Objektumorientált programozás

Az OO paradigma

- **Mitől OO egy program?**
- **Objektum**
- **Osztály**
- **Öröklődés**

A valós világ modellezése

- Az ember a világ megértéséhez modelleket épít
- **Modellezési alapelvek**
  - Absztrakció
    - az a szemléletmód, amelynek segítségével a valós világot leegyszerűsítjük, úgy, hogy csak a lényegre, a cél elérése érdekében feltétlenül szükséges részekre összpontosítunk.
    - Elvonatkoztatunk a számunkra pillanatnyilag nem fontos, közömbös információktól és kiemeljük az elengedhetetlen fontosságú részleteket.

A valós világ modellezése

- Megkülönböztetés
  - Az objektumok a modellezendő valós világ egy-egy önálló egységét jelölik.
  - Az objektumokat a számunkra lényeges tulajdonságaik, viselkedési módjuk alapján megkülönböztetjük.

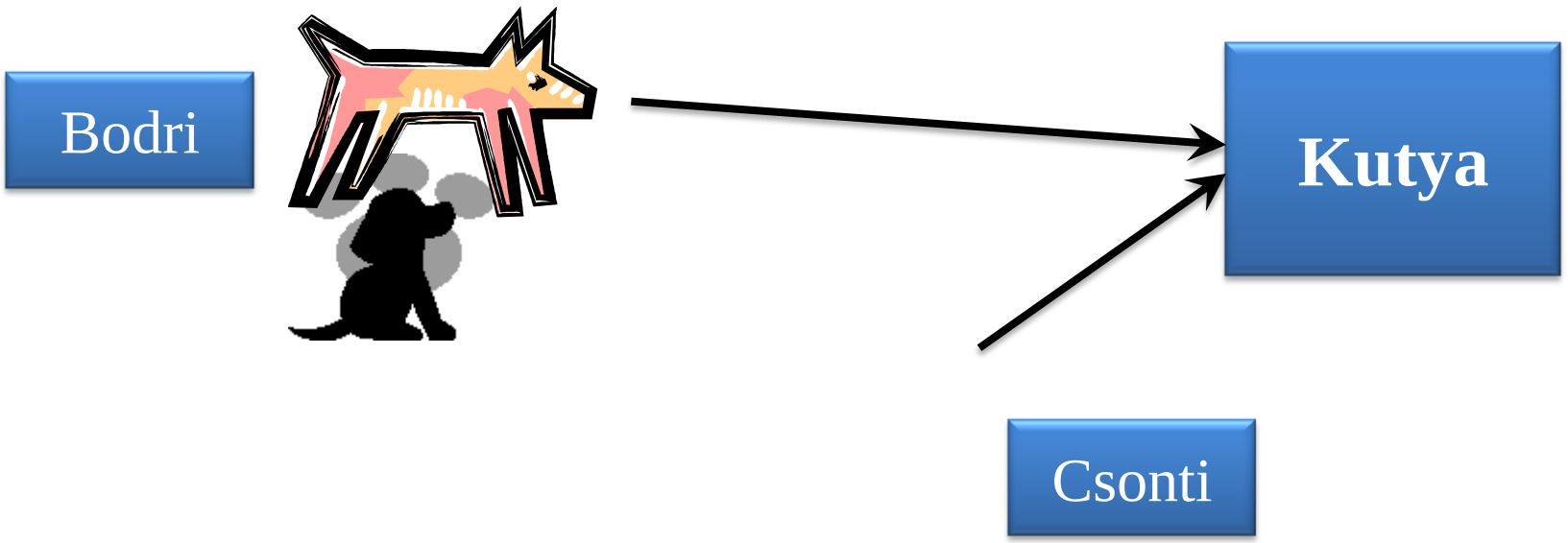
A valós világ modellezése

– Osztályozás

- Az objektumokat kategóriákba, osztályokba soroljuk, oly módon, hogy a hasonló tulajdonságokkal rendelkező objektumok egy osztályba, a különböző vagy eltérő tulajdonságokkal rendelkező objektumok pedig külön osztályokba kerülnek.
- Az objektum-osztályok hordozzák a hozzájuk tartozó objektumok jellemzőit, objektumok **mintá**inak tekinthetők.

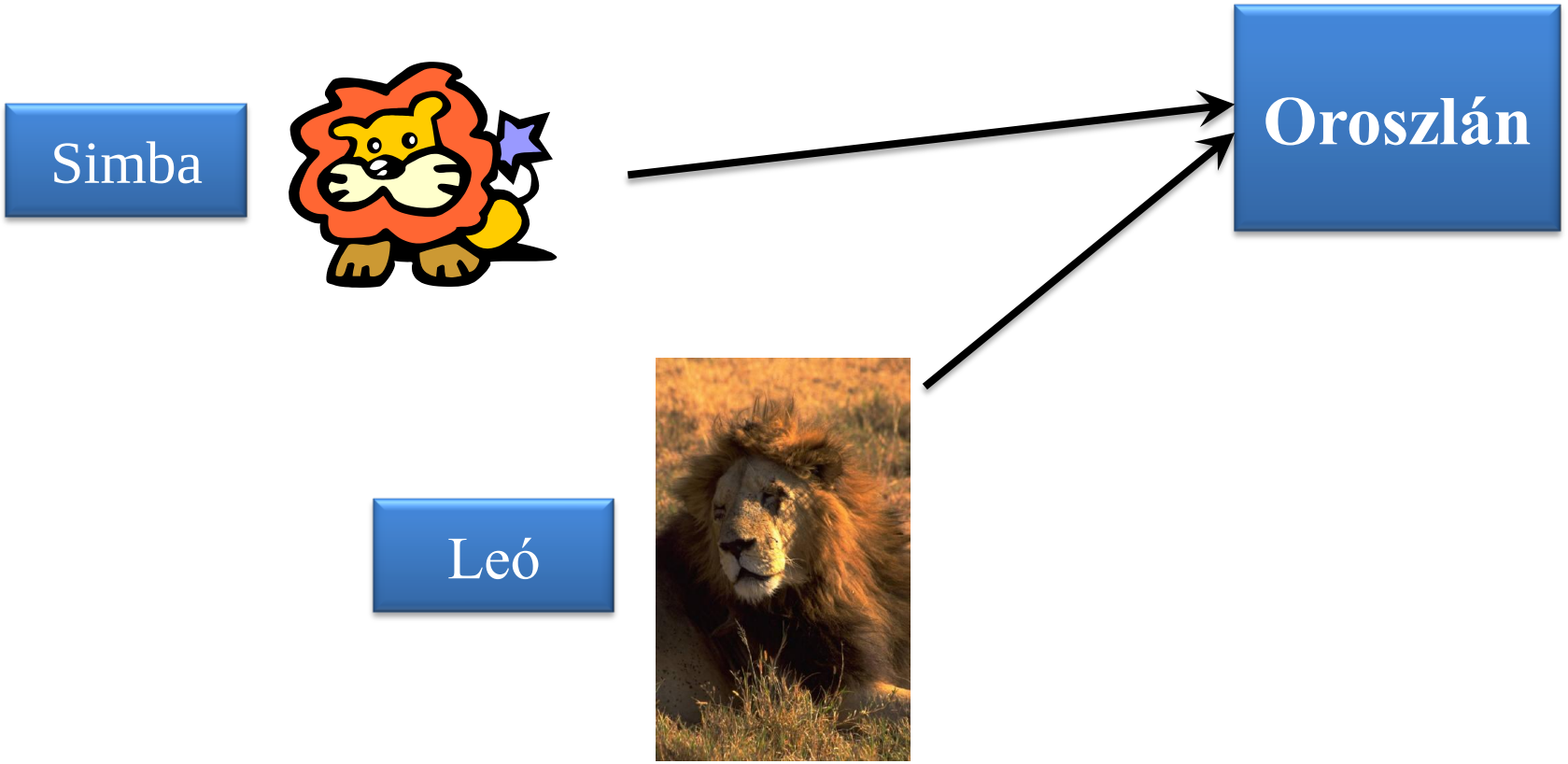
A valós világ modellezése

- Osztályozás



A valós világ modellezése

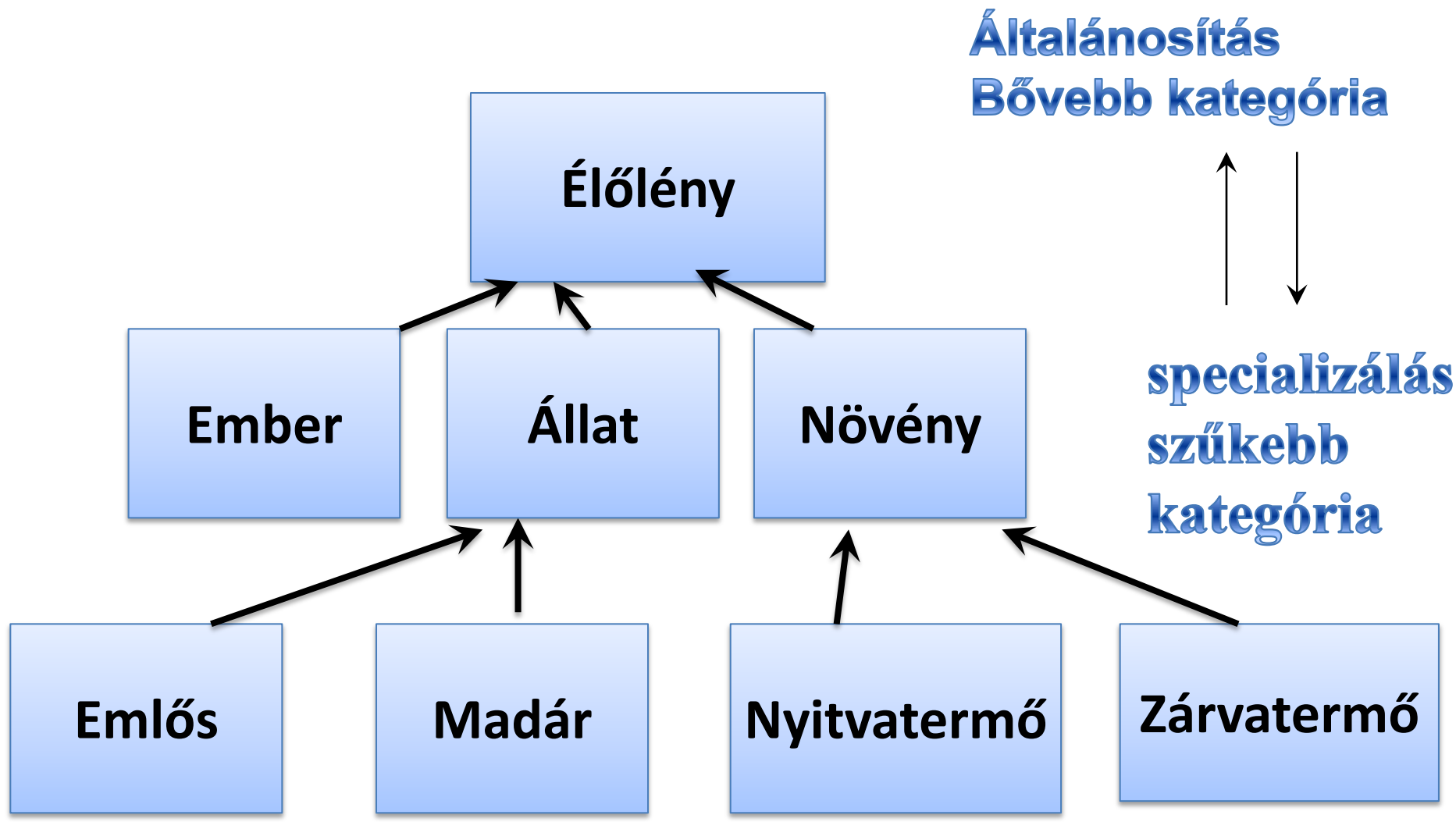
- Osztályozás



A valós világ modellezése

– Általánosítás, specializálás
  - Az objektumok között állandóan hasonlóságokat vagy különbségeket keresünk, hogy ezáltal bővebb vagy szűkebb kategóriákba, osztályokba soroljuk őket.

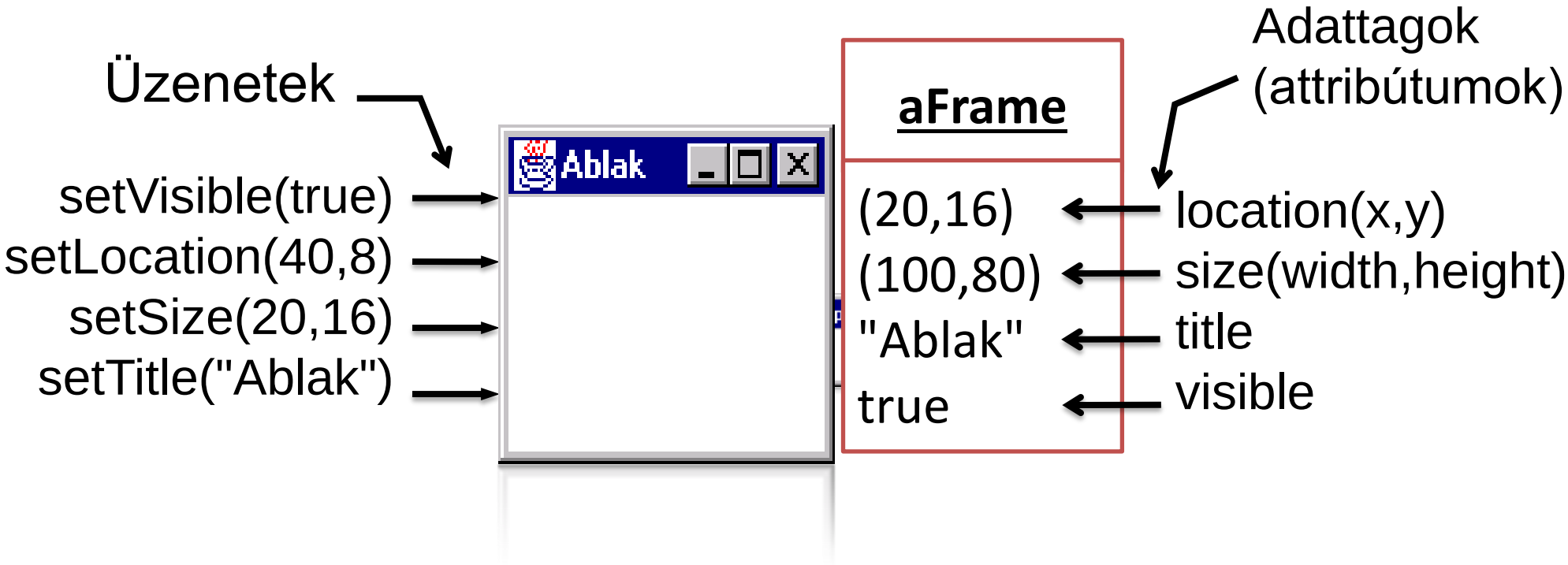
A valós világ modellezése



Objektum

- Belső állapota van, ebben információt tárol, (adattagokkal valósítjuk meg)
- kérésre feladatokat hajt végre – metódusok - melyek hatására állapota megváltozhat
- üzeneteken keresztül lehet megszólítani – ezzel kommunikál más objektumokkal
- Minden objektum egyértelműen azonosítható

