

Objektumorientált programozás

2. előadás

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

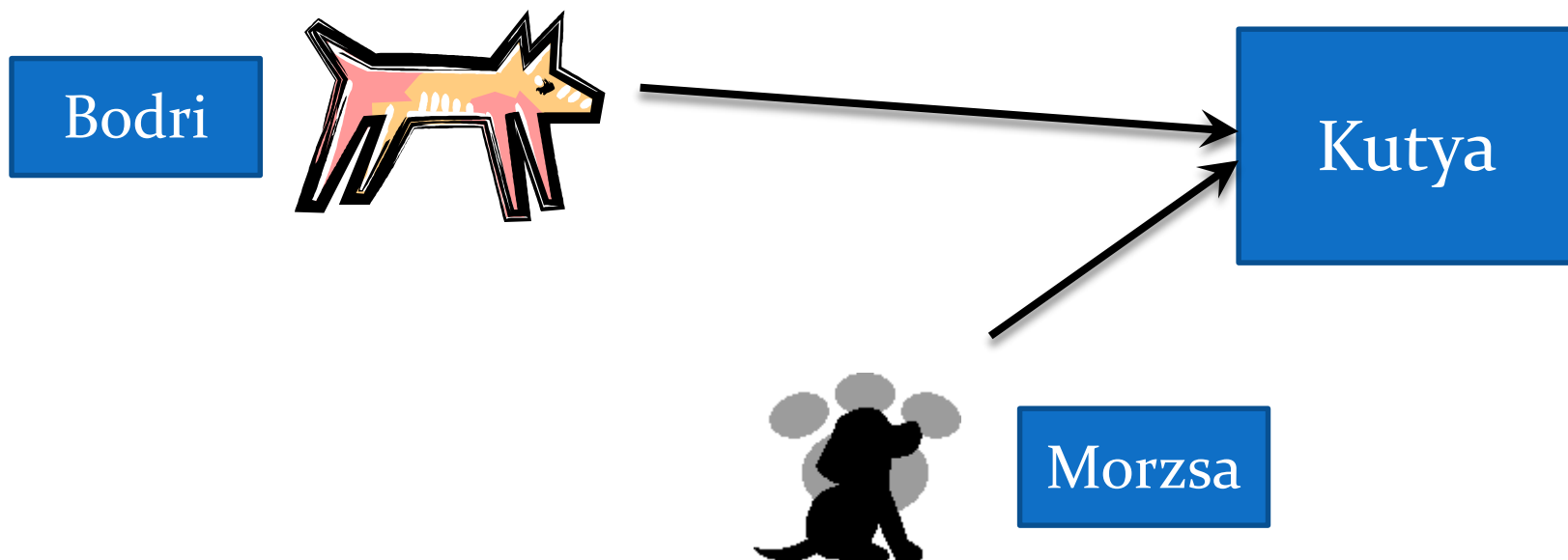
- Megkülönböztetés
 - Az objektumok a modellezendő valós világ egy-egy önálló egységét jelölik.
 - Az objektumokat a számunkra lényeges tulajdonságaik, viselkedési módjuk alapján megkülönböztetjük.

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 mintáinak tekinthetők.

A valós világ modellezése

- Osztályozás



A valós világ modellezése

- Osztályozás

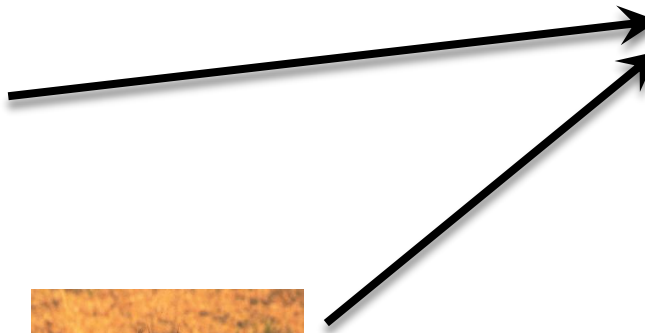
Simba



Leó



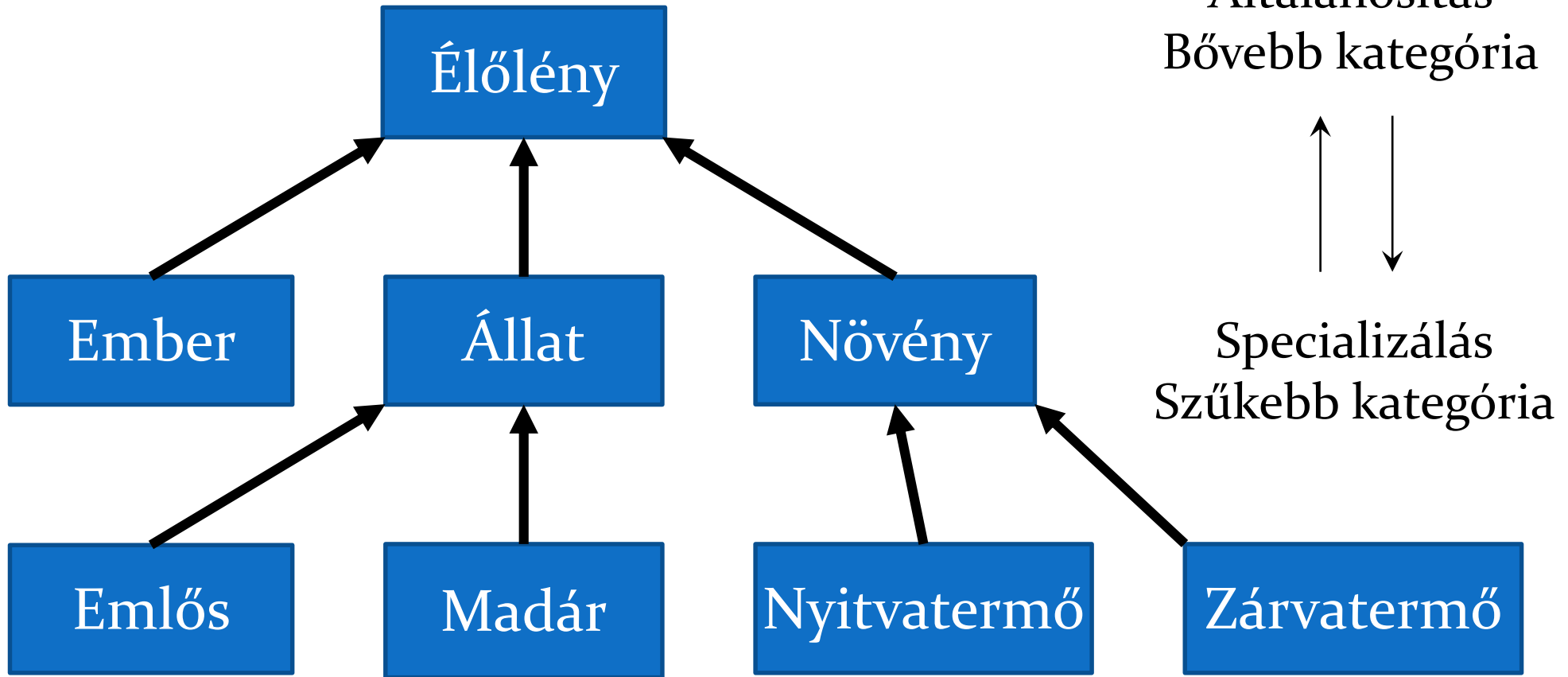
Oroszlán



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



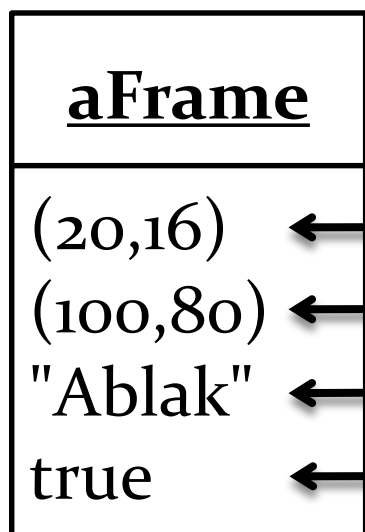
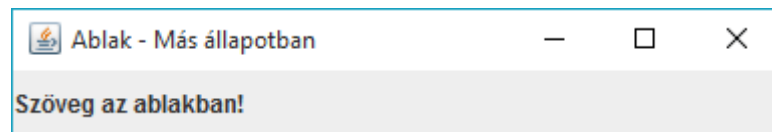
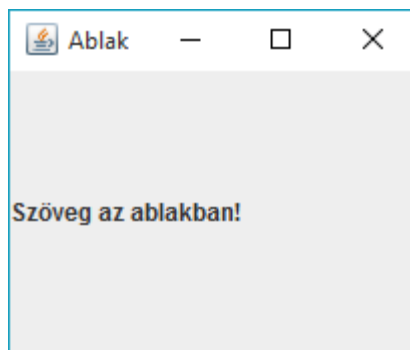
Objektum

- 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

Üzenetek ↘

`setVisible(true)` →
`setLocation(40,8)` →
`setSize(20,16)` →
`setTitle("Ablak")` →



