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

VII.2.2. Privát konstruktor

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

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

Példa:

```
using System;

class nincs_peldanya
{
    public static int x=4;

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

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



```
class prog
{
    public
    {
        /
        C
        /
        n
    }
}
```

VII.2.3. Saját de

Ahogy egy o
kezdeti lépésekre,
bizonyos lépéseke
magának memóriá
vényt, amelyik
destruktornak vag
tály konstruktorán
az osztály nevéve
destruktor meghí
használjuk, a kere
visszaadja az oper

Konstruktorna
tornak mindig pub

Ha egy osztál
rendszer automat
illetve destruktort
nem 'készül' auto

A destruktor
algoritmusnak ne
nal, az objektum
hanem akkor, ami
nem írunk saját d
objektum által má
ált destruktor füg
kényszer saját des
Finalize függvény

Ez azért lehet
vényt definiál, ak
szerint:

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

VII.2.3. Saját destruktor

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 destruktor függvénynek nevezzük. Már tudjuk, hogy az osztály konstruktorának neve az osztály nevével egyezik meg. A destruktor neve is az osztály nevével azonos, csak az elején ki kell egészíteni a ~ karakterrel. A destruktor meghívása 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 destruktor automatikus meghívásának a folyamatát szemétgyűjtési algoritmusnak nevezzük (garbage collection, GC). Ez az algoritmus nem azonnal, az objektum blokkjának, élettartamának a végén hívja meg a destruktort, hanem akkor, amikor az algoritmus „begyűjti" ezt a szabad memóriaterületet. Ha nem írunk saját destruktor függvényt, akkor is a garbage collector minden, az objektum által már nem használt memóriát felszabadít az automatikusan definiált destruktor függvény segítségével. Ez azt jelenti, hogy nincs túlzottan nagy kényszer saját destruktor definiálására. A garbage collector valójában az osztály *Finalize* függvényét hívja meg.

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

Példa:

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

A fenti osztálydestruktorból a fordító az alábbi kódot generálja:

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

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

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

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

Kezdeményezzük a keretrendszer szemétgyűjtő algoritmusának indítását:

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

A hívó végrehajtási szál addig várakozik, amíg a destruktorok 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.SupressFinalize(Object o);
```


Ha egy objektumnak
függvényhívásra, akkor a

```
void System.GC.Re
```

Máris feliratkozunk
többször hívjuk meg, akk

Ahogy korábban em
meghívni, a destruktort
mégis szükségünk lenne
akkor a *Dispose* függvé
logikai változó mutatja,
Dispose metódusát. Ezer

```
bool disposed=fa
...
protected overri
{
    if( !dispos
    {
        if (di
        {
            /
        }
        this.d
        // ha
        base.D
        // err
        //hívr
        GC.Sup
    }
}
```

Itt kell szót ejteni
(lásd a *Nyelvi utasítás*

A destrukció foly
nyújt segítséget. Ennel
a mai számítógépek v
között szerepel, azér
memóriahiányban sze
szemétgyűjtési algor

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

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

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

Ahogy korábban említettük a destruktorral kapcsolatosan, mi nem tudjuk meghívni, a destruktort a keretrendszer szemétgyűjtő algoritmusa 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 override void Dispose( bool disposing )
{
    if( !disposed )
    {
        if (disposing)
        {
            // ide jön az erőforrás felszabadító kód
        }
        this.disposed=true;    // nem kell több hívás
        // ha van ősosztály, akkor annak dispose hívása
        base.Dispose( disposing );
        // erre az objektumra már nem kell finalize függvényt
        //hívni a keretrendszernek
        GC.SupressFinalize(this);
    }
}
```

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

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

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

Példa:

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

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

```
            // NEM DEFINIÁLUNK DESTRUKTORT
            // MERT AZ AUTOMATIKUS SZEMÉTGYŰJTÉSI
            // 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 erdeményként az *alma* szót látjuk.

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

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

VII.3. Konstans, csak olvasható mezők

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

VII.3.1 Konstans mezők

Ahogy korábban említettük, egy változó módosíthatóságát a *const* 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.

