

# Objektum Orientált Programozás 1.

Répasné Babucs Hajnalka

Répás Csaba

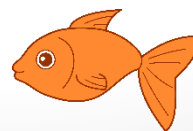




# Strukturált programozás



- Dijkstra: Structured Programming (1972)
- Programépítési alapelv
- Az elvégzendő feladatot kisebb, egymáshoz csak meghatározott módon kapcsolódó részfeladatokra kell felbontani
- Vezérlési szerkezetek
- Adatok és a műveletek nincsenek összekapcsolva





# Út az OOP-ig



- Szoftverkrízis: 1960-as évek vége
  - 3. generációs számítógépek, lényegesen hatékonyabb erőforrások
  - Növekvő fejlesztési igények
  - A szoftverprojektek jelentős része sikertelen
    - Tervezettnél drágább
    - Hosszabb idő kell a fejlesztéshez
    - Nem az igényeknek megfelelő
    - Rossz minőségű
    - Nehezen karbantartható
- Új elvek, módszerek, szoftver szabványok és technológiák kidolgozására van szükség
- 1969: Alan Curtis Kay diplomamunkájában leírja az alapelveket
- Windows 3.1-ben (1992) már megtalálhatjuk Alan Kay elképzeléseit





# OOP alapelvek



## ➤ **Egységbezárás** (**encapsulation**)

- Az adatok és a hozzájuk kapcsolódó tevékenységek (metódusok) egységbe zárása ( => osztályok)



## ➤ **Öröklődés** (**inheritance**)

- Az osztályok egy részét már meglévő osztályok továbbfejlesztésével hozzuk létre úgy, hogy további adattagokkal, illetve metódusokkal bővítjük.



## ➤ **Többalakúság** (**polymorphism**)

- Ugyanolyan elnevezésű, de más tevékenységű metódusok használata
  - Függvények túlterhelése (overloading)
  - Paraméteres polimorfizmus (generikusság)
  - Altípusosság (öröklődésnél)





# Kiegészítő elvek



## ➤ **Adatrejtés**

- Az adattagok a külvilág számára a lehető legnagyobb mértékben rejtve vannak, növelve a biztonságos programozást

## ➤ **Újrafelhasználhatóság (WORA / WORE)**

- Write once, run anywhere / Write once, run everywhere
- Jó tervezés és megfelelő dokumentáltság esetén az elkészített osztályok más programokban is változtatás nélkül, esetleg továbbfejlesztve beépíthetők
- Lehetőség van önálló csomagok, komponensek, keretrendszerek kialakítására

## ➤ **RAD (Rapid Application Development – Gyors alkalmazás fejlesztő rendszer)**







# Alapfogalmak



- Osztály vagy objektum-osztály (**class**) – típus
  - Azonos tulajdonságokkal (jellemzőkkel) és viselkedéssel rendelkező egyedeket zárja egységbe
- Objektum (**object**) – osztály típusú változó
  - Az osztály egy példánya





# Osztály tartalma



## ➤ Mezők (**field**)

- Normál és csak olvasható változók, konstansok (**constant**)

## ➤ Metódusok (**method**)

- Normál metódusok, konstruktorok (**constructor**), destruktorok (**destructor**)

## ➤ Tulajdonságok (**property**) és indexelők (**indexer**)

## ➤ Események (**event**)

## ➤ Operátorok (**operator**)

## ➤ Beágyazott típusok (**nested type**)

- Osztályok (**class**), struktúrák (**struct**), rekordok (**record**), interfészek (**interface**), delegátok (**delegate**)





# Fogalmak



- **Inicializálás:** az objektum alaphelyzetbe állítása. Minden mezőnek valamilyen kezdőértékkel kell rendelkeznie.
- **Példányosítás:** valamely objektum számára memória foglalás és inicializálás. Ennek során legalább egy konstruktor meghívása kötelező, hogy a mezők alaphelyzetbe állítása biztosan megtörténjen.
- **Felelősség:** egy objektum felelős azért, hogy az inicializálás után máris megfelelő értékekkel rendelkezzenek a mezői, és később se fordulhasson elő, hogy a mezőkbe hibás érték kerül.
- **Adatrejtés:** a mezők közvetlen hozzáférés (elsősorban írás) elleni védelme, melyet egy objektum azért alkalmaz, hogy ne kerülhessen valamely mezőjébe értelmetlen, hibás érték.







