

Alkalmazásfejlesztés II. Gyakorlat

2023/2024
I. félév

Utoljára frissítve: 2023.10.04.



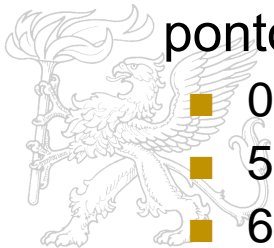
Elérhetőség

- ▶ Gyakorlatvezető: Dr. Márkus András
- ▶ E-mail: markusa@inf.u-szeged.hu
- ▶ CooSpace kurzusfórum / üzenet
- ▶ Szoba: Árpád téri épület fsz. #14
- ▶ Honlap: <https://www.inf.u-szeged.hu/~markusa/>



Követelmények

- ▶ Hivatalos: Coospace „Követelmények” fül
- ▶ A gyakorlatok látogatása kötelező
- ▶ A megengedett igazolatlan hiányzások maximális száma 2, ennél több igazolatlan hiányzás esetén a hallgató teljesítménye nem értékelhető
- ▶ Folyamatos számonkérés
 - 6 teszt: maximum 20 pont szerezhető (3-3-3-4-3-4)
 - 3 projekt munka: maximum 30 pont szerezhető (10-10-10)
 - 3 ZH: maximum 50 pont szerezhető (10-20-20)
- ▶ Javítási, pótlási lehetőség nincs, az érdemjegy a megszerzett pontok alapján lesz meghatározva:
 - 0-49 elégtelen,
 - 50-62 elégséges,
 - 63-75 közepes,
 - 76-88 jó,
 - 89-100 jeles

