Objektumorientált paradigma
Alapok

Vámossy Zoltán
vamossy.zoltan@nik.uni-obuda.hu

Óbudai Egyetem
Neumann János Informatikai Kar

Előadás tematika

Cél: az objektumorientált programozási paradigma szemléletének és elemeinek megismertetése

Tartalom:

- OOP kialakulásának oka, főbb jellemzői
- Absztrakció, osztály, objektum
- Egységbezárás, adatrejtés, öröklés*, polimorfizmus*, kódújrafelhasználás

*: Az öröklés és polimorfizmus részletei a következő félévben kerülnek érdemi ismertetésre

Objektumorientált paradigma

AZ OOP KIALAKULÁSÁNAK OKA ÉS FŐBB JELLEMZŐI

Bevezetés – Miért OOP?

- A hálózati-, és multimédia rendszerek készítésével a szoftverek egyre bonyolultabbá váltak és a minőségi követelmények is növekedtek
- Csoportfejlesztésű rendszerek
- *A szoftver krízis** a szoftverfejlesztés válsága, miszerint egy hagyományos módszer már nem képes az igényeknek megfelelő, **minőségi szoftver** előállítására

*: Részleteket lásd majd Szoftvertechnológia előadásban

Szoftver minőség 1.

A felhasználó a minőséget *külső jegyekben* méri le:

- A program *helyesen*
- és *gyorsan működjön,*
- legyen *megbízható,*
- *felhasználóbarát*
- és *továbbfejleszthető*

Szoftver minőség 2.

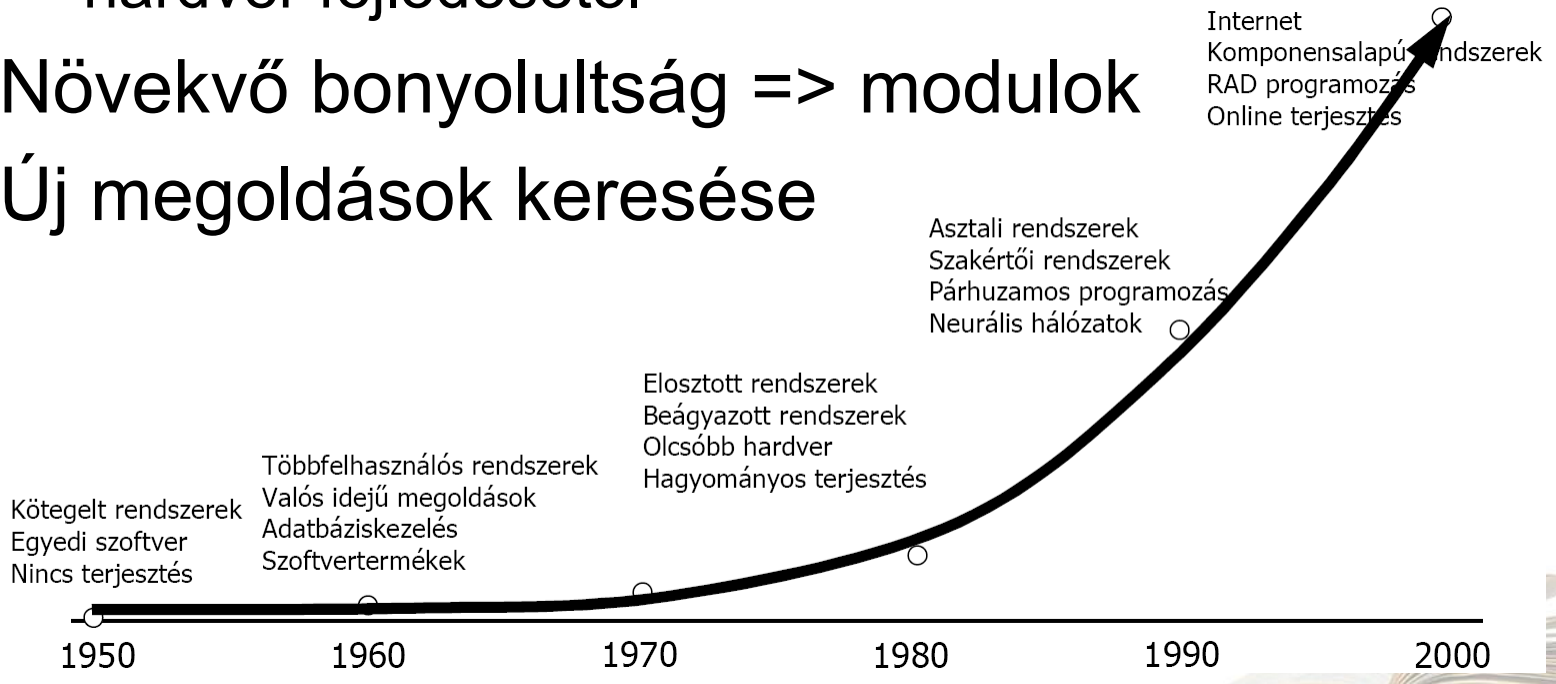
- A szoftverfejlesztő a *kívülről látható* és mérhető *minőséget* csak egy belső rend felállításával érheti el. Ezért *a belső minőség* kialakítása állandó törekvés.
- A programok bonyolultsága következtében azonban a minőséget egyre kevésbé lehetett tartani az addigi, hagyományos (strukturált) módszerekkel.

Szoftver minőség 3.

- Helyesség
- Hibatűrés
- Karbantarthatóság, bővíthetőség
- Újrafelhasználhatóság
- Felhasználó-barátság
- Hordozhatóság
- Hatékonyság
- Ellenőrizhetőség
- Szabványosság
- Integritás (sérthetetlenség)

Szoftverkrízis

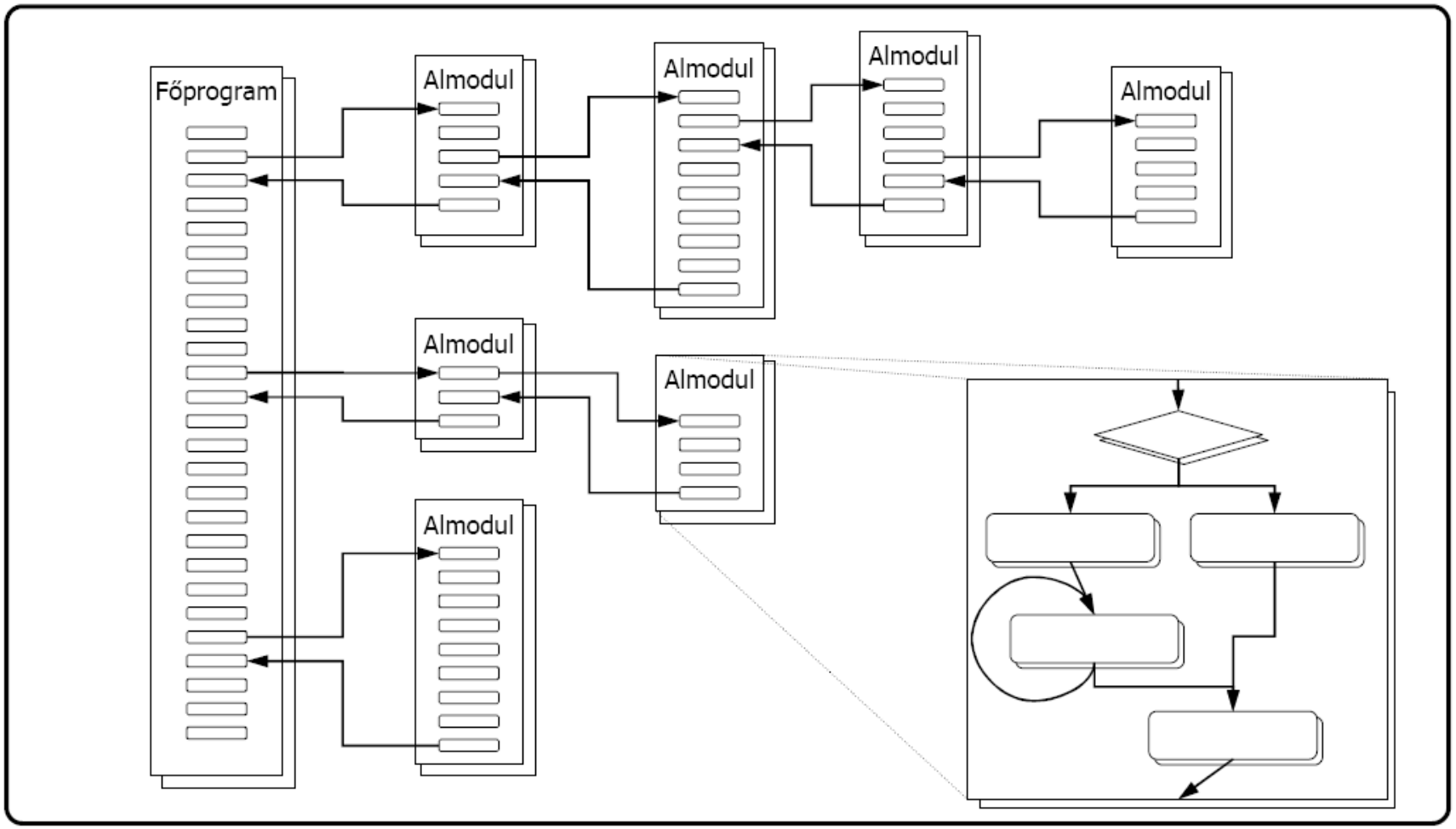
- A szoftveres megoldások szerepe folyamatosan erősödik, de
  - a szoftverek fejlődési üteme egyre jobban lemarad a hardver fejlődésétől
- Növekvő bonyolultság => modulok
- Új megoldások keresése



Procedurális/strukturált program

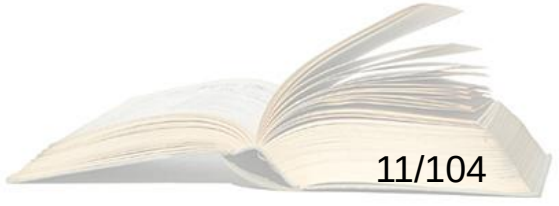
- A problémát az **algoritmus** (a **kód**, a **funkció**) oldaláról közelíti meg
  - A szükséges funkciók meghatározása (funkcionális dekompozíció)
  - A programvégrehajtás (vezérlés) pontos menetének leírása
  - A funkciók számára szükséges adatstruktúrák meghatározása
  - A probléma megoldását a funkciók egymás után, a megfelelő sorrendben történő végrehajtása adja meg

Strukturált program felépítése



Procedurális/strukturált program

- Jellemzők:
  - A függvények (modulok) definíciója határozza meg a programszerkezetet
  - Globális adatstruktúrákkal dolgozik
  - Egy ún. „főprogram” fogja össze, amely „függvényeket” „hív meg”
    - A főprogram komoly szerepet játszik és gyakran igen bonyolult
  - A végrehajtás menetét szigorúan megszabja a megírt programkód
  - Top-down jellegű feladatlebontás



Példa strukturált megoldásra*

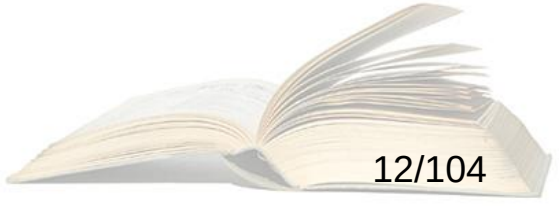
Feladat:

- Hány különböző elem van egy adott tömbben?
  - Feltételezzük, hogy a tömb pozitív egészeket tartalmaz

Terv:

- tömb feltöltése és kiírása
- számlálás: végig nézzük a tömb elemeit, és ha egy elem korábban még nem fordult elő (ehhez keresni kell őt a tömb korábbi elemei között), akkor megszámoljuk
- eredmény megjelenítése

*: Gregorics Tibor (ELTE IK) mintapéldája alapján



Példa strukturált megoldásra

Implementálás:

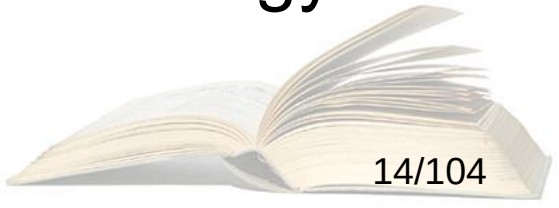
- Négy függvényt készítünk a Main() függvényen kívül: Feltölt(), Kiír(), KülönbözőtSzámlál(), Eldöntés()
- A tömböt, mint központi adatot a függvényeket tartalmazó osztály szintjén definiáljuk
- A Main() függvény hívja a Feltölt(), Kiír(), KülönbözőtSzámlál() függvényeket, az utóbbi hívja a Eldöntés()-t.
- Inputellenőrzésre Olvas() függvényt is készítünk.

Objektumorientált paradigma (OOP)

Új szemléletű módszerre volt szükség

- *Objektumorientált paradigma* a programnyelvek széles körében elterjedt:
  - Modularizáltság jellemzi (adat dekompozíció)
  - Áttekinthető, karbantartható
  - Gyorsabban és biztonságosabban érhetünk célt a szoftverfejlesztés (nagy rendszerek) területén
  - Újrafelhasználhatóságot támogatja

Cél: minden adat és hozzá tartozó funkció egy helyen jelenjen meg!

