

# Objektumorientált programozás

C# és .NET fejlesztői alapképzés



Bárhol. Bármikor.

# Osztályok és objektumok

## C# programozási alapok



Bárhol. Bármikor.

# Objektumorientált programozás

- A programunk *a valós világban felmerülő* igényre vagy probléma megoldására készül
- A programunkban felismerhető *összefüggő, magasabb szintű dolgokat* fel kell ismernünk
  - Állatkerti jegyrendelés: jegyek (felnőtt/gyermek), árak, akciók
  - Autóversenyzők navigációs rendszere: sisak, HUD, sebességmérő
  - Szuperhősöket katalogizáló wiki: szuperhős, bejegyzés
  - Elvontabb, de továbbra is a valós világban felmerülő igény pl.: egy .NET-hez készülő keretrendszer, ami megkönnyíti a vezérlési szerkezetek használatát
    - Ekkor egy ilyen „összefüggő dolog” maga a *vezérlési szerkezet*

# Osztályok

- Ha azonosítottuk a magasszintű koncepciókat, ezeket a programban **osztályok** formájában definiáljuk
- Az osztály (**class**) ugyanolyan vagy hasonló működéssel bíró dolgok „sablonja”
- Az osztályok önmagukban nem „léteznek”, ameddig nem kezdünk el létrehozni az osztály alapján **példányokat** belőlük
- Az osztályokból létrehozott dolgok példányát **objektumnak** nevezzük
  - Osztálypéldány = objektum
  - Objektumpéldány (redundáns) = objektum
  - Minden objektum tehát *egy konkrét osztály példánya*
- Az osztály egy ún. „típus”; további típusok is léteznek, amiket magunk definiálhatunk (pl. struktúrák, enumok)

# Állatkerti jegyrendelés példa

```
using System;
using System.Collections.Generic;

class Jegy
{
    public DateTime eladasDatuma;
    public string eladoPenztarosNeve;
    public string tipus;
    public decimal eredetiAr;
    public decimal eladasiAr;
}
```

```
class Allatkert
{
    public string nev;
    public List<Jegy> eladottJegyek;

    public void JegyEladasa(string eladoPenztaros, string tipus,
                           decimal eredetiAr, decimal eladasiAr)
    {
        var jegy = new Jegy();
        jegy.eladasDatuma = DateTime.Now;
        jegy.eladoPenztarosNeve = eladoPenztaros;
        jegy.Tipus = tipus;
        jegy.eredetiAr = eredetiAr;
        jegy.eladasiAr = eladasiAr;
        eladottJegyek.Add(jegy);
    }

    public decimal GetOsszesBevetel()
    {
        decimal szumma = 0;
        foreach(var jegy in eladottJegyek)
        {
            szumma += jegy.eladasiAr;
        }
        return szumma;
    }
}
```

# Állatkerti jegyrendelés példa

```
class Jegy
{
    public DateTime eladasDatuma;
    public string eladoPenztarosNeve;
    public string tipus;
    public decimal eredetiAr;
    public decimal eladasiAr;
}

class Allatkert
{
    public string nev;
    public List<Jegy> eladottJegyek;

    public void JegyEladasa(string eladoPenztaros, string tipus,
                           decimal eredetiAr, decimal eladasiAr)
    { /* ... */ }

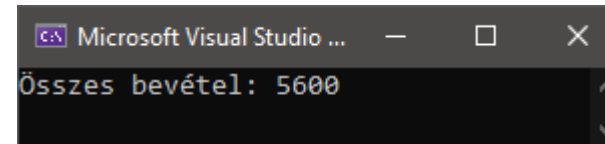
    public decimal GetOsszesBevetel()
    { /* ... */ }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        var allatkert = new Allatkert();
        allatkert.eladottJegyek = new List<Jegy>();
        allatkert.nev = "Fővárosi Állat- és Növénykert";

        allatkert.JegyEladasa("Gábor", "felnőtt", 3000, 2800);
        allatkert.JegyEladasa("Gábor", "felnőtt", 3000, 2800);

        var osszesBevetel = allatkert.GetOsszesBevetel();

        Console.WriteLine($"Összes bevétel: {osszesBevetel}");
    }
}
```



```
Microsoft Visual Studio ...
Összes bevétel: 5600
```

# Az objektumorientáltság gyakorlati haszna

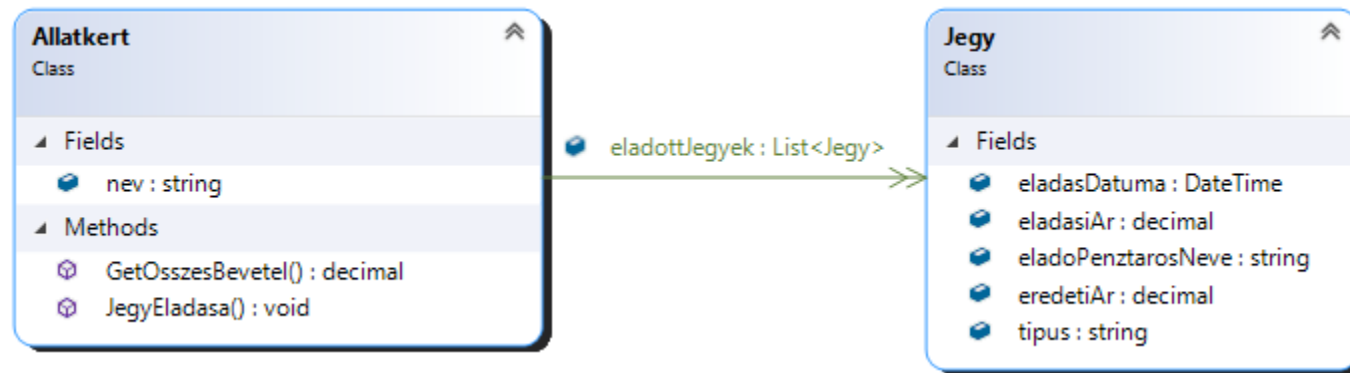
- Ugyanezt az eredményt elérjük az alábbi programmal is:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine($"Összes bevétel: {2800 + 2800}");
    }
}
```

- Minek bonyolítottuk akkor túl?
  - Most már jóval könnyebb az alkalmazást további funkciókkal ellátni!
    - Hány felnőtt/hány gyermek jegyet adtunk el?
    - Mennyi a csak akciós jegyekből befolyó bevétel?
    - Mennyi a tavalyi teljes éves bevétel és a tavalyelőtti bevétel aránya?
    - Melyik pénztáros adta el a legtöbb jegyet?
    - Kezelhetünk több állatkertet is!

# Az objektumorientáltság gyakorlati haszna

Az összefüggések könnyen feltárhatók az osztályaink között, így lehetséges érvelnünk a működésükről





# Az osztályok azonosítása

- Milyen szempontok szerint azonosítjuk a programunkban létrehozandó osztályokat?
  - Adatok (mezők), amelyek közeli kapcsolatban vannak az objektummal, „hozzá tartoznak” (pl. az állatkert neve vagy az általa eladott jegyek)
  - Metódusok (objektumok függvényei), amik (elsősorban) az objektumhoz tartozó adatokon dolgozó műveleteket írnak le (pl. egy jegy eladása)
- Ha mindent egy helyre teszünk: „God object” ☹
- Érdemes törekedni a *minél kisebb* osztályok használatára
  - Ha túlzásba esünk, akkor viszont túl sok kódot kell írunk
  - A programozás egyik legnehezebb feladata megtalálni az arany középutat
- Bármilyen lehet osztály, aminek logikailag van értelme:
  - A doménben („üzletben”), pl. állatkert, jegy stb.
  - Technikailag/műszakilag, pl. Console, Program, List

# Tulajdonságok

## C# programozási alapok



Bárhol. Bármikor.

# Tulajdonságok vs. mezők

- A tulajdonság („property”) objektumunk egy *lekérdezhető és/vagy beállítható* vonatkozása, jellemzője
- A mező („field”) az objektumunk technikai memóriabeli reprezentációja kapcsán hozzá kapcsolódó (nyers) adat
- A tulajdonság tehát az objektumhoz tartozó logikai adat, a mező az objektumhoz tartozó technikai adat

# Tulajdonságok vs. mezők - példa

## Mezők:

