

# Szoftvertervezés és -fejlesztés I.

## Objektum-orientált programozás

*kertesz.gabor@nik.uni-obuda.hu*

# Tartalom

- **Bevezetés**
- **Objektum-orientáltság alapfogalmai**
- **Objektum-orientált programozás C# környezetben**
- **Névterek**
- **Objektum-orientáltság alapelvei**

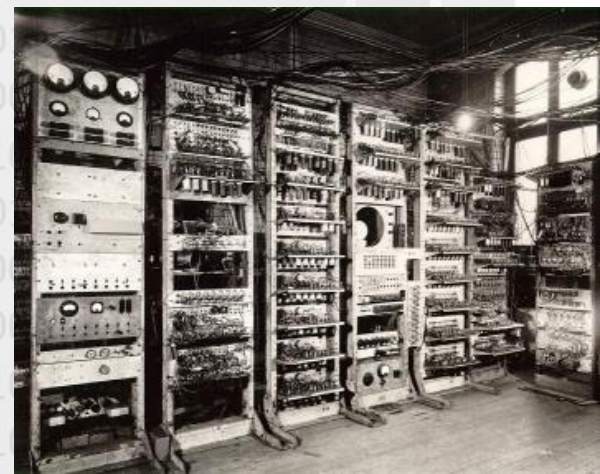
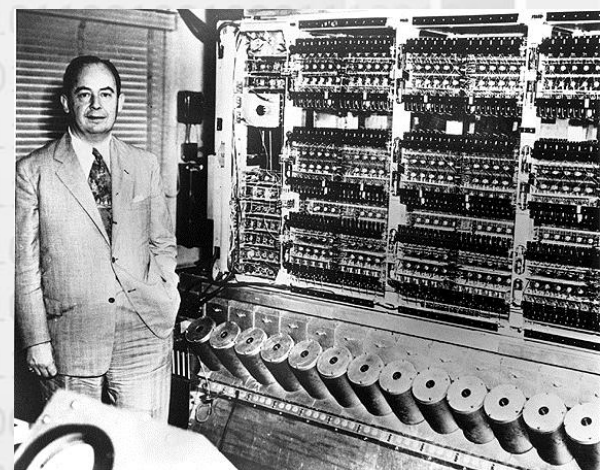
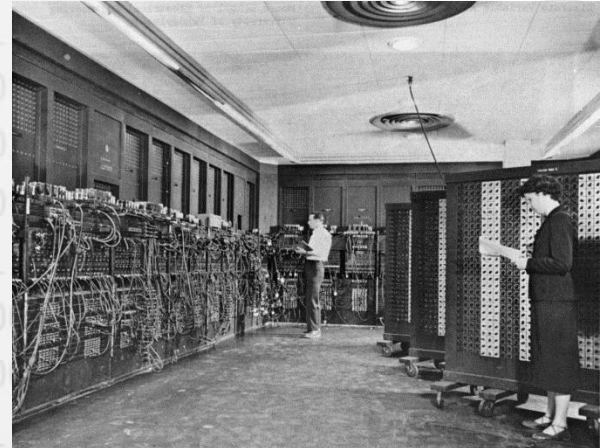
# BEVEZETÉS

# Programozástörténelem

- **Első számítógépek: 1940-es évek**
  - ENIAC (1946): kapcsolótáblás programozás
  - EDVAC (1948): Neumann-elvű gép
  - MARK 1 (1952): az első programnyelv, fordított gépi kód

## Neumann-elvek

1. Teljesen elektronikus működés
2. Kettes számrendszer használata
3. Belső memória használata
- 4. Tárolt program elve. A számításokhoz szükséges adatokat és programutasításokat a gép azonos módon, egyaránt a belső memóriában (operatív tár) tárolja.**
5. Soros utasításvégrehajtás
6. Univerzális felhasználhatóság
7. Szerkezet: öt funkcionális egység (aritmetikai egység, központi vezérlőegység, memóriák, bemeneti és kimeneti egységek)



# Programozástörténelem

- **Általános célú gépek megjelenése (1950-es évek)**
  - Eddig jellemzően matematikusok programoztak
  - A fejlesztés jellemzően „gépi nyelven” történt
  - Minden megírt program szorosan kötődött a számítógéphez

## Gépi nyelv: Assembly

Alacsonyszintű programozási nyelv, specifikus egy adott architektúrához: rövid utasításokból áll, amelyek kb. a processzor utasításkészletét képezik le.

Binárisan: 10110000 01100001

Hexadecimálisan: B0 61

Assembly nyelven: **MOV AL, 61h**

AL regiszterbe 61h (97 decimális) érték betöltése



# Programozástörténelem

- **Szoftverkrízis (1960-as évek vége)**

- Nagyobb erőforrású hardverek
- Bonyolultabb problémák megoldására is alkalmasak
- Változó felhasználói kör
- Megjelennek a szoftver minősége iránti igények
- A minőséget egyre kevésbé lehetett tartani a hagyományos módszerekkel

## Szoftverminőség

A felhasználó a minőséget külső jegyekben méri le:

- Helyesség
- Megbízhatóság
- Felhasználóbarátság
- Alkalmazkodóképesség

A minőség csak belső rend kialakításával érhető el, amely állandó törekvés:

- Átláthatóság
- Általánosság/"szépség"
- Módosíthatóság
- Újrafelhasználhatóság
- Egyéniségfüggetlenség

# Programozási paradigmák

## Programozási paradigma

"(...) a valóságos tudományos gyakorlat egyes elfogadott mintái – ezek a minták magukban foglalják a megfelelő törvényt, elméletet, az alkalmazást és a kutatási eszközöket együtt – olyan modellek, amelyekből a tudományos kutatás sajátos összefüggő hagyományai fakadnak."

T. S. Kuhn: A tudományos forradalmak szerkezete, 1970

"A paradigma (vagy korszellem) a gondolkodásoknak, vélekedéseknek, értékeknek és módszereknek egy adott társadalom vagy szűkebben egy tudományos közösség minden tagja által elfogadott összességét jelenti"

Varga Csaba: Új elmélethorizontok előtt, Kuhn alapján, 2003

- **Strukturált programozás**
- **Procedurális programozás**
- **Objektum-orientált programozás**
- ...

Részletesen később, Szofvertechnológia előadáson.

# Strukturális programozás

- Előtte strukturálatlan: tesztelés és ugrás
- 1966: Böhm-Jacopini strukturált programozási tétel

Böhm, C.; Jacopini, G. (May 1966). "Flow diagrams, Turing machines and languages with only two formation rules". *Communications of the ACM* 9 (5): 366–371

- Bármely algoritmus leírható három vezérlési szerkezet (szekvencia, szelekció, iteráció) segítségével

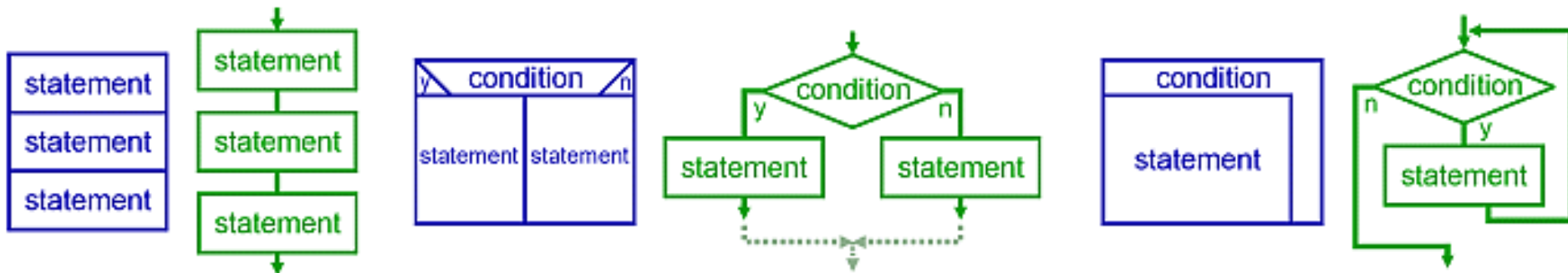
- 1968: Go To utasítás károsnak minősítése

Dijkstra, E. W. (March 1968). "Letters to the editor: go to statement considered harmful". *Communications of the ACM* 11 (3): 147–148

- Nyelvek: ALGOL, Pascal, Ada

```
10 Input A  
15 Input B  
20 B = A + 10  
30 IF B > 12 GOTO 60  
40 C = B / 3  
50 IF C < 24 GOTO 10  
60 Write C  
70 IF Write Failed GOTO 15  
80 Input D
```