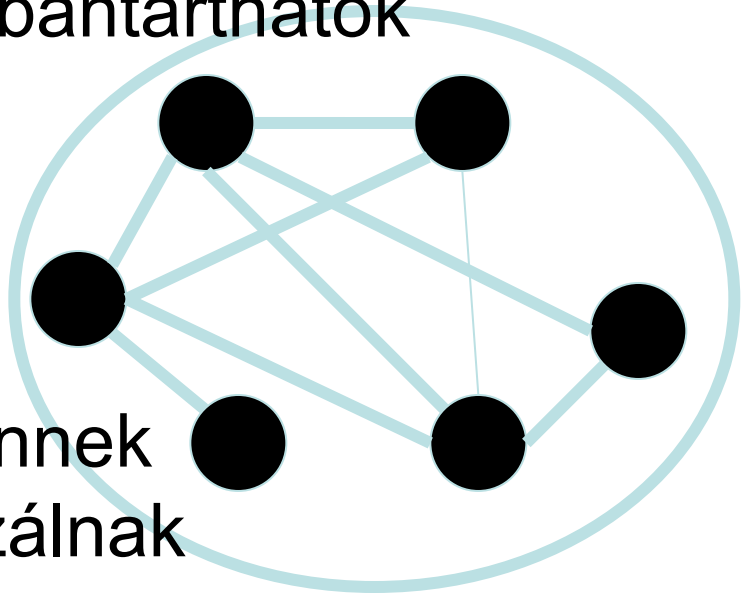


Moduláris és OO programozás 1.

OO modulok jellemzői

OO modulok:

- Külön-külön jól definiált feladatot látnak el
- Könnyen változtathatók, karbantarthatók
- Védettek
- Újrafelhasználhatók
- Könnyen összeépíthetők
- Objektumok születnek és tűnnek el, más objektumokat aktivizálnak
- Egy objektum metódusa hívja meg a másik objektum metódusát és kezeli annak eredményét

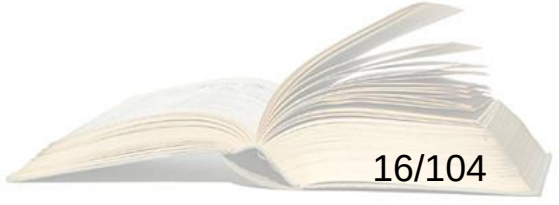


Moduláris és OO programozás 2.

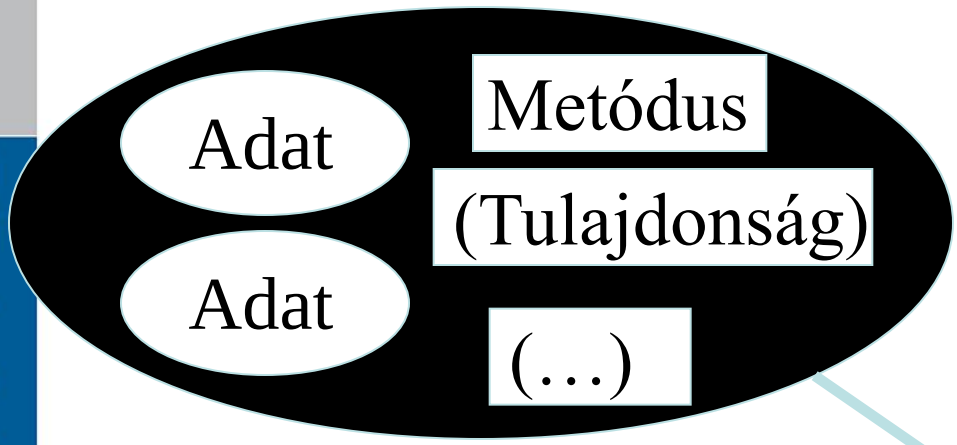
OO modulok jellemzői

Modulon *belüli erős összetartás* és a modulok *közötti laza kapcsolat* elveit kell betartani:

- Szintaktikailag jól elkülöníthető egységek
- Minél kevesebb legyen a modulok közötti kapcsolatok száma
- Minél kisebb legyen a modul kapcsolódási pontja, legyen egyértelmű, jól definiált
- A modul minél nagyobb része legyen zárt és sérthetetlen – rejtsük el azt az információt



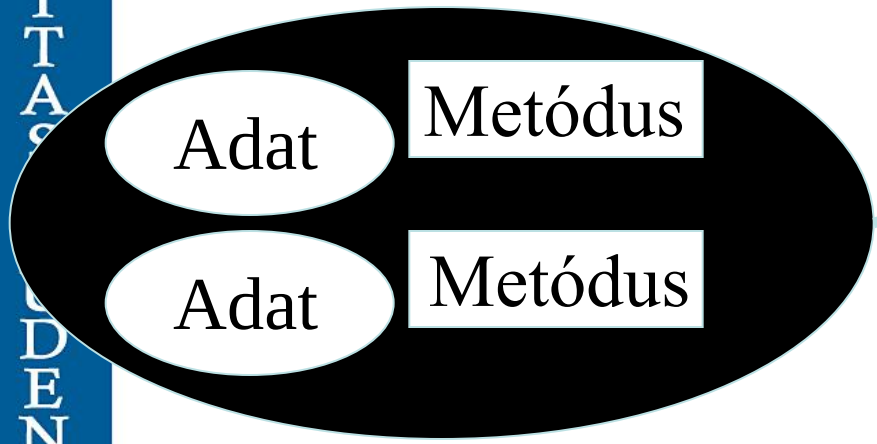
Modulok az OO rendszerben

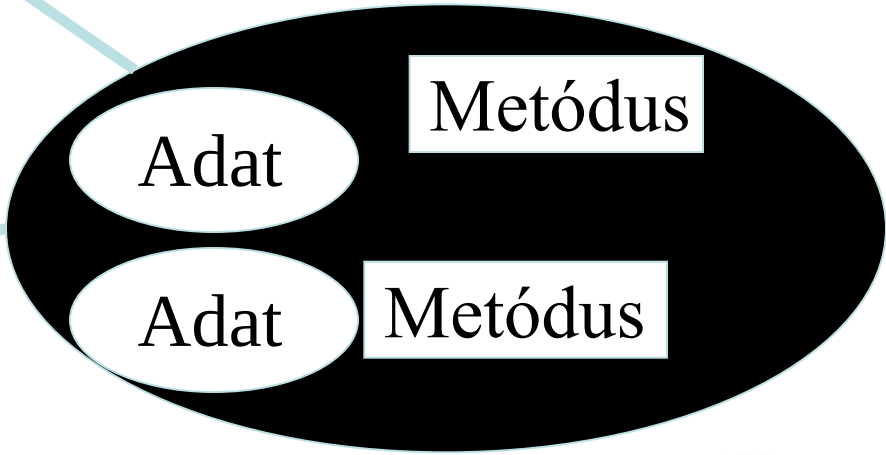


Szinonimák:

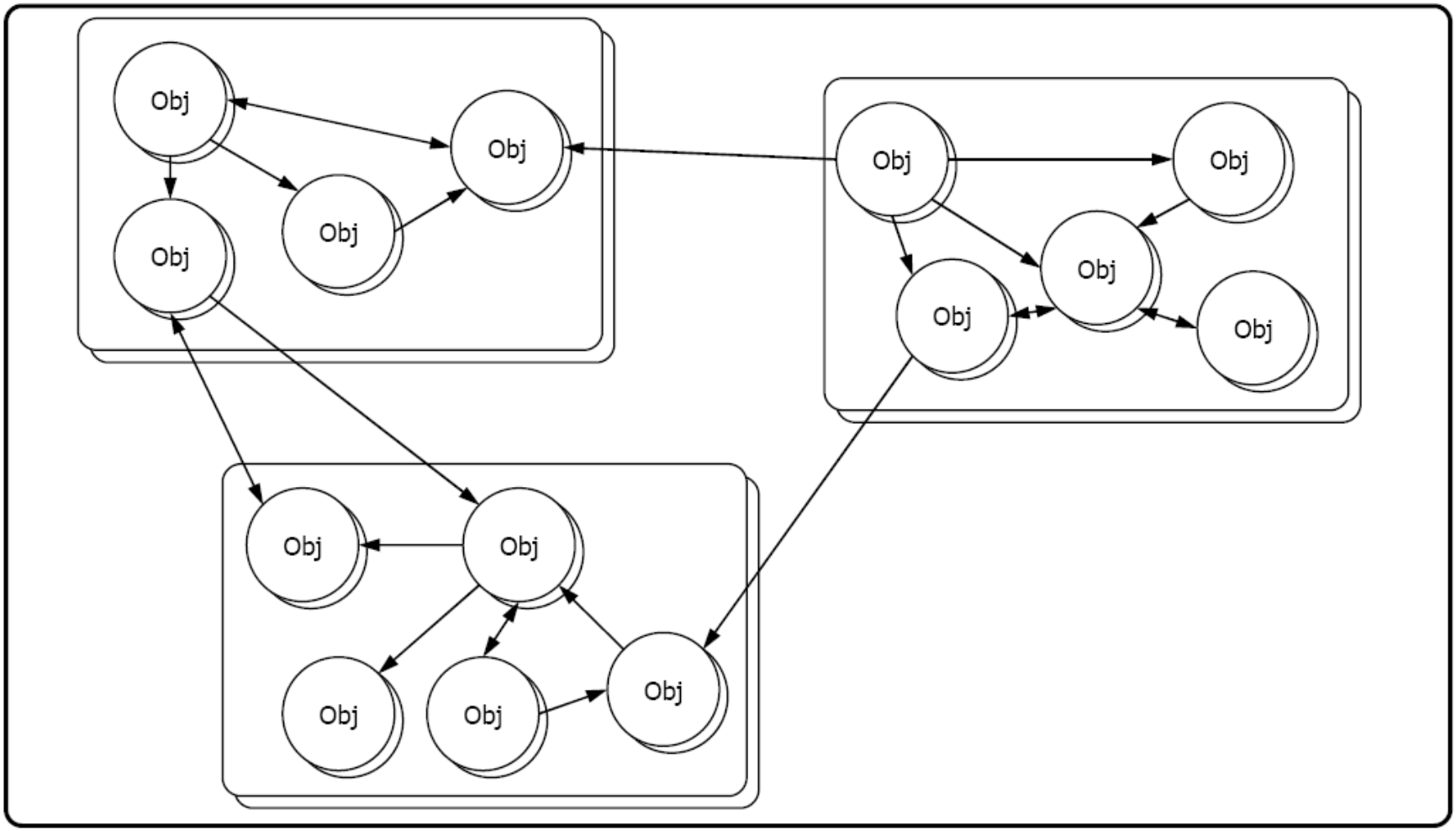
Adat = Adattag = Mező =
= Adatmező =Attribútum

Metódus = Tagfüggvény



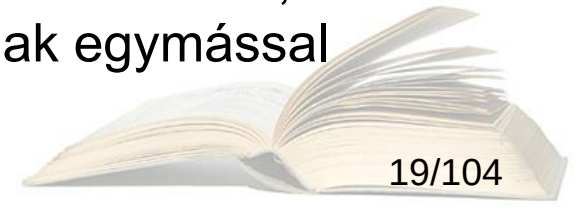


Objektumorientált program felépítése



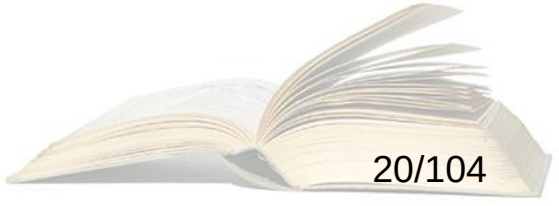
Objektumorientált program

- A problémát az **adatok** oldaláról közelíti meg
  - A szükséges absztrakt rendszerelemek meghatározása
  - A fenti rendszerelemek adatainak és (az adatokkal végezhető) absztrakt műveleteinek meghatározása, majd ezek összerendelése
    - Ezzel csoportokba („típusokba”) soroljuk az egyes elemeket
  - A probléma megoldását az egyes objektumok közötti kommunikáció, az egyes műveletek állapotváltozásoktól függő végrehajtása adja meg
    - Az objektumok kapcsolódási felülettel rendelkeznek, melynek segítségével üzeneteket váltanak egymással

