }
        return 0;
    }
}
```

A kliens fordítása parancssorból kényelmesebb:

```
csc /r:remoteserver1.exe,remoteserver2.exe kliens.cs
```

Indítsuk el egy-egy ablakban először a szervert, majd a kliensprogramot, ahogy a következő képen is látszik.



17. ábra

Természetesen a szervertablakban látható eredmény a szemléletességet szolgálja, a valós alkalmazások (mivel nem is futnak önálló ablakban) nem írogatnak semmit a képernyőre.

A korábbi feladathoz hasonló kliens-szerver alkalmazáskészítési lehetőséget is biztosít a keretrendszer, úgynevezett *WebRequest*, *WebResponse* modellt használva, vagy a klasszikus *TCP*, *UDP* protokollok használatával (*TcpListener*, *TcpClient*, *UdpClient*). Az osztályok a *System.Net* névtérben találhatók. A *System.Net* összes szolgáltatása a *System.Net.Sockets* névtér szolgáltatásaira épül. A *System.Net.Sockets* névtér a WinSock32 API megvalósítása. Ezen hálózati alkalmazások készítésének lehetősége túlmutat e könyv keretein, így nem is részletezzük azokat.

### XIII.3. Webkönyvtárak használata

A webkönyvtárak használata (*Web Services, webszolgáltatások*) valójában a távoli könyvtárhívás *http* alapú alkalmazásának webkiszolgálón keresztüli megvalósításához hasonlít, a webszerver tölti be a kiszolgáló rendszerbe integrálását és biztosítja a távoli elérhetőséget.

Ekkor azon a kiszolgálón, ahol webkönyvtárat akarunk elhelyezni, ott MS IIS webszervernek kell futni.

Ebben az esetben is legalább két alkalmazás készítéséről beszélhetünk, a szerveroldali alkalmazás készítéséhez egy külön sablont találunk (*ASP.NET Web Service* névvel), míg a kliensalkalmazás tetszőleges C# alkalmazás lehet, például a klasszikus konzol.

Szerveralkalmazás készítéséhez kezdjük új *ASP.NET Web Service* alkalmazást. Ebben az alkalmazásban egyszerűen a *[Webmethod]* attribútummal ellátott függvények érhetők el a külső kliensek számára

Nézzük ezek alapján a bajnokra vonatkozó példánk webkönyvtárral megvalósított forrását:

Példa:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace WebService1
{
    /// <summary>
    /// Summary description for Service1.
    /// </summary>
    public class Service1 : System.Web.Services.WebService
    {
        public Service1()
        {
            InitializeComponent();
        }

        //Component Designer generated code

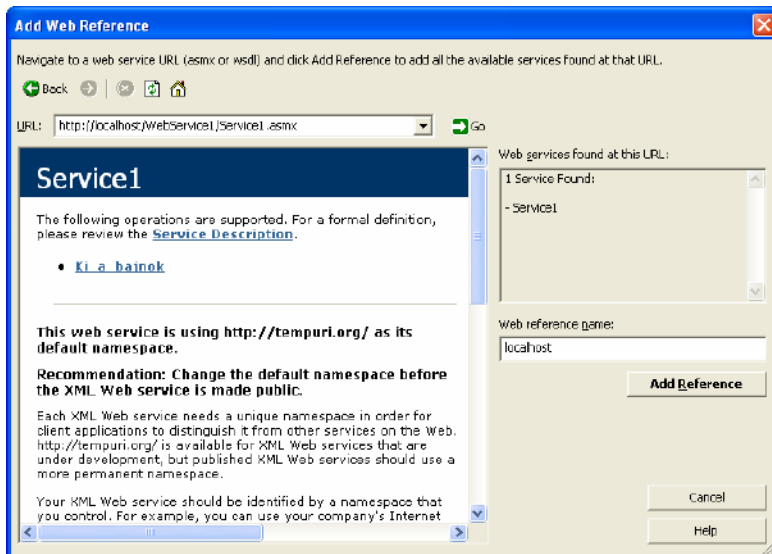
        [WebMethod]
        public string Ki_a_bajnok(int ev)
        {
```

```
string nev="Nem tudom!";
switch (ev)
{
    case 2002: nev="Dunaferr";
        break;
    case 2003: nev="MTK";
        break;
    case 2004: nev="Ferencváros";
        break;
}
return nev;
}
}
```

A forráskód *service1.asmx.cs* néven található. Fordítás után az IIS kiszolgáló *webservice1* virtuális könyvtárába kerülő *service1.asmx* állományra hivatkozva tudjuk futtatni a feladatot. A fenti kiszolgáló használatához webreferenciát kell a készítendő projekthez adni, ahol meg kell adni a Web Service címét a következő módon:

<http://localhost/webservice1/service1.asmx>

Ezt legegyszerűbben az *Add Web Reference* menüpontban tehetjük meg:



18. ábra

A használatát az alábbi forráskód mutatja:

```
...
using System;
namespace webhasznal

{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //
            // a webreferencia neve localhost
            //
            localhost.Service1 s=new localhost.Service1();

            Console.WriteLine(s.Ki_a_bajnok(2004));
        }
    }
}
```

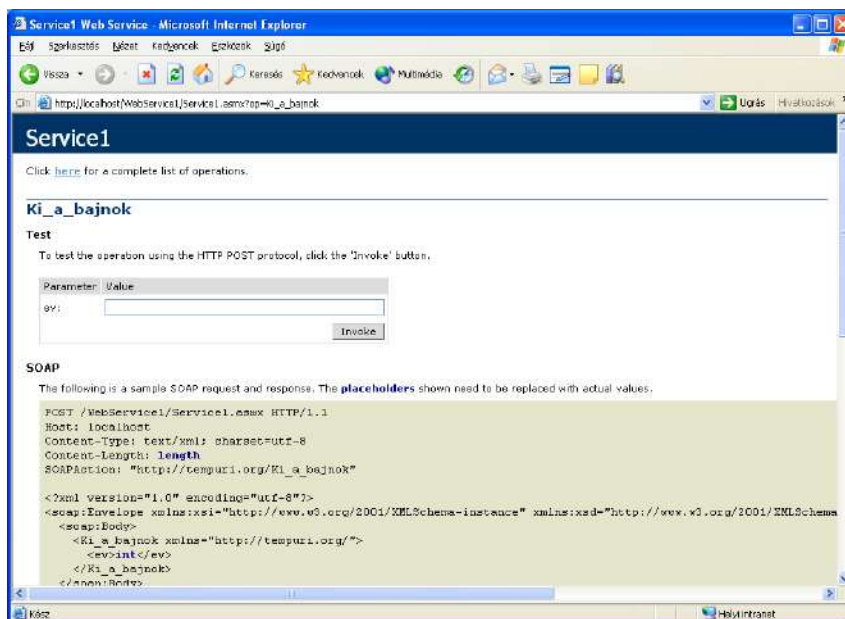
A futási eredmény megadja a várt csapatnevet. Természetesen a *using* névtér használatával az *s* objektumdefiníciót egyszerűbben írhatjuk.

```
using webhasznal.localhost;
// projekt neve után jön a webreferencia neve
...
Service1 s=new Service1();
...
// A használatban már nincs változás
```

A webkiszolgáló szolgáltatását, nemcsak egy kliens programból, hanem a legáltalánosabban használt webes kliens programunkból, például az Internet Explorerből is kipróbálhatjuk (19. ábra).

A szolgáltatásokat általánosan leíró *nyelv* (*Web Service Description Language, WSDL*) természetesen XML formában adja meg, ezt az alábbi kéréssel tudjuk megnézni:

<http://localhost/webservice1/service1.asmx?WSDL>



19. ábra

A webszolgáltatásnak ezt a használatát gyakran szinkronhívásnak nevezzük. Ugyanis ebben az esetben a *Ki\_a\_bajnok* hívása az eredmény visszaérkezéséig nem tér vissza. Ez webes kiszolgálás esetén gyakran hosszabb időt is igénybe vehet. Sőt az sem kizárt, hogy adott időn belül vissza sem tér.

