



Eötvös Loránd Tudományegyetem
Informatikai Kar

Eseményvezérelt alkalmazások

1. előadás

A .NET platform és a C# programozási nyelv

Cserép Máté
mcserep@inf.elte.hu
<https://mcserep.web.elte.hu>

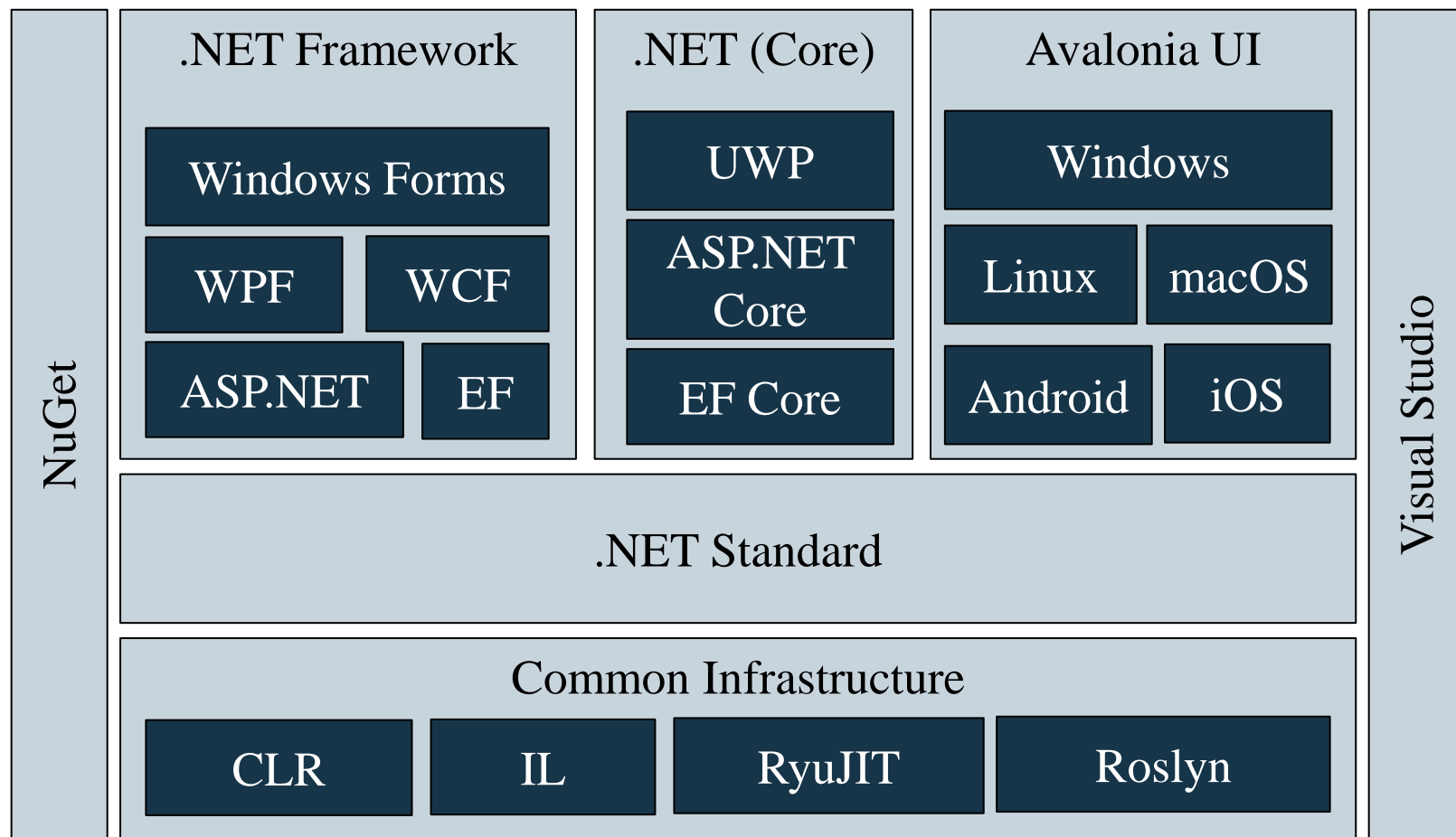
A .NET platform

A .NET platform

- A *.NET platform* a Microsoft szoftverfejlesztési platformja, amely több keretrendszer és programcsomag együttese
 - egy egyéges alapra (*.NET Standard*) épül, többplatformos szoftverfejlesztést tesz lehetővé (*.NET Framework*, *.NET Core*, *Mono Framework*, *WinUI*, *Xamarin / MAUI*, *Avalonia UI*, *Unity*)
 - központi programozási nyelve a *C#*, de számos nyelvet támogat (pl. *Visual C++*, *Visual Basic.NET*, *F#*), a programok egy közös köztes nyelvű programkódra (*Intermediate Language*, *IL*) fordulnak, amely platformfüggetlen
 - a köztes nyelvű kódot a virtuális gép (*Common Language Runtime*, *CLR*) interpretálja, amely biztosítja a futás felügyeletét (pl. szemétygyűjtés) és a dinamikus programozás támogatását (pl. reflexió)

A .NET platform

A .NET platform



A .NET platform

.NET Framework és .NET Core keretrendszerek

- A Microsoft historikus okokból készített két keretrendszert.
- A .NET Framework 1.0 verziója 2002-ben jelent meg, a folyamatos fejlődés mellett több probléma jelent meg az idők folyamán:
 - Windows-központú megközelítés
 - Monolitikus, nem megfelelően modularizált felépítés
 - Zárt forráskód
- A .NET Core (megjelent 2016-ban) ezekre nyújt megoldást:
 - Cross-platform (Windows, Linux, macOS)
 - Modularizált felépítés, csak az alkalmazáshoz szükséges komponenseknek kell jelen lennie.
 - Nyílt forráskód (<https://github.com/dotnet/core>)

A .NET platform

A .NET Standard

A .NET Standard egységes könyvtárat minden, a platformra épülő keretrendszer implementálja, és megadható a verziók közötti megfeleltetés:

