

Alkalmazásfejlesztés II. Gyakorlat

2024/2025
I. félév

Utoljára frissítve: 2024.10.07.



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/>
- ▶ Gyakorlati anyagok: Coospace
 - A sed-es oktatási oldal nem minden esetben tartalmaz aktuális információkat!



Követelmények

- ▶ Hivatalos: Coospace „Követelmények” fül
- ▶ A gyakorlatok látogatása kötelező
- ▶ Amennyiben a hallgató 2-nél több alkalommal igazolatlanul hiányzik, akkor a gyakorlat értékelése „*nem értékelhető*”
- ▶ Folyamatos számonkérés
 - 3 teszt 8-8-8 pontért (nincs javítási és pótlási lehetőség)
 - 3 házi feladat 5-7-7 pontért (nincs javítási és pótlási lehetőség)
 - 3 zárthelyi dolgozat 17-20-20 pontért
- ▶ A házi feladatok értékelése a zárthelyi dolgozattal együtt történik
- ▶ A zárthelyi dolgozatoknál a hallgatónak a saját házi feladat megoldásából kell kiindulnia
- ▶ A dolgozatok nem javíthatóak, de igazolt hiányzás mellett a vizsgaidőszak első hetében egy dolgozat pótolható



Időbeosztás

Hét (dátum)	Hétfő	Szerda	Csütörtök
1 (09.09.)			
2 (09.16.)	Elmarad		
3 (09.23.)			Elmarad
4 (09.30.)	HF 1 kiadása		
5 (10.07.)	Teszt 1		
6 (10.14.)	C# konzolos alkalmazás ZH		
7 (10.21.)		Munkaszüneti nap	
8 (10.28.)	HF 2 kiadása		
9 (11.04.)	Teszt 2		
10 (11.11.)	WinForms alkalmazás ZH		
11 (11.18.)			
12 (11.25.)	HF 3 kiadása		
13 (12.02.)	Teszt 3		
14 (12.09.)	ASP.NET alkalmazás ZH		