Ha figyelmesen megnézzük a projekt könyvtárunkat, akkor ebben megjelent egy *Web References* könyvtár is. Ebben aztán annyi könyvtárt találunk, ahány webreferenciát csatoltunk a projektünkhöz. Esetünkben találunk egy localhost könyvtárat (*localhost* a neve a <http://localhost/webService1> URL által megadott webkönyvtárnak), ebben egy *References.cs* állományt. Ha ezt megnézzük, látjuk, hogy nem csak a megírt függvényünk van leírva benne, hanem egy *BeginKi\_a\_bajnok* és egy *EndKi\_a\_bajnok* hívás is.

Ezek a függvények adnak lehetőséget arra, hogy egy ilyen webkiszolgálón elhelyezett szolgáltatást ne csak a saját nevével, úgynevezett szinkronhívással tudjunk elérni, hanem aszinkron módon is.

A *Begin*-nel kezdődő függvény mindig azonnal visszatér, eredményül egy *IAsyncResult* objektumot kapunk. Ezen az objektumon keresztül kérdezhetjük meg, hogy befejeződött-e a végrehajtás. Esetünkben a második paraméter és a harmadik is a *null*, jelezve azt, hogy az aszinkron hívás eredményét az *IAsyncResult* objektumon keresztül akarjuk lekérdezni. Egyébként a második paraméter egy delegált (*callback*) függvény, ami által adott függvény kerül végrehajtásra akkor, amikor megérkezik az eredmény. A harmadik paraméter

egy kérés állapotot jelző objektum, amiben a *callback* függvény meg tudja nézni az eredményparamétereket.

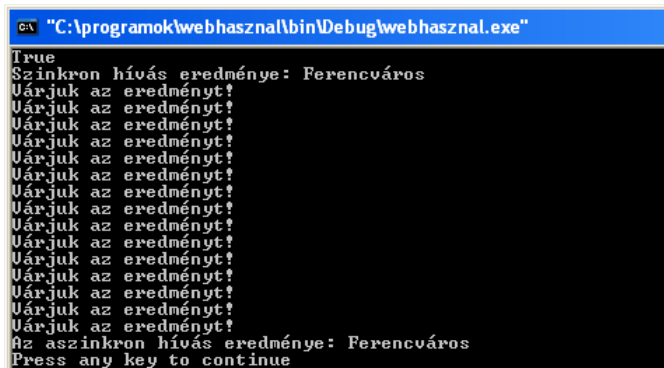
Az alábbi példa nem használja ezeket a paramétereket, hanem az *IAsyncResult IsCompleted* tulajdonságát figyelve várakozik az eredményre.

Példa:

```
static void Main(string[] args)
{
    //
    // TODO: Add code to start application here
    //
    localhost.Service1 s=new localhost.Service1();
    Console.WriteLine("Szinkronhívás eredménye:
                                {0}",s.Ki_a_bajnok(2004));
    IAsyncResult e=s.BeginKi_a_bajnok(2004,null,null);
    // elindítottuk a függvényhívást, majd addig
    //várakozunk, amíg az eredmény meg nem érkezik
    while(e.IsCompleted!=true)
        Console.WriteLine("Várjuk az eredményt!");
    // megjött az eredmény
    // kiolvassuk azt
    string eredmény=s.EndKi_a_bajnok(e);
    Console.WriteLine("Az aszinkronhívás eredménye:
                                {0}",eredmény);
}
```

Eseményvezérelt környezetben, grafikus alkalmazás készítésekor ez a módszer kézenfekvő lehet.

A program futásának eredménye jól illusztrálja az aszinkron végrehajtás jellegzetességét, amit az alábbi futási kép is jól szemléltet:



## 20. ábra



### **XIII.4. Feladatok**

1. Milyen rendszer könyvtári típusokat ismer?
2. Mi a különbség a rendszer könyvtári és a web könyvtár szolgáltatás között?
3. Mit nevezünk szinkron, illetve aszinkron könyvtári hívásnak?
4. Írjon programot, amely egy listában tárolja a kifizetett telefonszámlák összegét! Valósítsa meg a beolvasást, kiírást függvényként!
5. Készítsen Webservice-t, amely egy adott névről eldönti, hogy olimpiai bajnok neve-e!

# ***XIV. Az előfeldolgozó***

## **XIV.1. Szimbólumdefiníció használata**

Az előfeldolgozónak vagy előfordítónak szóló utasítások mindig a # karakterrel kezdődnek.

Az előfordítónak szóló makró definiálási lehetőségének a formája:

`#define név`

Példa:

```
#define menu_h           // menu_h szimbólum definiálva
```

A definíciók hatása az adott modul végéig tart.

Egy azonosító definiálásának gyakori formája a következő:

Példa:

```
#define ALMA
```

Ekkor nem definiálunk helyettesítési értéket, így ez csak annyit mond meg az előfordítónak, hogy ettől kezdve az ALMA szó legyen ismert. Ez gyakran használt mód a feltételes fordítás azonosítóinak definiálására.

Ha már nincs szükségünk egy korábban definiált azonosítóra, és meg szeretnénk kérni az előfordítót, hogy felejtse azt el, akkor a következő előfordítónak ezt az utasítást adhatjuk:

`#undef név`

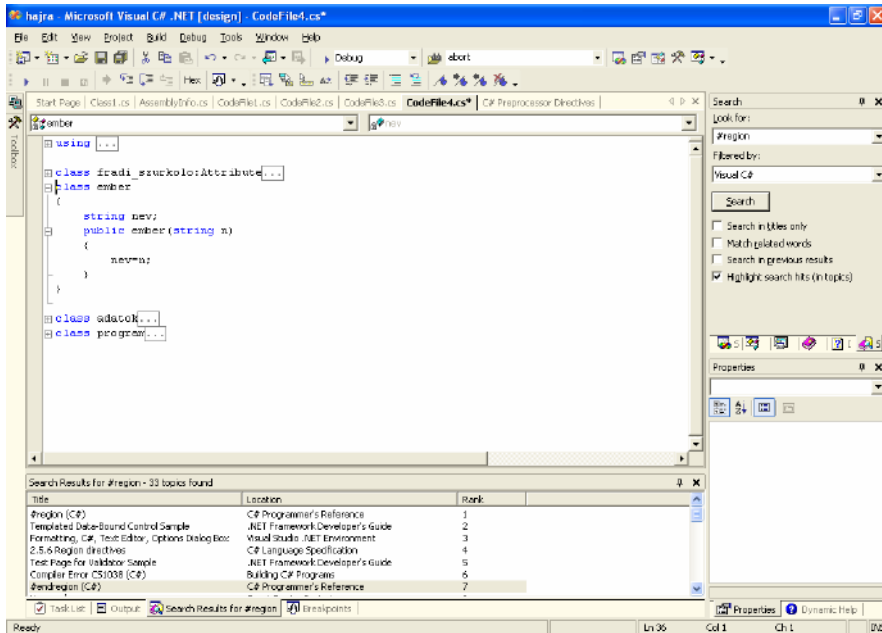
Például az imént definiált *ALMA* azonosító esetében:

Példa:

```
#undef ALMA
```

## XIV.2. Területi jelölés

A fejlesztőrendszer szövegszerkesztője nyújtja alapértelmezésben azt a kényelmi szolgáltatást, hogy egy függvény vagy osztálydefiníció törzsét a szövegszerkesztő bal oldalán található + vagy – gombokkal elrejthetjük vagy megtekinthetjük.



21. ábra

A fenti képen az *ember* osztály tartalmát látjuk, míg a többi egységet (adatok, program...) nem, azokat csak jelöli a szövegszerkesztő, hogy léteznek, de pillanatnyilag nem érdekesek. Ez a segítség nagyobb állományok szerkesztésénél hasznos, hiszen a szerkesztőablakban jobban a lényegre tudunk figyelni.