Ablak objektum

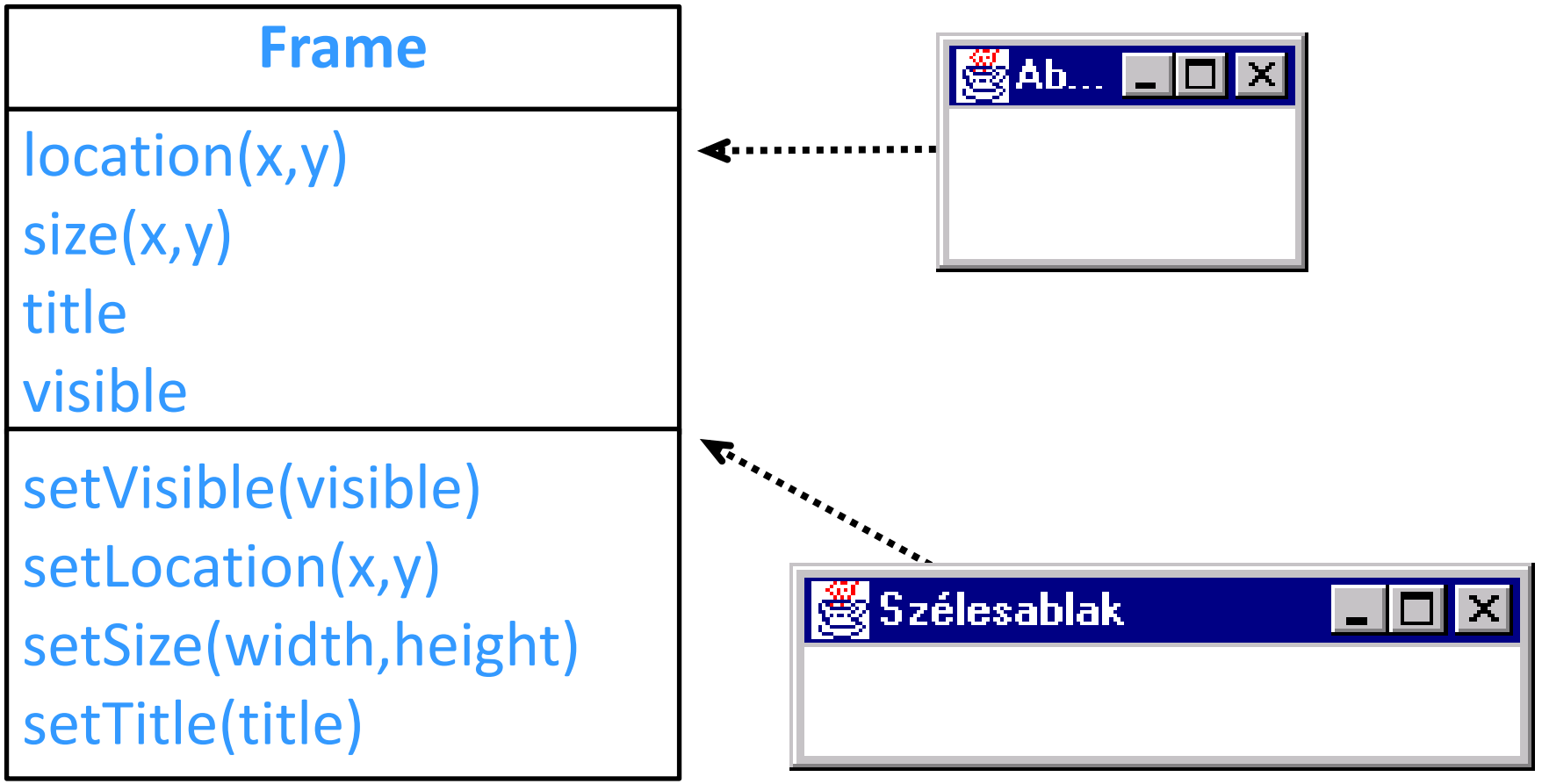


Osztály, példány

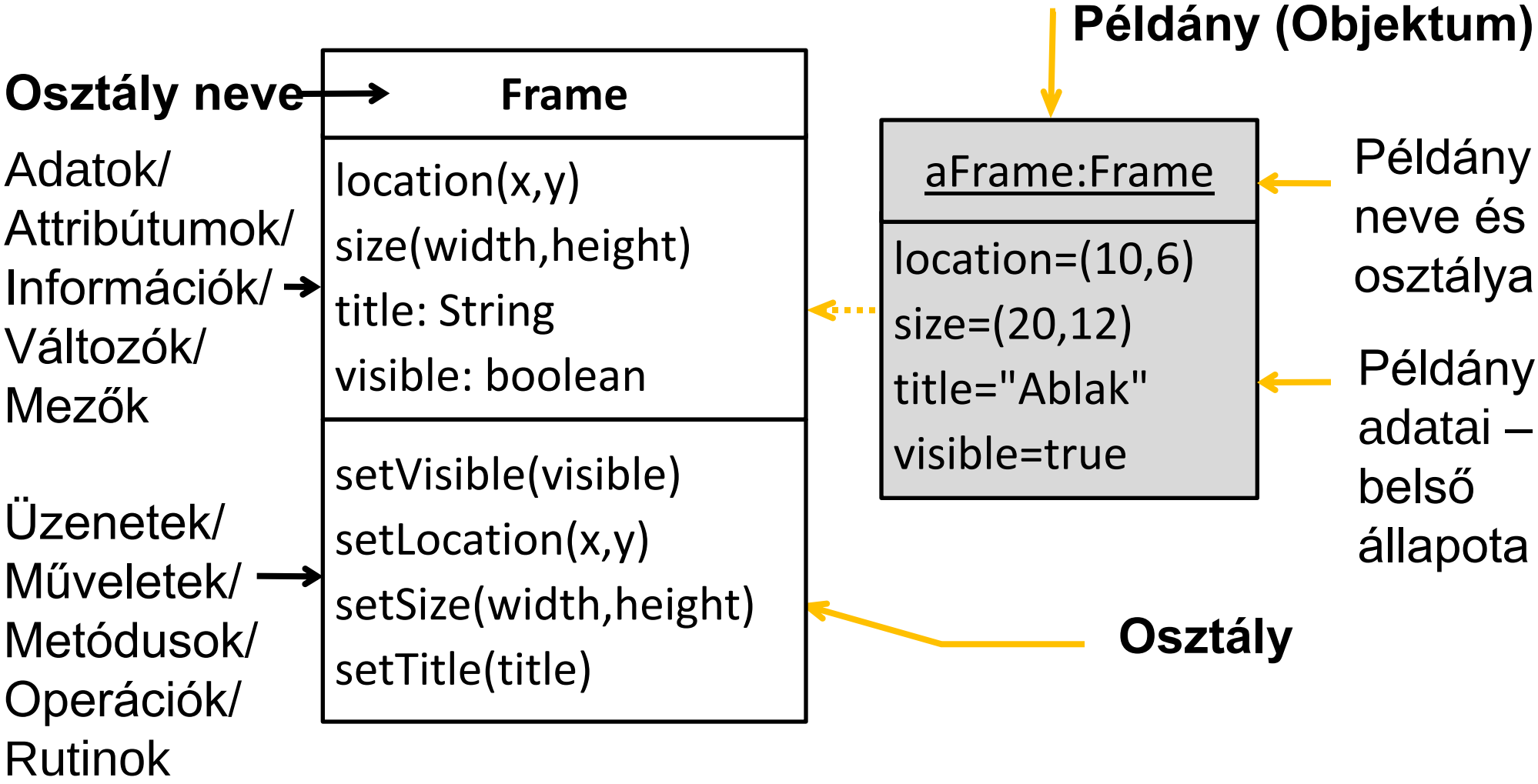
- Osztály (class)
  - Olyan objektumminta vagy típus, mely alapján példányokat (objektumokat) hozhatunk létre
  - Példány (instance)

  Minden objektum születésétől kezdve egy osztályhoz tartozik

Frame osztály és példányai



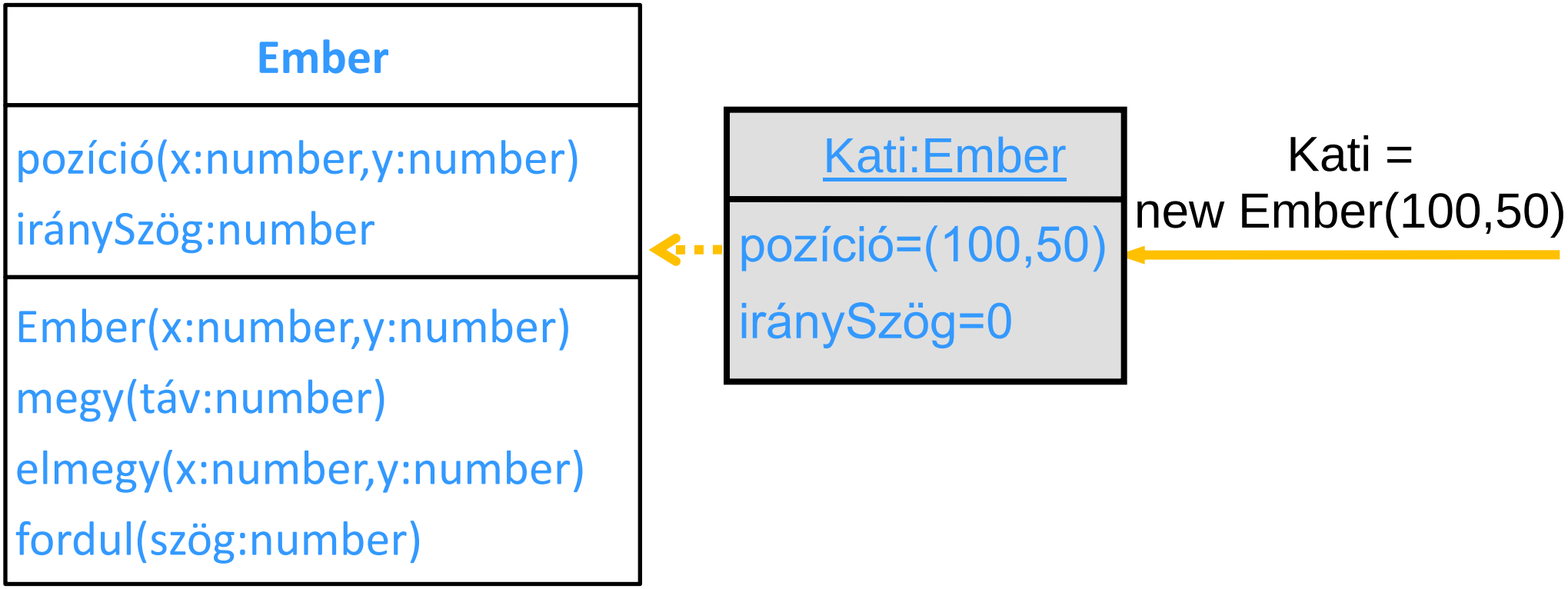
Osztály és példány az UML-ben



Objektum létrehozása, inicializálása

- Objektum életciklusa: „megszületik”, „él”, „meghal”
- Az objektumot létre kell hozni és inicializálni kell!
- Objektum inicializálása
  - konstruktor (constructor) végzi
  - adatok kezdőértékadása
  - objektum működéséhez szükséges tevékenységek végrehajtása
  - típusinvariáns beállítása

Objektum létrehozása, inicializálása



Objektum műveletei

- Export műveletek
  - amelyeket más objektumok hívhatnak
  - pl. verem: push, pop, top stb.
- Import műveletek
  - amelyeket az objektum igényel ahhoz, hogy az
  - export szolgáltatásait nyújtani tudja
  - pl. verem, ha fix méretű (vektoros) reprezentáció:
  - vektorműveletek

Objektum műveletei

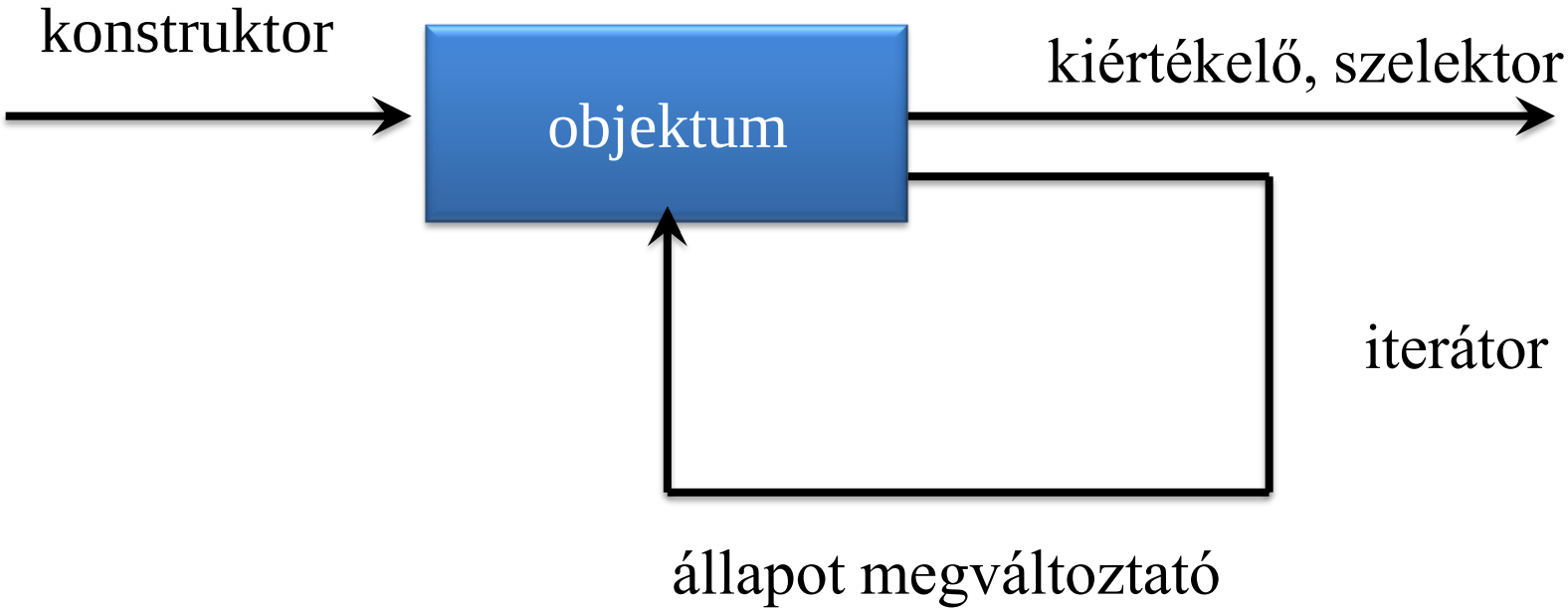
- Export műveletek csoportosítása:
  - Létrehozó (konstruktor)
    az objektum létrehozására, felépítésére
    - pl. veremnél:
    - create: →Verem
  - Állapot megváltoztató
    - pl. veremnél:
    - pop: Verem → Verem
    - push: Verem x Elem → Verem

