

# Középhaladó

A neheze csak most jön >:D

## 10. Struktúrák és Enumok

### Struktúrák

C#-ban egy **struktúra** adott változótípusok csoportosítása, így új (komplex) adattípus jön létre.

Létrehozását a `struct` szóval kezdjük.

pl:

```
struct Könyv                                // elnevezzük 'Könyv'-nek a típusunkat
{
    public string cím;                       // (string) Könyv.cím
    public int oldalSzám;                   // (int) Könyv.oldalSzám
    public string[] lapok;                  // (string[]) Könyv.lapok
    public void legyen(string cím, int oldalSzám, string[] lapok) // saját eljárások
    {
        this.cím = cím;
        this.oldalSzám = oldalSzám;
        this.lapok = lapok;
    }
}
```

A struktúrák más nyelvekben eléggé primitívek, C#-ban viszont lehet **eljárásokat**,

**konstruktorokat** (de nem destruktorokat!), **tulajdonságokat**, **operátor eljárásokat**, etc... fűzni hozzá.

Mi a különbség a `struct` ok és osztályok között?

- *osztályok* referens típusok; *struktúrák* **érték** típusok
- struktúráknál **nem kell** a `new` kulcsszó
- struktúrák **nem támogatják** az öröklést

- struktoknak **nem lehet** alap konstruktorjuk

## Enumok

Egy **enumerációnak** (felsorolásnak) lényege az egyszerű számozások elnevezése.

Kifejezetten hasznos például fájlok megnyitási módjának kiválasztásakor

(sokkal könnyebb megérteni a `FileAccess.Write` nevet, mint megjegyezni a '2' számot),

de használatuk feladatonként eltér.

```
enum Irány                // a számsor neve 'Irány'
{
    Fel, Jobbra, Le, Balra // az alap számsor 0-tól kezdődik és 1-el növekszik
                          // Irány.Fel = 0; Irány.Jobbra = 1; Irány.Le = 2 ; Irány.Balra = 3
}
```

Az enumok *külön adattípusok*, így nem tudnak örökölni típust, és

a fordítóprogram nem fogja érteni az *implicit* (operátorok nélküli *sima*) típusváltást,

ezért fontos, hogy **típuskényszerítsünk**.

```
System.Console.WriteLine( (int)Irány.Fel ); // kimenet: 0
System.Console.WriteLine( (int)Irány.Le );  // kimenet: 2
```

```
In [ ]: struct Könyv                // elnevezzük 'Könyv'-nek a típusunkat
{
    public string cím;              // (string) Könyv.cím
    public int oldalSzám;           // (int) Könyv.oldalSzám
    public string[] lapok;          // (string[]) Könyv.lapok
    public IrodalmiMűnem műnem;     // (IrodalmiMűnem) Könyv.műnem
    public void legyen(string cím, int oldalSzám, string[] lapok) // saját eljárások
    {
        this.cím = cím;
        this.oldalSzám = oldalSzám;
        this.lapok = lapok;
    }
    public void legyen(string cím, int oldalSzám, IrodalmiMűnem műnem) // túltöltés
    {
        this.cím = cím;
        this.oldalSzám = oldalSzám;
        this.műnem = műnem;
    }
}
```

```

    }
}

enum IrodalmiMűnem
{Epikus = 1, Lírai, Drámai}           // Lehet kezdőszámot, vagy akár külön az összeset beállítani

Könyv k1;                             // csinálunk egy Könyv típusú változót (nem kell a 'new' kulcsszó!)
k1.cím = "A Gyűrűk Ura";               // a k1 változónkon belül a címet módosítjuk
k1.oldalSzám = 655;
k1.műnem = IrodalmiMűnem.Epikus;       // enum előhívás

Könyv k2;
k2.legyen("Háry Péter", 6969, new string[] {"A gyerek aki élt.", "Vége."});

Könyv k3;
k3.legyen("Édes Anna", 300, IrodalmiMűnem.Epikus);

System.Console.WriteLine("k1 címe: {0}", k1.cím); // előhívás
System.Console.WriteLine("k2: {0} {1}", k2.cím, k2.oldalSzám); // előhívás
foreach (var elem in k2.lapok)
{
    System.Console.WriteLine(elem);
}
System.Console.WriteLine(
    "k3 infó: {0}; {1}; {2}; {3}", k3.cím, k3.oldalSzám, k3.műnem, (int)k3.műnem
    // a műnem nevét kapjuk ha implicit alakítunk enumot stringbe!
);

```

## 11. Névterek, Osztályok, Tárgyak

### Alapok

Ahogy az alap típusok tudtak elengedhetetlen adatokat tárolni, úgy a magasabb szintű feladatokhoz lehet saját típusokat megadni.

Egy OOP nyelvben az **osztály** egy adattípus ami csoportosít egyéb változókat és eljárásokat az osztályhoz vagy **tárgyaihoz**.

Például egy *csavarhúzó* tárgynak vannak adatai és eljárásai:

- fej típusa (csillag, lapos, torx, imbusz, etc.)

- fej mérete (M16, M10, M22, etc.)
- becsavarás, kicsavarás

Ha a csavarhúzó *tervét*, **osztályát** akarjuk meghatározni, használjunk **class** -t.