```
class Jegy
{
    public DateTime eladasDatuma;
    public string eladoPenztarosNeve;
    public string tipus;
    public decimal eredetiAr;
    public decimal eladasiAr;
}
```

## Tulajdonságok

```
class Jegy
{
    public DateTime EladasDatuma { get; set; }
    public string EladoPenztarosNeve { get; set; }
    public string Tipus { get; set; }
    public decimal EredetiAr { get; set; }
    public decimal EladasiAr { get; set; }
}
```

# Auto-implemented property

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

Használatban nagyon hasonlít a mezőre, *látszólag* ugyanúgy működik:

```
allatkert.Nev = "Fővárosi Állat- és Növénykert";  
Console.WriteLine($"Az állatkert neve: {allatkert.Nev}");
```

# A tulajdonság

Megadja a lehetőséget arra, hogy mi magunk implementáljuk a lekérdezés és a módosítás műveletét:

```
private string nev;  
public string Nev  
{  
    get { return nev; }  
    set { nev = value; }  
}
```

Akár bonyolultabban:

```
private string nev;  
public string Nev  
{  
    get { return nev ?? "-Ismeretlen-"; }  
    set  
    {  
        if (value.Length > 50)  
            nev = value[..50];  
        else  
            nev = value;  
    }  
}
```

# Jó tudni a tulajdonságokról...

- Lehet csak *get*, csak *set* vagy mindkettő
- Az autoimplementált property ugyanúgy viselkedik, mint a mező
- Megfeleltethető egy *x* *getX()* és egy *void setX(x)* függvénynek
  - A háttérben a fordító valójában ezt generálja!
- A teljes („full”) tulajdonság mögött van normál mező is
- Egyedi implementációban nem „illik” kivételt dobni
- Használhatjuk az *expression bodied* property szintaxist is:

```
private string nev;  
public string Nev  
{  
    get => nev ?? "-Ismeretlen-";  
    set => nev = value.Length > 50 ? value[..50] : value;  
}
```

- Csak lekérdezhető tulajdonság:

```
private string nev;  
public string Nev => nev ?? "-Ismeretlen-";
```

# Ökölszabályok

- Mindig tulajdonságot használjunk mező helyett!
- Ne használjunk publikus mezőket!
- Az legyen lekérdezhető tulajdonság, ami az objektum adott állapotát reprezentálja
  - Tehát kerüljük a X GetX() függvényeket!
- Lehetnek egymásba áthívó tulajdonságaink is (pl. HomersekletC/HomersekletF)
- Ne használjunk „drága” számításokat a tulajdonságok megvalósításában!
  - Ha muszáj, ilyenkor mégis függvényekben valósítsuk meg!



# Konstruktorok, láthatóság

## C# programozási alapok

# Konstruktorok

- Amikor létrehoztuk az állatkert objektumot, akkor még nem volt neve...

```
var allatkert = new Allatkert();  
allatkert.EladottJegyek = new List<Jegy>();  
allatkert.Nev = "Fővárosi Állat- és Növénykert"; // Csak itt állítottuk be a nevét
```

- A konstruktorfüggvény („konstruktor”, „constructor”) arra szolgál, hogy miután létrehozzuk az objektumot, az már *stabil állapotban* jöjjön létre
  - Inicializáljuk a tagokat, hogy a konstruktor végére érve már egy „jó” objektumunk legyen
  - Ellenőrizzük a paramétereket, hogy helyes értékek legyenek
- A konstruktorfüggvény neve az osztály nevével megegyezik és nem tér vissza semmivel, de paramétereket várhat

# Konstruktor: példa

```
class Allatkert
{
    public string Nev { get; set; }
    public List<Jegy> EladottJegyek { get; set; }

    public Allatkert(string nev)
    {
        Nev = nev;
        EladottJegyek = new List<Jegy>();
    }
}
```

```
class Allatkert
{
    public string Nev { get; }
    public List<Jegy> EladottJegyek { get; }

    public Allatkert(string nev)
    {
        Nev = nev;
        EladottJegyek = new List<Jegy>();
    }
}
```

```
class Allatkert
{
    public string Nev { get; private set; }
    public List<Jegy> EladottJegyek { get; private set; }

    public Allatkert(string nev)
    {
        Nev = nev;
        EladottJegyek = new List<Jegy>();
    }
}
```

```
class Allatkert
{
    public string Nev { get; }
    public List<Jegy> EladottJegyek { get; } = new List<Jegy>();

    public Allatkert(string nev) => Nev = nev;
}
```

# Több konstruktor

- Egy objektumnak több konstruktora is lehet
  - Mivel a konstruktor nem tér vissza semmivel és a neve adott, ezért csak a paraméterlistában különböznek

```
class Allatkert
{
    public string Nev { get; }
    public List<Jegy> EladottJegyek { get; } = new List<Jegy>();

    public Allatkert(string nev) => Nev = nev;

    public Allatkert() => Nev = "-Ismeretlen-";
}
```

# A new kulcsszó

- A konstruktorokat a new kulcsszóval hívjuk meg

```
var allatkert = new Allatkert();
```

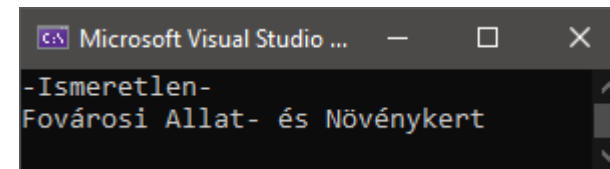
▲ 1 of 2 ▼ Allatkert()

```
var allatkert = new Allatkert();
```

▲ 2 of 2 ▼ Allatkert(string nev)

- Az IntelliSense segít, hogy lássuk, milyen paraméterezései lehetnek a típus (pl. osztály) konstruktorainak
- Ha a céltípus (a változó típusa) a konstruktor hívásának pillanatában már ismert, akkor a típus neve elhagyható

```
Allatkert ismeretlenAllatkert = new();  
Console.WriteLine(ismeretlenAllatkert.Nev);  
  
