



PROGRAMOZÁS II

1. ÓRA: BEVEZETÉS, C# ALAPOK

Miről lesz szó a félévben?

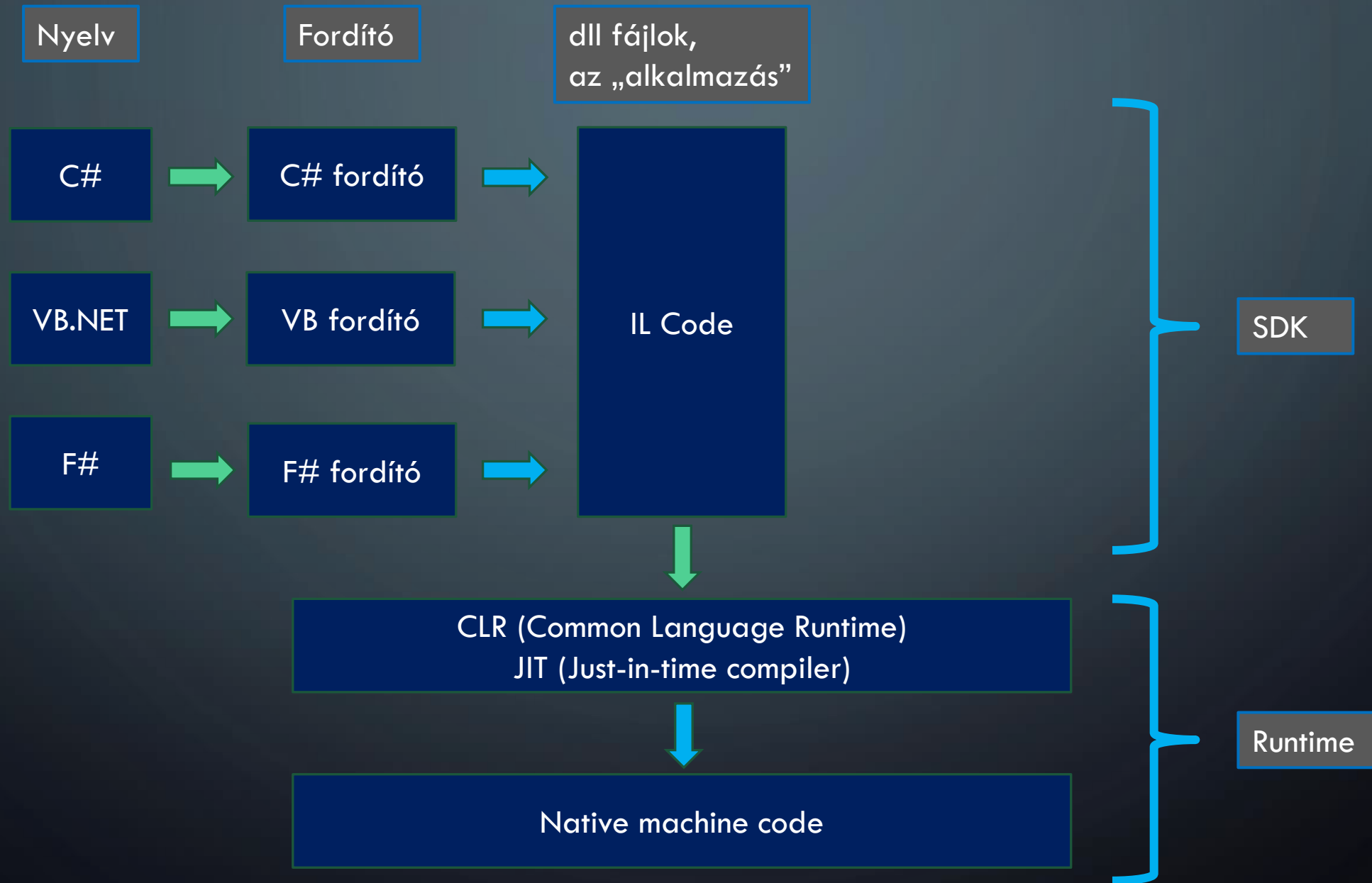
- Összefoglalva: objektum-orientált programozás
- Részletesebben:
 - C# alapjai
 - Osztályok, objektumok
 - Öröklődés
 - Polimorfizmus
 - Interfészek
 - Statikus adattagok és metódusok
 - Még még pár apróság ...