Ezt a szolgáltatást egészíti ki a

```
#region alma
    osztályok, függvények helye
#endregion alma
```

régió, területdefiníció. Ekkor az *alma* „blokkban”, régióban definiált osztályok, függvények egyszerre húzhatók össze vagy nyithatók ki.

### XIV.3. Feltételes fordítás

<code>#if konstans kif</code>	Igaz-e konstans kif.
<code>#ifdef azonosító</code>	Van-e ilyen makró
<code>#ifndef azonosító</code>	Nincs-e ilyen makró

Mindhárom alakot követheti a `#else` direktíva, majd kötelezően zárja:

```
#endif  
#line sorszám
```

A fordító úgy viselkedik, mintha a következő sor a sorszámmal megadott sor lenne.

`#line default`                      eredeti sorszám visszaállítása

### XIV.4. Hibaüzenet

Ha már az előfordító felfedez valamilyen hibát, akkor hibaüzenetet ad, és befejezi az előfordítást.

A vezérlődirektíva formája:

`#error hibaszöveg`

Példa:

```
#ifndef c#  
#error Sajnos nem a megfelelő fordítót használja!  
#endif
```

### XIV.5. Feladatok

1. Mik az előfordító jellemző szolgáltatásai?
2. Hogy definiálhatunk és szüntethetünk meg egy azonosítót?
3. Mit jelent a régió definíció?
4. Az eddig elkészített programjait módosítsa régió definíciókkal! Figyelje meg, hogy mennyivel áttekinthetőbbé vált a programjának a kódja!
5. Definíálja úgy a másodfokú egyenletet megoldó függvényt, hogy csak akkor fordítsuk le, ha „szükséges”!

## ***XV. Nem felügyelt kód használata***

A C# nyelvű program fordítása egy felügyelt eredményprogramot ad, amin a felügyeletet a .NET keretrendszer biztosítja. Szükség lehet azonban arra, hogy a rendelkezésre álló nem felügyelt, rendes Win32 API könyvtárak szolgáltatásait elérjük, és az ezekkel kapcsolatos adatainkat tudjuk használni, a könyvtárhoz hasonló nem felügyelt kódrészletet tudjunk írni.

### **XV.1. Nem felügyelt könyvtár elérése**

Talán leggyakrabban ez az igény merül fel, hiszen ha megvan egy jól megírt szolgáltatás a korábbi Windows API könyvtárban, akkor azok használata mindenképpen kifizetődőbb, mint helyettük megírni azok menedzselt változatát.

Erre ad lehetőséget a *DllImport* attribútum használata. Egy könyvtári függvény használatához három lépést kell megtenni:

1. A *DllImport* attribútumnak meg kell mondani a *Dll* állomány nevét, amit használni akarunk.
2. A könyvtárban lévő függvényt a fordító számára deklarálni kell, ebben az extern kulcsszó segít. Ezeket a függvényeket egyúttal statikusnak is kell jelölni.
3. A *System.Runtime.InteropServices* névteret kell használni.

Ezek után nézzük meg példaként a *MessageBox* API függvény használatát.

Példa:

```
using System;
using System.Runtime.InteropServices;
class dllhasznál
{
    [DllImport("user32.dll")]
    static extern int MessageBox(int hwnd, string msg,
                                string caption, int type);

    public static void Main()
    {
        MessageBox(0, "Hajrá Fradi !", "Ez az ablakfelirat!", 0);
    }
}
```

## XV. Nem felügyelt kód használata

A program futtatása után az alábbi rendszer-üzenetablak jelenik meg:



22. ábra

### XV.2. Mutatók használata

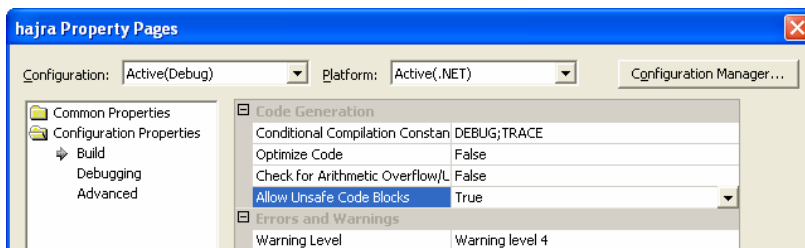
Ahogy korábban is volt szó róla, a keretrendszer a C++ jellegű mutatók használatát nem támogatja. Előfordulhat viszont az, hogy külső erőforrások eléréséhez, illetve azok adatai miatt szükség lehet nem menedzselte, nem biztonságos környezet engedélyezésére. Ebben a környezetben aztán a C++ nyelvben használt mutatófogalom használható.

A nyelv egy függvényt vagy egy utasításblokkot tud nem biztonságossá, nem felügyelt kódrészletté nyilvánítani az *unsafe* kulcsszó használatával.

Emellett egy menedzselte adatot *fixed* jelzővel tudunk ellátni, ha azt akarjuk, hogy a GC által felügyelt területből egy típushoz (menedzselte típus) nem biztonságos mutató hozzáférést kapjunk. Ez természetesen óvatos használatot kíván, hiszen könnyen előfordulhat, hogy az objektumunk a Garbage Collection eredményeként már régen nincs, mikor mi még mindig a mutatójával bűvészkednénk!

A mutatókat csak *unsafe* blokkban használhatjuk.

Ahhoz, hogy a fordító engedélyezze az *unsafe* blokkot, a projekttulajdonságok között be kell állítani az „*Allow Unsafe Code Blocks*” opciót igazra, ahogy az a következő *Tulajdonság* ablakban is látszik.



23. ábra

Ezt a beállítást parancssori környezetben a *csc* fordítónak az */unsafe* kapcsolója használatával érhetjük el.

Ezek után nézzünk egy klasszikus C++ nyelvszerű maximumérték meghatározást. Az alábbi példában a *max* függvény egy egész vektor legnagyobb értékét határozza meg.

Példa:

```
using System;
class unmanaged
{
    unsafe int max(int* v, int db)
    {
        int i=0;
        int m=v[i++];
        while(i<db)
        {
            if (m<v[i]) m=v[i];
            i++;
        }
        return m;
    }
    int[] s=new int[10]{1,2,3,4,5,0,21,11,1,15};
    unsafe public static void Main()
    {
        int h=0;
        unmanaged u=new unmanaged();
        fixed (int* m= &u.s[0])
        {
            h=u.max(m,10);
        }
        Console.WriteLine(h);
    }
}
```

### XV.3. Feladatok

1. Mit jelent a nem felügyelt kód (*unsafe*)?
2. Hogyan tudunk Win32 API függvényt meghívni?
3. Mi a *fixed* változó?
4. Hogyan használhatunk mutatókat egy C# programban?
5. Készítsen *unsafe* függvényt, amelyik a paraméter vektort nagyság szerint sorbarendezi!

## ***XVI. Grafikus alkalmazások alapjai***

A mai grafikus felhasználói felületeken az egyik leginkább kedvelt vagy elvárt alkalmazáskészítési lehetőség a grafikus programok készítése. Emellett az is elmondható, hogy egy program futtatását nem biztos, hogy a helyi gépen szeretnénk végezni. Az internet jelenlegi elterjedését figyelembe véve, egyre gyakrabban merül fel az az igény, hogy az alkalmazást bárki elérhesse egy szabványos internetböngésző program (Internet Explorer, Netscape, Opera stb.) segítségével.

Miután megismertük a korábbi fejezetekben a C# nyelvi és legfontosabb keretrendszeri szolgáltatásait, befejezésképpen nézzünk meg egy-egy példát Windows alapú és „Webes” alapú alkalmazások készítésére.

### **XVI.1. Windows alkalmazások alapjai**

Ahogy eddig is láttuk, a legfontosabb alkalmazási típusok készítéséhez a fejlesztőkörnyezet kész sablont bocsát a fejlesztők rendelkezésére. Így van ez ebben az esetben is.