Objektum műveletei

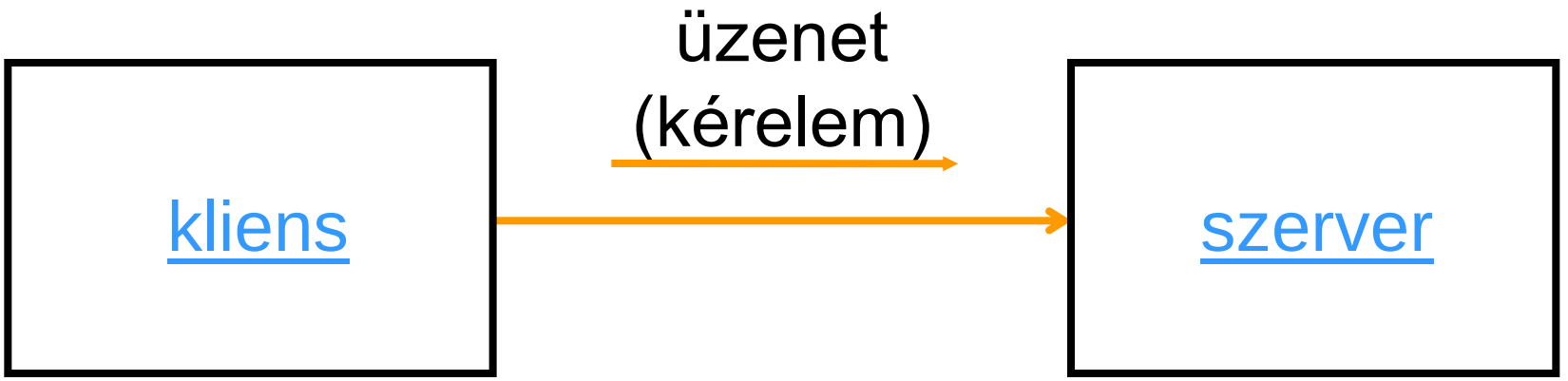
- Export műveletek csoportosítása:
  - Szelektor
    kiemeli az objektum bizonyos részét
    - pl. vektor adott indexű elemét
    - access: Vektor x Index → Elem
  - Kiértékelő
    objektum jellemzőit lekérdező műveletek
    (size, has, stb.)
  - Iterátor - bejáráshoz

Objektum műveletei

- Export műveletek csoportosítása:



Kliens üzen a szervernek



Kívülről elérhető metódus hívása

A feladatot elvégeztető objektum

A feladatot elvégző objektum

Kliens üzen a szervernek

- Kliens: aktív objektum, másik objektumon végez műveleteket, de rajta nem végeznek. Nincs export felülete. Pl. óra – meghatározott időközönként művelet egy regiszteren.
- Szerver: passzív objektum, csak export felülete van. Másoktól érkező üzenetekre vár, mások szolgáltatását nem igényli. Nincs import felülete.
- Ágens: általános objektum, van export és import felülete.

Osztály, példány

Minden objektum?

Akkor az osztályok is …

- Lehet belső állapotuk,
- Küldhetünk üzeneteket neki …

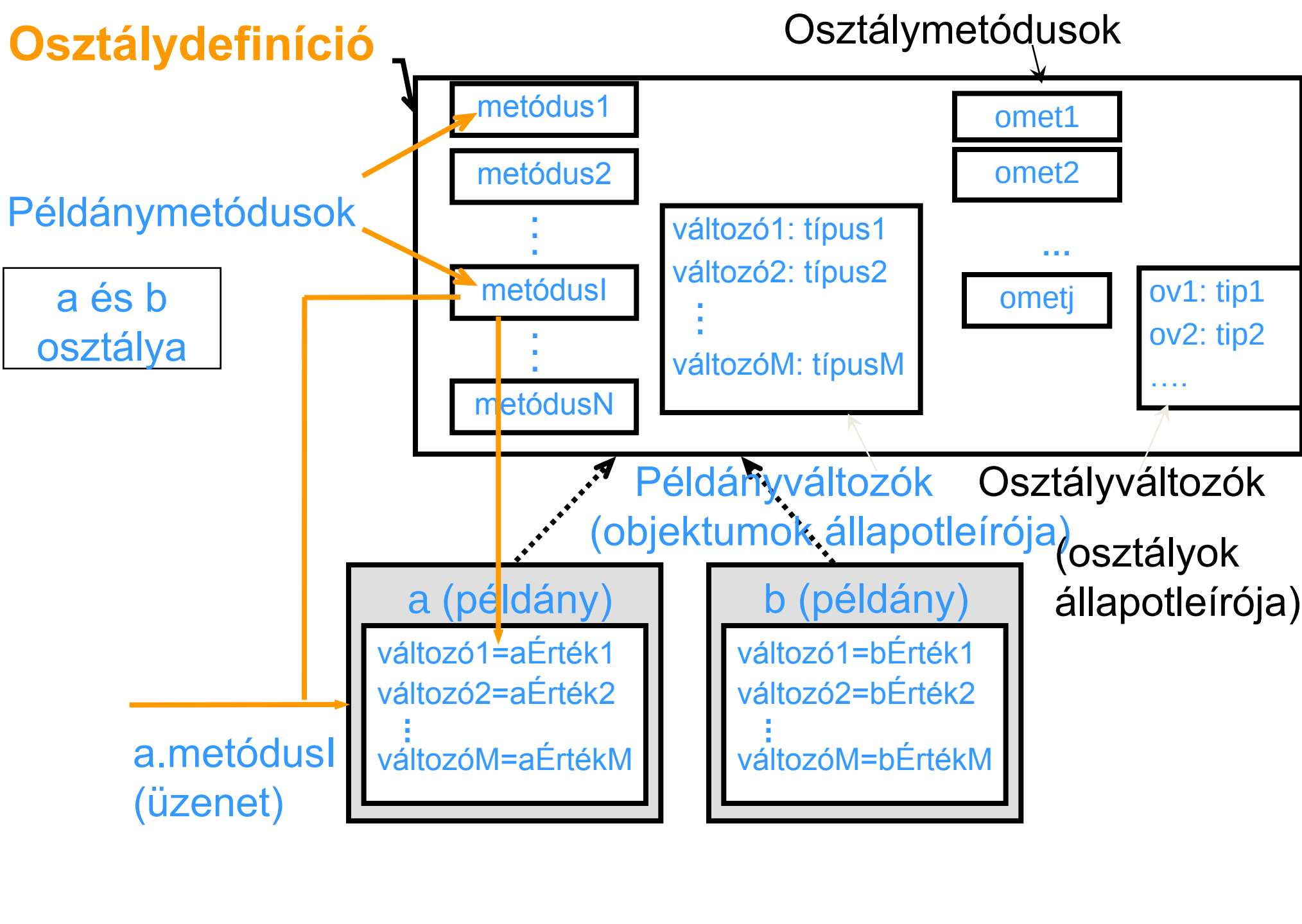
Minek az objektuma?

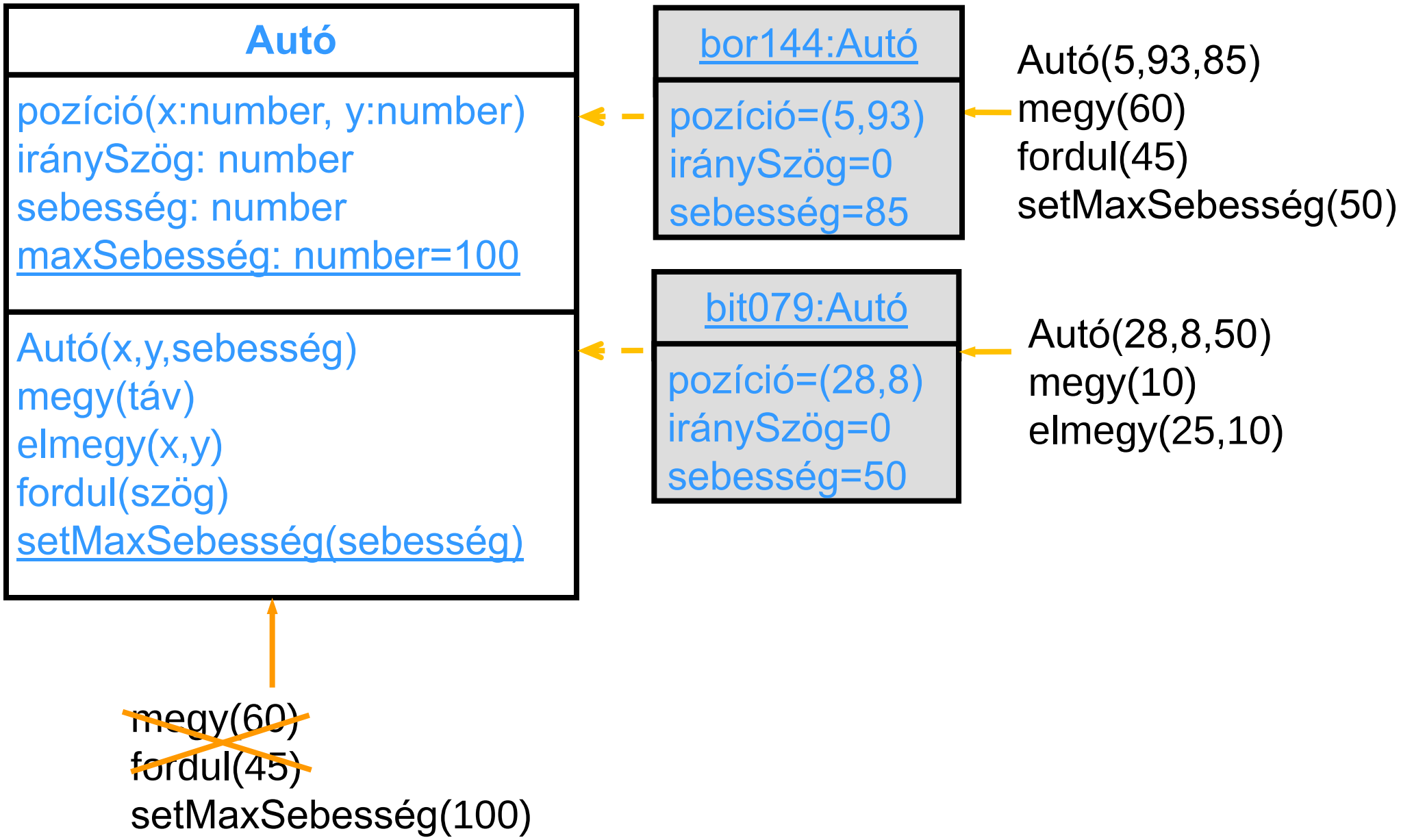
- metaosztály
  - singleton objektum

És a metaosztály? … ☺

Osztály, példány

- Osztálydefiníció:
  - **Példányváltozó**
    - Példányonként helyet foglaló változó
  - **Példánymetódus**
    - Példányokon dolgozó metódus
  - **Osztályváltozó**
    - Osztályonként helyet foglaló változó
  - **Osztálymetódus**
    - Osztályokon dolgozó metódus


Osztálydefiníció
Osztálymetódusok
metódus1
metódus2
metódusI
metódusN
omet1
omet2
ometj
változó1: típus1
változó2: típus2
változóM: típusM
ov1: tip1
ov2: tip2
….
Példánymetódusok
a és b
osztálya
Példányváltozók
(objektumok állapotleírója)
Osztályváltozók
(osztályok
állapotleírója)
a (példány)
változó1=aÉrték1
változó2=aÉrték2
változóM=aÉrtékM
b (példány)
változó1=bÉrték1
változó2=bÉrték2
változóM=bÉrtékM
a.metódusI
(üzenet)


Autó
pozíció(x:number, y:number)
iránySzög: number
sebesség: number
maxSebesség: number=100
Autó(x,y,sebesség)
megy(táv)
elmegy(x,y)
fordul(szög)
setMaxSebesség(sebesség)
bor144:Autó
pozíció=(5,93)
iránySzög=0
sebesség=85
Autó(5,93,85)
megy(60)
fordul(45)
setMaxSebesség(50)
bit079:Autó
pozíció=(28,8)
iránySzög=0
sebesség=50
Autó(28,8,50)
megy(10)
elmegy(25,10)
megy(60)
fordul(45)
setMaxSebesség(100)