Fontos: előismeretek

- A Programozás II tárgy tananyaga épít a Programozás I és A programozás alapjai tárgyak tananyagaira
- A korábbi anyagok megfelelő ismerete elengedhetetlen a mostani féléves tananyagok megértéséhez!

BEVEZETÉS

C# (.NET) működése



Kötelező kód a C# programban

- C-vel ellentétben itt nem csak egy main függvényt kell írni egy fájlba, ahhoz, hogy a program elinduljon
- Már a belépési pont is egy kötöttebb, objektum orientált szerkezetben létezik

```
namespace ...  
{  
    class ...  
    {  
        public static void Main(string[] args)  
        {  
            ...  
        }  
    }  
}
```

Kell egy név a fő névtérnek

Kell egy név a fő osztálynak

Itt lesz a program fő kódja

Hello world!

```
using System;

namespace MyProgram
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
        }
    }
}
```

Hello world!

```
using System;

namespace MyProgram
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
        }
    }
}
```

Ezekről majd
később lesz szó

A Main függvény a program belépési pontja

Ez az utasítás kiírja a kívánt üzenetet

TÍPUSOK, VÁLTOZÓK, BEOLVASÁS, KIÍRÁS

Típusok

- Az alapvető (primitív) típusok a C-hez hasonlóak:
- Egész (4 bájt): **int**
- Hosszú egész (8 bájt): **long**
- Egyszeres lebegőpontos (4 bájt): **float**
- Dupla lebegőpontos (8 bájt): **double**
- Karakter (**2 bájt**): **char**

Vannak előjel nélküli verziók is (**uint**, **ulong**), de ritkán használjuk őket

Új alap (primitív) típusok : **bool**

- Logikai igaz-hamis értéket tárol
- Az értéke lehet **true** vagy **false**

```
int szam = 50;  
bool paros = szam%2 == 0;  
Console.WriteLine(paros);  
if (paros)  
{  
    Console.WriteLine("Paros");  
}  
else  
{  
    Console.WriteLine("Paratlan");  
}
```

Új alap (primitív) típusok : **string**

- Szöveget (karakterek sorozatát) tárolja

```
string fixSzoveg = "Ez egy fix szoveg";  
string bekertSzoveg = Console.ReadLine();  
Console.WriteLine(fixSzoveg);  
Console.WriteLine(bekertSzoveg);
```

- A karakterek egyesével is végig járhatóak, mint egy tömb
- A szöveg hossza lekérdezhető (szoveg.**Length**)

```
for (int i = 0; i < bekertSzoveg.Length; i++)  
{  
    Console.WriteLine(bekertSzoveg[i]);  
}
```

Kiírás

- A parancssorba kiírást tipikusan a `Console.Write()` vagy a `Console.WriteLine()` függvényhívások segítségével végezhetjük.
 - Utóbbi új sort is tesz a kiírás végére, míg az előbbi nem
 - Mindkettő egy szöveget ír ki, de a nem szöveg típusokat tudja konvertálni
 - A kiírt szöveget részekből összefűzve is fel lehet építeni
 - Most csak gyakori felhasználási módokat nézzük meg, ami a kurzus során bőven elég, ha valakit érdekel, utána nézhet, hogy mi van még

Kiírás: egyszerű példák

```
int egész = 43;  
double lebego = 1234.5678;  
char ch = 'A';  
bool logikai = false;  
string szoveg = "alma";  
Console.WriteLine("Pelda kiiratas");  
Console.WriteLine(egész);  
Console.WriteLine(lebego);  
Console.WriteLine(ch);  
Console.WriteLine(logikai);  
Console.WriteLine(szoveg);
```

Kiírás: összefűzéses példák

- A + jellel változók és fix szövegek könnyen összefűzhetők

```
int egész = 43;
double lebego = 1234.5678;
char ch = 'A';
string szoveg = "alma";

Console.WriteLine("Az egész változó értéke: " + egész);
//Az egész változó értéke: 43
Console.WriteLine("Egy lebegőpontos érték: " + lebego);
//Egy lebegőpontos érték: 1234.5678
Console.WriteLine("'" + ch + "' az egyik kedvenc karakterem");
//'A' az egyik kedvenc karakterem
Console.WriteLine("Ezen a fan naponta " + egész + " darab " + szoveg + " terem");
//Ezen a fan naponta 43 darab alma terem
```