# Object ősz osztály



- C#-ban minden objektum őse az Object osztály
  - Nem kell jelezni, ha az Object a közvetlen ősz osztály
- Tartalmaz általánosított metódusokat, amelyek minden típuson működnek





# System.Object osztály fontosabb metódusai



- **public Type GetType()** Visszaadja a példány típusát reprezentáló objektumot
- **public virtual bool Equals(object obj)** Egyenlőség vizsgálat, saját osztály esetén célszerű felülírni
- **public virtual int GetHashCode()** Visszaad egy hash értéket, saját osztály esetén célszerű felülírni
- **public virtual string ToString()** Tetszőleges szöveget ad vissza, a gyakorlatban gyakran jól használható
- **public static bool ReferenceEquals(object objA, object objB)** Statikus metódus a referencia szerinti egyenlőségvizsgálathoz
- **public static bool Equals(object objA, object objB)** Statikus metódus a tartalom szerinti egyenlőségvizsgálathoz





# Láthatósági szintek



Szint	Hozzáférési lehetőség
public	Korlátlan
protected	Az adott osztályban és a leszármazottaiban
internal	Az adott projektben
protected internal	Az adott projektben és az osztály leszármazottaiban
private	Csak az adott osztályban
private protected	C# 7.2 óta elérhető A projekten belül protected-ként viselkednek, azon kívül private-ként



# Részleges (partial) osztályok



- A.NET 2.0-ban jelentek meg.
- Olyan osztály, amely több forrásfájlban helyezkedik el.
  - A Windows Forms tervezőhöz találták ki, mivel a Visual Studio a felület megjelenését generálja egy designer.cs fájlba. Így a felhasználó által írt kód és a generált kód elválasztódik egymástól, logikailag azonban mégis összetartoznak.
- A partial módosító metódusokra is alkalmazható.





# Mezők



- Adatelemek, amelyek részei egy osztálynak
- Kezdőérték adható nekik (inicializálás)
  - string jegy = "jeles";
  - int x = -6;
- Alapértelmezett láthatósági szint: private
- Típusai:
  - Olvasható / írható mezők
  - Csak olvasható mezők
  - Konstans mezők







# Mező típusok



## ➤ Olvasható / írható mezők

- Értékük tetszés szerint olvasható és módosítható

## ➤ Csak olvasható mezők

- Kulcsszó: **readonly**
- Értékük kizárólag inicializálással vagy konstruktorból állítható be
- `readonly string SosemVáltozomMeg = "Forever young";`



## ➤ Konstans mezők

- Kulcsszó: **const**
- Értéküket a fordítóprogram előre letárolja, futási időben sosem módosíthatók
- `const double  $\pi$  = 3.14159265;`





# Konstansok



- A **const** kulcsszóval konstansokat definiálunk, amit a fordító értékel ki
- Az értékét a deklaráció során meg kell adni, csak ott lehet definiálni
- A konstans az osztály része, a konstans értékének kiolvasásához nem kell példány
  - Minden konstans static
- Példa: `const double pi = 3.14259265;`





# Readonly



➤ A **readonly** mezők a program futása során kapnak értéket

➤ Csak egyszer írható mező

➤ Az osztály konstruktorában inicializáljuk

➤ Értéke megadható a konstanshoz hasonlóan kezdőérték adással is, de ez tervezési hiba

➤ A konstruktoron kívül az osztály egyik metódusa sem állíthatja az értékét

➤ Példányszintű, a kiolvasásához példány kell

➤ Példa: `readonly float verzio = 1.23;`





# Tulajdonságok



- A privát adattagokat kívülről nem tudjuk kiolvasni és módosítani
  - Ez az objektumorientáltság egyik alapelve: a megvalósítást elrejtjük a felhasználó elől



- Az adattagok módosítását ellenőrzötten szeretnénk megoldani (például csak bizonyos értékhatárok közötti értéket engedünk meg)
- Az adatnak korlátozzuk a hozzáférését, csak az olvasást vagy csak az írást engedélyezzük kívülről
- Az adat értéke kiszámolható más, már tárolt adatokból
- A megoldás: tulajdonságok használata





# Tulajdonságok deklarációja



- Beilleszthető a **propfull** snippet használatával

```
private int myVar;  
public int MyProperty  
{  
    get { return myVar; }  
    set { myVar = value; }  
}
```







# Tulajdonság ellenőrzött értékbeállítással



```
int kedvezmeny;
```

```
public int Kedvezmeny
```

```
{
```

```
    get { return kedvezmeny; }
```

```
    set { kedvezmeny = value > 100 ? 100 : value; }
```

```
}
```