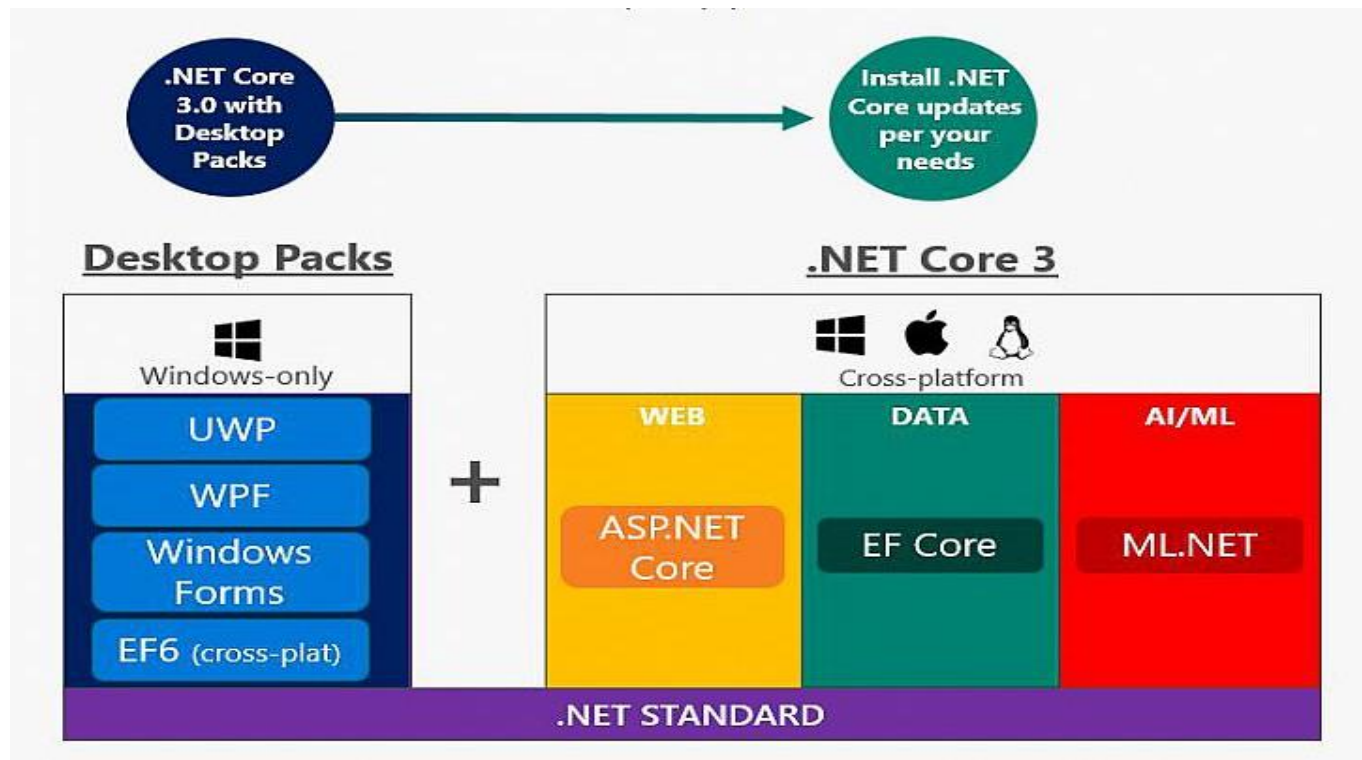
.NET Standard	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1	N/A
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.16
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.16
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	10.0
Universal Windows Platform	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	N/A
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2021.2

Forrás: <https://docs.microsoft.com/en-us/dotnet/standard/net-standard>

A .NET platform

Grafikus alkalmazásfejlesztés .NET Core 3 keretrendszerrel

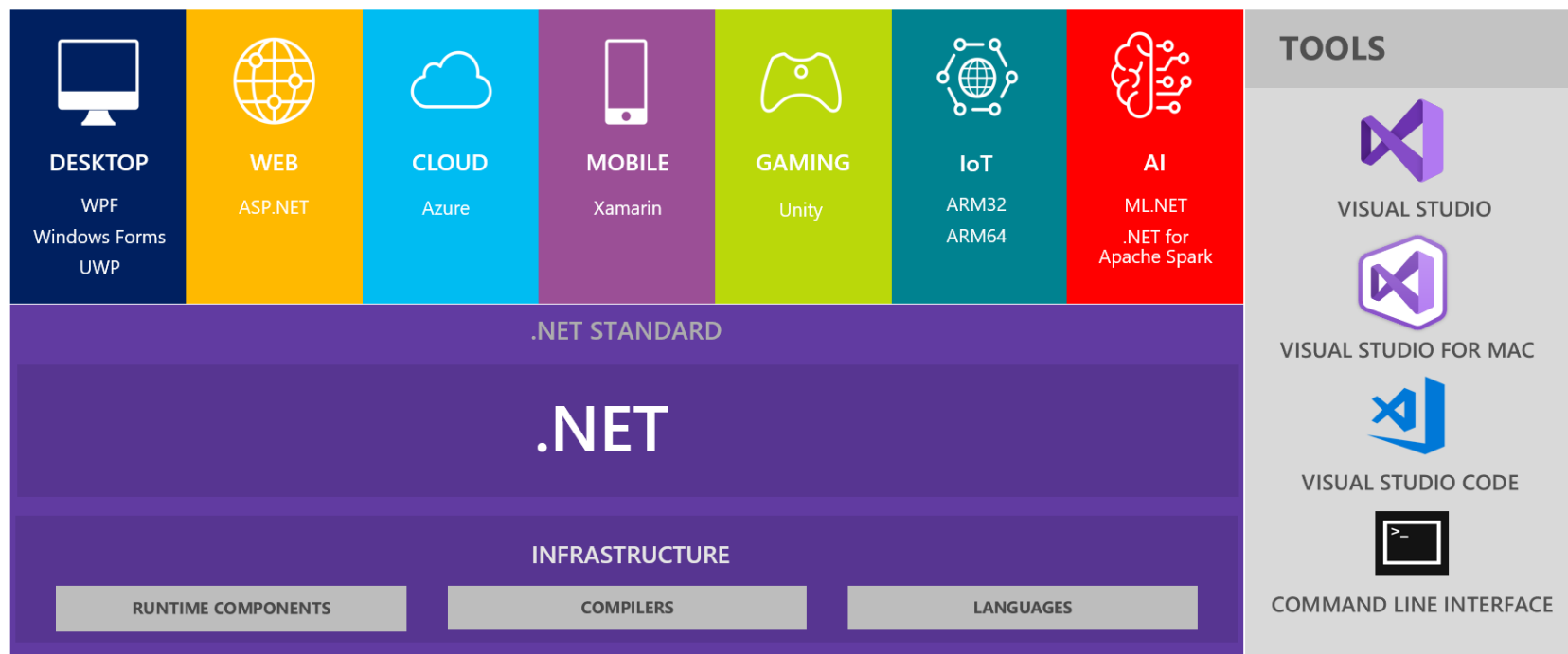
- A .NET Core 3-tól a keretrendszer tartalmazhat ún. *desktop package*et, amelyek csak bizonyos platformokon érhetőek el.
- Ilyen módon .NET Core alatt is elérhetővé vált az asztali alkalmazásfejlesztés WinForms és WPF alkalmazásával.