Kiírás: szöveg interpoláció

- A szöveg interpoláció egy modernebb megoldás arra, hogy a szövegbe változó értékeket illesszünk
- Figyeljük meg, hogy a szöveg előtt **\$** található!
- Akár kifejezés is írható bele

```
int egész = 43;  
double lebego = 1234.5678;  
char ch = 'A';  
string szoveg = "alma";
```

```
Console.WriteLine($"Az egész változó értéke: {egész}"); //Az egész változó értéke: 43  
Console.WriteLine($"Egy lebegőpontos érték: {lebego}");  
Console.WriteLine($"\"' {ch} \"' az egyik kedvenc karakterem");  
Console.WriteLine($"Ez egy fa, melyen naponta {egész} darab {szoveg} terem");  
Console.WriteLine($"{{egész}} + 23 = {{egész + 23}}"); //43 + 23 = 66  
Console.WriteLine($"{{egész}} < {{lebego}} --?-- {{egész < lebego}}");  
//43 < 1234.5678-- ? --True
```


Formázott kiírás

- Létezik printf-hez hasonló verzió is
- A szövegben a megjelenítendő értékek helyét a `{0}`, `{1}`, `{2}`, ... placeholder-ekkel jelöljük

```
int egesz = 43;  
double lebego = 1234.5678;  
char ch = 'A';  
string szoveg = "alma";
```

```
Console.WriteLine("Az egesz változó értéke: {0}", egesz); //Az egesz változó értéke: 43  
Console.WriteLine("Egy lebegőpontos érték: {0}", lebego);  
Console.WriteLine("' {0} ' az egyik kedvenc karakterem", ch);  
Console.WriteLine("Ez egy fa, melyen naponta {0} darab {1} terem", egesz, szoveg);  
Console.WriteLine("{0} + 23 = {1}", egesz, egesz + 23); //43 + 23 = 66  
Console.WriteLine("{0} < {1} --?-- {2}", egesz, lebego, egesz < lebego);  
//43 < 1234.5678-- ? --True
```

Bekérés: szöveg

- Bekérésre is lehet több módszert találni
- Számunkra a legegyszerűbb a `Console.ReadLine()`
- Egy teljes sort olvas be szövegbe

```
string szoveg = Console.ReadLine();
```

Bekérés: számok

- Ha számot szeretnénk beolvasni, azt is a teljes sor bekérésével kezdjük
- A beolvasott szöveget utána konvertálni kell
- Ez a **Parse** függvénnyel tehető meg, de elé be kell írni a cél típust

```
int egész = int.Parse(Console.ReadLine());  
double lebego = double.Parse(Console.ReadLine());
```

- Ez mindig teljes sort olvas be, tehát egy sorban egy érték legyen!
- Ha nem tudja a beolvasott szöveget a megfelelő típusra alakítani, akkor hibával kilép a program

Alapvető vezérlési szerkezetek

- Az alapvető szerkezetek ugyanúgy működnek, mint C-ben:
- Elágazás: **if, else, switch-case**
- Ciklus: **for, while, do-while**
- Vezérlés: **break, continue, return**
- Ahol lesz kiegészítés vagy fontos változás, ott erről külön beszélünk

REFERENCIÁK

Referenciák

- C#-ban nincs C szintaxisú mutató
- Ellenben minden összetett adatot memóriacímként (referenciaként) tárolunk és dinamikus memórafoglalást igényel
- Manuális felszabadítás nem kötelező
- A memória közvetlen kezelését alapértelmezetten a .NET framework intézi

Referenciák

- C#-ban nincs C szintaxisú mutató

Nem teljesen igaz, de ebbe ezen a kurzuson megyünk bele

- Ellenben minden összetett adatot memóriacímként (referenciaként) tárolunk és dinamikus memória foglalatást igényel

Nem teljesen igaz, de ebbe ezen a kurzuson megyünk bele

- Manuális felszabadítás nem kötelező

Erről majd később még lesz szó

- A memória közvetlen kezelését alapértelmezetten a .NET framework intézi

Tömbök

```
int[] tomb1;
```

Ez így még csak egy referencia, de nincs mögötte tényleges tömb

```
int[] tomb1 = new int[5];
```

Ez így már egy 5 elemű tömb

```
int[] tomb1 = { 3, 4, 6, 8, 9 };
```

Meg lehet adni kezdő értékekkel is

```
int[] tomb1 = new int[5]{ 3, 4, 6, 8, 9 };
```