A konstansokat és változókat **tulajdonságoknak** nevezzük.

Az osztályon belüli eljárásokat **metódusoknak** nevezzük.

Az osztály részeit együttesen **tagoknak** nevezzük.

Miután deklaráltuk az osztályunkat, előhívhatunk egy *példányt* annak mintájára (**instanciálás**).

// a main()-ben:

```
Osztály pdny = new Osztály();
```

```
pdny.Használ();
```

```
In [ ]: class Csavarhúzó           // CSAK MÁS osztályokon és eljárásokon kívül!!!
{
    public int iFejMéret;         // tulajdonságok
    string fejTípus;

    public void Használ()        // osztályon belüli eljárás = metódus
    {
        Console.WriteLine("> Használtad a(z) {0}mm méretű csavarhúzót.", this.iFejMéret);
    }
}

Csavarhúzó cs1 = new Csavarhúzó();
cs1.iFejMéret = 16;
cs1.Használ();
```

**public** és **private**

A **public** kulcsszó megengedi az osztályon *kívülieknek*, hogy egy tárgynak azt a *tulajdonságát* szabadon elérje.

A **private** ennek az ellentettje, a tulajdonságot csak az osztályon *belüli eljárások* vehetik igénybe.

Van még egy: a **protected**, ami kicsit később lesz fontos.

```
cs1.iFejMéret = 16;
cs1.nyélTípus = "gumírozott";
Console.WriteLine(cs1.nyélTípus);
```

```
In [ ]: class Ember
{
    public int iÉletkor;
    string sNév;
    public void Köszön()
    {
        Console.WriteLine("Szia!");
    }
}

Ember cs1 = new Ember();
cs1.Köszön();
cs1.iÉletkor = 20;
Console.WriteLine(cs1.iÉletkor);
```

## Memória

Már átbeszéltük a paraméterek átadását C#-ban, így az osztályokkal könnyebb dolgunk van.

Default esetben *értékként* adja tovább a paramétereket a C#.

Ez azért van, mert a beépített alap típusok *érték-típusok*. Ezek a **stack**-be kerülnek.

A **stack** egy *memória-részleg* ahova az értékek kerülnek, és a változók az értékekre **mutatnak**.

```
int x = 100;           //           x      bé
char bé = 'b';        //stack:  ...[ ] [ ] [100] [ ] ['b'] [ ]...
```

Ha tárgyakat csinálunk akkor *mint referencia* kapjuk meg őket.

A tárgy változó a **stack**-ben lesz, viszont a tárgy adatai egy rendezetlen helyre,

a **heap**-be kerülnek.

A változóban ekkor egy *referencia* van, egy *cím* a **heap**-beli tárgyhöz.

```
int x = 100;           //           x      e1
// stack:  ...[100] [ ] [ ] [0x052f1] [ ]...
```

```
Ember e1 = new Ember();           //           ↓ rámutat!  
                                   // heap:    ...   |Ember tárgy| ...
```

A `stack` -et *statikus* memóriaként használjuk (a méret **előre megadva**),

A `heap` -et pedig *dinamikus*ként, mivel az egyéni példányok lehet **más memóriát** foglalnak idővel.

## Konstruktor

Hogyha új példányt csinálunk egy tárgyból, a kezdeti paramétereket szeretnénk beállítani, esetleg start-funkciókat elindítani.

Erre való az osztály **építőeljárása** (*konstruktorja*). A *konstruktor* minden példány **készítésekor** lefut.

A konstruktor neve **meg egyezik az osztály nevével**, és **nincs visszértéke**,

ráadásul **mindig** `public` kulcsszót kap, hogy máshonnan is lehessen hívni (pl. egy deklarálásnál).

Fontos: a konstruktort is túl lehet tölteni.

```
class Személy  
{  
    private int Kor;           // privát változó, biztonság érdekében  
  
    public Személy(int x)  
    {  
        Kor = x; /* itt a konstr. paramétereit bevisszük egy példányba*/  
        Console.WriteLine("Szia! Én {0} éves vagyok!", Kor);  
    }  
}
```

Így ha egy új Személyt hívunk, így fog kinézni:

```
Személy sz1 = new Személy(16);    // példányosít -> lefuttatja a konstruktort
```

```
In [ ]: class Személy  
        {  
            private int kor;  
            private string név;  
  
            public Személy(int x)  
            {
```

```

        kor = x;
        Console.WriteLine("Szia! Én {0} éves vagyok!", kor);
    }

    public Személy(string s, int x)
    {
        név = s;
        kor = x;
        Console.WriteLine("Szia! {0} vagyok és {1} éves!", név, kor);
    }

    public void setNév(string s) {név=s;}
    public void setKor(int a) {kor=a;}
    public string getNév() {return név;}
    public int getKor() {return kor;}
}

```

```

Személy s1 = new Személy(16);
Személy s2 = new Személy("Péter", 42);

```

```

s1.setNév("Anna");

```

```

Console.WriteLine($"s1: {s1.getKor()} {s1.getNév()}");
Console.WriteLine($"s2: {s2.getKor()} {s2.getNév()}");

```

## Destruktor

Az *építőeljárás* ellentettje a **lebontó** (*destruktor*).