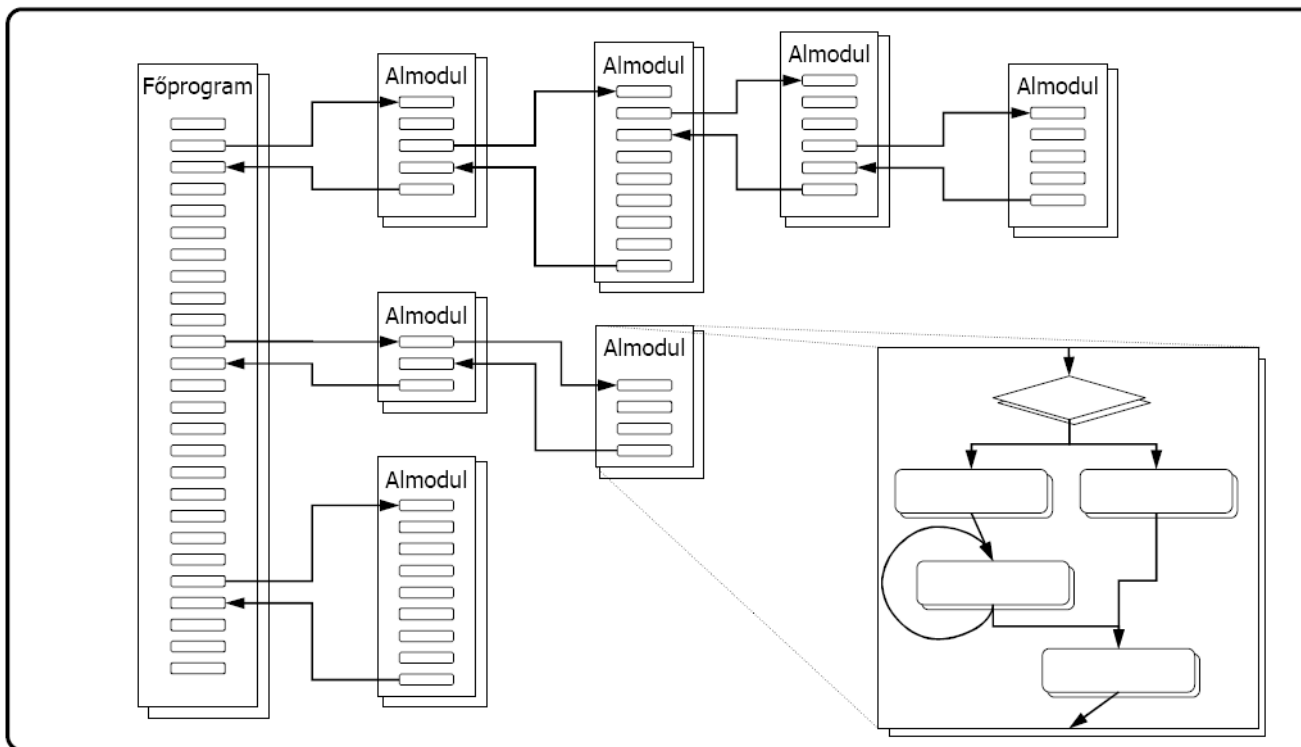
[http://www.yitsplace.com/Programming/images/SW\\_str2.gif](http://www.yitsplace.com/Programming/images/SW_str2.gif)





# Procedurális programozás

- Kiegészítve a strukturális elveket
- A program procedúrákból (rutin, szubrutin, függvény, eljárás) áll
- A „főprogram” procedúrákat hív meg
  - Ez gyakran igen bonyolult
- Globális adatstruktúrák
- Modularitás, kódújrafelhasználhatóság
- Nyelvek: Pascal, C, Fortran, BASIC



# Objektum-orientált programozás

- **A problémát nem az algoritmus, vezérlés oldaláról, hanem az adatok oldaláról közelíti meg**
- **Absztrakt rendszerelemek meghatározása**
  - Az adatok és a rajtuk végezhető műveletek kiválasztása, és ezek összerendelése
  - Ezzel az elemek csoportokba sorolhatóak
  - A cél hogy minden adat és a hozzá tartozó művelet egy helyen jelenjen meg
- **Modularitás, áttekinthetőség, karbantarthatóság, újrafelhasználhatóság**

## Absztrakció

Az absztrakció szót hétköznapi használatában szokásosan elvonatkoztatásnak fordítják, és a lényeges és lényegtelen tulajdonságok elválasztását, a lényeges tulajdonságok kiemelését és a lényegtelen tulajdonságok figyelmen kívül hagyását értik rajta.

Bakos Ferenc: Idegen szavak szótára. Akadémiai Kiadó, Bp., 1983

# Objektum-orientált programozás

- Adatok és műveletek összefogva osztályokat definiálnak
- A konkrét szereplők az objektumok:

Minden objektum egy osztályhoz tartozik (az osztály egy példánya), az osztály által definiált adatokat tartalmazza és azokkal az osztályban megadott műveleteket képes elvégezni.

- **Jellemzők:**

- Az egyes objektumok magukban foglalják az algoritmusokat
  - Minden objektum a probléma egy részét írja le és magában foglalja a részfeladat megoldásához tartozó algoritmikus elemeket
- A főprogram jelentősége igen csekély
  - Gyakorlatilag csak indítási pontként szolgál, lényegi funkciót általában nem lát el

- **Nyelvek: C++, Java, C#, ...**

**Ebben az előadásban C# nyelvi minták találhatók, de az objektum-orientált paradigma általános, más alkalmas nyelvben is alkalmazható szemlélete kerül elsősorban bemutatásra.**

# Példa

- Szerepjáték

- Játékos

- Adatok: név, ütőerő, pajzserő, élet, tárgyak, életben van
    - Műveletek: támadás, menekülés, tárgyhasználat

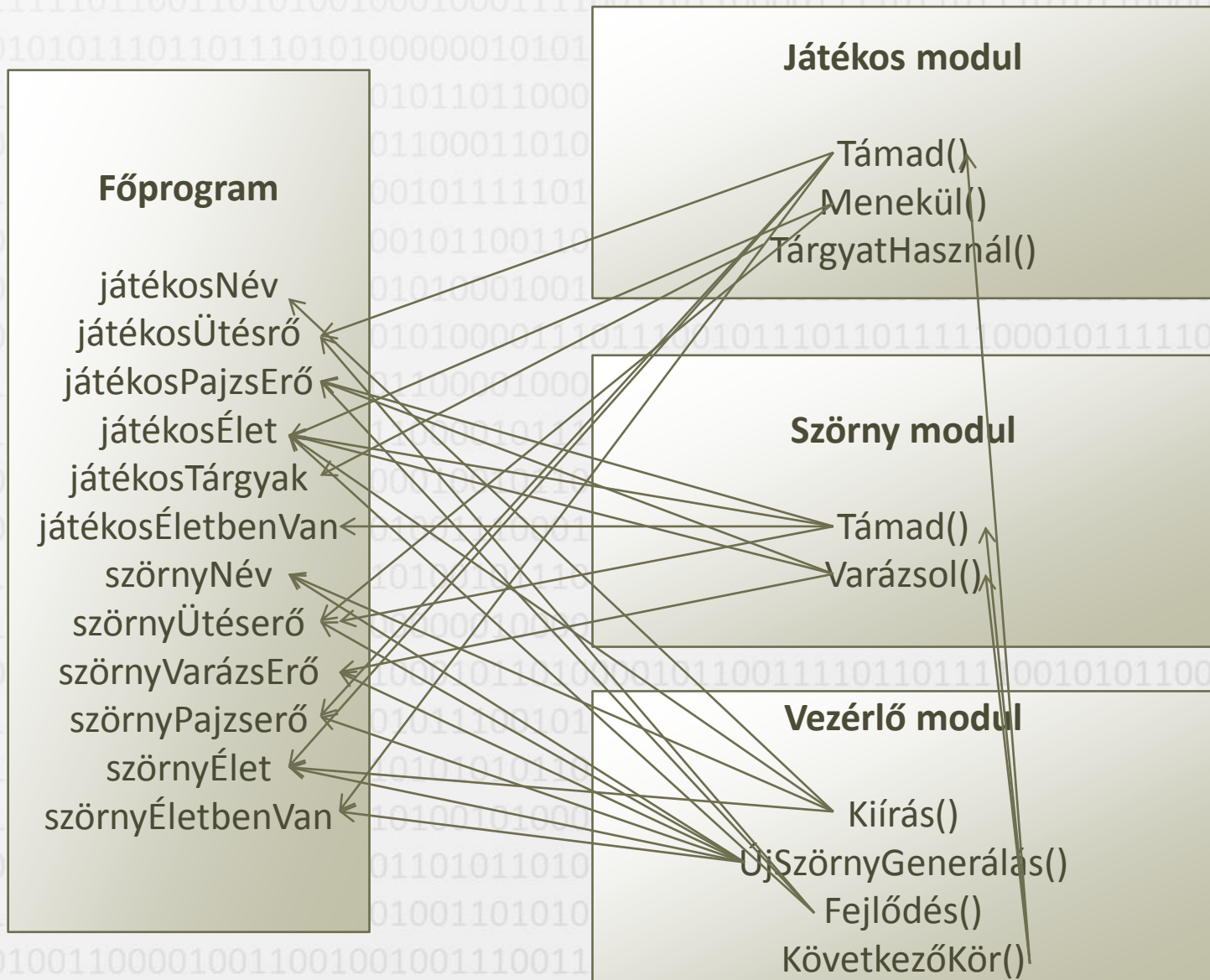
- Szörny

- Adatok: név, ütőerő, varázserő, pajzserő, élet, életben van
    - Műveletek: támadás, varázsol





# Példa - procedurális



# Példa - procedurálisan

- **Hasonló logikájú elemek és műveletek nehezen újrafelhasználhatók**
  - Játékos és Szörny támadása hasonló elven zajlik
- **Nehéz karbantarthatóság**
  - Módosítással a kész logika sérülhet
  - Hibalehetőség a sok globális változó
- **Hibalehetőségek fejlesztés során**
  - Sok függvény, sok változó
- **Spagetti-kód**
  - Nehezen átlátható vezérlés

# Példa - OO

## Karakter osztály

*Név*  
*ÜtésErő*  
*PajzsErő*  
*Élet*  
*ÉletbenVan*

*Támadás()*  
*Megsebzés()*  
*ÉletétVeszti()*

## Vezérlő osztály

*KövetkezőKör()*  
*SzörnyGenerálás()*

## Játékos osztály

*Tárgyak*

*TárgyHasználat()*  
*Fejlődés()*

## Szörny osztály

*Varázserő*  
*Varázsol()*

# Példa - OO

- **Jól tervezett osztály, kódújrafelhasználás örökítéssel**

- A Játékos és a Szörny objektumok gyakorlatilag Karakterek, amelyek közös változókkal és eljárásokkal rendelkeznek

(Bővebben erről majd Szoftvertervezés és -fejlesztés II-n!)

- **Egységbezárás**

- Az adatok és a rajtuk végzett műveletek egy helyen vannak definiálva: az osztályban

- **Adatrejtés**

- Az osztályok némely adattagja elrejthető kívülről, így segíti a hibák elkerülését

- **Átláthatóság, olvashatóság, karbantarthatóság**

- „valós világbeli” elemek absztrakciói az objektumok



# Példa- OO



- A játékos és a szörny metódusai nem érik el egymás adatait

- Az objektumok üzenetváltással kommunikálnak:

*jonSnow.Tamadas(szorny)*

Ez hatással lehet *jonSnow* objektum tulajdonságaira is, ugyanúgy, ahogy a *szorny* tulajdonságaira is.