A .NET platform

A .NET keretrendszer jövője

- A .NET Framework nem kerül tovább fejlesztésre, a 4.8 az utolsó verzió
- A .NET Core 3.1 után a következő verzió a .NET 5 nevet kapta, a verziózásból adódó félreértések lehetőségének csökkentése érdekében.
 - Jelenleg a .NET 8 (LTS verzió) és a .NET 7 az aktuális kiadások.



A .NET platform és a C# programozási nyelv

Lehetőségei

- A C# tisztán objektumorientált programozási nyelv, amely teljes mértékben a .NET platformra támaszkodik
 - szintaktikailag nagyrészt C++, megvalósításában Java



A .NET platform és a C# programozási nyelv

Lehetőségei

- A C# tisztán objektumorientált programozási nyelv, amely teljes mértékben a .NET platformra támaszkodik
 - szintaktikailag nagyrészt C++, megvalósításában Java
 - egyszerűsített szerkezet, strukturált felépülés névterekkel
 - tisztán objektumorientált, egyszeres öröklődéssel, minden típus egy .NET osztály, vagy leszármazottja
 - támogatja a sablon-, eseményvezérelt, és funkcionális programozást
 - a forrásfájl kiterjesztése: **.cs**
 - kódolás: Unicode 3.0

A .NET platform és a C# programozási nyelv

A „Hello, World!” program

```
using System; // névtér használatba vétele

namespace Hello // névtér
{
    class HelloWorld // osztály
    {
        static void Main() // statikus főprogram
        {
            Console.WriteLine("Hello, World!");
            // kiírás konzol képernyőre (a Console
            // osztály statikus WriteLine metódusa)
        }
    }
}
```

A .NET platform és a C# programozási nyelv

Top level statements

- A C# nyelv .NET 6-os verziója (10.0) óta a program belépési pontját tartalmazó fájl procedurális szintaxissal is elkészíthető.
 - Valójában ilyenkor is objektum-orientált program készül, a fordító kódgenerálása révén.
 - Hasznos lehet egyszerűbb programokhoz és kezdő programozóknak.

- Pl.:

```
using System;  
Console.WriteLine("Hello, World!");
```

A .NET platform és a C# programozási nyelv

Névterek

- A névterek biztosítják a kód logikai felbontását, minden osztálynak névtérben kell elhelyezkednie
 - hierarchikusan egymásba ágyazhatóak (ponttal jelölve)
 - van egy globális névtér is (**global**)
- Névtereket használatba venni a **using** *<névtér>* utasítással lehet (az utasítás hatóköre a fájl)
 - pl.: **using System;**
using System.Collections.Generic;
 - az osztálynév előtt is megadhatjuk a névteret (így nem kell **using**), pl.:
System.Console.WriteLine("Hello, World!");

A .NET platform és a C# programozási nyelv

Névterek automatikus használata

- .NET 6 óta lehetőség van a leggyakrabban használt névterek automatikus betöltésére.
 - A `<ImplicitUsings>enable</ImplicitUsings>` direktívát kell elhelyezni a projektállományban (`csproj`).
 - Projekttípustól függően tölt be automatikusan névtereket, konzolos alkalmazásoknál például:

```
using System;  
using System.IO;  
using System.Collections.Generic;  
using System.Linq;  
using System.Net.Http;  
using System.Threading;  
using System.Threading.Tasks;
```

A .NET platform és a C# programozási nyelv

Típusok

- A nyelv három típuskategóriát különböztet meg:
 - *érték*: érték szerint kezelendő típusok, mindig másolódnak a memóriában, és a blokk végén törlődnek
 - *referencia*: biztonságos mutatókon keresztül kezelt típusok, a virtuális gép és a szemétygyűjtő felügyeli és törli őket
 - *mutató*: nem biztonságos mutatók, amelyek csak felügyeletmentes (**unsafe**) kódrészben használhatóak
- Minden típus objektumorientáltan van megvalósítva, és része a teljes származtatási hierarchiának
- A *primitív típusok* két névvel rendelkeznek: C# programozási nyelvi név és .NET könyvtárbeli megfelelő típusnév

A .NET platform és a C# programozási nyelv

Primitív típusok

- Primitív típusok:
 - logikai: **bool** (**Boolean**)
 - egész: **sbyte** (**SByte**), **byte** (**Byte**), **short** (**Int16**), **ushort** (**UInt16**), **int** (**Int32**), **uint** (**UInt32**), **long** (**Int64**), **ulong** (**UInt64**)
 - lebegőpontos: **float** (**Single**), **double** (**Double**)
 - tizedestört: **decimal** (**Decimal**)
 - karakter: **char** (**Char**)
 - objektum (minden osztály őse): **object** (**Object**)
 - szöveg: **string** (**String**)

A .NET platform és a C# programozási nyelv

Primitív típusok

- A primitív típusok is intelligensek, azaz támogatnak számos műveletet és speciális értéklekérdezést, pl.:
 - speciális értékek: `Int32.MaxValue`, `Double.NaN`, `Double.PositiveInfinity`, `String.Empty`
 - konverziós műveletek: `Double.Parse(...)`
 - karakterműveletek: `Char.ToLower(...)`,
 - szöveg műveletek: `str.Length`, `str.Find(...)`, `str.Replace(...)`
- A konstans literálok is intelligens objektumok, pl.
`10.ToString()`, `"Hello World".Substring(0, 5)`