Amikor egy példány nem kell többé, memóriefelszabadítás érdekében **kitöröljük**, ekkor kapcsol be a *destruktor*.

Jele: `~`.

A *destruktor* neve **megegyezik az osztály nevével**, csak **EGY** lehet belőle, nem lehet **semmilyen kulcsszó** rajta, és **nem lehet paramétere**.

```

class Asztal
{
    public Asztal()
    {

```

```

        //konstruktor
    }

    ~Asztal()
    {
        //destruktor
    }
}

```

### Enkapszuláció és hozzáférés

Az adatok az osztályok belsejében eléggé *fontosak* lehetnek, és a kontrollált hozzáférés hiánya nagy biztonsági rés. Egy rosszakaró lehet ki tudja aknázni a szabad elérés meglétét.

Ez ellen - a biztonságos kezelés érdekében - **becsomagolást** (enkapszulációt) használunk, amely védi a nyers adatokat.

C#-ban ezek a kulcsszavak szabják meg az hozzáférést:

`public`, `private`, `protected`, `internal` és `protected internal`

A következő részletben a belső változót közvetlenül nem lehet elérni, csakis a társ-funkciókkal.

```

In [ ]: class BankSzámla
{
    private double egyenleg = 0;        // ez privát, tehát "BankSzámla.egyenleg = 1000;" HIBA lenne
    public void Betétel(double x)      // +
    {
        egyenleg += x;
    }
    public void Kivétel(double x)      // -
    {
        egyenleg -= x;
    }
    public double Lekérdez()           // kikéri az egyenleget
    {
        return egyenleg;
    }
}

```



```
}  
}
```

## Getterek és Setterek

Ha **EGY** taggal akarunk *privát* változót **lekérni** és/vagy **változtatni** *közvetlen hozzárendelés* nélkül, **tulajdonságokat** (*property-ket*) használunk.

A **get** és **set** rész eljárások, amik feldolgozzák az áthaladó adatot.

A *property*-knek a fordító **implicit** módon (*kontextustól függően*) ad értéket.

Tehát ha *értéket váró* eljárásba rakjuk, a **get** -et használja,

ha pedig egy *értéket adunk* ennek, akkor a **set** -et.

Syntax:

```
class Osztály  
{  
    public int aProp  
    {  
        get { /*Lekérés*/ }  
        set { /*bevitel*/ }  
    }  
  
    public char bProp { get; set; }    //gyors szintaxis, get->return;set->hozzárendelés;  
}
```

```
In [ ]: class Személy  
        {  
            private int kor;  
            private string név;  
  
            public int Kor  
            {  
                get {return kor;}           // getter  
                set                           // setter  
                {  
                    if(value < 99 && value > 0) // bármilyen logika használható  
                        kor = value;          // pl. bevitel tisztítására  
                }  
            }  
        }
```

```

        else kor = -1;
    }
}
public string Név { get; set; }    // gyors syntax
}

```

```
Személy s1 = new Személy();
```

```
s1.Név = "Anna";
s1.Kor = 100;
```

```
Console.WriteLine($"s1: {s1.Név} {s1.Kor}");
```

## Hozzáférésjelző kulcsszavak

- `public` - Teljes nyitottság; bármilyen program bármilyen kódja hozzáfér
- `protected` - Megengedi a leányosztályoknak hogy hozzáférjenek egy (egyébként privát) résztvevőhöz, amit örökölnek.
- `protected internal` - u.a. plusz Programzártság (ld. ezalatt).
- `internal` - Programzártság; csak a programfájl részei férnek hozzá
- `sealed` - Megakadályozza az öröklést.
- `private` - Teljes zártság; csak a saját osztálya fér hozzá

## Módjelző kulcsszavak

`static`, `const`

Ezt a kulcsszót már láttuk több helyen, legelőször a `Main()` funkció előtt.

```
static void Main(string[] args)
```

Az osztály tagjai (*változók*, *eljárások*, *tulajdonságok*) lehetnek **statikus**ként deklarálva, ami *magához* az **osztályhoz** köti őket, nem a példányokhoz.

Ebből kifolyólag az osztály statikus résztvevőinek csak **egy** 'változata' van, kötve az *osztályhoz* globálisan.

**Konstansok** azok a résztvevők, amelyeket *nem lehet változtatni*. Ezek alából **statikusak**.

```
In [ ]: class Matek
{
    private const double pi = 3.1415;           // konstans = static, nem változtatható
    public Matek() {számláló++;}

    public static int számláló = 0;              // statikus változó
    public static int Négyzet(int a) {return a*a;} // statikus eljárás
    public static double Pi {get {return pi;}}   // statikus tulajdonság
}

System.Console.WriteLine(Matek.Pi);             // statikus -> csak az osztály nevéből lehet hívni
System.Console.WriteLine(Matek.Négyzet(6));

Matek a1 = new Matek(); /* 1 */
Matek a2 = new Matek(); /* 2 */
System.Console.WriteLine("Példányok száma: "+Matek.számláló); // a statikus változó az osztályé, nem példányosul!
```

### readonly

A **readonly** szó **csak olvashatóvá** teszi az adott osztályt/tulajdonságot.