# **OBJEKTUM-ORIENTÁLTSAĞ ALAPFOGALMAI**

# OO alapfogalmak – osztály vs objektum

- Az osztály definiálja azokat a „tulajdonságokat és képességeket”, amelyeket az adott osztályba tartozó összes objektum birtokol
  - Az osztályleírás így egyfajta sablont ad az objektumokhoz
  - Tartalmazza, hogy az objektumok hogyan jönnek létre és hogyan semmisülnek meg, milyen adatokat tartalmaz egy objektum, és ez az adat milyen módon manipulálható
- Az objektum egy, az osztályleírás alapján létrejött diszkrét entitás
  - Minden objektum valamilyen létező osztályba tartozik (=adott osztály példánya), egy osztálynak több objektuma is lehet
  - Az osztály által meghatározott adatokat tartalmazza

# OO alapfogalmak - objektum

- **Az objektum a benne tárolt adatok felhasználásával feladatokat hajt végre és egyéb objektumokkal kommunikál**
  - Adatokat tartalmaz és pontosan meghatározott algoritmusok kapcsolódnak hozzá
  - Saját feladatait önállóan végzi
    - Saját „életciklussal” rendelkezik
    - A „külvilággal” meghatározott üzeneteken keresztül tartja a kapcsolatot
- **Az objektum saját adatait mezőknek, a beépített, hozzátartozó algoritmusait metódusoknak nevezzük**
  - Az objektumok e metódusokkal vesznek részt az üzenetváltásokban
  - Az üzenetek elemei: célobjektum, metódus, paraméterek, eredmény



# Példa: hallgatók modellezése

- Szeretnénk egy olyan alkalmazást készíteni, amelyben egyetemi hallgatókat tudunk kezelni
- A hallgatók alapadatain túl ismert az is, hogy milyen a tanulmányi státuszuk (aktív vagy passzív)

# Példa: hallgatók adatai

- **Az összetartozó adatok kerülnek közös osztályba**
  - A hallgatók neve és neptun azonosítója az alapadatok része, és bizonyos lekérdezésekhez szeretnénk a hallgató születési évét is tárolni
  - Fontos észrevenni, hogy a hallgatót jellemző tulajdonság a szeme színe, vagy a vizsgáinak a száma, netán a lakhelye is, de esetünkben ezek nem lényeges információk, így nem tároljuk
- **A tanulmányi státusz tárolására és módosítására is szükség van**

# Példa: Hallgató osztály

- Az életkor számolható a születési évből, metódusként definiáljuk
- A StátuszVáltás is metódus legyen, amely a tanulmányi státuszt változtatja

## Hallgató

Név

NeptunAzonosító

SzületésiÉv

TanulmányiStátusz

Életkor

StátuszVáltás

# Példa: Hallgató osztály tagjai

- A név és neptunkód szöveges típussal reprezentálható
- A születési év egy egész érték
- A tanulmányi státusz kétféle lehet, így célszerű egy logikai típust alkalmazni: igaz ha aktív, hamis ha passzív
- Az életkor egész értéket ad vissza
- A státuszváltás egy eljárás, üres visszatérésű, paraméterként kapja az új értékét

## Hallgató

string Név

string NeptunAzonosító

int SzületésiÉv

bool AktívStátusz

int Életkor()

void StátuszVáltás(ertek: bool)



# Példa: Hallgató osztály és objektumok

## Hallgató

string Név  
string NeptunAzonosító  
int SzületésiÉv  
bool AktívStátusz

int Életkor()  
void StátuszVáltás(allapot: bool)

### Hallgató1 : Hallgató

Név = "Alapos Aladár"  
NeptunAzonosító = "A1A1A1"  
SzületésiÉv = "1997"  
AktívStátusz = "igen"

### Hallgató2 : Hallgató

Név = "Bukott Bendegúz"  
NeptunAzonosító = "BATMAN"  
SzületésiÉv = "1991"  
AktívStátusz = "nem"

### Hallgató3 : Hallgató

Név = "Csupaötös Csaba"  
NeptunAzonosító = "C5UP45"  
SzületésiÉv = "1995"  
AktívStátusz = "igen"

# Példa: Hallgató osztály és objektumok

Osztály

## Hallgató

string Név  
string NeptunAzonosító  
int SzületésiÉv  
bool AktívStátusz

int Életkor()  
void StátuszVáltás(allapot: bool)

### Hallgató1 : Hallgató

Név = "Alapos Aladár"  
NeptunAzonosító = "A1A1A1"  
SzületésiÉv = "1997"  
AktívStátusz = "igen"

Objektumok

### Hallgató2 : Hallgató

Név = "Bukott Bendegúz"  
NeptunAzonosító = "BATMAN"  
SzületésiÉv = "1991"  
AktívStátusz = "nem"

### Hallgató3 : Hallgató

Név = "Csupaötös Csaba"  
NeptunAzonosító = "C5UP45"  
SzületésiÉv = "1995"  
AktívStátusz = "igen"

Hallgató1.Életkor()  
Hallgató2.StátuszVáltás("igaz")

# Példa: Hallgató osztály és objektumok

## Hallgató

string Név  
string NeptunAzonosító  
int SzületésiÉv  
bool AktívStátusz

int Életkor()  
void StátuszVáltás(allapot: bool)

### Hallgató1 : Hallgató

Név = "Alapos Aladár"  
NeptunAzonosító = "A1A1A1"  
SzületésiÉv = "1997"

#### Adatmezők

### Hallgató2 : Hallgató

Név = "Bukott Bendegúz"  
NeptunAzonosító = "BATMAN"  
SzületésiÉv = "1991"  
AktívStátusz = "nem"

#### Metódusok

Név = "Csupaötös Csaba"  
NeptunAzonosító = "C5UP45"  
SzületésiÉv = "1995"  
AktívStátusz = "igen"

# OO alapfogalmak - objektum

- **Az objektum állapotát mezői aktuális értéke határozza meg**
  - Az objektum állapota az elvégzett műveletek hatására megváltozhat
  - Két objektum állapota akkor egyezik meg, ha minden megfelelő mezőértékük megegyezik
  - Az objektum mindig „megjegyzi” aktuális állapotát
- **Az objektum viselkedését az osztályában definiált metódusok határozzák meg**
- **Minden objektum egyértelműen azonosítható**
  - Az objektumok önállóak (saját életciklusuk határozza meg őket)
  - *Ha két objektum állapota megegyezik, maguk az objektumok akkor sem azonosak*

## Hallgató1 : Hallgató

Név = "Alapos Aladár"  
NeptunAzonosító = "A1A1A1"  
SzületésiÉv = "1997"  
AktívStátusz = "igen"

## Hallgató2 : Hallgató

Név = "Bukott Bendegúz"  
NeptunAzonosító = "BATMAN"  
SzületésiÉv = "1991"  
AktívStátusz = "nem"

# OO alapfogalmak - osztály

- **Az osztály egy adott objektumtípust határoz meg**

- Az osztályok egyfajta mintát, sablont adnak az objektumokhoz
- Az osztályok tehát azonos adatszerkezetű és viselkedésű objektumokat írnak le, azok absztrakciója lévén kerülnek definiálásra

- **Minden objektum valamilyen létező osztályba tartozik**

- Az egyes objektumok azon osztályok **példányai**, amelyekhez tartoznak
- Egy osztályból több példány is létrehozható

- **Az egyes példányok létrehozásuk pillanatában azonos állapotúak, ezt követően viszont önállóan működnek tovább**

## Hallgató

string Név  
string NeptunAzonosító  
int SzületésiÉv  
bool AktívStátusz

int Életkor()  
void StátuszVáltás(állapot: bool)



# Példa – osztályszerkezet

```
class Hallgato
```

```
{
```

```
    string nev, neptun;
```

```
    int szuletesiev;
```

```
    bool aktivstatusz;
```

```
public Hallgato(string nev, string neptun, int szuletesiev)
```

```
{
```

```
    //...
```

```
}
```

```
public int Eletkor() { /* */ }
```

```
public void StatuszValtas(bool allapot) { /* */ }
```

```
}
```

## Adatmezők

„Egy hallgatónak van neve”

„Egy hallgatónak van neptun azonosítója”

„Egy hallgatónak van születési éve”

„Egy hallgatónak van tanulmányi státusza”

## Metódusok

„Egy hallgató életkora kiszámolható”

„Egy hallgató tud státuszt váltani”

# OO alapfogalmak - elnevezések

- **Osztály:** definíció
- **Objektum:** példány
- **Adatmező:** objektumban tárolt adat, változó
- **Metódus:** objektum képessége, osztályleírásban definiált
- **Tag:** mezők és metódusok együttes elnevezése

# Metódusok - konstruktor

## Objektum létrehozása: konstruktor segítségével

- Ahhoz, hogy az objektumokat használhassuk, először létre kell hozni őket. Ez a példányosítás

- Alapja az osztály megadott definíciója
- A példányosítást követően érhetőek el az objektumhoz tartozó metódusok és a mezőinek értéke

- A konstruktor objektumpéldányokat hoz létre

- Feladatai:

- Új objektum létrehozása
- Az objektumhoz tartozó mezők kívánt kezdőértékének beállítása
- Egyéb szükséges kezdeti műveletek végrehajtása

- Minden osztályhoz tartoznia kell konstruktornak

- Sokszor automatikusan létrejön

# Metódusok - konstruktor

```
class Hallgato
{
    string nev, neptun;
    int szuletesiev;
    bool aktivstatusz;

    public Hallgato(string n, string kod, int ev)
    {
        nev = n;
        neptun = kod;
        szuletesiev = ev;
    }

    //...
}
```