Új alkalmazás (*project*) készítésekor a „*Windows Application*” sablont választva kapjuk a kicsit preparált forráskódot. Ez az állomány *form1.cs* névre hallgat, és valójában a programot adó *Main* függvény törzse ki van töltve:

```
static void Main()  
{  
    Application.Run(new Form1());  
}
```

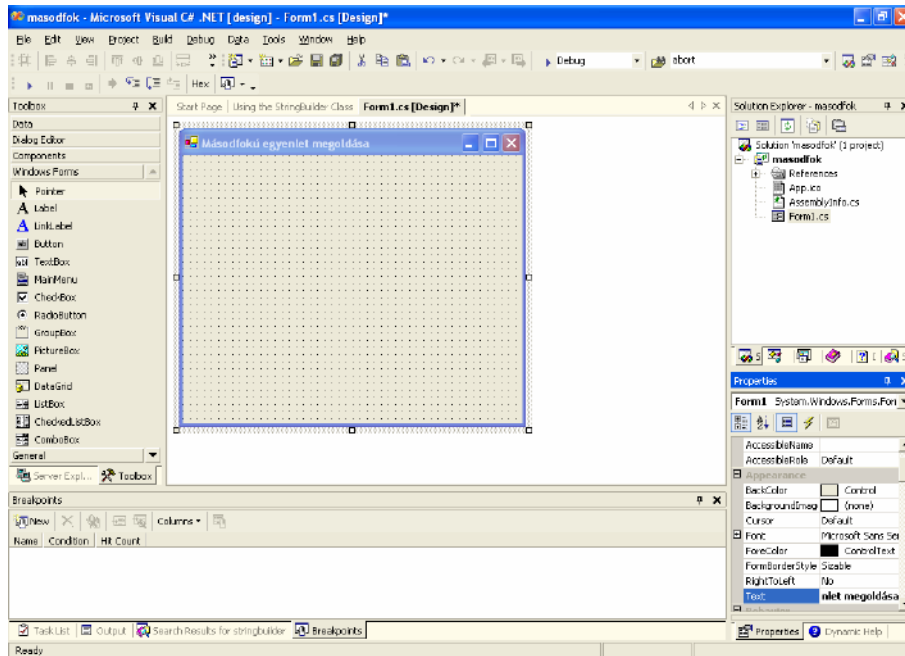
Ez a kódsor azt jelenti, hogy a form utódosztályunk (*Form1*) által képviselt grafikus felület illeszkedjen az operációs rendszer felügyeletébe, és az ablak jelenjen meg. Ez az ablak először természetesen üres, a fő feladat éppen az, hogy megfelelő tartalommal lássuk el, ezáltal elkészítve a kívánt programot.

A program elkészítésében a legnagyobb segítséget a grafikus könyvtári elemek, a *Windows Forms* névtér objektumai (form ablak, címke, nyomógomb stb.) adják. Ezt a lehetőséghez a *Toolbox* ablak mutatja, amit a „rajzszög” segítségével gyakran a képernyőre helyezünk.

Készítsünk a legjellemzőbb tulajdonságok bemutatására egy másodfokú egyenletet megoldó programot. Az új projekt nevének adjuk meg a *masodfok* nevet, válasszuk ki a *Windows Application* sablont, majd a kapott felületre rajzszögezzük ki a *Toolbox* ablakot.



Az így kapott képernyő a következőképpen néz ki:



24. ábra

A forrásállományt megnézve azt láthatjuk, hogy ebben az állapotában a programunk valójában egy form (ablak) objektumból áll (*new Form1()*). Ezt a *Tulajdonságok (Properties)* ablak lenyíló mezőjéből is megállapíthatjuk, hiszen nem tudunk másik objektumot kiválasztani.

A *Tulajdonságok (Properties)* ablakban tervezéskor állíthatjuk be a kiválasztott komponensünk legjellemzőbb tulajdonságait, kezdő adatait. Ezek a tulajdonságok futás közben is hasonló módon megváltoztathatók.

Az egyszerű adatok mellett ez az ablak ad lehetőséget egyes objektumok eseménykezelő paraméterének beállítására. Ez azért lényeges, mert ebben a környezetben a programok valódi tevékenységét ezek a függvények végzik. Így valójában programkészítés címen kicsit egyszerűsítve nincs másról szó, mint hogy olyan grafikus elemekkel építsük fel a programablakot, amelyek eseménykezelői éppen a kívánt feladatot oldják meg.

Ezek után nézzük a célul kitűzött egyszerű feladatot, a másodfokú egyenlet megoldását, amin keresztül a legjellemzőbb lépéseket szemléltetni tudjuk.

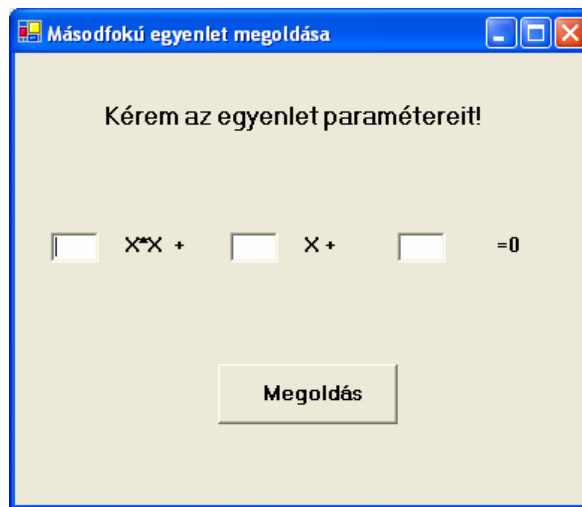
Mielőtt nekifognánk a megoldásnak, le kell szögeznünk, hogy a karakteres felületen használt beolvasási és kírasi lehetőségek nem használhatóak. Erre a

célra a *Toolbox* ablak elemeit tudjuk használni. A leggyakrabban használt elem kiírásra a címke (*Label*), míg beolvasásra a szövegdoboz (*TextBox*).

Ezen elemek segítségével alakítsuk ezután ki a program felhasználói felületét, ahol a címkekkel információt írunk ki, míg a szöveges beolvasó elemek a paraméterek beolvasását biztosítják.

A formra tegyünk címkéket, és a címke *Text* tulajdonság mezőjébe a megjeleníteni kívánt szöveget írjuk bele. A szövegmezők alapértelmezett *Text* mező értékét pedig töröljük ki.

Ezek alapján egy kevés munkával az alábbi felület alakítható ki:



25. ábra

Azt természetesen nem állítom, hogy ez a legszebb kialakítás, de a célnak megfelel.

A fenti grafikus tervezés után láthatjuk a *Tulajdonság* ablak objektum kiválasztómezőjében, hogy minden egyes önálló vezérlő (*label*, *textbox*) egy-egy változó névvel jelenik meg. Ezeket a neveket (*label1*, *label2*, stb.) a keretrendszer automatikusan adja, és ha szükségünk van ezek későbbi használatára, akkor a programtervezési szempontok figyelembevételével adjunk 'beszédesebb' neveket ezen változóknak. Ezt a *Tulajdonság* ablak *Name* mezőjének módosításával tehetjük meg.

A három együtttható beolvasását biztosító textmezőnek rendre az *a*, *b*, *c* neveket adtam, míg a megoldást megjelenítő címkének a *megoldas* nevet. Az ablakot reprezentáló C# nyelvi forráskód az alábbiak szerint módosul.