A .NET platform és a C# programozási nyelv

Típuskezelés

- A nyelv *szigorúan típusos*, tehát minden értéknek fordítási időben ismert a típusa, és nem enged meg értékvesztést
 - nagyobb halmazra implicit típuskonverzió, kompatibilis halmazra explicit típuskonverzió használható, pl.:

```
int x = 1; double y = 2; string z;  
y = x; // implicit típuskonverzió  
x = (int)y; // explicit típuskonverzió  
z = (string)y; // hiba, nem kompatibilisek
```
 - primitív típusok konverziójához a **Convert** osztály, illetve egyéb metódusok is rendelkezésre állnak, pl.:

```
x = Convert.ToInt32(y) ;  
z = Convert.ToString(y) ; // vagy y.ToString() ;  
x = Convert.ToInt32(z) ; // vagy Int32.Parse(z) ;
```

A .NET platform és a C# programozási nyelv

Példányosítás

- Változókat bármely (nem névtér) blokkon belül létrehozhatunk a programkódban típus, név és kezdőérték megadásával
 - pl.: **Int32 myInt = 10;**
 - felhasználás előtt mindenképpen kell kezdőértéket kapnia
 - összetett típusok esetén a **new** operátort használjuk, pl.:
Stack<Int32> s = new Stack<Int32>();
 - a típusnév feloldható fordítási időben (**var**), pl.:
var myInt = 10;
 - típusok futási időben is feloldhatóak (**dynamic**), és manipulálhatóak (pl. **ExpandableObject**)
- Konstansokat a **const** kulcsszóval, konstruktorban értékül adható mezőket a **readonly** kulcsszóval adhatunk meg

A .NET platform és a C# programozási nyelv

Vezérlési szerkezetek

- *Szekvencia*: a ; tagolja az utasításokat
- *Programblokk*: { **<utasítások>** }
- *Elágazás*: lehet kétágú (**if**), illetve többágú (**switch**), utóbbinál az ágakat le kell zárni (**break**, **goto**, **return**)
- *Ciklus*:
 - számláló (**for**), előtesztelő (**while**), utántesztelő (**do ... while**)
 - bejáró (egy **IEnumerable** gyűjtemény elemein halad végig):
foreach (**<deklaráció>** in **<gyűjtemény>**)
 <utasítás>;

A .NET platform és a C# programozási nyelv

Osztályok

- A .NET platform és a C# programozási nyelv *tisztán objektumorientált*, ezért minden érték benne objektum, és minden típus egy osztály
 - az osztály lehet érték szerint (*struct*), vagy referencia szerint kezelt (*class*), utóbbi élettartama független a bloktól
 - az osztály tagjai lehetnek mezők, metódusok, események, tulajdonságok (*property*), illetve más (beágyazott) osztályok
 - a tulajdonság lényegében a lekérdező (*get*) és beállító műveletek (*set*) absztrakciója
 - minden tagnak, és az osztálynak is jelöljük a láthatóságát (**public**, **private**, **protected**, **internal**), minden tag a . operátorral érhető el

A .NET platform és a C# programozási nyelv

Osztályok

```
<láthatóság> class/struct <osztálynév> {  
    <láthatóság> <típus> <mezőnév>; // mező  
    ...  
    <láthatóság> <típus> <metódusnév>  
        ([ <paraméterek> ]) { <törzs> } // metódus  
    ...  
    <láthatóság> <típus> <tulajdonságnév> {  
        [ get { <törzs> } ]  
        [ set { <törzs> } ]  
    } // tulajdonság  
    ...  
    <láthatóság> event <delegált> <eseménynév>;  
        // esemény  
}
```

A .NET platform és a C# programozási nyelv

Osztályok felépítése

- A *mezők* típusból és névből állnak, illetve kaphatnak kezdőértéket (csak referencia szerinti osztályban)
 - a mezők alapértelmezett értéket kapnak, amennyiben nem inicializáljuk őket
- A *metódusok* visszatérési típussal (amennyiben nincs, akkor **void**), névvel és paraméterekkel rendelkeznek
 - a konstruktor neve megegyezik a típussal, a destruktort általában nem valósítjuk meg (szemétgyűjtés miatt)
 - lehetnek cím szerinti (**ref**), kimenő (**out**), alapértelmezett, tetszőleges számú (**params**) paraméterek
 - a paraméterek átadhatóak név szerint

A .NET platform és a C# programozási nyelv

Osztályok felépítése

- Pl. C++:

```
class Rational {  
    private:  
        int num;  
        int denom;  
  
    ...  
  
    public:  
        Rational(int, int) ;  
  
    ...  
};  
  
...  
Rational::Rational(int n, int d) {  
    num = n; denom = d;  
}
```

A .NET platform és a C# programozási nyelv

Osztályok felépítése

- Pl. C#:

```
struct Rational { // elemi osztály
    private Int32 num; // mező
    private Int32 denom;
    // mindenhol jelöljük a láthatóságot
    ...
    public Rational(Int32 n, Int32 d) { // metódus
        num = n;
        denom = d;
        // a deklaráció és a definíció nem
        // választható el
    }
    ...
} // nem kell a végén ; ☺
```

A .NET platform és a C# programozási nyelv

Osztályok felépítése

- A tulajdonság egy könnyítés a programozónak a lekérdező és író műveletek absztrakciójára
 - az író tulajdonság a **value** pszeudováltozó veszi át az értéket

- pl. C++:

```
class Rational {  
    ...  
    int getDenominator() { return denom; }  
    void setDenominator(int value) {  
        denom = (value == 0) ? 1 : value;  
    }  
    // publikus lekérdező és beállító művelet  
};
```

A .NET platform és a C# programozási nyelv

Osztályok felépítése

- Pl. C#:

```
struct Rational {  
    ...  
    public Int32 Denominator {  
        get { return denom; }  
        set { denom = (value == 0) ? 1 : value; }  
    } // változóhoz tartozó publikus tulajdonság  
}  
...  
Rational r = new Rational(10, 5);  
r.Denominator = 10; // a 10 kerül a value-ba
```

- külön definiálható csak lekérdező, csak beállító művelet, a láthatósági szintjük is lehet eltérő

A .NET platform és a C# programozási nyelv

Osztályok felépítése

- Amennyiben a *getter* és *setter* működése triviális, a kódunk könnyen repetitívvé válhat:

```
struct MyClass {  
    private Int32 data;  
    public Int32 Data {  
        get { return data; }  
        private set { data = value; }  
    }  
}
```