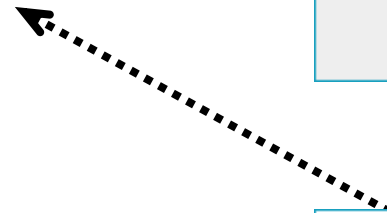
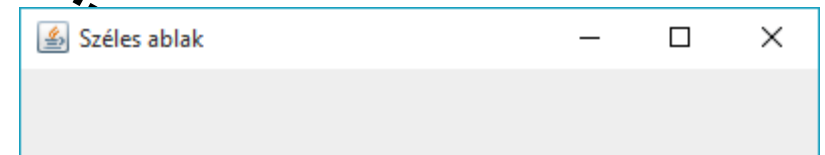
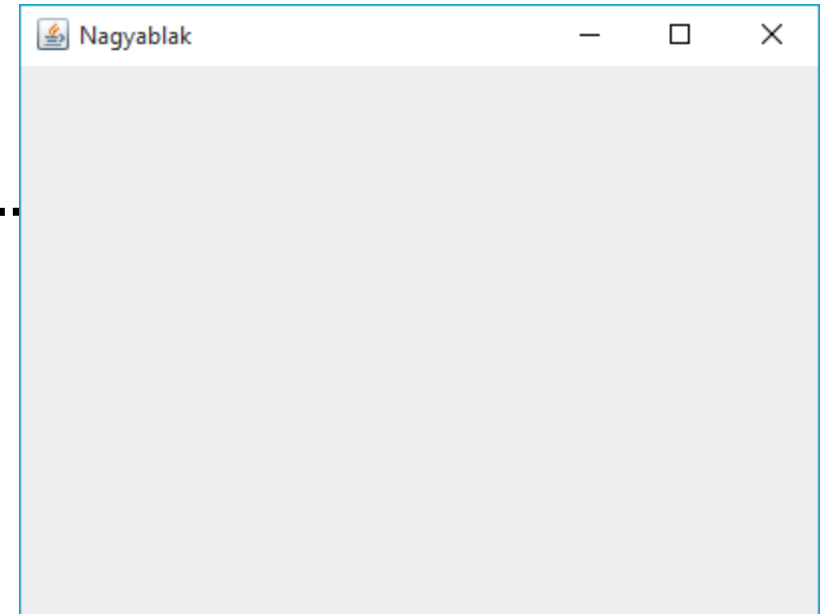
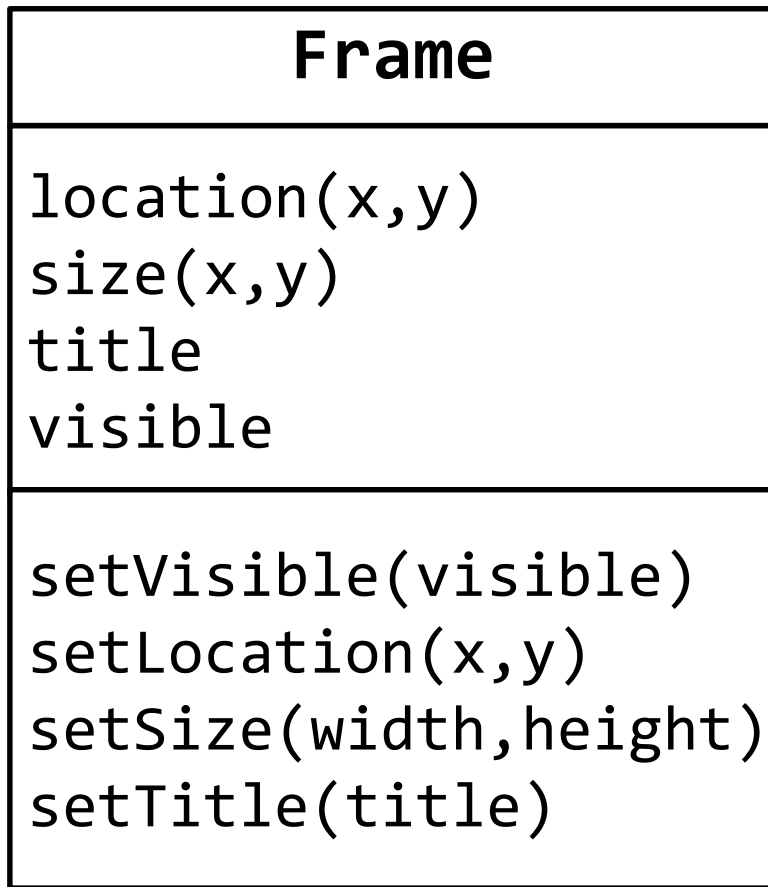
Adattagok
(attribútumok)

location(x,y)
size(width,height)
title
visible

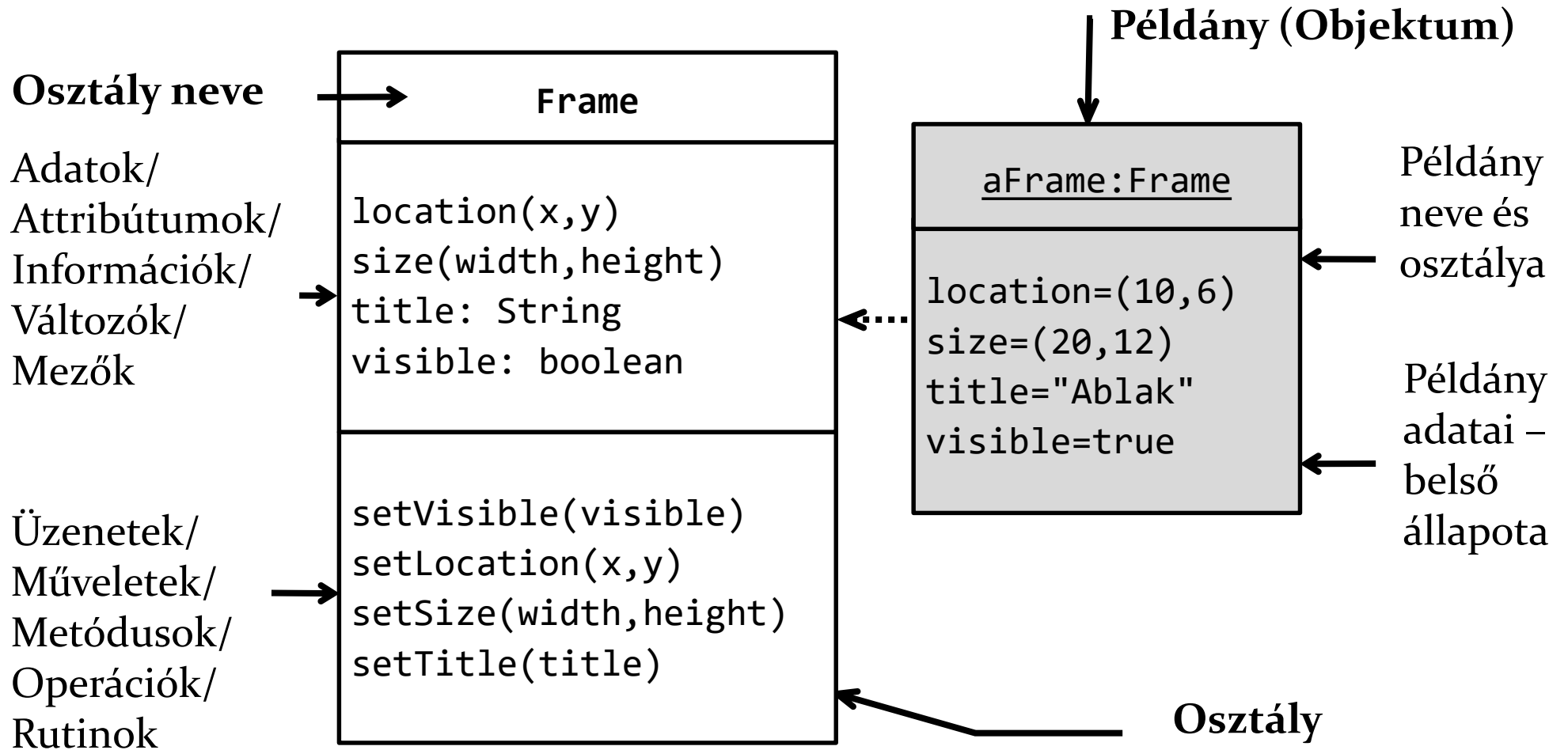
Osztály, példány

- Osztály (class)
 - Olyan objektumminta vagy típus, mely alapján példányokat (objektumokat) hozhatunk létre
- Példány (instance)
 - Egy osztály (minta) alapján létrejött konkrét példány
 - Minden objektum születésétől kezdve egy osztályhoz tartozik

Frame osztály és példányai



Osztály és példány az UML-ben



Osztály és példány a C++-ban

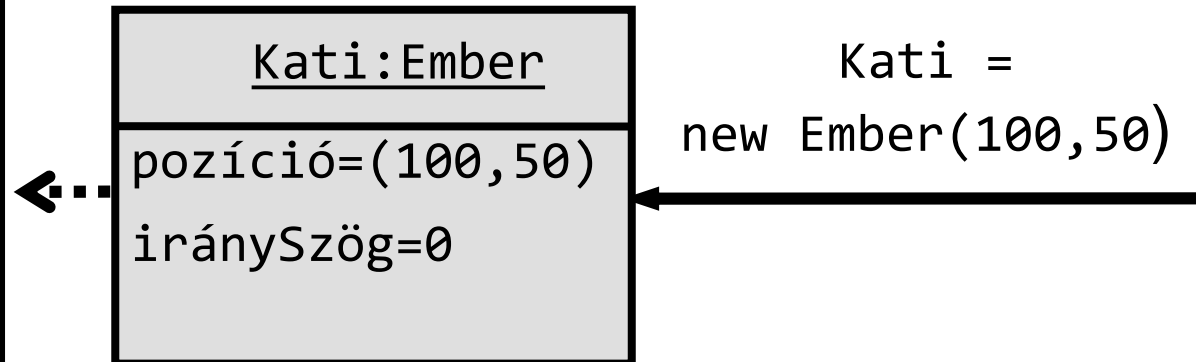
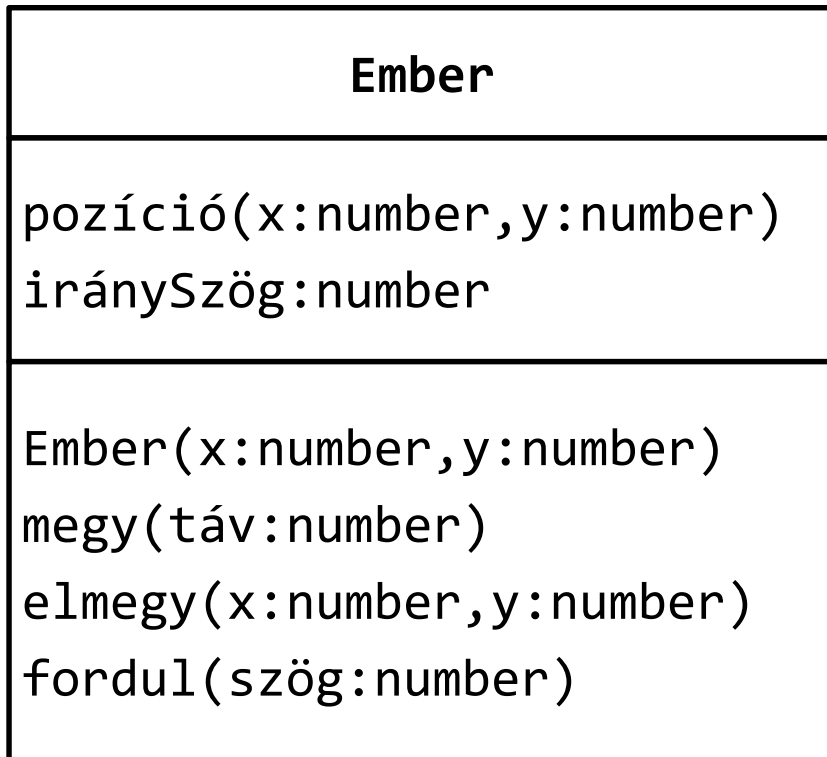
```
class Employee {
    string first_name, family_name;
    short department;
    // az adattagok alapértelmezésben privát hozzáférésűek!
public:
    void print() const { //... }
    string name() const { //... }
}

...
Employee empl;
Employee * emplp;
```

Objektum létrehozása, inicializálása

- Objektum élelciklusa:
„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 létrehozása, inicializálása C++-ban

```
class Employee {  
    string first_name, family_name;  
    short department; //...  
public:  
    Employee (const string& f, const string& n, short d):  
        first_name(f), family_name(n), department(d){}  
//...  
}
```

C++

- Konstruktorok: „nincs objektum konstruktor nélkül”, ha kell, implicit hívódnak
 - Neve megegyezik az osztály nevével
 - Ha már megadtunk egy konstruktort, akkor default konstruktor nem definiálódik
 - A default konstruktor meghívja az attribútumok konstruktorát, de a beépített típusokat nem inicializálja (konzisztensen a C-vel)
 - A konstruktornak nem lehet visszatérési értéke
- A destruktort is expliciten lehet hívni a delete operátorral, vagy implicit hívódik a blokkból való kilépéskor – fordított sorrendben