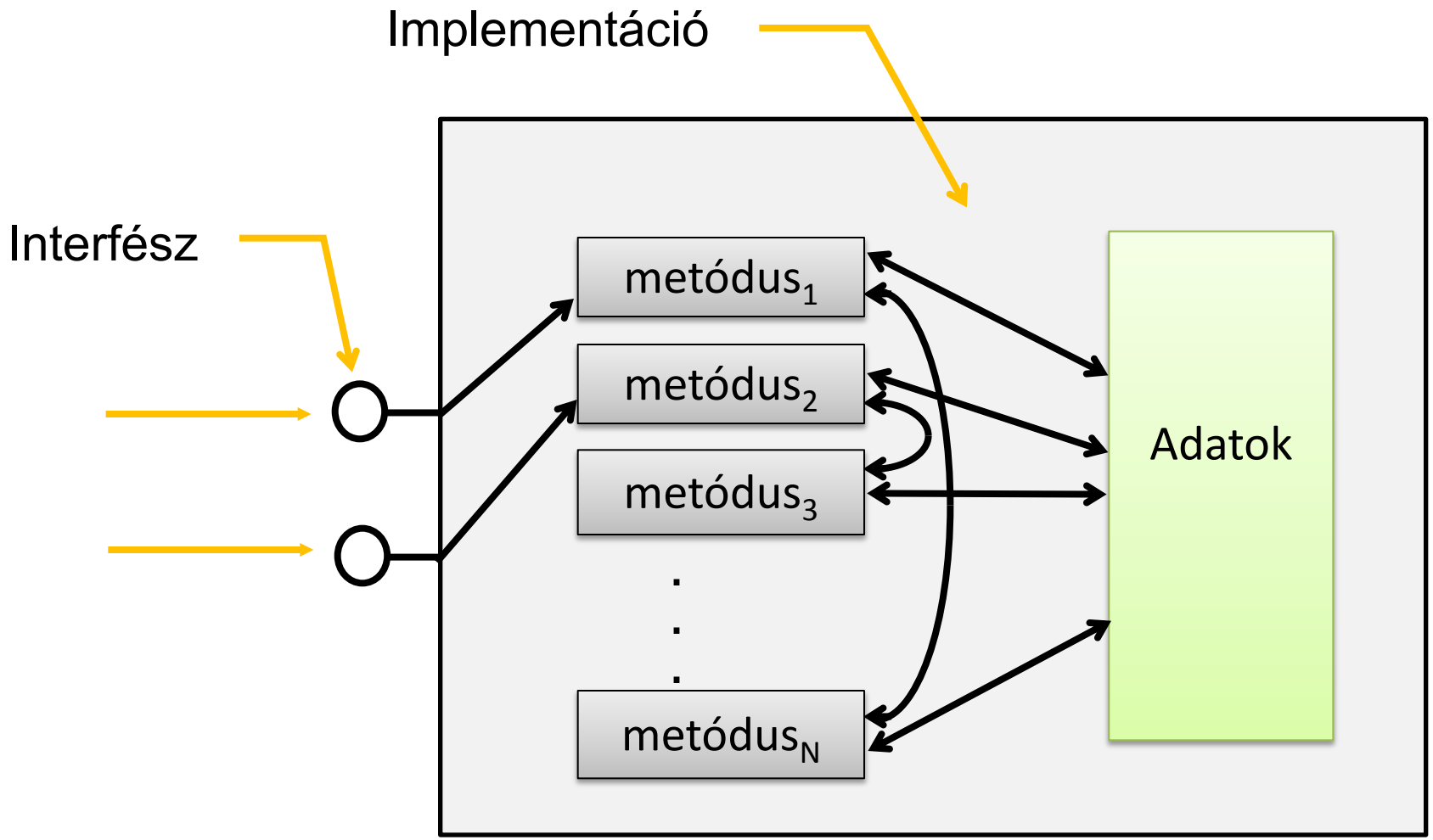
A „this”

- Ha egy osztályból több objektumot példányosítunk, honnan tudjuk, hogy éppen melyik objektum hívta meg a megfelelő metódust, és a metódus melyik objektum adataival fog dolgozni?
- Szükségünk van egy olyan mutatóra, amely mindig a metódust meghívó objektumpéldányra mutat. Ezt szolgálja a „this” „paraméter”. Ez metódushíváskor egyértelműen rámutat azokra az adatokra, amelyekkel a metódusnak dolgoznia kell.
- Ez azt is jelenti, hogy ha az objektum saját magának akar üzenetet küldeni, akkor a this.Üzenet(Paraméterek) formát kell, hogy használja, vagyis a metódustörzsekben az adott példányra mindig a this segítségével hivatkozhatunk.
  (Ez számos nyelvben alapértelmezett.)

OOP elvárások

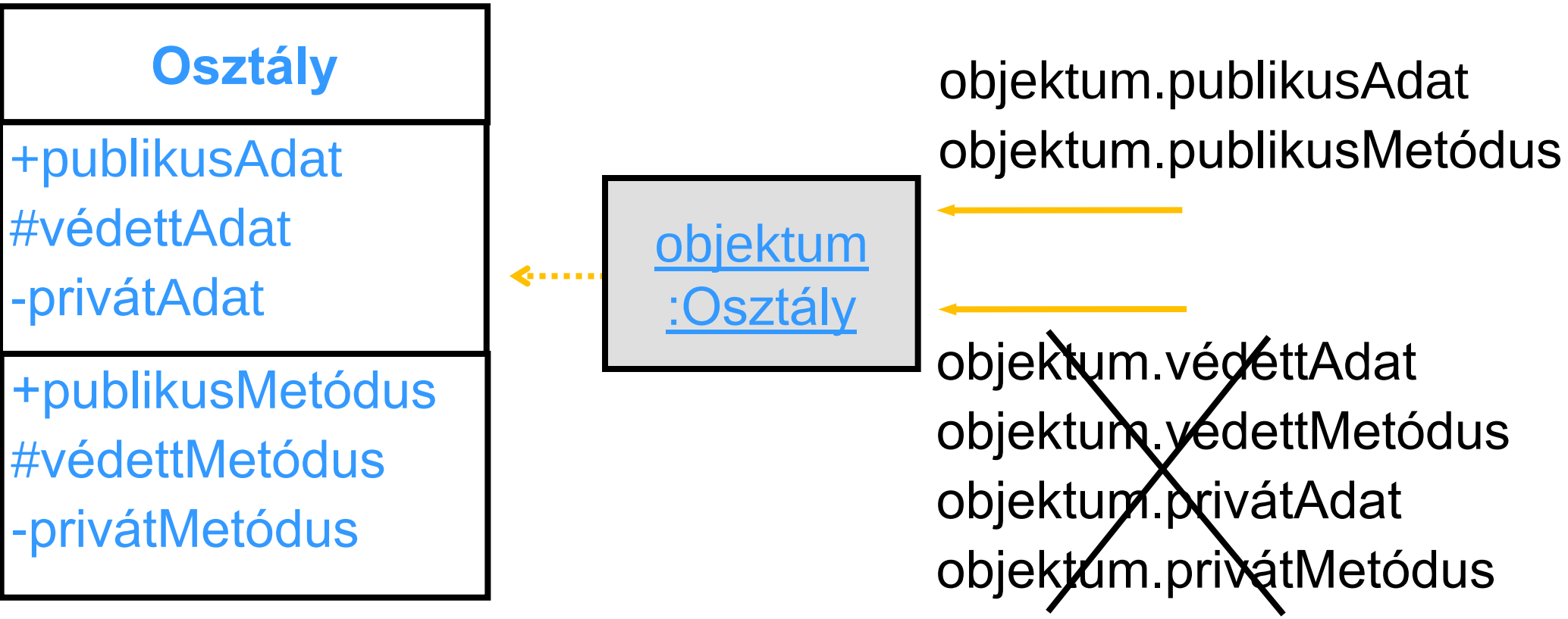
- Bezárás (encapsulation)
  - adatok és metódusok összezárása
  - egybezárás, egységbezárás - osztály (class)
- Információ elrejtése (information hiding)
  - az objektum „belügyeit” csak az interfészen keresztül lehet megközelíteni (láthatóságok!)
- Kód újrafelhasználása (code reuse)
  - megírt kód felhasználása példány létrehozásával vagy osztály továbbfejlesztésével