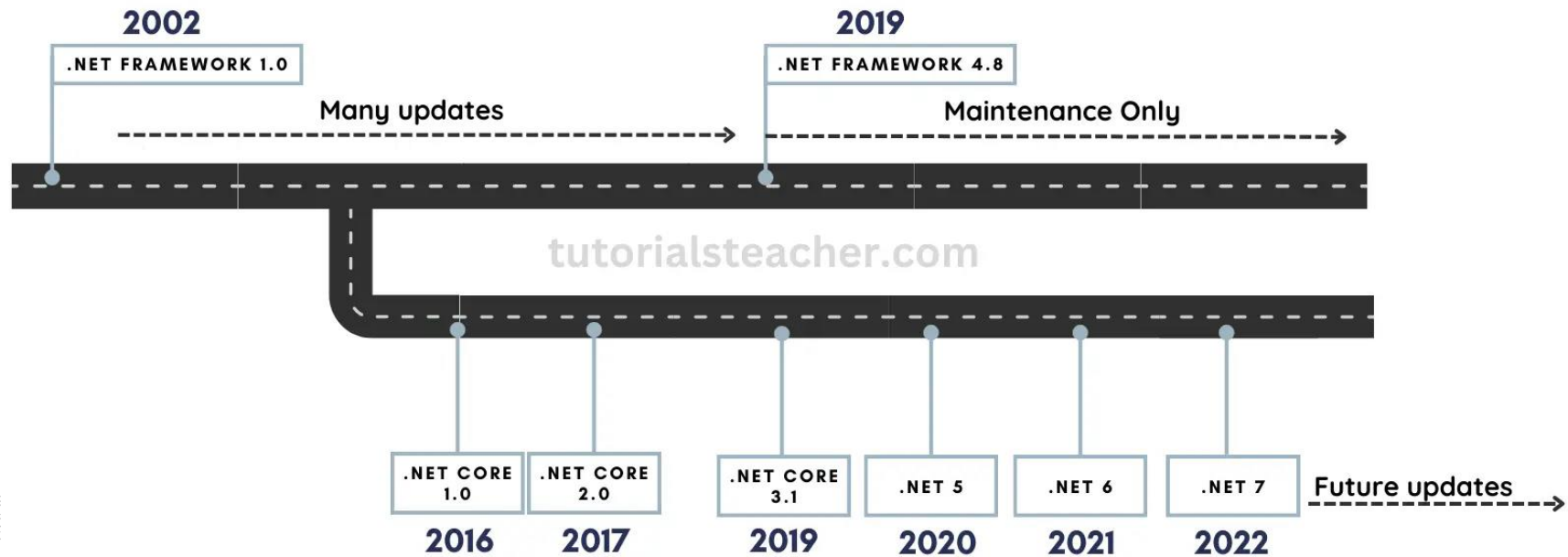
.NET platform

- ▶ A .NET Framework 2002-es kiadásával jelent meg a C#
- ▶ 2016-ban jelent meg a .NET Core, a keretrendszer nyílt forráskódú, cross-platform alkalmazások írását támogató verziója
- ▶ .NET 6-os verzió kiadásával együtt jelent meg a Visual Studio 2022 is, amely lehetővé teszi a projektek fejlesztését felhőre, böngészőre, IoT eszközre, mobilra és asztali környezetre egy egységesített platformon (megegyező .NET könyvtár, SDK, futtatókörnyezet)
- ▶ Microsoft által kifejlesztett átfogó környezet különböző típusú alkalmazások fejlesztésére
- ▶ A .NET-re különböző eszközök, nyelvek és könyvtárak nagy családjaként lehet tekinteni
- ▶ Aktív dotnet verzió a 8-as, a 6-ost 2024. novemeber 12-ig támogatják
- ▶ <https://dotnet.microsoft.com/>
- ▶ <https://github.com/dotnet>