Objektum műveletei

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

Objektum műveletei

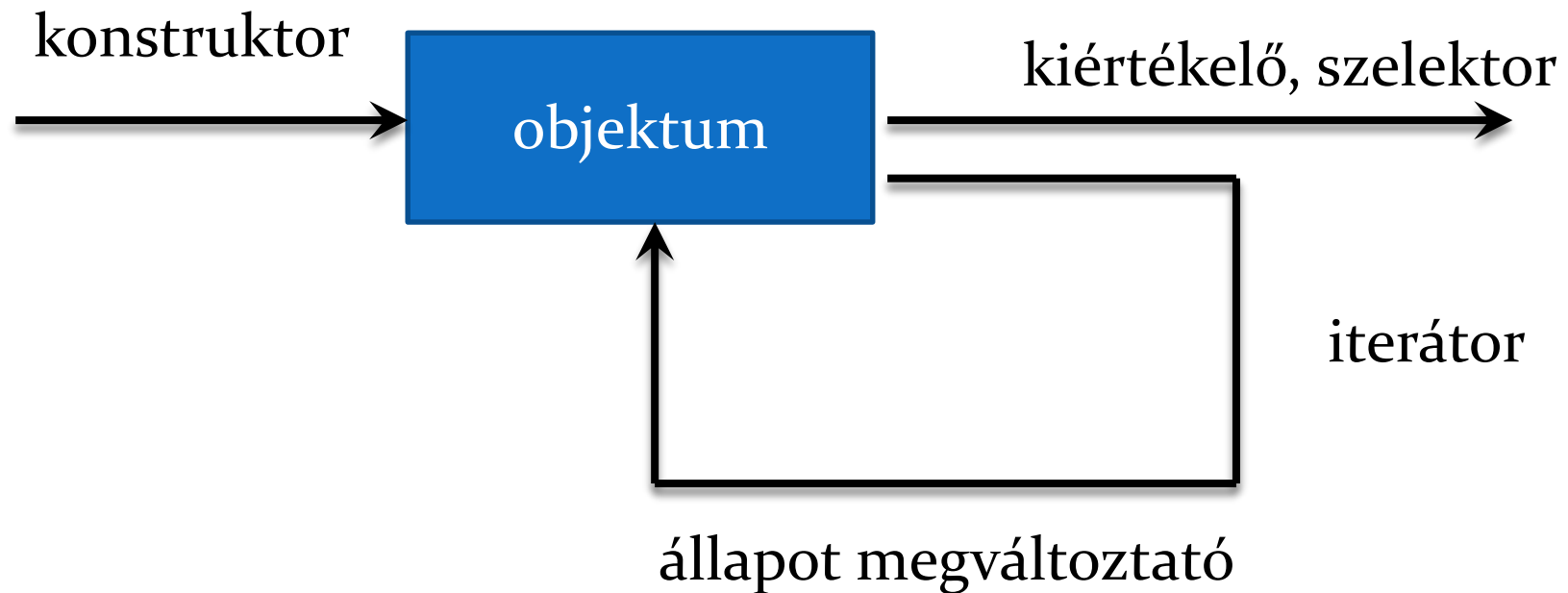
- Export műveletek csoportosítása:
 - Létrehozó (konstruktor)
az objektum létrehozására, felépítésére
 - Példa veremnél
 - create: \rightarrow Verem
 - Állapot megváltoztató
 - Példa verem esetén
 - pop: Verem \rightarrow Verem \times Elem
 - push: Verem \times Elem \rightarrow Verem

Objektum műveletei

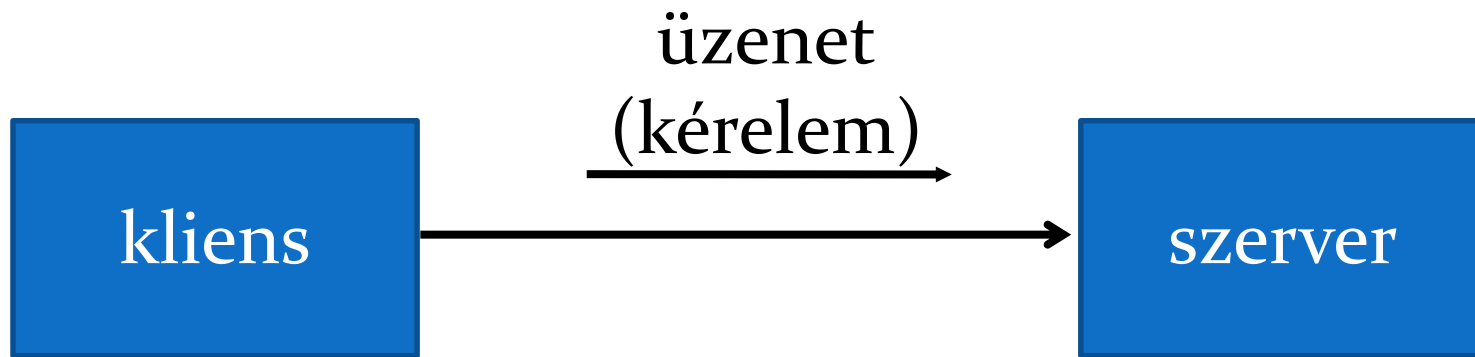
- Export műveletek csoportosítása:
 - Szelektor – kiemeli az objektum bizonyos részét
 - Például
 - vektor adott indexű elemét
 - access: Vektor \times Index \rightarrow Elem
 - Kiértékelő – objektum jellemzőit lekérdező műveletek (size, has, stb.)
 - Iterátor – bejáráshoz

Objektum műveletei

- Export műveletek csoportosítása:



Kliens üzen a szervernek



A feladatot
elvégeztető
objektum

Kívülről elérhető
metódus hívása

A feladatot
elvégző
objektum

Kliens üzen a szervernek

- Kliens
 - Aktív objektum, másik objektumon végez műveleteket, de rajta nem végeznek
 - Nincs export felülete
 - Például óra (órajel)
 - Meghatározott időközönként művelet egy regiszteren
- Szerver
 - Passzív objektum
 - Csak export felülete van
 - Másoktól érkező üzenetekre vár, mások szolgáltatását nem igényli
- 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 is objektum?

Osztály, példány

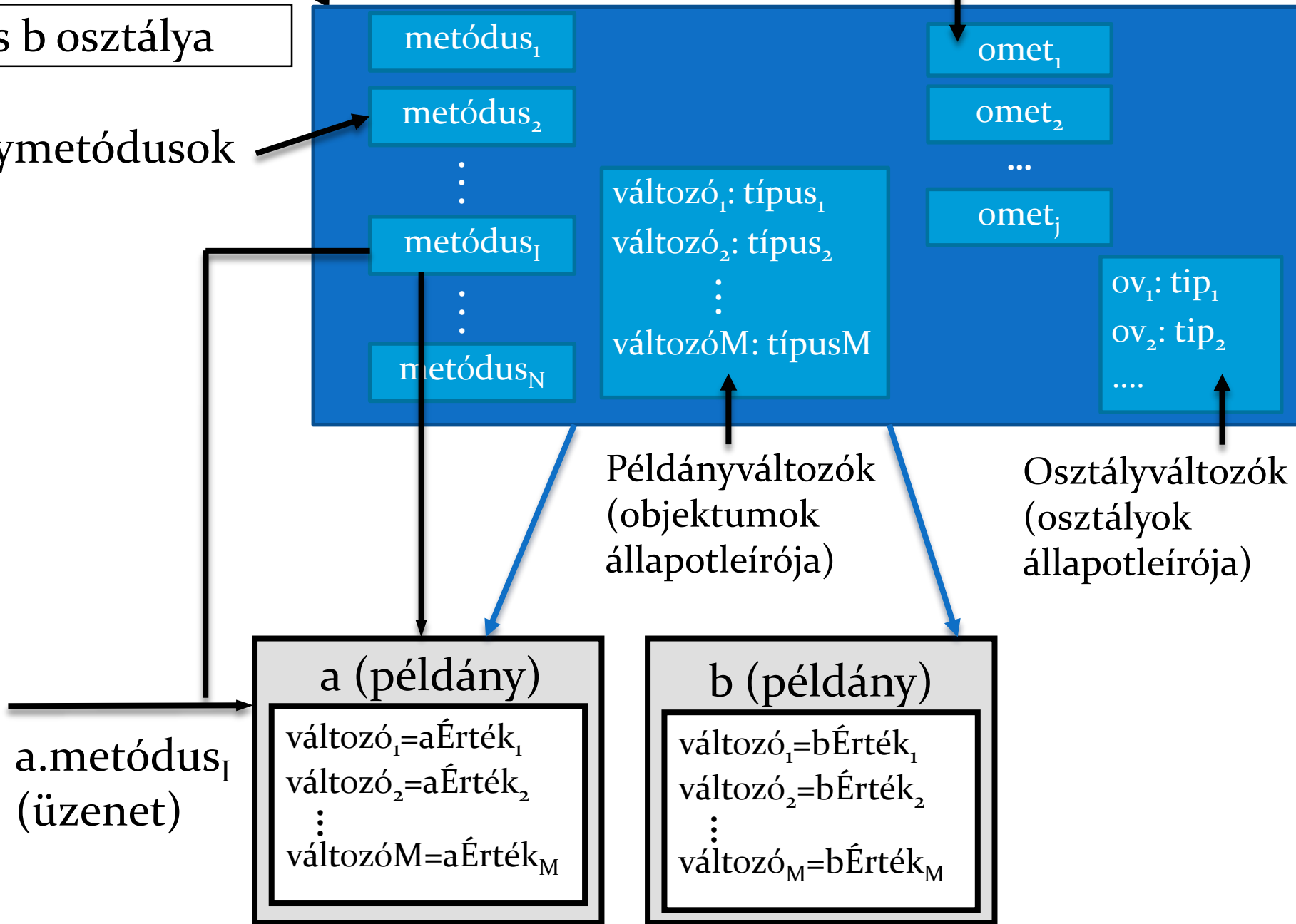
- Osztálydefiníció:
 - **Példányváltozó**
 - Példányonként helyet foglaló változó
 - **Példánymetódus**
 - Példányokon dolgozó metódus
 - **Osztályváltozó**
 - Osztályonként helyet foglaló változó
 - **Osztálymetódus**
 - Osztályokon dolgozó metódus

Osztálydefiníció

Osztálymetódusok

a és b osztálya

Példánymetódusok



Osztályváltozó, osztálymetódus C++-ban

```
class Employee {  
    string first_name, family_name;  
    short department;  
    static int num_emp;  
    //...  
};  
int Employee::num_emp(0);
```

- Az osztályon kívül definiálni kell!