Allatkert fovarosiAllatkert = new("Fővárosi Állat- és Növénykert");  
Console.WriteLine(fovarosiAllatkert.Nev);
```



```
Microsoft Visual Studio ...  
-Ismeretlen-  
Fővárosi Állat- és Növénykert
```

- Ez jellemzően hosszú/bonyolult típusnevek esetén hasznos, vagy ha nem deklarációval együtt példányosítunk

# Láthatóság

- Eddig több helyen láttuk már a `public` kulcsszót...
- Nem minden esetben akarjuk, hogy az objektumunk belső állapotát „mások” piszkálják
  - Az egyik objektumorientált alapkoncepció a *felelősségi körök szétválasztása* („separation of concerns”)
  - Az osztályokat is úgy fogalmazzuk meg, hogy a saját adataikon dolgozzanak
- Az objektumunk szeretné a tagjait (tulajdonságok, mezők, metódusok stb.) kívülről elérhetővé tenni, akkor a `public` kulcsszót kell használni
  - Különben a tag „privát” lesz, tehát csak az osztályon belül elérhető
  - A `public` elhagyása tagok esetében ugyanaz, mintha `private` kulcsszót írnánk
  - A szándék kifejezése érdekében **érdemes mindig kiírni** a láthatósági módosítószt

# Láthatóság: példa

- A konkrét jegyeket az állatkertben nem kell publikálnunk, elég, ha az állatkert maga ismeri őket:

```
private List<Jegy> EladottJegyek { get; } = new();
```

- Ugyanakkor lehet, hogy szeretnénk elérhetővé tenni kívülről, de beállítani csak mi szeretnénk tudni:

```
public List<Jegy> EladottJegyek { get; private set; } = new();
```

- Metódusok esetén szintén használjuk:

```
private string GetLegjobbDolgozo() { /*...*/ }
```

# További láthatósági módosítók

- `protected`: csak saját maga és leszármazott számára elérhető
- `internal`: csak az aktuális szerelvényből (projektből) elérhető
- `protected internal`: a szerelvényből vagy leszármazottból elérhető
- `private protected`: a szerelvényen belül leszármazottból elérhető



# Konstruktorok: jó tudni

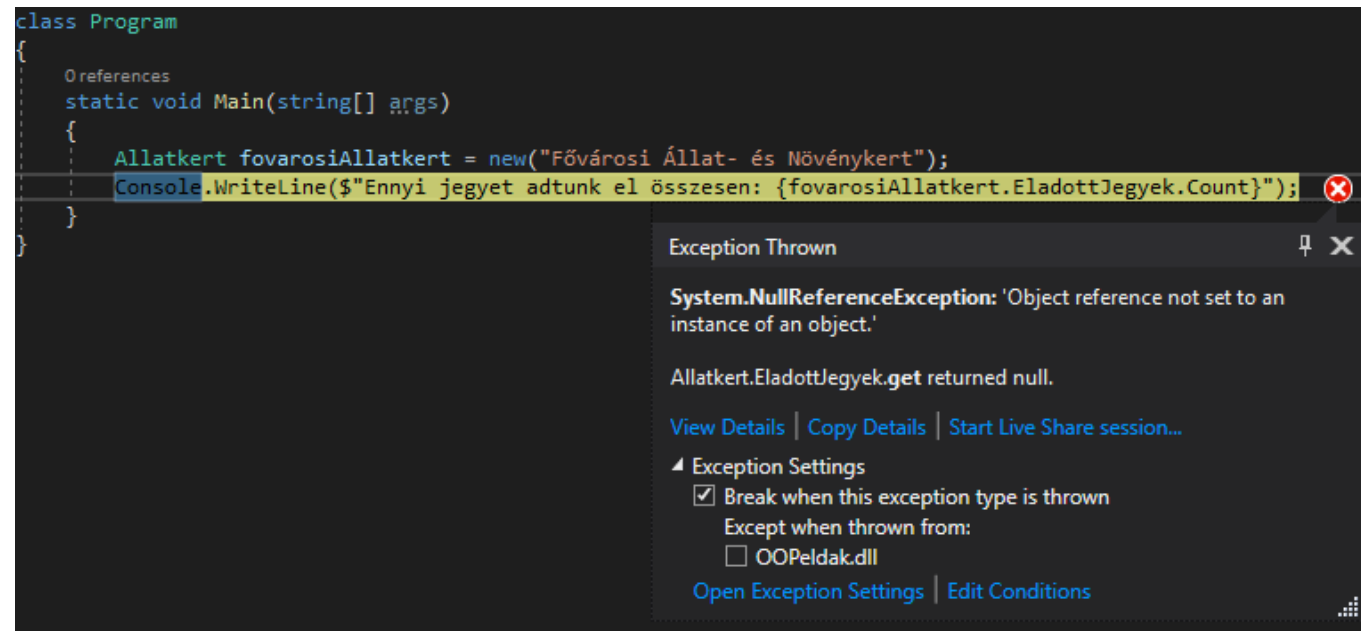
- Ha nem készítünk konstruktort, úgy létrejön egy ún. alapértelmezett („default”) konstruktor, ami publikus, nem vár paramétert, és nem csinál semmit
- Ha készítünk bármiféle konstruktort, a default konstruktor nem jön létre
- Konstruktoroknak is van láthatósága...
  - Ha a konstruktor privát, az objektum csak saját magán belül példányosítható, de mivel nem tudunk létrehozni példányt, ezért nem példányosítható saját magán belül sem
    - Erre is szolgál a később bemutatandó statikus objektum
- Ha egy objektumot megpróbálunk létrehozni (new), akkor az vagy teljesen létrejön és megkapjuk a new eredményeképp, vagy kivétel lesz, és nem kapunk semmilyen (hibás) objektumot

# A milliárd dolláros hiba: NullReferenceException

- C# programozási alapok

# Mi történik, ha olyan változót használunk, aminek nem adtunk értéket?

```
Allatkert fovarosiAllatkert = new("Fővárosi Állat- és Növénykert");  
Console.WriteLine($"Ennyi jegyet adtunk el összesen: {fovarosiAllatkert.EladottJegyek.Count}");
```



A milliárd dolláros hiba: NullReferenceException

# null

- Azt az objektumot, aminek nincsen értéke, a „null” érték reprezentálja
- Explicit is adhatunk egy objektumnak null értéket
- Az objektumon ekkor bármilyen tagját próbáljuk meg elérni, `NullPointerException` (NRE) kivételt kapunk
- Tony Hoare találta ki a koncepciót, 2009-ben saját „milliárd dolláros hibájának” nevezte<sup>[1]</sup>

[1] <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/>

# null!

- A C#-ban elérhető egy funkció, amivel a fordítónak jelezhetjük, hogy egy objektum szabad, hogy `null` legyen
  - Csak a kivételes esetet kell jelezni tehát, azt nem, ha egy objektum *nem* lehet `null`
- Ehhez a projektünket leíró `.csproj` fájlba fel kell venni az alábbi értéket:

```
<Project Sdk="Microsoft.NET.Sdk">  
  <PropertyGroup>  
    <OutputType>Exe</OutputType>  
    <TargetFramework>net5.0</TargetFramework>  
    <Nullable>enable</Nullable>  
  </PropertyGroup>  
</Project>
```

# null?

- Nullable=enable esetén jelezhetjük, hogy az EladottJegyek lehet null értékű is egy ? karakterrel:

```
class Allatkert
{
    public string Nev { get; }
    public List<Jegy>? EladottJegyek { get; set; }
    public Allatkert(string nev) => Nev = nev;
    public Allatkert() => Nev = "-Ismeretlen-";
}
```

```
Allatkert fovarosiAllatkert = new("Fővárosi Állat- és Növénykert");
Console.WriteLine($"Ennyi jegyet adtunk el összesen: {fovarosiAllatkert.EladottJegyek.Count}");
```



🔧 List<Jegy>? Allatkert.EladottJegyek { get; set; }

'EladottJegyek' may be null here.

CS8602: Dereference of a possibly null reference.

Show potential fixes (Alt+Enter or Ctrl+.)

? .      ??      ?? =      ! .

`fovarosiAllatkert.EladottJegyek?.Count`

- Ha `EladottJegyek` `null`, akkor `null`, különben `EladottJegyek.Count`

`(fovarosiAllatkert.EladottJegyek ?? new()).Count;`

- Ha `EladottJegyek` nem `null`, akkor annak, egyébként egy új `List<Jegy>()`-nek a `Count`-ját adja vissza

`fovarosiAllatkert.EladottJegyek ??= new();`

- Ha `EladottJegyek` `null`, értékül adunk neki egy új `List<Jegy>()` listát

`fovarosiAllatkert.EladottJegyek!.Count`

- Csak elcsendesítjük a fordítót, így nem ad figyelmeztetést
- Ugyanaz, mintha simán ponttal érnénk el a tagot, ha nincs bekapcsolva a `Nullable`
- Járjunk el különösképp körültekintően, mert `NullReferenceException` keletkezhet!

A  Elvis operátor néven is ismert

A milliárd dolláros hiba: `NullReferenceException`

# Leszármazás, absztrakció, interfészek

C# programozási alapok





# Leszármazás

- Gyakori, hogy szeretnénk kibővíteni egy meglévő osztály funkcióit úgy, hogy a meglévő osztályhoz nem akarunk (vagy tudunk) hozzányúlni
- A leszármazás egy ún. „leszármazási hierarchiát” ír le: egy objektumnak mindig egyetlen „ősosztálya” lehet
- Sőt, minden saját osztályunk egyébként is az Object osztályból származik!
- A leszármazást az osztály neve után kettősponttal jelöljük
- A leszármazottból az őst a base kulcsszóval érjük el
  - Saját magát a `this`-szel, de az redundáns, ezért nagyon ritkán használjuk

# Object

A `System.Object` minden objektum (közvetlen vagy közvetett) őse

```
Assembly System.Runtime, Version=5.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a

#nullable enable

namespace System
{
    ...public class Object
    {
        ...public Object();
        ...~Object();

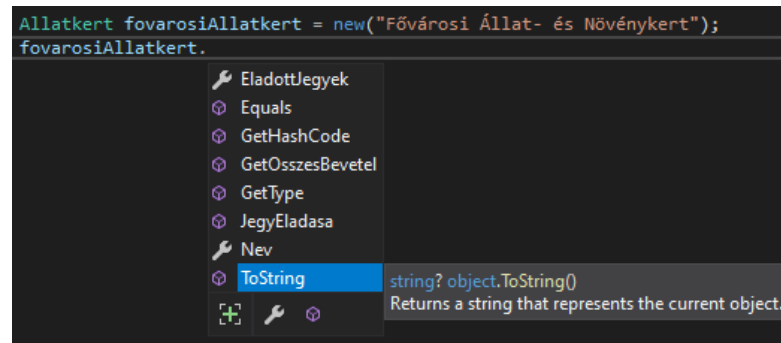
        ...public static bool Equals(Object? objA, Object? objB);
        ...public static bool ReferenceEquals(Object? objA, Object? objB);
        ...public virtual bool Equals(Object? obj);
        ...public virtual int GetHashCode();
        ...public Type GetType();
        ...public virtual string? ToString();
        ...protected Object MemberwiseClone();
    }
}
```

# Object

Az Allatkert is származik az Object-ből, tehát ugyanaz, mintha így hoztuk volna létre:

```
class Allatkert : Object
{
    /* ... */
}
```

A leszármazott megörökli az ősből definiált tagokat:



# Saját leszármazási lánc

- Tehát saját osztályt is készíthetünk, amiből leszármazhatunk...
- Legyen bármely létesítménynek **neve**, tudjon **eladni jegyeket**
- Minden létesítmény tudja (magáról) megmondani, hogy mennyi az összes bevétele
- Legyen az állatkert egy *speciális* létesítmény, ahol állatok is vannak

# Saját leszármazási lánc: példa

```
class Letesitmeny
{
    1reference
    public string Nev { get; }
    2references
    public List<Jegy> EladottJegyek { get; } = new();
    0references
    public Letesitmeny(string nev) => Nev = nev;

    0references
    public void JegyEladasa(string eladoPenztaros, string tipus, decimal eredetiAr, decimal eladasiAr) {...}

    0references
    protected decimal GetOsszesBevetel() {...}
}
```

```
class Allat
{
    1reference
    public string Nev { get; }
    1reference
    public DateTime SzuletesiDatum { get; }

    0references
    public Allat(string nev, DateTime szuletesiDatum)
    {
        Nev = nev;
        SzuletesiDatum = szuletesiDatum;
    }
}
```

```
class Allatkert : Letesitmeny
{
    0 references
    public List<Allat> Allatok { get; } = new();
}
```