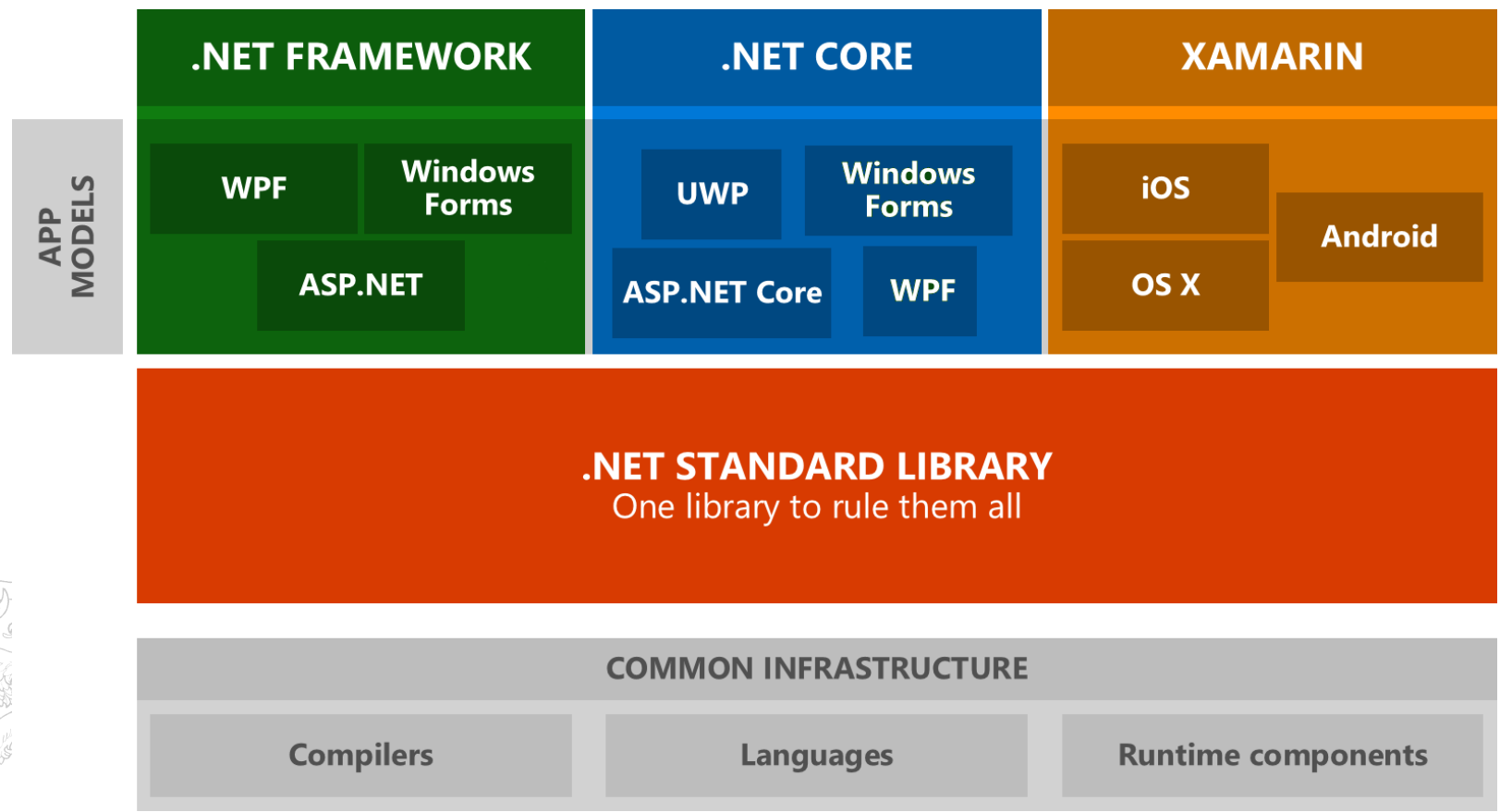


.NET idővonal

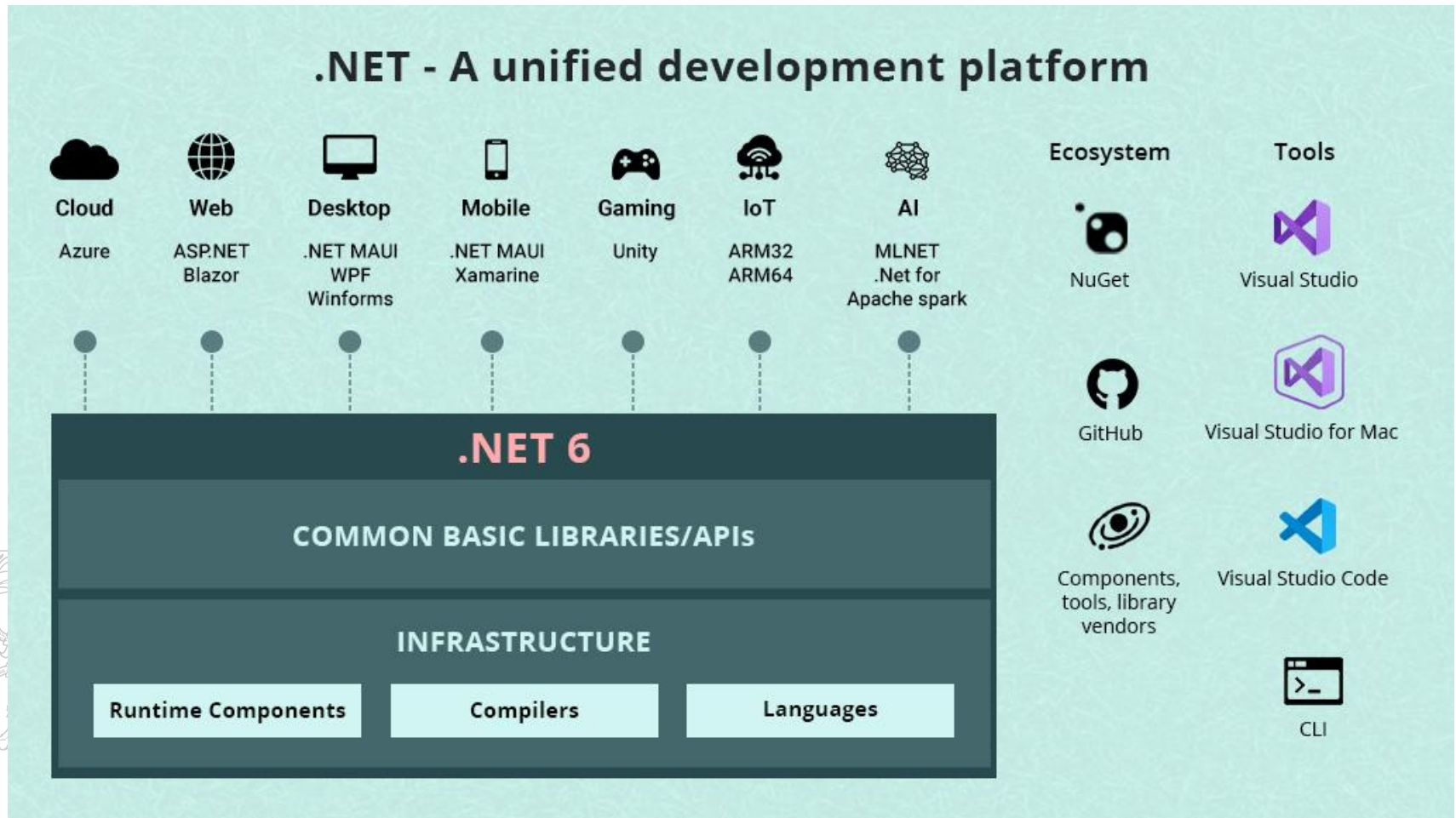




.NET architektúra elemei



Egységesített fejlesztés



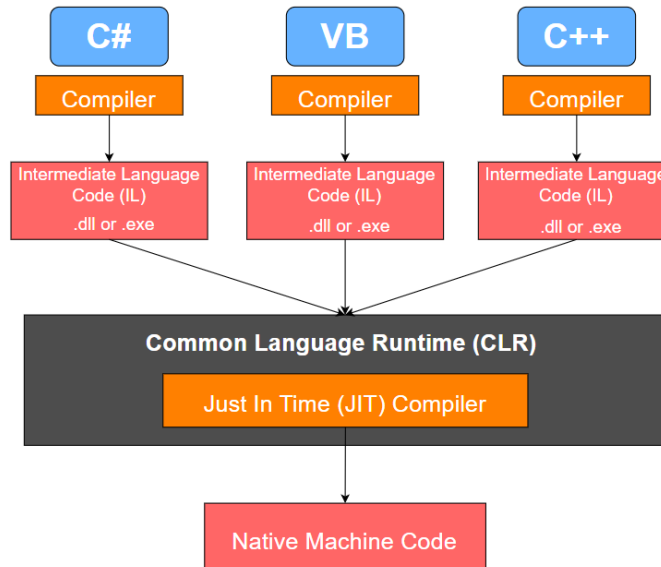


Menedzselt kód

- ▶ A menedzselt kód .NET környezetben olyan kódot jelent, amelyet a CLR, a .NET futtatókörnyezete kezel és futtat
 - Memóriakezelés
 - Típusbiztonság
 - Hibakezelés
 - Platformfüggetlenség

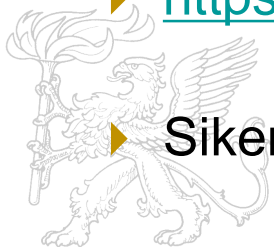
Source Code

Managed Code



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
- ▶ Az SDK külön is telepíthető (pl. Linux-ra, MacOS-re)
- ▶ <https://dotnet.microsoft.com/en-us/download/dotnet/8.0>
- ▶ Sikeres telepítés ellenőrzése parancssorban: *dotnet –info*



Visual Studio

Installing — Visual Studio Community 2022 — 17.11.2

Workloads Individual components Language packs Installation locations

Need help choosing what to install? [More info](#)

Web & Cloud (4)

- ☒ **ASP.NET and web development**
Build web applications using ASP.NET Core, ASP.NET, HTML/JavaScript, and Containers including Docker supp...
- ☐ **Python development**
Editing, debugging, interactive development and source control for Python.
- ☐ **Azure development**
Azure SDKs, tools, and projects for developing cloud apps and creating resources using .NET and .NET Framework...
- ☐ **Node.js development**
Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

Desktop & Mobile (5)

- ☐ **.NET Multi-platform App UI development**
Build Android, iOS, Windows, and Mac apps from a single codebase using C# with .NET MAUI.
- ☒ **.NET desktop development**
Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET and .NET Frame...