# Tulajdonságok használata



- A tulajdonság típusa általában megegyezik az adattag típusával, de ez nem kötelező
- Általában publikus láthatóság
- Speciális típusok:
  - Csak olvasható tulajdonság: nincsen set rész
  - Csak írható tulajdonság: nincsen get rész
  - Automatikus tulajdonság: nem kötjük adattaghoz, a fordító foglal neki memóriahelyet
  - Számított tulajdonság: az osztály más adattagjaiból, tulajdonságaiból kiszámított érték





# Automatikus tulajdonság



- Létrehozása: **prop** snippet használatával



- `public int MyProperty { get; set; }`

- Nem kell létrehoznunk sem az adattagot, sem a teljes tulajdonságot, a fordító mindkettőt legenerálja helyettünk



- Az adattag a fordítás pillanatában még nem létezik, nem lehet rá hivatkozni, csak a tulajdonságon keresztül használhatjuk

- Ha a tulajdonság értékét csak osztályon belül engedjük módosítani: **propg** snippet



- `public int MyProperty { get; private set; }`



- Ha a tulajdonság értékét csak konstruktor állíthatja, használjuk a `private set` helyett az `init` –et (**prop\_i** snippet)! (Ez a legszűkebb beállíthatóság.)



- `public int MyProperty { get; init; }`



=> használata a tulajdonságoknál



C# 5.0

```
1 | public string FullName
2 | {
3 |     get
4 |     {
5 |         return FirstName + " " + LastName;
6 |     }
7 | }
```

C# 6

```
1 | public string FullName => FirstName + " " + LastName;
```

C# 7.0

```
1 | private int _x;
2 | public X
3 | {
4 |     get => _x;
5 |     set => _x = value;
6 | }
```





# Mező vagy tulajdonság?



- Ha egy adatot csak az osztályon belül használok, akkor mezőt hozok létre private láthatósággal
- Ha az adatot a példányokból is el kell érnem, de nem kaphat tetszőleges értéket, akkor private adatmező, hozzá kapcsolva egy publikus tulajdonság
- Ha az adatot a példányokból is el kell érnem, de tetszőleges értéket kaphat, akkor elég egy publikus, automatikus tulajdonságot létrehozni.
  - Ha az értékét nem engedem osztályon kívül módosítani, akkor automatikus tulajdonság private set résszel
  - Ha csak a konstruktor állíthatja be az értékét, akkor automatikus tulajdonság init résszel







# Mikor nem kell védeni a mezőt?



- Nem szükséges védelmet alkalmazni egy mezőre, ha a mező értéke bármikor megváltozhat, de a típusa pontosan leírja a felvehető értéket
- Pl.: Logikai vagy enum típusú mező
  - A fordító amúgy is ellenőrzi, hogy csak megfelelő értéket adhatunk meg
  - Felesleges property-t készíteni hozzá, mert semmilyen plusz védelmet nem biztosít, és a property-k használata lassabb, mint ha közvetlenül olvassuk vagy írjuk a mező értékét





# Statikus mezők



- Olyan mezők, amelyek az egész osztályhoz és nem az egyes példányokhoz köthetők
- Elég egyetlen példányban tárolni őket, mivel minden objektumpéldány esetén ugyanolyan értéket hordozna
- Létrehozásakor **static** kulcsszóval kell megjelölni
- A static módosítóval deklarált mezők a memóriában példánytól függetlenül, egyszer szerepelnek.
  - Már akkor is benne van a bennük tárolt érték a memóriában, amikor az osztályból még egyetlen példányt sem készítettünk.





# Statikus adatok használata



➤ Statikus taghoz az osztály nevén keresztül férünk hozzá

➤ Tulajdonság is lehet statikus

➤ Példa: Egy konkrét mobil telefon típus



➤ A mérete és tömege mindegyiknek ugyanakkora, tehát lehet static, ugyanakkor konstansként is felvehető (ami automatikusan static)

➤ Többféle színű bevonata lehet, ezért ez példányhoz tartozó adat

➤ A telefon gyártási száma a telefonhoz tartozik, telefononként egyedi, és nem megváltoztatható, ezért ez readonly

➤ Tipikus példa static adattagra:

➤ Példányszám tárolás





# Metódusok



- Utasítások logikailag összefüggő csoportja, melynek önálló neve és visszatérési értéke van
- Segítségével megváltoztathatjuk egy objektum állapotát, vagy kiolvashatjuk tulajdonságainak értékét, leírhatjuk viselkedését
- C#-ban metódusok csak osztályon belül definiálhatók (**tagfüggvény**)
- Ha nem adjuk meg, akkor az alapértelmezett hozzáférési szint a **private**.