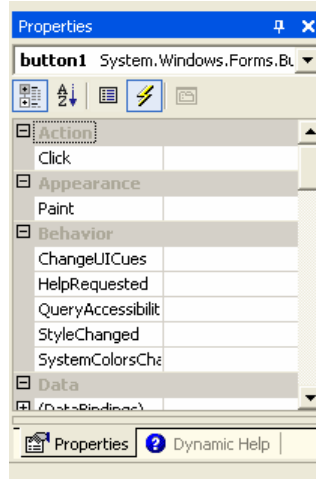
```

namespace masodfok
{
    /// <summary>
    /// Summary description for Form1.
    /// </summary>
    public class Form1 : System.Windows.Forms.Form
    {
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.Label label2;
        private System.Windows.Forms.Label label3;
        private System.Windows.Forms.Label label4;
        private System.Windows.Forms.Button button1;
        private System.Windows.Forms.TextBox a;
        private System.Windows.Forms.TextBox b;
        private System.Windows.Forms.TextBox c;
        private System.Windows.Forms.Label megoldas;
        ...
    }
}

```

Ezeket a bejegyzéseket a keretrendszer automatikusan elvégzi. Azonban meg kell jegyezni, hogy a vezérlőink elnevezését csak ebben a kódrészben végzi el a keretrendszer, így ha utólag nevezünk át vezérlőket, akkor a programkódbeli változásokról magunknak kell gondoskodnunk.

A program tényleges megoldását az egyetlen nyomógomb eseménykezelő függvénye fogja elvégezni. A vezérlőelem eseményeit az alábbi *Tulajdonság* ablakban láthatjuk.



26. ábra

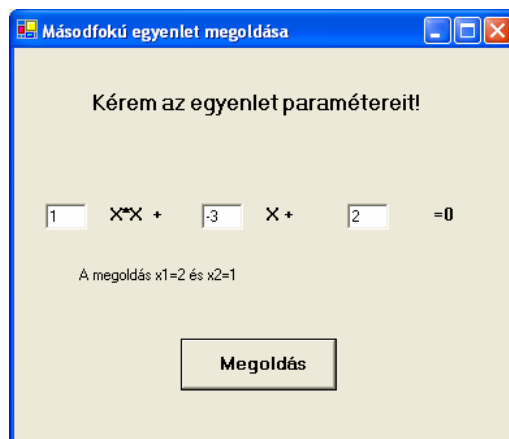
## XVI. Grafikus alkalmazások alapjai

Kettőt kattintva a nyomógombra, a keretrendszer beállítja a nyomógomb *Click* eseménykezelőjét, a forráskódba beírja ennek a függvénynek a keretét, és számunkra nem marad más hátra, mint a valódi programkódot a függvény törzsébe beírni. A feladat ismertsége megengedi, hogy különösebb magyarázat nélkül lássuk az eseménykezelő függvény törzsét:

```
private void button1_Click(object sender, System.EventArgs e)
{
    double a1=Convert.ToDouble(a.Text);
    double b1=Convert.ToDouble(b.Text);
    double c1=Convert.ToDouble(c.Text);
    double m1,m2;
    double d=b1*b1-4*a1*c1;           // diszkrimináns
    if (d<0)
        megoldas.Text="Nincs megoldás, a diszkrimináns negatív.";
    else
    {
        m1=(-b1+Math.Sqrt(d))/2*a1;
        m2=(-b1-Math.Sqrt(d))/2*a1;
        megoldas.Text=String.Format("A megoldás x1={0} és
                                     x2={1}",m1,m2);
    }
}
```

A programot futtatva, miután beírjuk a megfelelő együtthatókat, az alábbi formában kapjuk meg az eredményt.

(Természetesen további finomítás ráérne erre a programra, de ennek elvégzését a kedves Olvasóra bízom.)



27. ábra

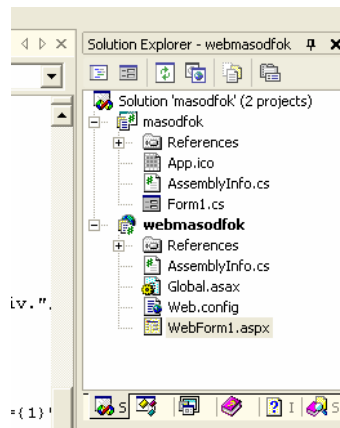
## XVI.2. Webes alkalmazások alapjai

Napjainkban az internetes elérhetőség, rendelkezésre állás már-már szinte alapkövetelmény. Ennek a felhasználói igénynek a kielégítésére a keretrendszer lehetőséget ad Web alapú alkalmazások készítésére.

Az ilyen jellegű alkalmazás készítésének alapfeltétele az, hogy egy web kiszolgáló eléréséhez megfelelő jogosultsággal rendelkezünk. Ez a gyakorlatban az alábbiakat jelenti:

- Azon a gépen ahol fejleszteni szeretnénk, először fel kell installálni az Internet Information Service (IIS) szolgáltatást.
- Fel kell installálni a Visual Studio.NET alkalmazást.
- Ez létrehoz két felhasználói csoportot a számítógépen, *Debugger Users* és *VS Developers* névvel.
- Azokat a fejlesztőket rakjuk bele ezekbe a csoportokba, amelyektől ilyen alkalmazások fejlesztését várjuk. (A *Debugger Users* csoportba minden fejlesztőt bele kell rakni, különben az operációs rendszer a nyomkövetési módban fordított állományt nem enged a keretrendszerből futtatni!)

Ha a fenti feltételek megvannak, akkor *ASP.NET Web Application* sablont választva készíthetünk webes alkalmazást. Ezek alapján készítsunk egy *webmasodfok* projektet a fejlesztőkörnyezetben, ahogyan az a következő képen látszik, az előző *masodfok* projekt mellett elhelyezve. (A keretrendszer *Solution* fogalma több projektet enged egy keretbe foglalni, hiszen gyakran előfordul, hogy egy feladatmegoldást nem egy projekttel célszerű megadni. Példánk esetében nincs erről szó.)

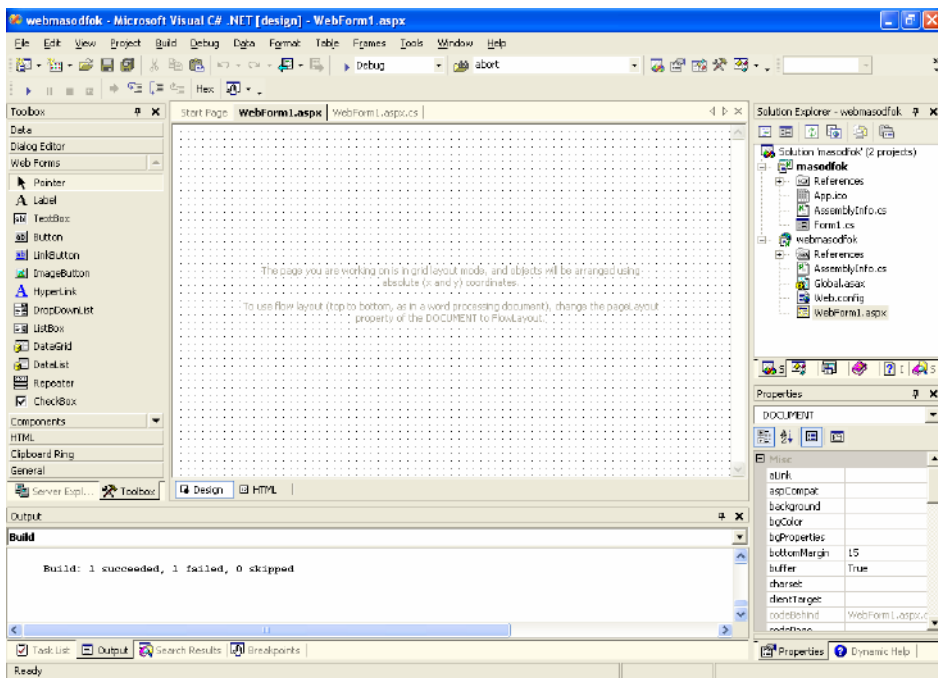


28. ábra

## XVI. Grafikus alkalmazások alapjai

Miután megadtuk a nevet, elkészül a következő üres weboldal. A létrehozott *WebForm1.aspx* az üres HTML oldal (ezért is van HTML nézete) és a háttérben meghúzódo programfájl (*WebForm1.aspx.cs*). Látható, hogy a *Toolbox* ablakban a korábbi *Windows Forms* felirat helyett *Web Forms* olvasható, mutatta, hogy ezek a vezérlők web alapú alkalmazásokhoz használhatóak. A *Tulajdonság* ablak egyetlen objektuma a *DOCUMENT* objektum lesz, ami természetesen magának a HTML dokumentumnak felel meg. Ezek tulajdonságértékeit módosíthatjuk, például a *Title* mező értékét, megadva ezzel a weboldal címekjét, vagy a *bgColor* paraméterrel beállítva a kívánt háttérszínt.