Osztályváltozó, osztálymetódus C++-ban

- Az osztályon belül elérhető, pl.:

```
class Employee {  
    string first_name, family_name;  
    short department;  
    static int num_emp;  
public:  
    Employee (const string& f, const string& n,  
short d):  
        first_name(f), family_name(n),  
department(d){num_emp++;}  
};
```

Osztályváltozó, osztálymetódus C++-ban

- Kívülről hozzáférni csak public metódussal lehet:

```
class Employee {  
    string first_name, family_name;  
    short department;  
    static int num_emp;  
public:  
    static int get_num_emp(){ return num_emp;}  
    static void print_num_emp(){  
        cout << "Az objektumok szama:" << num_emp  
        << '\n';  
    };  
};
```

Osztályváltozó, osztálymetódus C++-ban

- Kívülről hozzáférni csak public metódussal lehet:

```
int main()
{
    Employee emp ("Kati", "Fekete", 3);
    emp.print_num_emp();
    // vagy:
    Employee::print_num_emp();
}
```


Osztályváltozó, osztálymetódus C++-ban

```
class Datum{
    int nap, ho, ev;
    static Datum alapert_datum;
public:
    Datum(int nn=0, int hh=0, int ee=0); //...
    static void beallit_alapert(int,int,int);
}
```

- Mire jó? Pl., ha paraméter nélküli konstruktor hívás történik, akkor az alapert_datum értéket kapja meg az új objektum.
- Az előző konstruktor helyett:

```
Datum::Datum(int nn, int hh, int ee){
    nap = nn ? nn : alapert_datum.nap;
    ho = hh ? hh : alapert_datum.ho;
    ev = ee ? ee : alapert_datum.ev;
}
```

Osztályváltozó, osztálymetódus C++-ban

```
void Datum::beallit_alapert(int n, int h, int e){  
    //a statikus adattag értékének megváltoztatása  
    Datum::alapert_datum = Datum(n,h,e);  
}
```

- Definiálni kell, mielőtt használjuk!

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)

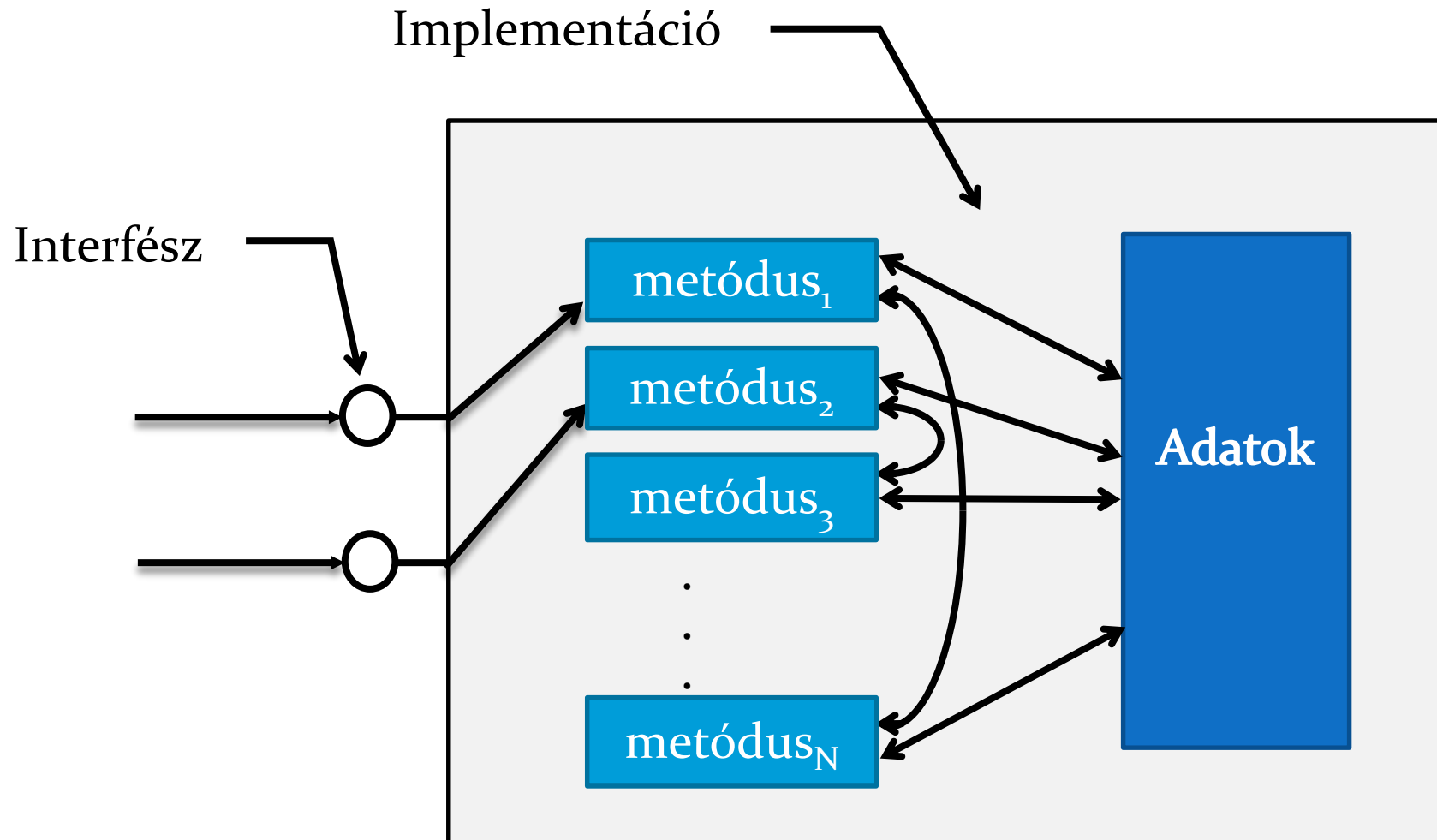
A „this”

- 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

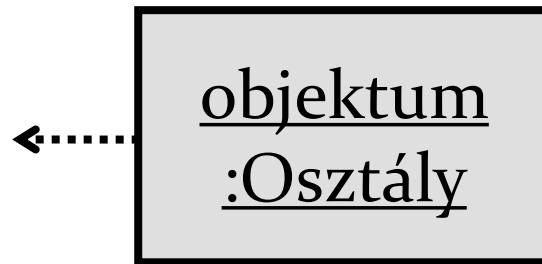
- Bezárás (encapsulation)
 - Adatok és metódusok összezárása
 - Egybezárás, egységbezárás - osztály (class)
- Információ elrejtése (information hiding)
 - Az objektum „belügyeit” csak az interfészen keresztül lehet megközelíteni (láthatóságok!)
- Kód újrafelhasználása (code reuse)
 - Megírt kód felhasználása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

Információ elrejtése



Láthatóság – Objektum védelme

Osztály
+publikusAdat #védettAdat -privátAdat
+publikusMetódus #védettMetódus -privátMetódus

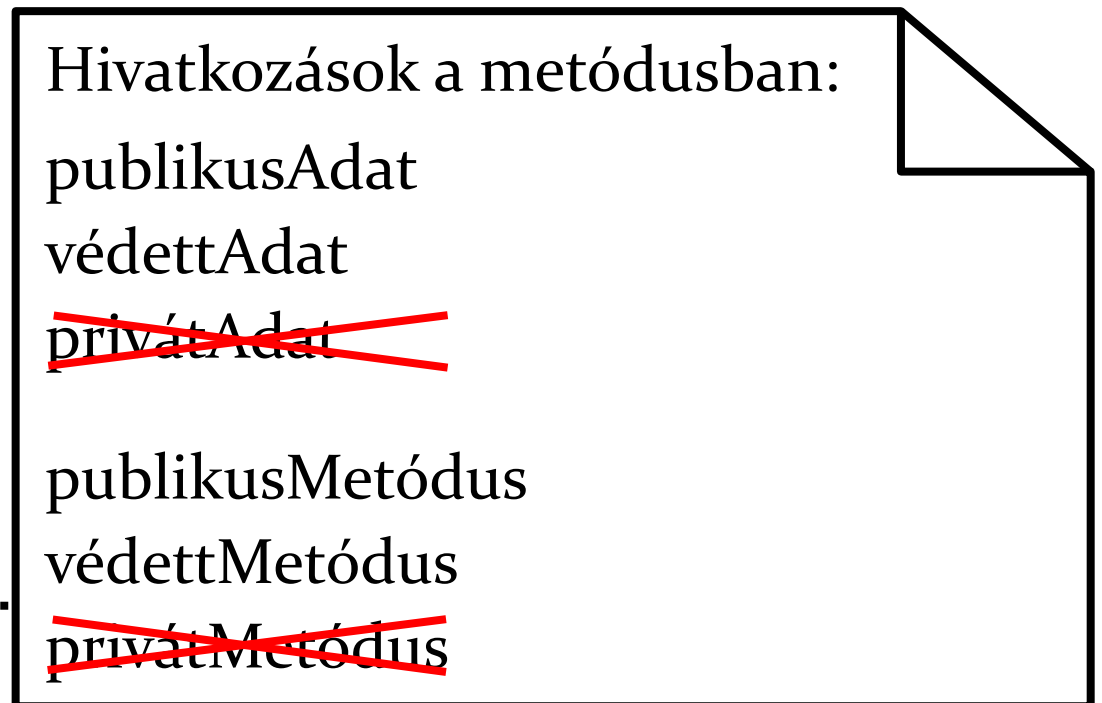
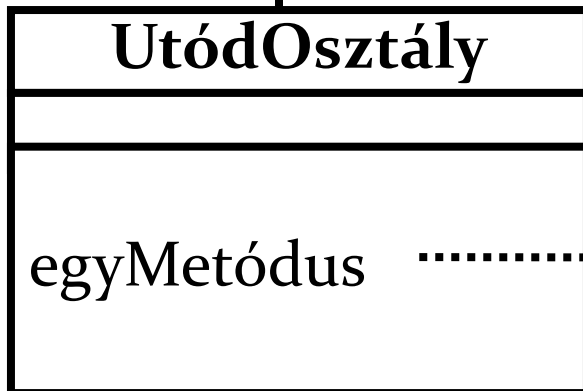
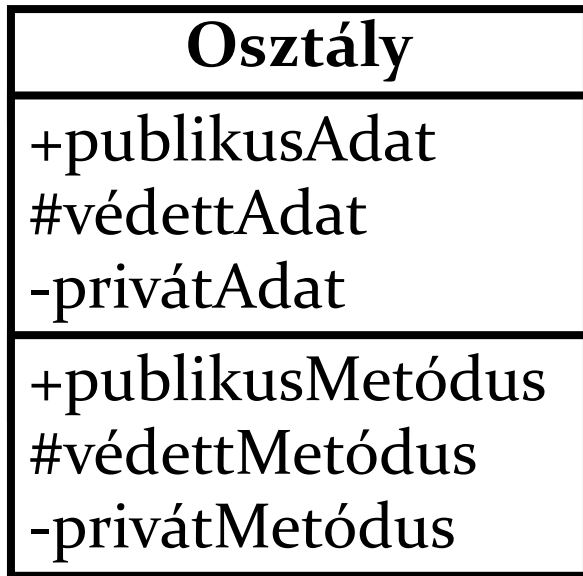


objektum.publikusAdat
objektum.publikusMetódus



~~objektum.védettAdat
objektum.védettMetódus
objektum.privátAdat
objektum.privátMetódus~~