## Konstruktor

„A hallgató `nev` mezője felveszi az `n` paraméterként kapott értéket”

„A hallgató `neptun` mezője felveszi a `kod` paraméterként kapott értéket”

„A hallgató `szuletesiev` mezője felveszi az `ev` paraméterként kapott értéket”

```
Hallgato hallgato1 = new Hallgato("Alapos Aladár", "A1A1A1", 1997);
```

# Metódusok - destruktor

## Objektum megszüntetése: destruktor

- **Az objektumokat az utolsó használat után fel kell számolni, a destruktor az ezt végző metódus**
  - Minden objektum önállóan létezik, ezért külön-külön szüntethető meg
- **Az objektumok felszámolása lehet a programozó feladata vagy történhet automatikusan is**
  - Egy objektum akkor számolható fel automatikusan, ha a későbbiekben már biztosan nincs rá szükség
  - Az automatikus felszámolás (*szemétgyűjtés: GC*) fejlettebb és jóval kevésbé hibaérzékeny megoldás
  - Automatikus felszámolás esetén nincs feltétlenül szükség a destruktor definiálására



# További metódustípusok

- **Módosító metódusok (*setter*)**
  - Megváltoztatják az objektum állapotát
- **Kiolvasó metódusok (*getter*)**
  - Hozzáférést biztosít az objektum adataihoz, de nem változtatják meg őket (így az objektum állapotát sem)
- **Iterációs metódusok**
  - Az objektum adatainak valamely részhalmazán „lépkednek végig”, és arra vonatkozóan végeznek el műveleteket
- **Indexelők**
  - Az objektum adatai valamely részhalmazának indexelt elérését biztosítják
- **Operátorok**
  - Az objektumokkal egyszerű műveleteknek (+, -, \*, /, ...) az adott osztály kontextusában értelmezett elvégzését támogatják

# További metódustípusok

- **Tulajdonság**

- Felfogható intelligens mezőként
- Írási és olvasási műveletéhez egy-egy metódus tartozik (getter, setter)
  - Ezek segítségével a tulajdonságot tartalmazó osztály kontrollálhatja, hogy mi történjen a tulajdonság írásakor, olvasásakor

- **Nyelvtől függően az előzőekben említett metódustípusokon és mezőkön kívül még mások is előfordulhatnak az osztályleírásban**

- **A konkrét működési mechanizmus is nyelvenként különbözhet**

# Tulajdonság

```
class Hallgato
{
    bool aktivstatusz;
    public bool AktivStatusz
    {
        get
        {
            return aktivstatusz;
        }
        set
        {
            aktivstatusz = value;
        }
    }
    //...
}
```

## Tulajdonság

„Olvasáskor add vissza az aktivstatusz értékét”  
„Íráskor módosítsd az aktivstatusz értékét”

```
Hallgato1.AktivStatusz = true;
```

# Láthatóság

- Tulajdonságot gyakran alkalmazunk privát mezők értékének kontrollált elérésére
- Az osztályok minden tagjához (mezőhöz, metódushoz) láthatósági szint van rendelve
- A láthatósági szint azt befolyásolja, hogy az adott tag honnan elérhető
  - Bárhonnan, „nyilvános”
  - Vagy csak az osztályon belülről lehet rá hivatkozni, „privát”
  - A legtöbb nyelv további szinteket is definiál (adott osztályból és utódosztályokból elérhető, adott programon belülről elérhető... stb.)

# Láthatóság

```
class Hallgato
{
    private string nev, neptun;
    public int szuletesiev;
    bool aktivstatusz;

    //...
}
```

## Láthatóság

A nev és neptun változók csak osztályon belül érhetőek el  
A szuletesiev változó kívülről is elérhető  
Az aktivstatusz csak belülről érhető el

```
class Program
{
    static void Main(string[] args)
    {
        Hallgato hallgato1 = new Hallgato();
        hallgato1.szuletesiev = 1997;
        hallgato1.nev = "Alaptalan Aladár";
        hallgato1.neptun += "A";
        hallgato1.aktivstatusz = !hallgato1.aktivstatusz;
    }
}
```



# Példányszintű tagok

## Objektumokhoz tartozó mezők és metódusok

- **A példányszintű tagok az objektumpéldányok...**
  - saját adatmezői, valamint
  - saját adatain műveleteket végző metódusai
- **A példányszintű mezők tárolják a példányok állapotát**
- **A metódusok kódját nem tartalmazza külön-külön az osztály minden példánya**
  - A metódusok minden példánynál azonosak és nem módosíthatók, ezért a metódusok programkódját az osztályon belül szokás tárolni

# Osztályszintű tagok

## Osztályokhoz tartozó mezők és metódusok

- **Az osztályszintű tagokkal (mezőkkel, metódusokkal) példányoktól független adattartalom és működés reprezentálható**
- **Osztályszintű adatmezők**
  - Minden osztály pontosan egyet tartalmaz belőlük, függetlenül az osztályból létrehozott objektumpéldányok számától
  - Elérhető akkor is, ha egyetlen példány sem létezik
  - Minden objektumra egyaránt érvényes adatok esetében alkalmazott, amikor nem célszerű minden példányban külön tárolni

# Osztálysztintű tagok

- **Osztálysztintű metódusok**

- Akkor is elérhető, ha az osztályból egyetlen objektum sem lett példányosítva
- Konkrét példányt nem igénylő feladatok végrehajtására alkalmasak
  - Példa: konverziós függvényeknél ne kelljen példányt létrehozni
  - Példa: operátorok
  - Példa: főprogram megvalósítása, IO kezelés
- Speciális osztálysztintű metódus az osztálysztintű konstruktor
  - Feladata az osztálysztintű adatmezők kezdőértékének beállítása

# Osztály- és példányszintű tagok

	Osztály szintű tagok	Példány (objektum) szintű tagok
Mezők (adattagok)	Egyetlen, az <u>osztályhoz</u> tartozó példány	Minden egyes <u>objektumhoz</u> saját példány
Metódusok (funkciótagek)	Osztályonként egy-egy példány ( <u>nincs</u> konkrét objektum, amelyen működnek)	Osztályonként egy-egy példány ( <u>van</u> konkrét objektum, amelyen működnek)

# Példa: metódusok

## Hallgató

string Név  
string NeptunAzonosító  
int SzületésiÉv  
bool AktívStátusz

int Életkor()

void StátuszVáltás(bool állapot)

### Hallgató1 : Hallgató

Név = "Alapos Aladár"  
NeptunAzonosító = "A1A1A1"  
SzületésiÉv = "1997"  
AktívStátusz = "igen"

### Hallgató2 : Hallgató

Név = "Bukott Bendegúz"  
NeptunAzonosító = "BATMAN"  
SzületésiÉv = "1991"  
AktívStátusz = "nem"

### Hallgató3 : Hallgató

Név = "Csupaötös Csaba"  
NeptunAzonosító = "C5UP45"  
SzületésiÉv = "1995"  
AktívStátusz = "igen"

- A metódusok kódját nem tartalmazza külön-külön az osztály minden példánya



# Példa: objektumok

## Hallgató

string Név  
string NeptunAzonosító  
int SzületésiÉv  
bool AktívStátusz

int Életkor()  
void StátuszVáltás(bool állapot)

### Hallgató1 : Hallgató

Név = "Alapos Aladár"  
NeptunAzonosító = "A1A1A1"  
SzületésiÉv = "1997"  
AktívStátusz = "igen"

### Hallgató2 : Hallgató

Név = "Bukott Bendegúz"  
NeptunAzonosító = "BATMAN"  
SzületésiÉv = "1991"  
AktívStátusz = "nem"

### Hallgató3 : Hallgató

Név = "Csupaötös Csaba"  
NeptunAzonosító = "C5UP45"  
SzületésiÉv = "1995"  
AktívStátusz = "igen"

- Az objektumok *egyértelműen azonosíthatók*
  - a példány memóriabeli helye alapján
- Az osztály típusának megfelelő változó egy **referencia (mutató)**, amely az objektumra hivatkozik

# Példa: osztályszintű mező

## Hallgató

string Név

string NeptunAzonosító

int SzületésiÉv

bool AktívStátusz

double ÖsztöndíjKüszöb = 3,0

int Életkor()

void StátuszVáltás(bool állapot)

## Hallgató1 : Hallgató

Név = "Alapos Aladár"

NeptunAzonosító = "A1A1A1"

SzületésiÉv = "1997"

AktívStátusz = "igen"

## Hallgató2 : Hallgató

Név = "Bukott Bendegúz"

NeptunAzonosító = "BATMAN"

SzületésiÉv = "1991"

AktívStátusz = "nem"

## Hallgató3 : Hallgató

Név = "Csupaötös Csaba"

NeptunAzonosító = "C5UP45"

SzületésiÉv = "1995"

AktívStátusz = "igen"

- **Osztályszintű adatmezők**

- Minden osztály pontosan egyet tartalmaz belőlük, függetlenül az osztályból létrehozott objektumpéldányok számától

# Példa: osztályszintű metódus

## Hallgató

string Név

string NeptunAzonosító

int SzületésiÉv

bool AktívStátusz

double ÖsztöndíjKüszöb = 3,0

int Életkor()

void StátuszVáltás(bool állapot)

void KüszöbBeállít(double érték)

## Hallgató1 : Hallgató

Név = "Alapos Aladár"

NeptunAzonosító = "A1A1A1"

SzületésiÉv = "1997"

AktívStátusz = "igen"

## Hallgató2 : Hallgató

Név = "Bukott Bendegúz"

NeptunAzonosító = "BATMAN"

SzületésiÉv = "1991"

AktívStátusz = "nem"

## Hallgató3 : Hallgató

Név = "Csupaötös Csaba"

NeptunAzonosító = "C5UP45"

SzületésiÉv = "1995"

AktívStátusz = "igen"

- **Osztályszintű metódusok**

- Akkor is elérhetők, ha az osztályból egyetlen objektum sem lett példányosítva
- Konkrét példányt nem igénylő feladatok végrehajtására alkalmasak



# **OBJEKTUM-ORIENTÁLT PROGRAMOZÁS C# KÖRNYEZETBEN**

# Objektumreferencia

- Sok nyelvben az osztályok referenciatípusok
- Az osztály típusának megfelelő változó egy referencia (mutató), amely az objektumra hivatkozik
- Több változó is hivatkozhat ugyanarra az objektumra

```
Hallgato h4;
```

```
...
```

```
h4 = new Hallgato(„Deokos Dénes”, „DEDEDE”, 1997);
```

```
...
```

```
Hallgato h5;
```

```
h5 = h4;
```

```
Console.WriteLine(h4 == h5);
```

```
h5 = new Hallgato(„Eszes Elek”, „EINST3”, 1998);
```

```
Console.WriteLine(h4 == h5);
```

Objektumreferencia  
(ezzel nem jön létre az objektum!)