```
class Allatkert : Letesitmeny
{
    1 reference
    public Allatkert(string nev) : base(nev) { }
    0 references
    public List<Allat> Allatok { get; } = new();
}
```

- ❌ CS7036 There is no argument given that corresponds to the required formal parameter 'nev' of 'Letesitmeny.Letesitmeny(string)'
- ❌ CS1729 'Allatkert' does not contain a constructor that takes 1 arguments

# Konstruktorok és leszármazás

- Ha az ősnek nincsen paraméter nélküli konstruktora, a leszármazott konstruktorainak explicit kell megmondani, melyik ősbeli konstruktort hívják meg
- A konstruktorok a leszármazási láncban „lefelé” hívódnak meg
  - Először tehát az Object konstruktora hívódik meg (mert minden osztály abból származik), aztán az első leszármazott fut végig, aztán a második és így tovább, végül pedig a saját konstruktor
  - Így garantálható, hogy az ős által definiált állapot már „stabil”, amikor a saját konstruktorunk törzse fut

# Absztrakt osztályok

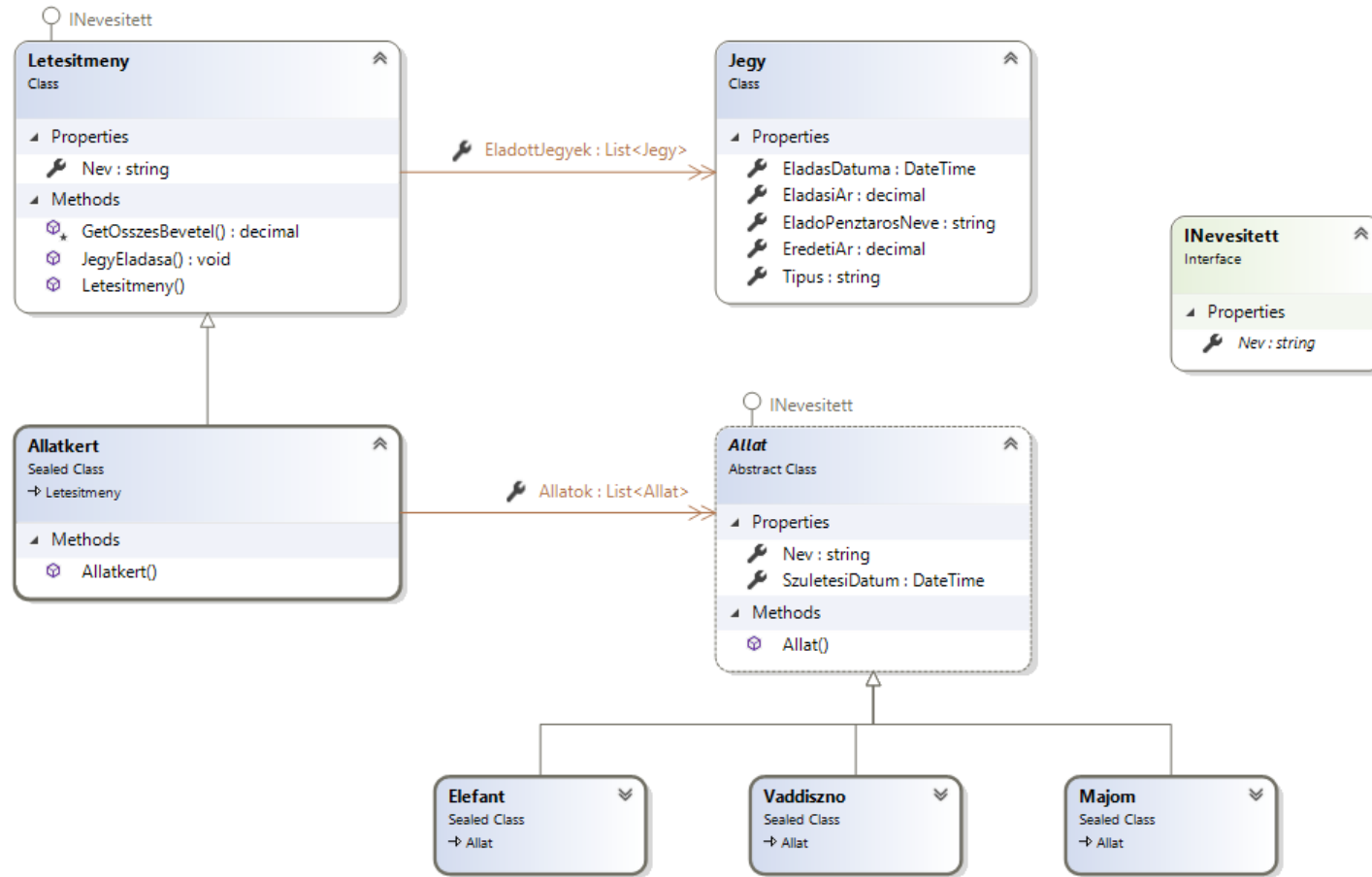
- Az `abstract` kulcsszóval olyan osztály hozható létre, amelyből le lehet származni, de nem lehet belőle önálló példányt létrehozni
  - Esetünkben például a `Letesitmeny` ne legyen absztrakt, mert nem feltétlenül szükséges, ha nem akarjuk a cirkuszokat speciálisan kezelni
  - Az `Allat` viszont lehetne absztrakt, ha minden esetben a konkrét állatra van szükségünk (pl. `Elefant`, `Vaddiszno`, `Majom`)
- A `sealed` kulcsszóval a leszármazás meggátolható, tehát ha nem szeretnénk, hogy valaki a `Majom` osztályból leszármazzon, akkor az osztályt a `sealed class` kulcsszavakkal kell definiálni

# Interfészek

- Interfészt osztályhoz hasonlóan definiálunk, de nem a `class`, hanem az `interface` kulcsszóval
- Az `interface` nem példányosítható, hanem csak elvárásokat fogalmaz meg a megvalósítókkal szemben
- Lehet tehát egy interfészünk, pl. „`INevesített`”, ami elvárja, hogy legyen az objektumnak `Nev` lekérdezhető tulajdonsága
- Ezt az interfészt a `Letesitmeny` és az `Allat` is megvalósíthatja, így minden belőle leszármazó is (örökletesen) megvalósítja



# Saját leszármazási lánc: példa



# Absztrakció, interfészek: miért?

- Ha készítünk egy metódust, ami szeretné kiírni egy létesítmény nevét a konzolra, akkor végülis miért nem elég nekünk egy olyan objektum, aminek neve van?
- Ezzel a megközelítéssel meg tudjuk érteni egy másik fontos objektumorientált alapvetés, az „interfész szegregáció” alapgondolatát
  - Mindig csak annyira absztrakt dolgon dolgozzunk, amennyire éppen szükséges
  - Így az implementációnk újrahasznosítható lesz
- Ha vannak interfészeink és őssosztályaink, azokat kezelhetjük egységesen
  - A különféle állatokat gond nélkül kezelhettük egy állatokat gyűjtő listában

# Felülírás

- Az ősből definiált virtuális (`virtual` kulcsszóval ellátott) tagokat a leszármazottak opcionálisan felüldefiniálhatják az `override` kulcsszóval
- Mivel mindenki származik az `Object`-ből, ezért például a `ToString()` metódus minden objektum által felüldefiniálható
- Sőt, a felülírás elvárható, ha a tagot `abstract` kulcsszóval látjuk el

# Statikus objektumok

## C# programozási alapok



Bárhol. Bármikor.

# Statikus tagok

- Statikus: időben nem változó (vö. dinamikus)
- Bármilyen osztálynak lehetnek statikus tagjai, ezeket a `static` kulcsszóval jelöljük
- A statikus tagokat az osztályon (nem pedig a példányon) keresztül érjük el
  - Pl.: `Console.WriteLine();`
- A statikus tagok a *statikus példányhoz* tartoznak, nem bármely példányhoz

# Statikus osztály

- Bármely osztályt megjelölhetjük statikusként a `static` kulcsszóval
- Ekkor az osztályt nem példányosíthatjuk a `new` kulcsszóval, hanem az osztály *nevén* keresztül érhetjük el a további statikus tagokat
- A statikus osztálynak csak statikus tagjai lehetnek

# Statikus osztály példa: a Console