Osztály védelme

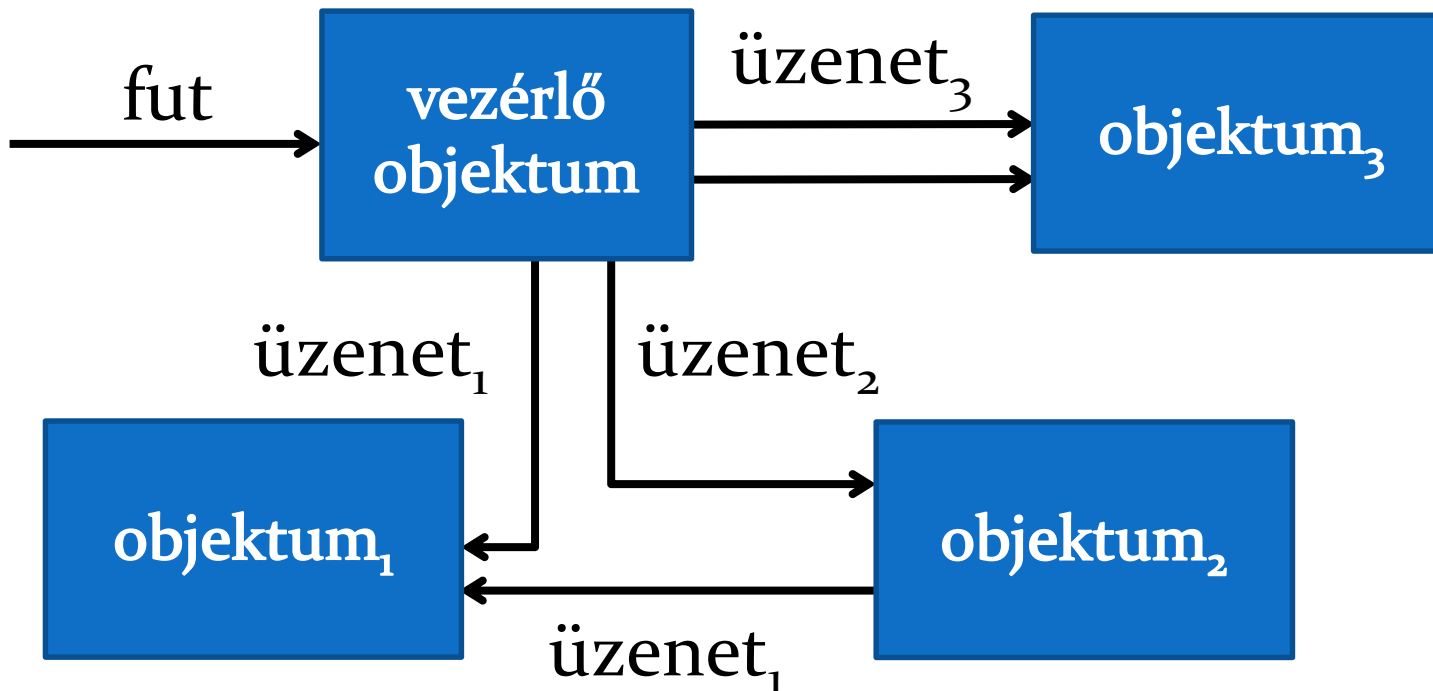


C++

- Az adattagok és metódusok elrejtése megoldott
- A láthatóság minősítője lehet
 - public
 - a külső felhasználók elérik
 - protected
 - csak a leszármazottak érhetik el
 - private
 - csak az adott osztály és „barátai” számára elérhető – alapértelmezés az osztályoknál

OO program

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



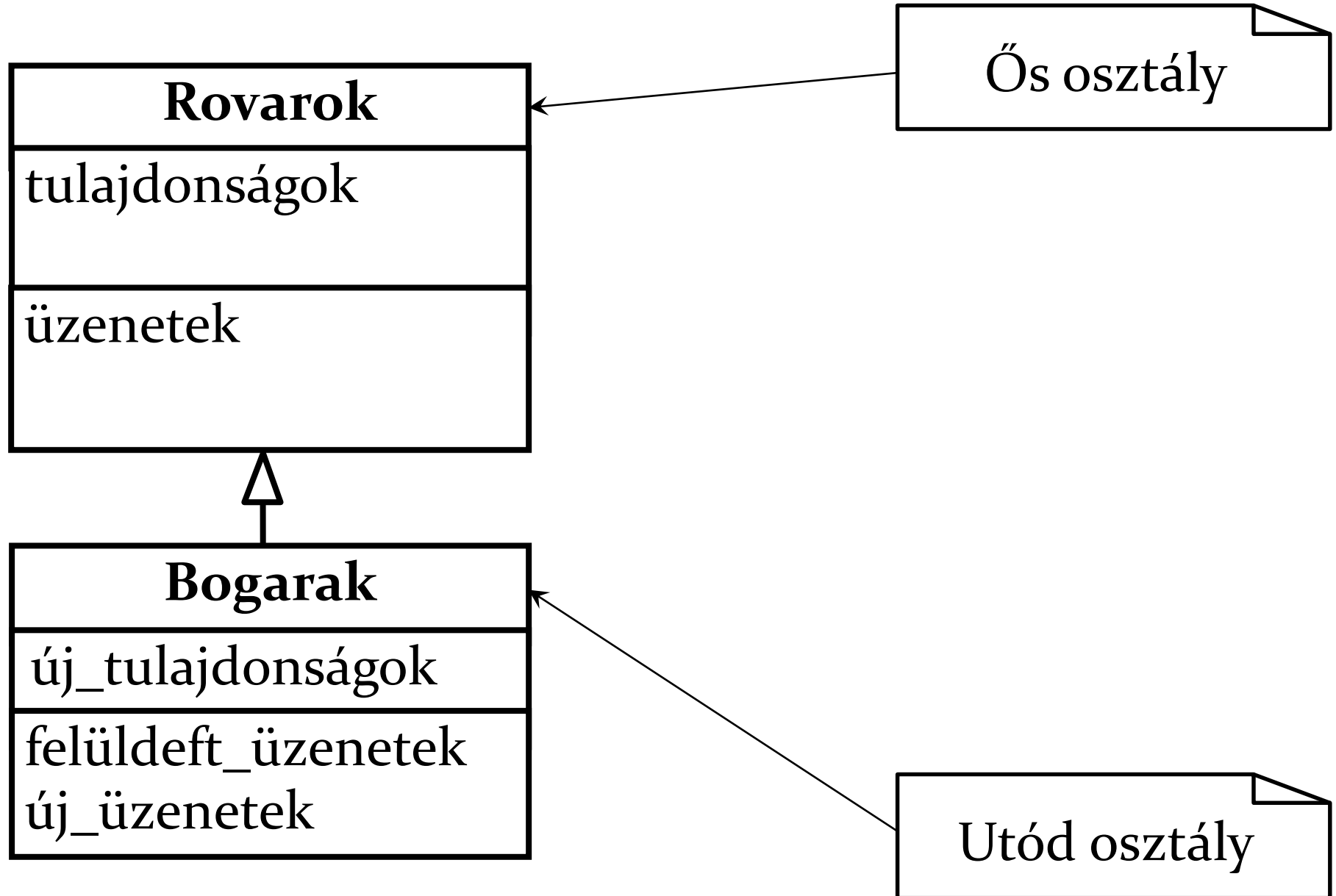
Öröklődés

- Az alapgondolat: a gyerekek öröklik ősük metódusait és változóit
- Az *örököl* terminus azt jelenti, hogy az ősosztály **minden** metódusa és adattagja a gyerekosztálynak is metódusa és adattagja lesz

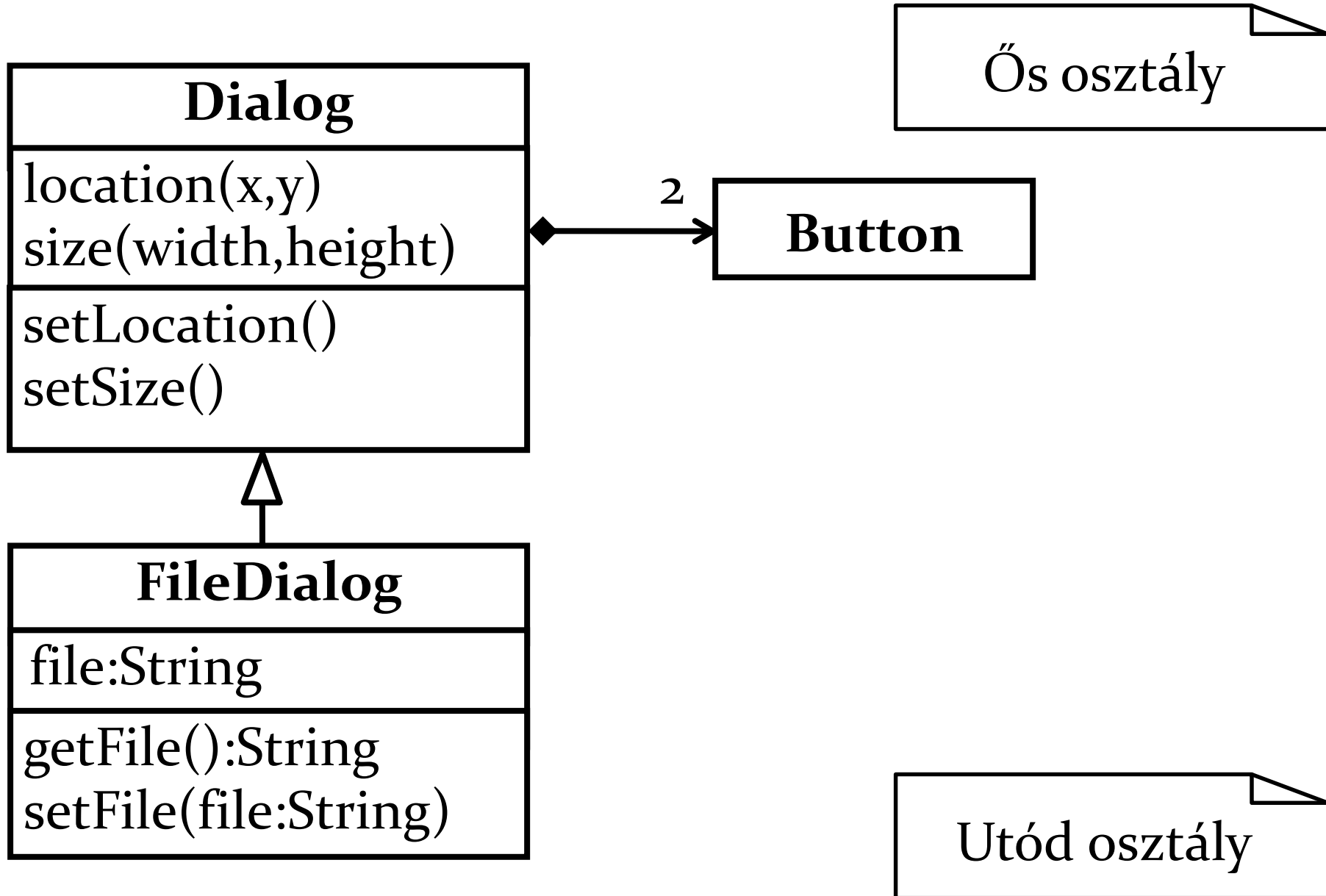
Öröklődés

- Mit tehet a leszármazott?
- Bevezethet új műveleteket és adattagokat
 - Ezek egyszerűen hozzáadódnak az örökölt metódusokhoz és adattagokhoz.
- Változtathat az örökölt viselkedésén - átdefiniálhatja az örökölt metódusokat, ezek a hierarchiában felülbírálják az örökölt metódust.