Információ elrejtése

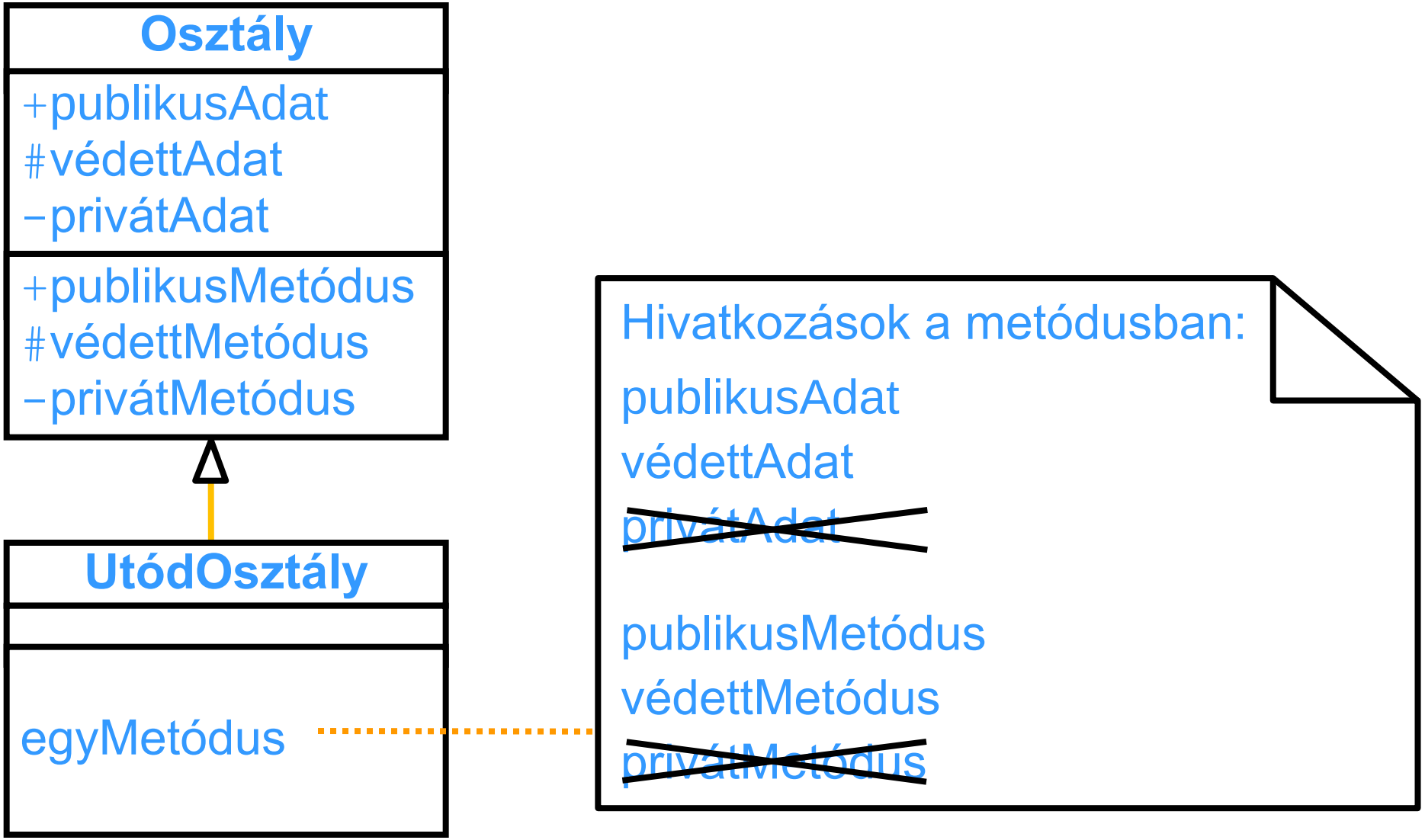


Láthatóság

Objektum védelme

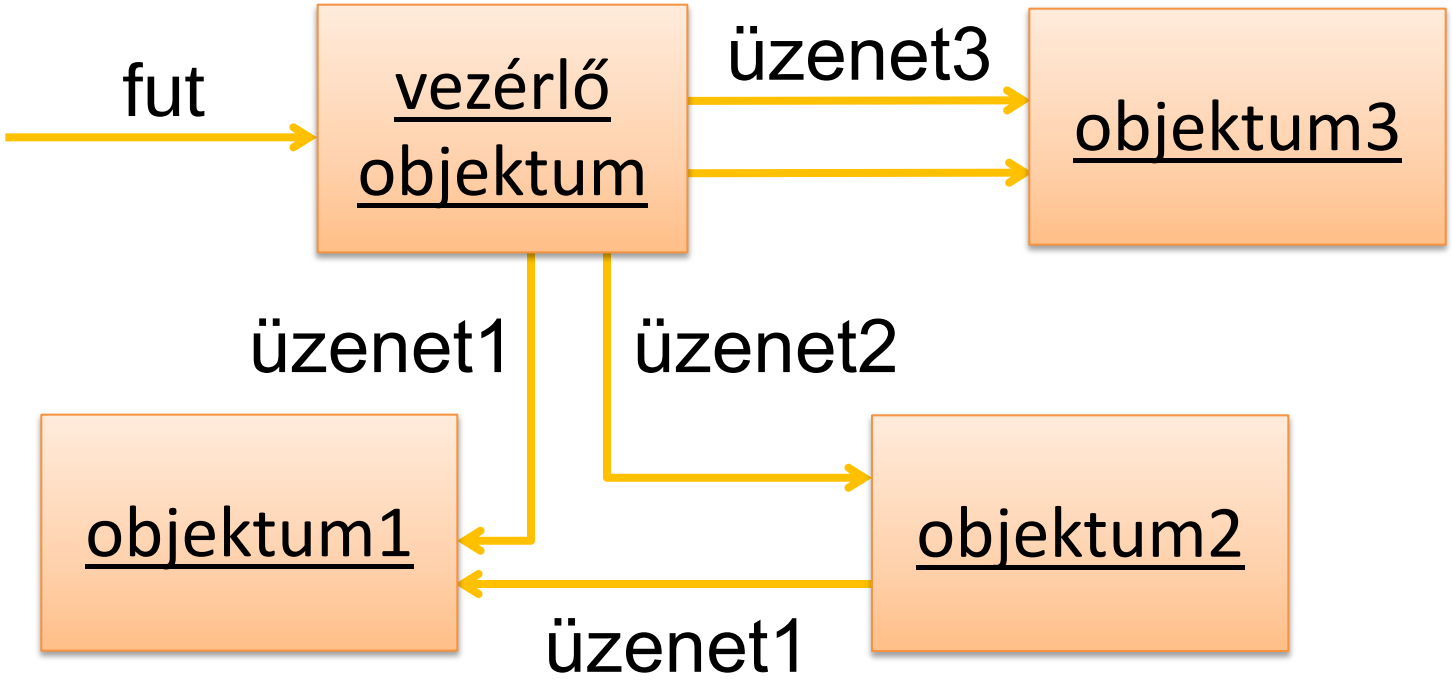


Osztály védelme



OO program

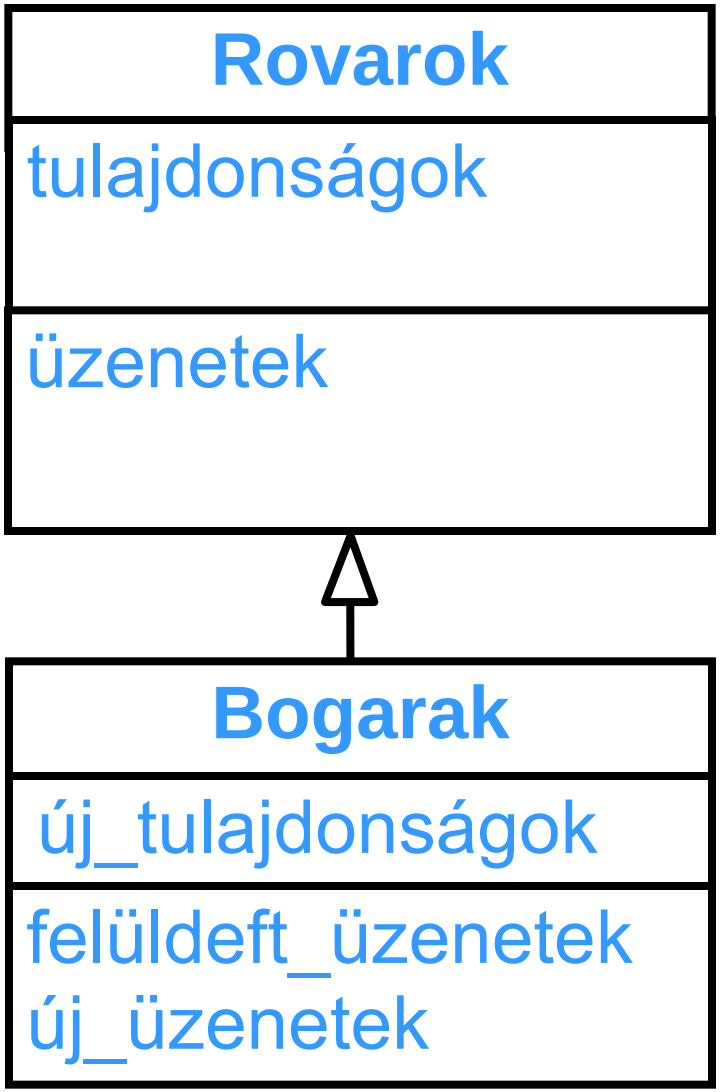
- Egy objektumorientált program egymással kommunikáló objektumok összessége, melyben minden objektumnak megvan a feladatköre.



Öröklődés

- Az alapgondolat: a gyerekek öröklik őseik metódusait és változóit.
- Az *örököl* terminus azt jelenti, hogy az ősosztály **minden** metódusa és adattagja a gyerekosztálynak is metódusa és adattagja lesz.
- A gyerek minden új művelete vagy adattagja egyszerűen hozzáadódik az örökölt metódusokhoz és adattagokhoz.
- Minden metódus, amit átdefiniálunk a gyerekben, a hierarchiában felülbírálja az örökölt metódust.

Öröklődés

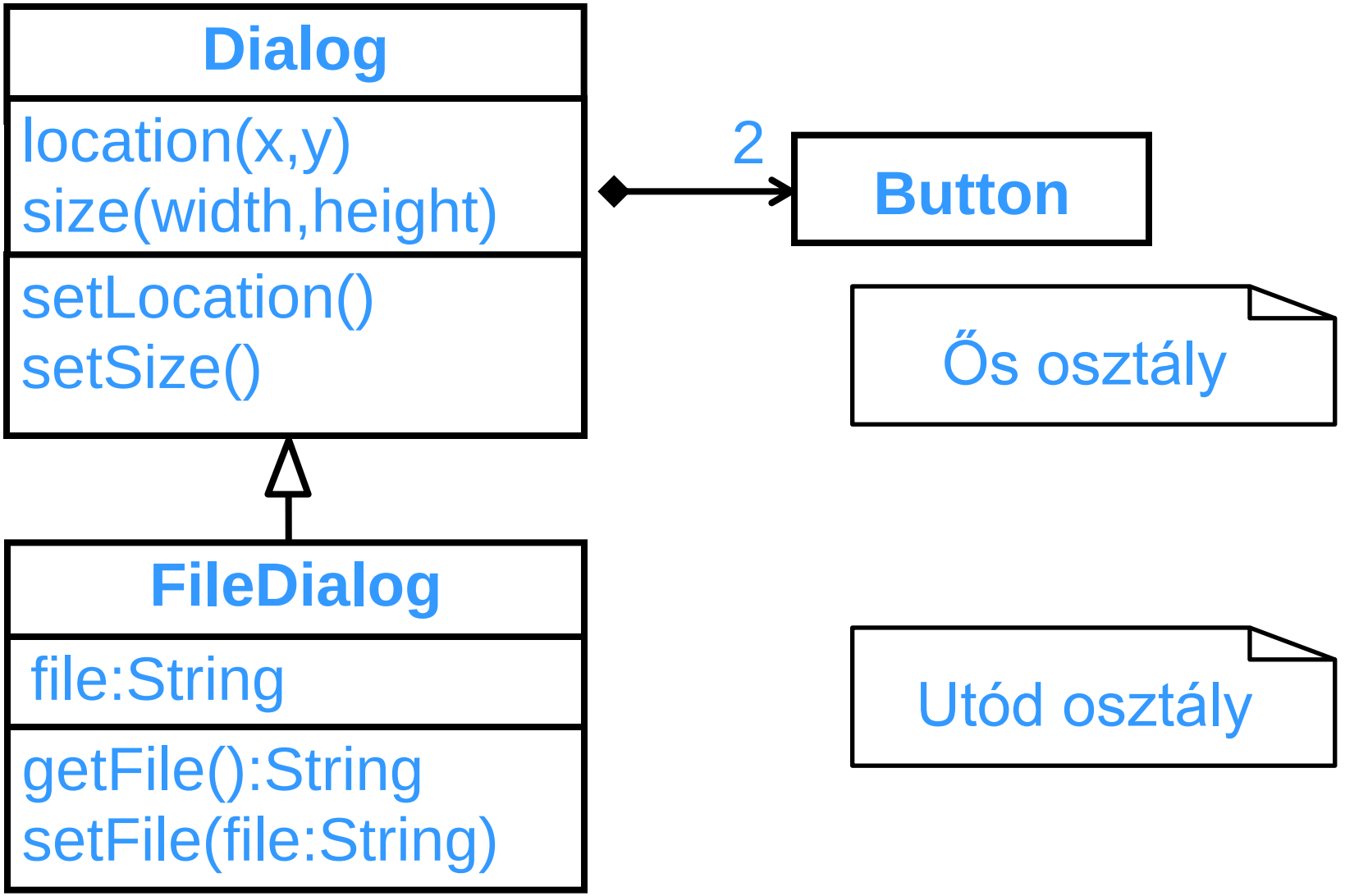


„IS-A” reláció

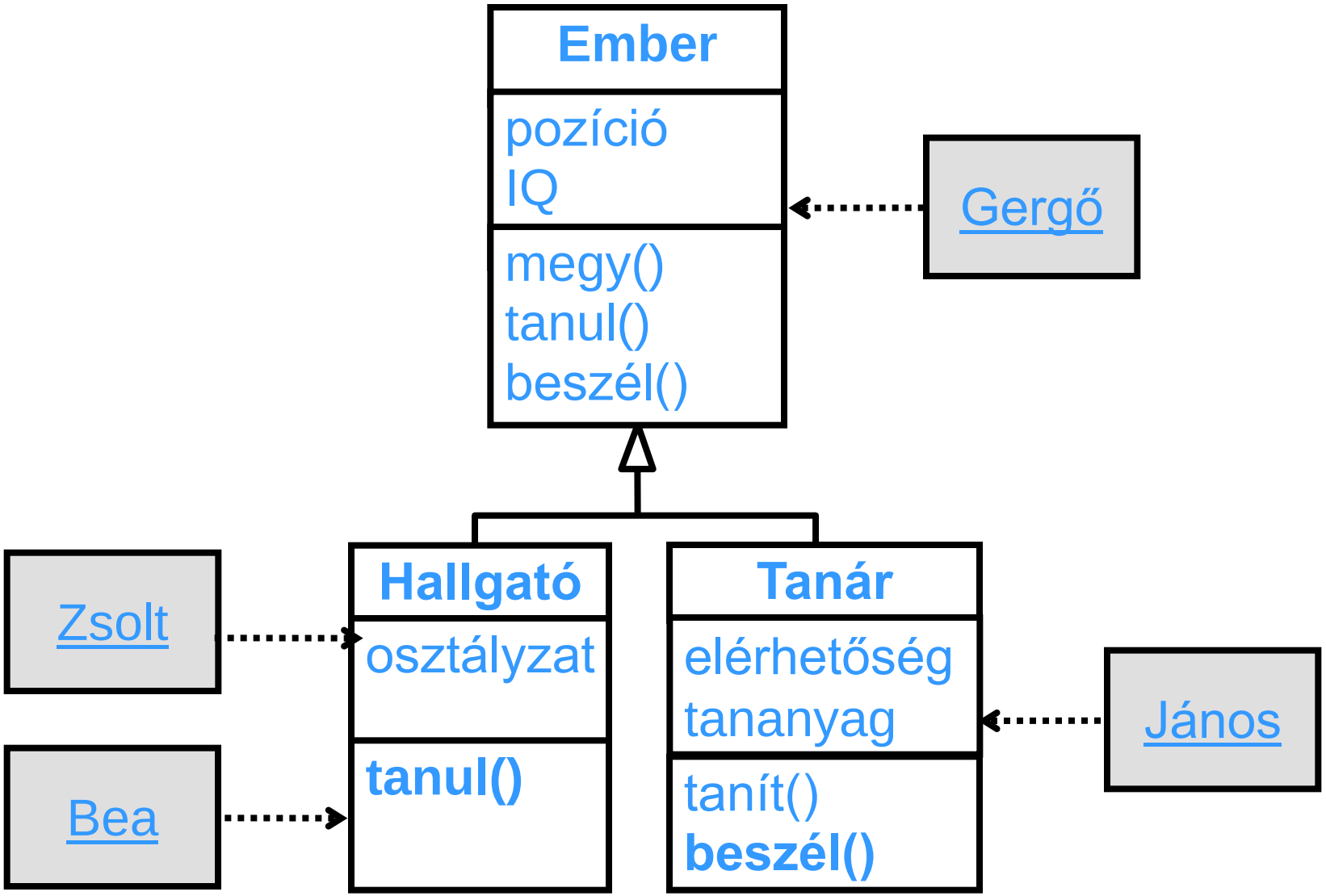
Ős osztály

Utód osztály

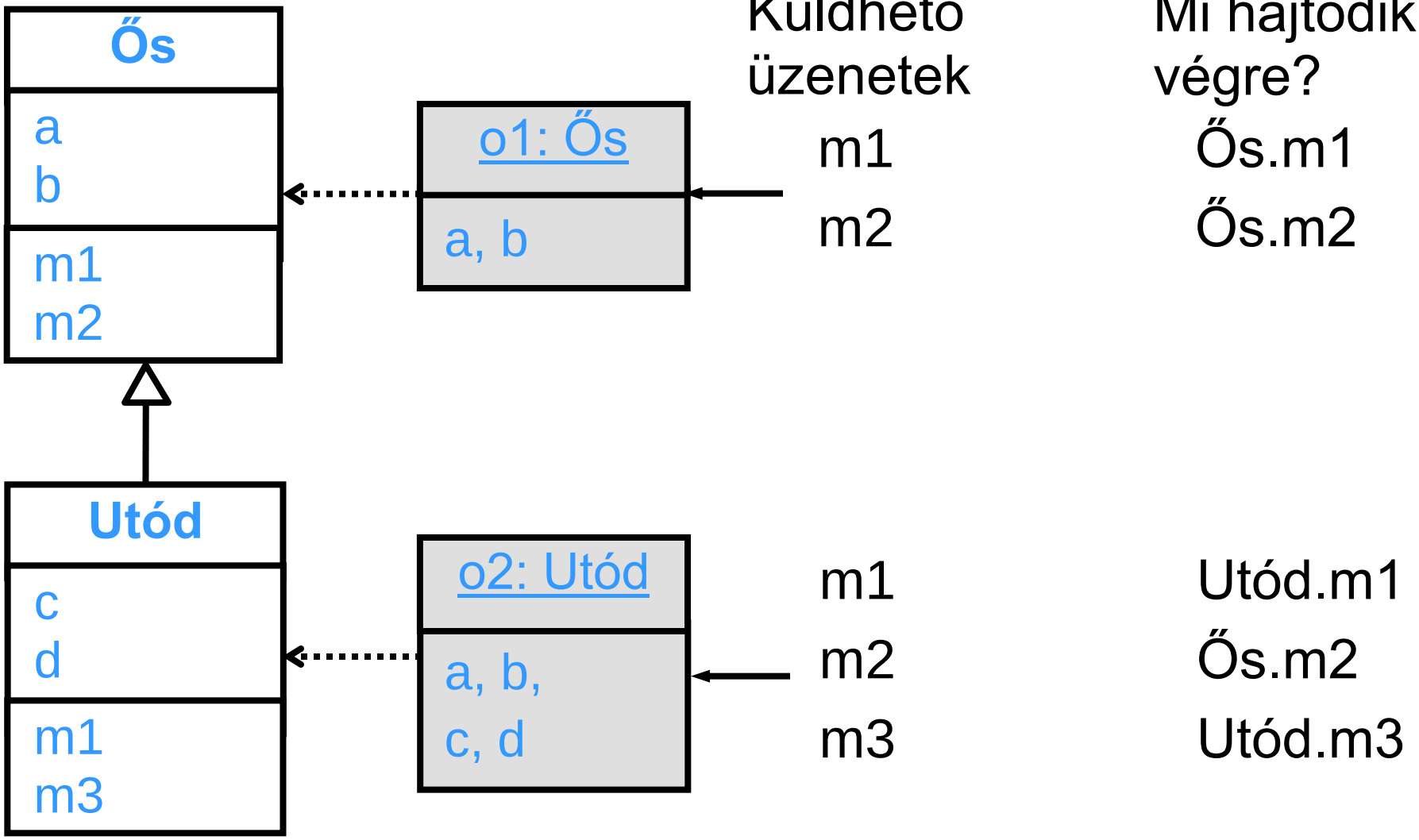
Öröklődés


Dialog
location(x,y)
size(width,height)
setLocation()
setSize()
2
Button
Ős osztály
FileDialog
file:String
getFile():String
setFile(file:String)
Utód osztály

Ki mire képes?