A kapott képernyő a következő alakú lesz:



29. ábra

A *Windows Form* megoldáshoz hasonlóan készítjük el a másodfokú egyetlen megoldását ebben a környezetben is. Ehhez első lépésként alakítsuk ki a programunk felületét az előző példához hasonlóan, majd a nyomógomb eseménykezelő függvényét definiáljuk.

A kapott forráskód (*Webform1.aspx.cs*) a következő lesz:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace webmasodfok
{
    /// <summary>
    /// Summary description for WebForm1.
    /// </summary>
    public class WebForm1 : System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.Label Label1;
        protected System.Web.UI.WebControls.Label Label2;
        protected System.Web.UI.WebControls.Label Label3;
        protected System.Web.UI.WebControls.Label Label4;
        protected System.Web.UI.WebControls.TextBox a;
        protected System.Web.UI.WebControls.TextBox b;
        protected System.Web.UI.WebControls.TextBox c;
        protected System.Web.UI.WebControls.Label megoldas;
        protected System.Web.UI.WebControls.Button Button1;

        private void Page_Load(object sender, System.EventArgs e)
        {
            // Put user code to initialize the page here
        }

        #region Web Form Designer generated code
        override protected void OnInit(EventArgs e)
        {
            //
            // CODEGEN: This call is required by the ASP.NET
            //                                     Web Form Designer.
            //
            InitializeComponent();
            base.OnInit(e);
        }
    }
}
```

```
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>

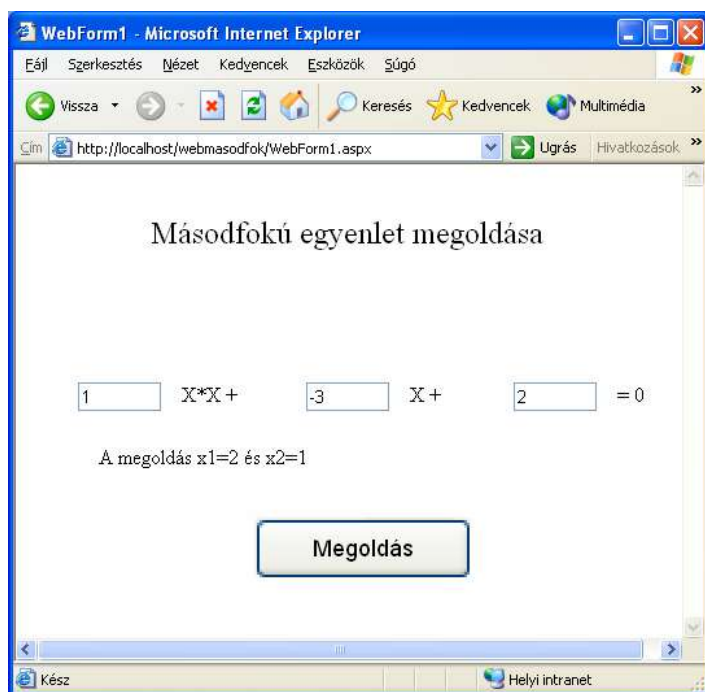
private void InitializeComponent()
{
    this.Button1.Click += new
        System.EventHandler(this.Button1_Click);
    this.Load += new System.EventHandler(this.Page_Load);
}
#endregion