Objektumlétrehozás

# Osztály szerkezete

```
class Hallgato
```

```
{
```

```
    string nev, neptun;
```

```
    int szuletesiev;
```

```
    bool aktivstatusz = true;
```

```
public Hallgato(string nev, string neptun, int szuletesiev)
```

```
{
```

```
    //...
```

```
}
```

```
public int Eletkor() { /* */ }
```

```
public void StatuszValtas(bool allapot) { /* */ }
```

```
}
```

## Osztály

Deklarálás a `class` kulcsszóval.  
Az osztályok deklarációja tartalmazza az összes tag definícióját.

## Adatmezők deklarálása

Értékük inicializálható

## Metódusok


A visszatérési értéket és a paramétereket magában foglaló, a kifejtést nem tartalmazó megadási formát szokták a függvény szignatúrájának nevezni



# Metódusok túlterhelése

- Egy osztályon belül létrehozhatunk több azonos nevű, de eltérő paraméterlistával rendelkező metódust
  - Ezzel a technikával ugyanazt a funkciót többféle paraméterrel meg tudjuk valósítani ugyanazon a néven
  - Logikusabb, átláthatóbb programozási stílust tesz lehetővé

```
public void StatuszValtas(bool  
allapot)  
{  
    /* */  
}  
  
public void StatuszValtas()  
{  
    /* */  
}
```



A visszatérési értéket és a paramétereket magában foglaló, a kifejtést nem tartalmazó megadási formát szokták a függvény szignatúrájának nevezni

# Metódusok túlterhelése

- Az osztály saját metódusainak belsejében elérjük az osztályhoz tartozó változókat, illetve az osztályban deklarált metódusokat is használhatjuk

```
class Hallgato
{
    bool aktivstatusz;
    //...
    public void StatuszValtas(bool allapot)
    {
        aktivstatusz = allapot;
    }

    public void StatuszValtas()
    {
        aktivstatusz = true;
    }
}
```

# Példányosítás, konstruktor

```
class Hallgato
```

```
{  
    public string nev, neptun;  
    public int szuletesiev;  
    public bool aktivstatusz;  
}
```

```
class Program
```

```
{  
    static void Main(string[] args)  
    {  
        Hallgato hallgato1 = new Hallgato();  
        hallgato1.szuletesiev = 1997;  
        hallgato1.nev = "Alaptalan Aladár";  
        hallgato1.neptun = "A1A1A1";  
    }  
}
```

Minden osztálynak rendelkeznie kell konstruktorral

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

A `new` operátor elvégzi az objektum számára a memórafoglalást, és meghívja a megfelelő konstruktort

Osztályok és objektumok tagjainak elérése: „.” operátor  
Példányszintű tagoknál a példány nevét, osztálysintű tagoknál (lásd később) az osztály nevét kell az operátor elé írunk  
Az osztály saját metódusainak belsejében nem kell kiírni semmit, ilyenkor egyértelmű hogy a saját tagokra gondolunk

# Konstruktor

```
class Hallgato
```

```
{
```

```
    string nev, neptun;
```

```
    int szuletesiev;
```

```
    bool aktivstatusz;
```

```
    public Hallgato(string nev_, string neptun_, int szuletesiev_)
```

```
{
```

```
        nev = nev_;
```

```
        neptun = neptun_;
```

```
        szuletesiev = szuletesiev_;
```

```
}
```

```
}
```

A konstruktor neve mindig megegyezik az osztály nevével  
Nincs visszatérési értéke (void sem)

A névütközések elkerülése végett a paraméterlista változónevei eltérőek kell legyenek

```
Hallgato hallgato1 = new Hallgato("Alapos Aladár", "A1A1A1", 1997);
```

# Konstruktor

```
class Hallgato
```

```
{
```

```
    string nev, neptun;
```

```
    int szuletesiev;
```

```
    bool aktivstatusz;
```

```
    public Hallgato(string nev_, string neptun_, int szuletesiev_)
```

```
    {
```

```
        nev = nev_; neptun = neptun_; szuletesiev = szuletesiev_;
```

```
    }
```

```
    public Hallgato(string nev_, string neptun_)
```

```
    {
```

```
        nev = nev_; neptun = neptun_; szuletesiev = 1800;
```

```
    }
```

```
}
```

Több különböző paraméterlistájú konstruktor is definiálható

```
Hallgato hallgato1 = new Hallgato("Alapos Aladár", "A1A1A1", 1997);
```

```
Hallgato hallgato2 = new Hallgato("Benedek", "BEBEBE");
```

# Konstruktor

```
class Hallgato
```

```
{
```

```
    string nev, neptun;
```

```
    int szuletesiev;
```

```
    bool aktivstatusz;
```

```
    public Hallgato(string nev, string neptun, int szuletesiev)
```

```
{
```

```
        nev = nev;
```

```
        neptun = neptun;
```

```
        szuletesiev = szuletesiev;
```

```
}
```

```
}
```

Megegyező nevek esetén ez egy ugyan helyes, de értelmetlen művelet  
Ugyanis mindkét változónév ugyanarra a változóra utal, a legközelebbire, azaz a paraméterre



# this referencia

```
class Hallgato
```

```
{
```

```
    string nev, neptun;
```

```
    int szuletesiev;
```

```
    bool aktivstatusz;
```

```
    public Hallgato(string nev, string neptun, int szuletesiev)
```

```
{
```

```
    this.nev = nev;
```

```
    this.neptun = neptun;
```

```
    this.szuletesiev = szuletesiev;
```

```
}
```

```
}
```

Referencia arra az objektumra,  
amelyik a metódust éppen  
végrehajtja

Nem kell deklarálni, ezt a  
fordítóprogram automatikusan  
megteszi

A `this` hivatkozás az aktuális  
példányra, így a példány  
adatmezőinek adjuk értékül a  
paraméterlista azonos nevű  
változóinak értékét

# this referencia

```
class Hallgato
{
    bool aktivstatusz;
    //...
    public void StatuszValtas(bool allapot)
    {
        aktivstatusz = allapot;
    }

    public void StatuszValtas()
    {
        this.StatuszValtas(true);
    }
}
```

A példány saját metódusaira is  
hivatkozhatunk.

Ezesetben persze nem szükséges a használata,  
hiszen a metódus szignatúrája egyértelműen  
azonosítja a célt.

# this referencia

```
class Hallgato
```

```
{
```

```
    private string nev, neptun;
```

```
    public int szuletesiev;
```

```
    bool aktivstatusz;
```

```
    public Hallgato(string nev, string neptun, int szuletesiev)
```

```
{
```

```
        this.nev = nev;
```

```
        this.neptun = neptun;
```

```
        this.szuletesiev = szuletesiev;
```

```
}
```

this referencia alkalmazható arra is, hogy egyik konstruktor meghívjon egy másikat

```
public Hallgato(string nev, string neptun)
```

```
    : this(nev, neptun, 1800)
```

```
{ }
```

```
}
```

# Destruktor

- .NET-ben az objektumok megszüntetése automatikus (GC), az objektum akkor szűnik meg, amikor biztosan nincs már rá szükség
- A destruktork egy olyan metódus, amely akkor fut le, amikor a példány megsemmisül
  - Fő feladata: erőforrás-felszabadítás
  - Nem hívható, a GC hívja *ismeretlen időben*
  - C#-ban a felesleges destruktork lassítják a GC működését – kerüljük, és üres destruktort sohasem írunk
    - Neve egy tildéből (~) és az osztály nevéből áll
    - Nincs visszatérési értéke, paramétere, láthatósági szintje

```
~Hallgato()  
{  
    Console.WriteLine("Viszlát!");  
}
```

# Láthatóság

- Az osztály minden tagjához, a mezőkhöz és a metódusokhoz is láthatósági szint van rendelve
- Azt befolyásolja, hogy elérhető-e az adott tag osztályon kívülről, vagy csak belül érvényes
  - Publikus: nyilvános, osztályon kívülről is elérhető
  - Privát: csak az osztályon belül érhető el
  - Léteznek egyéb láthatóságok is, ezeket a következő félévekben tárgyaljuk
- Célunk a lehető legszűkebb elérés megvalósítása!
- A tagok alapértelmezett láthatósága privát!

```
private string nev, neptun;  
public int szuletesiev;  
bool aktivstatusz;
```

ÓÉ-NIK,

```
hallgato1.szuletesiev = 1997;  
hallgato1.nev = "Alaptalan Aladár";  
hallgato1.neptun += "A";  
hallgato1.aktivstatusz = !hallgato1.ak
```

# Tulajdonság

- Tulajdonság segítségével az olvasás és írás metódusok működése a programban megadhatóak
- A felhasználó kód számára metódusként viselkednek
- A hozzáférési metódusok bármilyen műveletet végrehajthatnak
  - Nem célszerű hosszan tartó műveletekkel lelassítani a tulajdonság elérését

```
class Hallgato
```

```
{
```

```
    private int szuletesiev;
```

```
    public int SzuletesiEv
```

```
{
```

```
        get { return szuletesiev; }
```

```
        set { szuletesiev = value; }
```

```
    }
```

```
}
```