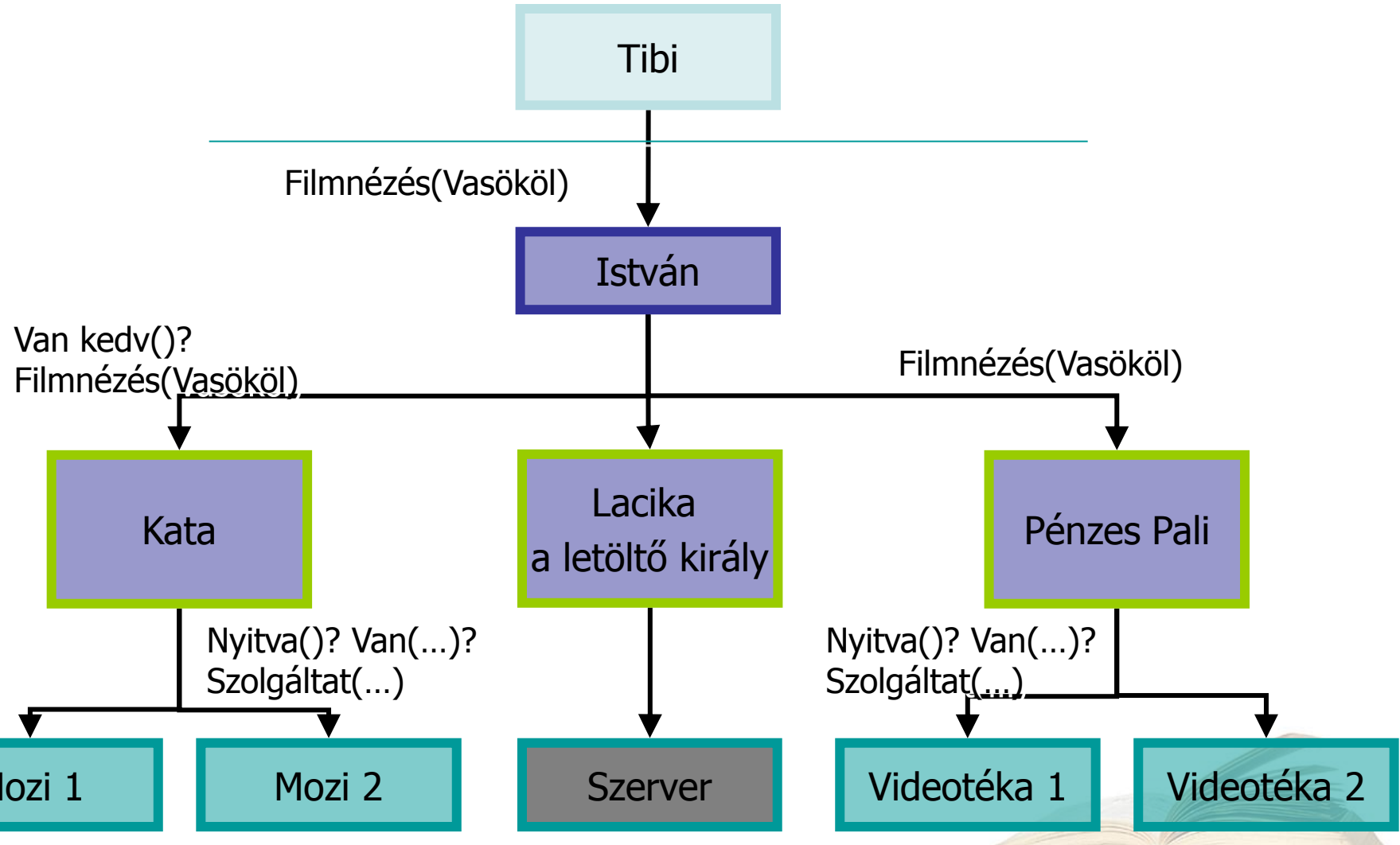


Objektumorientált program

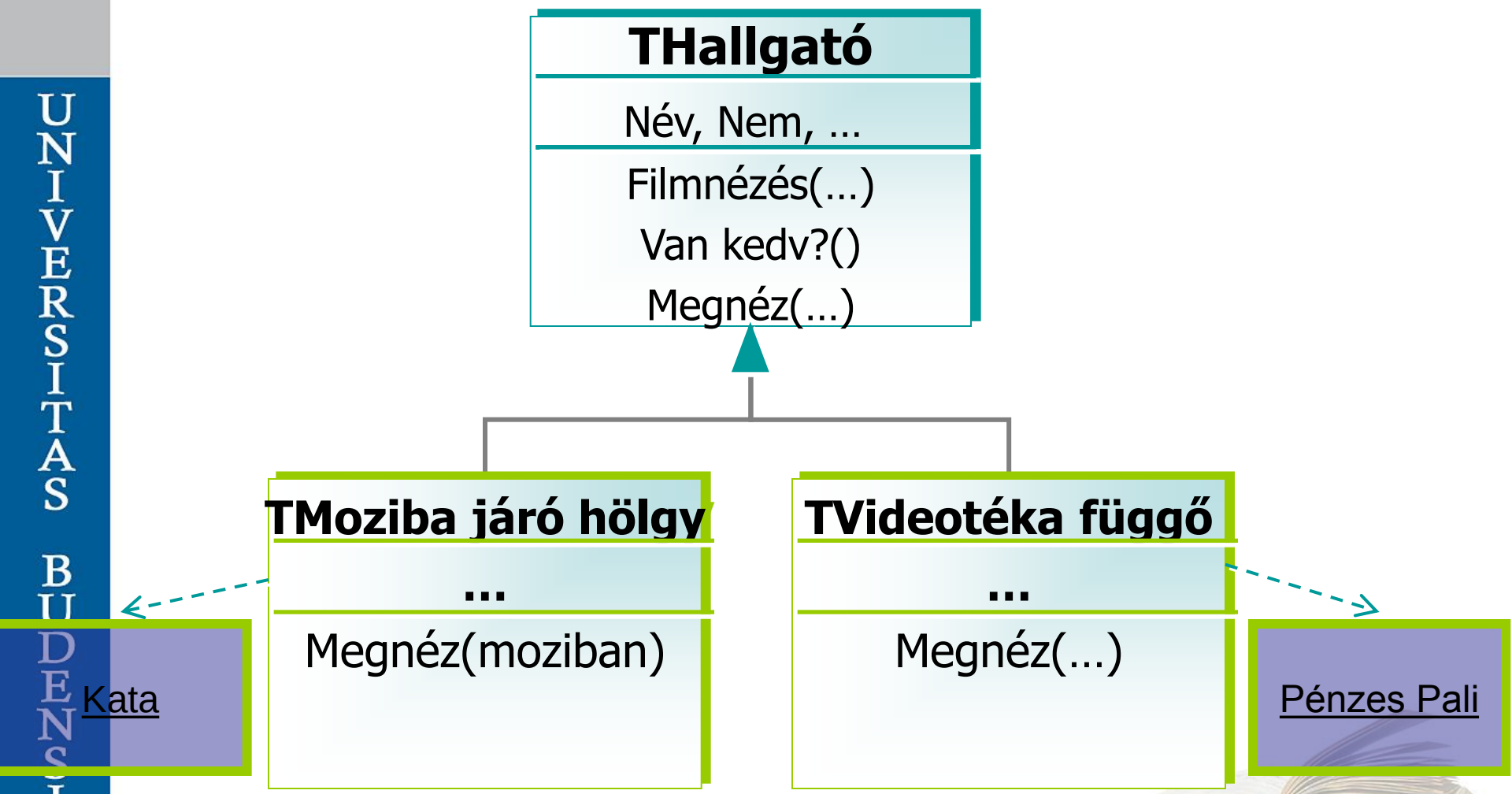
- Jellemzők:
  - Az egyes objektumok magukban foglalják az algoritmusokat
    - Minden objektum a probléma egy részét írja le és magában foglalja a részfeladat megoldásához tartozó algoritmikus elemeket
  - A főprogram jelentősége igen csekély
    - Gyakorlatilag csak indítási pontként szolgál, lényegi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



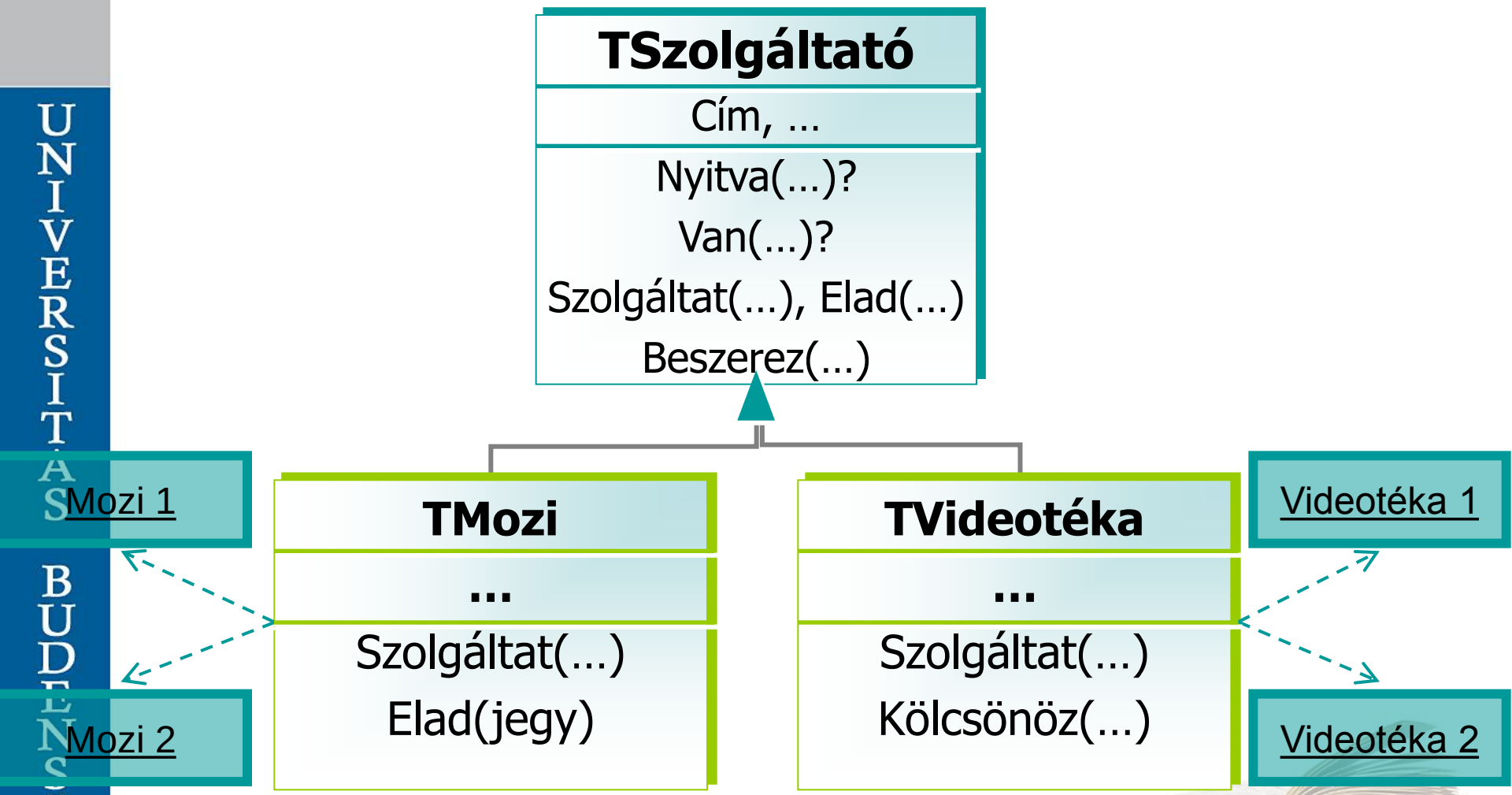
OO példa


Tibi
Filmnézés(Vasököl)
István
Van kedv()?
Filmnézés(Vasököl)
Filmnézés(Vasököl)
Kata
Lacika
a letöltő király
Pénzes Pali
Nyitva()? Van(...)?
Szolgáltat(...)
Nyitva()? Van(...)?
Szolgáltat(...)
Mozi 1
Mozi 2
Szerver
Videotéka 1
Videotéka 2

OO példa – osztályok, példányaik



OO példa – osztályok, példányaik



Az OO paradigma alapelvei

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
  - Üzenetváltások típusa
  - Pontosan definiált, megbízható kapcsolódási felületek
- ...mindezt a megvalósítás konkrét részleteinek ismerete nélkül

Az OO paradigma alapelvei

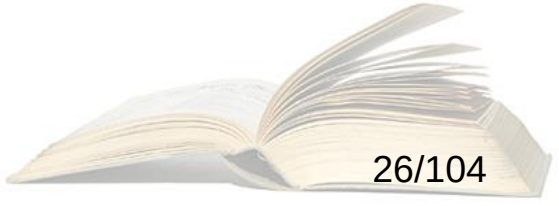
2. alapelv: **Egységbezárás**

- Az objektumok adatait és a rajtuk végezhető műveleteket szoros egységbe zárjuk
  - Az adatok csak a definiált műveletek segítségével érhetők el
  - Más műveletek nem végezhetők az objektumokon
  - Az objektum felelős a feladatai elvégzéséért, de a „hogyan” az objektum belügye
- Az egységbezárás védi az adatokat a téves módosításoktól

Az OO paradigma alapelvei

3. alapelv: **Adatrejtés**

- Az absztrakciók megvalósításának részleteit elrejtjük a „külvilág” elől
- Az objektumokon belül elkülönítjük a belső (privát) és a külső (nyilvános) adatokat és műveleteket
  - A privát adatok és műveletek a konkrét megvalósításhoz szükségesek
  - A nyilvános adatok és műveletek a szoftverrendszer többi objektuma számára (is) elérhetők
    - Tájékozódás az objektum állapotáról
    - Az objektum állapotának módosítása
    - Üzenetváltás