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 olvas
{
    public
    {
        cs
        Co
        cs
        Co
        p
    }
}
```

Természetesen az esetben a mez hetjük el.

VII.4. Tulajd

Egy osztály adattagjaihoz mil Hagyományos ob kaptunk, maradta függvényneveket.

VII.4.1. Tulajdo

A C# nyelv a egyszerű és kény speciális függvén sem (), és a függ egyszerű értékad hivatkozhatunk a

Példa:

```
using Syst
class Embe
{
    priva
    priva
    // a
    publi
    {
        s
    }
```

```
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 adattagjaihoz milyen módosítási lehetőségeket engedélyezzünk, biztosítsunk. Hagyományos objektumorientált nyelvekben ehhez semmilyen segítséget nem kaptunk, maradtak az általános függvénydefiniálási lehetőségeink. Ezeket a függvényneveket jellemzően a set illetve a get előtagokkal módosították.

VII.4.1. Tulajdonság, property függvény

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

Példa:

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

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

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

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

VII.4.2. Index függvény (indexer)

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

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

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

Lássuk az alábbi példát, ami a jellemző használatot mutatja.

Példa:

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



```
    ...
    public ir
    {
        get
        {

        }
        set
        {

        }
    }
}
class prograr
{
    static v
    {
        val
        a[2
        Con
    }
}
```

Az indexer esetc
feltétlenül kell minc
akkor csak olvasha
függvényről beszélü

Az indexerhasz
néhány olyan külön

- Az *indexer* f
  paramétere

Példa:

```
public int i
{
    get
    {
        re
    }
    set
    {
        x=
    }
}
```

- az *indexer* fü
  A fenti *inde*

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

Az indexer esetében is, hasonlóan a tulajdonságdefinícó 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 indexerhasználat kísértetiesen hasonlít a vektorhasználathoz, de van néhány olyan különbség, amit érdemes megemlíteni.

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

Példa:

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

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

Példa:

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

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

VII.5. Osztályok függvényparaméterként

Egy függvény 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 in
    {
        a.X=5
        retu
    }

    static vo
    {
        péld
        Cons
    }
}
```

A program futtat

```
Konstruktorhí
25
```

Mikor egy függ
függvényparaméter
eredeti osztályváltoz
Teljesen hasonl
visszatérési értékkér

VII.6. Operáto

A már korább
használhatók. Oszt
automatikusan hasz
értékadást az új osz
definiálni nem lehe
Hasonlóan nem
definiálási lehetősé
A legtöbb oper
alábbi operátorok d

Egyoperandusú
Kétoperandusú

A fenti hagy
konverziós operátc

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

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

A program futtatása a következőket írja a képernyőre:

```
Konstruktorhívás!
25
```

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

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

VII.6. Operátorok újradefiniálása

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

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

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

Egyoperandusú operátorok: +, -, !, ~, ++, --, true, false
Kétoperandusú operátorok: +, -, *, /, %, &, |, ^, <<, >>, <=, >=, ==, !=, <, >

A fenti hagyományosnak mondható operátorkészlet mellett még típus-konverziós 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 *(+)* operátorát 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 kétoperandusú operátor esetén két paramétere van a függvénynek.

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

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

Példa:

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

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

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

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

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

Az eredmény a következő lesz:

```
Komplex szám példa.
Összeadás eredménye: A keresett szám:7:2
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Az explicit verzió meghívása a következőképpen történik:

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

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

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

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

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

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

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

```
if (a) utasítás;
```

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

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

Definiáljuk újra ezeket az operátorokat a már megismert komplex osztályunkhoz:

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

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

Példa:

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

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

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

==, !=	azonosság, különbözőség megadása
<, >	kisebb, nagyobb
<=, >=	kisebb vagy egyenlő, nagyobb vagy egyenlő

VII.7. Interface definiálása

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

VII.7.1. Interface fogalma

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

Az interfacedefiníció formája:

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

Példa:

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

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

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

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

Példa:

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

VII.7.2. Interface használata

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

Példa:

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

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

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

VII.7.3. Interface-ek kompozíciója

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

Példa:

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

public interface szállítható
{
    bool szállít();
}

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

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

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

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

Példa:

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

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

VII.8. Osztályok öröklődése

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

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

Hasonlóan az osztályok mezőhozzáférési rendszeréhez, több nyelvben is lehetőség van arra, hogy öröklés esetén az utódosztály az ősosztály 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 ősosztálybeli jelentését.

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

Egy őstípusú referencia bármely utódtípusra hivatkozhat.