# Metódusok felépítése



[módosító] visszatérési\_érték tagfüggvénynév([paraméterlista])

{ //Metódus törzse

lokális deklarációk;

utasítások;

[ return [kifejezés]; ]

}



```
class Program
{
    static void Main(string[] args)
    {
    }
}
```







# Paraméterek



- A metódusok rendelkezhetnek megadott vagy változó darabszámú paraméterrel (**params** kulcsszó)
  - void ParaméternélküliMetódus();
  - void EgyparaméteresMetódus(bool feltétel);
  - void TöbbparaméteresMetódus(int a, float b, string c);
  - void MindenbőlSokatElfogadóMetódus(params object[] paraméterTömb);





# Paraméterlista



➤ A **paraméterlista** arra szolgál, hogy a hívó programmodul értékeket adjon át a meghívott metódusnak, amellyel az el tudja végezni a feladatát.



➤ A metódus deklarációjában szereplő paraméterlistát szokás **formális paraméterek**nek nevezni



➤ A **metódus hívás**kor megadott argumentumok alkotják az **aktuális paraméterek**et









➤ Lehetnek változó, konstans értékek vagy kifejezések is



➤ Az aktuális paraméterlistának típusban, sorrendben és számban meg kell egyeznie a formális paraméterlistával



# Paraméter típusok

- 
- 
- 
- A paraméterek lehetnek érték, referencia, vagy out típusúak
    - Érték típusú: tipikusan bemenő paraméter. Nincs külön jelölése.
    - A referencia paraméter mind bemenő, mind visszatérő értéket tárolhat – a paraméter megváltozhat. A **ref** kulcsszó jelzi ezt a típust. A ref kulcsszónak a hívásnál is szerepelnie kell!
    - Az **out** kulcsszó jelzi az output paramétert. Az out kulcsszónak a hívásnál is szerepelnie kell!
- 
- 
- 



# Érték szerinti paraméterátadás



- A metódusban létrejövő lokális paraméterváltozóba átmásolódik a híváskor megadott adat vagy hivatkozás
- Az aktuális paraméterlista értékei inicializálják a formális paraméterlistában szereplő változókat
- A hívó oldalon szereplő aktuális paraméterek és a metódus formális paraméterlistáján szereplő változók más-más memóriaterületet foglalnak el
  - Ennek köszönhető, hogy a metódusok az érték szerinti paraméterátadásnál a hívó oldal változóinak értékét nem változtatják meg





# Cím szerinti (referencia) paraméterátadás



- A metódus hívásakor a megadott objektumra mutató hivatkozás adódik át a formális paraméternek
- A paraméteren keresztül magát az eredeti objektumot érjük el, legyen akár érték vagy referencia típusú
- A metódus formális paraméterlistáján és a híváskor az aktuális paraméterlistán is a **ref** kulcsszóval jelezni kell, hogy hivatkozást adunk át a metódusnak







# Kimenő (output) paraméterek



- Arra használjuk őket, hogy a metódusból adjunk vissza értékeket a hívó programkódba.
- Az aktuális paraméterlistán kifejezés vagy konstans nem feleltethető meg az out módosítóval rendelkező formális paraméternek, csak változó
- Mindenképpen értéket kell kapnia a metódusból való kilépés előtt
- Az **out** kulcsszóval jelzett formális és aktuális paraméter ugyanarra a memóiahelyre mutat, hasonlóan a referenciatípushoz
- A metódus hívása előtt az aktuális paramétert nem kell inicializálni





# Paramétertömbök



- A params módosító lehetővé teszi, hogy nulla vagy több aktuális paraméter tartozzon a formális paraméterhez
- Egy paraméterlistán csak egy paramétertömb lehet, és ennek kell az utolsónak lennie
- A paramétertömb csak azonos típusú elemeket tartalmazhat



```
private static int Osszegez(params int[] szamok)
{
    int osszeg = 0;
    foreach (var szam in szamok)
    {
        osszeg += szam;
    }
    return osszeg;
}
```

```
int t1 = Osszegez(1, 2); //3
int t2 = Osszegez(5, 10, 15, 30); //60
```