Az OO paradigma alapelvei

4. alapelv: **Öröklés**

- A már meglévő objektumtípusok alapján készíthetünk új típusokat, melyek rendelkeznek az „őstípus” tulajdonságaival
  - Ez egy specializációs művelet („származtatás”)
- A „leszármazottak” „öröklik” az őstípus tulajdonságait
  - A leszármazottak bővíthetik, esetenként akár szűkíthetik az őstípus állapotterét, illetve műveleteit
  - Teljes leszármazási hierarchiákat is létrehozhatunk
- Az alapelv következetes alkalmazásával elérhető, hogy a már megvalósított funkcionalitás később a megvalósítás részleteinek ismerete nélkül is felhasználható legyen
  - Jól átgondolt előzetes tervezést igényel

Az OO paradigma alapelvei

5. alapelv: **Többalakúság/polimorfizmus**

- A különböző, egymásból származó objektumtípusok azonos megnevezésű műveletei a konkrét objektumtól függően más-más konkrét megvalósítással rendelkezhetnek
  - Ugyanarra az üzenetre eltérő megvalósítású művelet lehet a válasz az őstípus és a leszármazott típus esetében
  - Az alapelv lehetőséget teremt rá, hogy azonos névvel hivatkozzunk az azonos célú, de a leszármazási hierarchia különböző szintjein más-más megvalósítást kívánó műveletekre
- Az egyes őstípusok leszármazottai mindenre alkalmasak, amire az adott őstípus alkalmas volt
  - Minden olyan helyzetben és funkcióban, ahol az őstípus szerepelhet, annak bármely leszármazottja is szerepelhet

Az OO paradigma alapelvei

5. alapelv: **Többalakúság/polimorfizmus**

Futás alatti kötés:

- Bizonyos műveletekről csak a futás során derül ki, hogy konkrétan melyikre van szükség. Ezen műveletek címei nem fordítási időben, hanem futási időben kötődnek a programhoz

Az OO paradigma alapelvei

6. alapelv: **Kódújrafelhasználás**

- A már megvalósított objektumtípusokat kész (bináris) formában más programokban is felhasználhatjuk
  - Jó tervezés és dokumentálás esetén az objektumok nyilvános adatai és műveletei elegendőek a későbbi felhasználáshoz
- Szintaktikai bővítésekkel (pl. „tulajdonságok”, „események”) kényelmesebbé tehető a külső felhasználás
- Az egyes objektumtípusokat egymásba ágyazva összetettebb típusokat hozhatunk létre
  - A kész, újrafelhasználható objektumtípusokat csoportokba fogva akár nagyobb „szoftver-építőelemeket” (komponenseket és komponensgyűjteményeket) is létrehozhatunk
- A korábban említett alapelvekre építve a kódújrafelhasználás lehetősége jelenti az igazi áttörést a szoftvertechnológiában

Példa OO jellegű megoldásra*

Feladat:

- Hány különböző elem van egy adott tömbben?
  - Feltételezzük, hogy a tömb pozitív egészeket tartalmaz

Terv:

- Input-output kezelésre osztály
- Halmaz tulajdonságot értékelő osztály:
  - Az eredeti tömb értékei, mint indexek alapján egy reprezentatív tömbben vizsgáljuk, hogy az elem már szerepel-e?
  - Inicializálás
  - Halmazméret visszaadás
- Vezérlő osztály: indítás, vezérlés

*: Gregorics Tibor (ELTE IK) mintapéldája alapján

Objektumorientált paradigma

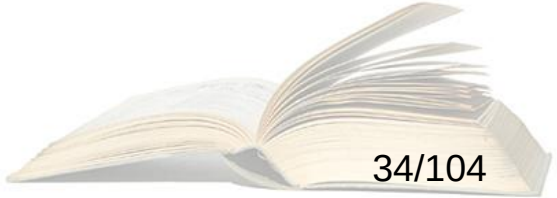
OSZTÁLY, OBJEKTUM, PÉLDÁNYOSÍTÁS

Objektum - definíció

- Az objektum állapottal rendelkező diszkrét entitás, amely 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

Objektum – állapotok, viselkedés, és azonosság

- 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 általa ismert metódusok (algoritmusok) 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*



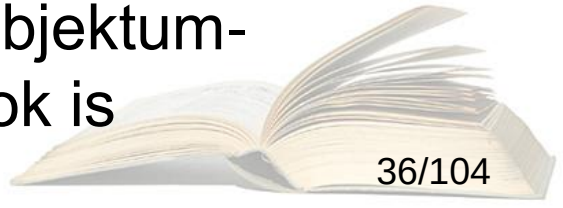
Osztály - definíció

- Az osztály egy adott objektumtípust határoz meg annak adataival (adatmezők) és beépített algoritmusaival (metódusok)
  - Az osztályok egyfajta mintát, sablont adnak az objektumokhoz
    - Az osztályok tehát azonos adatszerkezetű és viselkedésű objektumokat írnak le, azok absztrakciója lévén kerülnek definiálásra

Osztály

- Minden objektum valamilyen létező osztályba tartozik
  - Az egyes objektumok azon osztályok példányai, amelyekhez tartoznak
  - Egy osztályból több példány is létrehozható
  - Egy osztály összes példánya ugyanazokat a mezőket tartalmazza
    - Az egyes példányok létrehozásuk pillanatában azonos állapotúak, ezt követően viszont önállóan működnek tovább

Léteznek az osztályra (és nem az egyes objektum-példányokra) jellemző mezők és metódusok is

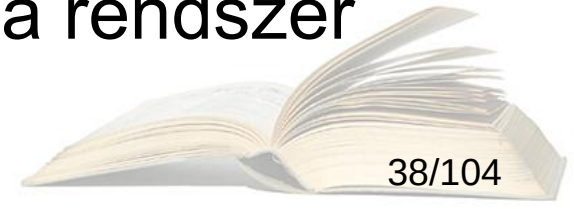


Osztályok vázlatos szerkezete

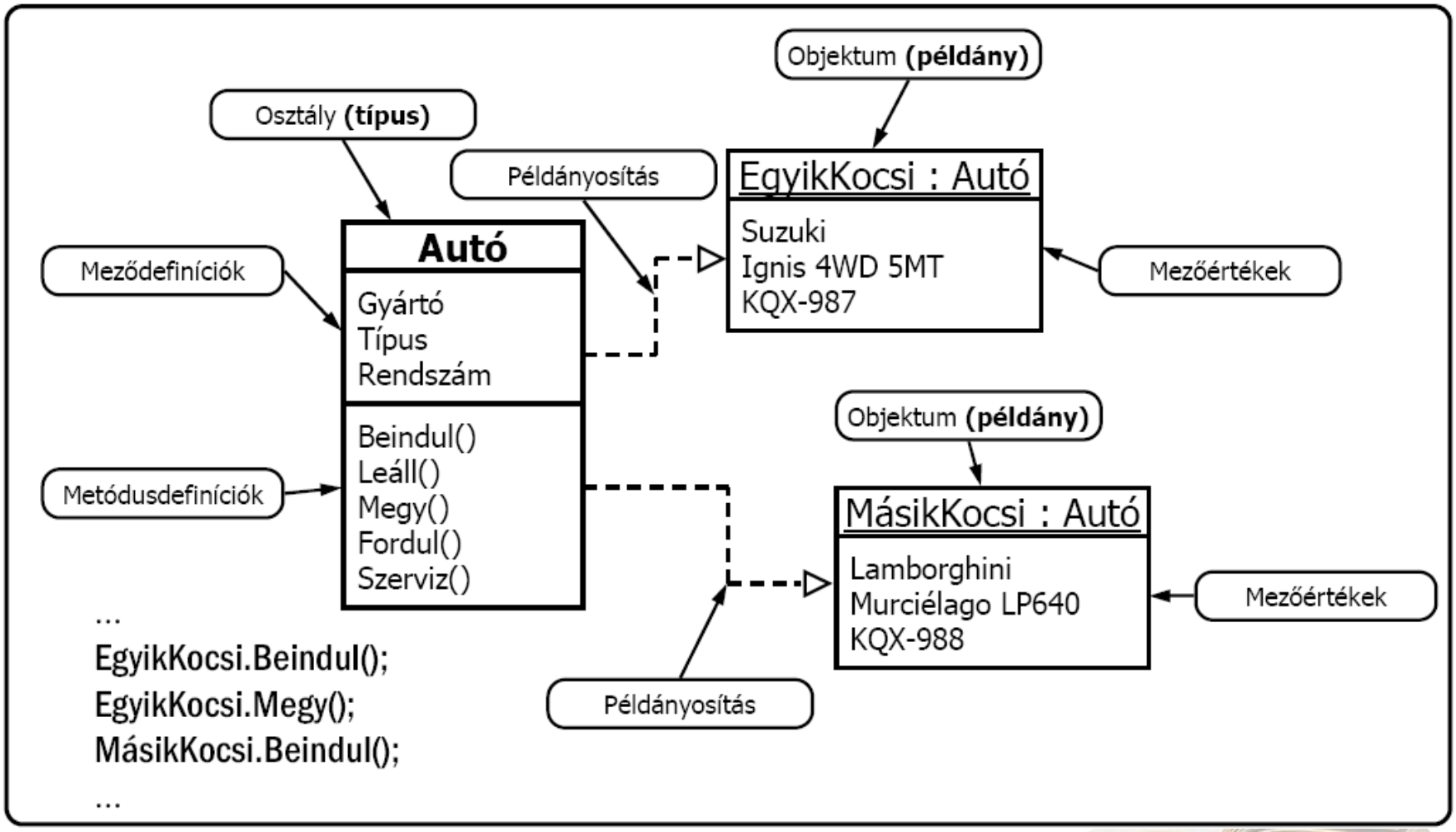
```
class Osztály1
{
        adattag1
        adattag2
        metódus1() { … } // részfeladatokat megoldó tagfüggvény
        metódus2() { … változók … utasítások … metódus hívások }
}
class Osztály2
{
        adattagok
        metódusok
        class Osztály3
        {
                adattagok
                metódusok
        }
}
```


Osztály és objektumpéldány

- Az *osztály* – összetett típus – egy *minta*, mely alapján objektum példányokat (objektumokat) hozhatunk létre: **példányosítás**
- Egy objektum keletkezésekor az osztálya által egyértelműen meghatározott a felépítése
- Minden mezőről egyedi másolat jön létre az objektumban
- Az osztály minden példányát ugyanazok a **metódusok** módosítják (**egyszer vannak a memóriában**), az összetartozást a rendszer felügyeli

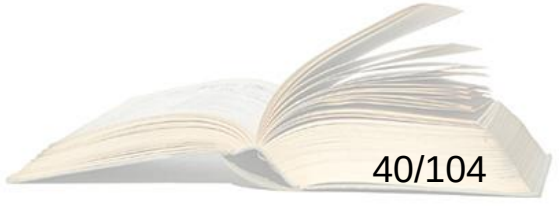


Osztály és példányai



Metódusok – üzenetküldés

- Egy objektum "felkérhet" más objektumokat különböző feladatok elvégzésére
- Az üzenet nem más, mint egy *kívülről elérhető metódus* hívása
- Az objektumnak lehetnek *belső metódusai* is, melyek kívülről nem elérhetők (lásd majd láthatóság témakörnél)



Metódusok általános típusai

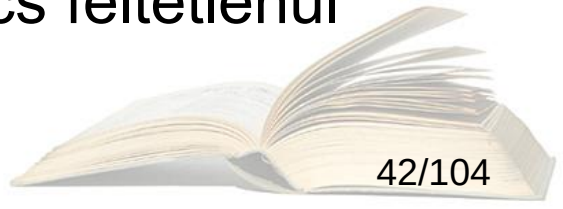
Objektum létrehozása: **konstruktor**

- Ahhoz, hogy az objektumokat használhassuk, először létre kell hozni őket. Ez a **példányosítás**
  - Alapja az osztály megadott definíciója
  - A példányosítást követően érhetők 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 általános típusai

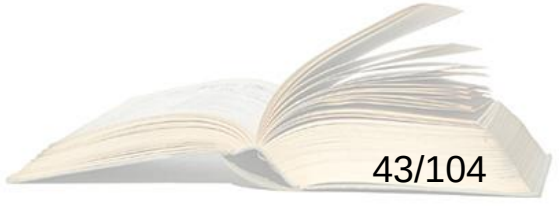
Objektumok megszüntetése: **destruktor**

- 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 destruktorra



Metódusok általános típusai

- Módosító metódusok (írók: *setter*)
  - Megváltoztatják az objektum állapotát
- Kiválasztó metódusok *(*kiolvasók*: 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 az adott részhalmazra vonatkozóan végeznek el műveleteket
- …



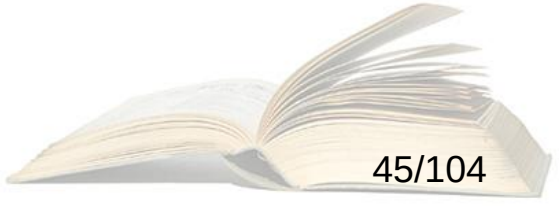
Tulajdonságok

- A tulajdonság olyan nyelvi elem, amely felhasználás szempontjából mezőként, megvalósítás szempontjából metódusként viselkedik
  - Az adott osztály felhasználói mezőnek „látják”
    - A külvilág mezőként hivatkozhat a tulajdonságra
  - A tulajdonságot megvalósító osztály külön-külön metódust rendelhet a tulajdonság olvasási és írási műveletéhez
    - Olvasáskor a megfelelő metódus egy valódi (általában privát) adatmező értékét is visszaadhatja, de akár számítással is előállíthatja a visszaadott értéket
    - Íráskor a megfelelő metódus egy valódi (általában privát) adatmező értékét is módosíthatja, de végezhet egyéb műveleteket is, vagy akár módosítás előtt ellenőrizheti az átadott értéket
- A tulajdonság tehát felfogható intelligens mezőként

Példány szintű tagok 1.