Így körülbelül egyenértékű a **const** -tal, de mégis sokban különböznek.

1. A konstansokat mindig deklaráláskor **be kell állítani**. A readonlykat *nem*.
2. Egy readonly változónak a **konstruktor adhat értéket**, a konstans **nem változhat egyáltalán**.
3. Egy readonly változó értéke lehet **egy számítás eredménye**, a konstans viszont **hard-coded** (kézzel gépelt a programba).

```
In [ ]: class Asdfgh
{
    private readonly int szám;           // nem kell inicializálás
    const double PI = 3.1415;           // ide kell
}
```

```
// const double PI = Math.Cos(60); // HIBA! konstans beállítása nem történhet számításból

public Asdfgh(int param)
{
    szám = param + 15;           // ez szabályos, mivel a konstruktor végzi, és szabad számításból beállítani

    // PI = 2.41461;           // HIBA! konstans nem változhat
}
}
```

`this`

A `this` szó a futás közben jelenlevő akkori példányra utal.

A példány saját magából kéri ki az adatot.

```
class Ember
{
    private string név;
    public Ember(string név)
    {
        this.név /* példány név változója */ = név; /* konstruktor paramétere */
    }
}
```

## Inherencia és polimorfizmus

Ez a két szó (sorban) nem jelent mást, mint ezt a két fontos fogalmat:

- **öröklésnek** nevezzük, mikor egy osztály felhasznál egy másik osztályt alapjaként
- **sokoldalúságnak** pedig egy résztvevő sok formáját jelenti *típustól* függően

### Öröklés

Az öröklés például sok kicsi osztály általános tulajdonságainak közös tárgyalására hasznos.

Például egy `Állat` alaposztály hasznos lehet `Kutya` és `Macska` leányosztályok írásában,

hiszen csak *egy helyen* kellhet módosítani a **közös** tényezőket, viszont az **egyéni** dolgaikat a *saját osztályukban* írhatjuk meg.

C#-ban egy osztály **kizárólag csak EGY** darab öröklést engedélyez. Viszont *egybeágyazott öröklés* működőképes (későbbi téma az *interfészek* használata).

Az örökölt konstruktorok is lefutnak az egyéniekkel együtt, ám a sorrend:

[**Any** ctor, *Leány ctor*, *Leány dtor*, **Any** dtor] (*kívülről befelé*)

A `protected` kulcsszó megengedi a leányosztályoknak hogy módosítsanak egy (egyébként privát) résztvevőt, amit örökölnek.

A `sealed` kulcsszó megakadályozza az öröklést teljesen.

```
In [ ]: class Állat                                     // alaposztály / anyaosztály
{
    public int LábSzám {get;set;}
    public int Életkor {get;set;}
}

class Kutya : Állat                                     // syntax: class [osztálynév] : [anyaosztály]
{
    public Kutya() {LábSzám = 4;}                       // anyaosztályból örökölt tényező a konstruktorban
    public void Ugat() => Console.WriteLine("Vau"); // saját tényező
}
```

## Sokoldalúság

Ez szimplán egy alaposztály örökölt résztvetőinek, a **leányosztályokban megváltoztatott**, sokféle formáját jelenti. Máshogy fogalmazva: **felülírható eljárások** az alaposztályban. Ezt a `virtual` kulcsszóval érjük el:

```
class Síkidom {
    public virtual void Rajzol() {
        Console.WriteLine("Alap rajz eljárás");
    }
}
```

Így ha egy leányosztály egy speciálisabb (*de ugyanolyan nevű*) eljárást akar, az `override` kulcsszót használja:

```
class Téglalap : Síkidom {  
    public override void Rajzol() {  
        Console.WriteLine("Téglalap rajzolás!");  
    }  
}
```

Így felülíródik az alaposztály `Rajzol` eljárása.

```
In [ ]: class Síkidom {  
        public virtual void Rajzol() {  
            Console.WriteLine("Alap rajz eljárás");  
        }  
    }  
    class Téglalap : Síkidom {  
        public override void Rajzol() {  
            Console.WriteLine("Téglalap rajzolás!");  
        }  
    }  
    Síkidom t1 = new Téglalap(); // típusa Síkidom, de benne egy Téglalap van
```

### Absztrakt osztályok

Ha pedig nincs értelme egynéhány eljárást definiálni az alaposztályban, akkor használjunk `abstract` osztályt.

Ez *értéktelen*, de *felülírható* eljárásokat engedélyezi.

```
abstract class Síkidom {  
    public abstract void Rajzol();  
}
```

*// osztály elé kell; eljárások elé ahova kell, oda  
// CSAK absztrakt osztályban lehet absztrakt eljárás!!  
// és a leányosztályoknak értéket KELL adni nekik!!!*

Az absztraktok **saját példányosítása tilos**.

```
In [ ]: abstract class Síkidom {  
        public abstract void Rajzol();  
    }  
    class Téglalap : Síkidom {
```

```

    public override void Rajzol() {
        Console.WriteLine("Téglalap rajzolás!");
    }
}
class Kör : Síkidom {
    public override void Rajzol() {
        Console.WriteLine("Kör rajzolás woooooooooooo");
    }
}
Síkidom t1 = new Téglalap();
Síkidom k1 = new Kör();

t1.Rajzol();
k1.Rajzol();

```

## Interfészek

**Interfészeket** használunk hogyha *teljesen absztrakt* osztályt csinálunk, azaz megvannak az általános eljárások, de a leányosztályokban akarjuk definiálni ezeket.