A tulajdonság elnevezése gyakran az adatmező nevének nagy kezdőbetűs változata

A get az olvasáskor, a set az íráskor lefutó műveleteket tartalmazza.  
Olvasáskor vissza kell adni egy értéket a return kulcsszóval.  
Írás esetén a híváskor átadott értéket a value nevű rejtett paraméter tartalmazza



# Tulajdonság

```
public int SzuletesiEv { get { return szuletesiev; } }  
public string Neptun { set { nev = value; } }  
public string Adatok  
{  
    get { return nev + " (" + neptun + "), született: " + szuletesiev; }  
}  
public string Nev  
{  
    get { return nev; }  
    set  
    {  
        if (value != "")  
            nev = value;  
    }  
}
```

Csak olvasható és csak írható tulajdonságok is definiálhatóak

Ha nem adjuk meg az írási, illetve olvasási metódust, akkor csak olvasható, illetve csak írható tulajdonság jön létre

Készíthető olyan tulajdonság, amely olvasáskor „on-the-fly” kap értéket

Nem tartozik hozzá adatmező

Bármilyen művelet végrehajtható

# Tulajdonság

```
Console.WriteLine(h1.Adatok);  
h1.Adatok = "valami";  
Console.WriteLine(h1.Neptun);  
h1.Neptun = "valami";  
Console.WriteLine(h1.Nev);  
h1.Nev = "";  
Console.WriteLine(h1.SzuletesiEv);  
h1.SzuletesiEv = 1900;
```

Csak olvasható, illetve csak írható  
értékek

- ❌ 1 Property or indexer 'ConsoleApplication1.Hallgato.Adatok' cannot be assigned to -- it is read only
- ❌ 2 The property or indexer 'ConsoleApplication1.Hallgato.Neptun' cannot be used in this context because it lacks the get accessor
- ❌ 3 Property or indexer 'ConsoleApplication1.Hallgato.SzuletesiEv' cannot be assigned to -- it is read only

# Osztályszintű tagok

	Osztály szintű tagok	Példány (objektum) szintű tagok
Mezők (adattagok)	Egyetlen, az <u>osztályhoz</u> tartozó példány	Minden egyes <u>objektumhoz</u> saját példány
Metódusok (funkciótagok)	Osztályonként egy-egy példány ( <u>nincs</u> konkrét objektum, amelyen működnek)	Osztályonként egy-egy példány ( <u>van</u> konkrét objektum, amelyen működnek)

- A **static** kulcsszóval képezzük
- Az eddig definiált tagok mind példányszintűek voltak

```
class Hallgato
```

```
{
```

```
    public static int alapOsztondij = 100;
```

```
    static double osztondijkuszob = 3.0;
```

```
    public static void KuszobNovel(double ertek) { /* */ }
```

```
    public static double OsztondijKuszob
```

```
    {
```

```
        get { return osztondijkuszob; }
```

```
    }
```

```
    //...
```

```
}
```

Adatmezők lehetnek statikusak,  
és ugyanúgy láthatóság is  
megadható  
Célszerű inicializálni

Metódusok és tulajdonságok is  
lehetnek osztályszintűek

# Osztályszintű tagok

- Az osztály nevére kell hivatkozni eléréskor
- Nem szükséges hogy létezzen példány

```
Hallgato.alapOsztonDij = 150;  
Console.WriteLine(Hallgato.OsztonDijKuszob);  
Hallgato.KuszobNovel(0.6);
```

- Sok példa található a beépített osztályokban

– `int.Parse()`, `Console.ReadLine()`, vagy a `Main()` metódus

```
static void Main(string[] args)  
{  
    int a = int.Parse("23");  
    Console.ReadLine();  
}  
  
static void Koszon(string nev) { /**/ }
```

Eddig – nem teljesen tudatosan – csak osztályszintű metódusokat készítettünk a laborgyakorlatokon

# Példányszint vs osztálysint

- Példányszintű módszer elérheti az osztálysintű tagokat
- Ellenben az osztálysintű módszer nem éri el a példányszintű adattagokat

```
class Hallgato
{
    //...
    double osztondijAtlag;
    static int alapOsztondij = 100;
    static double osztondijkuszob = 3.0;

    public int Osztondij()
    {
        if (osztondijAtlag < osztondijkuszob)
            return 0;
        return (int)(osztondijAtlag * alapOsztondij);
    }
}
```

# Mikor alkalmazzunk osztályszintű metódust?

- **Osztályszintű, ha az adott példánytól független a működés, amit reprezentál**
- **Példányszintű, ha adott példány állapotát módosítja**
  - Sok esetben ugyanazt a funkcionalitást statikus és nem statikus formában is meg tudjuk valósítani
- **Gyakran bizonytalan**
  - Ízlés és a fejlesztői közösség szokása dönthet, sokan a példányszintű metódusokat preferálják ilyenkor
- **Egyéb szempontok:**
  - Ha bárhonnán, bármikor el kell tudni érni (kisebb, gyakran használt segédfüggvények), akkor sokszor statikus (pl. Math osztály függvényei)



# Példák osztályszintű metódusokra

- Példányoktól független működésű, pl.: `Parse()`

```
class Hallgato
{
    //...
    public static Hallgato Parse(string s)
    {
        string[] arr = s.Split(';');
        return new Hallgato(arr[0], arr[1], int.Parse(arr[2]));
    }
}
```

Az osztályszintű metódus visszatérési értéke egy azonos osztályú objektum!

```
Hallgato h1 = Hallgato.Parse("Alapos Aladár;A1A1A1;1997");
```

# Példák osztályszintű metódusokra

```
class Hallgato
{
    double osztondijAtlag;
    //...
    public static Hallgato JobbAtlagu(Hallgato egyik, Hallgato masik)
    {
        if (egyik.osztondijAtlag >= masik.osztondijAtlag)
            return egyik;
        return masik;
    }
}
```

Mivel az objektumok referenciák, ezért a visszakapott érték az egyik paraméterként átadott objektum lesz

```
Hallgato jobb = Hallgato.JobbAtlagu(h1, h2);
```

# Példafeladat

- **Készítsünk alkalmazást, amelyben hallgatók ösztöndíj-kifizetéseit tudjuk kezelni!**
- **Ösztöndíjat az ösztöndíjátlaguk alapján kaphatnak a hallgatók, ez az összeg az alapösztöndíj és az átlag szorzata.**
- **Csak olyan aktív státuszú hallgatók kaphatnak ösztöndíjat, akik elérik a meghatározott ösztöndíjküszöböt!**

# Példafeladat – szükséges adatok





- A hallgatók alapadatain kívül (név, neptun, tanulmányi státusz) szükség van az ösztöndíjátlagukra is, amely egy 0 és 5 közötti valós érték.
- Az alapösztöndíj és az ösztöndíjküszöb nem függ az egyes hallgatóktól: előbbi egy egész érték, utóbbi 0 és 5 közötti valós
- Legyen lehetőség az alapösztöndíj módosítására és a küszöb növelésére is!















# Példafeladat - osztályszerkezet

- Az `alapOsztondij` és `osztondijKuszob` a példányoktól független, így osztálszintű adatmezőként definiálható
- Minden adatmezőt rejtett láthatósággal látunk el
- Az elérésükre tulajdonságokat hozunk létre
- Az osztálszintű mezőkhöz is
- A küszöb növelésére osztálszintű metódust definiálunk

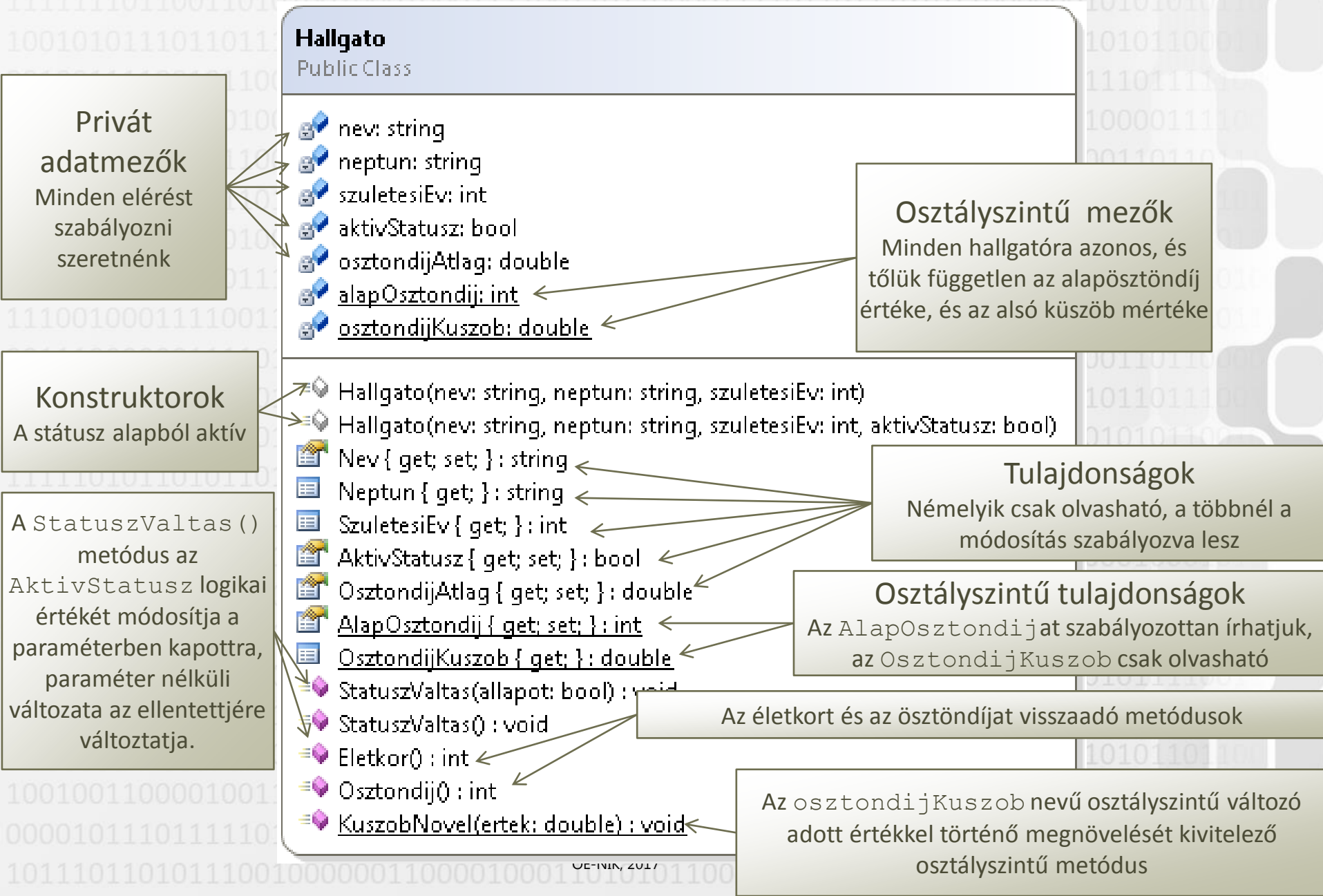
## Hallgato

Public Class

 `nev: string`  
 `neptun: string`  
 `szuletesiEv: int`  
 `aktivStatusz: bool`  
 `osztondijAtlag: double`  
 `alapOsztondij: int`  
 `osztondijKuszob: double`

