



Belépő a tudás közösségébe

Szakköri segédanyagok tanároknak



C#: egy nyelv
ezernyi lehetőség

C# nyelv fontosabb lehetőségei

Dr. Illés Zoltán

A kiadvány „A felsőoktatásba bekerülést elősegítő
kétségfejlesztő és kommunikációs programok
megvalósítása, valamint az MTMI szakok népszerűsítése
felsőoktatásban” (EFOP-3.4.4-16-2017-006) című pályáz
keretében készült 2018-ban.



Eötvös Loránd Tudományegyetem
Informatikai Kar



MAGYARORSZÁG
KORMÁNYA

SZÉCHENYI 2020

Európai Unió
Európai Szociális
Alap



BEFEKTETÉS A JÖVŐBE

C#: egy nyelv – ezernyi lehetőség

C# nyelv haladó lehetőségei

Felelős kiadó

ELTE Informatikai Kar
1117 Budapest, Pázmány Péter sétány 1/C.

ISBN szám

ISBN XXXXXXXXX

A kiadvány „A felsőoktatásba bekerülést elősegítő készségfejlesztő és kommunikációs programok megvalósítása, valamint az MTMI szakok népszerűsítése a felsőoktatásban” (EFOP-3.4.4-16-2017-006) című pályázat kereténben készült 2018-ban

1. Bevezetés

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

Új nyelvek, új környezetek jelennek meg, ami állandó az a változás! A nyelveket, szoftver fejlesztési környezeteket tekintve az utóbbi időkben a sok változás mellett stabil és az egyik legnépszerűbb, legsokoldalúbb környezetét jelenti a Visual Studio (VS) fejlesztőrendszer a C# programozási nyelvvel.

Ebben a tananyagban folytatjuk a fejlesztési eszköztárnak az alapjait, a C# programozási nyelvet ismertető tananyagot. Ebben a részben a nyelven összetettebb, haladó eszközeiről, azok lehetőségeit ismertetjük.

Megjegyzés

Az előző tananyag készítésénél a megoldásokat Visual Studio 2017 Enterprise (verzió 15.5.4.) segítségével készítettük, 4.7 .Net Framework-öt használva. Jelen dokumentum készítésekor a Visual Studio 2019 a legfrissebb fejlesztési környezet!

Remélem, hogy ez a tankönyv széles olvasótábornak fog hasznos információkat nyújtani. A programozással most ismerkedőknek egy új, hatékony világot mutat meg, míg a programozásban már jártas Olvasók talán ennek a könyvnek a segítségével megérik azt, hogy ez a nyelv az igazi.

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

II. Osztályok

Ahogy láttuk az előző anyagban, a C# nyelvben az osztályok a már korábban ismert összetett adatszerkezetek (struktúrák) egy természetes kiterjesztései. Nézzük meg röviden, mintegy ismétlés, összefoglalásképpen a legfontosabb jellemzőket.

II.1. Osztályok fontosabb tulajdonságai

Az osztályok nemcsak adattagokat, hanem operátorokat, függvényeket is tartalmazhatnak. A függvényezőket gyakran metódusoknak is nevezzük. A függvényező (metódusok) mellett jellemzően adatezőket tartalmaznak az osztályok. Ezek az adatok az osztály egy példányának állapotát adják meg, így ezen osztályező általában az „állapotjelző” adatező.

Új igények, módosítások esetén lehetőségünk van az eredeti osztály megtartása mellett, abból a tulajdonságok megőrzésével egy új osztály származtatására, amely *örökli (inheritance)* az őosztály tulajdonságait (az osztályok elemeit). A C++ nyelvben lehetőségünk van arra, hogy több osztályból származtassunk utódosztályt (*multiple inheritance*), esetünkben ezt a jellemzőt a C# nem támogatja.

Lehetőségünk van *interface* definícióra, amiket egy osztály implementálhat. Egy *interface* metódus előírásokat tartalmazhat, kifejtés, implementáció nélkül. Egy osztály több *interface*-t is implementálhat.

Általában is elmondható, hogy ha a programokban az osztályok közös vonásokkal rendelkeznek, akkor törekedni kell univerzális bázisosztály létrehozására.

Egy általános megoldás ennek biztosítására egy közös ő interface definiálása.

Másik megoldás lehet, hogy az absztrakt függvények, osztályok lehetőségét használjuk. Egy osztályban definiálhatunk absztrakt függvény(ek)e)t, amelyeknek hiányzik a kifejtése. Gyakran ezt *absztrakt bázisosztálynak* is nevezzük. Ebből származtatjuk a megfelelő utódokat, melyekben az absztrakt metódust az igényeknek megfelelően

II. Osztályok

implementálunk, míg az absztrakt osztály általános függvényei a közös tulajdonságokat adják.

Az osztályok definiálásakor kerüljük a „nyitott” (publikus) adatmezők használatát.

Az osztálykulcsszó a *class*, *struct* szó lehet, bár ez utóbbi inkább már csak a korábbi rendszerek kompatibilitási formáját mutatja, kevésbé használt. Ha az osztály valamely bázisosztályból származik, akkor az osztálynév, majd kettőspont után az őosztály megadása történik.

Az osztályok tagjainak öt hozzáférési szintje lehet, *private*, *public*, *protected*, *internal* és *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.)

Az egyes mezőnevekre való hivatkozás a pont (.) segítségével történik.

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

II. Osztályok

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 osztálypéldányra 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.

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";
```

II. Osztályok

```
...  
    };  
  
...  
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 osztálypéldány (objektum) „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
```

II. Osztályok

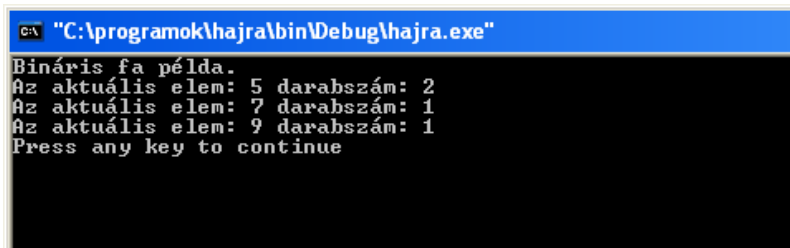
```
{
    int x;                                // elem
    int db;                                // elemszám
    public binfa(int i)                    // konstruktor
    {
        x=i; db=1;
        bal=jobb=null;
    }
    public binfa bal, jobb;
    public void illeszt(int i)
    {
        if (x==i) 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.");
    }
}
```


II. Osztályok

```
        bf.illeszt(5);  
        bf.illeszt(9);  
        bf.illeszt(5);  
        bf.bejar();  
    }  
}
```

Futási eredményül az alábbiit 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
```

1. 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_konstruktor  
{  
    public static int x;
```

II. Osztályok

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

VII.2.2. Statikus osztály

Egy osztály rendelkezhet statikus tagokkal, statikus konstruktorral, ahogy azt korábban is említettük.

Definiálhatunk olyan osztályt is, ami maga is statikus. Ilyen osztállynak csak statikus mezői lehetnek. Egy ilyen mező lehet akár egy statikus konstruktor is! Az ilyen osztályokból nem készíthetünk példányokat, nem származtathatunk belőlük újabb típusokat.

Példa:

```
public static class statikus_osztály  
{  
    static int x;  
    static statikus_osztály()  
    {
```

II. Osztályok

```
        x=5;
    }
    public static int szoroz(int mit)
    {
        return x*mit;
    }
}
class program
{
    public static void Main()
    {
        // tagfüggvény hívás
        Console.WriteLine(statikus_osztaly.szoroz(6));    //30
    }
}
```

A könyvtári szolgáltatások között talán a leggyakrabban használt osztályunk (System.Console) is statikus.

VII.2.3. Függvények kiterjesztése (Extension methods)

Egy statikus osztály csak statikus tagokkal (adat vagy függvény) rendelkezhet. Ez a programunk számára azt jelenti, hogy ezen adatok, függvények ha publikusak, bárki számára elérhetők

Ilyen statikus osztályban megengedett az a függvénydefiníciós forma is, mikor az első paramétert megelőzi a **this** kulcsszó. (Lehetnek további paraméterek is!)

```
public static típus függvéynév(this típus név,
    tov.paraméterek)
    {
        függvénytörzs;
    };
```

II. Osztályok

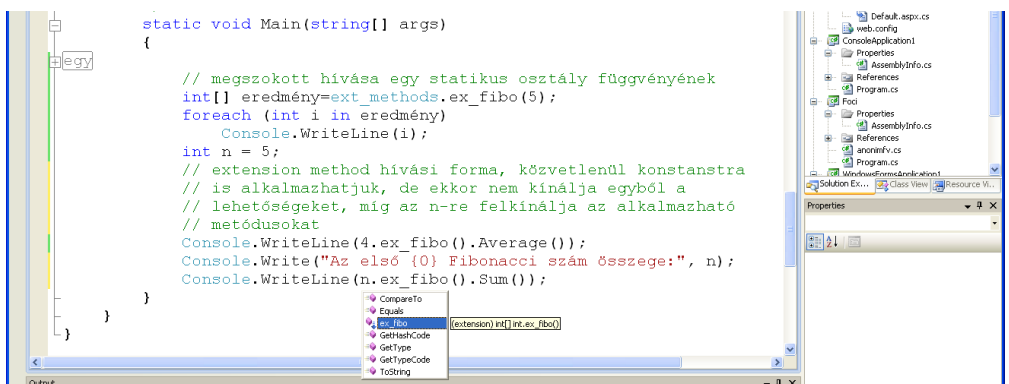
Az így definiált függvényt meghívhatjuk „hagyományos” módon is, de úgy is mint az első paraméternek, mint objektumnak egy függvényeként!

Nézzünk egy konkrét példát. Készítsük el azt a függvényt, amelyik egy n egész számhoz megadja az első n Fibonacci számot.

Példa:

```
static class ext_methods_pelda
{
    public static int[] ex_fibo(this int n) // első n fibonacci
szám
//Az egész típusnak egy kiterjesztése
    {
        int[] eredmény = new int[n];
        eredmény[0] = eredmény[1] = 1;
        for (int i = 2; i < n; i++)
            eredmény[i] = eredmény[i - 1] + eredmény[i - 2];
        return eredmény;
    }
}
```

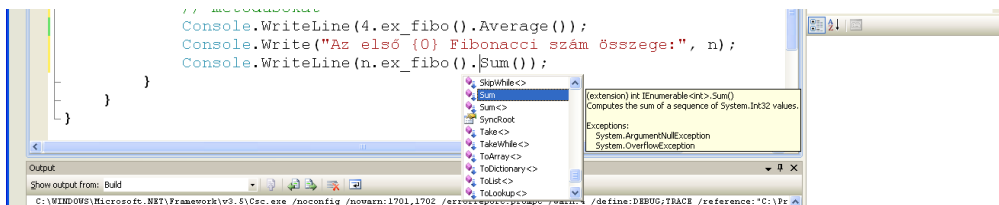
Ekkor az így definiált *ex_fibo* függvényünket egy egész típusú érték (n) kiterjesztés függvényének is nevezzük. Ennek kétféle hívását láthatjuk a következő képen.



11. ábra

II. Osztályok

Az `ex_fibo` függvényünk eredménye egy egész tömb. Ha egy ilyen tömbre akarom használni a függvénykiterjesztés (*extension methods*) hívási formát, azaz egy pontot írok le az `ex_fibo()` hívás után, akkor az előző ábrához hasonló formájú segítséget kapunk. Ahogy a következő ábrán látszik, válogathatunk a .NET keretrendszer tömbre vonatkozó könyvtári függvénykiterjesztései közül.



12. ábra

A függvénykiterjesztés hívási sorrendje (balról jobbra) a fordítottja a rendes egymásbaágyazott függvényhívások sorrendjének. (jobbról balra)

Ahogy az ábrákon is látszik, felsorolható (*IEnumerable*) típusokon számos könyvtári függvénykiterjesztés (*extension methods*) is rendelkezésünkre áll. Használtuk is az összeg, átlag metódusokat. Extension metódusok használata gyakran előfordul a LINQ lekérdezések készítése kapcsán, amit a X. fejezetben tárgyalunk.

VII.2.4. 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;
```

II. Osztályok

```
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();
        // Az alábbi sor hibát adna.
        // nincs_peldanya np=new nincs_peldanya();
    }
}
```

VII.2.5. 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

II. Osztályok

a ~ karakterrel. A destruktorki hívása automatikusan történik, és miután az objektumot nem használjuk, a keretrendszer automatikusan lefuttatja, és a felszabaduló memóriát visszaadja az operációs rendszernek.

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

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

A destruktorki automatikus hívásának 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 destruktorki függvényt, akkor is a garbage collector minden, az objektum által már nem használt memóriát felszabadít az automatikusan definiált destruktorki függvény segítségével. Ez azt jelenti, hogy nincs túlzottan nagy kényszer saját destruktorki definiálására. A garbage collector valójában az osztály *Finalize* függvényét hívja meg.