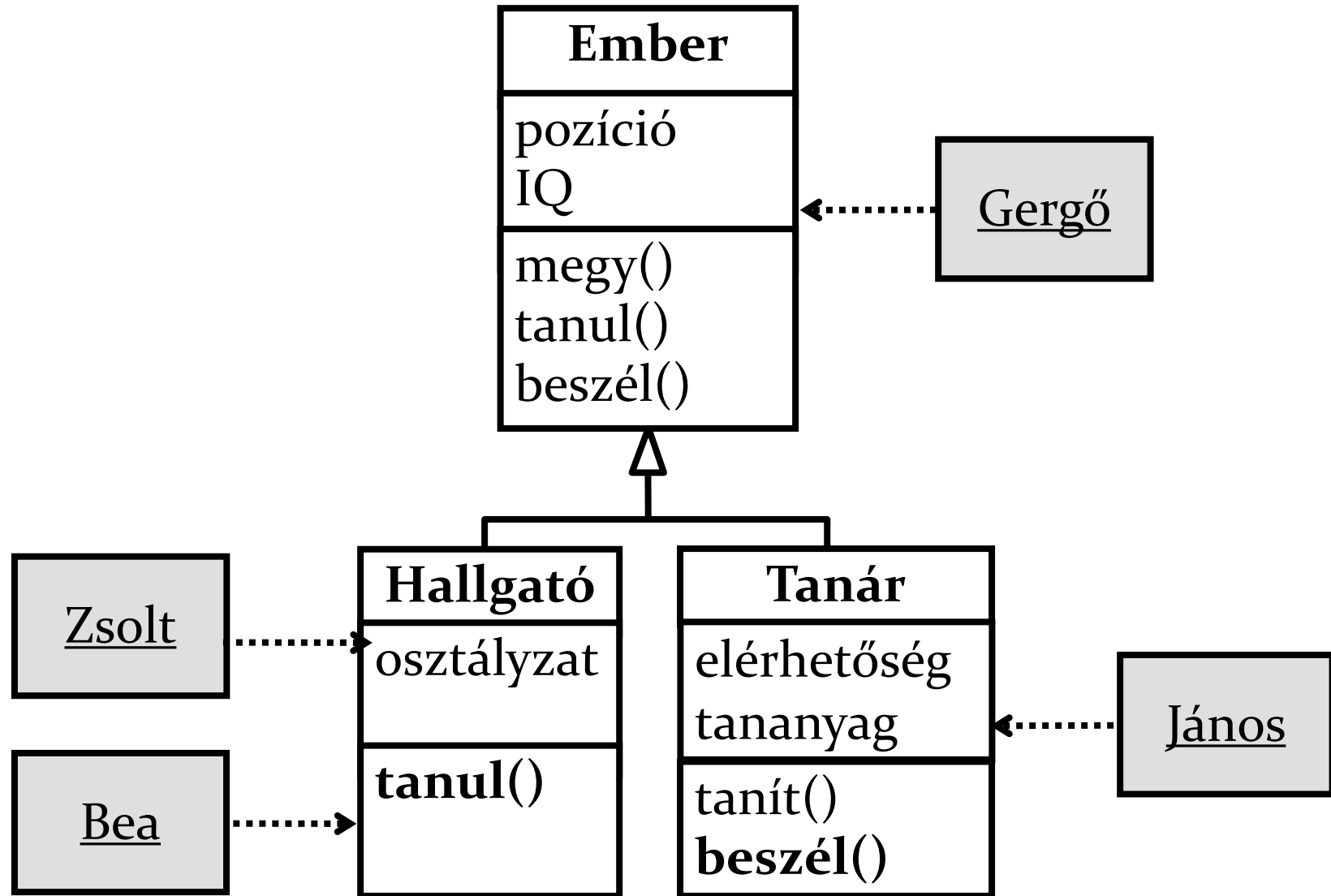
Öröklődés – IS-A reláció



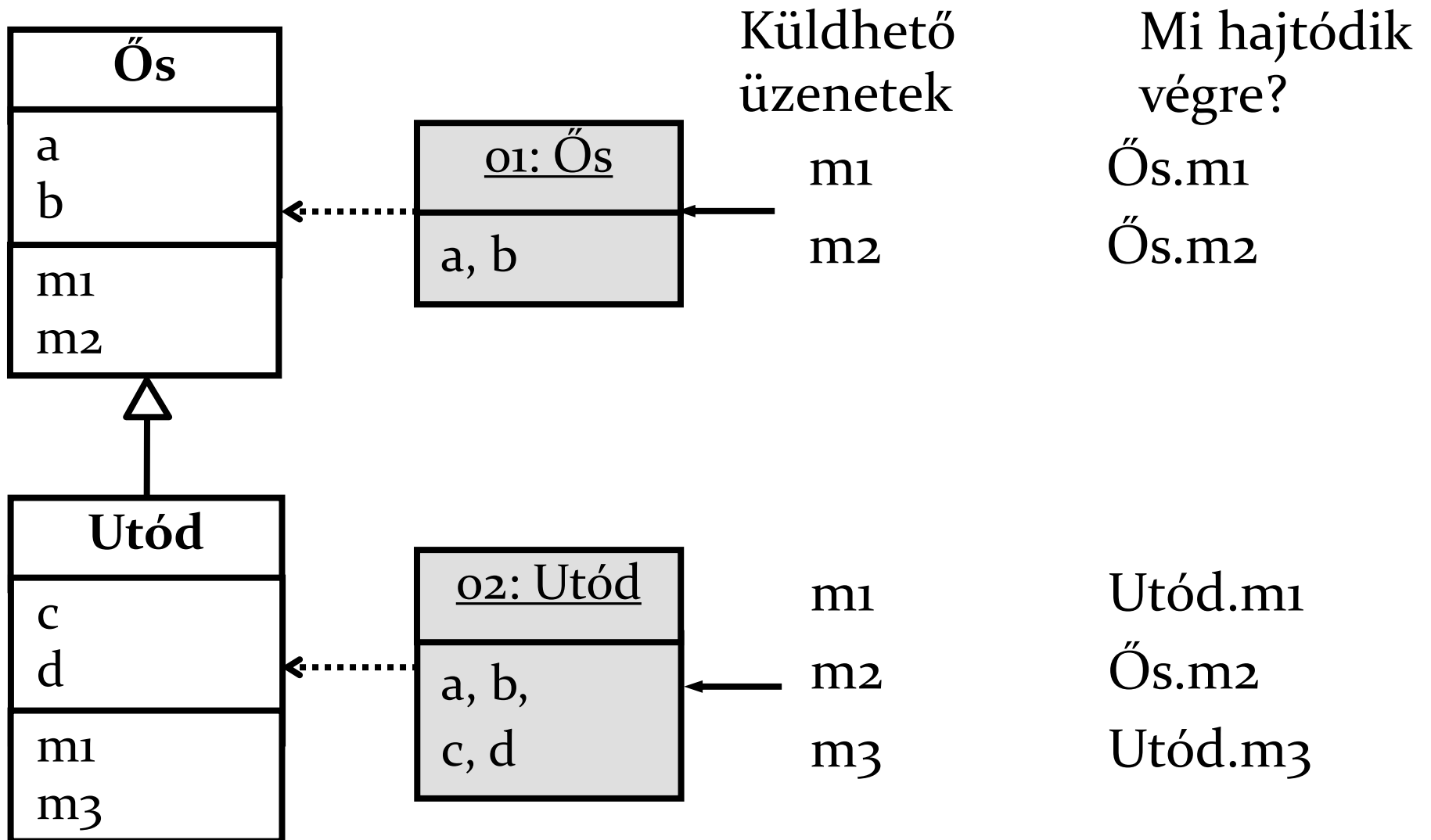
Öröklődés



Ki, mire képes?



Utód adatai, küldhető üzenetek



C++ példa az öröklődésre:

```
class Employee {
    string first_name, family_name;
    short department;
    static int num_emp;
public:
    Employee (const string& f, const string& n, short d):
        first_name(f), family_name(n), department(d)
        {num_emp++;}
    void print() const {
        cout << "A vezeteknev:" << name() << '\n';
    }
    string name() const { return family_name; }
```

C++ példa az öröklődésre:

```
static int get_num_emp(){
    return num_emp;
}
static void print_num_emp(){
    cout << "Az objektumok szama:" << num_emp << '\n';
}
//...
};
```

Példa folytatása

```
class Manager: public Employee {
    Employee* group;
    short level;
public:
    Manager (const string& f, const string& n, short
d,    short lvl): Employee(f,n,d), level(lvl){};

    void print () const {
        cout << "A keresett nev:" << name() << '\n';
        cout << "A szint:" << level << '\n';
    }
};
```

A main

```
int main()
{
    Employee emp ("Kati", "Fekete", 3);
    Manager m("Jozsef", "Kovacs", 3, 2);
    emp.print ();
    m.print();
}
```

Polimorfizmus

- Polimorfizmus (többalakúság): 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.