```
namespace System
{
    ...public static class Console
    {
        ...public static bool IsInputRedirected { get; }
        ...public static int BufferHeight { get; set; }
        ...public static int BufferWidth { get; set; }
        ...public static bool CapsLock { get; }
        ...public static int CursorLeft { get; set; }
        ...public static int CursorSize { get; set; }
        ...public static int CursorTop { get; set; }
        ...public static bool CursorVisible { get; set; }
        ...public static TextWriter Error { get; }
        ...public static ConsoleColor ForegroundColor { get; set; }
        ...public static TextReader In { get; }
        ...public static Encoding InputEncoding { get; set; }
        ...public static bool IsErrorRedirected { get; }
        ...public static int WindowWidth { get; set; }
        ...public static bool IsOutputRedirected { get; }
        ...public static bool KeyAvailable { get; }
        ...public static int LargestWindowHeight { get; }
        ...public static int LargestWindowWidth { get; }
        ...public static bool NumberLock { get; }
        ...public static TextWriter Out { get; }
        ...public static Encoding OutputEncoding { get; set; }
        ...public static string Title { get; set; }
        ...public static bool TreatControlCAsInput { get; set; }
        ...public static int WindowHeight { get; set; }
        ...public static int WindowLeft { get; set; }
    }
}
```

# Statikus konstruktor

- A statikus osztályoknak is lehet konstruktora, viszont ezt nem mi döntjük el, hogy mikor hívjuk meg
  - Az osztályhoz történő első hozzáférés előtt hívódik meg
- Fontos, hogy a statikus konstruktor semmiképp ne dobjon kivételt, mert ha mégis (és le is kezeljük a keletkező `TypeInitializationException`-t), akkor az alkalmazás futásának végéig már nem fogjuk tudni elérni a típust
- A statikus konstruktort értelemszerűen `static Tipusnev() { ... }` szintaxissal írjuk; nem várhat semmilyen paramétert és nem térhet vissza semmivel
  - Miért...?



# Konstruktorok és statikus konstruktor

- Az alábbi sorrendben hívódnak meg a konstruktorok leszármazás és statikus konstruktorok esetén:
  - A leszármazási láncban fentről lefelé a *statikus* konstruktorok
  - A leszármazási láncban fentről lefelé a *példány* konstruktorok

# Gyakori minta: „az egyke”

- Az „egyke” („singleton”) egy tervezési minta, amivel kikényszeríthetjük, hogy egy osztályból csak legfeljebb egy példány jöjjön létre

```
public sealed class Egyke
{
    private Egyke() { /* Más ne példányosítson! */ }
    private static Egyke? instance;
    public static Egyke Instance
    {
        get
        {
            if (instance == null)
                instance = new Egyke();
            return instance;
        }
    }
}
```

```
public sealed class Egyke
{
    private Egyke() { /* Más ne példányosítson! */ }
    private static Egyke? instance;
    public static Egyke Instance => instance ??= new Egyke();
}
```

# Gyakori hiba: „static smell”

- A programozásban a ránézésre nem stimmelő (átgondolatlan, hibás elképzelésre épülő, túlbonyolított, esetleg tapasztalatlanságból adódó stb.) kódbeli mintázatok „code smell”-eknek nevezzük
- Egy példa erre a static (túl) gyakori használata:
  - Kellene egy statikus tag → vegyük fel!
  - A statikus tagból el kellene tudnunk érni nem statikus tagokat is... → legyen az is statikus!
  - Előbb-utóbb minden statikus lesz, és elveszítettük az objektumorientáltság minden előnyét
    - Ha csak statikus osztályaink/tagjaink vannak, akkor gyakorlatilag „csak” globális állapotunk és azon dolgozó függvényeink vannak → visszatértünk a 20. századba

# A static rossz?

- Nem, a static nem rossz 😊
- Viszont nagyon kevés probléma van, amit programozóként megold
  - Annál több problémát tud viszont kreálni
- Jó ökölszabály: *semmi ne legyen statikus!*
  - Az ún. *függőséginjektálás* miatt nem az osztály felelőssége eldönteni, hogy belőle egy vagy több példány szerepeljen
  - A leggyakoribb kivétel a „bővítő metódus” („extension method”)

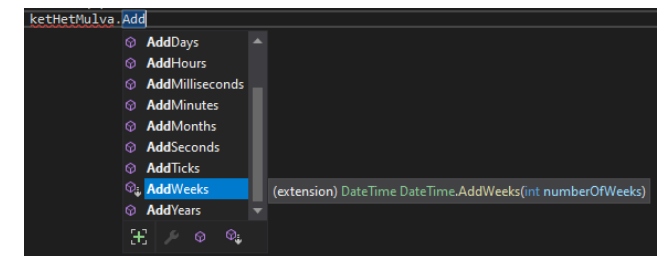
# Extension method

- Ezzel a konstrukcióval látszólag egy általunk nem hozzáférhető típusra rá tudunk tenni egy új metódust

```
public static class DateTimeExtensions
{
    public static DateTime AddWeeks(this DateTime dateTime, int numberOfWeeks)
    {
        return dateTime.AddDays(numberOfWeeks * 7);
    }
}
```

- Statikus osztályban statikus metódus, aminek első paramétere `this` módosítóval van ellátva

```
var ma = DateTime.Now; // Statikus tulajdonság!
var ketHetMulva = ma.AddWeeks(2); // Ezt a metódust mi írtuk!
```



# Referenciák és struktúrák

## C# programozási alapok



Bárhol. Bármikor.

# Referenciák

- Ahányszor csak példányosítunk egy objektumot vagy létrehozunk egy változót, az helyet foglal a memóriában (RAM-ban)
- A memóriában bináris adat található (0-k és 1-ek)
- A RAM-ot a futtatókörnyezet (.NET) egész számokkal (`int`) címezi meg, ami 32-64 bites (CPU architektúrától függően)
- A .NET szigorúan típusos, mindig tudjuk, hogy milyen típusú objektumot keresünk a memóriában a megadott helyen
  - Kis túlzással „mindig”

# Referenciák

- A „változó” valójában nem más, mint a memória egy szeletére mutató szám (C++-ban „pointer”)
- Ha tehát kérem az „allatkert” változót, az alábbi történik:
  - A futtatókörnyezet kiértékeli az allatkert változóhoz tartozó int értékét
  - A megadott címnek megfelelő helyre ugrik a memóriában
  - A címen található bináris adatot „Allatkert” típusként értelmezi
- .NET-ben ez a mutató valójában egy „*mutatóra mutató mutató*”, amit **referenciának** hívunk
  - A keretrendszer a háttérben odébb tudja tenni a memóriában a változónkat, hogy az operációs rendszernek egybefüggő memóriaterület szabadítson fel
  - Mi ebből nem látunk semmit, mert a „külső” mutatónk nem változik



# Struktúrák

- A struktúrák nagyon hasonlítanak az osztályokra, de őket a `class` helyett a `struct` kulcsszóval definiáljuk
- A struktúrák ún. **értéktípusok**, ők **nem referenciák**
  - Más értéktípusok beépítettek, pl.:
    - Számok: (s)byte, (u)short, (u)int, (u)long, float, double, decimal
    - A karakter (valójában szám): char
    - Igaz-hamis (Boolean) értékek: bool (true/false)
- Külön „helyen” vannak a memóriában:
  - A referenciák a RAM-ban az ún. veremben (heap) tárolódnak
  - A struktúrák a RAM-ban az ún. halomban (stack) tárolódnak
  - Ezeket a helyeket a keretrendszer (.NET) jelöli ki a program indulásakor

# Speciális referencia: a `string`

- A `string` minden típus közül a legspeciálisabb
- A háttérben valójában egy értéktípus, de referenciatípusként működik
- Fel sem tűnik használat közben, de egy `string` immutábilis (nem módosítható)
- Valójában minden `string` művelet (pl. összefűzés, kivágás) egy **új példányt** hoz nekünk létre
  - Ezért nagyon fontos tisztában lenni vele, hogy a gyakori/tömeges `string` műveletek nagyon erőforrásigényesek lehetnek
  - Kivétel, ha az eredményük ugyanaz, akkor a két `string` referenciánk valójában csak egyszer van a memóriában, de két referenciánk van rá

# Referenciák és struktúrák működése

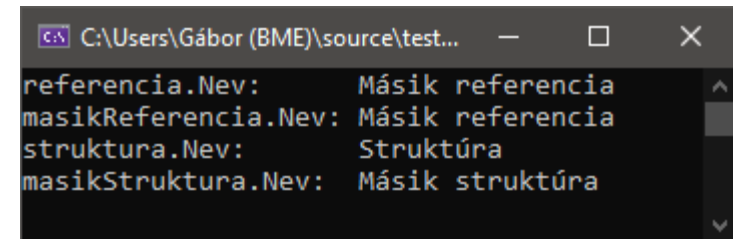
```
public class Referencia
{
    public Referencia(string nev) => Nev = nev;
    public string Nev { get; set; }
}
```