Ez ugyanaz

```
int[] tomb1 = new int[] { 3, 4, 6, 8, 9 };
```

Meg ez is

```
Console.WriteLine(tomb1.Length);
```

A tömb tudja a méretét, nem kell külön elmenteni

Szövegek: **string**

```
string szoveg;
```

Ez így még csak egy referencia, de nincs mögötte tényleges szöveg

```
string szoveg = new string("valami");
```

Ez így már egy szöveg

```
string szoveg = "valami";
```

Így is működik

```
string masik = szoveg;
```

Nem jön létre másik **string**, csak egy másik referencia ugyanarra az objektumra

```
string szoveg = Console.ReadLine();
```

Nincs szükség explicit memórafoglalásra, a **ReadLine** hozza létre a szöveget

Szövegek tömbje?

```
string[] szovegek;
```

Ez így még csak egy referencia, de nincs se tömb, se szöveg

```
string[] szovegek = new string[10];
```

Ez egy 10 elemű tömb
Csak referenciákat tárol, nem szövegeket!

```
szovegek[0] = new string("első elem");
```

Minden egyes szöveget külön létre kell hozni

```
szovegek[1] = "második elem";
```

Így is jó

```
string[] szovegek = { "első", "második", "harmadik" };
```

Így is jó

Példa: **string** tömb bekérés és kiírás

```
Console.Write("Mennyi szovegunk legyen? ");
int meret = int.Parse(Console.ReadLine());

string[] szovegek = new string[meret];

for (int i = 0; i < szovegek.Length; i++)
{
    Console.Write($"Kerem a(z) {i + 1}. szoveget: ");
    szovegek[i] = Console.ReadLine();
}

Console.WriteLine("Mit is olvastam be? Lassuk:");
for (int i = 0; i < szovegek.Length; i++)
{
    Console.WriteLine(szovegek[i]);
}
```

Több dimenziós tömbök: 2D példa

```
int[,] tomb2d = new int[4, 3];  
for (int i = 0; i < 4; i++)  
{  
    for (int k = 0; k < 3; k++)  
    {  
        tomb2d[i, k] = int.Parse(Console.ReadLine());  
    }  
}
```

```
int[,] tomb2d = { {1, 2, 3 },  
                  { 4, 5, 6 },  
                  { 7, 8, 9 },  
                  { 10, 11, 12 } };
```

Több dimenziós tömbök: 3D példa

```
int[, ,] tomb3d = new int[2, 3, 2];  
for (int i = 0; i < 2; i++)  
{  
    for (int k = 0; k < 3; k++)  
    {  
        for (int l = 0; l < 2; l++)  
        {  
            tomb3d[i, k, l] = int.Parse(Console.ReadLine());  
        }  
    }  
}
```

```
int[, ,] tomb3d = { { { 1, 2 }, { 3, 4 }, { 5, 6 } },  
                    { { 7, 8 }, { 9, 10 }, { 11, 12 } } };
```

Több dimenziós tömbök: méret lekérése

```
int[, ,] tomb3d = new int[2, 3, 2];
```

```
Console.WriteLine($"Elemek szama: {tomb3d.Length}");
```

A **Length** az összes elem száma

```
Console.WriteLine("Dimenziok:");
```

```
for (int i = 0; i < tomb3d.Rank; i++)
```

A **Rank** a dimenziók száma

```
{
```

```
    Console.WriteLine(tomb3d.GetLength(i));
```

Adott dimenzió elemszámának lekérése

```
}
```

Tömbök tömbje

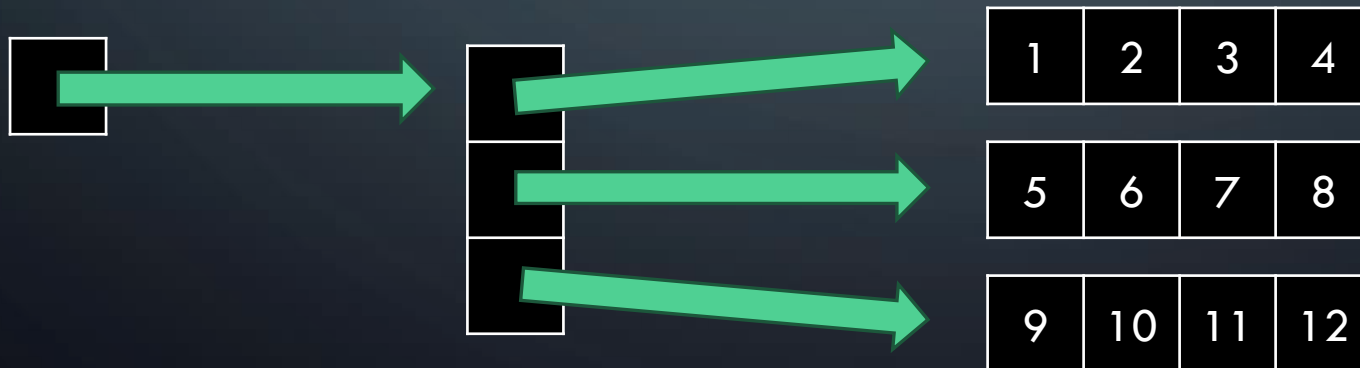
- Ahogy C-ben, a több dimenziós tömb itt is több irányból közelíthető
 - Egy nagy memória terület több indexszel

```
int[,] tomb2d = new int[3,4];
```