# Visszatérési érték



- A metódusok rendelkezhetnek visszatérési értékkel
  - Nem kötelező, ha nincs, ezt a **void** kulcsszóval kell jelölni
  - `void NincsVisszatérésiÉrtékem();`
  - `int EgészSzámotAdokVissza(float paraméter);`
- A visszatérési érték tetszőleges saját típus is lehet
- A paraméterek és a visszatérési érték határozzák meg azt a protokollt, amelyet a metódus használatához be kell tartani
  - Ez a metódus **szignatúrája**





# Metódus törzs



- Végrehajtandó utasítások, melyek használhatják a bemenő paramétereket
- A **függvény** visszatérő értéke a **return** kulcsszót követi
  - Ebből több is lehet a program különböző ágain
- Visszatérési érték nélküli (void) metódusnál a return utasítás nem kötelező
  - Ezeket a metódusokat hívjuk **eljárás**nak





# Alapértelmezett értékű, opcionális paraméterek

- Ha a metódusnak van olyan paramétere, amit az esetek többségében ugyanazzal az értékkel hívunk meg, a paraméternek adhatunk alapértelmezett értéket.
- Opcionális paraméterek csak a kötelező paraméterek után szerepelhetnek.

```
private static int Osszegez(int meddig=10)
{
    int osszeg = 0;
    for (int i=1; i<meddig; i++)
    {
        osszeg += i;
    }
    return osszeg;
}
```

```
int t1 = Osszegez(); //első 10 szám összege
int t2 = Osszegez(100); //első 100 szám összege
```







# Nevesített paraméterek



- Metódus hívásakor a paraméterekre a nevükkel is hivatkozhatunk.



```
void OptionalisParameteres(int kotelezo, string optionalisSzoveg = "tigris",  
                           int optionalisSzam = 10)
```



- Helyes meghívások:

```
OptionalisParameteres(6);
```

```
OptionalisParameteres(6, "kistigris");
```

```
OptionalisParameteres(6, optionalisSzam: 12);
```

```
OptionalisParameteres(optionalisSzam: 12,  
                       kotelezo: 6 );
```



- Hibás meghívások:

```
OptionalisParameteres();
```

```
OptionalisParameteres(6, ,12);
```



# Konstruktor



➤ Speciális metódus



➤ Minden osztálynak rendelkeznie kell konstruktorral

➤ A konstruktor gondoskodik az osztály példányainak létrehozásáról

➤ A konstruktorok neve mindig megegyezik az osztály nevével

➤ Több konstruktort is létrehozhatunk más-más paraméterlistával

➤ Egy konstruktor a **this** kulcsszó segítségével meghívhat egy másik konstruktort is

➤ Ha mi magunk nem deklarálunk konstruktort, a C# fordító automatikusan létrehoz egy paraméter nélküli alapértelmezett konstruktort

➤ Visszatérési értéke nincs, nem is lehet, ezért a deklarációnál sem szabad megadni (fordítási hiba).





# this használata



```
class pelda
{
    private int a;
    private int b;

    pelda(int a, int b)
    {
        this.a = a;
        this.b = b;
    }
}
```

```
class KonstruktorPelda
{
    private int _szam;

    //paraméter nélküli konstruktor
    public KonstruktorPelda() : this(42)
    {
        //Ha ide kódot tennénk, akkor az a Paraméteres
        //konstruktor futása után futna le
    }

    //paraméteres konstruktor
    public KonstruktorPelda(int szam)
    {
        _szam = szam;
    }

    public void Kiir()
    {
        Console.WriteLine(_szam);
    }
}
```



# Statikus konstruktor



- A statikus osztályok és a nem statikus osztályok is rendelkezhetnek egy statikus konstruktorral
- Az osztály bármely statikus tagjának első használata előtt fog közvetlenül lefutni
- Csak 1 lehet belőle, paramétere nem lehet
- Nem lehet ellátni elérés módosítóval és közvetlenül nem lehet meghívni
- Ha az osztály rendelkezik statikus és nem statikus konstruktorral, akkor a statikus konstruktorban elhelyezett kód a nem statikus konstruktor kódja előtt fog futni, az osztály első példányosításakor egy alkalommal





# Destruktor



- Az osztályoknak nem kötelező destruktorral rendelkezniük
- A destruktor neve egy „~” karakterből és az osztály nevéből áll
- Az objektumok megszüntetése automatikus
  - Akkor szűnik meg egy objektum, amikor már biztosan nincs rá szükség, a futtatókörnyezet gondoskodik a megfelelő destruktor hívásáról
- Nem kell (és nem is lehet) közvetlenül meghívni az osztályok destruktoraikat
  - A destruktor nem tudhatja, pontosan mikor hívódik meg
- Destruktorra ritkán van szükség

