programozas_gyorstalpalo_nehezebbik

August 27, 2022

1 A neheze csak most jön

1.0.1 10. Struktúrák és elszámolások (enumok)

Struktúrák C#-ban egy **struktúra** adott változótípusok csoportosítása, így új adattípus jön létre.
Létrehozását a `struct` szóval kezdjük.

```
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... fúzni hozzá.

De miért használjunk structokat osztályok helyett? - osztályok referens típusok; struktúrák érték típusok - 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 megjegyezni mint a '2' számot), de használatuk feladatonként eltér.

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

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
```

[ ]:
```
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}", k3.cím, k3.oldalSzám, k3.műnem     // a műnem␣
 ↪nevét kapjuk ha implicit alakítunk stringbe!
    );
```

1.0.2 11. Névterek, Osztályok, Tárgyak

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

Egy OOP nyelvben az **osztály** egy adattípus ami csoportosít egyéb változókat és eljárásokat az osztály **tárgyai**hoz.
Például egy *csavarhúzó* tárgynak vannak adatai:

- fej típusa (csillag, lapos, torx, imbusz, etc.)
- fej mérete (M16, M10, M22, etc.)
- nyél típusa (gumírozott, fa, fém)
- ára, anyaga, etc.

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

Ezt így lehetne felírni:

```
                                // CSAK MÁS osztályokon és eljárásokon kívül!!!
class Csavarhúzó                // elnevezés csak szokásosan!
{
    public int iFejMéret;       // tulajdonságok
    public string nyélTípus;
    string fejTípus;            // alapként mindegyik 'private' és csak azonos kódfájlból lehet

    public void Használ()       // osztályon belüli eljárás
    {
        Console.WriteLine("> Használtad a csavarhúzót.");
    }
}
```

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

```
// a main()-ben:
Csavarhúzó cs1 = new Csavarhúzó();          // cs1 változóba kreálunk egy új csavarhúzót a 'new'
cs1.Használ();                              // majd a példány eljárását (metódusát) hívjuk a '.'
```

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);
```

[ ]:
```
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 viszont *mint referencia* kapjuk meg őket.
Ekkor a tárgy változó a `stack`-ből a `heap`-re mutat (**pointer**), de maga a példány egy másik helyen, a `heap`-ben van.
A változóban ekkor egy *referencia* van, egy *cím* a heap-beli tárgyhoz.

```
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 *dinamikusként*, mivel az egyéni példányok lehet **több memóriát** kérhetnek.

Enkapszuláció és elérhetőség 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 elérhetőséget:
`public`, `private`, `protected`, `internal` és `protected internal`

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

[ ]:
```
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;
    }
}
```

Építők és Lebontók Hogyha új példányt csinálunk egy tárgyból, a kezdeti paramétereket szeretnénk beállítani,
esetleg start-funkciókat elindítani.
Erre való az osztály **építőfunkciója** (*konstruktorja*). A konstruktor
minden példány készítésekor lefut.

```
class Személy
{
    private int Kor;                // privát változó, biztonság érdekében

    public Személy(int x)    // KONSTRUKTOR (mindig publikus, hogy lehessen hívni)
    {                                // u.a. a neve mint az osztálynak és nincs vissz-értéke!
        Kor = x; /* itt a konstr. paramétereit bevisszük */
        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
```

Fontos, hogy a konstruktort is lehet túltölteni.

[ ]:
```
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()}");
```

Az építő ellentettje a **lebontó** (*destruktor*).
Amikor egy példány nem kell többé, memóriafelszabadítás érdekében kitöröljük,
ekkor kapcsol be a destruktor.