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

Példa:

```
class osztály
{
    ~osztály()
    { ...
        // destruktorki függvénytorzs
    }
}
```

A fenti osztálydestruktorból a fordító az alábbi kódot generálja:

```
protected override void Finalize()
{
    try
    { ...
        // destruktorki függvénytorzs
    }
}
```

II. Osztályok

```
    }  
    finally  
    {  
        base.Finalize();  
    }  
}
```

A fenti destruktorkonstrukciónak 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);
```

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.

II. Osztályok

Ahogy korábban említettük a destruktorral kapcsolatosan, mi nem tudjuk meghívni, a destruktort a keretrendszer szemétyűjtő 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 őszosztály, 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 szemétyűjtési algoritmusnak ahhoz, mely területeket lehet visszaadni a rendszer részére. A kritikus esetben visszaadható

II. Osztályok

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 destruktorral kapcsolatban zárásképpen 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++ beli *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
```

II. Osztályok

```
{
    x=new object[db];        // helyet foglalunk a vektornak
    mut=0;                   // az első szabad helyre mutat
}

    // NEM DEFINIÁLUNK DESTRUKTORT
    // MERT AZ AUTOMATIKUS SZEMÉTTYŰJTÉSI
    // ALGORITMUS FELSZABADÍTJA 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 osztálydefinícióban vagy a konstruktor függvényben explicit kijelölhető.

II. Osztályok

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* kulcsszóval 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.

II. Osztályok

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 végezhetjük 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.

II. Osztályok

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          // ez hajtódik végre mikor a
                     // tulajdonságot írjuk

        {
            nev=value;
        }
    }
    public Ember(string n, int k)
    {
        nev=n; kor=k;    // nev, kor privát mezők elérése
    }
}
class program
{
    public static void Main()
```

II. Osztályok

```
{
    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.

Gyakran szükség lehet arra, hogy a tulajdonságok egyikéhez ne publikus legyen a hozzáférés. Ez általában a set blokkra vonatkozik, ugyanis amíg egy tulajdonság értékére mindenkinek szüksége lehet, de annak módosítását azért ne mindenki tudja elvégezni. Ezt az aszimmetrikus elérési lehetőséget elérhetjük ha a get, set blokkok elé írjuk a kívánt (protected, privát, internal) jelzőket.

Példa:

```
public string Nev // kis- és nagybetű miatt nev!=Nev
{
    get //
    {
        return nev;
    }
    protected set // az utódból engedem a módosítást
    {
        nev=value;
    }
}
```

Tulajdonságfüggvények megvalósítása nagyon gyakran olyan egyszerű rutinsorokat jelent, mint láttuk az előző példában. Elővesszük az adatmező értékét, illetve új értéket adunk neki.

II. Osztályok

Ilyen esetben, ha csak jelöljük a tulajdonság függvény meglétét, akkor a fordító hozzá gondolja (implicit típus) a szükséges adattípust is, és a tulajdonság függvény implementációt is.

Példa:

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

namespace autoprop
{
    class ember
    {
        // automatikus property-k, tulajdonságfüggvények
        // a fordító kitalálja, hogy kell egy string és egész típusú
        // változó, amihez a „szabványos” get, set utasítástörzset
        // hozzáadja
        public string Név {get; set;}
        public int Kor {get; private set;}
        // kívülről csak olvasható tulajdonság
        public ember(string név, int kor)
        {
            Név = név;          // Név tulajdonság elérése!!!
            Kor = kor;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ember em=new ember("Kiss Mari", 25);
            em.Név="Nagy Péterné "+em.Név;
            //em.Kor=43;          // hiba lenne!!!
            Console.WriteLine(em.Név);
        }
    }
}
```


II. Osztályok

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
    }
}
```

II. Osztályok

```
        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, és egy függvénynek pedig nem csak egész (*index*) paramétere lehet.

Példa:

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

II. Osztályok

```
    }  
}
```

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

VII.5. Osztályok függvényparaméterként

Egy függvény paraméterei 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)  
    {
```

II. Osztályok

```
        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 lehetőségében ezzel egyenértékű.

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

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

II. Osztályok

Kétooperandusú 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 overloadingnak 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 ősosztály operátor függvényei igény szerint újradefiniálhatóak.

Az operátor függvény argumentumával kapcsolatosan elmondhatjuk, hogy egyoperandusú operátor esetén egy paramétere van, míg kétooperandusú 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;
```

II. Osztályok

```
public komplex(float x,float y) // konstruktor
{
    re=x; im=y;
}
float Re
{
    get
    {
        return re;
    }
    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()    // Példány szöveges alak
{
```

II. Osztályok

```
        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);
        // a k+4 komplex indirekt toString hívás
    }
}
```

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;
    // rövidebben így írható
    // return new komplex(a.re+b.re,a.im+b.im);
}
```

II. Osztályok

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.

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)
{
```


II. Osztályok

```
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.

```
...  
komplex k= new komplex(3,2);  
komplex k1= 5 + k1;    // implicit hívás
```

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:

II. Osztályok

```
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 korábban 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)  
{
```

II. Osztályok

```
        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><, ></code>	kisebb, nagyobb
<code><=, >=</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.

II. Osztályok

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
}
```

II. Osztályok

VII.7.2. Interface használata

Az *IAлма* 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))
            return 230;
        else return 0;
    }
}

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

II. Osztályok

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()    //IAlma-ban előírt függvény
    {
        return false;
    }
    public double termésátlag()    //Szintén IAlma
    {
        return 250;
    }
    public string csomagolás_módja()    // szállítható_alma
    {
        return "konténer";
    }
}
```

II. Osztályok

```
public bool szállít()           // szállítható
{
    return false;
}
}
class program
{
    public static void Main()
    {
        vonatán j=new vonatán();
        Console.WriteLine(j.termésátlag()); // 250 kg az átlagtemés
        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:

```
class utódnév: ősnév
{
    // ...
}
```

II. Osztályok

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

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

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

Egy őstípusú referencia bármely utódtípusra hivatkozhat.

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. Természetesen az utódosztálynak lesz egy *i* egészmezője is, csak az közvetlenül nem lesz elérhető.

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.

II. Osztályok

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

```
class utód: ős, interfaced1, ... {  
    //  
};
```

Konstruktorok és destruktorkok használata öröklés esetén is megengedett. Egy típus definiálásakor a konstruktor függvény kerül meghívásra, és ekkor először az ősz osztály konstruktora, majd utána az utódosztály konstruktora kerül meghívásra, míg destruktorkor esetén fordítva, először az utódosztály majd utána az ősz osztály destruktorkát hívja a rendszer. Természetesen a destruktork 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)  
  
{  
    // ...  
}
```

Többszintű öröklés esetén először az őskonstruktorok kerülnek a definíció sorrendjében végrehajtásra, majd az utódbeli tagosztályok konstruktorai és legvégül az utódkonstruktor következik. A destruktorkok 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 destruktork");}  
}  
class b: a
```

II. Osztályok

```
{
    public b()
    { Console.WriteLine("B konstruktor");}
    ~b()
    { Console.WriteLine("B destruktorktor");}
}
class program
{
    public static void Main()
    {
        b x=new b(); // b típusú objektum keletkezik, majd
'kimúlik'
        // Először az a majd utána a b konstruktorát 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, amelyikre 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
```

II. Osztályok

```
{  
  
}
```

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;
```

II. Osztályok

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

II. Osztályok

VII.10. Virtuális tagfüggvények, függvényelfedés

A programkészítés során, ahogy láttuk, az egyik hatékony fejlesztési eszköz az osztályok öröklési lehetőségének kihasználása. Ekkor a függvénytörzsimorfizmus alapján természetesen lehetőségünk van ugyanazon névvel mind az ősztyályban, 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* ősztyályt, 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()
    {
```

II. Osztályok

```
        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
        p=k;                //Egy őstípus egy utódra hivatkozik
        p.kiir();           //Mit ír ki?
    }
}
```

A program futásának eredményeként először a *p* pont adatai (2,1), majd a *kor* adata (5) kerül kiírásra.

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 utód típusra 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 pontban van *kiir*, ezért attól függetlenül, hogy időközben a program során (futás közbeni – dinamikus – módosítás után) a *p* már egy kör objektumot azonosít, azaz a kör *kiir* függvényét kellene meghívni, még mindig a fixen, fordítási időben hozzákapcsolt *pont.kiir* függvényt hajtja végre.

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

Példa:

```
using System;
```

II. Osztályok

```
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 kiir2,1
        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
```

II. Osztályok

```
        // 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 ősoosztályban nincs szükségünk egy függvényre, míg a származtatott osztályokban már igen, és szeretnénk, ha virtuálisként tudnánk definiálni. Ebben az esetben az ősoosztályban egy absztrakt függvénydefiníciót 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);
```


II. Osztályok

```
    }  
}  
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, ezek virtuális függvények. A *new* hatására a *legkedvenc_focicsapat* osztály *nevkiir* függvénye eltakarja az ősz osztály(ok) 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="Fradi";  
    }  
}
```

II. Osztályok

```
    }  
    new public void nevkiir()  
    {  
        Console.WriteLine("Kedvenc csapatunk a: {0}",csapat);  
    }  
}  
class legkedvenc_csapat : kedvenc_csapat  
{  
    public legkedvenc_csapat()  
    {  
        csapat = "Debrecen";  
    }  
    new public void nevkiir()  
    {  
        Console.WriteLine("Legkedvenc csapatunk a: {0}",  
csapat);  
    }  
}  
public class MainClass  
{  
    public static void Main()  
    {  
        legkedvenc_csapat lk = new legkedvenc_csapat();  
        lk.nevkiir();    // legkedvenc nevkiir hívás  
        focicsapat cs = new legkedvenc_csapat();  
        cs.nevkiir();    // kedvenc_csapat hívás  
    }  
}
```

A *new* utasítás hatására egy öröklési sor megszakad, ezért a *cs.nevkiir()* illesztés a sorban csak a *kedvenc_csapat* osztályig jut, így a *kedvenc_csapat* osztályban definiált *nevkiir* függvényt hívja meg.

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

1. .

VIII. Kivételkezelés

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

1. A függvény vagy programrész végrehajtás során semmilyen „rendellenesség” nem lép fel.
2. A függvény vagy programrész aktuális hívása nem teljesíti a paraméterekre vonatkozó előírásainkat, így ekkor a „saját hibavédelmünk” eredményekénti hibával fejeződik be a végrehajtás.
3. 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

VIII. Kivételkezelés

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 végrehajtott

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 végrehajtott
}
```

A *try* blokkot követheti legalább egy *catch* blokk, de több is következhet. Ha több *catch* blokk van, azok sorrendje nem közömbös a típus miatt. Az elkapó blokkokat utódosztály- őssztály sorrendben célszerű írni. 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*

VIII. Kivételkezelés

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
}
```

VIII. Kivételkezelés

```
}  
...  
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 abnormalis jelenség miatt lehet szükség kivételkezelésre, ekkor az egyik „kivételkezelő” a másiknak adhatja át a kezelés lehetőségét,

VIII. Kivételkezelés

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 hibajelenség 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 lekezeli az *Indexhibát* is, de ha a kezelő formális paraméterezése érték szerinti, 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()
    {
    }
}
```

VIII. Kivételkezelés

```
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);
    // A konstruktor szöveg paraméterét adja meg.
    Console.WriteLine(h.Message);
}
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!");
}
```


VIII. Kivételkezelés

}

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 SZEMÉTKYÚJTÁSI
    // 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
    }
}
```

VIII. Kivételkezelés

```
        else throw new VeremUres("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)
    {
    }
}

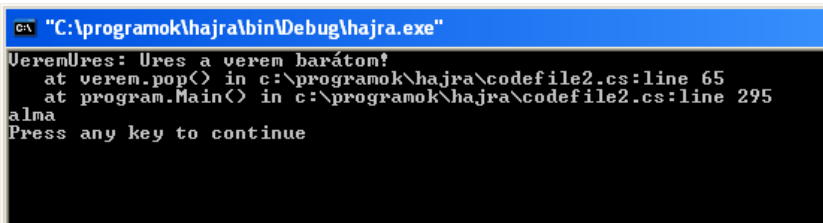
class program
{
    public static void Main()
    {
        verem v=new verem(6);
        try
        {
            Console.WriteLine(v.pop());
            // mivel a verem üres, kivételt fog dobni
        }
        catch (VeremUres ve)
        // amit itt elkapunk
        {

```

VIII. Kivételkezelés

```
        Console.WriteLine(ve);  
    }  
    v.push(5);  
    v.push("alma");  
    Console.WriteLine(v.pop());  
}  
}
```

A program futása az alábbi eredményt adja:



```
C:\ "C:\programok\hajra\bin\Debug\hajra.exe"  
VeremUres: Ures 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
```

2. ábra

IX. Típusparaméter, Generikus típusok, metódusok

Egy függvény vagy osztály definiálásakor a legelőször feltett kérdés az szokott lenni, hogy milyen paramétere(i) legyen(ek) a függvénynek, milyen adatmezőket kell az osztályon belül léltrehozni. Ha mondjuk egy tevékenységnek kétféle típussal is meg kell birkóznia, akkor definiálhattunk két azonos nevű függvényt, mindegyiket a megfelelő paraméter fogadására felkészítve. (VI.4. Függvénynevek átdefinálása)

Ez a megoldás egyszerű, csak az a legnagyobb baj vele, hogy nem a legtömörebb kódot eredményezi. A típusparaméter lehetőségének használata a hasonló esetekben tömörebb, sokkal kifejezőbb lehetőséget jelent. Ezt a lehetőséget gyakran „sablon” név alatt, -például C++ nyelvben (template)-, találjuk. A C# angol nyelvű környezete „Generic” névvel jellemzi ezt a tulajdonságot.

IX.1. Könyvtári típusparaméteres gyűjtemények

A könyvtári gyűjteményeket (tömb, lista, sor, ..) a *System.Collections* névtérben találhatjuk. Ezen hagyományos adattípusoknak megfelelő típusparaméterrel megvalósított változata a *System.Collections.Generic* névtérben található. Az üres projektváz is már a *System.Collections.Generic* névtérre való hivatkozást tartalmazza. Ez valószínűleg utalás akar lenni arra, hogy ha egy mód van rá, akkor javasolt a típusparaméteres változat használata.

A típusparaméter azonosítóját < > jelek között kell jelölni. Több típusparaméter is lehet, ekkor vesszővel kell elválasztani az azonosítókat.

A részletesebb magyarázat mellőzésével nézzük meg a könyvtári gyűjteményeket, és azok típusparaméteres megfelelőjét.

A Generic névtér típusaihoz az alábbi megfeleltetést adhatjuk a *System.Collections* névtérből.

<i>System.Collections.Generic</i>	<i>System.Collections</i>
<i>Comparer<T></i>	<i>Comparer</i>
<i>Dictionary<K,T></i>	<i>HashTable</i>
<i>LinkedList<T></i>	-
<i>List<T></i>	<i>ArrayList</i>
<i>Queue<T></i>	<i>Queue</i>
<i>SortedDictionary<K,T></i>	<i>SortedList</i>
<i>Stack<T></i>	<i>Stack</i>
<i>ICollection<T></i>	<i>ICollection</i>
<i>IComparable<T></i>	<i>System.IComparable</i>
<i>IDictionary<K,T></i>	<i>IDictionary</i>
<i>IEnumerable<T></i>	<i>IEnumerable</i>
<i>IEnumerator<T></i>	<i>IEnumerator</i>

XVIII. Grafikus alkalmazások alapjai

`IList<T>`

`IList`

Példaként nézzük talán a beszédes nevűre változott `Dictionary<K,T>`, (szótár típus) használatát. Ennek a konstrukciónak, adatszerkezetnek az a jellemzője, hogy a több adatot tartalmazó szerkezetben az egyes elemekre nem (csak) számmal, hanem szöveges index-el tudunk hivatkozni. Ezt a típust gyakran asszociatív tömb névvel is nevezzük.

Példa:

```
static void Main(string[] args)
{
    Dictionary<string, string> szótár=new Dictionary<string, string>();
    szótár.Add("apple", "alma");
    szótár.Add("grape", "szőlő");
    szótár.Add("pear", "körte");
    szótár.Add("peach", "barack");
    // Az adatszerkezet kulcs-érték párait egy anonym típusba
    // rakva végignézzük.
    foreach(var szó in szótár)
        Console.WriteLine("{0} : {1}",szó.Key,szó.Value);
    // Kiíratjuk, az adatpárok számát.
    Console.WriteLine(szótár.Count);
    // „Extension” metódus, egy tevékenységet végez ezen
    // (felsorolható) elemeken, egy transzformáció végrehajtásával.
    // Ezt a transzformációt jellemzően egy Lambda kifejezéssel adjuk
    // meg. Egy elemet helyettesítünk egy másik értékkel.
    //
    Console.WriteLine(szótár.Average(szó1 => szó1.Value.Length));
    // eredmény: 5
}
```

IX.2. Osztályok típusparaméterrel

Egy új osztály definiálásakor gyakran felmerül, hogy valamelyik adatmezőről nem tudjuk eldönteni, hogy milyen típusú legyen. Hasonló igény az, hogy abban az adatban többféle típust tudjunk tárolni. Ekkor szeretnénk a definiálandó osztályban az adott mező típusát nem egy konkrét típussal (int, string, stb) jelölni.

Ezt a típus azonosítót tetszőleges névvel jelölhetjük. Az irodalomban ezt leggyakrabban a *type* szóból adódóan T betűvel szokták jelölni, így mi is ezt a formát használjuk.

A következő példában nézzük az előző fejezetben is használt verem típust. Azon túl, hogy a típus nevet T-vel jelöljük, az osztály definiálásakor a típus nevet < T > formában az osztálynév után kell jelölni. Ehhez a formához hasonlóan a példányok definiálásakor a <> jelek közt kell megadni, hogy az adott példány milyen típussal jön létre. (Pl: <string>)

Példa:

```
using System;
```

XVIII. Grafikus alkalmazások alapjai

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

namespace genpelda
{
    class verem<T>           // a típus nevet T itt jelölni kell
    {
        T[] v;               // egy tömbben tároljuk az elemeket
        int verem_tető;
        public verem(int db)
        {                     // konstruktorban foglaljuk le a tömböt
            v = new T[db];
            verem_tető = 0;
        }
        public void push(T t)
        {
            if (verem_tető < v.Length)
                v[verem_tető++] = t;
            else
                throw new Exception("Tele a verem!");
        }
        public T pop()
        {
            if (verem_tető == 0)
                throw new Exception("Üres a verem");
            return v[--verem_tető];
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // verem string elemekkel
            verem<string> sver = new verem<string>(5);
            sver.push("Alma");
            // verem egészek számára
            verem<int> ver = new verem<int>(10);
            try
            {
                ver.push(10);
                int x1 = ver.pop();
                Console.WriteLine(x1); // 10
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

XVIII. Grafikus alkalmazások alapjai

```
    }  
    }  
}
```

A típusparaméteres megoldás legfontosabb előnyei:

- Nincs boxing, unboxing az általános Object típusból
- Nem kell típuskényszerítést(cast) használni.
- Kód újra felhasználható.
- Fordítási időben kap a típus értéket, nagyon hatékony kódot eredményez.

Természetesen felmerülhet az igény, hogy a definiálandó osztályunkban ne csak egy típusparaméter legyen, hanem kettő vagy több. Ez megengedett, és a típusparamétereket vesszővel kell felsorolni a <> jelek között: <T, T1, T2>.

A példa után rögtön meg kell jegyezni, hogy a típusparaméterrel (T) rendelkező elemeken nem lehet aritmetikai műveleteket végezni. (C++ template definíciókor lehet!) Ha a definiált típussal valamilyen műveletet akarunk végezni, akkor azt a típusra vonatkozó megszorításokkal, vagy paraméterül átadott műveleti függvénnnyel érhetjük el.

Tegyük fel, hogy a fenti példában a verem legnagyobb elemét szeretnénk meghatározni. Ekkor a T elemtípusra előírást tudunk (kell) megfogalmazni, hogy T milyen lehet ahhoz, hogy két T típusú elemet tudjunk összehasonlítani. A paramétertípusra ilyen előírásokat a where kulcsszó segítségével tudunk megadni.

Példa:

```
class verem<T> where T : System.IComparable<T>  
{  
    ...  
    public T maximum()  
    {  
        T t = default(T); // T inicializálása  
        t=v[0];           // legnagyobb a veremben az első.  
        for(int i=0;i<verem_tető;i++)  
            if (t.CompareTo(v[i]) < 0)  
                t=v[i];  
        return t;  
    }  
}  
  
static void Main(string[] args)  
{  
    verem<int> ver = new verem<int>(10);  
    // Mivel az int IComparable ezért rendben van  
    verem<string> ver1 = new verem<string>(5);  
    // string verem, ez is jó  
    // verem<Program> ver2 = new verem<Program>(5);  
    // Program verem, ez hibás lenne, mert a Program osztály nem  
    // implementálja IComparable-t.  
    try
```

XVIII. Grafikus alkalmazások alapjai

```
{
    ver.push(5);
    ver.push(10);
    ver.push(8);
    Console.WriteLine("Verem maxelem: {0}.", ver.maximum());
// 10

    int x1 = ver.pop();
    Console.WriteLine(x1);
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
}
```

Az iménti példában nem a teljes forráskódot adtuk meg, csak az előző példához képesti változásokat.

A *default* kulcsszó a T inicializálását segíti, ha T érték típusú, akkor 0 lesz az eredmény, míg referencia esetén null.

Mielőtt tovább mennénk nézzünk egy példát arra is, hogyan tudunk egy aritmetikai műveletet (összeadás, .stb) megvalósítani egy generic osztályban.

Ami biztos, és maradjunk az összeadásnál, két T típusú elemet nem tudunk összeadni! Viszont a közvetlen műveletet végző, két egész, két valós stb. összeadását végző függvényt egy paraméterben át tudjuk adni a kívánt metódusnak!

Nézzük konkrétan az iménti típusparaméteres veremünket. Tegyük fel, hogy szükségünk van egy a verem elemeit összegző metódusra. Amit tudunk a korábban leírtakból az, hogy az alábbi megoldás biztosan rossz (a jelenlegi 3.0 verzióban).

Példa:

```
T összeg()
{
    T szum=default(T);
    for(int i=0;i<verem_tető;i++)
        szum+=v[i]; // ez hibás, szum is és v[i] is T típusú
        // így ezeket nem tudjuk összeadni!!!
    return szum;
}
```

Ezen egyszerű összegzés helyett kicsit ravaszabb formát kell használnunk olyan módon, hogy paraméterül adjuk át a közvetlen műveletet megvalósító utasítástörzset. Ez nem jelent mást, mint azt, hogy definiálnunk kell egy delegáltat (egy delegált valójában a konkrét használatkor egy valós függvényt jelent) és ezt paraméterül adjuk az összeg függvénynek!

Példa:

```
class verem<T> where T : System.IComparable<T>
{
    // az aritmetikai műveletet végző konkrét függvény formája
    public delegate K művelet<K>(K a, K b);
```


XVIII. Grafikus alkalmazások alapjai

```
T[] v;
int verem_tető;
public verem(int db)
{
    v = new T[db];
    verem_tető = 0;
}
public void push(T t)
{
    if (verem_tető < v.Length)
        v[verem_tető++] = t;
    else
        throw new Exception("Tele a verem!");
}
public T pop()
{
    if (verem_tető == 0)
        throw new Exception("Üres a verem");
    return v[--verem_tető];
}
public T maximum()
{
    T t = v[0];
    for(int i=1; i<verem_tető; i++)
        if (t.CompareTo(v[i]) < 0)
            t=v[i];
    return t;
}
public T összeg(művelet<T> f)
{
    T szum = default(T); //szum inicializálása
    for (int i = 0; i < verem_tető; i++)
        szum = f(szum, v[i]);
    return szum;
}
}
```

Az iménti *összeg* függvényünk funkcionalitását tekintve többféle műveletet végezhet, kicsit másféle módon is bejárhatná az osztály adatait, és eredményül adná a kívánt eredményt, „következtetést” (összeg, átlag, szórás stb.). Az ilyen jellegű függvényt a dokumentációban gyakran *aggregate* néven találunk.

Ezzel persze nem oldottuk meg meg a problémát, csak az *f* műveleti paraméterrel jeleztük, hogy ide valamilyen függvény kerül átadásra. Az így módosított veremünk használatához többféle lehetőségünk is van.

XVIII. Grafikus alkalmazások alapjai

1. Használhatjuk a klasszikus módszert. Definiáljuk a műveletet, egy konkrét függvényt, majd ezzel egy delegáltat inicializálunk és ezt átadjuk paraméterként. (Ez a leghosszabb megoldás, így ezt az olvasóra bízom!)
2. Anonymus metódus használatával.
3. Lambda kifejezés használatával.

Példa:

```
// main függvénybeli részlet
...
// Anonymus metódus módszer
Console.WriteLine("Verem elemek összege: {0}.",
ver.összeg(delegate (int a,int b){ return a+b;}));

// Lambda kifejezés segítségével, delegált megadása
Console.WriteLine("Verem elemek összege: {0}.",
ver.összeg((int a, int b) => { return a + b; }));

// Szintén Lambda kifejezés, csak elhagyható a típus megadás
// ezt a fordító kitalálja az aktuális használatból verem<int>
// ha a törzs egy utasítás, akkor a return kulcsszó is és
// a {} zárójelek is elmaradhatnak
Console.WriteLine("Verem elemek összege: {0}.",
ver.összeg((a,b)=> a+b ));
```

A generic T típusra az alábbi formájú és jelentésű előírásokat fogalmazhatunk meg.

1. where T: struct T value típusú lehet
2. where T: class T referencia típusú lehet
3. where T: new() T rendelkezik paraméter nélküli konstruktorral
4. where T: osztály T adott osztály típusú
5. where T: interfész T implementálja adott interfészt
6. where T: U T és U is paraméter, és T valójában U típusú

Egyszerre több előírás is megadható egy osztálydefiníció során.

Példa:

```
class EmberList<T> where T : Ember, System.IComparable<T>, new()
{
    // Emberlista típusparamétere T Ember típusú,
    // IComparable-t implementálnia kell és
    // rendelkezni kell paraméter nélküli konstruktorral.(new)
    ...
}
```

Típusparaméter nem lehet statikus típus. Nem lehet generic attribútumot definiálni (XIII. fejezet), de lehet genericben attribútumot használni! Nem lehet generic web szervízt sem létrehozni.

XVIII. Grafikus alkalmazások alapjai

IX.3. Típusparaméteres függvények

A típusparaméter után kiáltó függvények mintáját a VI.4 fejezetben láttuk. Kétféle maximum értéket kiválasztó függvényt is írtunk. Típusparaméter segítségével egy is elég lesz, és ebben a paraméter típusát csak jelöljük egy azonosítóval. Azaz nemcsak a paraméter, hanem annak típusa is paraméter lesz.

A jelölés hasonló az előző fejezetben tárgyalt generic osztálytípusnál megismerttel. Szintén a < > jelek között a függvény neve után kell megadni a típusparaméter azonosítóját (T).

Ha a függvényben a T típussal valamilyen műveletet akarunk végezni, akkor azt biztosítani kell, hogy a T típus alkalmas legyen arra a műveletre! Például nem tudok összehasonlítani két T típusú változót csak akkor, ha a megfelelő hivatkozást megadtuk ehhez! (IComparable)

Minden további részletes és általános leírás helyett nézzük az így kapott maximum függvényünket.

Példa:

```
// Generic maximum függvény
// Itt is alkalmazni kell a T típusra vonatkozó megszorítást
// where T: ..., mint korábbi verem példában
T maximum<T>(T x, T y) where T : System.IComparable<T>
{
    if (x.CompareTo(y)>0)
        return x;
    else
        return y;
}

static void Main(string[] args)
{
    Program p=new Program();
    double s = p.maximum(5, 6.2);
    Console.WriteLine(s);
}
```

X. Adatlekérdezés: LINQ

A Visual Studio 2008 keretrendszer részeként megjelent C# 3.0 programozási nyelv talán legfontosabb újítása a *LINQ* (*L*anguage *I*Ntegrated *Q*uery). Ezt a programozási nyelvbe integrált adatlekérdezési lehetőséget már az elnevezéséből is ismerősnek halljuk. Méginkább így van ez akkor, amikor a tulajdonság jellemző kulcsszavait is leírjuk: *from*, *where*, *select*. Igen, a már régóta használt SQL nyelvből ismerősek ezek a szavak, és a C# nyelvbe integráltan azt a lehetőséget nyújtják, hogy lekérdező kifejezéseket alkossunk a segítségükkel. Ezeket a kifejezéseket gyakran *LINQ* lekérdezés, vagy *LINQ* kifejezés néven nevezzük.

A lekérdező kifejezések segítségével akár komplex szűrő, sorrendiséget megadó, csoportosításokat végző műveleteket adhatunk meg egyszerű formában. Ezek a kifejezések egy adatforráson végzik el a műveleteket. Ez a lekérdezés független az adatforrástól, ami lehet akár egy SQL adatbázis, egy XML dokumentum vagy egy .NET gyűjtemény(collection), ami a legegyszerűbb esetben egy szimpla tömb.

X.1. LINQ alapok

Mielőtt belemennénk a részletekbe, lássunk egy egyszerű, de teljes példát a LINQ használatára. A példa előtt elmondhatjuk, hogy egy lekérdezéses feladat végrehajtása jellemzően három részből áll.

1. Adatforrás definiálása
2. Lekérdező kifejezés megadása
3. A lekérdezés eredményének felhasználása

Példa:

```
static void Main()
{
    // 1. adatforrás megadása. Ez legegyszerűbb esetben
    // egy tömb, jelen esetben egész tömb
    int[] halszám = new int[] { 4, 5, 7, 6, 3 };
    //
    // 2. Lekérdező kifejezés megadása.
    IEnumerable<int> halszámok =
        from hal in halszám
        where hal > 4
        select hal;

    // 3. A lekérdezés végrehajtása. A kapott felsorolható
```

XVIII. Grafikus alkalmazások alapjai

```
// adatsoron (IEnumerable<int>) a foreach végiglépdel.  
foreach (int i in halszámok)  
{  
    Console.WriteLine(i);  
}  
// Eredmény: 5, 7, 6  
}
```

X.2. Adatforrás megadása

A kívánt lekérdező kifejezés definiálása előtt a legfontosabb lépés az, hogy megadjuk azt az adatforrást, amelyen a lekérdezést szeretnénk végrehajtani.

Ahogy az előző példában is láttuk, a legegyszerűbben megadható adatforrás egy tömb. Általában elmondható, hogy a LINQ adatforrása olyan típus, ami egy `IEnumerable<T>` interfészt implementál, vagy annak utóda.

A nem generic `IEnumerable` típusok, mint például az `ArrayList`, vagy a sima tömb (`int[]` halszám;) szintén megfelelő LINQ adatforrás.

Példa:

```
// egy tömb megfelelő LINQ forrás  
int[] halszám = new int[] { 4, 5, 7, 6, 3};  
// Egy ArrayList is az  
ArrayList adatok=new ArrayList();
```

Az adatforrások megadásánál jelentős szereppel bír az, mikor nem egy memóriatípusban (tömb), hanem egy állományban vannak az adatok. Ezen fájlok nem lehetnek tetszőleges formátumúak. Az XML állományok lekérdezhetőségét az `XElement` osztály biztosítja.

Példa:

```
// XML dokumentum adatforrás megadás.  
using System.Xml.Linq;  
...  
XElement contacts = XElement.Load(@"c:\adatok.xml");  
// contacts lekérdezhető
```

Adatbázis állományok (Access vagy SQL) közvetlen lekérdezhetőségét a `DataContext` osztály biztosítja. Ahhoz, hogy ezt az osztályt el tudjuk érni, először a projekthez kell adni a `System.Data.Linq` referenciát. (Project menüpont, Add Reference) Mielőtt a `DataContext` típusból használni tudnánk az adatforrást, a projekthez kell adni egy az adatbázis táblát (mezőket) leíró osztályt. Az osztály és mezőnevekhez a megfelelő hozzárendelési attribútumot meg kell adni. (Mapping)

XVIII. Grafikus alkalmazások alapjai

Példa:

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

namespace linqtosql1
{
    class Program
    {
        // Táblanév(attribútum) megadása, ha több táblánk lenne, akkor
        // a csapatok osztályhoz hasonlóan, a többi tábla-osztály
        // hozzárendelést is meg kell adni!
        [Table(Name = "csapatok")]
        public class csapatok
        {
            // a csapatok táblában két mező (oszlop) van definiálva
            private string _sorszam;
            // Tábla sorszám oszlopnév(attribútum) megadása
            [Column(Storage = "_sorszam")]
            public string sorszam
            {
                get { return _sorszam; }
                set { _sorszam = value; }
            }

            private string _csapatnev;
            // Tábla csapatnév oszlopnév(attribútum) megadása
            [Column(Storage = "_csapatnev")]
            public string csapatnev
            {
                get { return _csapatnev; }
                set { _csapatnev = value; }
            }
        }

        static void Main(string[] args)
        {
            // Kapcsolat létrehozása
            DataContext dc = new DataContext(@"c:\flsz.mdf");
            // Típusos csapatnevek tábla objektum létrehozása
            Table<csapatok> csapatnevek = dc.GetTable<csapatok>();
            //Ez a tábla már IEnumerable<>, IQueryable<> lesz , így
            //a LINQ lekérdezés már tud működni a csapatnevek táblán
        }
    }
}
```

XVIII. Grafikus alkalmazások alapjai

```
//lekérdezzük a neveket
var nevek = from a in csapatnevek
            select a.csapatnev;

foreach (string s1 in nevek)
    Console.WriteLine(s1);
}
}
```

Ha az adatforrásunk nem egy egyszerű adatbázis állomány, hanem egy adatbázis kiszolgáló, akkor egy sorában fog változni az iménti példa. A DataContext osztály konstruktora egy adatbázis kapcsolatot vár paraméterül. Ez azt jelenti, hogy az adatbázis kiszolgálóhoz a paramétereket megadó nevet, „connection string”-et kell paraméterül adni. Ez az ADO.NET világból már ismerős lehet. Ebben a könyvben nem részletezzük ezt a könyvtári szolgáltatás halmazt, de enélkül is érthető lesz a következő példa.

Példa:

```
// adatbázis fájl adatforrás megadás.
DataContext dcl =
new DataContext(@"Data Source=localhost\squlexpress;Initial Catalog=FLSZ;Integrated Security=True");
```

Az adatbázis kapcsolatot leíró karaktersorozatot vagy kézzel megadjuk, vagy a Server Explorer ablakban készítünk egy új adatkapcsolatot az adatbázis kiszolgálóhoz, és a megadott paraméterekből az ottani kis varázsló megadja. A kapcsolati szövegben a Data Source az adatbázis szervert adja meg, az Initial Catalog pedig az adatbázis nevét! (Ebben a példában ugyanazt az FLSZ adatbázist használtuk mind az önálló állománybeli elérésnél, mind a második példában, a helyi gép (localhost) SQL Express adatbázis kiszolgálóján keresztüli elérésnél is.)

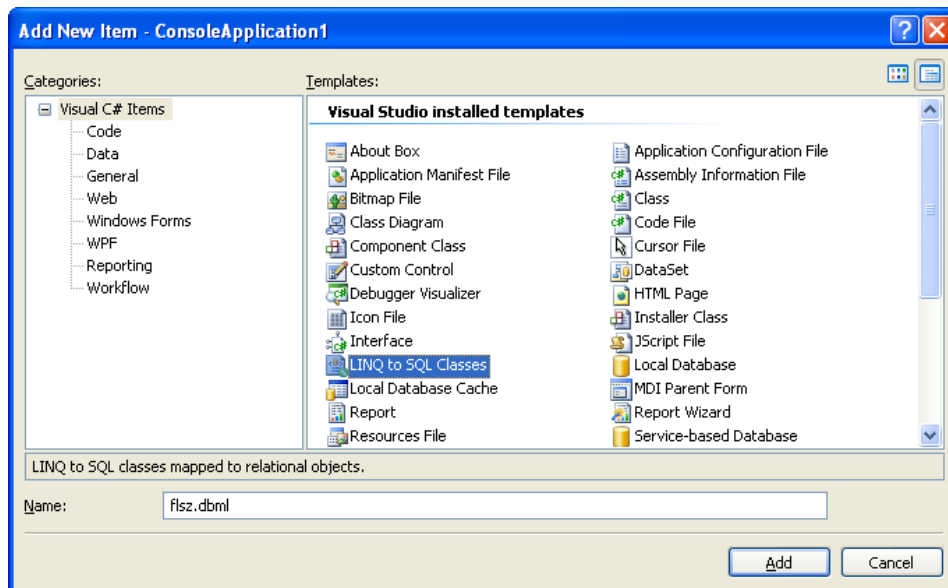
X.3. O/R tervező

Láttuk az előző részben, hogy az alkalmazásunkhoz szükséges adatforrást egy pár soros osztálydefinícióval meg tudjuk adni. Ha az adatbázis állomány sok táblát tartalmaz, tárolt eljárások, függvények állnak rendelkezésünkre, akkor kézzel megadni az összes hozzárendelést, bár nem bonyolult, de mindenképpen hosszadalmas.

A Visual Studio 2008 O/R Designer (Object Relational) néven segédeszközt biztosít ahhoz, hogy ne kelljen kézzel ilyen hozzárendeléseket végezni. Az O/R tervezőnek létezik egy parancssoros változata is (sqlmetal.exe), aminek bemutatását mellőzzük.

Az O/R tervező bekapcsolása nem jelent mást, mint azt, hogy hozzá kell adni a projekthez egy LINQ to SQL osztályt.

XVIII. Grafikus alkalmazások alapjai



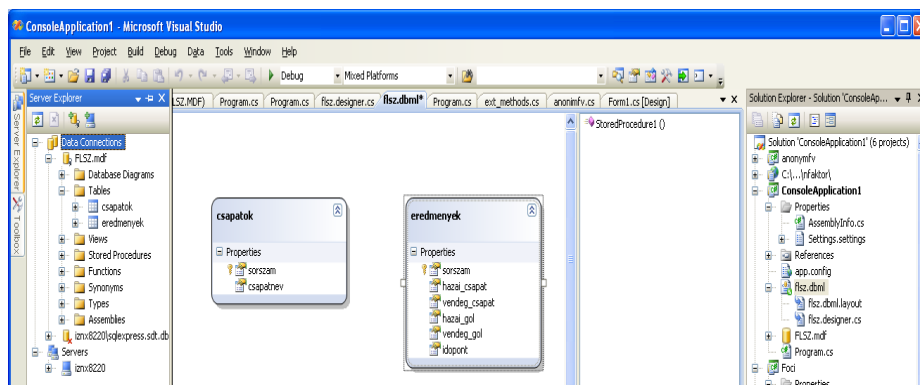
12. ábra

Ez egy .dbml állomány (sajátos XML fájl) és még két fájl (flsz.dbml.layout, flsz.designer.cs) hozzáadását jelenti.

A következő lépésben meg kell adnunk a projektben, hogy hol van az adatbázis állomány. Ezt a Server Explorer segíti. A Data Connection menüponton a jobb egérgombra kattintunk, és kiválasztjuk az Add Connection pontot. A megjelenő dialógus ablakban megadjuk azt, hogy SQL fájlról van szó, és pontosan meg is adjuk a helyét.

Ha mindezt sikeresen elvégeztük az alábbi felülethez hasonlóan kell kapnunk. (Az adatfájlt a projektbe másolja, látszik a Solution Explorer ablakban a megjelenő FLSZ.mdf, és a Server Explorer ablakban láthatjuk az adatbázis tábláit, tárolt eljárásait, stb.)

Az flsz.dbml felület bal oldali részére húzzuk rá a két adatbázis táblát, a jobboldali részre pedig a tárolt eljárásokat és függvényeket. (Ha vannak.)



13. ábra

Ezen művelet hatására az flsz.designer.cs állományt érdemes megnézni. Tartalmazza mindazokat a táblaleírókat hozzárendeléseket, melyeket kézzel kellett megadni, ahogy az előbbi fejezetben is láttuk. Láthatjuk azt is, hogy a DataContext osztály helyett egy flszDataContext osztályt kapunk. Ebbe el van „rejtve” az állományunk flsz.mdf neve. Látható az is, hogy a táblákhoz

XVIII. Grafikus alkalmazások alapjai

„sajátos” nevű tulajdonságfüggvény tartozik. A táblanév után kerül egy s betűt, és az így kapott nevű tulajdonság adja meg a táblákat.