- Tulajdonsággal lehet automatikusan mezőt is létrehozni:

```
struct MyClass {  
    public Int32 Data { get; private set; }  
}
```

A .NET platform és a C# programozási nyelv

Osztályok felépítése

- Használhatjuk *set* helyett az *init* ágat is (C# 9), ez esetben az érték beállítása csak az objektum inicializálás során lehetséges.

```
struct MyClass {  
    private Int32 data;  
    public Int32 Data {  
        get { return data; }  
        init { data = value; }  
    }  
}
```

- Az objektum konstruálás során beállítható a tulajdonság, utána azonban már nem:

```
var obj = new MyClass { Data = 42 }; // ok  
obj.Data = 100; // fordítási hiba
```

A .NET platform és a C# programozási nyelv

Felsorolási típusok

- A *felsorolási típus* (**enum**) értékek egymásutánja
 - pl.:
`enum Day { Monday, Tuesday, Wednesday, ... }`
 - a hivatkozás a típusnéven át történik, pl.:
`Day day = Day.Monday; ...`
`if (day == Day.Wednesday) { ... }`
 - az értékek egész számoknak feleltethetők meg (automatikusan 0-tól sorszámozva, de ez felüldefiniálható), pl.:
`enum Day { Monday = 1, Wednesday = 3, ... }`
 - ez is egy osztály a **System** névtérben:
`public abstract class Enum : ValueType, ...`

A .NET platform és a C# programozási nyelv

Elemi osztályok

- Az *elemi osztály* (*struct*) egy egyszerűsített osztály, amely:
 - mindig érték szerint van kezelve, ezért különleges bánásmódot igényel
 - nem szerepelhet öröklődésben, de implementálhat interfészt
 - alapértelmezett konstruktora mindig létezik, amely alapértelmezettre inicializálja a változóit

- Pl.:

```
struct Rational { ... } // elemi osztály
```

```
...
```

```
Rational r = new Rational(10, 5);
```

```
Rational t = r; // r érték szerint másolódik
```

```
t.Denominator = 10; // itt r.Denominator == 5
```

A .NET platform és a C# programozási nyelv

Elemi és referencia osztályok

- A *referencia osztály* (*class*) a teljes értékű osztály, amely öröklődésben is szerepelhet
 - csak egy őse lehet, de bármennyi interfészt megvalósíthat
 - mezőit lehet közvetlenül inicializálni
 - az öröklődés miatt lehet absztrakt osztály, és szerepelhetnek benne absztrakt és virtuális elemek

- Pl.:

```
class Rational { ... } // referencia osztály
```

```
...
```

```
Rational r = new Rational(10, 5);
```

```
Rational t = r; // r cím szerint másolódik
```

```
t.Denominator = 10; // itt r.Denominator == 10
```

A .NET platform és a C# programozási nyelv

Nullable típusok

- Mint azt korábban megismertük, a C# nyelv három típuskategóriát különböztet meg:
 - *érték*: érték szerint kezelendő típusok, mindig másolódnak a memóriában, és a blokk végén törlődnek
 - *referencia*: biztonságos mutatókon keresztül kezelt típusok, a virtuális gép és a szemétgyűjtő felügyeli és törli őket
 - *mutató*: nem biztonságos mutatók, amelyek csak felügyeletmentes (**unsafe**) kódrészben használhatóak
- Az érték szerinti típusok nem vehetnek fel *null* értéket, míg a referencia szerinti típusok igen.

A .NET platform és a C# programozási nyelv

Nullable érték szerinti típusok

- A nyelvbe korán (C# 2.0, 2005) bekerült a *nullable value types* fogalma, melyet a `?` operátor segítségével használhatunk
 - az `int` nullable típusa az `int?`, teljes nevén `Nullable<int>`
- Az ilyen típusok a megszokott értékeiken túl a null értéket is felvehetik
 - `bool?` esetén a változó értéke lehet `true`, `false` vagy `null`
 - a nullable értékek default értéke mindig a `null`.
- A nullable érték szerinti típusok tárolt értékét a `Value` tulajdonságuk adja meg, annak meglétét a `HasValue` jelzi

```
if (x.HasValue) // x típusa int?
    int y = x.Value; // x.Value típusa int
```

A .NET platform és a C# programozási nyelv

Nullable referencia szerinti típusok

- Újabban (C# 8.0, 2019, .NET Core 3) a *nullable reference types* fogalma is elérhető lett a nyelvben, amely elnevezés kicsit félrevezető lehet elsőre
 - A referencia típusok eddig is felvehettek *null* értéket
 - Ha nem kezeltük, hogy egy referencia *null* értéket is felvehet és mégis dereferáltuk (kiértékeltek, metódust hívtunk rajta, stb.), az **NullReferenceException**
 - Ez az egyik legelterjedtebb futásidejű hiba típussá vált az elmúlt években.
 - Ebben hivatott segíteni a *nullable reference types*

A .NET platform és a C# programozási nyelv

Nullable referencia szerinti típusok

- Projekt szinten engedélyezhető a *nullable reference types* használata, ekkor megkülönböztetjük a **T** és a **T?** típusokat
 - Engedélyezni a .csproj fájlban tudjuk, Visual Studio 2022 és .NET 6 használata esetén már ez az alapértelmezett:
<Nullable>enable</Nullable>
 - Ha null értéket adunk egy T típusú változónak, akkor fordítási idejű figyelmeztetést kapunk
string str = null; // fordítási idejű hiba
 - Kivétel a *null-forgiving* operátor használata:
string str = null!;
 - Ha elmulasztjuk a null ellenőrzést egy **T?** típusú érték dereferálása előtt, az is fordítási idejű figyelmeztetés

A .NET platform és a C# programozási nyelv

Statikus osztályok

- Lehetőségünk van *statikus osztályok, mezők, tulajdonságok és műveletek* létrehozására a **static** kulcsszó használatával, pl.:

```
static class NumClass { // statikus osztály
    private static Int32 nr = 10;
    // statikus mező 10 kezdőértékkel
    public static Int32 Nr { get { return nr; } }
    // statikus tulajdonság
    public static void Increase() { nr++; }
    // statikus metódus
}
```

```
Console.WriteLine(NumClass.Number) // eredmény: 10
NumClass.Increase();
Console.WriteLine(NumClass.Number) // eredmény: 11
```