 `Hallgato(nev: string, neptun: string, szuletesiEv: int)`  
 `Hallgato(nev: string, neptun: string, szuletesiEv: int, aktivStatusz: bool)`  
 `Nev { get; set; } : string`  
 `Neptun { get; } : string`  
 `SzuletesiEv { get; } : int`  
 `AktivStatusz { get; set; } : bool`  
 `OsztondijAtlag { get; set; } : double`  
 `AlapOsztondij { get; set; } : int`  
 `OsztondijKuszob { get; } : double`  
 `StatuszValtas(allapot: bool) : void`  
 `StatuszValtas() : void`  
 `Eletkor() : int`  
 `Osztondij() : int`  
 `KuszobNovel(ertek: double) : void`

# Példafeladat – osztályszerkezet





# Példafeladat – osztály, adatmezők

- Láthatóság megjelölése nélkül alapértelmezetten minden privát láthatóságú
- Adatmezőknél ez a célunk, a hozzáférést tulajdonságokon keresztül célszerű kivitelezni

```
class Hallgato
{
    string nev, neptun;
    int szuletesiEv;
    bool aktivStatusz;
    double osztondijAtlag;
    //...
}
```

# Példafeladat – konstruktorok

- A státusz megadása nem kötelező, ösztöndíjátlagot futás közben kap

```
public Hallgato(string nev, string neptun, int születésiEv, bool aktivStatusz)
{
    this.nev = nev;
    this.neptun = neptun;
    this.születésiEv = születésiEv;
    this.aktivStatusz = aktivStatusz;
}

public Hallgato(string nev, string neptun, int születésiEv)
{
    this.nev = nev;
    this.neptun = neptun;
    this.születésiEv = születésiEv;
    aktivStatusz = true;
}
```

Így kódismétlés

Az egyik konstruktor hívható a másikkal

# Példafeladat – konstruktorok

- A státusz megadása nem kötelező, ösztöndíjátlagot futás közben kap

```
public Hallgato(string nev, string neptun, int születésiEv, bool aktivStatusz)
{
    this.nev = nev;
    this.neptun = neptun;
    this.születésiEv = születésiEv;
    this.aktivStatusz = aktivStatusz;
}

public Hallgato(string nev, string neptun, int születésiEv)
    : this(nev, neptun, születésiEv, true)
{ }
```

A konstruktor, ahol az alapértelmezett érték van, hivatkozik a másikra a `this` referencián keresztül

# Példafeladat – tulajdonságok

- A név írható/olvasható, de üresen nem hagyható
- A neptun azonosító és a születési év olvashatóak

```
public string Nev
{
    get { return nev; }
    set
    {
        if (value != "")
            nev = value;
    }
}
```

```
public string Neptun { get { return neptun; } }
public int SzuletesiEv { get { return szuletesiEv; } }
```

# Példafeladat – tulajdonságok

- A státusz írható/olvasható
- Az átlag írható/olvasható, de 0.0 és 5.0 közötti

```
public bool AktivStatusz
{ get { return aktivStatusz; } set { aktivStatusz = value; } }

public double OsztondivAtlag
{
    get { return osztondijAtlag; }
    set
    {
        if (value >= 0.0 && value <= 5.0)
            osztondijAtlag = value;
    }
}
```

# Példafeladat – metódusok

- A státuszváltás adott értékre állítja, vagy átbillenti a státuszt
- Az életkort a születési évből számolni tudjuk

```
public void StatuszValtas(bool allapot)
{
    aktivStatusz = allapot;
}

public void StatuszValtas()
{
    aktivStatusz = !aktivStatusz;
}

public int Eletkor()
{
    return DateTime.Now.Year - születésiEv;
}
```

Az életkort visszaadó metódus „on-the-fly” kiértékelődő tulajdonságként is definiálható



# Példafeladat – metódusok

- A státuszváltás adott értékre állítja, vagy átbillenti a státuszt
- Az életkort a születési évből számolni tudjuk

```
public void StatuszValtas(bool allapot)
```

```
{  
    aktivStatusz = allapot;  
}
```

```
public void StatuszValtas()
```

```
{  
    aktivStatusz = !aktivStatusz;  
}
```

```
public int Eletkor
```

```
{  
    get { return DateTime.Now.Year - születésiEv; }  
}
```

Így nézne ki tulajdonságként  
De egyszerre mindkettő nem definiálható  
ugyanazon a néven, névütközést okoz!

# Példafeladat – osztályszintű tagok

- A továbbiakhoz szükségesek az osztályszintű tagok
- Mindkét osztályszintű mező privát, tulajdonság tartozik hozzájuk: a küszöb csak olvasható, az alap módosítható

```
static int alapOsztondij = 100;
static double osztondijKuszob = 3.0;

public static double OsztondijKuszob
{
    get { return osztondijKuszob; }
}

public static int AlapOsztondij
{
    get { return alapOsztondij; }
    set { alapOsztondij = value; }
}
```

Mivel osztályszintű mezők,  
ezért célszerű kezdőértékkel  
ellátni őket

# Példafeladat – osztályszintű metódus

- A küszöb növelésére saját metódust definiálunk
- A küszöb új értéke a metódusnak paraméterként adott értékkel való összegeként áll elő

```
public static void KuszobNovel(double ertek)
{
    osztondijKuszob += ertek;
}
```

# Példafeladat – ösztöndíj számítása

- Ösztöndíjat az ösztöndíjátlaguk alapján kaphatnak a hallgatók, ez az összeg az alapösztöndíj és az átlag szorzata.
- Csak olyan aktív státuszú hallgatók kaphatnak ösztöndíjat, akik elérik a meghatározott ösztöndíjküszöböt!

```
public int Osztondij()  
{  
    if (!aktivStatusz || osztondijAtlag < osztondijKuszob)  
        return 0;  
    return (int)(osztondijAtlag * alapOsztondij);  
}
```

A szorzás operandusai double illetve int típusúak, így az eredmény a bővebb halmazbeli double típusú lesz. Az eredményt így típuskényszeríteni kell egész típusúra.

# Példafeladat – alkalmazás

- Hallgató létrehozásával, és az ösztöndíj lekérdezésével tesztelhető a működés

```
static void Main(string[] args)
{
    Hallgato h1 = new Hallgato("Alapos Aladár", "A1A1A1", 1997);
    h1.OsztondijAtlag = 3.8;
    Console.WriteLine("{0} nevű hallgató ösztöndíja: {1} fabatka.",
h1.Nev, h1.Osztondij());
}
```

Alapos Aladár nevű hallgató ösztöndíja: 380 fabatka.

# Példafeladat – alkalmazás

- Hallgatókból álló tömböt kell létrehozni

```
static void Main(string[] args)
{
```

```
    Hallgato[] tomb = new Hallgato[3];
```

```
    tomb[0] = new Hallgato("Alapos Aladár", "A1A1A1", 1997);
```

```
    tomb[1] = new Hallgato("Bukott Bendegúz", "BATMAN", 1991, false);
```

```
    tomb[2] = new Hallgato("Csupaötös Csaba", "C5UP45", 1995);
```

```
    tomb[0].OsztondijAtlag = 3.8;
```

```
    tomb[1].OsztondijAtlag = 3.5;
```

```
    tomb[2].OsztondijAtlag = 5.0;
```

```
    foreach (Hallgato h in tomb)
```

```
        Console.WriteLine("{0} nevű hallgató ösztöndíja: {1}
```

```
        fabatka.", h.Nev, h.Osztondij());
```

```
    }
```

Fontos észrevenni, hogy a tömb deklarálásával még nincsenek objektumaink, a tömb minden eleme nullreferencia!