Location
C:\Program Files\Microsoft Visual Studio\2022\Community [Change...](#)

By continuing, you agree to the [license](#) for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the [3rd Party Notices](#) or in its accompanying license. By continuing, you also agree to those licenses.

[Remove out-of-support components](#)

Total space required 8,6 GB

[Install while downloading](#) [Install](#)

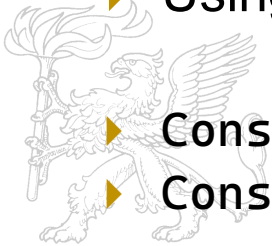
Installation details

- Visual Studio core editor
The Visual Studio core shell experience, including syntax-aware code editing, source code control and work item management.
- .NET desktop development
- ASP.NET and web development

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

Hello World

- ▶ A projekt egy Solution-ben helyezkedik el
- ▶ Olyan, mint egy konténer, mely több projektet tartalmazhat
 - .sln kiterjesztés
- ▶ A projekt konfigurációját és függőségeit definiáló fájl
 - .csproj
- ▶ Az osztályok névterekbe vannak sorolva
- ▶ Namespace-ek (~Java package-ek)
- ▶ Using (~Java import kulcsszó)
- ▶ `Console.WriteLine`
- ▶ `Console.ReadLine`
- ▶ String interpoláció: a \$ jel használatával történik a karakterlánc előtt, az interpolált változókat {} jelek közé kell helyezni



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(...);
```

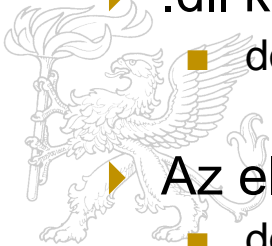
Debugging

- ▶ 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)



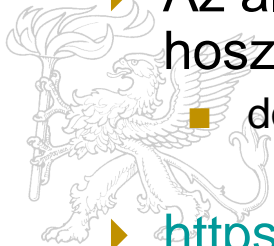
CLI

- ▶ 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, ha szükséges)
 - `dotnet run`
- ▶ `.dll` közvetlen futtatása (ha a megfelelő könyvtárban vagyunk)
 - `dotnet hello.dll`
- ▶ Az előző build-elés kimenetét törli
 - `dotnet clean`



Még egy kis CLI..

- ▶ NuGet csomag hozzáadása
 - `dotnet add package Serilog.Sinks.Console`
- ▶ Törli a hivatkozott csomagot projektből
 - `dotnet remove package Serilog.Sinks.Console`
- ▶ A .csproj fájlban található függőségeket megkeresi, szükség esetén letölti (tipikusan manuális módosításkor)
 - `dotnet restore`
- ▶ Az alkalmazást és a függőségeit egy mappába csomagolja hosztoláshoz (build-eli is a csomagolás előtt)
 - `dotnet publish`
- ▶ <https://learn.microsoft.com/en-us/dotnet/core/tools/>




Típusok C#-ban

- ▶ A .NET minden típusa direkt vagy indirekt módon a System.Object nevű típusból származik
- ▶ Referencia típus (heap-en)
 - string, object, class, interface, delegate, array, dynamic
- ▶ 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
- ▶ közvetlenül az osztályhoz tartozik, nem pedig az osztály példányához
- ▶ A *readonly* módosítóval ellátott objektumok is módosíthatatlanok, viszont itt a deklaráció és a definíció szétválik, a definíciónak elég a konstruktorban megtörténnie (*static* módosító nélkül példányszintű)



```
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 (tehát nem dinamikus típus!)
- ▶ Csak lokális változók deklarálására használhatjuk (tehát pl. metódus visszatérési értéke nem lehet implicit típus)
- ▶ 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 és van alapértelmezett értékük (pl. bool – false)
- ▶ Nullable segítségével mégis kaphatnak null értéket
- ▶ Hasznos lehet hiányzó értékek kezeléséhez



```
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>

- ▶ Rövid élettartamú, ideiglenes adatok tárolására

```
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 { }
```

- ▶ A gyerekosztály konstruktorában az ősosztály konstruktorát az `base` kulcsszó segítségével hívhatod meg

```
public Child(string name) : base(name) { }
```

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 Welcome() {}
```

▶ Leszármazott:

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



Property-k

- ▶ Egy speciális adattagja az osztálynak, mely lehetővé teszi privát változók kontrollált hozzáférését
- ▶ Első ránézésre adattagok, viszont speciális metódusok legtöbbször publikus láthatósággal ellátva
- ▶ Lehetőség van a required kulcsszó használatára: amikor egy osztályt példányosítasz, az összes required tulajdonságot meg kell adni, különben a fordító hibát fog jelezni
- ▶ 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; }
```

```
}
```


Nevesített + alapértelmezett paraméterek

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

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

- ▶ Ezt követően a példányosítás ennyi is lehet nevesített paraméterekkel, amelyek bármilyen sorrendben megadhatóak
- ▶ De a pozíció szerinti és nevesített paramétereket nem keverheted úgy, hogy a pozíció szerinti később következzen

```
Camel camel2 = new Camel(power: 2, color: "yellow");
```

Extension Method

- ▶ Már létező típusokhoz új metódusokat tudunk hozzáadni anélkül, hogy azok kódját módosítanánk vagy származtatnánk belőlük
- ▶ Minden esetben egy statikus osztály statikus metódusa kell, hogy legyen
- ▶ Az első paramétere az a típus, amelyhez hozzá szeretnéd adni a metódust, és ezt a paramétert a `this` kulcsszóval kell ellátni

```
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 SwapRef(ref Program p)
{
    p = new Program();
}
```

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

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

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

```
List<int> numbers = new List<int>();
```

- ▶ Dictionary <TKey, TValue>: kulcs-érték párokat tárol

```
Dictionary<string, int> people = new Dictionary<string, int>();
```

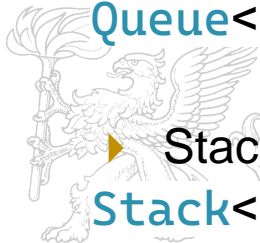
- ▶ Queue: FIFO lista

```
Queue<string> tasks = new Queue<string>();
```

- ▶ Stack: LIFO lista

```
Stack<string> tasks = new Stack<string>();
```

- ▶ Törekedjünk generikus kollekciók használatára elsősorban a típusbiztonság miatt!



Foreach & Yield

- ▶ Szinte minden kollekció az IEnumerable interfészre épül, így bejárható foreach ciklussal
- ▶ Amikor a yield kifejezést használod, a fordító automatikusan generál egy olyan osztályt, amely megvalósítja az IEnumerable<T> interfészt
- ▶ A metódus az egyes értékeket sorozatosan adja vissza, így nem kell a teljes kollekciót előre létrehozni
- ▶ Ezáltal használható legyen pl. a foreach ciklussal

```
static public IEnumerable<int> EnumerableMethod(int max){
    for (int i = 0; i < max; ++i)
    {
        if (i > 5)
        {
            yield break; // kilép a metódusból
        }
        yield return i; // megőrződik az állapot
    }
}
```

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



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. adatbáziskapcsolat lezárása)
- ▶ A finally blokk opcionális, és mindig végrehajtódik

```
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

Típusellenőrzés és konverzió

- ▶ Ellenőrzött konverzió numerikus értékek túlcsoordulásának ellenőrzéséhez:
<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/checked-and-unchecked>

```
int x = 10;
long y = x; // Az implicit konverzió automatikusan végbemegy a fordító által

int x = 300;
byte y = (byte) x; // explicit konverziónál adatvesztés történhet (32 bit vs 8 bit)
```

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

```
double szam = 6.76;
Console.WriteLine(szam is int);
Console.WriteLine(szam.GetType());
```



- ▶ Párja az *as*, az ellenőrzés mellett egy explicit típuskonverziót is végrehajt
- ▶ Ezzel az operátorral csakis referenciatípusra konvertálhatunk, értéktípusra nem (hiba esetén null-t ad vissza)

```
object b = "Hello";
string bb = b as string;
```

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
 - Egy objektum más objektumokat értesít, amikor valamilyen esemény bekövetkezik
- ▶ Mivel a létrehozott delegate egy objektum, így átadható függvényeknek, mint paraméter
- ▶ Olyan metódus(okat) tárolhat, ami egy int típusú paramétert fogad és int típusúval tér vissza:

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

```
TestDelegate testDelegate = FuctionName;
```

```
void 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;
```

```
anotherDel = Two;  
anotherDel += Two;  
anotherDel += Three;  
anotherDel -= Two;  
anotherDel();
```



Func & Action

- ▶ Beépített delegált típusok, amelyek egyszerűsítik a delegate-ek használatát
- ▶ A Func-nak lehetnek input paraméterei (akár nulla, de több is) és egy visszatérési értéke van
- ▶ Az utolsó paraméter a visszatérési érték

```
Func<int, int, long> multi = FuncTest;
```

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

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



Névtelen metódus és lambda kifejezés

- ▶ Névtelen metódus létrehozása a delegate kulcsszóval történik

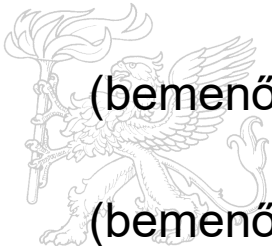
```
delegate int TestDelegate(int x);
```

```
TestDelegate del = delegate (int x) { return x * x; };
```

- ▶ A lambda kifejezés egy rövidített szintaxist biztosít a névtelen metódusokhoz
- ▶ Minden lambda kifejezés delegate-té alakítható (Func és Action)
- ▶ lambda operátor: =>
- ▶ Lehetséges formái a nyelvben:

(bemenő-paraméterek) => kifejezés

(bemenő-paraméterek) => { több utasítás }



LINQ

- ▶ A Language-Integrated Query, vagyis a LINQ lehetővé teszi a kollekciókban (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>
- ▶ A LINQ kétféleképpen használható: query szintaxis (SQL-szerű), illetve method szintaxis

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

```
IEnumerable<int> scoreQuery =
```

```
from score in scores
```

```
where score > 80
```

```
select score;
```

```
IEnumerable<int> scoreQuery2 = scores.Where(x => x > 80);
```



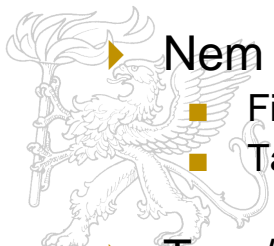
LINQ (folyt.)

- ▶ LINQ lehetővé teszi adatforrások kezelést új „nyelv” megtanulása nélkül (pl. SQL, XML)
- ▶ 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 (var): nehéz előre megadni az eredménylista típusát
 - Névtelen típus: sok esetben nincs szükségünk egy objektum minden adatára, ilyenkor feleslegesen foglalná egy lekérdezés eredménye a memóriát



LINQ kifejezések

- ▶ A ForEach metódus kizárólag a List<T> típusú kollekciókra van definiálva!
- ▶ Legfontosabb LINQ metódusok:
 - Select: a sorozat minden elemét átalakítja
 - Where: megszűri a sorozatot
 - OrderBy, OrderByDescending: rendezés
 - Max, Min, Count, Sum, Average: aggregáló kifejezések
 - Contains: tartalmazás
 - Distinct: duplikált értékek megszüntetése
 - Concat, Union, Except, Intersect: halmaz műveletek
- ▶ Exception-t dob ha üres a kollekció
 - First, Last: visszaadja az (adott feltételnek megfelelő) első/utolsó elemet
 - ElementAt: adott pozícióban lévő elem
- ▶ Nem dob Exception-t
 - FirstOrDefault, LastOrDefault, ElementAtOrDefault
 - Take: egy meghatározott számú elemet vesz ki
- ▶ True/False értékkel tér vissza
 - Any: van-e az adott feltételt teljesítő elem a listában
 - All: az összes elem teljesíti-e az adott feltételt



LINQ: order by + then by

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

- ▶ OrderBy: rendezi az elsődleges kulcs alapján
- ▶ ThenBy: további rendezési kritériumokat alkalmaz, amikor a korábbi kulcsok alapján az elemek azonosak

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

- ▶ Group By: egy kulcsot használ, amely alapján a csoportosítás történik
- Key: ami alapján a csoportosítás történik