Meglepő módon az `interface` kulcsszóval lehet deklarálni ezeket.

Általában nagy `I` -betűvel kezdjük a nevüket.

Interfészek **nem tartalmazhatnak változókat (mezőket)**.

Mi értelme *interfészt* használni ha van *absztrakt* is?

Azért éri meg *interfészt* használni, mert abból **több is szolgálhat alapul** örökléskor.

`class` Valami : IEgyik, IMásik, etc...

```

In [ ]: public interface ISíkidom {
        void Rajzol();           // default: publikus és absztrakt
    }

class Kör : ISíkidom {
    public void Rajzol() {        // nem kötelező itt a 'virtual' kulcsszó
        Console.WriteLine("Kör rajz :D");
    }
}

```

```
}  
  
ISíkidom k4 = new Kör();           // használjuk az interfészt  
k4.Rajzol();
```

## 12. Fájlok

A `System.IO` névtér tartalmaz pár osztályt az OS fájljainak kezelésére.

Lehet létrehozni, módosítani, törölni fájlokat.

### `File` osztály

A `File` osztály pont erre való:

```
string valami = "Valami szöveg.";  
File.WriteAllText("text.txt", valami);           // átírja a "text.txt" tartalmát a szövegre
```

### Fontos statikus `File` tulajdonságok

- `File.AppendAllText()` a végére fűz szöveget
- `File.Create()` létrehoz egy új fájlt, és visszaad egy `FileStream` példányt
- `File.Delete()` töröl egy fájlt
- `File.Exists()` létezés alapján igaz-hamisat ad vissza
- `File.Copy()` másol egyet
- `File.Move()` mozgat egyet

### `FileStream` osztály

A `FileStream` osztály segít alacsony szinten írni/olvasni/lezárni egy fájlt.

Ennek az osztálynak a szülőosztálya a `Stream` absztrakt.

```
FileStream fájlcska = new FileStream("fájlNév", fájlMód, fájlHozzáférésMód, fájlMegosztásMód);  
/*  
    ahol fájlMód egy FileMode enum = {Append, Create, CreateNew, Open, OpenOrCreate, Truncate}  
    fájlHozzáférésMód egy FileAccess enum = {Read, ReadWrite, Write}
```



```
        fájlMegosztásMód egy FileShare enum = {Inheritable, None, Read, ReadWrite, Write}
    */
```

```
In [ ]: using System.IO;                // ne feledd!
                                              // ebből a névtérből tudja a fordító csak, mi az a FileStream!

FileStream fájl = new FileStream("szia.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);
                                              // szia.txt, nyisd meg vagy hozd létre, írás+olvasás

for(int i = 65; i <= 71; i++)    // 65 mint byte = 'A'; 71 mint byte = 'G'
{
    fájl.WriteByte((byte)i);        // bájt-szinten ír!
}
fájl.Position = 0;                // a kurzort az elejére állítjuk
for(int i = 0; i <= 6; i++)
{
    Console.Write(fájl.ReadByte() + " "); // kiolvassuk a byteokat
}
fájl.Close();                    // és bezárjuk a fájlt, levesszük a lakatot róla
```

StreamReader és BinaryReader osztály

Másik technika a StreamReader / BinaryReader osztályt használni. Ez különben hasonló a standard bemenet olvasásához/írásához (a Console osztály részei TextReader osztályon alapulnak).

Itt használhatunk az automata bezárás érdekében using kulcsszót, ami megadja a fájl kezelőjének

```
StreamReader fájlforrás = new StreamReader("fájl", fájlHozzáférésMód);    // u.a. mint FileStream
string sor;
while((sor = sr.ReadLine()) != null)                                     // 1. Lehet hozzárendelésből visszértéket kapni és azzal
feltételt csinálni
{
    Console.WriteLine(sor);                                              // 2. a 'ReadLine()' eljárás u.a. mint a Console-é
                                                                    // 3. a jelentés: "ha a most beolvasott sor létezik" akkor
nyomtassuk
}
fájlforrás.Close();                                                    // fontos becsukni!

// másik technika:
```

```
using (StreamReader FFájl = new StreamReader("fájl",...)) // a 'using' a blokk végén be is zárja a fájlt,
{ // és felszabadít memóriát
    string sor;
    while ((sor = FFájl.ReadLine()) != null)
    {
        Console.WriteLine(sor);
    }
}
```

## 13. Hibák (exceptionök)

Mikor írunk programot, és mikor futtatjuk azokat, belefuthatunk fordítási és futási hibákba. Ezek jelzik a program nem megfelelő futását, és visszajelzik nekünk.

A *fordítási* hibákat csak is javítással lehet kiküszöbölni, nem lehet elfogni azokat a hibákat, amik nem futás-közben történnek.

A *runtime* (futás-idő) hibákat futtatáskor **dobja** a program, és **azonnal kilép**; a .NET keretrendszer tartalmaz jónéhány hibát a rossz kimenetek hirtelen lekötésére. Ezeket **Exception**-öknek (*kivételeknek, hibáknak*) nevezzük.