- Tömbök tömbje (egy tömb, ami referenciákat tárol egyéb tömbökre)

```
int[][] tombtomb = new int[3][];
```



Tömbök tömbje: létrehozás

```
int[][] tombtomb = new int[3][];  
for (int i = 0; i < tombtomb.Length; i++)  
{  
    tombtomb[i] = new int[4];  
    for (int k = 0; k < tombtomb[i].Length; k++)  
        tombtomb[i][k] = int.Parse(Console.ReadLine());  
}
```

```
int[][] tombtomb = { new int[] { 1, 2, 3, 4 },  
                    new int[] { 5, 6, 7, 8 },  
                    new int[] { 9, 10, 11, 12 } };
```


Tömbök tömbje: különböző méretek

```
int[][] tombtomb = new int[3][];  
for (int i = 0; i < tombtomb.Length; i++)  
{  
    int sormeret = int.Parse(Console.ReadLine());  
    tombtomb[i] = new int[sormeret];  
    for (int k = 0; k < tombtomb[i].Length; k++)  
        tombtomb[i][k] = int.Parse(Console.ReadLine());  
}
```

```
int[][] tombtomb = { new int[] { 1, 2, 3 },  
                    new int[] { 4, 5, 6, 7, 8 },  
                    new int[] { 9, 10, 11, 12 } };
```

Tömbök bejárása: **foreach**

- Olyan ciklus, ami index használata nélkül, közvetlenül az elemeken megy végig
- Az elemek indexe nem érhető el
- Az elemek módosítását nem teszi lehetővé
 - Ha az elem összetett adat, akkor a referencia nem módosítható, de amire hivatkozik, az igen

Megkerülhető, de ebbe nem megyünk bele

```
string[] szovegek = { "első", "második", "harmadik" };  
foreach (string s in szovegek)  
{  
    Console.WriteLine(s);  
}
```

Dinamikus memória felszabadítása

- C-ben megszoktuk, hogy ezt nekünk kell elvégezni
- C# esetén erre nincs szükség
- Az olyan területeket, amelyekre már nem hivatkozik referencia, a Garbage Collector periodikusan felszabadítja
- Nem azonnal történik, amint nincs a területre szükség, hanem valamikor később

Megkerülhető, de ebbe nem megyünk bele

Null referencia

- A referenciák esetén is lehet olyan, hogy nem hivatkozik tényleges objektumra
- Ez esetben a **null** értéket fogja tárolni, és null referenciának nevezzük

```
string szoveg = null;
Console.Write("Olvassunk be szoveget? ");
string kell = Console.ReadLine();
if (kell == "igen")
{
    Console.Write("Kerem a szoveget: ");
    szoveg = Console.ReadLine();
}
if (szoveg == null) Console.WriteLine("nincs szoveg");
else Console.WriteLine($"A szoveg: {szoveg}");
```

OSZTÁLYOK (ALAPSZINTEN)

Mi az az osztály?