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

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

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

class utód: ős
{
    ...
};
```

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

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

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

```
class utód: ős, interface1, ...  {
                                      //
                                  };
```

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

Paraméteres konstruktorok esetén az utódkonstruktor alakja:

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

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

Ezek után nézzük a fentieket egy példán keresztül.

Példa:

```
using System;

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

}
```

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

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

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

1. példa:

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

2. példa:

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

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

kt mezője, de
absztrakt

yból nem

VII.10. Virtuális tagfüggvények, függvényelfedés

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

VII.10.1. Virtuális függvények

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

Példa:

```
using System;

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

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

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

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

A *p=k* értékadás helyes, hiszen *p* (*pont* típus) típusa a *k* típusának *(kor)* az őse. Azt is szoktuk mondani, hogy egy őstípusú referencia tetszőleges utódtípusra hivatkozhat. Ekkor a második *p.kiir()* utasítás is, a *p=k* értékadástól függetlenül a *2,1* értékeket írja ki! Miért? Mert az osztályreferenciák alapértelmezésben statikus hivatkozásokat tartalmaznak a saját függvényeikre. Mivel a 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 szeretnéi
hívja meg, akkor az
függvényét felüldefir

Példa:

```
using Syste
class pont
{
    private
    public
    {
        x=
    }
    virtua
    {
        Cc
        Cc
    }
}
class kor:p
{
    privat
    public
    {
        r=
    }
    overri
    {
        C
    }
}
public cla
{
    public
    {
        k
        p
        p
        k
        p
        p
    }
}
```

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;
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
            // pedig az eredeti típusát kell figyelembe venni.
    }
}
```

adatai *(2,1)*, majd a
)bi két sorral:

hivatkozik

ı *k* típusának *(kor)* az
:ia tetszőleges utódtí-
, a *p=k* értékadástól
ztályreferenciák alap-
függvényeikre. Mivel
ben a program során
/ kör objektumot azo-
indig a fixen, fordítási

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

Gyakran előfordul, hogy az ősosztályban nincs szükségünk egy függvényre, míg a származtatott osztályokban már igen, és szeretnénk, ha virtuálisként tudnánk definiálni. Ebben az esetben az ősosztályban 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);
    }
}
public class MainClass
{
        public static void Main()
    {
        // pont p=new pont(); utasításhibát eredményezne
        kor k=new kor();
        k.mozgat(3,4);
    }
}
```


VII.10.2. Függvé

Az öröklés k
függvényére egyál
focicsapat osztály
tálynak is van ily
nevkiir függvénye

Példa:

```
class foci
{
    protec
    public
    {
        cs
    }
    public
    {
        C
    }
}
class kedv
{
    public
    {
        c
    }
    new pu
    {
        C
    }
}
public cla
{
    publi
    {
        }
        (
    }
}
```

A *new* utasí
definiált *nevkiir* f
Ennek az ell
ősosztály függvé
kor a függvényt,
kell megjelölni.

VII.10.2. Függvényeltakarás, sealed függvény

Az öröklés kapcsán az is előfordulhat, hogy a bázisosztály egy adott függvényére egyáltalán nincs szükség az utódosztályban. Az alábbi példában a focicsapat osztálynak van *nevkiir* függvénye. Az utód *kedvenc_focicsapat* osztálynak is van ilyen függvénye. A *new* hatására a *kedvenc_focicsapat* osztály *nevkiir* függvénye eltakarja az ősosztály 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";
    }
    new public void nevkiir()
    {
        Console.WriteLine("Kedvenc csapatunk a: {0}",csapat);
    }
}
public class MainClass
{
    public static void Main()
    {
        kedvenc_csapat cs=new kedvenc_csapat();
        cs.nevkiir();
    }
}
```

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

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

VII.11. Feladatok

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

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

VIII. Kiv

Egy program, p
nyesség tekintetében

- A függvény v
  nesség" nem
- A függvény v
  rekre vonatk
  ményekénti
- A függvény v
  jelenség lép

Ezen harmadik
sére nyújt lehetőség
második esetet teki
valamilyen „nem h
hozni, hogy hiba tö
dául egy egész ért
olyan egész értéket
Így ebben az esetbe
kezelés lehetősége

A kivételkeze
hibakeresés, hibav
lehet biztosítani. A
az ún. aszinkron ki
vermegszakítás *(in*

Ehhez hasonló
ERROR GOTO, C
Object Pascal nye

VIII.1. Kivét

A kivételkeze
nek felhasználásra

knak? Milyen kon-

nterface között?

kkal! (Bútor neve,
gokhoz készítse el a

ı az összeszerelési
ı *lapraszerelt bútor-*
biztosítva azt, hogy
tása.

VIII. Kivételkezelés

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

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

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

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

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

VIII.1. Kivételkezelés használata

A kivételkezelés implementálásához a következő új nyelvi alapszavak kerülnek felhasználásra:

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

A kivételkezelés használata a következő formában adható meg :

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

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

A C++ nyelv kivételkezelő lehetősége megengedi azt, hogy a *catch* blokk tetszőleges hibatípust kapjon el, míg a C# ezt kicsit korlátozza. A C# nyelvben a *catch* blokk típusa csak a keretrendszer által biztosított *Exception* osztálytípus, vagy ennek egy utódtípusa 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 (
{
    Co
}
finally
{
    Co
}
...
```