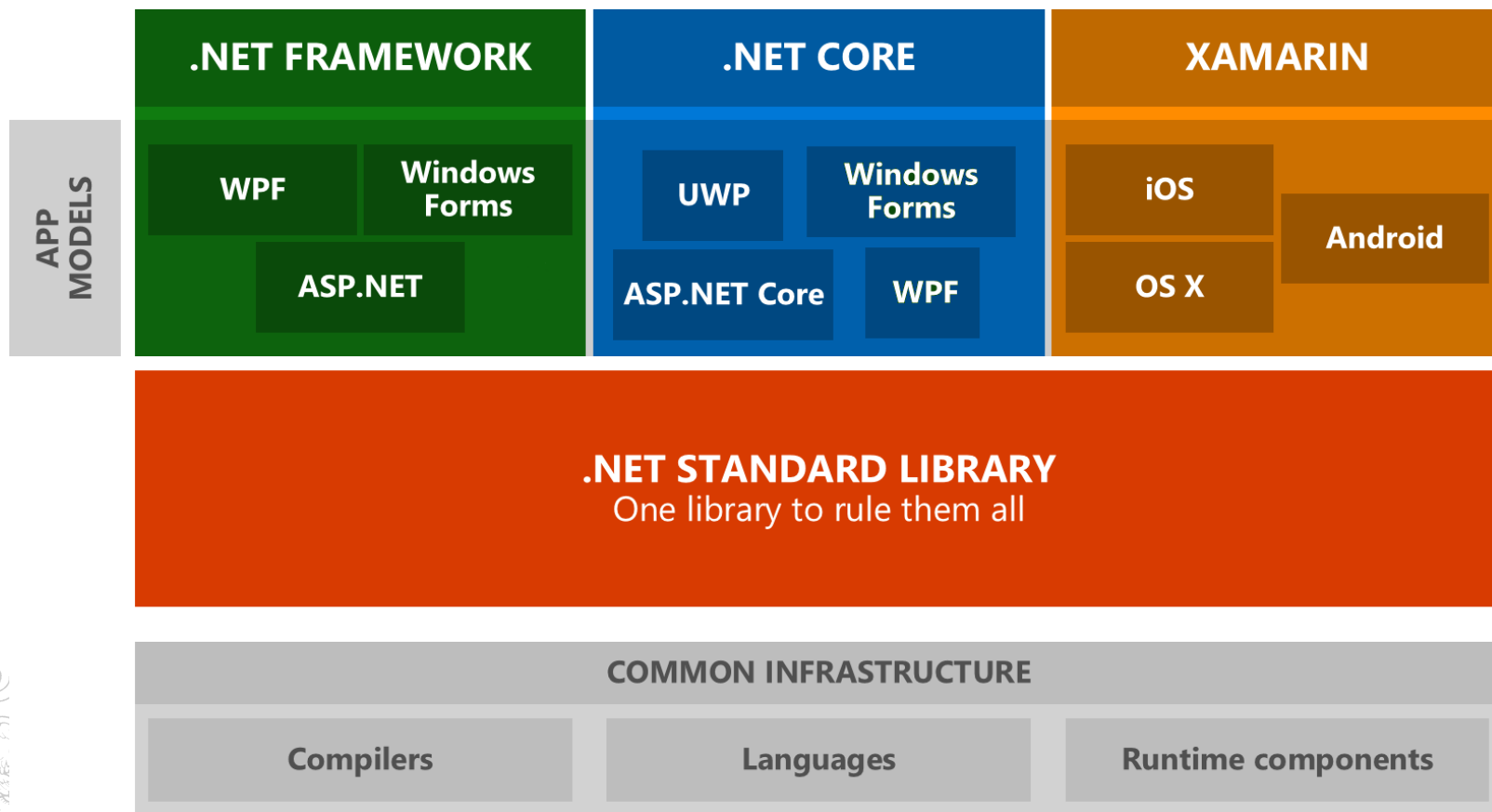


Időbeosztás

Gyakorlat	Hétfő	Szerda	Péntek
1. (09.13/15.)			
2. (09.20/22.)			
3. (09.27/29.)		Teszt 1	
4. (10.04/06.)	Projekt leadás 1	Teszt 2	
5. (10.11/13.)		C# konzolos alkalmazás ZH	
6. (10.18/20.)			
7. (10.25/27.)		Teszt 3	
8. (11.01/03.)		Mindenszentek	
9. (11.08/10.)	Projekt leadás 2	Teszt 4	
10. (11.15/17.)		WinForms alkalmazás ZH	
11. (11.22/24.)			
12. (11.29/12.01.)		Teszt 5	
13. (12.06/08.)	Pojekt leadás 3	Teszt 6	
14. (12.13/15.)		ASP.NET alkalmazás ZH	