private void Button1_Click(object sender, System.EventArgs e)
{
    double a1=Convert.ToDouble(a.Text);
    double b1=Convert.ToDouble(b.Text);
    double c1=Convert.ToDouble(c.Text);
    double m1,m2;
    double d=b1*b1-4*a1*c1;        // diszkriminás
    if (d<0)
        megoldas.Text="Nincs megoldás, a diszkriminás
                                negatív.";
    else
    {
        m1=(-b1+Math.Sqrt(d))/2*a1;
        m2=(-b1-Math.Sqrt(d))/2*a1;
        megoldas.Text=String.Format("A megoldás x1={0}
                                és x2={1}",m1,m2);
    }
}
}
```

A program fordítása és futtatása után a 30. ábrán látható Internet Explorer böngészőben futó alkalmazást kapjuk, vagyis a célunkat elértük.

Ebben a fejezetben –kitekintésként– csak a legfontosabb környezeti beállításokról, fogalmakról tudtunk szólni. A grafikus alkalmazások lehetőségeit, a segítségünkre lévő grafikus vezérlőelemek tulajdonságait, webes, mobil és/vagy adatbázis kapcsolattal rendelkező alkalmazások készítését egy következő kötet keretében szeretnénk megmutatni.





30. ábra

### XVI.3. Feladatok

1. Mi a feladata a grafikus alkalmazás *Main* függvényének?
2. Mi a különbség a Windows és az ASP.NET alkalmazás között?
3. Milyen feltételeknek kell teljesülni ahhoz, hogy ASP.NET alkalmazást tudjunk készíteni?
4. Készítsen alkalmazást, amely egy téglatest 3 oldalát beolvassa meghatározza a térfogatát és felszínét! (Használjon címkéket és szövegmezőket!)
5. Készítse el az előző feladatot webes alkalmazásként is.

# *Irodalomjegyzék*

1. Brian W. Kernigham, Dennis M. Ritchie : A C programozási nyelv  
Műszaki Könyvkiadó, Budapest 1985, 1988
2. Bjarne Stroustrup: C++ programming language  
AT&T Bell Lab, 1986
3. Tom Archer : Inside C#  
MS Press, 2001
4. Microsoft C# language specifications  
MS Press, 2001
5. John Sharp, Jon Jagger: Microsoft Visual C#.NET  
MS Press, 2002
6. Illés Zoltán: A C++ programozási nyelv  
ELTE IK, Mikrológia jegyzet, 1995-...
7. David S.Platt: Bemutatkozik a Microsoft.NET  
Szak Kiadó, 2001
8. Illés Zoltán: A C# programozási nyelv és környezete  
Informatika a felsőoktatásban 2002, Debrecen
9. David Chappell: Understanding .NET  
Addison-Wesley, 2002

## **A Jedlik Oktatási Stúdió informatikai könyvei:**

**1212 Budapest, Táncsics M. u. 92 • Tel/fax: 276-5335**

**Internet: [www.jos.hu](http://www.jos.hu) • E-mail: [jos@jos.hu](mailto:jos@jos.hu)**

### **Farkas Csaba: Bevezetés a Windows és Office XP használatába, ISBN: 963 00 8822 3**

A Bevezetés a Windows és Office XP használatába c. könyv olvasója nemcsak megismerheti az Office csomag szolgáltatásait, hanem a könyv didaktikus felépítése és számtalan példája, feladata alapján el is sajátíthatja, be is gyakorolhatja annak használatát. Kívánjuk, hogy Olvasónk hasznos tagja legyen az alkalmazók táborának, és sikeresen feleljen meg az európai (ECDL) elvárásoknak

### **Holczer József: Levelezés és csoportmunka Outlookkal, ISBN: 963 20 4374 X**

Az Outlook komplex információ-kezelő szoftver, mely az elektronikus levelezésen túl lehetőséget ad arra, hogy az együtt dolgozó emberek összehangolhassák időbeosztásukat (naptár), kezelhessék egymás és közös partnereik adatait (névjegyalbumok). Megkönnyíti az értekezletek összehívását, a feladatok kiosztását és nyomon követését (feladatkezelő), valamint a különböző események naplózását, így hatékonyabban és főleg egyszerűbben szervezhető a mindennapos munka.

### **Farkas Csaba: Windows XP és Office 2003 felhasználóknak, ISBN: 963 214 548 8**

Könyvünk bevezeti az Olvasót a Windows XP és az Office 2003 (Word, Excel, PowerPoint, Publisher, Outlook, Access, InfoPath, SharePoint, XML támogatás) használatába, de tartalmazza az OKJ és ECDL vizsgákhoz szükséges elméleti ismereteket is.