Utód adatai, küldhető üzenetek

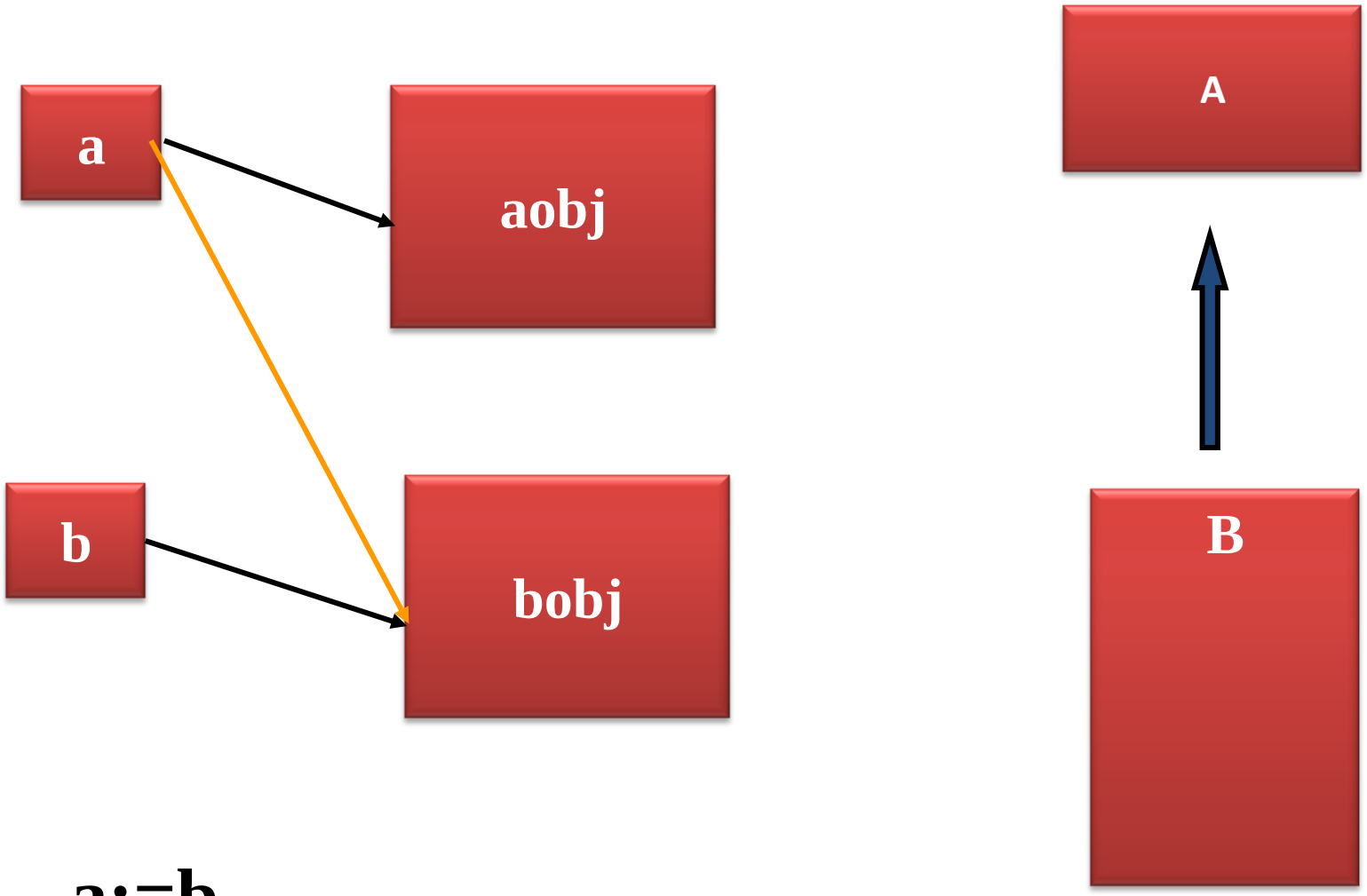


Polimorfizmus

- Polimorfizmus (többbalakúság): az a jelenség, hogy egy változó nem csak egyfajta típusú objektumra hivatkozhat.
  - Statikus típus: a deklaráció során kapja.
  - Dinamikus típus: run-time éppen milyen típusú objektumra hivatkozik = a statikus típus, vagy annak leszármazottja.

  - Aki Manager , az egy (is-a) Employee is.
  - A Háromszög az egy Alakzat.

Hogy lenne jó?



a:=b

39

Altípusos polimorfizmus

```
Shape *s;
…
s = new Triangle (...);
...
s = new Rectangle (...);
...
```

- Ha B (pl. Triangle, Rectangle) altípusa az A (pl. Shape) típusnak, akkor B objektumainak referenciái értékül adhatók az A típus referenciáinak.

```
Shape *a;
Triangle* h= new Triangle (...);
Rectangle *t= new Rectangle (...);
a = h; a->draw(); …
a = t; a->draw(); …
```

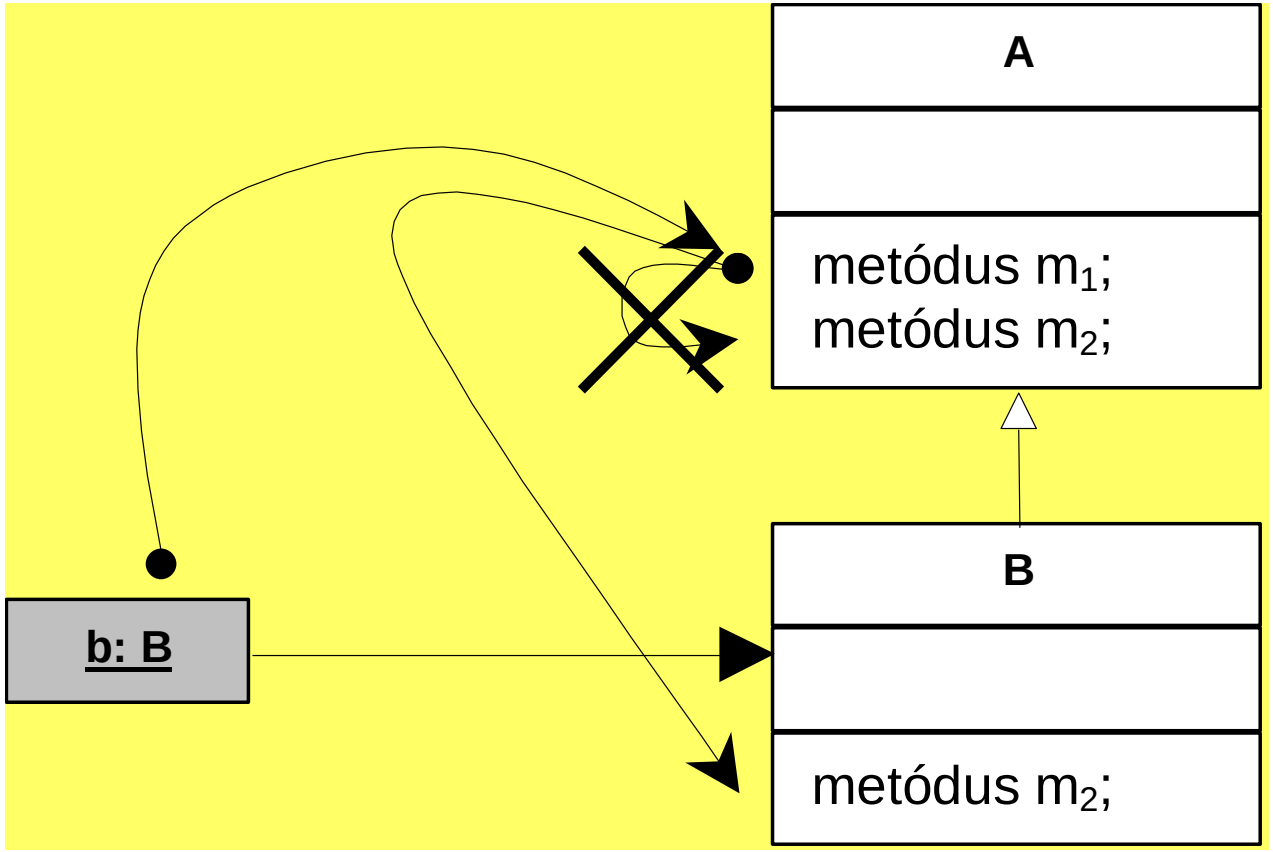
Melyik draw()?

Dinamikus összekapcsolás

- Run-time fogalom. Az a jelenség, hogy a változó éppen aktuális dinamikus típusának megfelelő metódus implementáció hajtódik végre.

- A háromszög kirajzolása,
- a manager kinyomtatása….

Dinamikus összekapcsolás (2)



Altípus

Szabályok:

$T \subseteq T$ [*reflexív-* $\subseteq$]

$$\frac{T_1 \subseteq T_2;\ T_2 \subseteq T_3}{T_1 \subseteq T_3}$$ [*tranzitív-* $\subseteq$]

Altípus - alprogramoknál

Tegyük fel, hogy Triangle $\subseteq$ Shape

fss = **proc** (Shape) **returns** (Shape)

fts = **proc** (Triangle) **returns** (Shape)

fst = **proc** (Shape) **returns** (Triangle)

ftt = **proc** (Triangle) **returns** (Triangle)

fts $\subseteq$ fss	?	fst $\subseteq$ fss	?
ftt $\subseteq$ fss	?	fss $\subseteq$ fts	?

Hogyan döntsünk?

Egy függvényt $A \rightarrow B$ alakban írunk, ha A típusú paramétere van és B típusú eredményt ad.

Ha $(A' \rightarrow B') <: (A \rightarrow B)$, - altípus – akkor

képesek kell legyünk arra, hogy az első függvénytípus egy elemét használhassuk minden olyan kontextusban, ahol a másodikat.

Tegyük fel, hogy van egy f: $A \rightarrow B$ típusú függvényünk.

Ha egy f' : $A' \rightarrow B'$ függvényt az f helyén akarunk használni, akkor A-beli argumentumot kell fogadnia és B-beli eredményt adnia.

Hogyan döntsünk?

Mivel az f’ értelmezési tartománya A’, így akkor alkalmazható A-beli elemekre, ha A<:A’, vagyis ha a : A tekinthető mint A’-beli, és f’(a) típus-helyes.

Másrészt, ha az f’ eredménye B’-beli, akkor B’<:B garantálja, hogy az eredmény B-beliként kezelhető.

Összegezve:

(A’ → B‘) <: (A → B), ha A<:A’ és B’<:B

Altípus – alprogramoknál (2)

fss = **proc** (Shape) **returns** (Shape)

fts = **proc** (Triangle) **returns** (Shape)

fst = **proc** (Shape) **returns** (Triangle)

ftt = **proc** (Triangle) **returns** (Triangle)

~~fts ⊆ fss~~

fst ⊆ fss

~~ftt ⊆ fss~~

fss ⊆ fts

Az eredmény lehet speciálisabb
(monoton = kovariáns)

A paraméterek **kevésbé** speciálisak lehetnek
(**anti-**monoton = kontravariáns)

Mi az objektumorientált programozás?

- A programozó definiálhat altípus kapcsolatokat
- A típusszabályok megengedik, hogy az altípus használható legyen a szupertípus helyén *(altípusos polimorfizmus)*
- Típus-vezérelt metódus elérés (dinamikus kötés)
- Implementáció megosztása (öröklődés)

Típus-vezérelt metódus elérés

```
s: Shape := new Triangle (3, 4, 5);
s.draw();
```

Statikus elérés:

A Shape draw metódusát hívja

Dinamikus elérés:

A Triangle draw metódusát hívja

Elérési döntések

- C++
  - Az **őstípus** `virtual`-nak deklarálja a metódust, amire megengedi a felüldefiniálást
- Java
  - Minden felüldefiniálható,
    (hacsak az **őstípus** nem deklarálta `final`-nak)
- Eiffel
  - Az **altípus** explicit `redefine` záradékot használ
    (ha nem „`frozen`”)