- Az objektum orientált fejlesztés egyik alappillére
 - Erről majd később beszélünk
- Egyelőre elég annyi, hogy itt ez van struktúra helyett

C struktúra → C# osztály

- A kezdő kulcsszó nem **struct**, hanem **class**
- Minden adattag előtt ott a **public**, erről majd későbbi órákon beszélünk

```
struct Tantargy
{
    char nev[30];
    char neptunKod[30];
    int kreditekSzama;
    int hetiOrakSzama;
};
```



```
class Tantargy
{
    public string nev;
    public string neptunKod;
    public int kreditekSzama;
    public int hetiOrakSzama;
}
```

C# osztály

- A használata innentől elég hasonló:
 - Mivel ez egy összetett adat, ezért referenciaként működik, és dinamikus memórafoglalással kell létrehozni
 - Az adattagok a pont segítségével érhetőek el

```
Tantargy targy = new Tantargy();  
targy.nev = "Programozas II";  
targy.kod = "VEMISAB256PF";  
targy.hetiOrakSzama = 4;  
targy.kreditekSzama = 6;  
Console.WriteLine($"{targy.nev}: {targy.kreditekSzama} kredit");
```


A program szerkezete osztályokkal

- Megfigyelhető, hogy a fő programban is van egy osztály
- Ennek okát és használatát majd később részletezzük

```
namespace MyProgram
{
    class MainClass ←
    {
        public static void Main(string[] args)
        {
            ...
        }
    }
}
```

A program szerkezete osztályokkal

- Ha új osztályt hozunk létre, azt megtehetjük
 - Ugyanabban a fájlban a fő osztályon kívül
 - Külön fájlban

Van még opció, de arról majd később

A program szerkezete osztályokkal

- Ha új osztályt hozunk létre, azt megtehetjük
 - Ugyanabban a fájlban a fő osztályon kívül
 - Külön fájlban

Van még opció, de arról majd később

```
namespace MyProgram
{
    class Tantargy
    {
        public string nev;
        public string kod;
        public int kreditekSzama;
        public int hetiOrakSzama;
    }
    class MainClass
    {
        ...
    }
}
```

A program szerkezete osztályokkal

- Ha új osztályt hozunk létre, azt megtehetjük

- Ugyanabban a fájlban a fő osztályon kívül

Van még opció, de arról majd később

- Külön fájlban

- Nincs `#include`, mint a C-ben, csak legyen a fájl a projekt része
 - Ez a preferált módszer

Tantargy.cs fájl tartalma:

```
namespace MyProgram
{
    class Tantargy
    {
        public string nev;
        public string kod;
        public int kreditekSzama;
        public int hetiOrakSzama;
    }
}
```

FÜGGVÉNYEK

Függvények létrehozása

- Alapvető elvek ugyanazok, mint C-ben
 - Visszatérési típus, név, paraméterlista, függvénytörzs, return

```
int ErtekLimit(int ertek, int alsohatar, int felsohatar)
{
    ...
    return ...;
}
```

```
void TargyatKiir(Tantargy targy)
{
    ...
}
```

Függvények létrehozása

- C#-ban a függvény csak osztályon belül lehet
 - Az erősen objektum orientált felépítés miatt van így, erről majd később beszélünk
- Egyelőre a fő osztályon belülre fogjuk írni a függvényeket
 - Minden függvény előtt legyen ott a **static** kulcsszó (ahogy a Main előtt is)
 - Ez majd később nem így lesz

```
class MainClass {  
    static int ErtekLimit(int ertekek, int alsohatar, int felsohatar) {  
        ...  
    }  
    static void TargyatKiir(Tantargy targy) {  
        ...  
    }  
    public static void Main(string[] args) ...  
}
```

Függvény példa: érték limitálása határok közé

```
class MainClass
{
    static int ErtekLimit(int ertek, int alsohatar, int felsohatar)
    {
        if (ertek < alsohatar) { return alsohatar; }
        else if (ertek > felsohatar) { return felsohatar; }
        else { return ertek; }
    }

    public static void Main(string[] args)
    {
        int ertek = int.Parse(Console.ReadLine());
        ertek = ErtekLimit(ertek, 0, 100);
        Console.WriteLine(ertek);
    }
}
```


Függvény példa: tantárgy kiírása

```
class MainClass
{
    static void TargyatKiir(Tantargy targy)
    {
        Console.WriteLine($"{targy.nev} ({targy.kod}):");
        Console.WriteLine($"    {targy.hetiOrakSzama} ora");
        Console.WriteLine($"    {targy.kreditekSzama} kredit");
    }

    public static void Main(string[] args)
    {
        Tantargy targy = new Tantargy();
        targy.nev = "Programozas II";
        targy.kod = "VEMISAB256PF";
        targy.hetiOrakSzama = 4;
        targy.kreditekSzama = 6;
        TargyatKiir(targy);
    }
}
```