**try-catch**

A hibákat a **try-catch** utasításokkal el lehet fogni, így nem fog a hiba miatt *kilépni* a futásból a program. A **finally** utasítás pedig hibától függetlenül lefuttat kódot. Syntax:

```
try
{
    // bizonytalan eljárások...
}
catch (Exception hiba)
{
    // hibakezelés itt (az elfogott hiba/hibák 'hiba' változóban lesznek)
}
finally
{
}
```

```
    // független a hibáktól...
}
```

```
In [ ]: try
{
    /* Unkommenteld az egyes sorokat a különféle hibákért */
    int[] számok = {1, 2, 3};
    // Console.WriteLine(számok[10]); // HIBA! `IndexOutOfRangeException`
    int nullakerdojel = 5-5; // nem konstans nulla, így lesz runtime hiba
    int nullávalOsztok = 10/nullakerdojel; // HIBA! `DivideByZeroException`
}
// érdekesség: ha konstans nullával osztasz, az fordítási hibának számít! (pl. "10/0"
catch (Exception e)
{
    Console.WriteLine(e.Message); // minden hibának van 'Message' tulajdonsága (üzenete)
}
finally
{
    Console.WriteLine("Akkor is lefuttatnak engem.");
}
```

```
throw
```

Ha pedig mi akarunk hibákat dobni és potenciálisan leállítani a hibás programot,  
a `throw` szóval meg tudjuk tenni. Egy hiba osztály (minimum `System.Exception`)  
**példányát** (kell a `new`) kell megadni.

```
ArithmeticException hibaPéldány = new ArithmeticException("Buta vagy!");
throw hibaPéldány; // előre példányosított
throw new Exception(); // futás közben egy újat példányosít
```

```
// vagy ha újradozni akarunk egy hibát
try
{
}
catch(Exception exc)
{
    throw; // visszadobja automatikusan az 'exc' hibát
}
```

```
In [ ]: int bemenet = 15;
        if(bemenet < 18)
        {throw new ArithmeticException("Nem vagy elég idős");}
        else
        {Console.WriteLine("Szia! Mit adhatok?");}
```

## 14. Általánosítás és általános eljárások

**Általánosítunk**, ha *több típusra* akarunk ugyanolyan (vagy hasonló) feladatot végző eljárást készíteni.

Például ha bevezetnénk egy `Csere` eljárást ami megcseréli két változó értékét a helyüket megtartva:

```
static void Csere(ref int a, ref int b)           // (a 'ref' szócskát vettük, referenciaként adja a változót)
{
    int ideigl = a;
    a = b;
    b = ideigl;
}
```

Ha több típusra is akarnánk ugyanezt használni, általánosítunk.

Könnyebb, olvashatóbb, és kezelhetőbb megoldás.

Használata: az eljárásnév után "<>"-t rakunk és beleírjuk az általános típusokat

```
In [ ]: static void Csere<Ált>(ref Ált a, ref Ált b)           // a "Ált" típus itt egy általános típust jelent
        {                                                       // bárminek nevezheted az általános típust
            Ált ideigl = a;                                       // több is lehet pl. Func<T,U,V>
            a = b;
            b = ideigl;
        }

        string első = "siuu", második = "hihi";
        Csere<string>(ref első, ref második);                   // hívás: meg kell szabni híváskor a konkrét típust!
        Console.WriteLine($"{(string) első} | második = {masodik}");
```

```
int egy = 669, ketto = -161616;  
Csere<int>(ref egy, ref ketto);           // hívás: meg kell szabni híváskor a konkrét típust!  
Console.WriteLine($"(int) egy = {egy} | ketto = {ketto}");
```

## 15. Delegátok, lambda op, és anonim eljárások

### Delegátok

Hogyha egy eljárásba paraméterként eljárást akarunk megadni, **delegátot** használunk.

(Bővebben: a *delegate* típus egy referens típus amiben eljárás-referencia lehet)

Ugyanúgy példányosítani kell, még hozzá a konstruktorában a megadott eljárás nevével.

Például itt egy delegát amibe egy **string**-paraméterű, **int**-visszértékű eljárás megy:

```
In [ ]: public delegate int Delegátus (string s);           // string-param int-return delegate  
  
public static int SzóBetűSzám(string szó) {return szó.Length;} // string-param int-return eljárás  
  
Delegátus d1 = new Delegátus(SzóBetűSzám); // új példány, benne ZÁRÓJEL NÉLKÜL a kiválasztott eljárás neve  
Console.WriteLine(d1("héber"));
```

### Multicasting

Lehet **több** delegát-példányt egybekötni, ezt **multicasting**-nak hívják.

Ezzel egy név alatt több, *ugyanolyan típusú* delegát hívható.