### **Holczer-Telek: Csoportmunka Office 2003-mal, ISBN: 963 865 140 7**

Az Office 2003-ban főleg az Outlook és a SharePoint támogatja a közös számítógépes munkát. Ezek alapos ismertetésén túl könyvünkben kitérünk a többi Office komponensre, a digitális aláírásra, titkosításra, a BCM-re, és a PDA-k csoportmunkát segítő felhasználására is.

**Fodor Gábor Antal: Esztétikus dokumentumok Worddel, ISBN 963 210 971 6**

A könyv áttekinti a kiadványok készítésének évszázadok alatt kialakult sajátosságait, majd típusonként tárgyalja azokat. Részletesen megismerkedhetünk a hivatalos dokumentumok, a tanulmányok, a marketing jellegű kiadványok készítésének szabályaival, de a szerző kitér a könyv és az újság jellegű kiadványok készítésére is. Eszközként mindvégig a Word szövegszerkesztőt használja.

**Szentirmai Róbert: Bevezetés a Microsoft Office Project 2003 használatába, ISBN: 963 86514 4 X**

A könyv részletesen tárgyalja a projectmenedzsment és nyomon követés elméleti alapjait és megvalósítását a Microsoft Project 2003 segítségével.

**Holczer József: Webszerkesztés egyszerűen, ISBN: 963 86514 9 0**

Könyvünk az önálló webszerkesztésbe vezeti be az Olvasót. Tárgyalja a FrontPage 2003 használatát, a HTML nyelv alapjait és a dinamikus elemek kezelését. Kezdőknek, középfeladóknek és érettségizőknek egyaránt ajánljuk.

**Farkas Csaba – Szabó Marcell: A programozás alapjai Visual Basicben, ISBN 963 214 293 4**

Könyvünkben szeretnénk egyfelől az Olvasót bevezetni programozás világába, másfelől egy olyan hatékony programozási nyelvet bemutatni, mellyel könnyedén tud új programokat készíteni (VB 6), automatizálhatja a Windows folyamatait (VB Script) és új funkciókkal bővítheti az Office programcsomagot (VB makrók). Figyelembe vettük az emelt szintű érettségi vonatkozó követelményeit is.

**Farkas Csaba: Programozási ismeretek haladó felhasználóknak, ISBN: 963 86514 2 3**

Könyvünkben az emelt szintű érettségire készülők megismerkedhetnek az elvárt webszerkesztési, SQL és programozási ismeretekkel, illetve a VB.NET-tel.

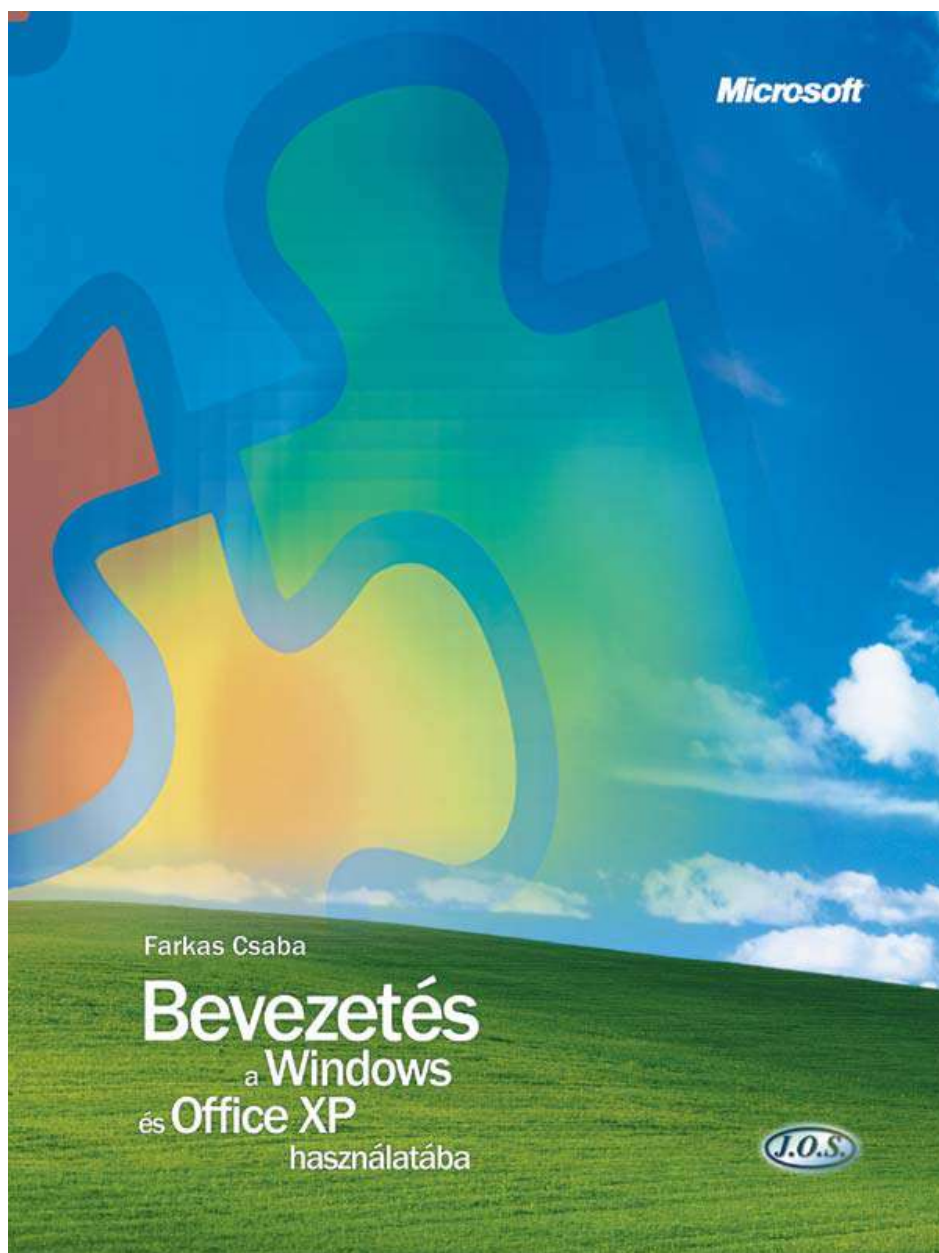
**Bódy Bence: Az SQL példákon keresztül, ISBN: 963 210 860 4**

Az SQL megismerésének leghatékonyabb eszköze kidolgozott mintapéldák tanulmányozása. A könyv ezért 30 feladatcsoportba rendezve több mint 80

fokozatosan nehezedő, valóság-hű feladat kidolgozásával vezeti be az Olvasót az SQL alkalmazásába. Egy-egy feladat megoldására több megoldást is közöl, s az önálló gyakorlás érdekében a fejezetek és a könyv végén közel 100, további feladatot is találhatunk.

**Holczer-Benkovics: Windows Server 2003 hálózatok kezelése, ISBN: 963 214 693 X**

A könyv a Windows Server 2003 alapú hálózatok üzemeltetésébe (hardver ismeretek, TCP/IP protokoll, Windows Server 2003, ISA Server 2000, Exchange Server 2003 üzemeltetése, telepítési ismeretek) tankönyvszerűen vezeti be a kezdő rendszergazdákat.



HOLCZER JÓZSEF

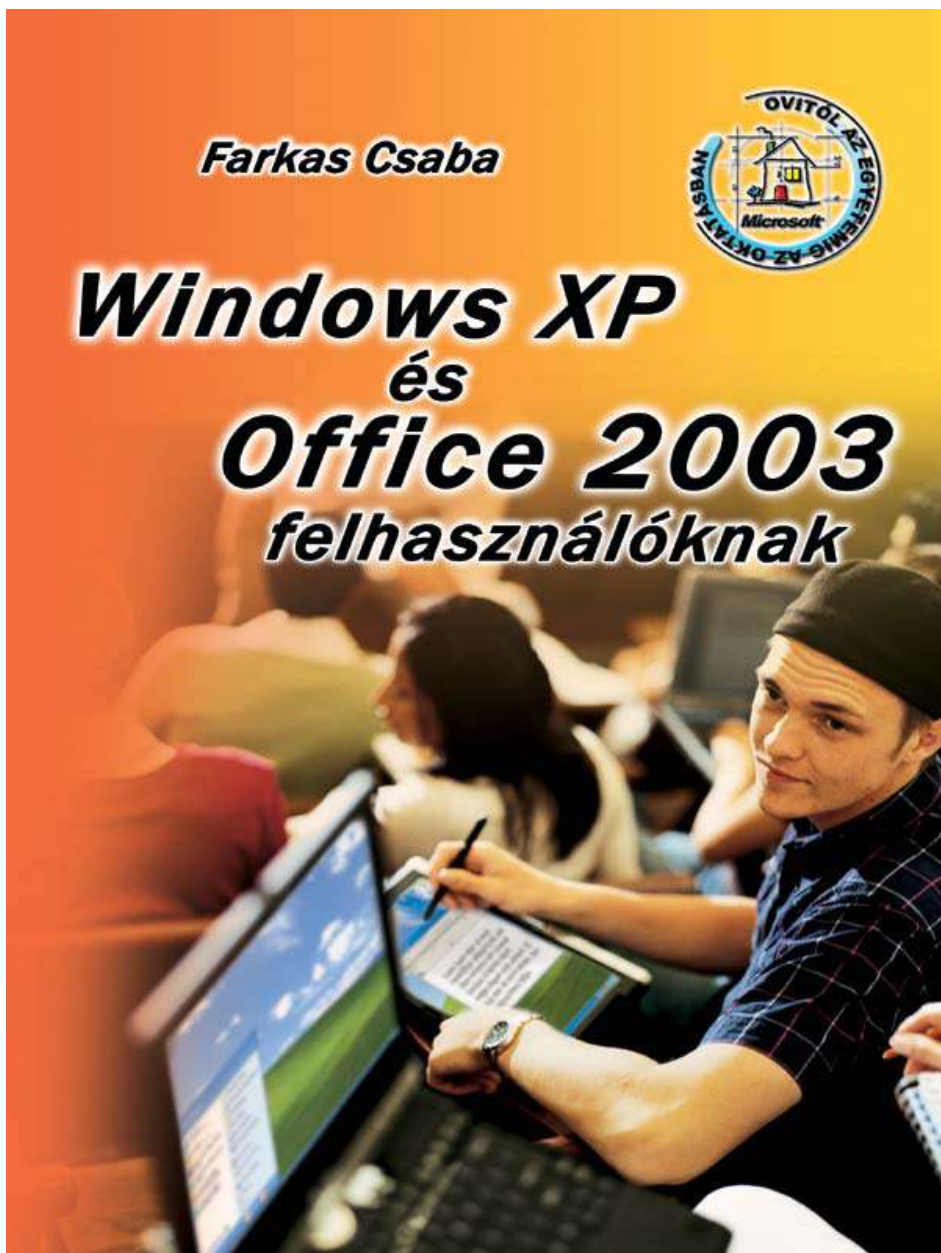
LEVELEZÉS ÉS  
CSOPORTMUNKA  
OUTLOOKKAL



**Farkas Csaba**



# **Windows XP és Office 2003 felhasználóknak**





**Holczer - Telek**



# ***Csoportmunka Office 2003-mal***

***Windows SharePoint Services***

***Outlook***



***Word***

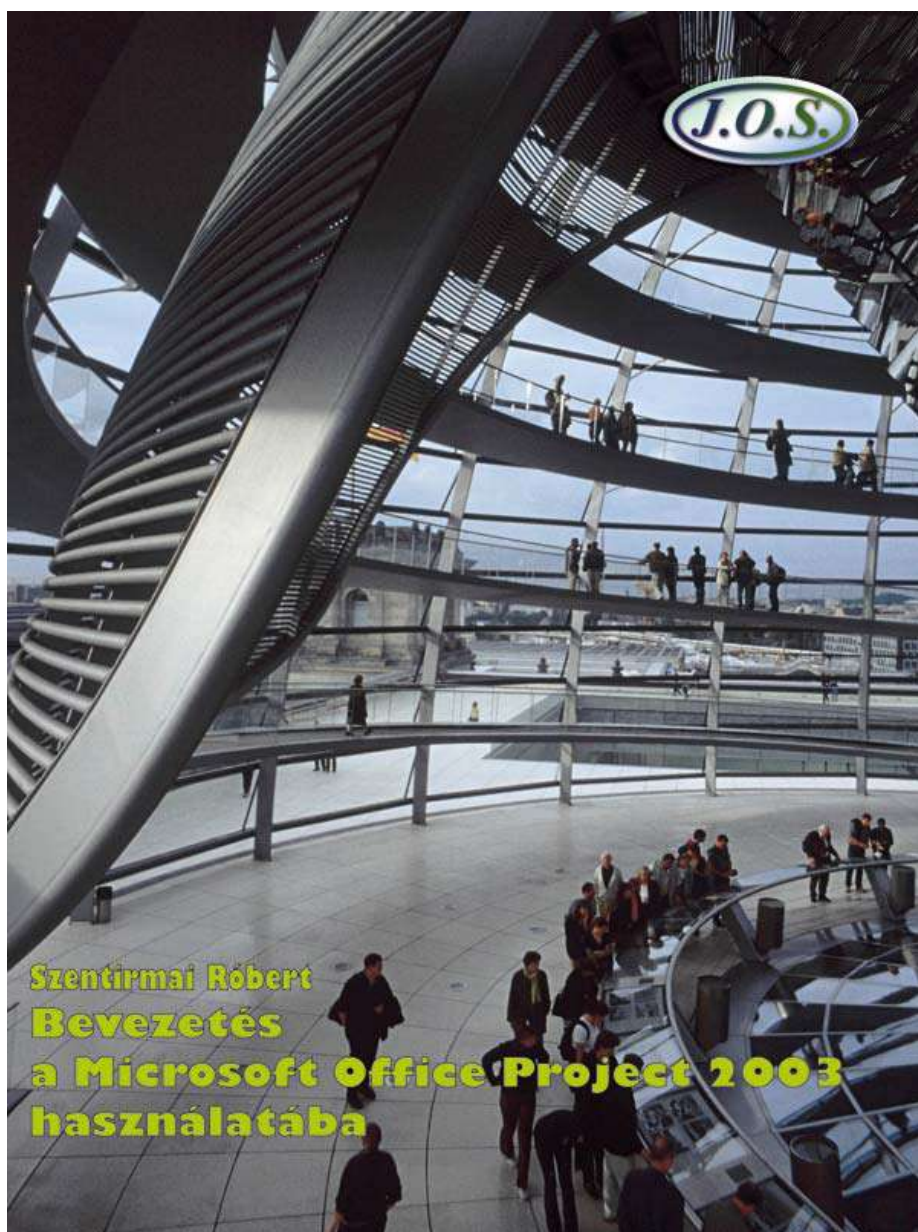
***Excel***

***Pocket PC***

***Business Contact Manager***



[illegible]





# **Webszerkesztés egyszerűen**



**Holczer József**

***Farkas Csaba - Szabó Marcell***



# ***A programozás alapjai Visual Basicben***



***VB6***



**Farkas Csaba**



# **Programozási ismeretek haladó felhasználóknak**



**VB.net**  
**algoritmizálás**



KEZDŐKNEK ÉS HALADÓKNAK

BÓDY BENCE

# AZ SQL

*példákon keresztül*





**Holczer - Benkovics**



# **Windows Server 2003**

**hálózatok**

**kezelése**