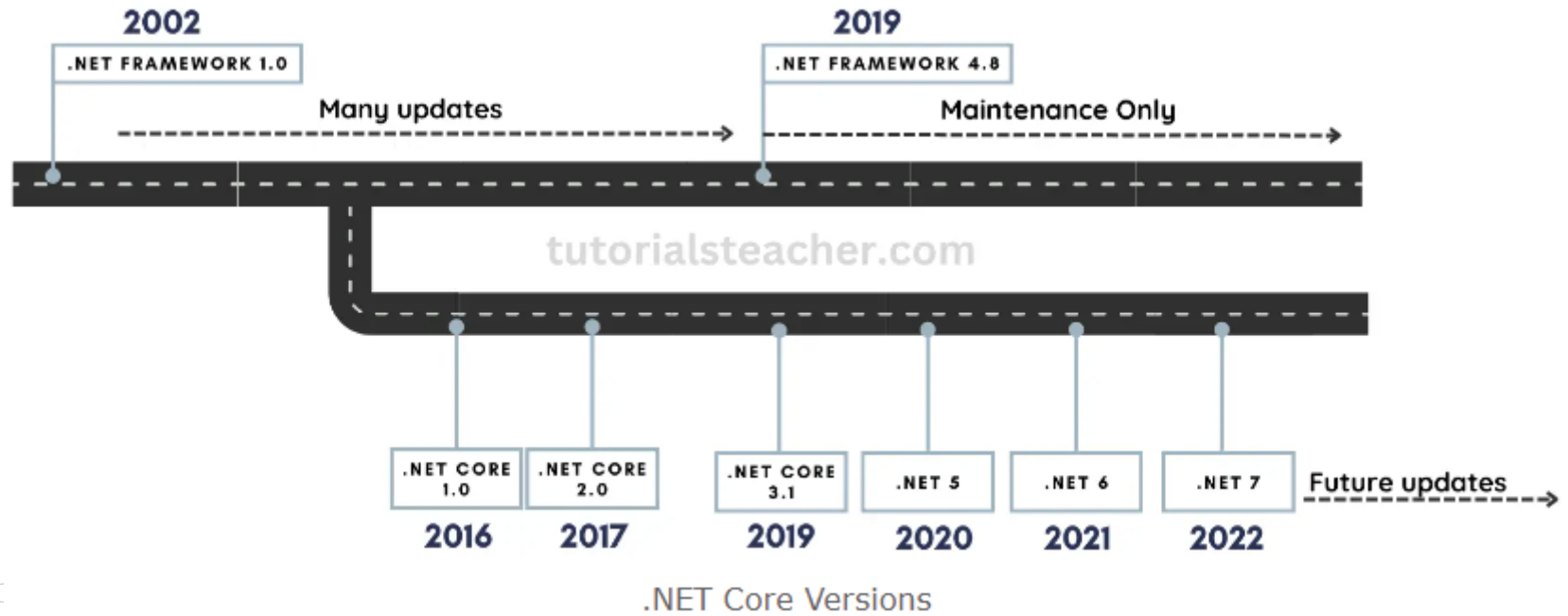
.NET architektúra elemei



- ▶ A .NET-re nyelvek és implementációk nagy családjaként lehet tekinteni



.NET Framework és .NET Core verziók

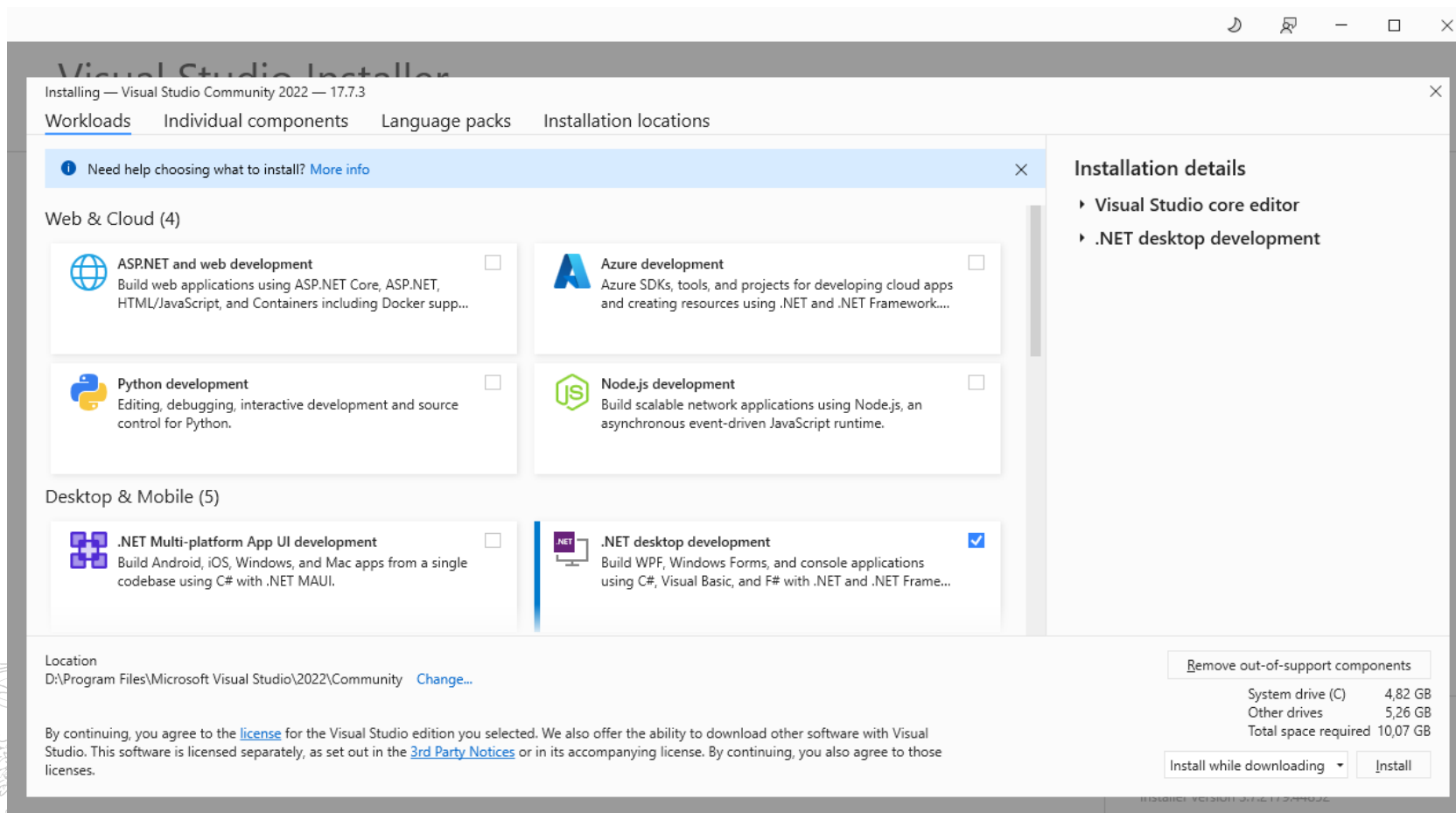


Visual Studio

- ▶ <https://visualstudio.microsoft.com/downloads/>
- ▶ 2022 Community Edition
- ▶ Visual Studio Installer: testreszabható a telepíteni kívánt fejlesztőkörnyezet (lásd köv. dia)
- ▶ Amennyiben a későbbiekben szükség lenne másra is, a VS Installer megnyitásával módosítható a már telepített környezet
- ▶ <https://dotnet.microsoft.com/en-us/download/visual-studio-sdks>
- ▶ Sikeres telepítés ellenőrzése parancssorban: dotnet –info
- ▶ <https://dotnet.microsoft.com/en-us/download/visual-studio-sdks>



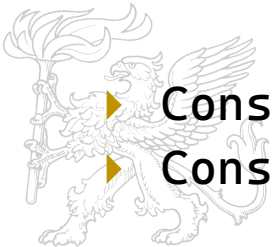
Visual Studio



- ▶ Az első indítás során egy bejelentkező ablak ugrik fel, a bejelentkezés opcionális

Hello World

- ▶ A projekt egy Solution-ben helyezkedik el
- ▶ Konténer, mely több projektet tartalmazhat
 - .sln kiterjesztés
- ▶ Projekt struktúra tartalmazza a projekt függőségeit definiáló fájlt
 - .csproj
- ▶ Az osztályok névterekbe vannak sorolva
- ▶ Namespace-ek (~Java package-ek)
- ▶ Using (~Java import kulcsszó)
- ▶ `Console.WriteLine`
- ▶ `Console.ReadLine`
- ▶ String interpoláció



NuGet csomagkezelő rendszer

- ▶ Lehetővé teszi újrafelhasználható kódok csomagokba szervezését és megosztását
- ▶ Amennyiben egy csomag telepítve lett a projektben, annak publikus API-ja elérhető a kódból
- ▶ Serilog.Sinks.Console
- ▶ <https://github.com/serilog/serilog/wiki/Configuration-Basics>

```
using Serilog;
```



```
var log = new LoggerConfiguration()  
    .WriteTo.Console()  
    .MinimumLevel.Debug()  
    .CreateLogger();
```

```
log.Debug(...);
```