Objektumokhoz tartozó mezők és metódusok

- A példány szintű tagok az objektumpéldányok
  - saját adatmezői, valamint
  - saját adatain műveleteket végző metódusai
- A példány szintű mezők tárolják a példányok állapotát



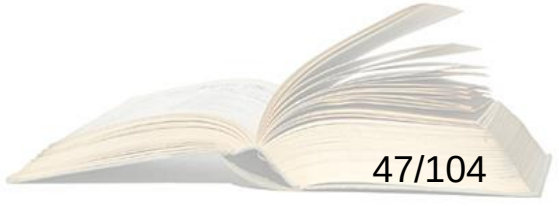
Példány szintű tagok 2.

- 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
  - Szükséges lehet, hogy a példány szintű metódusok hivatkozni tudjanak arra az objektumra, amelyen a műveletet végzik
    - Példa: átadás paraméterként vagy eredményként; elnevezések egyértelműsítése
  - Ezt általában egy rejtett paraméterrel valósítják meg
    - Megnevezése nyelvenként változik („this”, „Self”, „Me”...)
    - Mindig minden példánymetódusból elérhető, értéke a példány referenciája maga

Osztály szintű tagok 1.

Osztályokhoz tartozó mezők és metódusok

- Az osztály szintű tagok az egyes osztályokhoz tartozó
  - egyedi adatmezők,
  - valamint az ezeken műveleteket végző metódusok
- Osztály szintű adatmezők
  - Minden osztály pontosan egyet tartalmaz belőlük, függetlenül az osztályból létrehozott objektumpéldányok számától

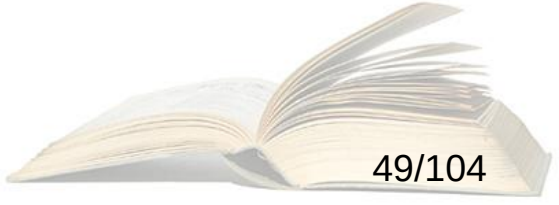


Osztály szintű tagok 2.

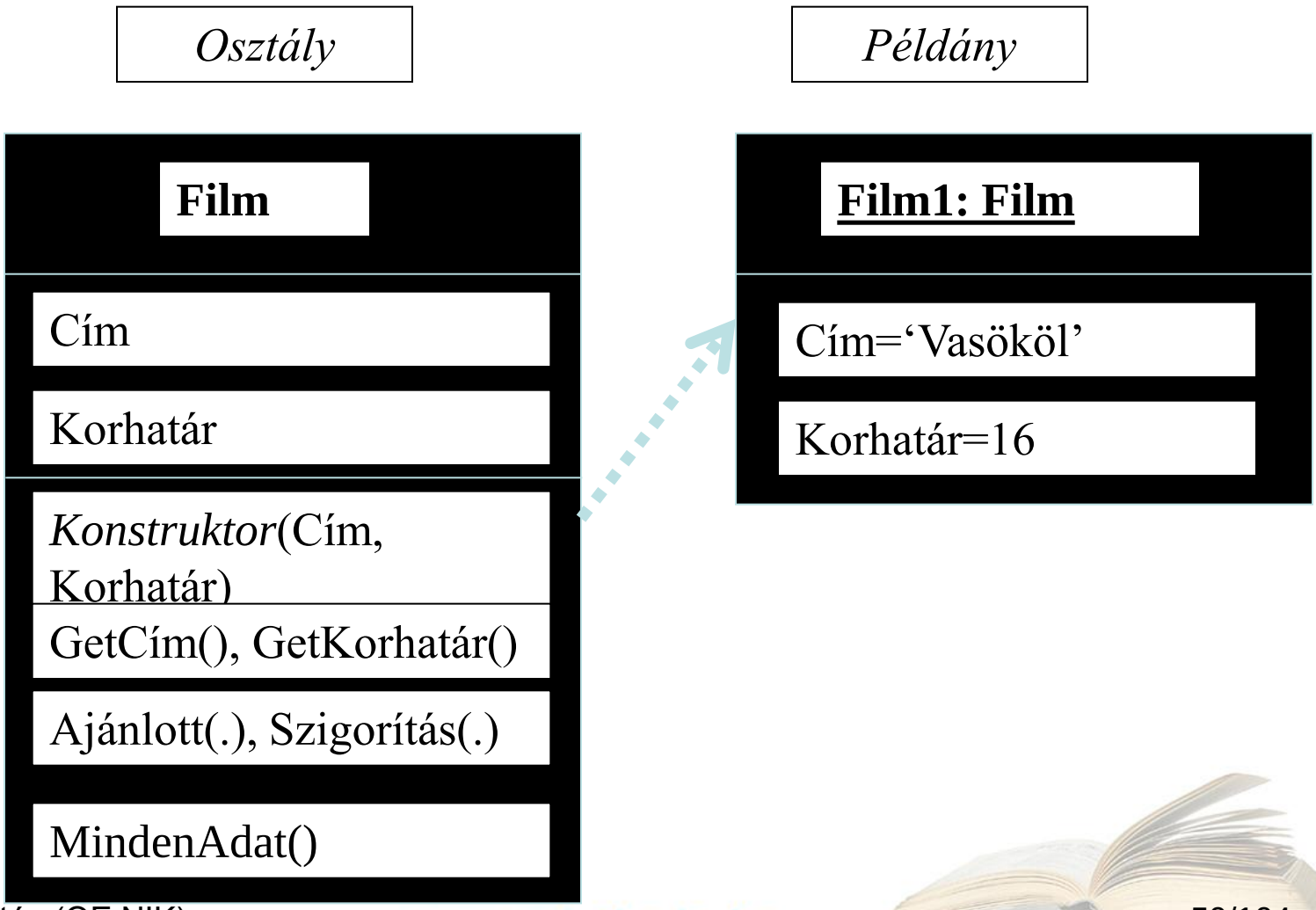
- Osztály szintű metódusok
  - Akkor is elérhetők, ha az osztályból egyetlen objektum sem lett példányosítva
  - Csak osztály szintű adatmezőket használhatnak
  - Speciális osztály szintű metódus az osztály szintű konstruktor
    - Feladata az osztály szintű adatmezők kezdőértékének beállítása
    - Általában az osztályra történő első hivatkozás előtt fut le automatikusan
  - Konkrét példányt nem igénylő feladatok végrehajtására alkalmasak
    - Példa: főprogram megvalósítása, io kezelés
    - Példa: konverziós függvényeknél ne kelljen példányt létrehozni

Osztály és példány szintű tagok

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



Osztály és objektumpéldány



Objektum létrehozása, -referencia

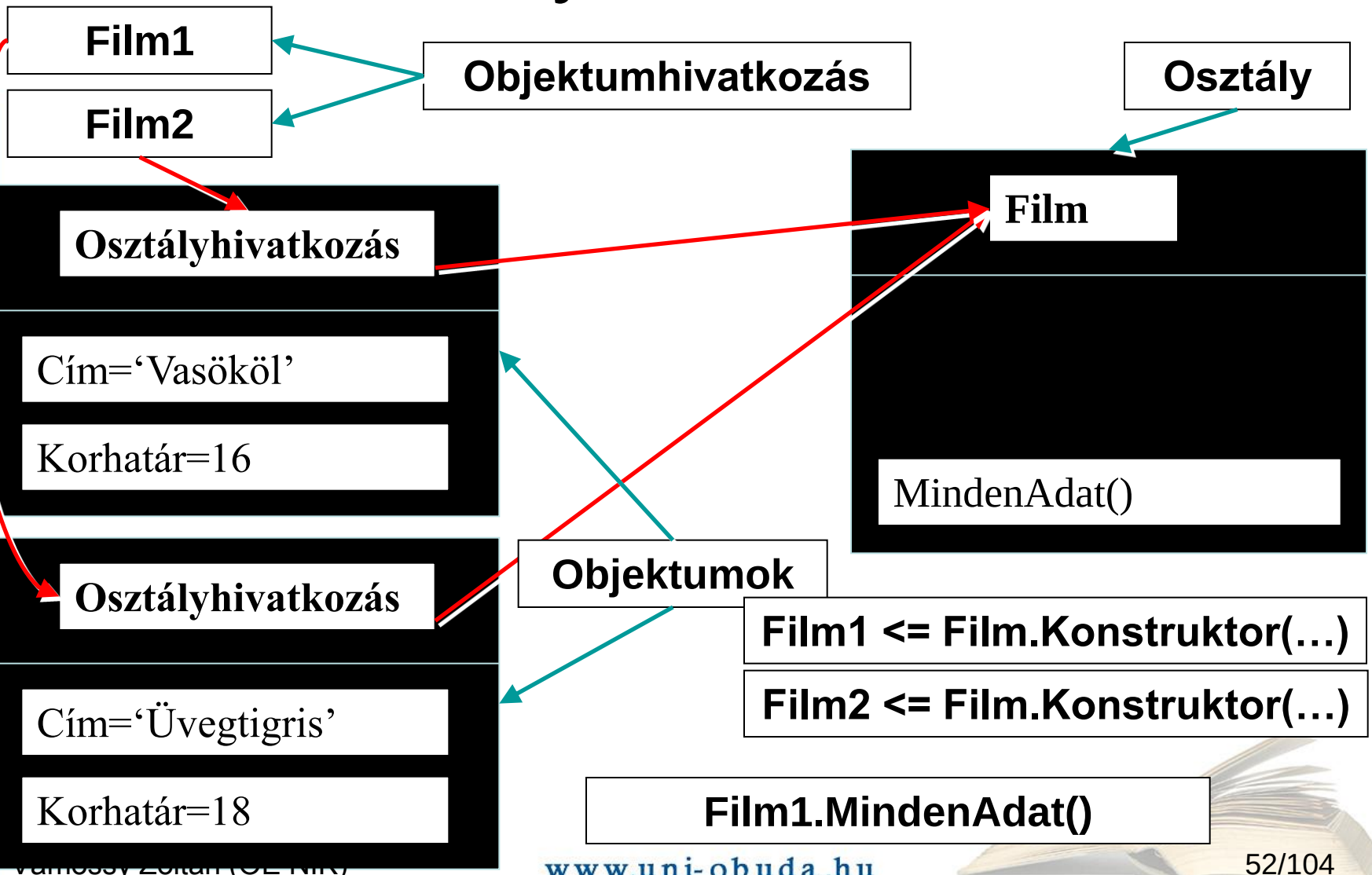
- Az objektumok *egyértelműen azonosíthatók*
  - a példány memóriabeli helye alapján
- Az osztály típusának megfelelő változó egy **referencia (mutató)**, amely az objektumra hivatkozik
- Több változó is hivatkozhat ugyanarra az objektumra

```
Film Film1;
…
Film1 <= Film.Konstruktor('Vasököl', 16)
```

Objektumhivatkozás
Ezzel nem jön létre az objektum!!!

Objektum létrehozás

Objektumhivatkozás, objektumok és osztály a memóriában



„Rejtett” paraméter 1.

- A rejtett paraméter egy referencia mutató, mely a megszólított objektum memóriacíme*
- Minden metódusnak ez egy rejtett paramétere és típusa maga az osztály
- Egy futó metódus innen tudja, hogy éppen melyik példányon dolgozik
- Egymásból hívott metódusok esetén automatikusan továbbadódik

*: ha osztály metódust hívunk, akkor az osztály címe

„Rejtett” paraméter 2.

Működési elv:

- Ha *Film1 Film* típusú, akkor a *Film1.GetCim()* üzenetet a fordító így értelmezi:
  Film.GetCim(Film1)
- Megjegyzés: Ezt az utolsó paramétert a fordító automatikusan beépíti az összes metódusba, tehát nem nekünk kell!!!

Objektumorientált paradigma

OBJEKTUMORIENTÁLTSÁG ALAPJAI C#-BAN 1.