```
public struct Struktura
{
    public Struktura(string nev) => Nev = nev;
    public string Nev { get; set; }
}
```

```
var referencia = new Referencia("Referencia");
var masikReferencia = Muveletek.NevValtoztatas(referencia, "Másik referencia");
Console.WriteLine($"referencia.Nev: {referencia.Nev}");
Console.WriteLine($"masikReferencia.Nev: {masikReferencia.Nev}");
```

```
var struktura = new Struktura("Struktúra");
var masikStruktura = Muveletek.NevValtoztatas(struktura, "Másik struktúra");
Console.WriteLine($"struktura.Nev: {struktura.Nev}");
Console.WriteLine($"masikStruktura.Nev: {masikStruktura.Nev}");
```

```
public static class Muveletek
{
    public static Referencia NevValtoztatas(
        Referencia referencia, string ujNev)
    {
        referencia.Nev = ujNev;
        return referencia;
    }
    public static Struktura NevValtoztatas(
        Struktura struktura, string ujNev)
    {
        struktura.Nev = ujNev;
        return struktura;
    }
}
```



```
C:\Users\Gábor (BME)\source\test...
referencia.Nev:    Másik referencia
masikReferencia.Nev: Másik referencia
struktura.Nev:    Struktúra
masikStruktura.Nev: Másik struktúra
```

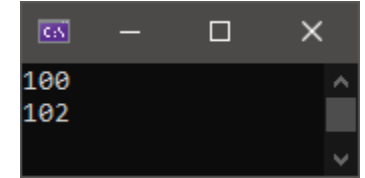
# Mi történt?

- A referenciatípusok **referenciája** kerül átadásra függvényeknek, tehát a függvény ugyanazon az objektumon dolgozik, mint a hívó
- A struktúrákból egy **másolat** kerül átadásra függvényeknek, tehát a függvény nem ugyanazt a példányt használja, mint hívó

# Miért?

```
static int PluszKetto(int szam)
{
    szam = szam + 2;
    return szam;
}
```

```
var szam = 100;
var szam2 = PluszKetto(szam);
Console.WriteLine(szam);
Console.WriteLine(szam2);
```

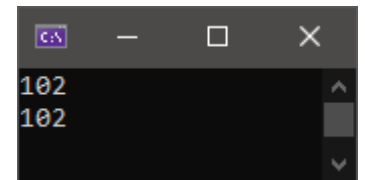


```
C# 100 102
```

- Az értéktípusok kezelése emberi intuíció miatt működik így
  - Ha definiálok egy számot, azt átadom egy függvénynek, attól még az én számom nem változik!
- Ettől is el tudunk térni, ha akarunk: „ref”

```
static int PluszKetto(ref int szam)
{
    szam = szam + 2;
    return ref szam;
}
```

```
var szam = 100;
var szam2 = PluszKetto(ref szam);
Console.WriteLine(szam);
Console.WriteLine(szam2);
```



```
C# 102 102
```

# Enumok, konverziók, operátorok

C# programozási alapok



Bárhol. Bármikor.

# Enumok

- Az enum (felsorolás típus, „enumeration type”) egy olyan értéktípus, ami egyszerű értékek felsorolt, nevesített, meghatározott értékkészletét tartalmazza
- A háttérben az enum valójában egy `int`
  - Ha szeretnénk, szabályozható más számtípusra is (pl. `long`, `byte`)
- Az enum legfontosabb előnye, hogy nem „varázs konstansok” használatát részesítjük előnyben, ha korlátozott értékkészletű lehetőségeket vehet fel egy érték

# Enumok: példa

```
public enum JegyTipus
{
    Felnott,
    Gyermeke,
    Nyugdijas,
    Csoportos
}
```

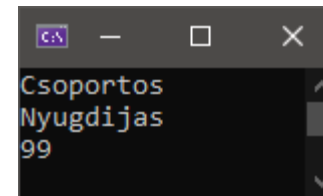
```
public enum JegyTipus : int
{
    Felnott = 0,
    Gyermeke = 1,
    Nyugdijas = 2,
    Csoportos = 3
}
```

```
public enum JegyTipus
{
    Felnott = 5,
    Gyermeke, // 6
    Nyugdijas = 2,
    Csoportos // 3
}
```

```
var jegy = new Jegy();
jegy.Tipus = JegyTipus.Csoportos;
Console.WriteLine(jegy.Tipus);
```

```
jegy.Tipus = (JegyTipus)2;
Console.WriteLine(jegy.Tipus);
```

```
jegy.Tipus = (JegyTipus)99;
Console.WriteLine(jegy.Tipus);
```





# Típuskonverziók

- `jegy.Tipus = (JegyTipus)2;`
- Az egyszerű 2 számértékből `JegyTipus` típusú enum értéket készítettünk
  - Ez azért lehetséges, mert az enum egy speciális `int` példány
  - Akkor lehet ezt használni más „speciális példány” esetben is?
- (Explicit) típuskonverzió („kasztolás”, „cast”): egy típust másik típusúvá konvertálni a típus zárójelben való megjelölésével lehet
  - Ha a típus konvertálható (pl. mert leszármazottja az elemnek), akkor a konverzió sikeres, különben kivételt kapunk
  - A típust „biztonságosan” is konvertálhatjuk az `as` kulcsszóval és megvizsgálhatjuk, hogy a típus adott típus (vagy leszármazott) az `is` kulcsszóval

# Típuskonverziók: példa