```
class Asztal
{
    public Asztal()
    {
        //konstruktor
    }

    ~Asztal()                    // u.a. a név!
    {                            // csak EGY destruktor lehet!
        //destruktor             // nem lehet hívni manuálisan!
    }                            // nem lehet elérh. kulcsszava, paramétere és vissz-értéke!
}
```

Getterek és Setterek Az utóbbi kódban látható az enkapszulációnak köszönhető
beállító (*setter*) és visszaadó (*getter*) funkciópáros.
Ezeket **tulajdonságoknak** (*property*) hívjuk.
A *property*-knek a fordító **implicit** módon (*magától értetődően*) ad értéket.

Tehát ha *értéket váró* eljárásba rakjuk, a `get`-et használja,
ha pedig egy *értéket adunk ennek*, akkor a `set`-et.

Az egyik egyszerűbb és gyorsabb módja ennek, ha
a tulajdonság-szinaxist használjuk:

```
class Autó
{
    private string típus;

    /*    tulajdonság    */
    public string Típus                 // a név független, lehetne bármi más
    {
        get {return típus;}             // ha egy fn. bemenetet kér
        set {típus = value;}            // ha hozzárendelésben használjuk
    }
}

// main():
Autó a1 = new Autó();
a1.Típus = "Hatchback";                 // itt a setter
System.Console.WriteLine(a1.Típus);     // itt a getter
```

A tulajdonságból szabadon ki lehet hagyni a get-et vagy a set-et,
bármilyen feladatot bele lehet tenni
és lehet külön elérhetőségi kulcsszavat adni neki.

[ ]:
```
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. bemenet tisztitására
                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}");
```

Statikus jelző, konstansok és readonlyk, this kulcsszó Ezt a kulcsszót már láttuk több helyen, legelősször
a `Main()` funkció előtt.

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

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

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

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

[ ]:
```
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!
```

A `readonly` szó **csak olvashatóvá** teszi az adott osztályt/tulajdonságot.
Így körülbelül egyenértékű a `const`-tal, de mégis sokban különböznek.

1. A *konstans*okat mindig deklaráláskor **be kell állítani**. A *readonly*kat *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).

[ ]:
```
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
    }
}
```

A `this` szó a futás közben jelenlevő akkori példányra utal.
Hasznos mikor különbséget kell tenni két név között.

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

1.0.3 Inheritancia 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* függöen

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.

Ez következőképpen néz ki:

[ ]:
```
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.Write("Vau"); // saját tényező
}
```

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:
[**Anya ctor**, *Leány ctor*, *Leány dtor*, **Anya dtor**] (kívülről befelé)

A `protected` kulcsszó megengedi a leányosztályoknak hogy módosítsanak egy (egyébként privát) résztvevőt, amit örökölnek.
A `sealed` kulcsszó megakadályozza az öröklést teljesen.

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.

[ ]:
```
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
```

Ha pedig nincs értelme egynéhány eljárást definiálni az alaposztályban, akkor használjunk `abstract` osztályt.
Ez *értéktelen*, de *felülírható* eljárásokat engedélyezi.

```
abstract class Síkidom {                // osztály elé kell; eljárások elé ahova kell, oda
    public abstract void Rajzol();      // CSAK absztrakt osztályban lehet absztrakt eljárás!!
}                                       // és a leányosztályoknak értéket KELL adni nekik!!!
```

Az absztraktok **saját példányosítása tilos**.

[ ]:
```
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 wooooooooo");
    }
}
Síkidom t1 = new Téglalap();
Síkidom k1 = new Kör();

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

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.

[ ]:
```
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();
```

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

Mi értelme *interfészt* használni ha van *absztrakt* is?
Azért éri meg *interfészt* használni, mert abból **több is szolgálhat alapul** öröklésкor.

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

1.0.4 12. Fájlok

A `System.IO` névtér tartalmaz pár osztályt az OS fájljainak kezelésére.
Lehet létrehozni, módosítani, törölni fájlokat.

A `File` osztály pont erre való:

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

Fontosabb statikus `File` tulajdonságok:

- `File.AppendAllText()` a végéreфűz 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

A `FileStream` osztály segít alacsony szinten írni/olvasni/lezárni egy fájlt.
Ennek az osztálynak a szülőosztálya a `Stream` absztrakt.

```
FileStream fájlocska = 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}
*/
```

[ ]:
```
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
```

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 FileStre
string sor;
while((sor = sr.ReadLine()) != null)                    // 1. lehet hozzárendelésből visszértél
{                                                       // 2. a 'ReadLine()' eljárás u.a. mint
    Console.WriteLine(sor);                             // 3. a jelentés: "ha a most beolvasot
}
fájlforrás.Close();                                     // fontos becsukni!