Példa:

```
flszDataContext dc = new flszDataContext ();  
Table<csapatok> csnevek = dc.csapatoks;  
var nevek = from a in csnevek  
            select a.csapatnev;  
foreach (string s in nevek)  
    Console.WriteLine(s);
```

X.4. Lekérdezés kifejezések készítése

A nyelvi elemek segítségével készített lekérdezések (LINQ) azt a célt szolgálják, hogy hatékony, könnyen olvasható lekérdezéseket, transzformációkat tudjunk készíteni. Gyakori, hogy egy adatforrásból származó adatokat a lekérdezés végén egy másik állományba írunk.

A lekérdezések fontosabb jellemzői a következők:

- Könnyű használhatóság, sok C# nyelvi konstrukciót használnak.
- A lekérdezés eredménye szigorúan típusos (ahogy a nyelv is), bár sok esetben a fordítóra ráhagyjuk, hogy találja ki az eredmény típusát!
- A lekérdezés valójában a foreach utasításban hajtódik végre.
- Lekérdezést vagy kifejezés formában vagy metódus (extension method) formában írhatunk.
- Bár a fordító is átalakítja a kifejezést metódus formára, azért tanácsos a kifejezés formának a használata a jobb és könnyebb olvashatóság miatt.
- Lekérdezéseket, extension metódusokat felsorolható, lekérdezhető típusokon (IEnumerable<>, IQueryable<>) végezhetünk. Legegyszerűbb ilyen típus a tömb.

Egy lekérdezés utasítás jellemző formája kifejezéses alakban a következő:

eredmény = **from** név **in** forrás
[kiegészítő feltételek]
select elem vagy **group** elem;

Tehát a lekérdezés kifejezésforma egy from kulcsszóval kezdődik, és a select vagy group parancs zár. A kettő között lehet még from parancs, illetve a where, orderby, join, let parancs.

A metódus forma ehhez nagyon hasonlít:

eredmény = forrás.
[kiegészítő feltételek.]

XVIII. Grafikus alkalmazások alapjai

Select(Lambda kifejezés);

A szűrési feltételek a Selecthez hasonlóan Lambda kifejezést várnak paraméterül.

Lássuk a legelső példánkat, mikor a legegyszerűbb adatforrásból (egész tömb) kérdezzük le adatokat.

Példa:

```
// egy tömb a forrás
int[] halszám = new int[] { 4, 5, 7, 6, 3 };
// a legegyszerűbb lekérdezés, nincs feltétel, minden
// elemet megkapunk
IEnumerable<int> halszámok =
    from hal in halszám // nem hagyható el lekérdezésből
    select hal;

// Lekérdező kifejezés megadása. Eredmény típusát
// konkrétan megadjuk
IEnumerable<int> halszámok1 =
    from hal in halszám
    where hal > 4 // elhagyható ha nem kell feltétel
    select hal;
// A lekérdezés végrehajtása.
foreach (int i in halszámok1)
{
    Console.WriteLine(i);
}
// Eredmény: 5, 7, 6
// A fenti kifejezés forma metódus (extension) hívásos
// alakja
// Eredmény típusát ráhagyjuk a fordítóra, hadd találja ki!
var hsz = halszám.
    Where(x => x > 4).
    Select(x => x);
foreach (int i in hsz)
{
    Console.WriteLine(i);
}
```

Ezek után nézzük egyenként a lekérdezés kifejezés parancsait.

from parancs

Ezzel kezdődik a lekérdezés. Meghatározza, hogy melyik adatforrást kérdezzük le, és megadja a forrás egységét, azt az objektumot, amelyekből az eredmény képződik. (from **hal** in halszám) Ezt

XVIII. Grafikus alkalmazások alapjai

a változót (hal) szokás, forrás változónak nevezni (range variable), hiszen ez a változó felveszi sorban az adatforrás értékeit.

Ha az adatforrás nem egy egyszerű tömb, hanem olyan gyűjtemény amelynek elemei közt is van gyűjtemény, akkor ezeken a gyűjteményelemeken egy újabb from paranccsal tudunk végigmenni.

Példa:

```
int[][] hőmérséklet = {new int[] { 2, 4, 1, 5, 8 },
                       new int[] { 6, 2, 4, 1 },
                       new int[] { 5, 2, 9, 1, 7, 4 } };

var er = from nap in hőmérséklet
         from fok in nap
         where fok > 6
         select nap;

Console.WriteLine("A gyűjtemény elemszáma:{0}",er.Count());
// eredmény 3, a harmadik napot kétszer kiválasztja!!!
foreach (var n in er)
{
    Console.WriteLine("Napi hőmérsékletek:");
    foreach (int j in n)
        Console.WriteLine(j);
}
```

A példában azt látjuk, hogy a hőmérséklet tömb elemei tömbök. Ezért ahhoz, hogy a belső tömb elemein is tudjunk valamilyen kiválasztást végezni, ezért kell a második from utasítás. Azokat a belső tömböket (nap) vesszük, amelyekben találunk 6 foknál magasabb hőmérsékletet. (fok>6).

Látjuk az eredmény gyűjtemény nap objektumokból (tömb) áll. A new utasítással akár egy új anonymous típusból álló eredménygyűjteményt is készíthetünk. Ezt mutatja a következő példa.

Példa:

```
var er = from nap in hőmérséklet
         from fok in nap
         where fok > 6
         select new { fok, nap };

// eredmény olyan típus lesz, melyiknek lesz fok és
// nap mezője
Console.WriteLine("A gyűjtemény elemszáma:{0}",er.Count());
foreach (var n in er)
{
    Console.WriteLine("Napi hőmérséklet:{0}",n.fok);
    foreach (int j in n.nap)
        Console.WriteLine(j);
}
```

select parancs

XVIII. Grafikus alkalmazások alapjai

Ahogy a from parancs nélkülözhetetlen egy lekérdezés kifejezés megadásakor, úgy a select is az. A from paranccsal megadjuk az adatforrást, a select-tel pedig a kiválasztást.

A használat legegyszerűbb formája a következő:

Példa:

```
int[] számok = { 2, 4, 1, 5, 8 };
var er = from x in számok
        select x;

// eredmény az eredeti számok
```

Egy lekérdezés vagy a select vagy a group utasítással fejeződik be. A select utasításnak leggyakrabban a forrás változó, vagy abból a new parancs segítségével létrehozott új változó a paramétere.

A klasszikus SQL lekérdezés az iménti kifejezés fordítottja. (select x from y)

group parancs

Említettük már, hogy a group (csoportosítás) parancs akár helyettesítheti a select parancsot. Míg a select parancs egyszerűen összeszedi egy gyűjteményben a kívánt elemeket, addig a group csoportok gyűjteményét adja eredményül. Egy ilyen lekérdezés pontos típusa:

IEnumerable<IGrouping<kulcs típus, elem típus>>

A csoportokban szereplő elemet a group parancs után, a csoportosítás kulcsát a by kulcsszó után kell megadni.

Nézzük a következő példát, amiben a neveket a kezdőbetűik alapján csoportosítjuk.

Példa:

```
// adatforrás megadása
string[] keresztnevek = { "Zoli", "Judit", "Zsolt", "József", "Zsuzsa" };
// a csoportosítás olyan csoportok sorozatát adja
// melyet a nevek első karaktere alapján végez el
// minden csoportban visszkapjuk a Key mező alakjában
// a csoport kulcsát, karakterét
IEnumerable<IGrouping<char, string>> csoportok =
    from név in keresztnevek
    group név by név[0];
foreach (IGrouping<char, string> gr in csoportok)
{
    Console.WriteLine("A kezdőbetű: {0}", gr.Key);
    foreach (string név in gr)
        Console.WriteLine(név);
}
```

Látható ezen csoportosításhoz tartozó típusoknál, hogy igen hasznos tulajdonsága a fordítónak, hogy kitalálja helyettünk a megfelelő típust, és így a következő rövidebben írható alak is használható:

```
var csoportok =
```

XVIII. Grafikus alkalmazások alapjai

```
from név in keresztnevek
group név by név[0];
foreach (var gr in csoportok)
{
    Console.WriteLine("A kezdőbetű: {0}", gr.Key);
    foreach (var név in gr)
        Console.WriteLine(név);
}
```

Összetett csoportosítási feltételt is megadhatunk, ekkor a csoport kulcs egy anonim típus lesz, és az előforduló különböző összetett adatok szerint fognak a csoportok létrejönni.

Példa:

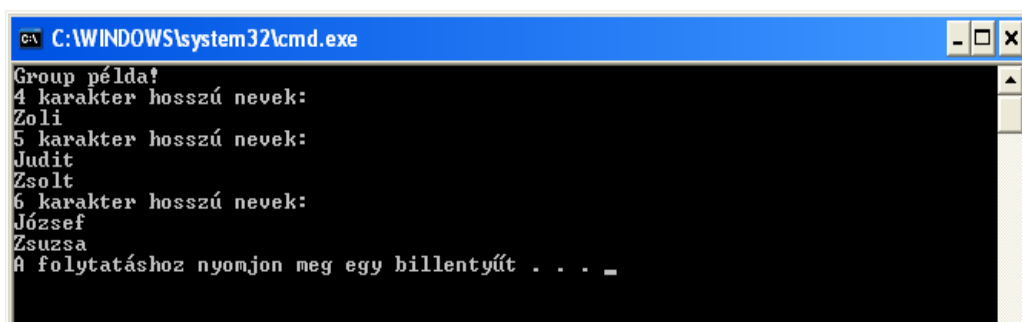
```
..
group személy by new {név = személy.név, ország=személy.ország};
// minden névhez, országnévhez egy-egy csoport lesz megadva
```

Logikai érték alapján (true, false) is csoportosíthatunk, vagy egy számolt érték alapján. Nézzük például, azt az esetet mikor a nevek hossza alapján csoportosítunk.

Példa:

```
static void Main(string[] args)
{
    string[] keresztnevek = { "Zoli", "Judit", "Zsolt", "József", "Zsuzsa" };
    //
    Console.WriteLine("Group példa!");
    var csoportok =
        from név in keresztnevek
        group név by név.Length;
    foreach (var gr in csoportok)
    {
        Console.WriteLine("{0} karakter hosszú nevek: ", gr.Key);
        foreach (var név in gr)
            Console.WriteLine(név);
    }
}
```

Az alábbi eredményt kapjuk a futás után.(14.ábra)



```
C:\WINDOWS\system32\cmd.exe
Group példa!
4 karakter hosszú nevek:
Zoli
5 karakter hosszú nevek:
Judit
Zsolt
6 karakter hosszú nevek:
József
Zsuzsa
A folytatáshoz nyomjon meg egy billentyűt . . . _
```

XVIII. Grafikus alkalmazások alapjai

14. ábra

into - azonosító

A group, join, select parancsok használata során felmerül gyakran, hogy a megfelelő objektumhoz hozzárendelünk egy ideiglenes változónevet, azért, hogy könnyítsük a számításainkat. Mivel módosul a forrásváltozó, és célszerű ezt új névvel elnevezni, hogy könnyebben hivatkozhassunk annak elemeire.

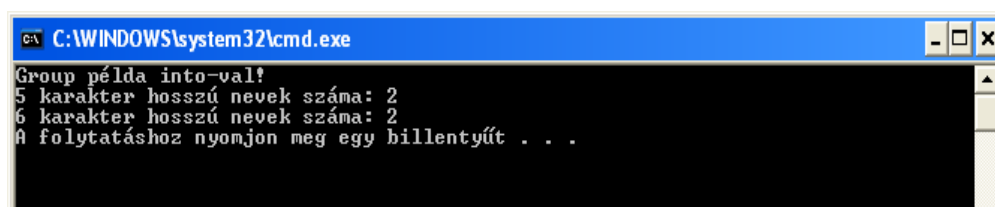
Nézzük az előző példát. A neveket a hosszuk szerint csoportosítottuk. A következő példában csak azokat a csoportokat vesszük, amikben nem egy név szerepel, és ebből a csoportból kivesszük a kulcsot, és a darabszámot. Ahhoz, hogy ezeket a számításokat elvégezzük, célszerű a group parancsban a csoportot egy ideiglenes változóba tenni. (into nevek)

Példa:

```
static void Main(string[] args)
{
    Console.WriteLine("Group példa into-val!");
    string[] keresztnevek = { "Zoli", "Judit", "Zsolt", "József", "Zsuzsa" };
    //
    var csoportok =
        from név in keresztnevek
        group név by név.Length into nevek
        where nevek.Count() > 1
        select new { hossz = nevek.Key, darab =nevek.Count() };
    foreach (var gr in csoportok)
    {
        Console.WriteLine("{0} karakter hosszú nevek száma: {1} ",
gr.hossz,gr.darab);
    }
}
```

Fontos megjegyezni, hogy a példában a valódi keresztnevek a nevek változóban maradnak, azt nem szelektáljuk ki! A szelektálás eredménye egy anonymous típusból (aminek két mezője van, hossz és darab) álló sorozat lesz.

Az előbbi példa futtatási eredményét az alábbi, 15. ábrán láthatjuk.



15. ábra

let – változó létrehozás

Az into azonosítónév létrehozó parancs mintájára, szintén új forrásváltozót hoz létre. A különbség annyi, hogy míg az into egy parancs kiegészítője (pl. group), addig a let egy önálló parancs. Ahol egy forrásváltozó helyett, abból egy új számítottal szeretnénk tovább dolgozni, ott ennek a parancsnak a segítségével megadjuk az új forrásváltozót.