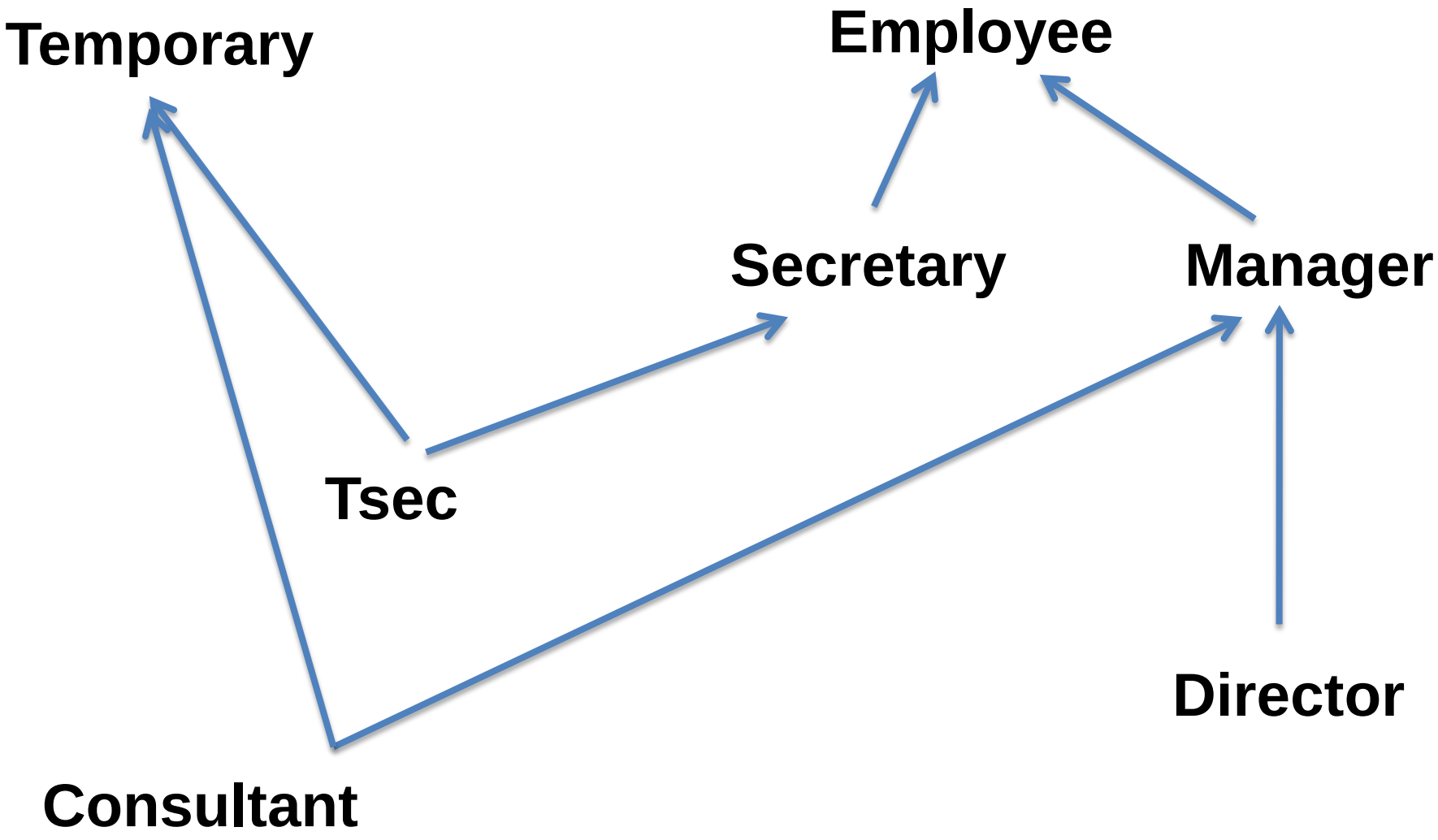
C++ példa

- egy leszármazott lehet ős is

```
class Employee { … };
class Manager : public Employee {…};
class Director: public Manager {…};
```

- Az osztályhierarchia lehet fa, de lehet általánosabb gráf is:

```
class Temporary { … };
class Secretary: public Employee { … };
class Tsec: public Temporary, public Secretary{ … };
class Consultant: public Temporary, public Manager { … };
```


Temporary
Employee
Secretary
Manager
Tsec
Director
Consultant

Implementáció újrahasznosítás: *alosztályképzés*

- Használd egy típus implementációját egy másik típus implementálására!
- *Gyakran* használjuk az őstípus implementációját az altípus implementálására
- A gyakran használt OO programozási nyelvek keverik az altípus és alosztály fogalmakat:
  - C++ - implementációs öröklés altípus nélkül: `private`, `protected` öröklés

Reprezentáció öröklése

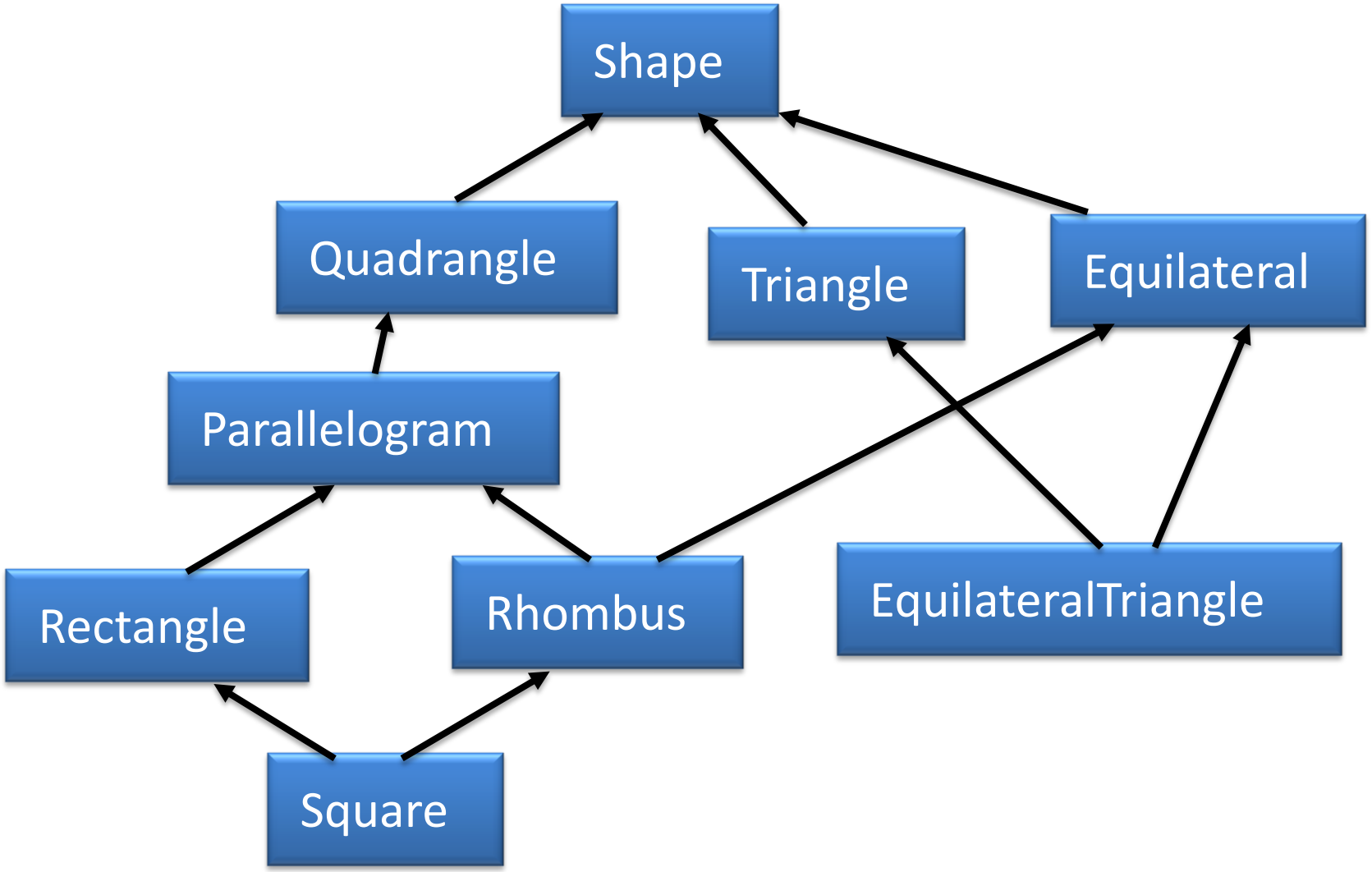
Eiffel

A FIXED_STACK a reprezentációját az ARRAY osztálytól örökli, de ezt elrejti a kliensei elől:

```
class FIXED_STACK [T] inherit
  STACK[T];
  ARRAY[T]
   rename
      put as array_put.....
    export
      {NONE} all
    end
  feature
......
```

Az all kulcsszó az összes, ezen osztálytól öröklött jellemzőre vonatkozik.

Egy típus- és osztályhierarchia



Adjunk hozzá egy attribútumot!

- Az alakzatoknak legyen színe – color – és egy set_color metódusa

A. Változtassuk meg a Shape, Quadrangle, Parallelogram, Triangle, Equilateral, EquilateralTriangle, Rhombus, Rectangle, Square stb. típusokat

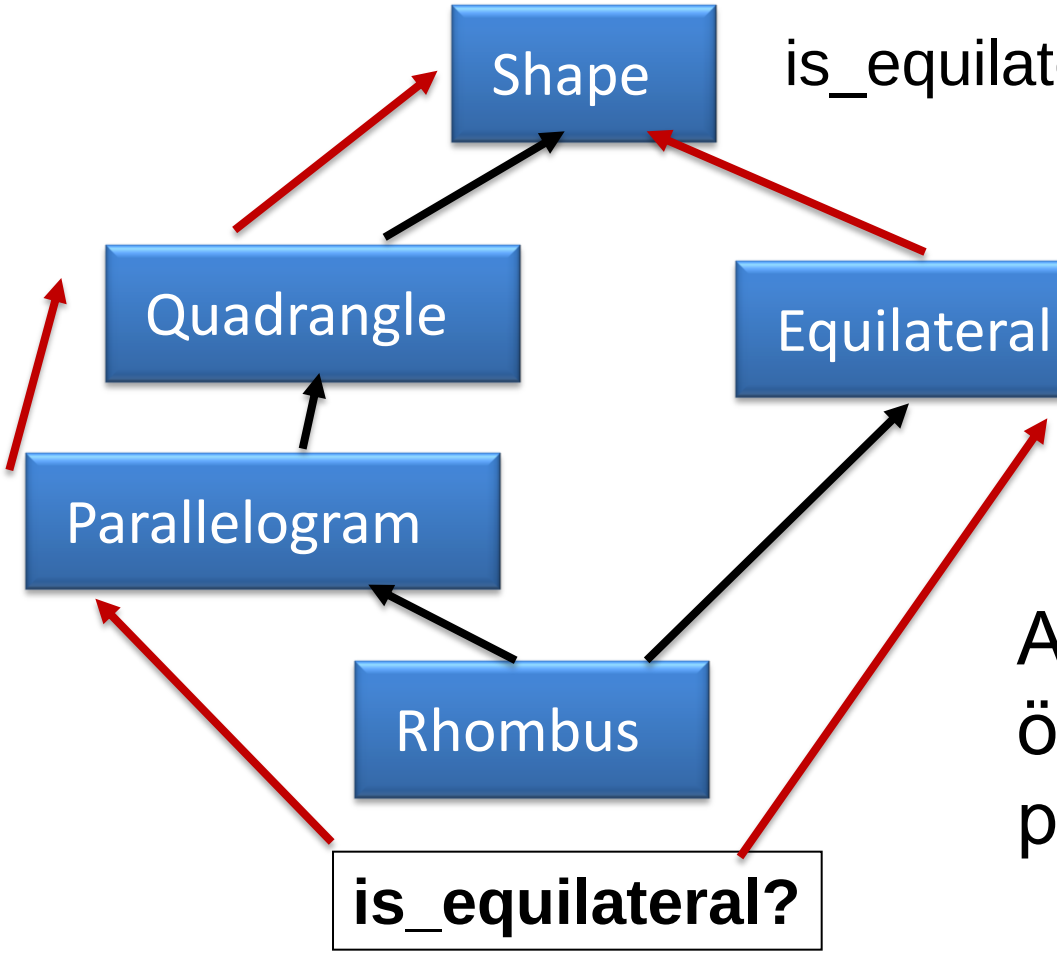
B. Változtassuk meg a Shape-t, a többiek öröklik az új attribútumot és metódust automatikusan

Adjuk hozzá az is_equilateral-t!

```
bool Shape::is_equilateral () {
  return false;
}

bool Equilateral::is_equilateral ()
  {
  return true;
}
```

Egy Rhombus egyenlőoldalú?




```
is_equilateral () { return false; }
```

```
is_equilateral () {
    return true;
}
```

A többszörös öröklődésnek lehetnek problémái!

Megoldások

- Java, Ada95, SmallTalk, C#, …
  - **Nem engedik**
    (Java, C#, stb.: interfészek a többszörös őstípusra, nem implementáció megosztás)
  - **Pro: biztonságos és egyszerű, Con: korlátozza az újrahasznosítást**
- Eiffel
  - **Explicit átnevezés vagy elrejtés (hiba, ha nem tesszük)**
- **C++**
  - **Explicit scope operátor szükséges**

még lesz róla szó…

Altípus reláció: $S \subseteq T$ biztonságos, ha:

Az altípus metódusai megőrzik az őstípus viselkedését:

- **Szignatúra:**
  - **Az argumentumok kontravarianciája, az eredmény kovarianciája**
  - **Az m_s kivételei benne vannak az m_T kivételei között**
- **Metódusok specifikációja:**
  - **előfeltétel**

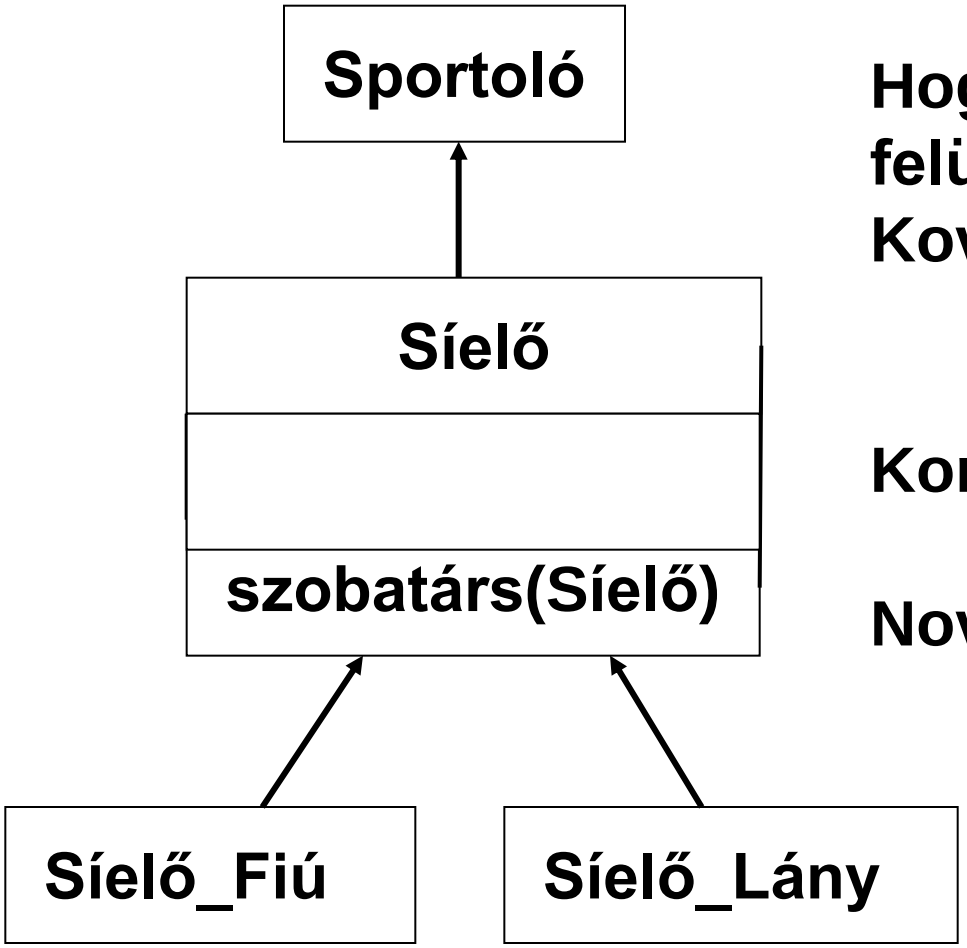
    “kontravariancia – altípus gyengébb”
  - **Utófeltétel**

    “kovariancia – altípus erősebb”

Az altípusok megőrzik a szupertípusok tulajdonságait – típusinvariánsukra:

“kovariancia – altípus erősebb”

Kontra/Ko-Variancia



Hogyan tudja Síelő_Lány felüldefiniálni szobatárs-at?