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

1.0.5 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 kimenetelek hirtelen lekötésére.
Ezeket `exception`-öknek (kifogásoknak, hibáknak) nevezzük.

Ezeket a `try-catch` szópárossal el lehet fogni, így nem fog a hiba miatt *kilépni* a futásból a program.
Syntax:

```
try
{
    // ide a bizonytalan eljárásokat
}
```

```
catch (Exception hiba)
{
    // hibakezelés itt (az elfogott hiba/hibák 'hiba' változóban lesznek)
}
```

Például egy tömblekérdezés rossz indexnél hibát dob (méghozzá `IndexOutOfRangeException` típusút)

```
int[] számok = {1, 2, 3};
Console.WriteLine(számok[10]); // HIBA! nincs 10-es indexű ebben a tömbben
```

Ekkor ezt egy `try`-testbe ágyazzuk, majd utána a `catch`-el kezeljük a problémát.

A `finally` szó pedig hibától függetlenül lefuttat kódot.

```
try
{
    int[] számok = {1, 2, 3};
    Console.WriteLine(számok[10]);
}
catch (Exception e)
{
    Console.WriteLine(e.Message);    // minden hibának van 'Message' tulajdonsága (üzenete)
}
finally
{
    Console.WriteLine("Akkor is lefuttatnak engem.");
}
```

[ ]:
```
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.");
}
```

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

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

1.0.6 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 adj
{
    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

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

1.0.7 15. Delegátok, lambda op, és anonim eljárások

Hogyha egy eljárásba paraméterként eljárást akarunk passzolni,
delegátot használunk. (Bővebben: a *delegate* típus egy referens típus amiben eljárás-referencia lehet)

Ugyanúgy példányosítani kell, méghozzá a konstruktorában a megadott eljárás nevével.

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

[ ]:
```
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"));
```

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.

[ ]:
```
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);
```

Legtöbbször *sorbarendezéskor*, *kiválasztáskor*, *szerkesztéskor* fogunk delegátokkal találkozni.
Ezekhez kell a **LINQ** (Language-Integrated Query) névtér. Magát a LINQ-szintaxist a következő pontban tárgyalom.

[ ]:
```
using System.Linq;                  // linq névtér!!

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

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 egésszámokat 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);
}
```

Hogyha pedig csak *egyszeri alkalommal* kell egy ilyen delegátot alkalmazni,
érdemes **lambda operátorral** (=>) dolgozni.
Ez egy **egy kifejezéses** v. **állítás-szerű** eljárást készít. (*expression-* v. *statement*-lambda)
Eljárás-deklarálásnál is lehet használni, ahol a vissz-érték megfelelő:

```
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(stri
```

Ha még nevet sem adunk neki, **anonim funkciónak** hívják (névtelen).
Ekkor a szintaxis a következő: `(paraméter(ek)) => érték/test`

```
[ ]: string[] szia = {"a", "abcde", "abc", "abcdefg", "ab"};
     string[] sorban = szia.OrderBy(x => x.Length).ToArray();    /* mivel az .
      ↪OrderBy egy 'int-return'-ös funkciót várt,
                                                                  *    (int)x.
      ↪Length-et adok vissza direkt az anonimban.
                                                                  * a paraméter␣
      ↪zárójel csak egy param esetén hagyható el
                                                                  */
     Console.WriteLine(String.Join(" ", sorban))
```

1.0.8 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***.