A .NET platform és a C# programozási nyelv

Öröklődés

- A .NET keretrendszerben az osztályok egy teljes származtatási hierarchiában vannak
 - minden osztály őse az **Object**, így megkapja annak műveleteit (pl.: **Equals (...)**, **GetHashCode ()**, **ToString ()**)
 - csak egyszeres öröklődés van, az ősosztály konstruktora és destruktora automatikusan meghívódik
 - az osztály saját tagjait a **this** kulcsszóval, az ős tagjait (beleértve a konstruktort) a **base** kulcsszóval érhetjük el
 - polimorfizmus során lehetőségünk van a típusazonosításra (**is**), valamint az explicit, illetve biztonságos típuskonverzióra (**as**)

A .NET platform és a C# programozási nyelv

Öröklődés

- Pl.:

```
class BaseClass /* : Object */ { // őosztály
    public Int32 Value;
    public BaseClass(Int32 v) { value = v; }
}
```

...

```
class DerivedClass : BaseClass { // leszármazott
    public DerivedClass(Int32 v) : base(v) { }
    //ős konstruktorának meghívása
}
```

...

```
Object o = new DerivedClass(1); // polimorfizmus
if (o is BaseClass) // típusazonosítás, konverzió
    Console.WriteLine((o as BaseClass).Value)
```

A .NET platform és a C# programozási nyelv

Öröklődés

- Öröklődés során a műveletek és tulajdonságok felüldefiniálhatóak, illetve elrejtethők
 - felüldefiniálni csak a *virtuális* (**virtual**) és *absztrakt* (**abstract**) műveleteket, tulajdonságokat lehet
 - a felüldefiniálást is jelölnünk kell (**override**)
 - a felüldefiniálhatóság lezárható (**sealed**)
 - absztrakt metódusok törzs nélküliek, absztrakt tulajdonságoknál csak azt kell jelezni, hogy lekérdezésre, vagy értékadásra szolgálnak-e
 - az ős működése elrejtethető (**new**), ekkor polimorfizmus esetén a statikus típus (jellemzően az ős) művelete érvényesül

A .NET platform és a C# programozási nyelv

Öröklődés

- Pl.:

```
class BaseClass { // őssosztály
    public void StandardMethod() {
        // lezárt (nem felüldefiniálható) művelet
        Console.WriteLine("BaseStandard");
    }
    public virtual void VirtualMethod() {
        // virtuális (felüldefiniálható) művelet
        Console.WriteLine("BaseVirtual");
    }
}
```

A .NET platform és a C# programozási nyelv

Öröklődés

- Pl.:

```
class DerivedClass : BaseClass {  
    public new void StandardMethod() {  
        // művelet elrejtés  
        Console.WriteLine("DerivedStandard");  
    }  
    public override void VirtualMethod() {  
        // művelet felüldefiniálás  
        base.VirtualMethod();  
        // a felüldefiniált művelet meghívása  
        Console.WriteLine("DerivedVirtual");  
    }  
}
```

A .NET platform és a C# programozási nyelv

Öröklődés

- Pl.:

```
DerivedClass dc = new DerivedClass();  
dc.StandardMethod(); // eredmény: DerivedStandard  
dc.VirtualMethod();  
    // eredmény:  
    // BaseVirtual  
    // DerivedVirtual  
  
...  
BaseClass bc = new DerivedClass();  
bc.StandardMethod(); // eredmény: BaseStandard  
bc.VirtualMethod();  
    // eredmény:  
    // BaseVirtual  
    // DerivedVirtual
```

A .NET platform és a C# programozási nyelv

Öröklődés

- Pl.:

```
abstract class BaseClass { // absztrakt őszosztály
    public abstract Int32 Value { get; }
    // absztrakt lekérdező tulajdonság,
    // felüldefiniálendő
    public abstract void AbstractMethod();
    // absztrakt metódus, felüldefiniálható
    public virtual void VirtualMethod() {
        Console.WriteLine(Value);
    }
}
...
BaseClass b = new BaseClass();
// hiba: absztrakt osztály nem példányosítható
```

A .NET platform és a C# programozási nyelv

Öröklődés

```
class DerivedClass : BaseClass {  
    public override Int32 Value {  
        get { return 1; }  
    } // tulajdonság felüldefiniálás  
    public sealed override void AbstractMethod() {  
        VirtualMethod();  
        Console.WriteLine(2 * Value);  
    }  
}  
...  
BaseClass bc = new DerivedClass();  
bc.AbstractMethod();  
// eredménye:  
// 1  
// 2
```

A .NET platform és a C# programozási nyelv

Interfészek

- Az *interfész* (*interface*) egy tisztán absztrakt osztály, deklarációk halmaza, amelyet az osztályok implementálnak
 - a többszörös öröklődés kiküszöbölésére szükséges

- Pl.:

```
interface IDoubleCompatible {  
    Double ToDouble(); // láthatóság, törzs nélkül  
}  
...  
struct Rational : IDoubleCompatible {  
    ...  
    // interfész megvalósítása:  
    public Double ToDouble() { ... }  
}
```

A .NET platform és a C# programozási nyelv

Attribútumok

- Az *attribútumok* (*attribute*) olyan speciális osztályok, amely elsősorban a virtuális gépnek (CLR) szolgálnak információkat (úgynevezett *metaadatokat*)
 - kiegészítik a kód deklarációit, és segítségre lehetnek a kód kezelésében, *reflexió* segítségével kezelhetők
 - a deklaráció előtt adjuk meg őket, alkalmazhatóak osztályra, metódusra, paraméterre, ...
- Pl.:

```
[Serializable] // attribútumok
[ComVisible]
class SomeClass { ... }
```


A .NET platform és a C# programozási nyelv

Előfordítási direktívák

- A nyelv tartalmaz előfordítási direktívákat, amelyek előzetesen kerülnek feldolgozásra, így lehetőséget adnak bizonyos kódsorok feltételes fordítására, hibajelzésre, környezetfüggő beállítások lekérdezésére, pl. **#if**, **#define**, **#error**, **#line**
- Mivel nem választható szét a deklaráció a definíciótól, a kód tagolását a *régiók* segítik elő, amelyek tetszőleges kódblokkokat foghatnak közre:

#region <név>

...