Kovariancia esetén:

szobatárs (Síelő_Lány) /
szobatárs (Síelő_Fiú)

Kontravariancia esetén:

szobatárs (Sportoló)

Novariancia esetén:

szobatárs (Síelő)

Probléma (kovariáns esetben):

s: Sielo; g: Sielo_Lany; b: Sielo_Fiu;
s := g; ... s. szobatárs (b); ☹

Mit tesz pl. a C++?

- Lehet bevezetni kovariáns metódusokat az altípusban, de ezek túlterhelik az eredeti metódust, nem átdefiniálják!
- Példa:

```
class sielo {
    public:
        virtual void szobatars(sielo * s){
         cout<<"\n sielo szobatarsa sielo \n";
        };
   };
```

```
class sielo_lany : public sielo {
  public:
     virtual void szobatars (sielo_lany *g){
       cout<<"\n sielo_lany szobatarsa lany ";
    };                          // túlterheli!
     virtual void szobatars (sielo *g){
       cout<<"\n szobatars sielo_lanyban” ;
    };     // átdefiniálja!
};
class sielo_fiu : public sielo {  … //hasonlóan
  }
```

```
void main(){
  sielo *s;
  sielo_lany *g;
  sielo_fiu *b;
  g = new sielo_lany;
  s = g;
  s->szobatars (b);   //  szobatars sielo_lanyban
                      // az átdefiniált
  g->szobatars (b);  //  szobatars sielo_lanyban
                  // ez is
  g->szobatars (g); // sielo_lany szobatarsa lany
                    // ez a túlterhelt
  s->szobatars (g); // szobatars sielo_lanyban (!)
                    // ez az átdefiniált
}
```

Mit tesz az Eiffel?

- Lehet bevezetni kovariáns metódusokat az altípusban
- Példa:

```
class Sielo
  feature  szobatars (Sielo s)…
end
class Sielo_Lany inherit
  Sielo
    redefine szobatars
  end
 feature  szobatars (Sielo_Lany g)…
end
```

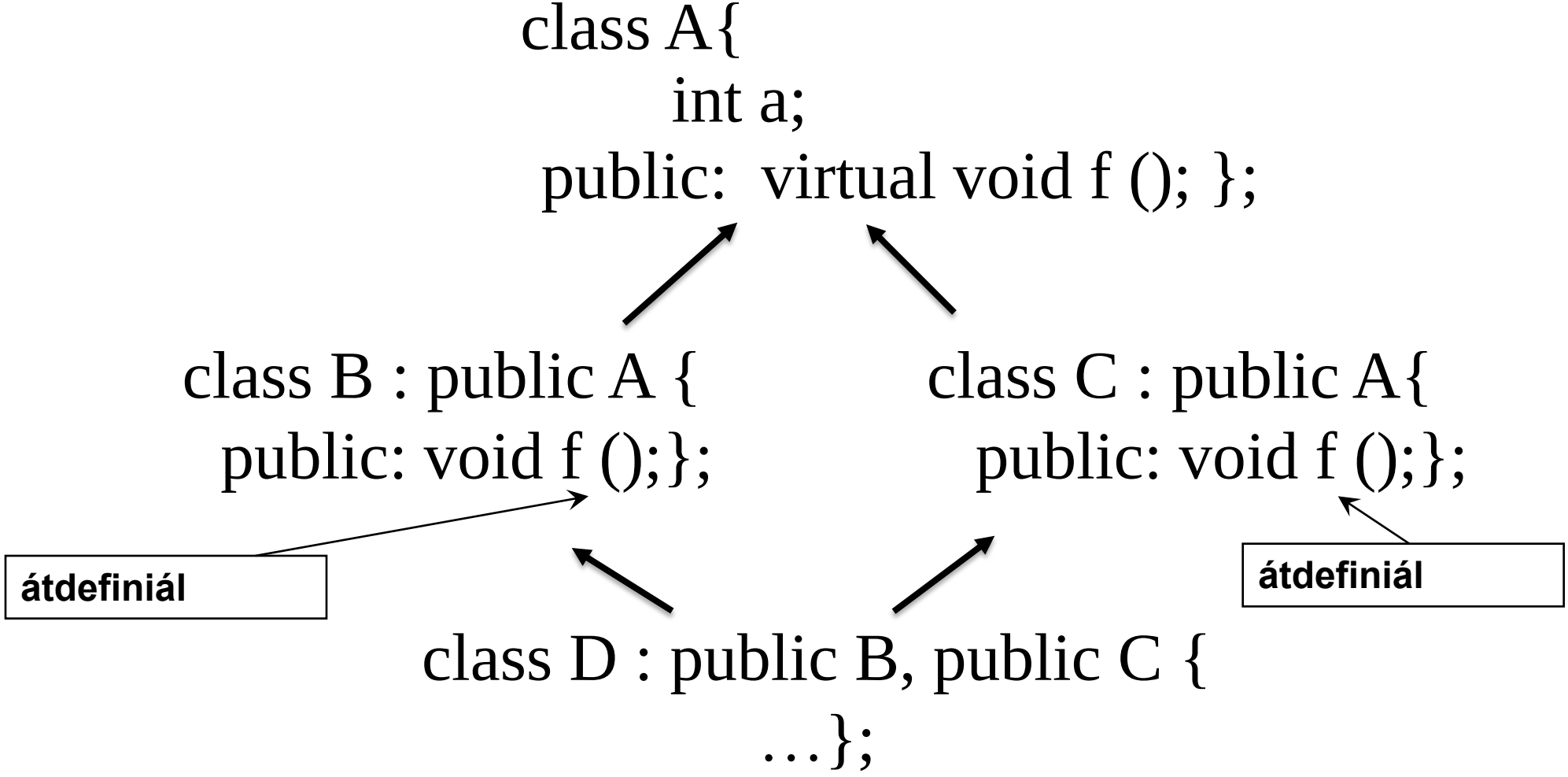
Nem típusbiztos!

```
class Sielo
    feature  szobatars (like Current)…
  end
```

Többszörös öröklődés

- Egy osztálynak egynél több közvetlen őse lehet
- Problémák:
  - Adattagok hányszor?
  - Melyik metódus?

A többszörös öröklődés problémái:



A többszörös öröklődés problémái:

1. Ha egy D-beli ‘f’-re (felüldefiniáltuk B-ben és/vagy C-ben) hivatkozunk, akkor az melyiket jelentse? (A v. B v. C)
2. Az ‘a’ attribútum hány példányban jelenjen meg D-ben?

A két kérdés lényegében ugyanazt a problémát veti fel: ha kétértelműség van, hogyan válasszunk?

Megoldási variációk:

- A legtöbb esetben az ilyen kódot nem lehet lefordítani, a fordító, vagy a futtató környezet kétértelműségre (*ambiguous*) hivatkozva hibajelzéssel leáll.
- Az ősosztály mondja meg, hogy mit szeretne tenni ilyen esetben.
- A származtatott osztály mondja meg, hogy melyiket szeretné használni.

C++

- „megoldás”:

```
D d; ….
     d.f();
```

- #error C2385: 'D::f' is ambiguous
-

- vagy:

```
class D : public B, public C
{
public:
  using C::f;
};
```

vagy:

A a B és C virtuális bázisosztálya kell legyen ⇒

a –t (minden adattagot) csak egyszer örökli jön később!

Eiffel

Rename (átnevezés):

```
class D inherit
    C
    rename a as ac
     end
    B
 feature
 ….
end
```

Eiffel

Select:

```
class A ....
  feature
    f...
    ....
end;
```

```
class B inherit
   A
   redefine
      f.....
   end;
feature
   f.....
.........
end;
```

```
class C inherit
     A
     redefine
       f.....
     end
feature
    f.....
.........
 end;
```

Eiffel (Select)

```
class D inherit
      B
        rename f as bf.....
      end;
      C
        rename f as cf
        select cf.....
          end
   feature
         .....
end;


a1:A;
d1:D;
.....
create d1; ….
 a1:=d1;
 a1.f;  -- melyik??
```

Absztrakt osztály

- Tervezés eszköze
- Egy felső szinten összefogja a közös tulajdonságokat
- A metódusok között **van olyan**, aminek csak specifikációja van, törzse nincs
- Nem hozható létre példánya.
- A leszármazott teszi konkréttá.

Interfész

- Típusspecifikáció támogatása
- „Szolgáltatások”
- Meg kell valósítani a szolgáltatásait
- Többszörös öröklődéssel nincs probléma
  - összefogja a közös jellemzőket
- Interfész ≠ Absztrakt osztály!

Interfész

```
public interface OperateCar {
   // constant declarations, if any
   // method signatures
   int turn(Direction direction,
         // An enum with values RIGHT, LEFT
         double radius, double startSpeed,
         double endSpeed);
  int signalTurn(Direction direction, boolean signalOn); int
   getRadarFront(double distanceToCar,
         double speedOfCar);
  int getRadarRear(double distanceToCar,
         double speedOfCar);
   ..... // more method signatures
   }
```

Interfész megvalósítása

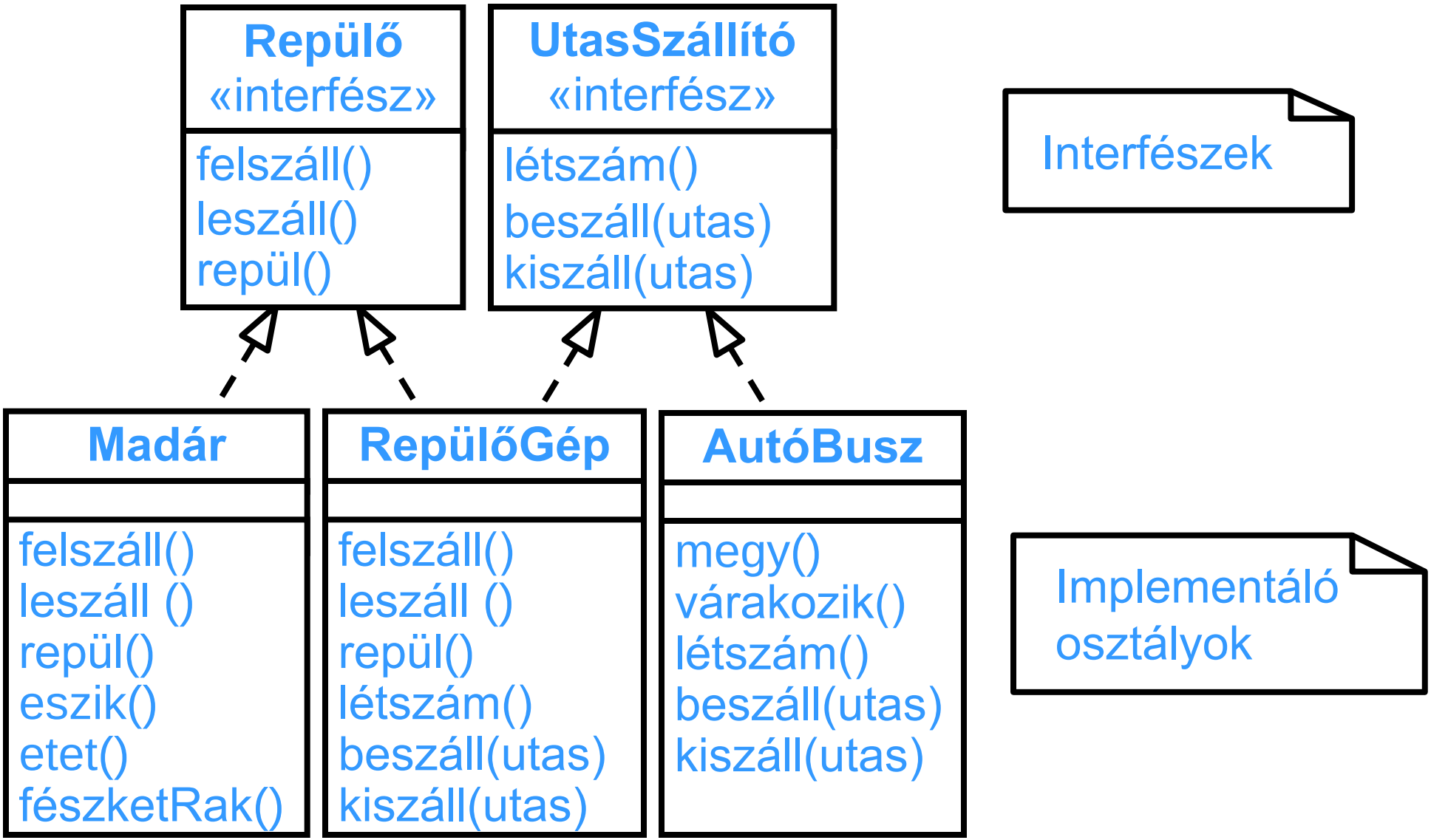
```
public class OperateBMW760i
        implements OperateCar {
         // az interfészbeli kódok megvalósítása

    .......

       // esetleges további kódok

     }
```

Interfész


Repülő
«interfész»
felszáll()
leszáll()
repül()
UtasSzállító
«interfész»
létszám()
beszáll(utas)
kiszáll(utas)
Interfészek
Madár
felszáll()
leszáll ()
repül()
eszik()
etet()
fészketRak()
RepülőGép
felszáll()
leszáll ()
repül()
létszám()
beszáll(utas)
kiszáll(utas)
AutóBusz
megy()
várakozik()
létszám()
beszáll(utas)
kiszáll(utas)
Implementáló
osztályok