A hozzákötést a **+** oppal, levételt a **-** oppal lehet.

```
In [ ]: public static int Összead(ref int n, int p) { // ref-int-param int-param int-return eljárás  
    n += p;  
    return n;  
}  
  
public static int Szoroz(ref int n, int q) { // ref-int-param int-param int-return eljárás  
    n *= q;  
    return n;  
}
```

```

public delegate int SzámVáltoztató(ref int a, int b); // ref-int-param int-param int-return delegát

int num = 10; /* 10-ről indulunk */

SzámVáltoztató nc; // példányosítások
SzámVáltoztató nc1 = new SzámVáltoztató(Összead);
SzámVáltoztató nc2 = new SzámVáltoztató(Szoroz);

nc = nc1; // bekötjük az összeadó példányt
nc += nc2; // hozzákötjük az összeadó példányhoz a szorzót SORBAN!

nc(ref num, 5); // a multicast hívása: 10 --[Összead]-> 15 --[Szoroz]-> 75
Console.WriteLine("Value of Num: {0}", num);

```

## Lambda-op és Anonim kifejezés

Az egyszerű eljárásokat egyetlen operátorral is megadhatjuk, ez a lambda-operátor ( `=>` ). Ez egy **kifejezéses** v. **állításos** eljárást készít. (*expression*- v. *statement*-lambda)

Ha egy kifejezés-lambda-t nem akarunk elnevezni, **anonim** (névtelen) eljárásnak hívjuk. Ezek visszértéke a bennük levő utolsó kifejezés értékén múlik Syntax:

(paraméterek) `=>` kifejezés

```

In [ ]: public static int Négyzet(int x) => x*x;
        // u.a. mint Négyzet(int x) {return x*x;}

public static void Szia(string s) => Console.WriteLine($"Szia {s}!");
        // u.a. mint Szia(string s) {Console.WriteLine($"Szia {s}!");}

List<int> tomb = new List<int>() {3, 5, 3, 2};
List<int> sorbaTomb = tomb.Select(x=>x*x).ToList();
        // minden elemet négyzetévé alakítottunk
        // u.a. mint foreach(x in tomb) {sorbaTomb.Add(x*x);}

Console.WriteLine(Négyzet(5));
Szia("Péter");

```

```
sorbaTomb.ForEach(Console.WriteLine);
```

## LINQ

A **LINQ** (Language-Integrated Query) egy olyan *lekérdezési* nyelvezet, aminek segítségével könnyedén tudunk számlálható ( `IEnumerable` interface; *foreach-elhető*) gyűjteményeken dolgozni.

Két szintaxisa létezik: **metódus-láncos** és **\*Query\***.

### Metódus-lánc syntax

A *metódus-láncos* alakot használtuk (ld. stringek sorba rendezése), ekkor egy gyűjtemény **metódusait** (saját eljárásait) előhívjuk sorban, tetszés szerint:

```
In [ ]: string[] szia = {"a", "abcde", "a", "abcdefgh", "abc"}; // egy gyűjtemény
var sorban = szia // (az újsorok nem változtatnak semmit, csak érthetőbb)
                .Where(y => y.Length >= 3) // "ahol: az elem hossza nem kevesebb mint 3"
                .OrderBy(x => x.Length) // "rendezd: elem-hosszuság szerint"
                .Distinct() // "csak: a különbözőket"
                .Select(e => e + $" [{e.Length}]") // "módosíts: elemenként ..."
                ; // vége //
foreach (var item in sorban) Console.WriteLine(item);
```

### Query (comprehension) syntax

Ezt a program-darabot egy másik módon, **Query**-vel írva hasonló struktúrájú, *könnyebben olvasható* szeletet kapunk.

A **Query** célja hogy az adatbázis-kezelő nyelvekhez (SQL, Visual Basic) hasonlítson, mint egy lekérdező (hence the name) nyelv, könnyítve a programozást.

Kulcsszavai *majdnem* egy az egyben egyeznek a metódusokkal:

```
In [ ]: string[] szia = {"a", "abcde", "a", "abcdefgh", "abc"}; // egy gyűjtemény
var sorban = (from elem in szia // kötelező sor, elem = ideiglenes változó
              where elem.Length >= 3 // "ahol:"
              orderby elem.Length // "rendezd:"
              select elem + $" [{elem.Length}]") // kötelező sor (select v. group)
```

```

        .Distinct() // ez sajna nincs bent a query szintaxisban
        /*.Select(e => e + $" [{e.Length}]"*/ // áthelyezve a query-be, (ld 5. sor)
        ;
foreach (var item in sorban) Console.WriteLine(item);

```

```

In [ ]: using System.Linq; // Linq névtér!!

static int Hossz(string bemenet) // string-param int-return eljárás
{
    return (int)(bemenet.Length); // azaz: egy eljárás, ami string-et vesz be és int-et ad
}

string[] szia = {"a", "abcde", "abc", "ghijkl"};

string[] sorban = szia.OrderBy(Hossz).ToArray();
/* sorrendben:
 *> szia = alap string[] tárgy
 *> .OrderBy(Hossz) = sorba rakjuk,
 * azaz minden egyes elemét keresztülvezetjük 'Hossz' eljáráson,
 * amíg egészsámokat nem kapunk (azokat sorba lehet rakni)
 * ám kimenetként a LINQ rendezett állományát kapjuk (System.Linq.IOrderedEnumerable)
 *> .ToArray() = visszaalakítjuk a 'System.Linq.IOrderedEnumerable'-t string[]-be
 */
foreach (var item in sorban)
{
    Console.WriteLine(item);
}