Debugging

- ▶ Debug / Start Debugging (F5)
- ▶ Ilyen állapotban a fejlesztői környezet Autos ablakában megtekinthetők a blokkban látható változók
- ▶ Illetve a Call Stack is, azaz, hogy a program milyen úton jutott el a megállási pontba
- Step Into: A következő utasításra ugrás.
 - Ha ez függvényhívás, akkor a függvényhívás törzsébe lép
- Step Over: A következő utasításra ugrás.
 - Ha ez függvényhívás, akkor a törzset lefuttatja és a jelenlegi kontextus következő utasításával folytatódik
- Step Out: A következő utasításra ugrás.
 - A jelenlegi függvényből kilépve (a függvény maradék törzsét lefuttatja és a függvényhívás helyét kapjuk meg)



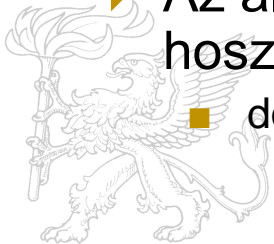
Parancssoros fejlesztés

- ▶ Új konzolos alkalmazás létrehozása
 - `dotnet new console -n hello`
- ▶ Az alkalmazás build-elése (végtermék többek között egy .dll fájl)
 - `dotnet build`
- ▶ Az alkalmazás futtatása (build-t utasítást kihagyjuk, a run elvégzi)
 - `dotnet run`
- ▶ .dll közvetlen futtatása (ha abban a könyvtárban vagyunk)
 - `dotnet hello.dll`
- ▶ NuGet csomag hozzáadása
 - `dotnet add package Serilog.Sinks.Console`
- ▶ <https://learn.microsoft.com/en-us/dotnet/core/tools/>



Még egy kis parancssoros fejlesztés..

- ▶ Az előző build-elés kimenetét törli
 - dotnet clean
- ▶ Törli a hivatkozott csomagot projektből
 - dotnet remove package Serilog.Sinks.Console
- ▶ A .csproj fájlban található függőségeket megkeresi, szükség esetén letölti
 - dotnet restore
- ▶ Az alkalmazást és a függőségeit egy mappába csomagolja hosztoláshoz (build-eli is a csomagolás előtt)
 - dotnet publish



Típusok C#-ban

- ▶ A .NET minden típusa direkt vagy indirekt módon a System.Object nevű típusból származik
- ▶ Referencia típus (heap-en)
 - Stringek, class, interface, delegate, tömbök
- ▶ Érték típus (stack-en)
 - Numerikus típusok (int, byte, double, stb.), bool, char, enum, struct
 - Metóduson belül, lokálisan deklarált értéktípusok a verembe kerülnek
 - A referenciatípuson belül adattagként deklarált értéktípusok pedig a halomban foglalnak helyet
- ▶ Pointer típus
 - Limitáltan, de támogatja a C#
 - *Unsafe* kód: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/unsafe-code>



Konstansok

- ▶ A *const* típusmódosító segítségével egy objektumot megváltoztathatatlaná tehetünk
- ▶ Fordítási időben ki kell tudnia értékelni a fordítónak
- ▶ A *readonly* módosítóval ellátott objektumok is módosíthatatlanok, viszont itt a deklaráció és a definíció szétválik, a definíciónak elég a konstruktorban megtörténnie

```
class Program
{
    readonly int num;

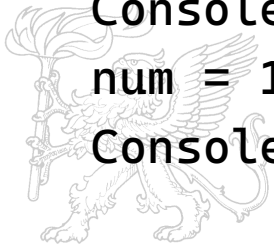
    Program()
    {
        num = 11;
    }
    static void Main(string[] args)
    {
        Program p = new Program();
        p.num++; // fordítási hiba.
    }
}
```



Implicit és Null(able) típus

- ▶ A nyelv szigorúan típusos, az implicit típusok csak kényelmi szolgáltatásként érthetők el a *var* kulcsszó használatával
- ▶ A referenciatípusok az inicializálás előtt automatikusan null értéket vesznek fel
- ▶ Az értéktípusok pedig az általuk tárolt adatot reprezentálják, ezért ők nem vehetnek fel null értéket
- ▶ Nullable segítségével mégis kaphatnak null értéket

```
int? num = null;  
Console.WriteLine(num.HasValue);  
num = 10;  
Console.WriteLine(num.Value);
```




További érdekes típusok

- ▶ Dynamic type: <https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/interop/using-type-dynamic>
- ▶ Haszna különösen a script alapú nyelvekkel való együttműködésben rejlik

```
dynamic dyn = 7;  
Console.WriteLine(dyn);  
dyn = "a string";  
Console.WriteLine(dyn);
```

- ▶ Névtelen típus: <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/anonymous-types>



```
var anonym =  
    new { id = 0, program = new Program() };  
anonym.program.num = 5; //feltételezve, hogy a Program osztálynak létezik num adattagja  
Console.WriteLine(anonym.program.num);
```