Osztályok vázlatos szerkezete

```
class Osztály
{
        // mezők
        Típus1 adattag1;
        Típus 2 adattag2;
        …
        // metódusok
        Típus Metódus1( … ) { … } // részfeladatokat megoldó függvény
        void Metódus2() { … változók … utasítások … metódus hívások }
        Osztály (…) { … } // konstruktor
        …
        // tulajdonságok
        Típus Tulajdonság
        {
                get { … }
                set { … }
        }
}
```


Osztályok deklarálása

- Az osztályok deklarálása a **class** kulcsszó segítségével történik
  - Az osztályok deklarációja egyben tartalmazza az összes tag leírását és a metódusok megvalósítását
    - Az osztályoknál tehát nincs külön deklaráció (létrehozás) és definíció (kifejtés)

```
class Példaosztály
{
    // Itt kell deklarálnunk az osztály összes tagját (mezőket, metódusokat... )
    // A metódusok konkrét megvalósítását szintén itt kell megadnunk
}
```

- Osztályok és objektumok tagjainak elérése: „ . ” operátor
  - Példány szintű tagoknál a példány nevét, osztály szintű tagoknál (ezeket lásd később) az osztály nevét kell az operátor elé írnunk
    - Az osztály saját tagjainak elérésekor (tehát az osztály saját metódusainak belsejében) nem kell kiírni a példány, illetve az osztály nevét

Láthatósági szintek – előzetes

Nem teljes, lásd később!

Mezők, metódusok védelme

- A láthatósági szintek segítségével különítjük el az osztály belső, illetve kívülről is elérhető tagjait
  - Az egyes mezők és metódusok láthatósága külön-külön szabályozható
- Public: Minden metódus használhatja
  - A kívülről „látható” felület
- Private: Az osztály privát deklarációja, csak az osztály saját metódusai férhetnek hozzá
  - Kívülről nem látható

Adattagok

- Az adattagok értéke a deklarációval egyszerre, helyben is megadható (inicializálás)

```
string jegy = "jeles";
int j = -10;
```

- Az adattagok lehetnek
  - Olvasható-írható adattagok
    - Értékük tetszés szerint olvasható és módosítható az osztály függvényei által
    - Kívülről a láthatóság függvényében
  - Csak olvasható adattagok
    - Értékük futásidőben áll be, konstruktorban vagy helybeni inicializációval
    - Utána nem módosíthatók

```
readonly string SosemVáltozomMeg = "I Will Stay The Same";
```

  - Konstans adattagok
    - Értéküket a fordítóprogram előre tárolja, futáskor nem módosíthatók

```
const double π = 3.14159265;
const int összeg = 23 * (45 + 67);
```

Metódusok

- Minden metódus tagja egy osztálynak (tagfüggvény)
- A metódusok rendelkeznek:
  - Megadott, nulla vagy változó darabszámú paraméterrel (params kulcsszó). A paraméterátadás módja érték, vagy cím szerinti (out vagy ref)

```
void EgyparaméteresMetódus(bool feltétel) { ... }
void TöbbparaméteresMetódus(int a, float b, out string c) { ... }
void MindenbőlSokatElfogadóMetódus(params object[] paraméterTömb) { ... }
```

  - Visszatérési értékkel (ha nincs: void)

```
void NincsVisszatérésiÉrtékem() { ... }
int EgészSzámotAdokVissza(float paraméter) { ... }
string VálaszomEgyKaraktersorozat(string bemenőAdat) { ... }
SajátTípus Átalakító(SajátTípus forrásObjektum, int
   egyikAdattagÚjÉrtéke, string másikAdattagÚjÉrtéke) { ... }
```

- A paraméterek és a visszatérési értékek határozzák meg azt a protokollt, amelyet a metódus használatához be kell tartani
  - szignatúra

```
void TöbbparaméteresMetódus(int a, float b, string c);
```

Konstruktor I.

- Minden osztálynak rendelkeznie kell konstruktorral
  - Ez gondoskodik az osztály példányainak létrehozásáról
    - Szokás „példánykonstruktornak” is nevezni
  - A konstruktorok neve mindig megegyezik az osztály nevével
  - Nincs visszatérési értéke (void sem)
  - Több konstruktort is létrehozhatunk más-más paraméterlistával
    - Egy konstruktor a **this** kulcsszó segítségével meghívhat egy másik konstruktort is, (a **base** kulcsszó segítségével pedig az ősét)
  - Ha mi magunk nem deklarálunk konstruktort, akkor és csak akkor a C# fordító automatikusan létrehoz egy paraméter nélküli alapértelmezett konstruktort

Konstruktor II.

- Általában inicializációs feladatot végez
- Új objektum a **new** operátor segítségével hozható létre
  - A **new** operátor gondoskodik a megfelelő konstruktor hívásáról, memóriafoglalásról
    - Az osztályok konstruktorait kívülről nem kell és nem is lehet más módon meghívni

```
System.Object igaziŐskövület = new System.Object();

SajátTípus példány1 = new SajátTípus(25);
```

Destruktor I.

- Az objektumok megszüntetése automatikus (GC)
  - Akkor szűnik meg egy objektum, amikor már biztosan nincs rá szükség
    - Az objektumok megszüntetésének időpontja nem determinisztikus (nem kiszámítható)
  - A futtatókörnyezet gondoskodik a megfelelő destruktor hívásáról
    - Nem kell (és nem is lehet) közvetlenül meghívni az osztályok destruktorait
    - A destruktor nem tudhatja, pontosan mikor hívódik meg
  - Destruktorra ritkán van szükség

Destruktor II.

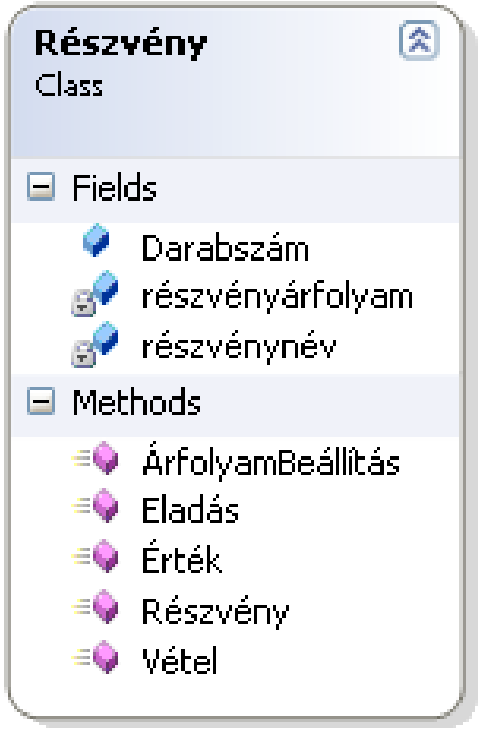
- Az osztályoknak nem kötelező destruktorral rendelkezniük
  - A destruktor neve egy „ ~ ” karakterből és az osztály nevéből áll

```
class SajátTípus
{
    // Destruktor
    ~SajátTípus()
    {
    }
}
```

Részvény példa

Készítsünk alkalmazást, amely részvényeink karbantartását végzi az alábbiak szerint:

- Minden részvényről tudjuk a nevét, az árfolyamát és a darabszámát
- Részvényt vehetünk, eladhatunk, változik az árfolyamuk
- Szeretnénk megtudni az aktuális értéküket



Részvény példa I.

```
class Részvény
{
    private readonly string részvénynév; // private és public jelentését majd később tárgyaljuk
    private double részvényárfolyam = 0.0;
    public int Darabszám; // A public itt ideiglenes. A helyes használatot lásd a tulajdonságoknál!

    public Részvény(string név, double árfolyam, int darabszám)
    {   // Konstruktor (neve megegyezik az osztály nevével) - beállítja az adatmezők kezdeti értékét
    }
    public void Vétel(int mennyiség)
    {   // A paraméterben megadott mennyiségű részvény vásárlása
    }
    public void Eladás(int mennyiség)
    {   // A paraméterben megadott mennyiségű részvény eladása
    }
    public void ÁrfolyamBeállítás(double árfolyam)
    {   // Az aktuális árfolyam beállítása a paraméterben megadott árfolyam alapján
    }
    public double Érték()
    {   // Részvény összértékének kiszámítása
    }
}
```

Részvény példa II.

```
class Részvénykezelő
{
    static void Main()
    {
        Részvény IBM = new Részvény("IBM", 77.59, 100);
        Részvény nVidia = new Részvény("nVidia", 21.49, 100);

        IBM.Vétel(50);
        nVidia.Vétel(25);
        nVidia.ÁrfolyamBeállítás(29.15);
        nVidia.Eladás(50);

        System.Console.WriteLine("IBM: " + IBM.Darabszám + " db ($" + IBM.Érték() + ")");
        System.Console.WriteLine("nVidia: " + nVidia.Darabszám + " db ($" + nVidia.Érték() + ")");
        System.Console.ReadLine();
    }
}
```

Metódusok túlterhelése*

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

*: overloading (néhol átdefiniálás)

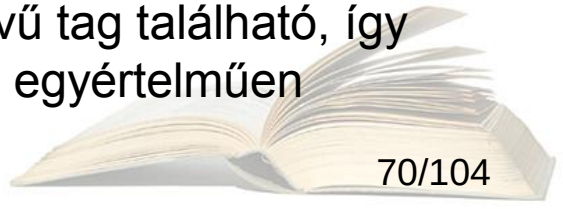
Részvény példa III.

Metódus túlterhelése:

```
class Részvény
{
    ...
    public void Vétel(int mennyiség)
    {   // A paraméterben megadott mennyiségű részvény vásárlása
        Darabszám += mennyiség;
    }
    public void Vétel(int mennyiség, double árfolyam)
    {   // A megadott mennyiségű részvény vásárlása a megadott árfolyam beállításával
        Darabszám += mennyiség;
        részvényárfolyam = árfolyam;
    }
}
```

A this referencia paraméter

- A példány szintű metódusokban szükség lehet rá, hogy hivatkozni tudjunk arra az objektumra, amelyik a metódust éppen végrehajtja
- E hivatkozás a rejtett **this** paraméter segítségével valósul meg
  - A **this** minden példány szintű metódusban az aktuális objektumra referencia
  - Nem kell deklarálni, ezt a fordítóprogram automatikusan megteszi
  - Általában a következő esetekben használatos:
    - Az aktuális objektumot paraméterként vagy eredményként szeretnénk átadni
    - Az érvényes hatókörön belül több azonos nevű tag található, így ezek a tagok csak segítséggel azonosíthatók egyértelműen



Részvény példa IV.

Milyen nehézség adódott volna, ha a példában az alábbi mezőneveket és paramétereket használjuk? A helyi változók elrejtik az osztály saját, azonos nevű adattagjait

```
class Részvény
{
    private string név;
    private double árfolyam;
    public int Darabszám;

    public Részvény(string név, double árfolyam, int Darabszám)
    {
        név = név;
        árfolyam = árfolyam;
        Darabszám = Darabszám;
    }

    public void ÁrfolyamBeállítás(double árfolyam)
    {
        árfolyam = árfolyam;
    }
}
```

Részvény példa IV.

- Megoldás this paraméter segítségével:

```
class Részvény
{
    private string név;
    private double árfolyam;
    public int Darabszám;

    public Részvény(string név, double árfolyam, int Darabszám)
    {
        this.név = név;
        this.árfolyam = árfolyam;
        this.Darabszám = Darabszám;
    }

    public void ÁrfolyamBeállítás(double árfolyam)
    {
         this.árfolyam = árfolyam;
    }
}
```

Tulajdonságok

- A tulajdonságok olyan „intelligens mezők”, amelyek olvasását és írását a programozó által megadott ún. hozzáférési metódusok végzik
  - Az osztályt felhasználó kód számára mezőnek tűnnek
  - Az osztály fejlesztője számára metóduspárként jelennek meg
- A hozzáférési metódusok bármilyen műveletet végrehajthatnak (pl. beírandó adat ellenőrzése, kiolvasandó adat kiszámítása…)
  - Nem célszerű hosszan tartó műveletekkel lelassítani a tulajdonság elérését
  - Kerülni kell a felhasználó kód számára váratlan mellékhatásokat

Tulajdonságok

- Létrehozhatók csak olvasható vagy csak írható tulajdonságok is
  - Ha nem adjuk meg az írási, illetve olvasási metódust, akkor csak olvasható, illetve csak írható tulajdonság jön létre
- Az olvasási és írási metódushoz hozzáférési szint (lásd később) adható meg
  - Ezzel a módszerrel például létrehozhatók nyilvánosan olvasható, de csak a saját vagy a leszármazott osztályok által írható tulajdonságok
- Az olvasási metódust a **get**, az írási metódust a **set** kulcsszó segítségével kell megadnunk
  - A hozzáférési metódusokat a tulajdonság definícióján belül kell megadnunk

Tulajdonságok (példa)

```
using System;

class Személy
{
    private string vezetéknév = "<üres>";
    private string keresztnév  = "<üres>";

    public string Vezetéknév
    {
        get     { return vezetéknév; }
        set     { if (value != " ") vezetéknév = value; }
    }
    public string Keresztnév
    {
        get     { return keresztnév; }
        set     { if (value != " ") keresztnév = value; }
    }
    public string Név
    {
        get     { return vezetéknév + " "+ keresztnév; }
    }
}
```

Gyakori megoldás, hogy a nyilvános tulajdonság neve nagybetűs, a hozzá tartozó privát mező neve pedig ugyanaz kisbetűs változatban

A tulajdonság értékét a return kulcsszóval adjuk vissza a hívónak

A tulajdonság megadott új értékét mindig egy value nevű rejtett paraméter tartalmazza

Nem adtunk meg set metódust, így ez a tulajdonság csak olvasható

Tulajdonságok (példa)

```
class Személykezelő
{
    static void Main()
    {
        Személy Pistike = new Személy();
        Console.WriteLine("Üres nevem: " + Pistike.Név);

        Pistike.Keresztnév = "István";
        Console.WriteLine("Félkész nevem: " + Pistike.Név);

        Pistike.Vezetéknév = "Kovács";
        Pistike.Keresztnév = " ";
        Console.WriteLine("Nevem: " + Pistike.Név);
        Console.ReadLine();
    }
}
```

Részvény példa V.