```
static void Etetes(Allat allat)
{
    if (allat is Majom)
    {
        Console.WriteLine($"Ez egy majom: {allat.Nev}");
        // Ekkor szabad kasztolni:
        var m1 = (Majom)allat;
        m1.Bananok++;
    }
}

if (allat is Majom m2) // allat és m2 ugyanaz az objektum!
{
    Console.WriteLine($"Ez egy majom: {m2.Nev}");
    m2.Bananok++;
}
```

```
var m3 = allat as Majom;
if (m3 != null) // Ha allat valóban Majom típusú
{
    Console.WriteLine($"Ez egy majom: {m3.Nev}");
    m3.Bananok++;
}

try
{
    var m4 = (Majom)allat;
    Console.WriteLine($"Ez egy majom: {m4.Nev}");
    m4.Bananok++;
}
catch (InvalidCastException)
{
    // Nem Majom
}
```

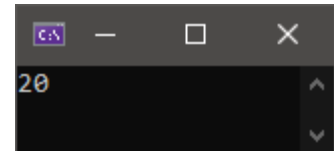
# Saját konverziók

- Van lehetőségünk saját konverziókat is írni
- Ezek speciális szignatúrájú statikus metódusok

```
sealed class Majom : Allat
{
    public Majom(string nev, DateTime születésiDatum) : base(nev, születésiDatum) { }
    public int Bananok { get; set; }

    public static explicit operator int(Majom m) => m.Bananok;
}
```

```
var vilma = new Majom("Vilma", new DateTime(2015, 07, 01));
vilma.Bananok = 20;
var vilmaMintSzam = (int)vilma;
Console.WriteLine(vilmaMintSzam);
```



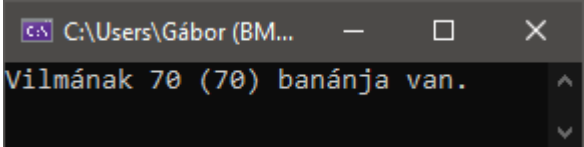
# Operátor túltöltés

- A beépített operátorok (+, -, --, && stb.) „túltölthetők” („overload”), így saját értelmet nyerhetnek:

```
sealed class Majom : Allat
{
    /* ... */
    public static int operator +(Majom m, int bananok)
    {
        majom.Bananok += bananok;
        return majom.Bananok;
    }
}

var vilma = new Majom("Vilma", new DateTime(2015, 07, 01));
vilma.Bananok = 20;

var ujBananok = vilma + 50;
Console.WriteLine($"Vilmának {ujBananok} ({vilma.Bananok}) banánja van.");
```



C:\Users\Gábor (BM... — □ ×  
Vilmának 70 (70) banánja van.

# Tömbök és listák

## C# programozási alapok



Bárhol. Bármikor.

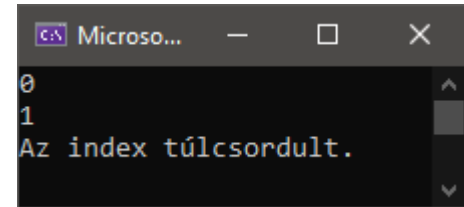
# Tömb

- A tömb („array”) a legegyszerűbb kollektíótípus
- Több elem tárolására alkalmas
- Típusos, tehát megadhatjuk (illik), hogy milyen elemeket szeretnénk benne tárolni
- A tömbnek kezdeti mérete van, amit vagy az inicializáláskor beletett elemek, vagy üres tömb esetén a tömb mérete explicit határoz meg
- A tömb **mérete nem módosítható**, ezért hozzáadni új elemet nem tudunk, csak a tömb adott indexében elhelyezni egyet (vagy azt eltávolítani)

# Tömbök: példa

```
var szamok = new int[3]; // Egy 3 elem méretű, kezdetben üres tömb
Console.WriteLine(szamok[0]); // Az indexelés 0-tól kezdődik, tehát 2-ig tart
szamok[0] = 1;
Console.WriteLine(szamok[0]);
szamok[1] = 2;
szamok[2] = 3;

try
{
    szamok[3] = 4; // A 3-as index "túlcsordul", mert 3 elemű a tömb
}
catch (IndexOutOfRangeException)
{
    Console.WriteLine("Az index túlcsordult.");
}
```



- A tömb kezdetben az int alapértékével, 0-val kerül feltöltésre
  - Referenciák esetén ez null értékeket jelent

# Listák

- A listák tömbökre épülő egyszerű adatszerkezetek
- Kezelésük:
  - Hasonlóképp indexelhető, mint a tömb (ugyanúgy túlcsordulhat)
  - Elem beszúrható a lista végére vagy bármely két eleme közé
  - Több elem is beszúrható egyszerre
  - Elem(ek) törölhető(k) adott pozícióból (/intervallumból)



# Bejárás

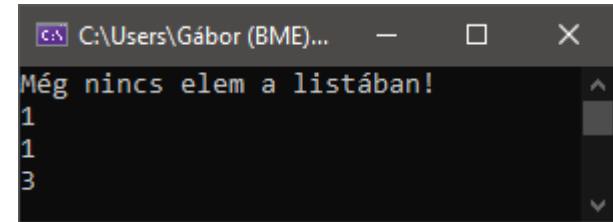
- Bármilyen `IEnumerable<T>` interfészt megvalósító kollekciótípus bejárható a `foreach` ciklus segítségével
  - A tömb és lista is megvalósítják ezt az interfészt
  - Az interfész „generikus”, ezt jelzi a `<T>`
    - A `T` helyére bármilyen konkrét típus kerülhet, pl. `int` vagy `Allat`, de akár `int[]` vagy `List<Allat>` is
- A bejárás „addig megy”, amíg a kollekcióban vannak hátralevő elemek
- A tömb bejárásában tehát mindig fix számú elem lesz (mert a tömb létrehozáskor fixálódik a mérete), a listában viszont mindig csak a benne található elemek számának megfelelő bejárás lesz a ciklusban

# Tömbök, listák, bejárás: példa

```
var szamok = new List<int>();
try
{
    Console.WriteLine(szamok[0]);
}
catch (ArgumentOutOfRangeException)
{
    Console.WriteLine("Még nincs elem a listában!");
}
szamok.Add(1);
Console.WriteLine(szamok[0]);
var szamTomb = new int[2];
szamTomb[0] = 2;
szamTomb[1] = 3;

szamok.AddRange(szamTomb);
szamok.RemoveAt(1);

foreach(var elem in szamok)
{
    Console.WriteLine(elem);
}
```



# Tömbök *vagy* listák?

- Legyünk tisztában vele, hogy mire jó a tömb
- Most használjunk tömb helyett inkább *mindenütt* listát
  - Később: sehol ne használjunk listát objektum *interfészén*, helyette használjuk az `IEnumerable<T>`-t
  - Később: a lista algoritmikus szempontból jó sorrendben történő bejárásra, de „nem jó”:
    - Kulcs szerinti keresésre
    - Elem beszúrására/eltávolítására
    - Egyedi elemek karbantartására
    - Gyors rendezésre
    - Stb.