Osztályok

- ▶ C# nem támogatja a többszörös öröklődést, így egy osztálynak csak egy őse lehet, viszont több interfészt is implementálhat
- ▶ Osztályok alapértelmezett láthatósága: internal
- ▶ Az osztályokban létrehozott metódusok adattagok alapértelmezett láthatósága: private
- ▶ <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers>



Caller's location	public	protected internal	protected	internal	private protected	private
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

Interfész, absztrakt osztály

- ▶ *sealed*
 - Egy osztályt lezárhatunk, azaz megtilthatjuk, hogy új osztályt származtassunk belőle
- ▶ *interface*
 - Az interfész magában meghatároz egy függvényhalmazt, melyet az interfészt implementáló osztálynak kell megvalósítania
 - Egy osztály több interfészt is megvalósíthat
- ▶ *abstract*
 - Nem példányosítható az osztály
 - absztrakt metódusnak nem lehet definíciója,
 - a leszármazottaknak definiálnia kell az öröklött absztrakt metódusokat.
 - Absztrakt osztály tartalmazhat nem absztrakt metódusokat is
 - Az öröklött absztrakt metódusokat az override kulcsszó segítségével tudjuk definiálni
- ▶ Az öröklődésnek kell az első helyre kerülnie, majd az interfészlista következik



```
class Camel : SuperAnimal, IAnimal {}
```

Property-k

- ▶ Egy speciális adattagja az osztálynak, mely lehetővé teszi privát változók kontrollált hozzáférését
- ▶ Első ránézésre adattagok, viszont speciális metódusok legtöbbször publikus láthatósággal ellátva
- ▶ Minden tulajdonság rendelkezhet ún. getter és setter blokkal
- ▶ Visual Studio-ban egyszerűen kiegészíthető a kód:
 - prop + TAB + TAB

```
public string Color { get; set; }
```

- propfull + TAB + TAB

```
private int power;
public int Power
{
    get { return power; }
    set { power = value; }
}
```

Virtuális metódusok

- ▶ Az őosztályban deklarált virtuális (vagy polimorfikus) metódusok viselkedését a leszármazottak átdefiniálhatják
- ▶ Virtuális metódust a virtual kulcsszó segítségével deklarálhatunk
- ▶ A leszármazott osztályokban az override kulcsszóval mondjuk meg a fordítónak, hogy szándékosan hoztunk létre az őosztályéval azonos szignatúrájú metódust, és a leszármazott osztályon ezt kívánjuk használni mostantól
- ▶ Egy override-dal jelölt metódus automatikusan virtuális is lesz, így az ő leszármazottai is átdefiniálhatják a működését

▶ Ős:

```
protected virtual void HeroWelcome() {}
```

▶ Leszármazott:

```
protected override void HeroWelcome() {}
```



Nevesített + alapértelmezett paraméterek

- ▶ Az alapértelmezett paramétereket lehetővé teszik, hogy paramétereknek alapértelmezett értékeket adjunk, ezáltal nem kell kötelezően megadnunk minden paramétert a metódus hívásakor

```
public Camel(string color = "red", int power = 1)
{
    Color = color;
    Power = power;
}
```

- ▶ Ezt követően a példányosítás ennyi is lehet:

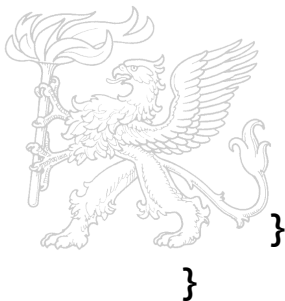
```
Camel camel2 = new Camel(color: "yellow");
```



Extension Method

- ▶ Már létező típusokhoz új metódusokat tudunk hozzáadni anélkül, hogy azokat módosítanánk vagy származtatnánk belőlük
- ▶ Minden esetben egy statikus osztály statikus metódusa kell, hogy legyen

```
public static class IntExtension
{
    public static void SajtBurger(this int i, int value)
    {
        if(i > value)
        {
            Console.WriteLine("Bezart a BK..");
        }
        else
        {
            Console.WriteLine($"Jar a {value} sajtburesz..");
        }
    }
}
```



Paraméterátadás

- ▶ A paramétereket átadhatunk érték és cím szerint is
- ▶ Előbbi esetben teljesen új példány jön létre az adott osztályból, amelynek értékei megegyeznek az eredetiével
- ▶ A másik esetben egy az objektumra mutató referencia adódik át, tehát az eredeti objektummal dolgozunk
- ▶ Az értéktípusok alapértelmezetten érték szerint adódnak át, míg a referenciatípusoknál a cím szerinti átadás az előre meghatározott viselkedés

```
public void Swap(ref int x, ref int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

```
void DoMagic(ref Program p)
{
    p = new Program();
}
```

- ▶ A cím szerinti átadás másik formájában nem inicializált paramétert is átadhatunk

```
void DoAnotherMagic(out Program p)
{
    p = new Program();
    p.num = 111;
}
```


Foreach & Yield

- ▶ A yield kifejezés lehetővé teszi, hogy egy ciklusból olyan osztályt generáljon a fordító, amely megvalósítja az IEnumerable interfészt
- ▶ Ezáltal használható legyen pl. a foreach ciklussal