Kétféle konstruktor is létezik, az egyik vár negyedik logikai paramétert, a másik nem

```
Alapos Aladár nevű hallgató ösztöndíja: 380 fabatka.
Bukott Bendegúz nevű hallgató ösztöndíja: 0 fabatka.
Csupaötös Csaba nevű hallgató ösztöndíja: 500 fabatka.
```



# Példafeladat – alkalmazás

- Az osztályszintű küszöb növelhető

```
//...  
Hallgato.KuszobNovel(1.0);  
foreach (Hallgato h in tomb)  
    Console.WriteLine("{0} nevű hallgató ösztöndíja: {1} fabatka.",  
        h.Nev, h.Osztondij());
```

```
Alapos Aladár nevű hallgató ösztöndíja: 0 fabatka.  
Bukott Bendegúz nevű hallgató ösztöndíja: 0 fabatka.  
Csupaötös Csaba nevű hallgató ösztöndíja: 500 fabatka.
```

# Példafeladat – alkalmazás

- És az alapösztöndíj is módosítható

```
//...  
Hallgato.AlapOsztondij = 250;  
foreach (Hallgato h in tomb)  
    Console.WriteLine("{0} nevű hallgató ösztöndíja: {1} fabatka.",  
        h.Nev, h.Osztondij());
```

```
Alapos Aladár nevű hallgató ösztöndíja: 0 fabatka.  
Bukott Bendegúz nevű hallgató ösztöndíja: 0 fabatka.  
Csupaötös Csaba nevű hallgató ösztöndíja: 1250 fabatka.
```

# NÉVTEREK

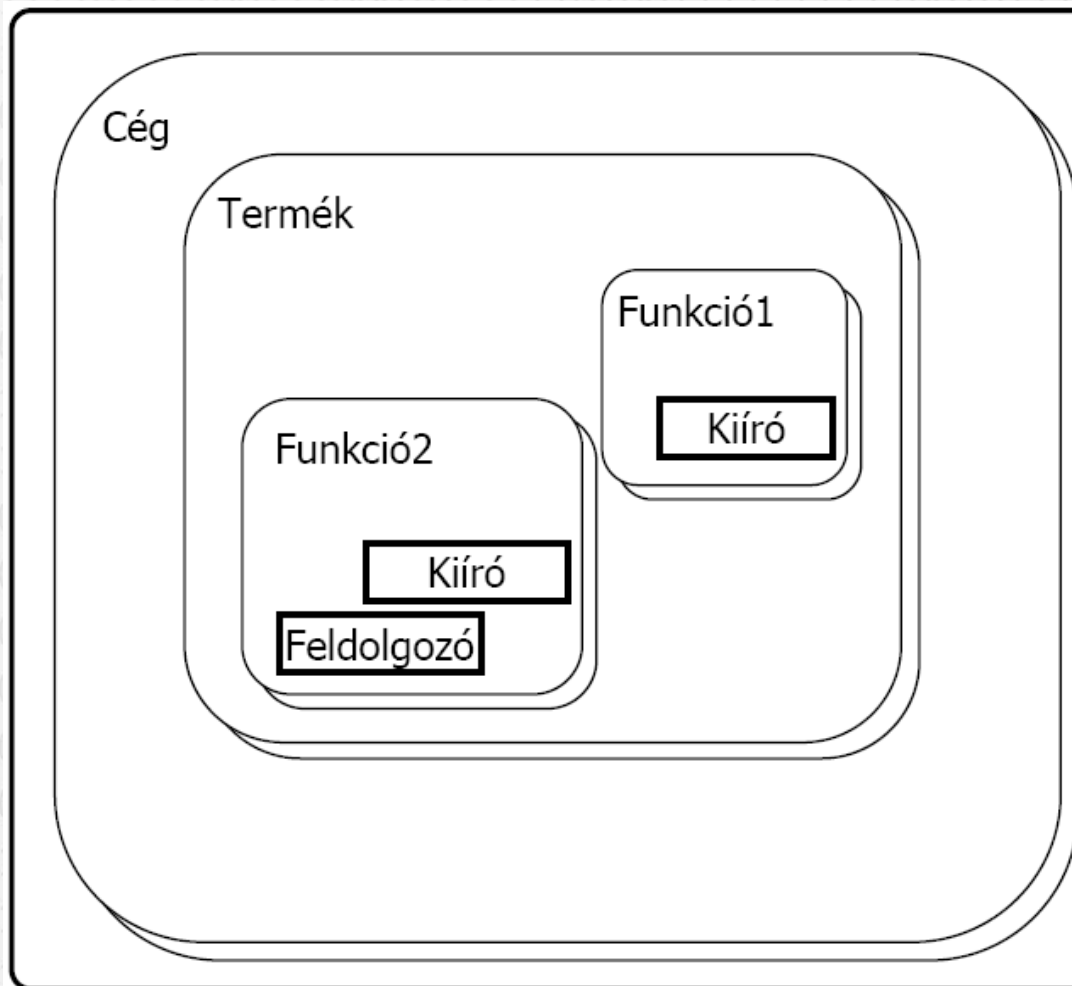
# Névterek 1.

- **Az OO paradigma jellemzője az elnevezések óriási száma**
  - Minden osztálynak, objektumnak, mezőnek, metódusnak egyedi nevet kell adni, hogy a későbbiekben hivatkozni lehessen rájuk
    - Nem könnyű jól megjegyezhető, a célt később is felidéző neveket adni
- **A programok méretével egyenes arányban nő a névütközések valószínűsége**
  - A programok általában nem csak saját osztályokat használnak fel

# Névterek 2.

- **A névtér, mint az elnevezések érvényességének tartománya, hierarchikus logikai csoportokra bontja az elnevezéseket**
  - Minden elnevezésre csak a saját névterén belül lehet hivatkozni – külön jelzés nélkül
  - Ennek megfelelően a saját névterén belül minden elnevezés egyedi
  - A névterek általában tetszőleges mélységben egymásba ágyazhatók
  - Azonos elnevezések más-más névtereken belül szabadon használhatók, így erősen lecsökken a névütközési probléma jelentősége

# Névterek - példa



Szintaktikai variációk névtérre:

Cég.Termék.Funkció1.Kiíró  
Cég.Termék.Funkció2.Kiíró  
Cég.Termék.Funkció2.Feldolgozó

Cég/Termék/Funkció1/Kiíró  
Cég/Termék/Funkció2/Kiíró  
Cég/Termék/Funkció2/Feldolgozó

„Névtérpótlék” technika:

Cég\_Termék\_Funkció1\_Kiíró  
Cég\_Termék\_Funkció2\_Kiíró  
Cég\_Termék\_Funkció2\_Feldolgozó



# Névterek

- **A névterek az elnevezések tetszőleges logikai csoportosítását teszik lehetővé**
  - **Nincs közülük a fizikai tároláshoz (fájlokhoz és mappákhoz)**
    - Egy fájlban több névtér, egy névtér több fájlban is elhelyezhető
  - **Tetszőlegesen egymásba ágyazhatók**
    - A beágyazott névterek tagjait a „.” karakterrel választhatjuk el
  - **A névtérbe be nem sorolt elemek egy ún. globális névtérbe kerülnek**

```
namespace A
{
    namespace B
    {
        class Egyik {...}
    }
}
```

```
...
A.B.Egyik példa = new A.B.Egyik();
```

```
namespace A.B
{
    class Másik {...}
}
namespace C
{
    class Harmadik {...}
}
```

```
...
A.B.Másik példa2 = new A.B.Másik();
C.Harmadik példa3 = new C.Harmadik();
```

- A névterek nevének Microsoft által javasolt formátuma:  
`<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]`

# Névterek

- Minden névre a saját névterével együtt kell hivatkozni

```
System.Console.WriteLine();  
System.Threading.Thread.Sleep(100);  
pi = System.Math.PI;
```

- Ez az ún. **teljesen minősített név** (fully qualified name), formája:  
névtér.alnévtér.alnévtér(...).elnevezés

- A névterek hivatkozás céljára előkészíthetők a **using** kulcsszó segítségével

- Ezt követően az adott névtérben található elnevezések elé hivatkozáskor nem kell kiírni a névteret, feltéve, hogy az elnevezés így is egyértelműen azonosítható

```
using System;  
using System.Text;
```

- A névtereknek álnév is adható

- Célja a hosszú névterek egyértelmű rövidítése

```
using System;  
using SOAP = System.Runtime.Serialization.Formatters.Soap;  
...  
SOAP.SoapFormatter formazo = new SOAP.SoapFormatter();  
Console.WriteLine(formazo);
```

# OBJEKTUM-ORIENTÁLTSAĞ ALAPELVEI

# OOP alapelvek

1. Absztrakció
2. Egységbezárás
3. Adatrejtés
4. Öröklés
5. *Polimorfizmus*
6. *Kódújrafelhasználás*

# 1. alapelv: absztrakció

- **Meghatározzuk a szoftverrendszer absztrakt elemeit**
- **Meghatározzuk az elemek állapotterét**
  - Adatelemek
- **Meghatározzuk az elemek viselkedésmódját**
  - Funkciók végrehajtása
  - Állapotváltoztatások
- **Meghatározzuk az elemek közötti kapcsolattartás felületeit és protokollját**
- **Az osztály mindig az adott feladatban fontos tulajdonságokat/képességeket tartalmazza.**
- **Oka: egyszerűsítés**
- **Eszközrendszere: class definíció**

## 2. alapelv: egységbezárás

- **Az objektumok adatait és a rajtuk végezhető műveleteket szoros egységbe zárjuk**
  - Pontosan definiáljuk az adatok eléréséhez/használatához szükséges műveleteket
  - Elvileg az adatok csak ezek segítségével érhetők el, és más műveletek az objektummal nem végezhetők
- **Meghatározott feladatot ellátó, „öntartalmazó” egységként kezeljük őket**
- **Oka:**
  - Átláthatóság
  - Újrafelhasználhatóság
  - Adatok fizikailag közel vannak a rajtuk dolgozó műveletekhez, ami a módosíthatóságot segíti
- **Eszközrendszer: class definícióban adatok+műveletek**

# 3. alapelv – adatrejtés

- **Pontosan meghatározzuk az osztály kifelé való „kapcsolódási pontjait”**
  - Az objektumokon belül elkülönítjük a belső (privát) és a külső (nyilvános) adatokat és műveleteket
  - A belső használatú műveletek privátok
  - A nyilvános adatok és műveletek a szoftverrendszer többi objektuma számára (is) elérhetők
    - Tájékozódás az objektum állapotáról
    - Az objektum állapotának módosítása
- **Oka:**
  - Pontos megvalósítás, pontos adatábrázolás elrejtése
  - Az objektum rejtett része meghatározza a szabadon módosítható részeket
  - Az adatelérés kontrollálása véd a hibáktól
- **Eszközrendszere: láthatóságok, tulajdonságok getter/setter metódusai (még publikus esetben is)**