- Tulajdonsághasználattal adatrejtés:

```
class Részvény
{
    private string név;
    private double árfolyam;
    private int darabszám;

    public int Darabszám
     {
        get { return darabszám; }
        set { darabszám = value; }
     }
}
```

Osztály szintű tagok

- Az osztály szintű mezők az osztály saját adatmezői
  - Minden osztály csak egyet tárol ezekből a mezőkből, függetlenül a később létrehozott példányok számától
    - Ezeket a mezőket tehát nem a példányok, hanem maga az osztály birtokolja
- Az osztály szintű metódusok magán az osztályon működnek
  - Akkor is hívhatók, ha még egyetlen példány sem létezik az osztályból
    - Csak osztály szintű mezőket használhatnak
    - Osztály szintű metódusoknál nem létezik aktuális objektum, így this paraméter sem
    - Konkrét példányt nem igénylő feladatra is alkalmasak (pl. főprogram megvalósítása)

Osztály szintű tagok - példa

```
using System;
class Példányszámláló
{
    private static int darabszám = 0;
    public static int Darabszám;
    {
        get { return darabszám ; }
    }
    static Példányszámláló()
    {
        darabszám = 0;
    }
    public Példányszámláló()
    {
        darabszám++;
    }
    ~Példányszámláló()
    {
        darabszám--;
        Console.WriteLine("Megszűnt egy példány. A fennmaradók száma: " + darabszám);
    }
}
```

Osztály szintű adatmező

Osztály szintű konstruktor
(egyik fő célja az osztály szintű mezők kezdeti értékének beállítása)

Konstruktor

Destruktor

Osztály szintű tagok - példa

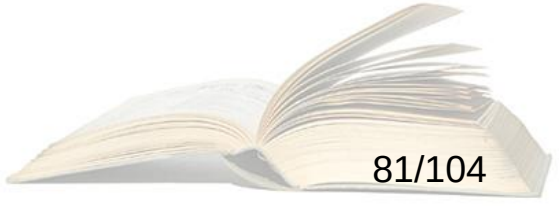
```
class PéldányszámlálóTeszt
{
    static void Main()
    {
        Példányszámláló teszt = new Példányszámláló();
        Console.WriteLine("Létrehoztam egy példányt");
        Console.WriteLine("Példányszám: " + Példányszámláló.Darabszám);

        for (int i = 0; i < 10; i++)
                new Példányszámláló();
        Console.WriteLine("Létrehoztam még tíz példányt");
        Console.WriteLine("Példányszám: " + Példányszámláló.Darabszám);

        Console.ReadLine();
    }
}
```

Objektumorientált paradigma

KÓDÚJRAFELHASZNÁLÁS



Kódújrafelhasználás

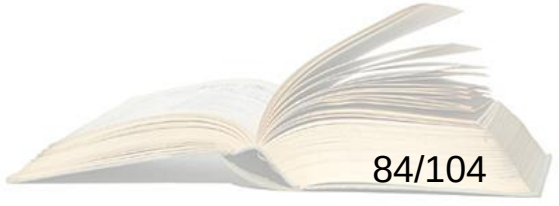
- A már megvalósított objektumtípusokat kész (bináris) formában más programokban is felhasználhatjuk
  - Jó tervezés és dokumentálás esetén az objektumok nyilvános adatai és műveletei elegendőek a későbbi felhasználáshoz
- Szintaktikai bővítésekkel (pl. „tulajdonságok”) kényelmesebbé tehető a külső felhasználás
- Az egyes objektumtípusokat egymásba ágyazva összetettebb típusokat hozhatunk létre
  - A kész, újrafelhasználható objektumtípusokat csoportokba fogva akár nagyobb „szoftver-építőelemeket” (komponenseket és komponensgyűjteményeket) is létrehozhatunk
- A korábban említett alapelvekre építve a kódújrafelhasználás lehetősége jelenti az igazi áttörést a szoftvertechnológiában

Komponensek

- A komponensek olyan osztályok vagy osztálygyűjtemények, amelyek készen (bináris formában), változtatás nélkül felhasználhatók és összekapcsolhatók
  - Önálló logikai (és általában fizikai) egységet képeznek
  - Felhasználásukhoz általában nincs szükség arra, hogy forráskódjuk rendelkezésre álljon
  - Az őket felhasználó osztályokkal metódusok, tulajdonságok útján érintkeznek
  - Különböző nyelveken írt komponensek is együttműködhetnek

Névterek 1.

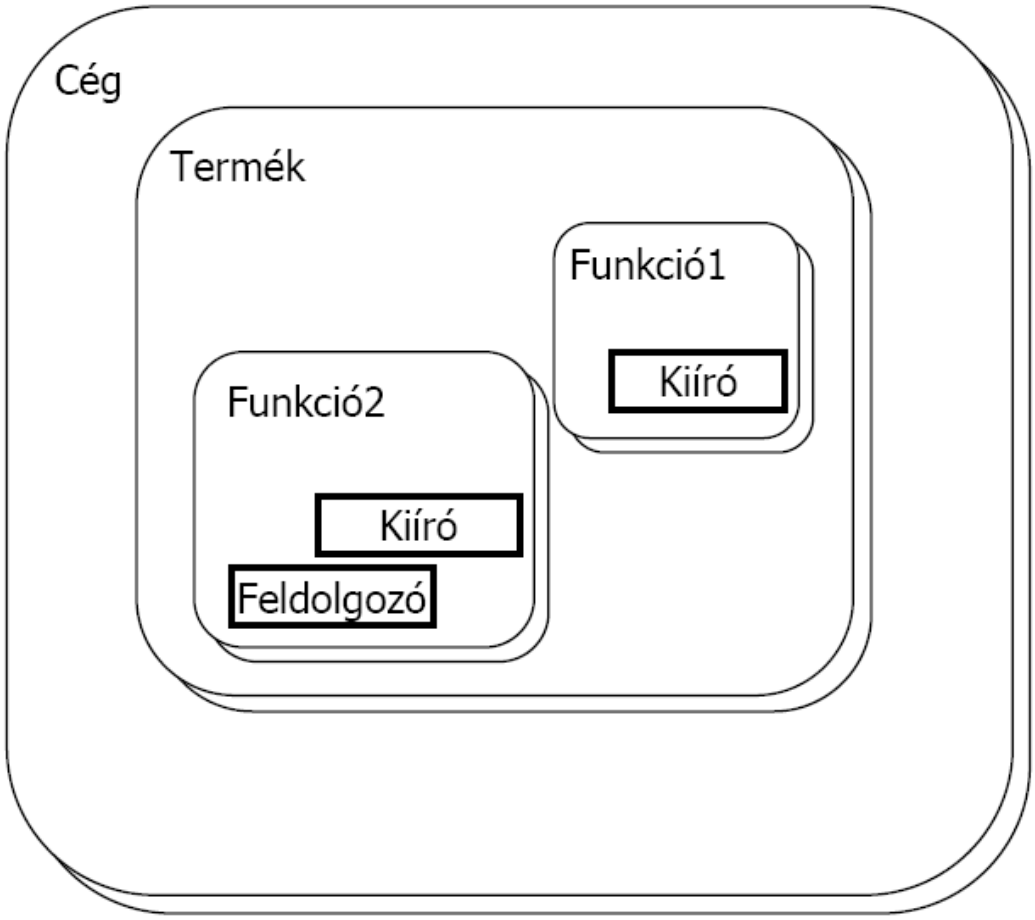
- Az OO paradigma jellemzője az elnevezések óriási száma
  - Minden osztálynak, objektumnak, mezőnek, metódusnak egyedi nevet kell adni, hogy a későbbiekben hivatkozni lehessen rájuk
    - Nem könnyű jól megjegyezhető, a célt később is felidéző neveket adni
- A programok méretével egyenes arányban nő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

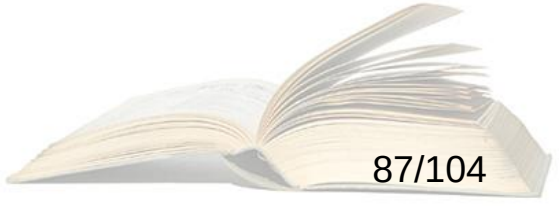
Cég/Termék/Funkció1/Kiíró
Cég/Termék/Funkció2/Kiíró
Cég/Termék/Funkció2/Feldolgozó

„Névtérpótlék" technika:

Cég_Termék_Funkció1_Kiíró
Cég_Termék_Funkció2_Kiíró
Cég_Termék_Funkció2_Feldolgozó

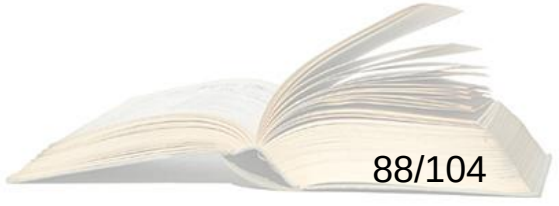
Objektumorientált paradigma

EGYSÉGBEZÁRÁS ÉS ADATREJTÉS



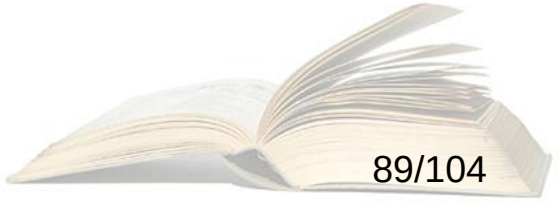
Egységbezárás

- Az objektumok adatait és a rajtuk végezhető műveleteket szoros egységbe zárjuk
  - Az adatok csak a definiált műveletek segítségével érhetők el
  - Más műveletek nem végezhetők az objektumokon
  - Az objektum felelős a feladatai elvégzéséért, de a „hogyan” az objektum belügye
- Az egységbezárás védi az adatokat a téves módosításoktól



Adatrejtés

- Az absztrakciók megvalósításának részleteit elrejtjük a „külvilág” elől
  - A programozási hibák elleni védelem céljából
- Az objektumokon belül elkülönítjük a belső (privát) és a külső (nyilvános) adatokat és műveleteket
  - A privát adatok és műveletek a konkrét megvalósításhoz szükségesek
  - A nyilvános adatok és műveletek a szoftverrendszer többi objektuma számára (is) elérhetők
    - Tájékozódás az objektum állapotáról
    - Az objektum állapotának módosítása
    - Üzenetváltás



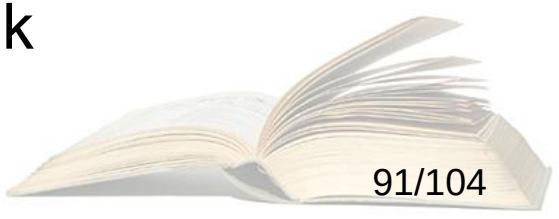
Láthatósági szintek

Mezők, metódusok védelme

- A láthatósági szintek segítségével különítjük el az osztály belső, illetve kívülről is elérhető tagjait
  - Az egyes mezők és metódusok láthatósága külön-külön szabályozható
- Public: Minden metódus használhatja akár objektumból, akár öröklésen keresztül
  - A kívülről „látható” felület
- Protected: Az osztály saját metódusai férhetnek hozzá + hozzáférés öröklésen keresztül
- Private: Az osztály privát deklarációja, csak az osztály saját metódusai férhetnek hozzá

Láthatósági szintek

- A láthatósági szintek segítségével hatékonyan valósítható meg az egységbezárás
  - Az osztályok a kívülről is látható elemeiket bocsátják más osztályok rendelkezésére
    - A nyilvános mezők adatokat, a nyilvános metódusok műveleteket tesznek elérhetővé
  - Az egyes osztályok megvalósítási részletei módosíthatók anélkül, hogy az osztályt használó más osztályoknak erről tudniuk kellene
    - A megvalósítást végző algoritmusok nincsenek kihatással az osztályt használó kódra
  - Konkrét OO nyelvi megvalósításokban általában további láthatósági szintek is léteznek