#endregion

- nem befolyásolják a kódot, csupán a fejlesztőkörnyezetben érhetőek el

A .NET platform és a C# programozási nyelv

Megjegyzések

- Az egyszerű *megjegyzések* a fordításkor törlődnek
 - sor végéig tartó: `// megjegyzés`
 - tetszőleges határok között: `/* megjegyzés */`
- A *dokumentációs megjegyzések* fordításra kerülnek, és utólag előhívhatóak a lefordított tartalomból
 - osztályok és tagjaik deklarációjánál használhatjuk
 - célja az automatikus dokumentálás elősegítése és a fejlesztőkörnyezetben azonnal segítség megjelenítése
 - a `///` jeltől a sor végéig tart, belül XML blokkok adhatóak meg, amelyek meghatározzák az információ jelentését

A .NET platform és a C# programozási nyelv

Megjegyzések

- pl.:

```
/// <summary>
/// Racionális szám típusa.
/// </summary>
/// <remarks>Két egész szám hányadosa.</remarks>
struct Rational {
    ...
    /// <summary>
    /// Racionális szám példányosítása.
    /// </summary>
    /// <param name="n">Számológép.</param>
    /// <param name="d">Nevező.</param>
    public Rational(Int32 n, Int32 d) { ... }
    ...
}
```

A .NET platform és a C# programozási nyelv

Kivételkezelés

- A .NET keretrendszerben minden hiba kivételként jelenik meg
 - a kivétel általános osztálya az **Exception**, csak ennek vagy leszármazottjának példánya váltható ki
 - kivételt kiváltani a **throw** utasítással tudunk:
`throw new <kivétel típusa>(<paraméterek>);`
 - kivételt kezelni egy kivételkezelő (**try-catch-finally**) szakasszal tudunk:
`try { <kivételkezelt utasítások> }
catch (<elfogott kivétel típusa>){
 <kivételkezelő utasítások>
}
finally { <mindenképp lefuttatandó utasítások> }`

A .NET platform és a C# programozási nyelv

Kivételkezelés

- Pl.:

```
class WorkingClass {  
    public void DoSomething(Int32 number)  
    {  
        if (number < 1)  
            throw new ArgumentOutOfRangeException();  
        // kivétel kiváltása (a paraméter hibás  
        // tartományban van)  
  
        ...  
  
        throw new Exception("Too lazy...");  
        // kivétel kiváltása (üzenettel)  
    }  
    public void Finish() { ... }  
}
```

A .NET platform és a C# programozási nyelv

Kivételkezelés

- Pl.:

```
WorkingClass wc = new WorkingClass();  
try // kivételkezelő blokk  
{  
    wc.DoSomething(42);  
}  
// a kivételt típustól függően kezelhetjük  
catch (ArgumentOutOfRangeException ex)  
{ ... }  
// az Exception típusú kivételt nem kezeljük le  
finally {  
    wc.Finish(); // de ez mindenképpen lefut  
}
```

A .NET platform és a C# programozási nyelv

Generikus típusok

- Generikus programozásra futási időben feldolgozott sablon típusok (*generic*-ek) segítségével van lehetőség
 - osztály, metódus és delegált lehet sablonos, a sablon csak osztály lehet
 - a sablon fordításra kerül, és csak a futásidejű fordításkor helyettesítődik be a konkrét értékre
 - pl.:

```
struct Rational<T> {  
    private T nom; // használható a T típusként  
    ...  
    public Rational(T n, T d) { ... }  
    ...  
}
```


A .NET platform és a C# programozási nyelv

Generikus típusok

...

```
Rational<SByte> r1 = new Rational<SByte>(10,5);  
Rational<Int64> r2 = new Rational<Int64>(10,5);  
// különböző értékkészletű racionálisok
```

- A szigorú típuskezelés miatt a sablonra csak az **Object**-ben értelmezett műveletek használhatóak, ezt a műveletkört növelhetjük megszorításokkal (**where**)

- pl.:

```
class Rational<T> where T : struct, IComparable,  
                        IFormattable, IConvertible { ...  
    // T elemi osztály, amire használható a fenti  
    // interfészek összes művelete  
}
```

A .NET platform és a C# programozási nyelv

Tömbök

- A tömbök osztályként vannak megvalósítva (**System.Array**), de egyszerűsített szintaxissal kezelhetők, pl.:

```
Int32[] myArray = new Int32[10]; // létrehozás  
myArray[0] = 1; // első elem beállítása
```

- referencia szerint kezeltek, méretnek változó is megadható, az értékek inicializálhatóak, pl.:

```
Int32[] myArray = new Int32[] { 1, 2, 3, 4 };  
// a tömb 4 hosszú lesz
```

- akár több dimenziósak is lehetnek, pl.:

```
Int32[,] myMatrix = new Int32[10, 5]; // mátrix  
myMatrix[0, 0] = 1; // első sor első eleme  
Double[, ,] myMatrix3D = new Double[10, 5, 10];  
// 3 dimenziós mátrix
```

A .NET platform és a C# programozási nyelv

Tömbök

- Fontosabb műveletei:
 - hossz lekérdezés (**Length**, **LongLength**, **GetLength**)
 - dimenziószám lekérdezése (**Rank**)
- Statikus műveletként számtalan lehetőségünk van, pl.:
 - másolás (**Copy**), átméretezés (**Resize**)
 - rendezés (**Sort**), fordítás (**Reverse**)
 - lineáris keresés (**Find**, **IndexOf**, **LastIndexOf**), bináris keresés (**Binary Search**)
- A tömböknél (és más gyűjteményeknél) alkalmazott indexelő művelet megvalósítható saját típusokra is (paraméteres tulajdonságként)

A .NET platform és a C# programozási nyelv

Gyűjtemények

- A gyűjtemények a **System.Collections** névtérben találhatóak, a legtöbb gyűjteménynek van általános és sablonos változata is, pl.:
 - dinamikus tömbök: **ArrayList**, **List<T>**, **SortedList**, **SortedList<Key, Value>**
 - láncolt listák: **LinkedList<T>**
 - verem: **Stack**, **Stack<T>**
 - sor: **Queue**, **Queue<T>**
 - asszociatív tömb: **Hashtable**, **Dictionary<Key, Value>**, **SortedDictionary<Key, Value>**
 - halmaz: **HashSet<T>**, **SortedSet<T>**