A fenti példáb
dául a nullával val
amit elkapunk a *ca*

A példa egész
használják az egé
módosítókat is. Ez

Ha az egész ar
maz, akkor a *chec*

Példa:

```
int i=
checke
{
    i
}
...
int j=
unchec
{
    j
}
Conso
```

Természetese
tásvégrehajtást n
vezérlést, hanem
mikor látja szüks

Példa:

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

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

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

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

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

Példa:

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

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

Példa:

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

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

A kivételkezelések egymásba ágyazhatók.

Többféle abnormális jelenség miatt lehet szükség kivételkezelésre, ekkor az egyik „kivételkezelő" a másiknak adhatja át a kezelés lehetőségét, mondván „ez már nem az én asztalom, menjen a következő szobába, hátha ott a címzett" *(throw)*. Ekkor nincs semmilyen paramétere a *throw*-nak. Ez az eredeti 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 *Indexhil*
egy őstípus egyb
blokkokat az örök

Példa:

```
using Syst
public cla
{
    publi
    {
    }
    publi
    {
    }
}
public cla
{
    publi
    {
    }
    publi
    {
    }
}
    ...
    int
    int
    try
    {

    }
    catch
    {

    }
    catch
    {

    }
    catch
    {

    }
    finally
    {

    }
```

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

Példa:

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

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

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

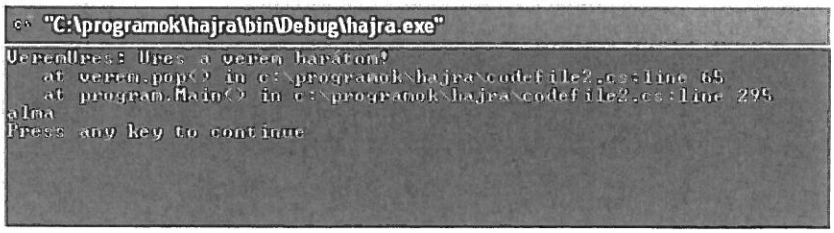
Befejezésül nézzük meg a korábban látott verem példa megvalósítását kivételkezeléssel kiegészítve.

Példa:

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

    }
    // NEM DEFINIÁLUNK DESTRUKTORT,
    // MERT AZ AUTOMATIKUS SZEMÉTGYŰJTÉSI
    // ALGORITMUS FELSZABADÍTJA A MEMÓRIÁT
    public void push(Object i)
    {
        if (mut<x.Length)
        {
            x[mut++]=i;         // beillesztettük az elemet
        }
        else throw new VeremTele("Ez bizony tele van!");

    }
    public Object pop()
    {
        if (mut>0) return x[--mut];
            // ha van elem, akkor visszaadjuk a tetejéről
        else throw new 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
            {
                Console.WriteLine(ve);
            }
            v.push(5);
            v.push("alma");
            Console.WriteLine(v.pop());
        }
    }
```

A program futása az alábbi eredményt adja:



11. ábra

VIII.3. Feladatok

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

IX. Inpu

A be- és kivit
szabványos könyvt
a kapcsolatot. Ez a
sajátossága. A C+-
output műveleteket

IX.1. Standar

Minden klassz
nálhatjuk a *System*
billentyűzet), kiírá
képernyő) művelet
donságaiban vanna
TextReader, míg
TextWriter a karak
ezen tulajdonságok
Mielőtt rövide
hogy Windows alk
grafikus felületű e
A *Console* osz
Kiírás:

```
Write(...);
WriteLine(.
```

Mindkét utas
Write(double) stb.
változata is:

```
Write( str
WriteLine(
```

Ez a változat
paraméter, mint e
paraméterek behe
formázási lehetősé
Ezen formázá
zás alakja a követ

juk használni?

ıolja ki a gyököket,
:erkönyvtár kivétel-

az előző feladatot

IX. Input-Output

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

IX.1. Standard input-output

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

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

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

Kiírás:

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

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

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

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

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

{sorszám[, szélesség][:kijelzési_forma]}

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

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

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

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

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

Példa:

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

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

Beolvasás:

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

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

Példa:

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

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


Példa:

```
string s=C
```

Ha nem sikerü
kivételt dob a *Rea*

A könyvtár ne
zésre állnak a *Con*

Példa:

```
string s=C
int j=Conv
int i=Conv
//konvertá
Console.Wr
```

Ha nem sikeri
telt dob a *Conve*
beolvasására, az a

Nézzük meg
téssel, hogy a *kor*

Példa:

```
using Syst
struct str
{
    publi
    publi
}

class str
{
    publi
    {
```

Példa:

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

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

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

Példa:

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

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

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

Példa:

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

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

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

IX.2. Fájl input – output

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

A könyvtári osztályok jellemző műveletei, tulajdonságai:

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

IX.2.1. Bináris f

A bináris mű

- BinaryRea
- BinaryWri

Mindkét osztá
objektumot kapni,
FileStream osztály
Egy *FileStre*
szoktak megadni

- a fájl nevé
- fájlmód pa
  - FileM
  - FileM
  - FileM
- fájlelérést
  - FileA
  - FileA
  - FileA
- megosztá
  - FileS
  - FileS
  - FileS
  - FileS

Természete
részletezését az
Bináris áll
(Seek), ezzel eg
Egy fájlren
osztályai is leh
Fileinfo osztály

A legfonto

- FileStrea
- FileStrea
- StreamV
- StreamR
- File.Exi

IX.2.1. Bináris fájl input-output

A bináris műveletek két alaposztálya:

- BinaryReader olvasásra,
- BinaryWriter írásra.

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

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

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

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

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

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

A legfontosabb *File* statikus függvények:

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

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

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

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

A legfontosabb *BinaryReader* függvények:

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

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

Ezek után nézzünk egy rövid példát:

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

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

IX.2.2. Szöveges fájl input-output

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

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

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

A legfontosabb *StreamReader* függvények:

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

Példa:

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

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

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

IX.3. Feladatok

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

X. Párhu

A feladatok g
években speciális
hajtás. Gondolha
például, hogy mi
szerkesztőt haszn
dásunkat serkente
ségével.

A párhuzamos
operációs rendszer

Például egy a
a másik szál pedig

A .NET kere
arra, hogy egy pr
tási szálakról az i
olvashatunk.

X.1. Szálak d

Általában el
lépéseket kell m
jellemzők):

- Definiálun
  résí típus
- Ennek a
  definiálu
- Az így kaj
- Meghívjul

A párhuzam
System.Threadin

Ezek után a

X. Párhuzamos programvégrehajtás

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

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

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

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

X.1. Szálak definiálása

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

- Definiálunk egy függvényt, aminek nincsenek paraméterei és a visszatérési típusa *void*. Ez több rendszerben kötelezően run névre hallgat.
- Ennek a függvénynek a segítségével egy függvénytípust, delegáltat definiálunk.
- 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()
    {
        Console.WriteLine("Ez a szálprogram!");
    }
    public static void Main()
    {
        Console.WriteLine("A főprogram itt indul!");
        ThreadStart szalmutato=new ThreadStart(program.szal);
        // létrehoztuk a fonal függvénymutatót, ami a
        // szal() függvényre mutat
        Thread fonal=new Thread(szalmutato);
        // létrehoztuk a párhuzamos végrehajtást végző objektumot
        // paraméterül kapta azt a delegáltat (függvényt)amit
        // majd végre kell hajtani
        fonal.Start();
        // elindítottuk a fonalat, valójában a szal() függvényt
        // a fonal végrehajtása akkor fejeződik be amikor
        // a szal függvényhívás befejeződik
        Console.WriteLine("Program vége!");
    }
}
```

X.2. Szálak vezérlése

Ahogy az előző példában is 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 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

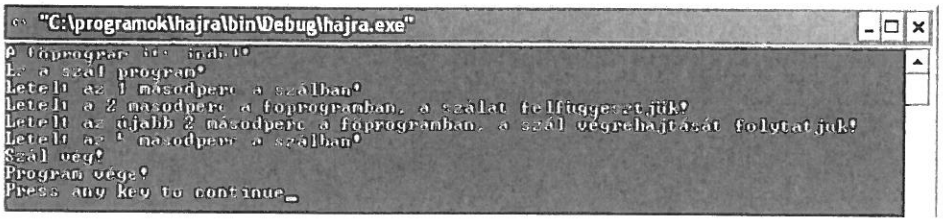
Ezek után nézzük meg a fenti programunk módosítását, illusztrálva ezen függvények használatát.

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

A program futása az alábbi eredményt adja:



12. ábra

Ha több szálat 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.