Objektumorientált paradigma

ÖRÖKLŐDÉS ÉS POLIMORFIZMUS

Öröklődés

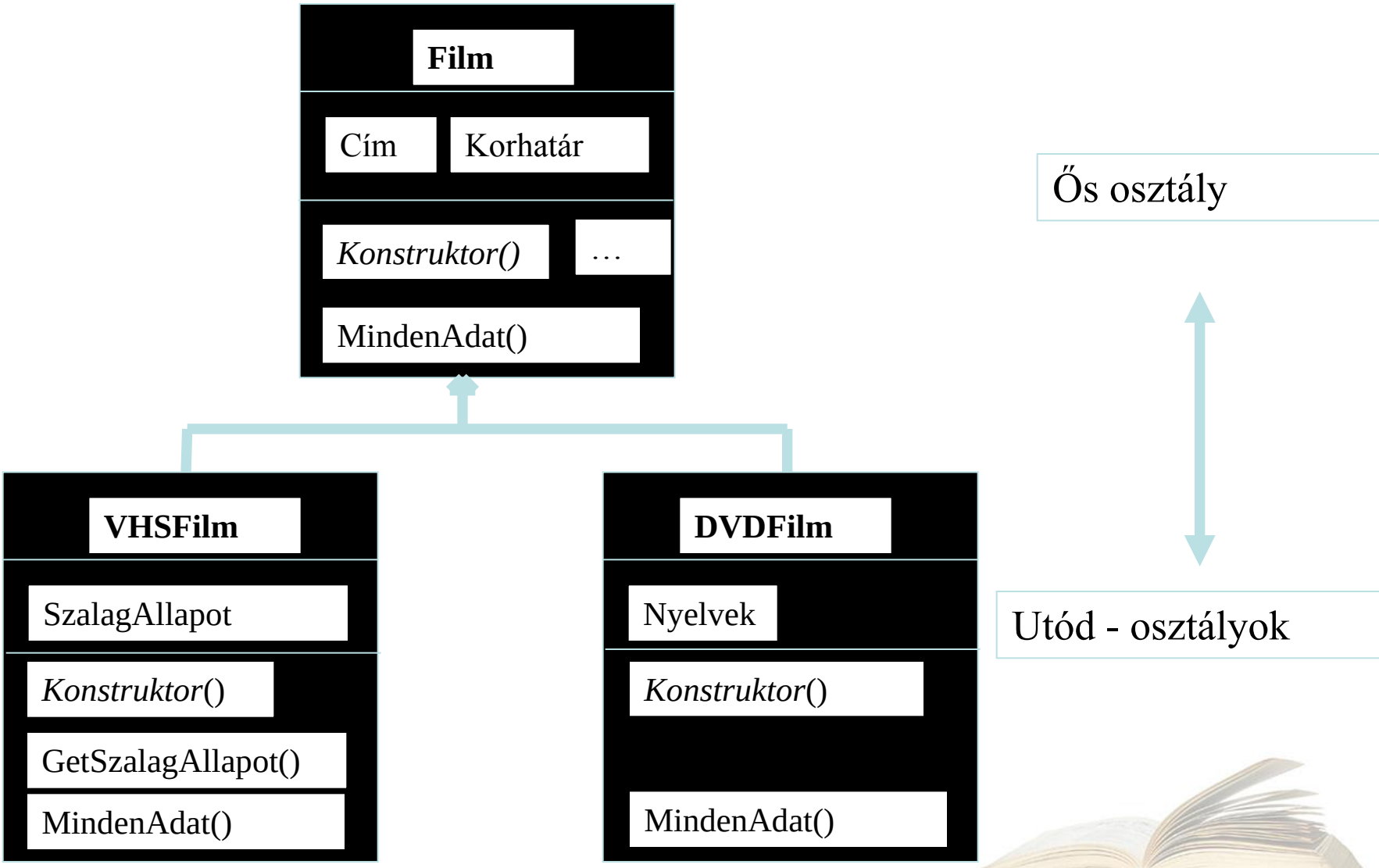
Általánosítás

Specializálódás

Öröklődés 1.

- Egy már meglévő osztály továbbfejlesztése
- A már meglévő osztály az *ős osztály*, a továbbfejlesztett osztály pedig a leszármazott, származtatott, illetve *utód osztály*. Az utód osztály az ős osztály specializálása
- *Egyszeres öröklésről* akkor beszélünk, ha egy osztálynak csak egy őse lehet
- *Alaposztály*: A hierarchia legfelső osztálya

Öröklődés példa 1.



Öröklődés 2.

Egy osztály örökítésekor három lehetőségünk van:

- *Új mező* felvétele (pl. *Nyelvek*)
- *Új metódusok* felvétele (pl. *GetSzalagAllapot()*)
- Meglévő *metódusok felülírása* (pl. *MindenAdat()*).

Megjegyzés. Új adatok megadása új metódusok nélkül általában értelmetlen, hiszen adatot csak metódussal szabad elérni (információ elrejtése), és az ős metódusok nyilvánvalóan nem ismerhetik az újonnan deklarált adatokat

Öröklődés 3.

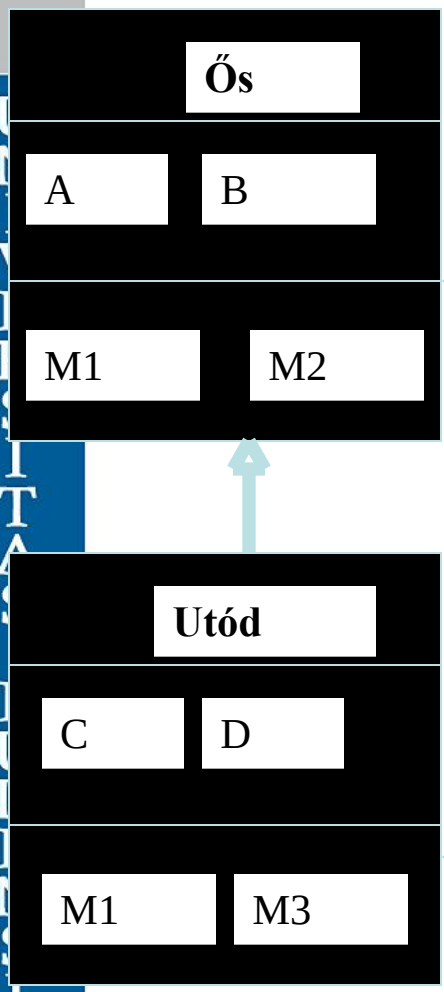
Adatok öröklése – a láthatóság beállítása szerint:

- Az utód osztály példányainak adatai = ős adatok + saját adatok. Adatokat nem lehet felüldefiniálni!

Metódusok öröklése:

- Bármely metódust felül lehet írni. Ugyanolyan nevű metódus mást csinálhat az utódban
- Az utód osztály metódusaiban használható bármely ős metódus

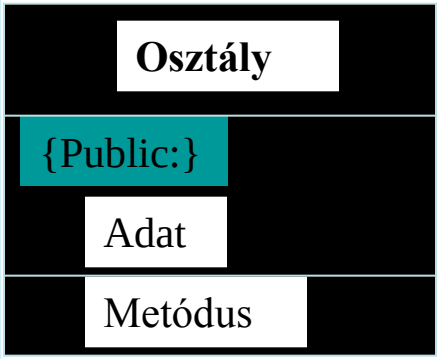
Öröklődés 4.



	Küldhető üzenetek	Mi hajtódik végre?
O1:Ős	O1.M1	Ős.M1
A, B	O1. M2	Ős.M2
O2:Utód	O2. M1	Utód.M1
A, B, C, D	O2. M2	Ős.M2
	O2. M3	Utód.M3

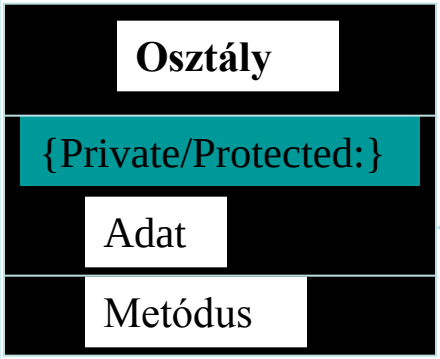
Adatok és metódusok védelme 1.

Objektumpéldány védelme: *private/protected public*



Objektum: Osztály

Objektum.Metódus

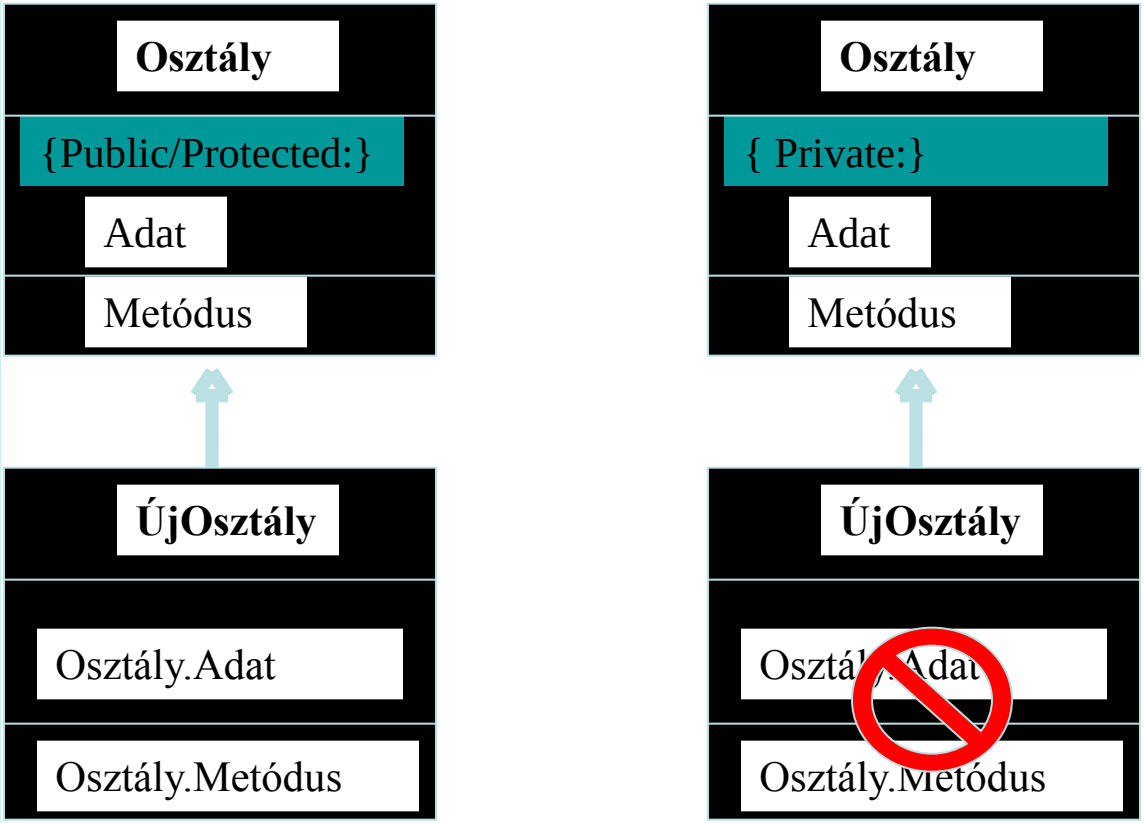


Objektum: Osztály

Objektum.Metódus

Adatok és metódusok védelme 2.

Osztály védelme örökítéskor: *private*
public /protected

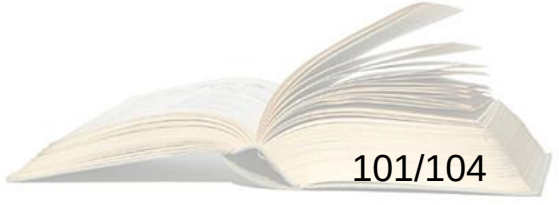


Adatok és metódusok védelme 3.

Összefoglalás:

- Public: Minden metódus használhatja akár objektumból, akár öröklésen keresztül.
- Protected: Az osztály saját metódusai férhetnek hozzá + hozzáférés öröklésen keresztül.
- Private: Az osztály privát deklarációja, csak az osztály saját metódusai férhetnek hozzá.

- A láthatóságot csak bővíteni lehet az öröklődés során!



Polimorfizmus

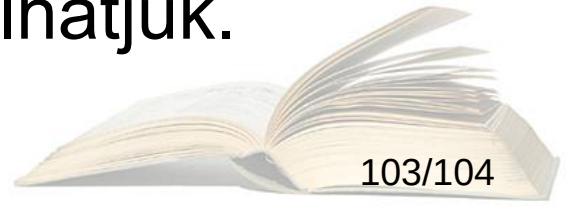
- A polimorfizmus (többalakúság) azt jelenti, hogy *ugyanarra az üzenetre különböző objektumok különbözőképpen reagálhatnak:* minden objektum a saját - (az üzenetnek megfelelő) metódusával
- Módszer polimorfizmus: Egy leszármazott osztály egy örökölt módszert újraimplementálhat (*MindenAdat()* a példában).
- Objektum polimorfizmus: Minden egyes objektum szerepelhet minden olyan szituációban, ahol az ősosztály objektuma szerepelhet, nem csak a saját osztálya példányaként használható =>

Többalakúság

A leszármazott osztály mindig használható az őstípusaként:

- változó értékadásnál
- paraméter átadásnál

Pl.: a programunkban szeretnénk VHS és DVD filmeket is tárolni, akkor nem kell két külön tömböt felvennünk (és mindenhol megduplázni a műveleteket), hanem elég egy ősosztály típusú tömb és ebbe mindkét leszármazott osztály példányait elhelyezhetjük, használhatjuk.



Köszönöm a figyelmet!