C++:

- Tekintsük az alábbi kódot

```
Employee emp ("Kati", "Fekete", 3);
```

```
Manager m("Jozsef", "Kovacs", 3, 2);
```

```
emp = m;
```

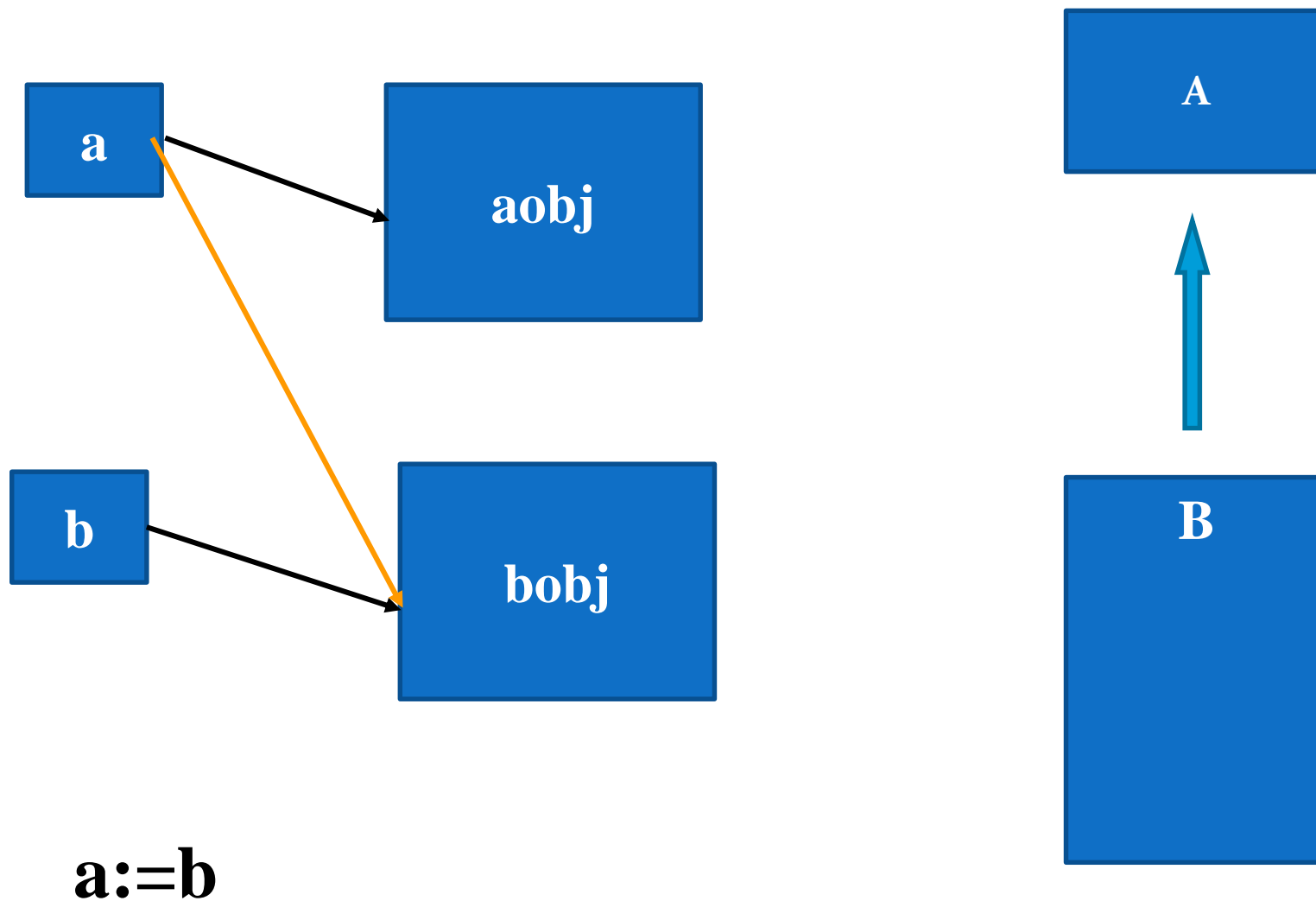
- Megengedett

```
emp.print();
```

```
m.print();
```

- Mi történik?

Hogy lenne jó?



Altípusos polimorfizmus

```
Employee* empp=new Employee ("Istvan", "Nagy",5);
```

```
...
```

```
Manager* mp=new Manager("Laszlo", "Hajto", 2, 3);
```

```
...
```

```
empp = mp;
```


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.

Altípusos polimorfizmus

```
Shape *a;
```

```
Triangle* h= new Triangle (...);
```

```
Rectangle *t= new Rectangle (...);
```

```
a = h; a->draw(); ...
```

```
a = t; a->draw(); ...
```

Melyik draw()?

Altípusos polimorfizmus

```
Employee* empp=new Employee ("Istvan", "Nagy",5);
```

```
...
```

```
Manager* mp=new Manager("Laszlo", "Hajto",2,3);
```

```
...
```

```
empp = mp;
```

```
...
```

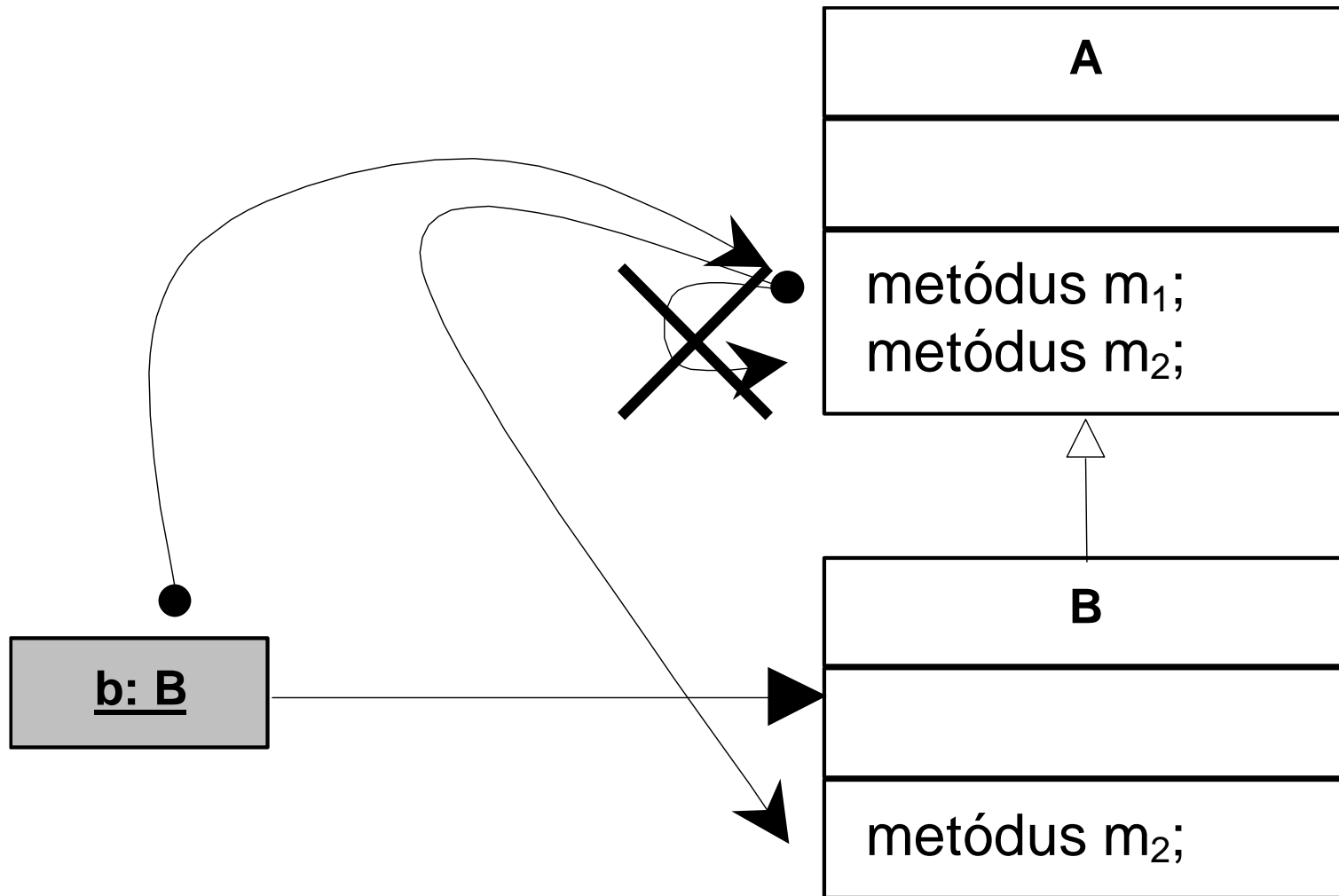
```
empp->print();
```

Melyik print?

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



Dinamikus összekapcsolás – C++-ban:

```
class Employee {  
    ...  
    public:  
    ...  
    virtual  
    void print() const {  
        cout << "A vezeteknev:" << name() << '\n';//...  
    }  
    ...  
}
```

Mi az objektumorientált programozás?

- A programozó definiálhat altípus kapcsolatokat
- A típuszabályok 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
- Más programozási nyelveknél ez másképp lehet
 - Erről a programozási nyelvek és módszerek tárgyon beszélünk

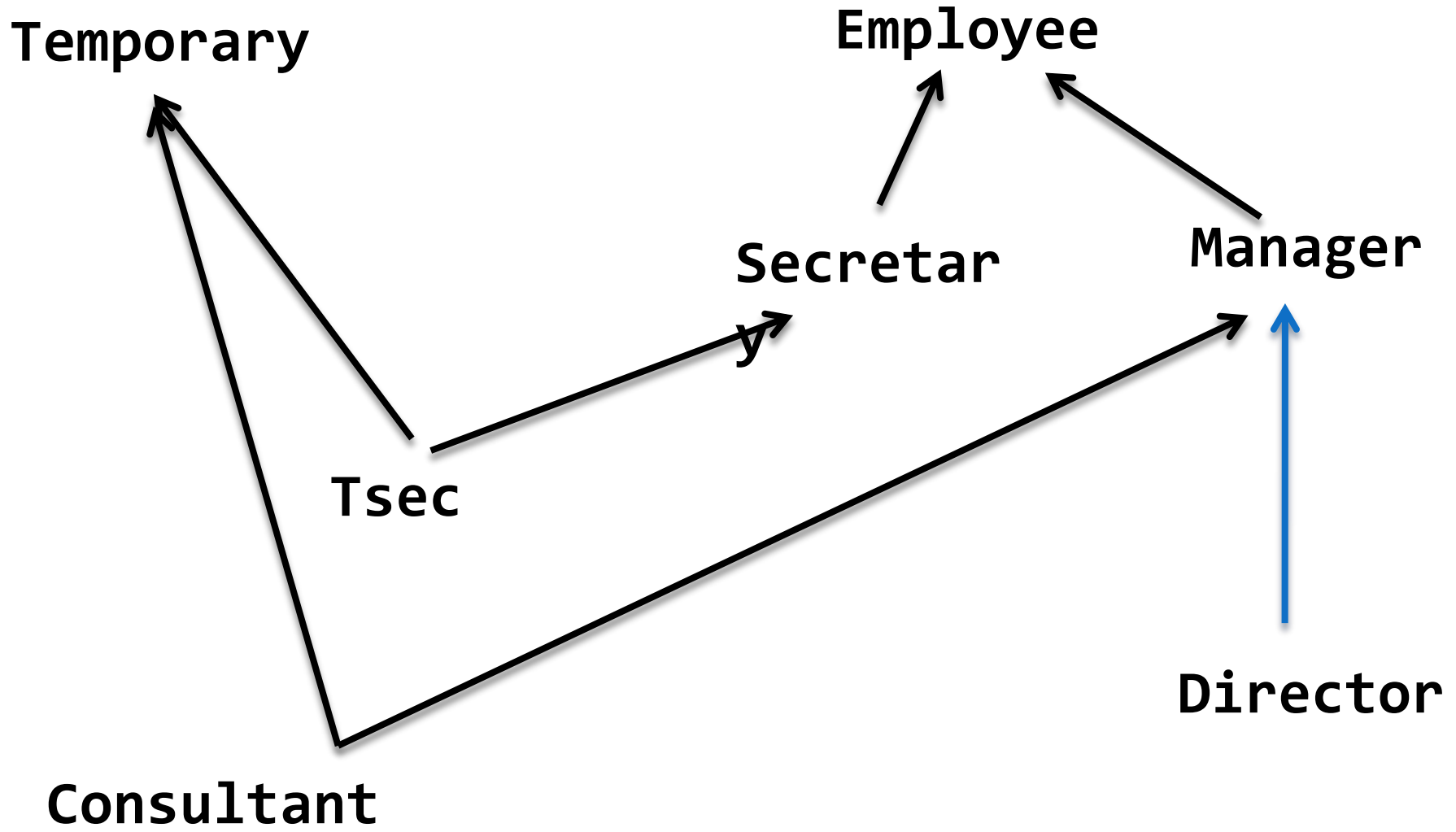
C++ példa (folyt.)