XVIII. Grafikus alkalmazások alapjai

Példa:

```
static void Main(string[] args)
{
    Console.WriteLine("Let példa!");
    string[] keresztnemek = { "Zoli", "Judit", "Zsolt", "József", "Zsuzsa" };
    //lekérdezés
    var hosszak = from n in keresztnemek
                  let h = n.Length
                  where h > 4
                  select new { nevhossz = h, nev = n };

    // eredmény kiírás
    Console.WriteLine("Az eredmény:");
    foreach (var adat in hosszak)
    {
        Console.Write("A név hossza: {0}", adat.nevhossz);
        Console.WriteLine(" Név: {0}", adat.nev);
    }
}
```

A lekérdezés indulásánál az adatforrásból (adatsorozat, keresztnemek), létrehozuk az *n* forrásváltozót, amely sorban felveszi a megadott keresztnemeket. Szükségünk van a nevek hosszára (let *h*=...), majd vesszük azokat a neveket, melyeknek hossza nagyobb mint 4, és ezen nevekből (*n*), és hosszakból képzett névtelen típusú struktúrasorozat lesz az eredmény.

orderby - parancs

Az *orderby* parancs a lekérdezés eredményének sorrendjét befolyásolja. A parancs után megadott forrásváltozó nagyság szerinti sorrendjében rendezi az eredményt. Alapértelmezésben növekvő sorrendet határoz meg a parancs, míg a *descending* kiegészítővel ezt csökkenőre módosíthatjuk. (A növekvő sorrendet az *ascending* parancs kiírásával jelölhetjük is, de ez nem fontos mert ez az alapértelmezés!)

Példa:

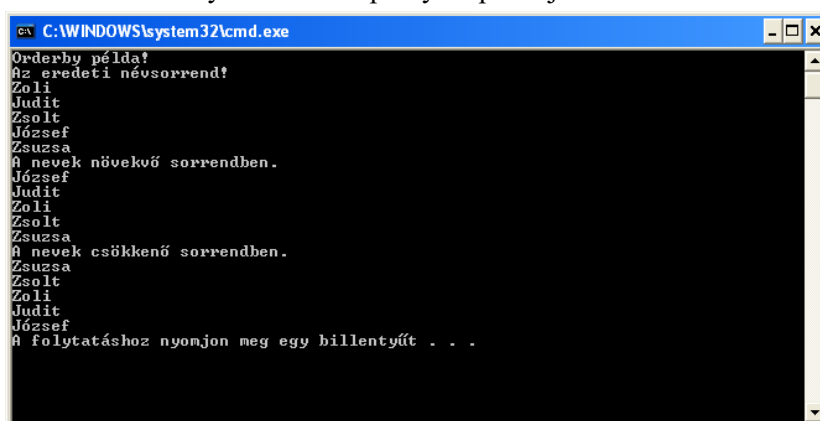
```
static void Main(string[] args)
{
    Console.WriteLine("Orderby példa!");
    string[] keresztnemek = { "Zoli", "Judit", "Zsolt", "József", "Zsuzsa" };
    //lekérdezés
    Console.WriteLine("Az eredeti névsorrend!");
    foreach (string név in keresztnemek)
        Console.WriteLine(nev);
    var újsorrend = from név in keresztnemek
                   orderby név // ascending, növekvő sorrend
                   // alapértelmezés ez, ezért nem kötelező kiírni
                   select név;

    Console.WriteLine("A nevek növekvő sorrendben.");
    foreach (string név in újsorrend)
```

XVIII. Grafikus alkalmazások alapjai

```
        Console.WriteLine(név);  
var csökkenő = from név in keresztnev  
               orderby név descending  
               select név;  
Console.WriteLine("A nevek csökkenő sorrendben.");  
foreach (string név in csökkenő)  
    Console.WriteLine(név);  
}
```

A futási eredmény az alábbi képernyőképet adja.



```
C:\WINDOWS\system32\cmd.exe  
Orderby példa!  
Az eredeti névsorrend!  
Zoli  
Judit  
Zsolt  
József  
Zsuzsa  
A nevek növekvő sorrendben.  
József  
Judit  
Zoli  
Zsolt  
Zsuzsa  
A nevek csökkenő sorrendben.  
Zsuzsa  
Zsolt  
Zoli  
Judit  
József  
A folytatáshoz nyomjon meg egy billentyűt . . .
```

16. ábra

Az `orderby` parancsnak megadhatunk többféle rendezési sorrendet is. Egyszerűen vesszővel elválasztva sorolhatjuk fel a rendezési elveket.

Példa:

```
orderby Vezetéknév, Keresztnév;
```

Ekkor a lekérdezés sorba rakja az elemeket vezetéknév szerint, majd az azonos vezetéknévű embereket még keresztnév szerint is sorba rendezi. Zárásként nézzünk egy kicsit összetettebb adatokat, és azokon értelmezett lekérdezéseket.

Példa:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
  
namespace orderby  
{  
    class orderbypélda  
    {  
        // Az elemi adatunk, összetett, nem egyszerű tömb.  
        // Kihasználva az automatikus tulajdonság lehetőséget  
        // csak jelöljük, hogy milyen műveletet akarunk az implicit  
        // definiált adatmezőkre (név, keresztnév, kor)  
        public class Hallgató
```


XVIII. Grafikus alkalmazások alapjai

```
{
    public string Vezetéknév { get; set; }
    public string Keresztnév { get; set; }
    public int Kor { get; set; }
}

static void Main(string[] args)
{
    // Hallgató lista definiálás.
    List<Hallgató> hallgatók = new List<Hallgató>
{
    new Hallgató {Vezetéknév="Pfaff", Keresztnév="Károly", Kor=22},
    new Hallgató {Vezetéknév="Illés", Keresztnév="Zoltán", Kor=19},
    new Hallgató {Vezetéknév="Illés", Keresztnév="Katalin", Kor=20},
    new Hallgató {Vezetéknév="Bakonyi", Keresztnév="Viktória", Kor=20},
    new Hallgató {Vezetéknév="Takács", Keresztnév="Attila", Kor=22},
    new Hallgató {Vezetéknév="Farkas", Keresztnév="Csaba", Kor=21},
    new Hallgató {Vezetéknév="Mód", Keresztnév="Károly", Kor=22},
};

    // Lekérdezés.
    IEnumerable<Hallgató> névsor1 =
        from hallgató in hallgatók
        orderby hallgató.Vezetéknév, hallgató.Keresztnév
        select hallgató;

    // Lekérdezés végrehajtása.
    Console.WriteLine("A hallgatóink névsor szerint.");
    foreach (Hallgató hallgató in névsor1)
        Console.WriteLine(hallgató.Vezetéknév);
    Console.WriteLine(" " + hallgató.Keresztnév);

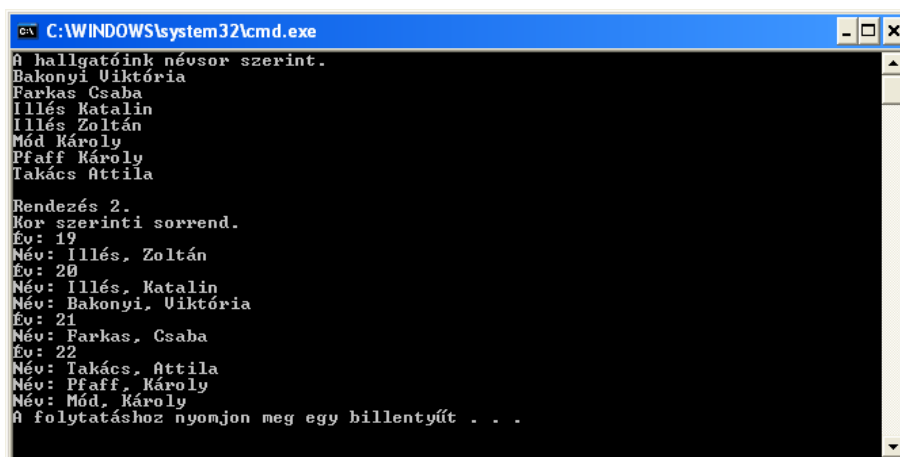
    // Lekérdezés 2.
    // Vesszük a hallgatókat sorban, majd rendezzük őket
    // keresztnév, és azon belül vezetéknév szerint.
    // Ezen a névsorba rendezett hallgatói listán Kor szerinti
    // csoportokat képezünk, és ezeket a csoportokat kor szerint
    // növekvő sorrendbe rakjuk.
    IOrderedEnumerable<IGrouping<int,Hallgató>> névsor2 =
        from hallgató in hallgatók
        orderby hallgató.Keresztnév, hallgató.Vezetéknév descending
        group hallgató by hallgató.Kor into csoport
        orderby csoport.Key
        select csoport;

    // Végrehajtás
    // A lekérdezések, és végrehajtások során az eredmény
    // típus valós esetben gyakran nem könnyen látható,
```

XVIII. Grafikus alkalmazások alapjai

```
// ezért sok példánál a var kulcsszóval jelölik, hogy a
// fordító találja ki a típust(IGrouping<int, Hallgató>)
Console.WriteLine(Environment.NewLine + "Rendezés 2.");
Console.WriteLine("Kor szerinti sorrend.");
foreach (IGrouping<int, Hallgató> csoport in névsor2)
// így is helyes lenne:
// foreach (var csoport in névsor2)
{
    Console.WriteLine("Év: {0}", csoport.Key);
    foreach (Hallgató h in csoport)
    {
        Console.WriteLine("Név: {0}, ", h.Vezetéknév);
        Console.WriteLine(h.Keresztnév);
    }
}
}
```

A futási eredmény az alábbi képernyőképen látható.



17. ábra.

join - parancs

A lekérdező kifejezések készítésének során gyakran több adatforrásunk is van. (Adatbázisok használatánál szinte mindig azt látjuk, hogy több tábla is van az adatbázisunkban.) Ezekben az adatforrásokban adott mezők kapcsolatot jelentenek az adatforrás adatai között. A join lekérdező parancs segítségével olyan összetett(ebb) lekérdezések készíthetők melyben több adatforrás szerepel, és ezek között az adott mezők teremtik meg a kapcsolatot.

A join parancs használatát tekintve ez azt jelenti, hogy a lekérdezés indul a szokásos from paranccsal, majd ehhez a forráshoz a join-al csatolunk egy másik forrást is. A két forrás közötti azonosítókat csak egyenlőségi kapcsolattal köthetjük össze. (*equals*) Ezzel egy adatkigyűjtéshez két különböző forrásból tudunk elemeket válogatni.

Nézzük a join használatát egy egyszerű sematikus példán:

XVIII. Grafikus alkalmazások alapjai

Példa:

```
var t=from név in hallgatók
join ered in eredmények on név.Azonosító equals ered.Azonosító
select ered.Tantárgy;
```

Két adatforrást (hallgatók, eredmények) definiál az iménti példa. Ezek között az egyenlőség (equals) kapcsolatot az azonosító mező adja meg. A lekérdezésben (a jelenlegi sematikus példában) csak egy adatot kérünk kiválasztani (Tantárgy).

Az egyenlőségi operátor (equals) használatánál nem mindegy a két összehasonlítandó mező sorrendje. A bal oldalra kell írni a from forrás mezőjét (név.Azonosító), míg a jobb oldalra a join-ban megadott forrás mezőt (ered.Azonosító). A from parancsban megadott forrást külső forrásnak, míg a join parancsban megadott forrást belső forrásnak is szokás nevezni.

Ezt a lekérdezés formát *inner join* lekérdezésnek nevezzük. Az elnevezés onnan származik, hogy a két adatforrás azon elemei kerülnek kiválasztásra, melyek között található reláció, így ezen adatok bekerülnek (inner) a lekérdezésbe.

Ezt a lekérdezési változatot módosíthatjuk azzal, hogy kiegészítjük a join parancsot az into csoportosítás parancssal.

Példa:

```
var lista1 = from név in hallgatók
              join ered in eredmények on
                  név.Azonosító equals ered.Azonosító into csoport
              select new {v=név.Vezetéknév,n=név.Keresztnév,cs=csoport};
```

A lekérdezés az előző példához képest abban módosul, hogy nem egy egyszerű adatsorozat, hanem csoport-ok sorozata az eredmény. Egy hallgatóhoz egy csoportba rakjuk az eredményeit. Jelen példánkban ezt még azzal egészítettük ki, hogy ezt a csoportot is egy újabb csoportba a v,n,cs mezők alkotta csoportba gyűjtöttük ki.

Ezen csoportosító tulajdonság miatt *group join* lekérdezésnek nevezzük ezt a kifejezésformát. Lényeges különbség az előző *inner join*-hoz képest, hogy azok a nevek is kiválasztásra kerülnek, amelyekhez nincs belső azonosító, így ehhez egy üres csoport kerül kiválasztásra!

A group lekérdezéstől eltérően a group join nem készít kulcs (Key) mezőt, a csoport egy egyszerű IEnumerable<> lesz!

Ezek után nézzük meg a fenti lekérdezéseket egy teljes példán keresztül.

Példa:

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

namespace join
{
    public class Hallgató
    {
        public int Azonosító { get; set; }
    }
}
```