```
using System.Collections;

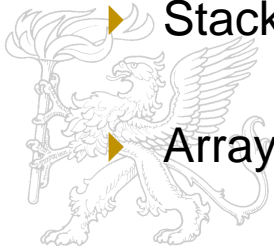
static public IEnumerable EnumerableMethod(int max)
{
    for (int i = 0; i < max; ++i)
    {
        if (i > 5)
        {
            yield break;
        }
        yield return i;
    }
}

foreach (var i in EnumerableMethod(10)) { }
```



Kollekciók

- ▶ a System.Collections.Generic névtérben található
- ▶ <https://learn.microsoft.com/en-us/dotnet/standard/collections/commonly-used-collection-types>
- ▶ List <T>: objektumok listáját tárolja, támogatja az indexelést, keresést, rendezést és a lista módosítását
- ▶ Dictionary <TKey, TValue>: kulcs-érték párokat tárol
- ▶ Queue: FIFO lista
- ▶ Stack: LIFO lista
- ▶ Array, ArrayList, LinkedList<T>, stb.



Kivételkezelés

- ▶ Kivételkezelés a már megszokott try-catch-finally segítségével valósítható meg
- ▶ A finally blokkra nem menedzselt kód esetén van szükség - ez az amit a GC nem old meg helyettünk
 - Pl. adatbázis műveletek során a kapcsolat bezárása

```
int[] array = new int[2];
try
{
    array[2] = 10;
}
catch (IndexOutOfRangeException e)
{
    Console.WriteLine(e.Message);
}
finally
{
    Console.WriteLine("Itt elhagyható..");
}
```



- ▶ Magunk is dobhatunk kivételt (throw) vagy magunk is készíthetünk a System.Exception-ből származtatva

Delegate

- ▶ A delegate olyan típus, amely egy vagy több metódusra hivatkozik
- ▶ Egy delegate deklarációjánál megadjuk, hogy milyen szignatúrával rendelkező metódusok megfelelőek
- ▶ Delegate nem deklarálható blokkon belül
- ▶ Általános használatuk: függvények átadása más függvényeknek argumentumként, így callback függvények hozhatók létre
- ▶ Mivel a létrehozott delegate egy objektum, így átadható függvényeknek, mint paraméter

```
delegate int TestDelegate(int x);
```

```
TestDelegate testDelegate = Pow;
```

```
Callback(TestDelegate del, int x);
```



Delegate (folyt.)

- ▶ A delegate-ekhez egynél több metódust is hozzáadhatunk a += és + operátorokkal, valamint elvehetjük őket a -= és - operátorokkal
- ▶ A delegate hívásakor a listáján lévő összes metódust meghívja a megadott paraméterre

```
delegate void AnotherDel();
```

```
AnotherDel anotherDel = null;
```

```
anotherDel += Another;
```

```
anotherDel += Another;
```

```
anotherDel -= Another;
```

```
anotherDel += Another;
```

```
anotherDel();
```



Func & Action

- ▶ A Func egy speciális delegate, melynek lehetnek input paraméterei (akár nulla, de több is) és egy visszatérési értéke van

```
Func<int, int, int> add = Sum;
```

- ▶ Hasonló az előző szekcióhoz, viszont az Action-nak pontosan egy bemenő paramétere van és nincs visszatérési értéke

```
Action<string> logger = Logger;
```



Típusellenőrzés és konverzió

- ▶ Kétféleképpen konvertálhatunk: implicit és explicit módon
- ▶ Ellenőrzött konverzió: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/checked-and-unchecked>

```
int x = 10;
```

```
long y = x;
```

```
int x = 300;
```

```
byte y = (byte)x;
```

```
Test t = new Test();
```

```
Type type = t.GetType();
```

- ▶ Az *is* operátort futásidejű típus-lekérdezésre használjuk

```
double szam = 6.76;
```

```
Console.WriteLine(szam is int);
```

- ▶ Párja az *as*, az ellenőrzés mellett egy explicit típuskonverziót is végrehajt. Ezzel az operátorral csakis referenciatípusra konvertálhatunk, értéktípusra nem

```
object b = "Hello";
```

```
string bb = b as string;
```

LINQ

- ▶ A Language-Integrated Query, vagyis a LINQ lehetővé teszi a kollekciókban (bármiben, ami az IEnumerable interfészt implementálja) történő keresést, rendezést és csoportosítást
- ▶ <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/standard-query-operators-overview>

```
int[] scores = new int[] { 1, 21, 34, 97, 92, 81, 60 };
```

```
IEnumerable<int> scoreQuery =  
    from score in scores  
    where score > 80  
    select score;
```



Amiről még lehetne szó..

- ▶ Boxing / unboxing: <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/types/boxing-and-unboxing>
- ▶ Operator overloading: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/operator-overloading>
- ▶ Expression Trees: <https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/expression-trees/>
- ▶ Base kulcsszó: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/base>
- ▶ Indexers: <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/>
- ▶ API kereső: <https://learn.microsoft.com/en-us/dotnet/api/?view=net-7.0>



Névtelen metódus és Lambda

- ▶ Névtelen metódus létrehozása a delegate kulcsszóval történik
- ▶ Minden lambda kifejezés delegate-té alakítható (Action, Func)