Paraméter átadás: érték vagy cím szerint

- Egyszerű típus (érték típus) érték szerint adódik át
 - Pl. int, double, bool, char
- Összetett típus (referencia típus) cím szerint adódik át
 - Pl. tömb vagy objektum

Paraméter átadás példa: érték típus

```
static int ParatlanigOszto(int szam)
{
    while (szam%2 == 0)
    {
        szam /= 2;
    }
    return szam;
}
```

Az **int** érték típus, így a függvény egy másolatot kap meg

```
public static void Main(string[] args)
{
    int szam = int.Parse(Console.ReadLine());
    int eredmeny = ParatlanigOszto(szam);
    Console.WriteLine($"{szam} : {eredmeny}");
}
```

A **szam** értéke még az eredeti, mert a függvény egy másolatot módosított

Paraméter átadás példa: objektum

```
static void OraszamModositas(Tantargy targy)
{
    Console.WriteLine("Kerem az uj oraszamot: ");
    targy.hetiOrakSzama = int.Parse(Console.ReadLine());
    Console.WriteLine("Kerem az uj kredit erteket: ");
    targy.kreditekSzama = int.Parse(Console.ReadLine());
}
```

A **Tantargy** referencia típus, így a függvény csak a memória címet kapja meg

```
public static void Main(string[] args)
{
    Tantargy targy = new Tantargy();
    targy.nev = "Programozas II";
    targy.kod = "VEMISAB256PF";
    OraszamModositas(targy);
    TargyatKiir(targy);
}
```

A **targy** adattagjai módosulnak, mert a függvény az eredetit használhatta

Paraméter átadás példa: tömb

```
static void Megfordit(int[] tomb)
{
    for (int i = 0; i < tomb.Length/2; i++)
    {
        int tmp = tomb[i];
        tomb[i] = tomb[tomb.Length-1-i];
        tomb[tomb.Length-1-i] = tmp;
    }
}

public static void Main(string[] args)
{
    int[] ertekek = { 1, 2, 3, 4, 5, 6, 7 };
    Megfordit(ertekek);
    for (int i = 0; i < ertekek.Length; i++)
    {
        Console.WriteLine(ertekek[i]);
    }
}
```

A tömb referencia típus, így a függvény csak a memória címet kapja meg

A tömb értékei megcserélődtek, mert a függvény az eredeti tömböt kezelte

Tömb, mint visszatérési érték

- A függvény adhat vissza tömböt is
 - Tulajdonképpen csak visszaadja a tömb címét, referenciaként

```
static int[] SzamokBekeresese()
{
    int darab = int.Parse(Console.ReadLine());
    int[] szamok = new int[darab];
    for (int i=0; i<darab; i++)
    {
        szamok[i] = int.Parse(Console.ReadLine());
    }
    return szamok;
}

public static void Main(string[] args)
{
    int[] ertekek = SzamokBekeresese();
    for (int i = 0; i < ertekek.Length; i++)
        Console.WriteLine(ertekek[i]);
}
```

Függvény túlterhelés

- A C# lehetőséget ad arra, hogy több ugyanolyan nevű függvényt is létrehozzunk
 - Feltéve hogy a paraméterek száma és/vagy típusa kellőképpen eltér
- A fordító a függvény hívás pontján az átadott paraméterek száma és típusa alapján eldönti, hogy melyiket hívja meg

Függvény túlterhelés példa: eltérő típus

```
static void TombKiir(int[] egeszek)
{
    for (int i = 0; i < egeszek.Length - 1; i++)
        Console.Write(egeszek[i] + ", ");
    Console.WriteLine(egeszek[egeszek.Length-1]);
}
static void TombKiir(double[] lebegok)
{
    for (int i = 0; i < lebegok.Length - 1; i++)
        Console.Write(lebegok[i] + ", ");
    Console.WriteLine(lebegok[lebegok.Length - 1]);
}
public static void Main(string[] args)
{
    int[] ertekek1 = { 2, 4, 5, 1, 5, 7, 2, 7 };
    double[] ertekek2 = { 2.3, 5.1, 7.2, 7.1, 2.6 };

    TombKiir(ertekek1);
    TombKiir(ertekek2);
}
```

Itt az **int**-es verzió hívódik meg

Itt a **double**-s verzió hívódik meg

Függvény túlterhelés példa: eltérő darabszám

```
static int[] Szamsor(int darabszam)
{
    int[] szamok = new int[darabszam];
    for (int i = 0; i < darabszam; i++)
        szamok[i] = i;
    return szamok;
}
static int[] Szamsor(int kezdes, int darabszam)
{
    int[] szamok = new int[darabszam];
    for (int i = 0; i < darabszam; i++)
        szamok[i] = kezdes + i;
    return szamok;
}
public static void Main(string[] args)
{
    TombKiir(Szamsor(5));
    TombKiir(Szamsor(2, 5));
}
```