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

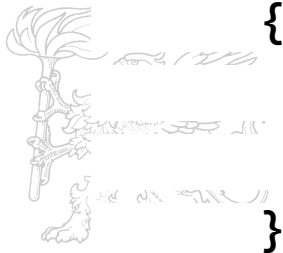
foreach (var key in groups) {
    Console.WriteLine(key.Key);
}
```



LINQ: listák összekapcsolása

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

```
var map = consumers.Join(  
    products,                                // első kollekció  
    consumer => consumer.Fav,               // második kollekció  
    product => product.Id,                  // első kulcsa  
    (consumer, product) => new              // második kulcsa  
    {                                         // eredmény  
        Consumer = 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

- ▶ Több könyvtár is elérhető a párhuzamos programozáshoz, a legfontosabbak:
 - Thread osztály: manuálisan létrehozott szálak
 - Task Parallel Library (TPL): magasabb absztrakciós szint (szálak automatikus kezelése)
 - async / await: aszinkron programozás
 - PLINQ: <https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq>
 - Konkurens kollekciók: <https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent?view=net-8.0>
- ▶ Minden egyes process (a futó program példánya) rendelkezik egy ún. fő szállal (main thread), amely a program belépési pontjával indul
- ▶ A többszálú programozás legfőbb kihívása a szálak és feladatainak megszervezése, az erőforrások megfelelő elosztása, valamint a versenyhelyzetek kezelése illetve a holtpontok elkerülése



Monitorozás

- ▶ Az folyamatok kilistázása a System.Diagnostics névtérből:

`Process.GetProcesses()`

- ▶ A jelenleg futó folyamat:

`Process.GetCurrentProcess()`

- ▶ A folyamat szálainak kilistázása:

`foreach (ProcessThread thread in currentProcess.Threads) {..}`

- ▶ .NET által kezelt main thread lekérése a System.Threading névtérből:

`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 paraméterei is vannak, akkor a ParameterizedThreadStart delegate-tel 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 pl. Monitor, Semaphore, Lock, Join, Mutex..)
 - Szinkronizáció
 - Kölcsönös kizárás
 - Holtpont



Task Parallel Library

- ▶ Az alacsony szintű szálkezelés helyett magasabb szintű absztrakcióval dolgozzunk
- ▶ Alapköve a Task osztály, amely egy komplett párhuzamosan végrehajtandó feladatot reprezentál (
- ▶ A Task objektum létrehozása és elindítása egy lépésben is végrehajtható a Run segítségével

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

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

- ▶ Egy háttérszál (mint például a Task) nem tartja működésben a menedzselt végrehajtási környezetet, ezért alapesetben a futásokat be kell várnunk a Wait()-tel

`Thread.CurrentThread.IsBackground`

- ▶ 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 programozás egy olyan programozási minta, amely lehetővé teszi a kód számára, hogy ne blokkolja a végrehajtást, amikor várakozik egy hosszú ideig tartó művelet (például hálózati kérés, fájlbeolvasás vagy adatbázis-művelet) befejezésére
- ▶ Ehelyett a vezérlés azonnal visszatér a hívó metódushoz, és a program más feladatokat végezhet, amíg az aszinkron művelet folyamatban van
- ▶ `async` kulcsszóval jelöljük, hogy egy metódus aszinkron módon fog futni, és hogy az adott metódus tartalmazhat `await` kifejezéseket
 - Az `async` kulcsszót olyan metódusoknál használjuk, amelyek visszatérési típusa `Task`, `Task<T>` vagy `void`
- ▶ Az `await` kulcsszó használatával megmondjuk, hogy várakozni szeretnénk egy aszinkron feladat befejeződésére, mielőtt folytatnánk a végrehajtást – az UI válaszképes marad
- ▶ A `Task.Wait()` szinkron módon blokkolja az aktuális szálat addig, amíg a feladat be nem fejeződik – az UI nem marad válaszképes
- ▶ <https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios>



Using

- ▶ Mechanizmust biztosít a nem menedzselte erőforrások felszabadítására
- ▶ 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
- ▶ A using szerkezet a háttérben egy try-finally blokkot generál, amely biztosítja, hogy a Dispose() metódus mindig meghívásra kerüljön, függetlenül attól, hogy a kód sikeresen lefutott-e vagy kivétel lépett fel
- ▶ <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)
```