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

```
[ ]: string[] szia = {"a", "abcde", "a", "abcdefgh", "abc"}; // egy gyüjtemény
     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);
```

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:

```
[ ]: string[] szia = {"a", "abcde", "a", "abcdefgh", "abc"}; // egy gyüjtemény
     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);
```

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

Metódus (***vissz-érték**-lambda*)	Query-szintaxis	Jelentés
`var` *x* `=` *gyűjt*	`from` *x* `in` *gyűjt*	A lekérdezés kezdete, `x` az ideiglenes változó `gyűjt` forrásból
`.Select(`***var**-lambda*`)`	`select` *valami*	Adatkiválasztás, query végén **kötelező**!
`.Where(`***bool**-lambda*`)`	`where` *feltétel*	Igaz-hamis feltételes kiválasztás
`.OrderBy(`***int**-lambda*`)` / `.OrderByDescending(`***int**-lambda*`)`	`orderby` *tulajd irány*	Rendszerezés szám-kifejezés alapján
`.Join(`*gyűjt2*`,` ***var**-lambda*`,` ***var**-lambda*`,` ***var**-lambda*`)`	`join` *y* `in` *gyűjt2* `on` *x-tulajd* `equals` *y-tulajd*	Összekapcsol két gyűjteményt egy közös kulcs használatával. Metódusként az első két *var-lambda* a query-nek az `equals` részével egyezik meg. A harmadik *var-lambda* megegyezik egy `select new {}` résszel

Metódus (***vissz-érték**-lambda*)	Query-szintaxis	Jelentés
`.GroupBy(`***var**-lambda*`)`	`group` *x* `by` *x-tulajd* `into` *csop*	Csoportosítja a bejövő adatokat egy tulajdonság szerint. Egy csoport egy kulcsból (`Key`) és elemeiből áll, ezért két `foreach` is kellhet
`.Skip(`*w*`)`	-	Kihagy `w` elemet az elejéről
`.Take(`*w*`)`	-	Kiválaszt `w` elemet az elejéről
`.First()` / `.FirstOrDefault()`	-	Az elsőt kiválasztja (vagy a gyűjtemény defaultját, általában `null`-t)*1*
`.Last()` / `.LastOrDefault()`	-	Az utolsót kiválasztja (vagy a gyűjtemény defaultját, általában `null`-t)*1*
`ElementAt(`*i*`)` / `ElementAtOrDefault(`*i*`)`	-	Az *i*-indexű elemet kiválasztja (vagy a gyűjtemény defaultját, általában `null`-t)

Metódus (***vissz-érték***-*lambda*)	Query-szintaxis	Jelentés
`.Distinct()` / `.DistinctBy(`***var***-*lambda*`)`	-	Kizárja az ismétlődéseket. A vizsgált adattípusnak tartalmaznia kell egy megfelelően felülírt `.Equals()` metódust. Az új .NET 6.0 óta létezik a `.DistinctBy()` metódus, ami egy tulajdonság alapján szűri ki csak. Régebbi verzióknál egyenértékű kód: `.GroupBy(*var-lambda*).Sel => x.First())`
`.Count(…)`	-	Megszámolja a gyűjtemény elemeit_1_
`.Any(…)`	-	Igaz-hamisat ad vissza ha tartalmaz elemet_1_
`.Min(`***int***-*lambda*`)`	-	Megkeresi a minimumot. Ha szám-gyűjtemény akkor nem szükséges lambdát írni_2_
`.Max(`***int***-*lambda*`)`	-	Megkeresi a maximumot. Ha szám-gyűjtemény akkor nem szükséges lambdát írni_2_

Metódus (***vissz-érték***-*lambda*)	Query-szintaxis	Jelentés
`.Avg(`***int***-*lambda*`)`	-	Megkeresi az átlagot. Ha szám-gyűjtemény akkor nem szükséges lambdát írni_2_
`.Concat(`gyűjt2`)`	-	Összekapcsol két gyűjteményt
`.ToArray()` / `.ToList()` / `.ToDictionary(…)`	-	Visszaalakítja a LINQ saját típusát *tömbbé*, *listává*, vagy *szótárrá*. A szótár-konvertálásba kell két paraméter: az első a *kulcs-hozzárendelés*, a második az *érték-hozzárendelés* lambdája

1: Opcionális paraméterként lehet **feltétel-lambdát** írni. Így mint egy *where*-ként választja ki csak a megfelelőket.

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