XVIII. Grafikus alkalmazások alapjai

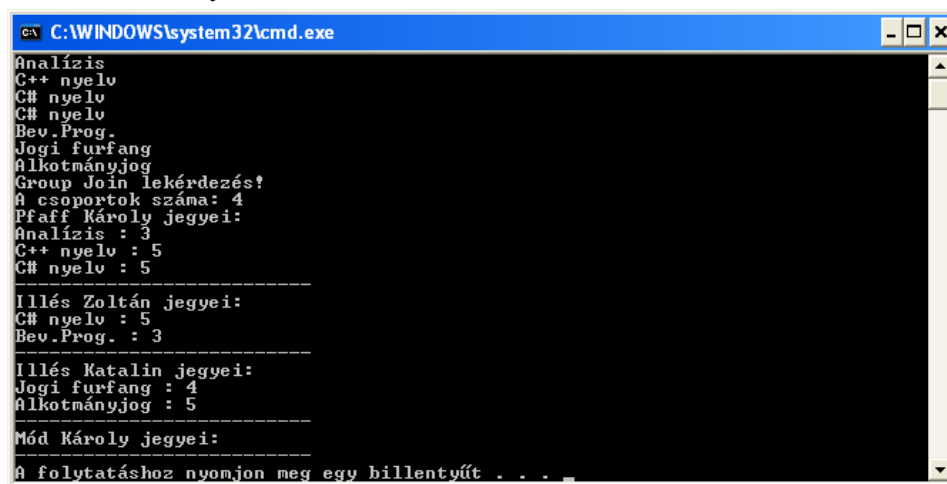
```
public string Vezetéknév { get; set; }
public string Keresztnév { get; set; }
public int Kor { get; set; }
}
public class Eredmény
{
    public int Azonosító { get; set; }
    public string Tantárgy { get; set; }
    public int Jegy { get; set; }
}
class joinpelda
{
    static void Main(string[] args)
    {
// a hallgatók lista lesz a külső adatforrás
        List<Hallgató> hallgatók = new List<Hallgató>
        {
            new Hallgató {Azonosító=1, Vezetéknév="Pfaff", Keresztnév="Károly", Kor=22},
            new Hallgató {Azonosító=2, Vezetéknév="Illés", Keresztnév="Zoltán", Kor=19},
            new Hallgató {Azonosító=3, Vezetéknév="Illés", Keresztnév="Katalin", Kor=20},
            new Hallgató {Azonosító=4, Vezetéknév="Mód", Keresztnév="Károly", Kor=22},
        };
// az eredmények lista lesz a belső adatforrás
        List<Eredmény> eredmények = new List<Eredmény>
        {
            new Eredmény {Azonosító=1, Tantárgy="Analízis", Jegy=3},
            new Eredmény {Azonosító=2, Tantárgy="C# nyelv", Jegy=5},
            new Eredmény {Azonosító=3, Tantárgy="Jogi furfang", Jegy=4},
            new Eredmény {Azonosító=2, Tantárgy="Bev.Prog.", Jegy=3},
            new Eredmény {Azonosító=1, Tantárgy="C++ nyelv", Jegy=5},
            new Eredmény {Azonosító=1, Tantárgy="C# nyelv", Jegy=5},
            new Eredmény {Azonosító=3, Tantárgy="Alkotmányjog", Jegy=5},
        };
// inner join
// végignézi a név.Azonosító-t, majd ehhez keresi a belső
// adatforrásból a vele egyező ered.Azonosító-t, majd az így kapott
// név és ered forrásváltozóból a kívánt adatokat kiválasztjuk.
        var lista = from név in hallgatók
                    join ered in eredmények on
                        név.Azonosító equals ered.Azonosító
                    select ered.Tantárgy;
        foreach (string s in lista)
            Console.WriteLine(s);
// group join
//
        var lista1 = from név in hallgatók
                    join ered in eredmények on
```

XVIII. Grafikus alkalmazások alapjai

```
név.Azonosító equals ered.Azonosító into csoport
select new {v=név.Vezetéknév,n=név.Keresztnév,cs=csoport};
Console.WriteLine("Group Join lekérdezés!");
Console.WriteLine("A csoportok száma: {0}",lista1.Count());
foreach (var s in lista1)
{
    Console.WriteLine("{0} {1} jegyei: ",s.v,s.n);
    foreach(var sl in s.cs)
        Console.WriteLine("{0} : {1}",sl.Tantárgy,sl.Jegy);
    Console.WriteLine("-----");
}
}
```

Az első lekérdezés az eredmények listából a tantárgyneveket írja ki. Az első 3 tantárgynév az 1-es azonosítóhoz tartozik stb. A második lekérdezés 4 elemből álló sorozatot ad eredményül (4 név szerepel a hallgatói listában). Minden elemben három mező van, a hallgató vezetékeve, keresztnéve és a hallgatóhoz tartozó tantárgycsoport. Ebben a tantárgycsoportban az eredménylista van, mindhárom mezőjével. A fenti példa ennek az eredményelemnek csak a tantárgy és a jegy elemét írjuk ki!

A futási eredmény az alábbi alakú:



```
C:\WINDOWS\system32\cmd.exe
Analízis
C++ nyelv
C# nyelv
C# nyelv
Bev. Prog.
Jogi furfang
Alkotmányjog
Group Join lekérdezés!
A csoportok száma: 4
Pfaff Károly jegyei:
Analízis : 3
C++ nyelv : 5
C# nyelv : 5
-----
Illés Zoltán jegyei:
C# nyelv : 5
Bev. Prog. : 3
-----
Illés Katalin jegyei:
Jogi furfang : 4
Alkotmányjog : 5
-----
Mód Károly jegyei:
-----
A folytatáshoz nyomjon meg egy billentyűt . . .
```

18.ábra

A futási képből is látható, hogy ha egy hallgatónak nincs bejegyzett eredménye, akkor üres csoportot kapunk a hallgató adatai mellé eredményül. Ha ez a csoport üres, akkor is szükségünk lehet valamilyen eredményre. Ezt a csoportürességet tesztelhetjük le a csoportra vonatkozó `DefaultIfEmpty` függvény kiterjesztéssel. Ennek paraméterül adhatunk egy alapértelmezett eredményt. Ezt a lekérdezést *left outer join* néven ismerjük. Ha a bal oldali külső forráselemhez nincs belső forráselem, akkor ezen bal oldali „külső” elemekhez (left outer) egy alapértelmezett adunk eredményül.

Lássuk a fenti forrásokra építve ezt a lekérdezési példát.

Példa:

```
var lista2 =
    from név in hallgatók
```

XVIII. Grafikus alkalmazások alapjai

```
join ered in eredmények on
név.Azonosító equals ered.Azonosító into csoport
from üres in csoport.DefaultIfEmpty(
    new Eredmény {Azonosító=0, Tantárgy="Nincs tantárgy", Jegy=0 })
select new { v = név.Vezetéknév, k=név.Keresztnév,n = üres.Tantárgy};
Console.WriteLine("Left outer Join lekérdezés!");
Console.WriteLine("A csoportok száma: {0}", lista2.Count());
foreach (var s in lista2)
{
    Console.WriteLine("{0} {1} tárgya: {2}", s.v, s.k,s.n);
    Console.WriteLine("-----");
}
```

A lekérdezés futása az alábbi eredményt adja:

19. ábra

Nyolc elemű az elemlista, hiszen minden hallgatóhoz egyenként kell venni a hozzá tartozó eredménycsoportot. Ha ilyen csoport nincs, akkor adjuk ehhez a névhez az egyelemű nincs tantárgy eredmény objektumot.

Ha a DefaultIfEmpty metódust paraméter nélkül hívjuk meg, akkor az üres forrásváltozó null referencia lesz. Ekkor nem használhatjuk közvetlenül az üres.Tantárgy nevet, mert null referenciahivatkozás kivételt dob a ciklus WriteLine utasítása, mert az s.n null érték lesz. Ezt az alábbi kiválasztás módosítással orvosolhatjuk:

```
select new { v = név.Vezetéknév, k=név.Keresztnév,n = (üres==null? "Nincs tantárgy":
üres.Tantárgy);
```

X.5. LINQ extension metódusok

Az előző részben azt láttuk, hogy a LINQ lekérdező eszköztárát hogyan használhatjuk, lekérdezések készítéséhez.

Ezen lekérdezések IEnumerable vagy IQueryable adatforráson működnek. Könyvtári függvény kiterjesztések (extension metódusok) szintén felsorolható adatforráson nyújtanak szolgáltatásokat.

XVIII. Grafikus alkalmazások alapjai

Ha figyelmesen megnézzük, akkor láthatjuk, hogy az adatsorokra nem csak könnyen kitalálható szolgáltatásokat kapunk (Max, Sum, Average, Count, ..), hanem a lekérdező kifejezések kulcsszavaihoz hasonló metódusokat is. Igen, ez azt jelenti, hogy a könyvtári szolgáltatások segítségével megvalósíthatjuk a LINQ lekérdezéseket is. Valójában a LINQ lekérdező parancsok, operátorok metódus kiterjesztéseként vannak definiálva. Tehát a LINQ kifejezések az extension metódusok egy részhalmazaként fogható fel. A nyelvi fordító valójában egy LINQ kifejezést először függvénykiterjesztés formára alakít, majd azt fordítja tovább.

A függvénykiterjesztés használata nagyon hasonlít a LINQ kifejezésekre. A from parancsnak nincs megfelelője, a kifejezés bal oldalán mindig az adatforrás áll, majd a pont operátor után a hívni kívánt metódust megadjuk. (A keretrendszer felbukkanó ablakban segít ilyenkor. Ezt láttuk már a függvénykiterjesztések tárgyalásánál is.)

A LINQ lekérdezések csak a kifejezést definiálják. Az adatok közvetlen lekérdezése, elérése csak a foreach során hajtodik végre. Ha szeretnénk az adatokat közvetlenül a kifejezés utasítással lekérdezni, akkor erre is extension metódus lehetőséget kapunk. A ToList a ToArray és a ToDictionary metódus áll rendelkezésre, hogy az eredményt egy lista vagy egy tömb szerkezetbe közvetlenül belerakjuk. Ezek a kifejezés azonnali kiértékelését elvégzik.

A következő példában nézzük ezen közvetlen kiértékelést végző függvénykiterjesztések használatára egy egyszerű példát, egyúttal az orderby lekérdezésnek megfelelő OrderBy metódushívásra is példát látunk.

Példa:

```
string[] keresztnemek = {"Zoli", "Judit", "Zsolt", "József", "Zsuzsa" };
var csökkenő = (from név in keresztnemek
                orderby név descending
                select név).ToList();
// nevek lekérdezése megtörtént, a csökkenő változó egy
// lista generic, ezért lehet az elemeit nézegetni.
Console.WriteLine("A lista utolsó neve: {0}",
    csökkenő.ElementAt(csökkenő.Count-1));
// orderby ... descending LINQ kifejezés megvalósítása
// függvénykiterjesztéssel, majd tömbbe rakjuk az eredményt
var csökkenő_1 = keresztnemek.OrderByDescending(s => s).ToArray();
// orderby eredmény Dictionary generic-ben, egyparaméteres
// esetben a kulcsot definiáljuk, ez az aktuális keresztnév Hash
// kódja lesz.
var csökkenő_2 = keresztnemek.OrderBy(s => s).ToDictionary
    (s=>s.GetHashCode());
Console.WriteLine("Az első csökkenő sorrendű név: {0}", csökkenő_1[0]);
foreach(var s in csökkenő_2)
    Console.WriteLine("Kulcs: {0} Név: {1}", s.Key,s.Value);
Console.WriteLine("A nevek csökkenő sorrendben.");
foreach (string név in csökkenő_1)
    Console.WriteLine(név);
```

A függvénykiterjesztések paraméterei között gyakran találunk delegált paraméter, ahogy az OrderBy, OrderByDescending esetén is. Ekkor legegyszerűbben egy Lambda kifejezéssel definiálhatjuk ezt a függvényt. A Lambda kifejezéssel azt adhatjuk meg, hogy a bemenő paraméterből (s, ami valójában az adatforrás forrásváltozójának tekinthető) hogyan számoljuk az eredményt. (Esetünkben változatlanul hagyjuk.)

XVIII. Grafikus alkalmazások alapjai

Könyvtári függvénykiterjesztések széles választéka áll rendelkezésünkre. Záró példaként nézzünk meg egy kicsit összetettebb formát. A lekérdezés LINQ alakját már láttuk egy korábbi példán, így könnyen érthető lesz az átírt változat is.

Példa:

```
// Ez az eredeti Group Join LINQ lekérdezés
// a hozzá tartozó adatforrásokat láttuk korábban
var lista1 = from név in hallgatók
              join ered in eredmények on
                  név.Azonosító equals ered.Azonosító into csoport
              select new {v=név.Vezetéknév,n=név.Keresztnev,cs=csoport};

//
// az előző lista1 lekérdezés metódus formában
var lista_1 = hallgatók.GroupJoin(eredmények, h => h.Azonosító,
    e => e.Azonosító, (h, csop) => new { v = h.Vezetéknév, n = h.Keresztnev, cs
= csop });
```

Ebben a metódus group join lekérdezésben, a hallgatók a külső forrás. A többi adat a metódus paramétere. Az első a belső forrás (eredmények), a második paraméter a külső forrás összehasonlító Lambda kifejezése (outer key selector), a harmadik paraméter a belső forrás kiválasztó kifejezése. Az utolsó paraméter egy két paraméteres Lambda kifejezés. Az első paraméter a külső forrás változó (egy hallgató), míg a második paraméter a belső forrásból készített csoport. Az eredmény pontosan ugyanaz az anonymous típus lesz mint az eredeti kifejezésben, vesszük a hallgató vezetéknévét, keresztnévét, ehhez hozzáadjuk az eredménycsoportot és ezek adják a típust!