Első verziót hívja meg: 0, 1, 2, 3, 4

Második verziót hívja meg: 2, 3, 4, 5, 6

Függvények opcionális paraméterekkel

- Olyan lehetőség is van, hogy a függvény bizonyos paramétereit kihagyhassuk a megívás során
- Ez akkor tehető meg, ha van olyan paraméter, amely rendelkezik alapértelmezett értékkel
- Opcionális paraméter csak a paraméterlista végén lehet

Opcionális paraméter példa: számsor

Először a kötelező paraméterek

Azután az opcionálisak

```
static int[] Szamsor(int darabszam, int kezdes = 0, int lepeskoz = 1)
{
    int[] szamok = new int[darabszam];
    for (int i = 0; i < darabszam; i++)
        szamok[i] = kezdes + i*lepeskoz;
    return szamok;
}
```

```
public static void Main(string[] args)
{
    TombKiir(Szamsor(5));
    TombKiir(Szamsor(5, 2));
    TombKiir(Szamsor(5, 2, 3));
}
```

5 darab, 0-tól indul, 1-es lépésköz: 0, 1, 2, 3, 4

5 darab, 2-től indul, 1-es lépésköz: 2, 3, 4, 5, 6

5 darab, 2-től indul, 3-as lépésköz: 2, 5, 8, 11, 14

Opcionális paraméter név szerint

- Az opcionális paraméterek tetszőleges sorrendben megadhatóak, és bármelyik elhagyható
- Ha nem a hagyományos sorrendet követjük, akkor meg kell őket nevezni

```
static int[] Szamsor(int darabszam, int kezdes = 0, int lepeskoz = 1)
{
    int[] szamok = new int[darabszam];
    for (int i = 0; i < darabszam; i++)
        szamok[i] = kezdes + i*lepeskoz;
    return szamok;
}

public static void Main(string[] args)
{
    TombKiir(Szamsor(5, lepeskoz: 3));
    TombKiir(Szamsor(5, lepeskoz: 3, kezdes: 2));
}
```

0, 3, 6, 9, 12

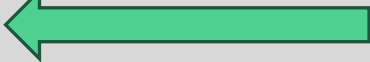
2, 5, 8, 11, 14

NÉVTEREK

Névterek

- Rendszerezik a kódot
- Jelölik a kód egybe tartozó részeit
- Segítik az osztályok és függvények azonosítását, megkülönböztetését
- Segítenek elkerülni a név ütközéseket
- A fő program is egy névtérben van

```
namespace MyProgram  
{  
    class MainClass  
    {  
        public static void Main(string[] args)  
        {  
            ...  
        }  
    }  
}
```



Névtér

- Saját névtér definiálásával jelölhetjük hogy az egyes osztályok és függvények a program mely moduljához, funkciójához tartoznak

```
namespace Neptun
{
    class Tantargy
    {
        public string nev;
        public string kod;
        public int kreditekSzama;
        public int hetiOrakSzama;
    }
}
```

Névtér tartalmának elérése

- A névtéren belüli osztályok teljes neve kibővül a névtér nevével
- A névtéren belüli elemeket a pont segítségével érjük el

```
class MainClass
{
    static void TargyatKiir(Neptun.Tantargy targy)
    {
        Console.WriteLine($"{targy.nev} ({targy.kod}):");
        Console.WriteLine($"    {targy.hetiOrakSzama} ora");
        Console.WriteLine($"    {targy.kreditekSzama} kredit");
    }

    public static void Main(string[] args)
    {
        Neptun.Tantargy targy = new Neptun.Tantargy();
        ...
        TargyatKiir(targy);
    }
}
```


Névtér egyszerűsítése

- Ha egy névtér elemeit gyakran használjuk, és más elemekkel nem okoznak név ütközést, akkor megoldhatjuk, hogy ne kelljen mindig kiírni
- Ezt a **using** segítségével tesszük meg, tipikusan a fájl elején

```
using System;  
using Neptun;
```

- A **Console** a **System** névtér része

```
Console.WriteLine("szoveg");  
System.Console.WriteLine("szoveg");
```