```
public delegate int Test(int x);
```

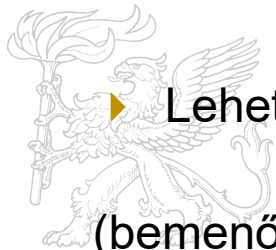
```
Test test = delegate (int x) {}
```

```
Test masik = (x) => {}
```

- ▶ lambda operátor: =>
- ▶ Lehetséges formái a nyelvben:

(bemenő-paraméterek) => kifejezés

(bemenő-paraméterek) => { több utasítás }



LINQ (folyt.)

- ▶ LINQ lehetővé teszi adatforrások kezelést új „nyelv” megtanulása nélkül (pl.SQL)
- ▶ A LINQ lekérdezések erősen típusosak, vagyis a legtöbb hibát még fordítási időben el tudjuk kapni és kijavítani
- ▶ Amit használunk a LINQ-hoz:
 - Extension method (IEnumerable interfészeket egészíti ki)
 - Lambda kifejezések
 - Inicializálók: deklarálásával egy időben beállíthassuk a tulajdonságokat
 - Implicit típus: nehéz előre megadni az eredménylista típusá
 - Névtelen típus: sok esetben nincs szükségünk egy objektum minden adatára, ilyenkor feleslegesen foglalná egy lekérdezés eredménye a memóriát



LINQ: kiválasztás

```
int[] scores = new int[] { 1, 21, 34, 1, 97, 92, 81, 60 };
```

```
IEnumerable<int> res1 =  
    from score in scores  
    select score + 1;
```

- ▶ Vagy egyszerűbben, lehetőség szerint ne keverjük a kétfajta írásmódot:

```
var res2 = scores.Select( number => number + 1 );
```



LINQ: where

```
int[] scores = new int[] { 1, 21, 34, 1, 97, 92, 81, 60 };
```

```
IEnumerable<int> res1 =  
    from score in scores  
    where score > 80  
    select score;
```

► Vagy:

```
var res2 = scores.Where(number => number > 80);
```



LINQ: order by

```
int[] scores = new int[] { 1, 21, 34, 1, 97, 92, 81, 60 };
```

```
IEnumerable<int> res1 =  
    from score in scores  
    where score > 80  
    orderby score descending // ascending  
    select score;
```

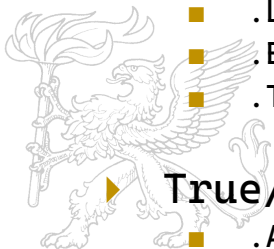
► Vagy:

```
var res2 = scores.Where(number => number > 80)  
    .OrderByDescending(number => number);  
// .OrderBy( number => number);
```



LINQ: hasznos metódusok

- ▶ `.Max()`, `.Min()`, `.Count()`, `.Sum()`, `.Average()`
- ▶ `.Contains(value)`
- ▶ `.Distinct()`
- ▶ Halmaz műveletek
 - `.Concat(list)`, `.Union(list)`, `.Except(list)`, `.Intersect(list)`
- ▶ Exception-t dob ha üres
 - `.First()` vagy `First(x => x > 10)`
 - `.Last()`
 - `.ElementAt()`
- ▶ Nem dob Exception-t
 - `.FirstOrDefault(value)`
 - `.LastOrDefault(value)`
 - `.ElementOrDefault(value)`
 - `.Take(value)`
- ▶ True/False
 - `.Any()` // Van-e elem a listában
 - `.All()` // Az összes elem teljesíti-e a feltételt



LINQ: order by + then by

```
List<string> names = new List<string>()
{
    "István", "Iván", „Imre”, „Imola”,
    "Viktória", "Vazul", "Viktor", "Valentina"
};
```

```
var res1 = from name in names
            orderby name[0], name[1]
            select name;
```

► Vagy:

```
var res2 = names.OrderBy(name => name[0])
                .ThenBy(name => name[1]);
```



LINQ: group by

```
List<string> names = new List<string>()
{
    "István", "Iván", "Imola", "Imre",
    "Viktória", "Vazul", "Viktor", "Valentina"
};
```

```
var res3 = from name in names
            orderby name[0]
            group name by name[0];
```



```
var res4 = names.OrderBy(name => name[0])
                .GroupBy(name => name[0]);
```

- ▶ Key: ami alapján a csoportosítás történik

LINQ: listák összekapcsolása

- ▶ SQL adatbázisoknál ezt primary key – foreign key kapcsolatként kezeljük
 - Consumer: ID, name, favourite product
 - Product: ID, name

```
var result =
```

```
from consumer in consumers  
join product in products  
on consumer.FavProductId equals product.Id  
select new  
{  
    Name = consumer.Name,  
    Product = product.Name,  
};
```



Gyakorlás

- ▶ Írjunk egy LINQ lekérdezést, amely kilistázza azokat a termékeket, amelyek senkinek sem kedvence.
- ▶ Írjunk egy LINQ lekérdezést, amely klub szerint csoportosítva kilistázza a klub játékosainak az átlagteljesítményét csökkenő sorrendben.



Párhuzamos programozás

- ▶ Minden egyes process rendelkezik egy ún. fő szállal (main thread), amely a program belépési pontja
- ▶ A többszálú programozás legnagyobb kihívása a szálak és feladataik megszervezése, az erőforrások elosztása
- ▶ Az OS thread-ek kilistázása:

```
Process.GetProcesses(".");
```

```
ProcessThreadCollection ptc = process.Threads;
```

- ▶ .NET által kezelt main thread lekérése:

```
Thread.CurrentThread.ManagedThreadId
```



Új thread létrehozása

- ▶ A Thread konstruktorában szereplő ThreadStart delegate-nek kell megadnunk azt a metódust, amelyet a másodlagos szál majd futtat
- ▶ Ha a meghívott metódusnak paramétere is vannak, akkor a ParameterizedThreadStart-tal kell létrehozni

```
public delegate void ParameterizedThreadStart(object? obj);
```

- ▶ Lambdad kifejezéssel viszont hívhatunk tetszőleges paraméterezésű metódust

```
new Thread(() => Sum(1,2,3))
```

- ▶ Ilyesfajta szálkezelésnél a szálinterakciókra különösen nagy figyelmet kell fordítani (alkalmas lehet: join, lock, mutex vagy semaphore)
 - Szinkronizáció
 - Kölcsönös kizárás
 - Holtpont

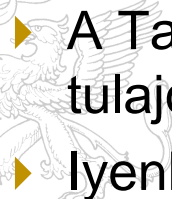


Task Parallel Library

- ▶ Az alacsony szintű szálkezelés helyett magasabb szintű absztrakcióval dolgozzunk
- ▶ Alapköve a Task osztály
- ▶ Egy Task objektum egy komplett párhuzamos feladatot reprezentál, míg egy szál csak a munkafolyamat része
- ▶ A Task objektum létrehozása és elindítása egy lépésben is végrehajtható a Run segítségével

```
Task task = new Task( () => { } );
```

```
Task task = Task.Run( () => { } );
```

- 
- ▶ A Task paraméteresített változatán megjelenik a Result tulajdonság
 - ▶ Ilyenkor a paraméter típusa lesz a visszatérési érték típusa

```
Task<int> task2 = new Task<int>( () => { } );
```

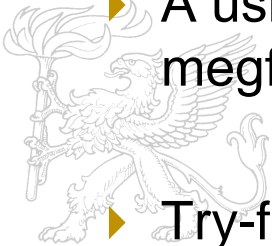
Async / Await

- ▶ Az aszinkron végrehajtás lényege, hogy az egyes feladatokat el tudjuk küldeni egy másik szálba, a fő szál pedig fut tovább, amíg a mellékszál(ak) vissza nem térnek
 - ▶ Grafikus alkalmazások esetén nem engedhető meg, hogy a felhasználói felület lefagyjon, amíg pl. adatbázisból lekért adatokra várunk
 - ▶ Az async kulcsszót csakis olyan metódusok/függvények esetében használhatjuk, ahol a visszatérési érték típusa void, Task vagy Task<T> (a Main-en nem lehet)
 - ▶ Az async megléte nem kötelez minket az await használatára, ez fordítva viszont nem igaz
- ▶ <https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios>



Using

- ▶ Mechanizmust biztosít a nem menedzselt erőforrások felszabadítására
- ▶ Menedzselt kód: amit a Common Language Runtime hajt végre (ez biztosítja az automatikus memóriamenedzsmentet is – GC)
- ▶ A using esetén az objektumnak az IDisposable interfészt kell megvalósítania
- ▶ A using biztosítani fogja, hogy az objektum Dispose() metódusa megfelelően legyen meghívva - akkor is ha kivétel keletkezik
- ▶ Try-finally fog generálódni a háttérben
- ▶ <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/using>



Fájl beolvasás, kiíratás

- ▶ A @"<path>" szükséges az útvonalban található \ , / karakterek feloldásához
- ▶ StreamReader/Writer esetén alapesetben a Close() hívással zárjuk, hogy biztosítsuk az összes adat helyes olvasását/írását – de a using ezt megoldja nekünk

▶ Read:

```
File.ReadAllText(filePath);
File.ReadAllLines(filePath);
new StreamReader(filePath)
```



▶ Write:

```
File.WriteAllText(filePath, text);
File.WriteAllLines(filePath, lines);
new StreamWriter(filePath)
```

Amiről még lehetne szó..

- ▶ Aszinkron delegate: <https://learn.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/calling-synchronous-methods-asynchronously>
- ▶ Foreground és background szálak: <https://learn.microsoft.com/en-us/dotnet/standard/threading/foreground-and-background-threads>
- ▶ ThreadPool: <https://learn.microsoft.com/en-us/dotnet/api/system.threading.threadpool?view=net-7.0>
- ▶ Parallel For: <https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-write-a-simple-parallel-for-loop>
- ▶ Parallel Invoke: <https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-use-parallel-invoke-to-execute-parallel-operations>
- ▶ Stopwatch: <https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-7.0>
- ▶ Task Factory: <https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory?view=net-7.0>