Látható, ahogy korábban is említettük, hogy könyvtári függvény kiterjesztésekből bő választék áll rendelkezésünkre. Ezeknek a könyvtári szolgáltatásoknak az egyenkénti tárgyalása meghaladja ezen könyv kereteit. Remélem, hogy az eddigi példák kellő segítséget adnak a többi nem tárgyalt metódus önálló megismeréséhez.

XII. Párhuzamos programvégrehajtás, aszinkron hívások

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.

XII.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 egy paramétere lehet és a visszatérési típusa *void*. Ez több rendszerben kötelezően *run* névre hallgat.
- Ennek a függvénynek a segítségével egy függvénytípust, delegáltat definiálunk. Könyvtári szolgáltatásként a *ThreadStart* ilyen, használhatjuk ez is, ahogy a következő példa is mutatja. Egy paraméteres delegált a *ParameterizedThreadStart*.
- Az így kapott delegáltat 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()
```

XVIII. Grafikus alkalmazások alapjai

```
{
    Console.WriteLine("Ez a szálprogram!");
}
public static void szall(object o)
{
    Console.WriteLine("A kapott paraméter: {0}!",o);
}
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
    //
    Thread fonall=new Thread(program.szall);
    //a ThreadStart vagy ParameterizedThreadStart közvetlen
    //elhagyható
    fonall.Start(10);
    Console.WriteLine("Program vége!");
}
}
```

XII.2. Szálak vezérlése

Ahogy az előző példában is láthattuk, 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
- *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 szálat abortáltunk, nem tudjuk a *Start* függvénnyel újraindítani, a start utasítás *ThreadStateException* kivételt dob. Ezt a

XVIII. Grafikus alkalmazások alapjai

kivételt akár mi is elkaphatjuk, és ekkor, ha úgy látjuk, hogy a szálat 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 függvénymutatót, ami a
        // szal() függvényre mutat, ezt rövidebben is írhatnánk
        Thread fonal=new Thread(szalmutato);
        fonal.Start();
        // elindítottuk a fonalat, valójában a szal() függvényt
        Thread.Sleep(2000);
        // várunk 2 másodpercet
        Console.WriteLine("Letelt a 2 másodperc a főprogramban,
                                a szálat 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();
    }
}
```

XVIII. Grafikus alkalmazások alapjai

```
// megvárjuk a fonalbefejeződést
Console.WriteLine("Program vége!");
}
}
```

A program futása az alábbi eredményt adja:



23. ábra

Ha több szál 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.

XII.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!");
    }
}
```

XVIII. Grafikus alkalmazások alapjai

```
        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

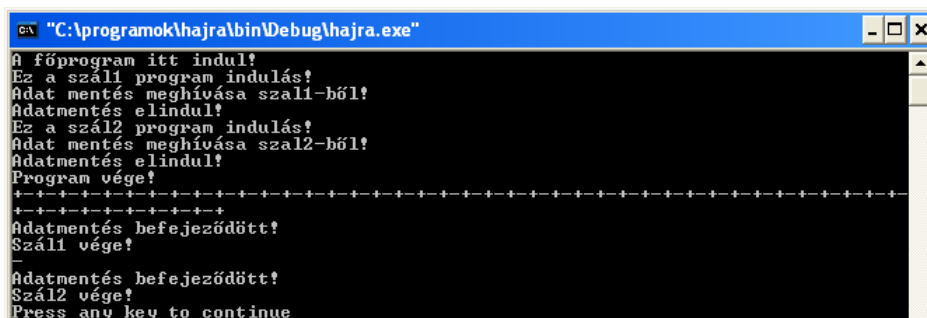
    // szál1 definiálás
    public static void szal1()
    {
        Console.WriteLine("Ez a szál1 program indulás!");
        // egy másodpercet várunk
        Console.WriteLine("Adatmentés meghívása szál1-ből!");
        a.mentes("+");
        Console.WriteLine("Szál1 vége!");
    }
    // szál1 definiálás
    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();
    //
}
```

XVIII. Grafikus alkalmazások alapjai

```
//  
Console.WriteLine("Program vége!");  
}  
}
```

A programot futtatva az alábbi eredményt kapjuk:



```
"C:\programok\hajra\bin\Debug\hajra.exe"  
A főprogram itt indul!  
Ez a szá11 program indulás!  
Adat mentés meghívása szá11-ből!  
Adatmentés elindul!  
Ez a szá12 program indulás!  
Adat mentés meghívása szá12-ből!  
Adatmentés elindul!  
Program vége!  
-----  
Adatmentés befejeződött!  
Szá11 vége!  
-----  
Adatmentés befejeződött!  
Szá12 vége!  
Press any key to continue
```

24. ábra

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 függvényfutas, 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, mikor 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
```

XVIII. Grafikus alkalmazások alapjai

```
                                // 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.

XVIII. Grafikus alkalmazások alapjai



25. ábra

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 szálat 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();          // Várunk amíg nem biztonságos  
        Console.WriteLine("Adatmentés elindul!");  
        for(int i=0;i<50;i++)  
        {  
            Thread.Sleep(1);  
        }  
    }  
}
```


XVIII. Grafikus alkalmazások alapjai

```
        Console.WriteLine(s);  
    }  
    Console.WriteLine("");  
    Console.WriteLine("Adatmentés befejeződött!");  
    m.ReleaseMutex(); // kizárást feloldjuk  
}  
}
```

Hasonló lehetőséget ad a *Semaphore* osztály is. Egy „szemafór” megengedi, hogy egyszerre több szál is hozzáférjen adott blokkhoz.

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 szálak önálló definiálása helyett, elsősorban grafikus alkalmazás környezetben, gyakran használjuk a *System.ComponentModel.BackgroundWorker* osztály szolgáltatását.

A szálakkal 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.

XV.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

XVIII. Grafikus alkalmazások alapjai

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 érjük 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 nem 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. Enter-re a *Main* program azért várakozik, mert ha engedjük befejeződni, mivel nem rendszer-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 {
```

XVIII. Grafikus alkalmazások alapjai

```
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="Dunaferr";
            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;
```

XVIII. Grafikus alkalmazások alapjai

```
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");
    }

    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
```

XVIII. Grafikus alkalmazások alapjai

```
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 );
}

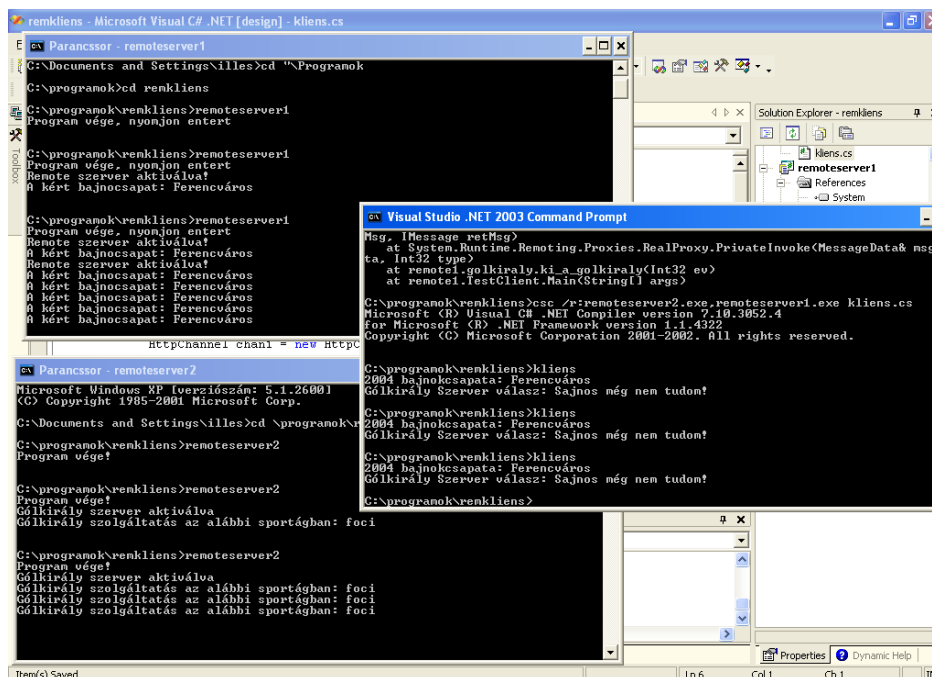
catch (Exception ioExcep) {
    Console.WriteLine("Remote IO Error" +
        "\nException:\n" + ioExcep.ToString());
    return 1;
}
return 0;
}
```

A kliens fordítása parancssorból kényelmesebb:

```
csc /r:remoteserver1.exe,remoteserver2.exe kliens.cs
```

Indítsuk el egy-egy ablakban először a szerver-, majd a kliensprogramot, ahogy a következő képen is látszik.

XVIII. Grafikus alkalmazások alapjai



26. ábra

Természetesen a szerverablakban 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. Attribútumok

Képzeljük 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 kikérhetnénk.

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 használathoz 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.

XIII.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* bázisosztályból kell származnia.

Ezek alapján a *fradi_szurkoló* attribútum definiálása a következőképpen történhet:

Példa:

```
class fradi_szurkoló:Attribute
{
    ...
}
```

XVIII. Grafikus alkalmazások alapjai

}

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)]
```


XVIII. Grafikus alkalmazások alapjai

XIII.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 reflektionak 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 lépdelünk 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...
    }
}
```

XVIII. Grafikus alkalmazások alapjai

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, 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_szurkolóAttribute
{
    private int ev;           // hány éve szurkoló
    private bool torzsszurkolo; // vajon törzsszurkoló-e
    public fradi_szurkoló(int e) // az e rendes, pozicionális
    {                           // paraméter
        ev=e;
    }
}
```

XVIII. Grafikus alkalmazások alapjai

```
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;
    }
}
```

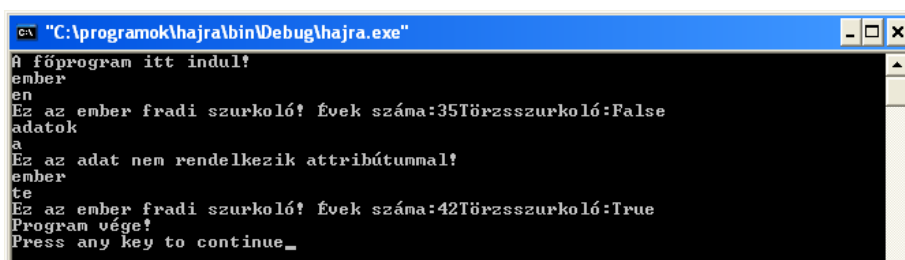
```
class adatok{} //nem fontos, hogy érdemi információt tartalmazzon
```

```
class program
{
    // attribútum beállítása
    [fradi_szurkoló(35)]
    public ember en= new ember("Zoli");
    // a normál konstruktort hívtuk meg, azok a mezők, amiket nem
    // inicializál, alapértelmés 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
```

XVIII. Grafikus alkalmazások alapjai

```
//
public adatok a=new adatok();
//
[fradi_szurkoló(42,Törzsszurkoló=true)]
public ember te= new ember("Pali");
// Pali már törzsszurkoló
public static void Main()
{
    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:



26. ábra

XIII.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 almafuggveny()  
{  
    Console.WriteLine("Alma volt!");  
}  
  
#define alma  
alfamfuggveny();           // hívás rendben  
  
#undefine alma  
alfamfuggveny();           // hívás elmarad
```

XVIII. Grafikus alkalmazások alapjai

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.

XVII. 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.

XVII.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.

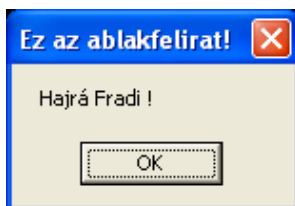
Ezek után nézzük meg példaként a *MessageBox* API függvény használatát.

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);
    }
}
```

A program futtatása után az alábbi rendszer-üzenetablak jelenik meg:



37. ábra

XVIII. Grafikus alkalmazások alapjai

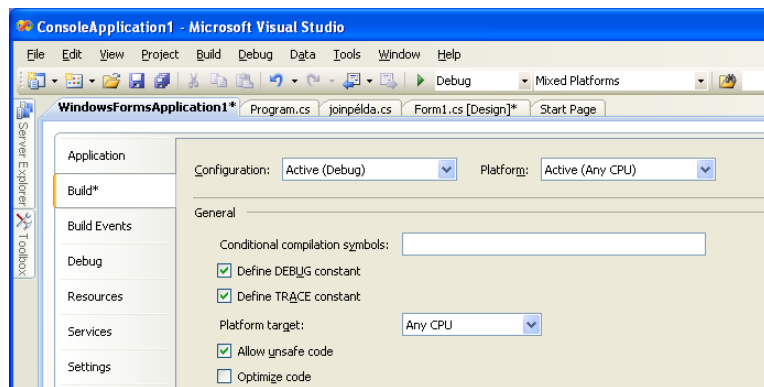
XVII.2. Mutatók használata

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 menedzselt, 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 menedzselt adatot *fixed* jelzővel tudunk ellátni, ha azt akarjuk, hogy a GC által felügyelt területből egy típushoz (menedzselt 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 ablak *Build* lapján be kell állítani az „*Allow Unsafe Code*” opciót.



38. ábra

Ezt a beállítást parancssori környezetben a *csc* fordítónak az */unsafe* kapcsolója használatával érhetjük el.

Ezek után nézzünk egy klasszikus C++ nyelv szerű 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* egész mutató
    {
        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};
```


XVIII. Grafikus alkalmazások alapjai

```
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);
}
```

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, 2007
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: Bemutatózik 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
10. Paolo Pialorsi, Marco Russo: Introducing Microsoft LINQ
MS Press, 2007