A .NET platform és a C# programozási nyelv

Gyűjtemények

- A nem sablonos gyűjteményekbe bármilyen elemeket helyezhetünk, de használatunk nem javasolt, kompatibilitási okokból érhetőek el
- A dinamikus tömbök indexelhetőek, és változtatható a méretük (bárhova beszúrhatunk, bárhonnan törölhetünk), pl.:

```
List<Int32> intList = new List<Int32>();  
    // üres tömb létrehozása  
intList.Add(1); ... // elemek hozzáadása  
intList.Insert(0, 100); // beszúrás az elejére  
...  
intList.Remove(100); // elem törlése  
for (Int32 i = 0; i < intList.Count; i++)  
    Console.WriteLine(intList[i]);  
    // lekérdezés  
intList.Clear(); // kiürítés
```

A .NET platform és a C# programozási nyelv

Lambda-kifejezések

- A *lambda-kifejezések* (*lambda-expressions*) funkcionális programozásból átvett elemek, amelyek egyszerre függvényként és objektumként is viselkednek
- A λ -kifejezést az \Rightarrow operátorral jelöljük, tőle balra a paraméterek, jobbra a művelet törzse írható le, pl.:
`a => a * a // négyzetre emelés`
`x => x.Length < 5 // 5-nél rövidebb szövegek`
`(x, y) => x + y; // összeadás`
`() => 5; // konstans 5`
- A λ -kifejezést elmenthetjük változóként is, típusa a sablonos `Func<...>` lesz, pl.:
`Func<String, Boolean> lt5 = x => (x.Length < 5);`

A .NET platform és a C# programozási nyelv

Lambda-kifejezések

- Az eltárolt kifejezés bármikor futtathatjuk, mint egy függvényt, pl.:

```
Boolean l = lt5("Hello!"); // l hamis lesz
```

- A λ -kifejezések tetszőlegesen összetett utasítássorozatot is tartalmazhatnak, nem csak egy kifejezés kiértékelését, ekkor a tartalmat blokkba kell helyezni, pl.:

```
Func<Int32, Int32> pow2 = x => {  
    x = x * x;  
    return x;  
};
```

- A λ -kifejezések speciális típusa az akció (**Action**), amely egy visszatérési érték nélküli tevékenység, pl.:

```
Action<Int32> write = value => {  
    Console.Write(value); };
```


A .NET platform és a C# programozási nyelv

Nyelvbe ágyazott lekérdezések

- A *nyelvbe ágyazott lekérdezések* (*Language Integrated Query*) célja, hogy objektumorientált környezetben valósíthassunk meg lekérdező utasításokat
 - hasonlóan a relációs adatbázisok SQL nyelvéhez
 - pl.:

```
List<Person> pList = new List<Person> { ... };  
var pQuery = from p in pList // honnan  
              where p.Age >= 18 // feltétel  
              select p.Name; // mit
```
 - az eredmény egy gyűjtemény (**IEnumerable**) lesz, és a kifejezés csak akkor értékelődik ki, amikor azt bejárjuk (*késleltetett végrehajtás*)
 - mi lesz az elemek típusa az eredményben?

A .NET platform és a C# programozási nyelv

Nyelvbe ágyazott lekérdezések

- A nyelvbe ágyazott lekérdezések mögött λ -kifejezésekkel dolgozó metódusok találhatóak, amelyek bármilyen gyűjteményre futtathatóak (akár külön-külön is)
 - pl.:

```
var pQuery = pList // honnan
                .Where(p => p.Age >= 18) // feltétel
                .Select(p => p.Name); // mit
```
- a metódusok úgynevezett *bővítő metódusként* (*extension method*) definiáltak, amelyek elérhetőek a **System.Linq** névtérben
- bonyolultabb lekérdezések is megvalósíthatóak (pl. unió, csoportosítás, összekapcsolás, rendezés, ...)

A .NET platform és a C# programozási nyelv

Nyelvbe ágyazott lekérdezések

- Pl.:

```
Int32[] s1 = { 1, 2, 3 }, s2 = { 2, 3, 4 };  
Int32 sum = s1.Sum(); // számok összege  
Int32 evenCount = s1.Sum(x => x % 2 == 0 ? 1 : 0);  
    // megadjuk, mit összegezzen, így a páros  
    // számok számlálása lesz  
var union = s1.Union(s2);  
    // két gyűjtemény uniója: { 1, 2, 3, 4 }  
var evens = union.Select(x => x % 2 == 0);  
    // páros számok kiválogatása  
Int32 evenCount =  
    s1.Union(s2).Sum(x => x % 2 == 0 ? 1 : 0);  
    // unió, majd a páros számok számlálása
```

A .NET platform és a C# programozási nyelv

Metódusok és tulajdonságok megadása kifejezés törzzsel

- A metódusok és tulajdonságok implementációját *kifejezés törzzsel* (*expression body*) is megadhatjuk, ez rövidebb eljárásoknál lehet hasznos.
- Pl.:

```
struct Rational {  
    // mezők ...  
  
    public Int32 Denominator {  
        get { return denom; }  
        set { denom = (value == 0) ? 1 : value; }  
    } // nevező tulajdonsága  
  
    public Double ToDouble() {  
        return num / denom;  
    } // konvertálás lebegőpontos értékre  
}
```

A .NET platform és a C# programozási nyelv

Metódusok és tulajdonságok megadása kifejezés törzzsel

- Így a funkcionális nyelvekből ismerős szintaxist kapunk.

```
struct Rational {  
    // mezők ...  
  
    public Int32 Denominator {  
        get => denom;  
        set => denom = (value == 0) ? 1 : value;  
    } // nevező tulajdonsága  
  
    public Double ToDouble() => num / denom;  
    // konvertálás lebegőpontos értékre  
}
```

A .NET platform és a C# programozási nyelv

C# verziótörténet

C# 1.0: 2002

C# 2.0: 2005

C# 3.0: 2007

C# 4.0: 2010

C# 5.0: 2012

C# 6.0: 2015

C# 7.0: 2017

C# 8.0: 2019

C# 9.0: 2020

C# 10.0: 2021

C# 11.0: 2022

C# 12.0: 2023

- Nyelvi elemek: <https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history>
- Visual Studio 2022-ben az alapértelmezett nyelvi verziók:
 - .NET Framework: C# 7.3
 - .NET Core 3.1: C# 8.0
 - .NET Standard 2.1: C# 8.0
 - .NET 8: C# 12.0