```

## Fontosabb LINQ tulajdonságok

Íme néhány fontos LINQ eljárás az IEnumerable interfészt használókra ( `System.Linq` névtér!):

Metódus ( <i>vissz-érték-lambda</i> )	Query-szintaxis	Jelentés
<code>var x = gyűjt</code>	<code>from x in gyűjt</code>	A lekérdezés kezdete, <code>x</code> az ideiglenes változó <code>gyűjt</code> forrásból
<code>.Select( <i>var-lambda</i> )</code>	<code>select valami</code>	Adatkiválasztás, query végén <b>kötelező!</b>
<code>.Where( <i>bool-lambda</i> )</code>	<code>where feltétel</code>	Igaz-hamis feltételes kiválasztás



Metódus ( <i>vissz-érték-lambda</i> )	Query-szintaxis	Jelentés
<code>.OrderBy( <i>int-lambda</i> ) / .OrderByDescending( <i>int-lambda</i> )</code>	<code>orderby tulajd irány</code>	Rendszerezés szám-kifejezés alapján
<code>.Join( <i>gyűjt2</i> , <i>var-lambda</i> , <i>var-lambda</i> , <i>var-lambda</i> )</code>	<code>join y in gyűjt2 on x-tulajd equals y-tulajd</code>	Összekapcsol két gyűjteményt egy közös kulcs használatával. Metódusként az első két <i>var-lambda</i> a query-nek az <code>equals</code> részével egyezik meg. A harmadik <i>var-lambda</i> megegyezik egy <code>select new {}</code> résszel
<code>.GroupBy( <i>var-lambda</i> )</code>	<code>group x by x- tulajd into csop</code>	Csoportosítja a bejövő adatokat egy tulajdonság szerint. Egy csoport egy kulcsból ( <i>Key</i> ) és elemeiből áll, ezért két <code>foreach</code> is kellhet
<code>.Skip( <i>w</i> )</code>	-	Kihagy <i>w</i> elemet az elejéről
<code>.Take( <i>w</i> )</code>	-	Kiválaszt <i>w</i> elemet az elejéről
<code>.First() / .FirstOrDefault()</code>	-	Az elsőt kiválasztja (vagy a gyűjtemény defaultját, általában <code>null</code> -t) <sup>1</sup>
<code>.Last() / .LastOrDefault()</code>	-	Az utolsót kiválasztja (vagy a gyűjtemény defaultját, általában <code>null</code> -t) <sup>1</sup>
<code>ElementAt( <i>i</i> ) / ElementAtOrDefault( <i>i</i> )</code>	-	Az <i>i</i> -indexű elemet kiválasztja (vagy a gyűjtemény defaultját, általában <code>null</code> -t)
<code>.Distinct() / .DistinctBy( <i>var-lambda</i> )</code>	-	Kizárja az ismétlődéseket. A vizsgált adattípusnak tartalmaznia kell egy megfelelően felülírt <code>.Equals()</code> metódust. Az új .NET 6.0 óta létezik a <code>.DistinctBy()</code> metódus, ami egy tulajdonság alapján szűri ki csak. Régebbi verzióknál egyenértékű kód: <code>.GroupBy(*var-lambda*).Select(x =&gt; x.First())</code>
<code>.Count( ... )</code>	-	Megszámolja a gyűjtemény elemeit <sup>1</sup>
<code>.Any( ... )</code>	-	Igaz-hamisat ad vissza ha tartalmaz elemet <sup>1</sup>
<code>.Min( <i>int-lambda</i> )</code>	-	Megkeresi a minimumot. Ha szám-gyűjtemény akkor nem szükséges lambdát írni <sup>2</sup>
<code>.Max( <i>int-lambda</i> )</code>	-	Megkeresi a maximumot. Ha szám-gyűjtemény akkor nem szükséges lambdát írni <sup>2</sup>
<code>.Avg( <i>int-lambda</i> )</code>	-	Megkeresi az átlagot. Ha szám-gyűjtemény akkor nem szükséges lambdát írni <sup>2</sup>
<code>.Concat( <i>gyűjt2</i> )</code>	-	Összekapcsol két gyűjteményt

Metódus ( <i>vissz-érték-lambda</i> )	Query-szintaxis	Jelentés
<code>.ToArray() / .ToList() / .ToDictionary( ... )</code>	-	Visszaalakítja a LINQ saját típusát <i>tömbbé, listává, vagy szótárrá</i> . A szótár-konvertálásba kell két paraméter: az első a <i>kulcs-hozzárendelés</i> , a második az <i>érték-hozzárendelés</i> lambdája

<sup>1</sup>: Opcionális paraméterként lehet **feltétel-lambda**t írni. Így mint egy *where*-ként választja ki csak a megfelelőket.

<sup>2</sup>: .NET 6-ban alternatívájuk a `...By()` metódus, ami nem a lambda-kifejezés típusával tér vissza, hanem az *eredeti tárolt típusával*.

## Licensz

© Daniel Adam Farkas 2022



Dieses Werk ist lizenziert unter einer [Creative Commons Namensnennung - Weitergabe unter gleichen Bedingungen 4.0 International Lizenz](#).



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).