- egy leszármazott lehet ős is

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

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

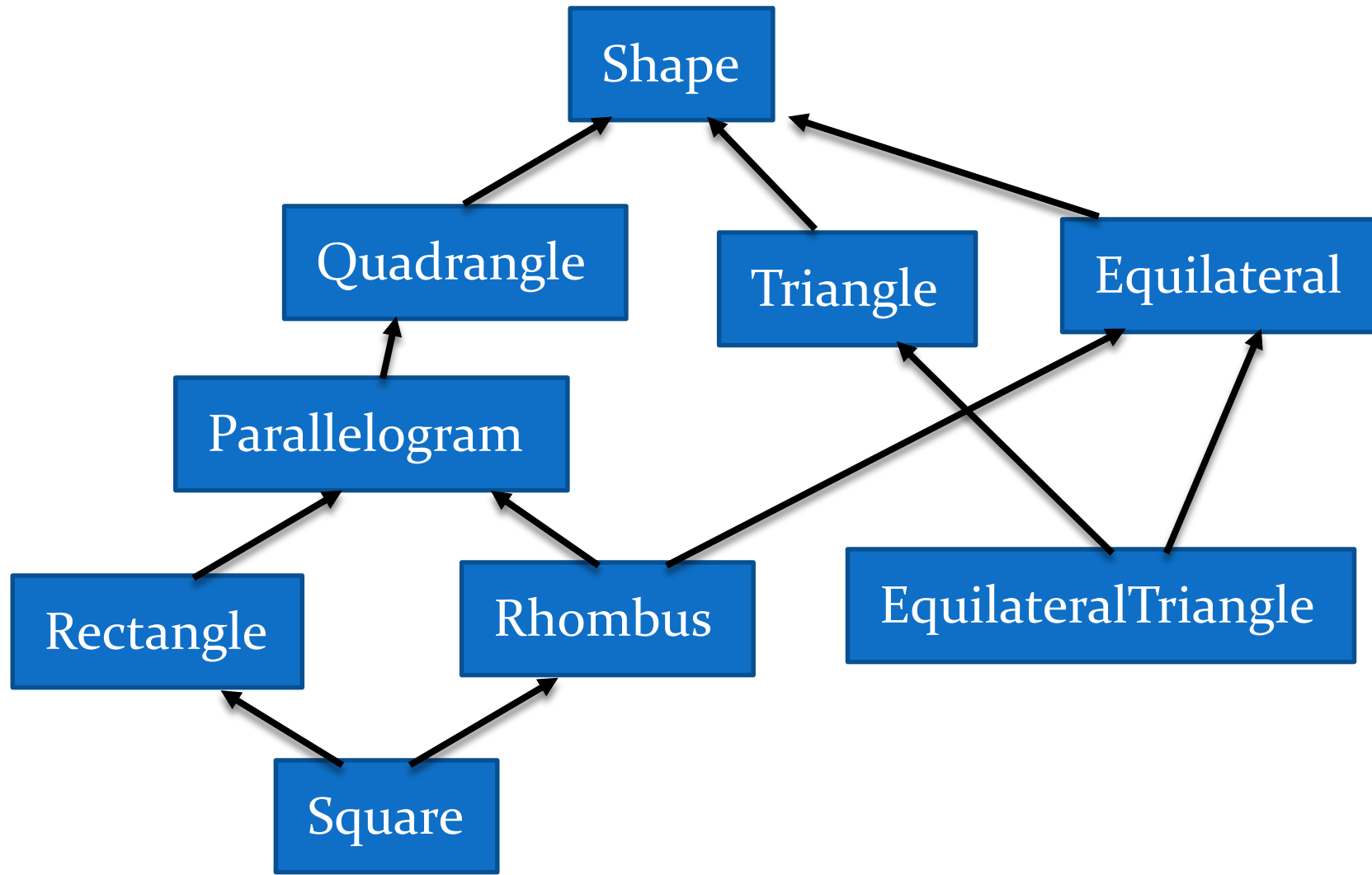
```
class Temporary { ... };  
class Secretary: public Employee { ... };  
class Tsec: public Temporary, public Secretary{ ... };  
class Consultant: public Temporary, public Manager { ... };
```



Implementáció újrahaznosítás: 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

Egy típus- és osztályhierarchia



Adjunk hozzá egy attribútumot!

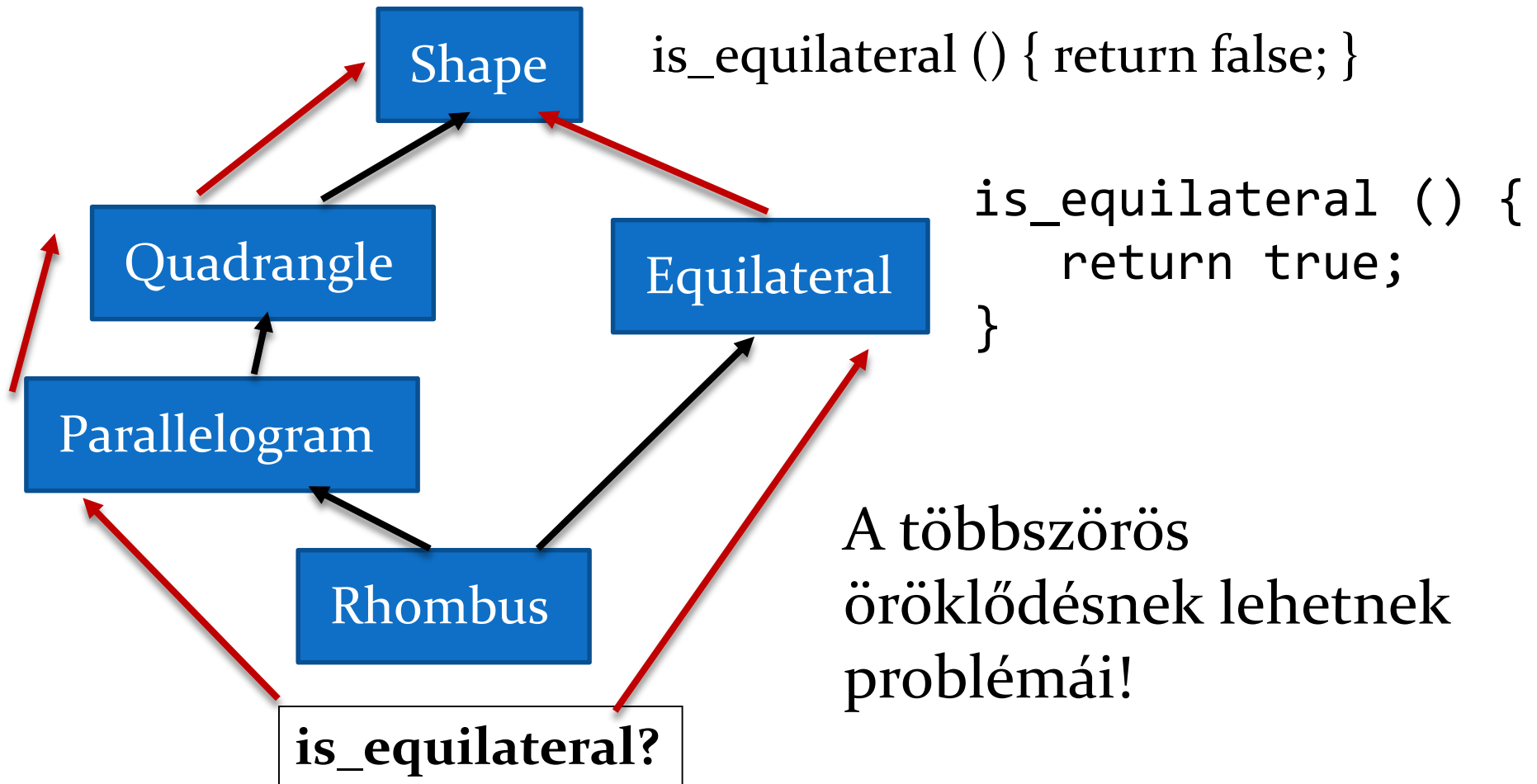
- Az alakzatoknak legyen színe – `color` – és egy `set_color` metódusa
 - A. Változtassuk meg a Shape, Quadrangle, Parallelogram, Triangle, Equilateral, EquilateralTriangle, Rhombus, Rectangle, Square stb. Típusokat
 - B. Változtassuk meg a Shape-t, a többiek **öröklík** az új attribútumot és metódust automatikusan

Adjuk hozzá az `is_equilateral`-t!

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

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

Egy Rhombus egyenlőoldalú?



Többszörös öröklődés

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

A többszörös öröklődés problémái:

```
class A{  
  int a;  
  public: virtual void f();  
};
```

```
class B : public A {  
  public: void f();};
```

```
class C : public A{  
  public: void f();};
```

```
class D : public B, public C  
{ ...};
```

átdefiniál

átdefiniál

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 lehetőségek

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

C++

A c++ „megoldása”

```
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

C++ – többszörös öröklődés

```
class Animal {  
    public: virtual void eat(); };
```

```
class Mammal : public Animal {  
    public: virtual Color getHairColor();  
    ...};
```

```
class WingedAnimal : public Animal {  
    public: virtual void flap();  
    ...};
```

// A bat is a winged mammal

```
class Bat : public Mammal, public WingedAnimal {  
    ...};
```

```
Bat bat;
```

Hogyan eszik??

C++ – többszörös öröklődés

```
class Mammal : public virtual Animal {  
    public: virtual Color getHairColor();  
    ...};  
class WingedAnimal : public virtual Animal {  
    public: virtual void flap();  
    ...};  
// A bat is still a winged mammal  
class Bat : public Mammal, public WingedAnimal {  
    ...};
```

Absztrakt osztály

```
class Alakzat{  
public:  
    virtual void kirajzol()=0;  
    virtual bool zart()=0;  
...  
}
```


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.

C++

```
class Sikidom {
    protected:
        int szelesseg, magassag;
    public:
        virtual int terület()=0;
        void beallit_ertekek(int a, int b)
            {szelesseg = a; magassag = b;}
};

class Teglalap: public Sikidom{
    public:
        int terület() {return (szelesseg * magassag);} ...
};

class Haromszog: public Sikidom{
    public:
        int terület() {return (szelesseg * magassag /2);}
};
```

C++

- Nem lehet:
Sikidom a;
- Lehet:
Teglalap t;
Haromszog h;

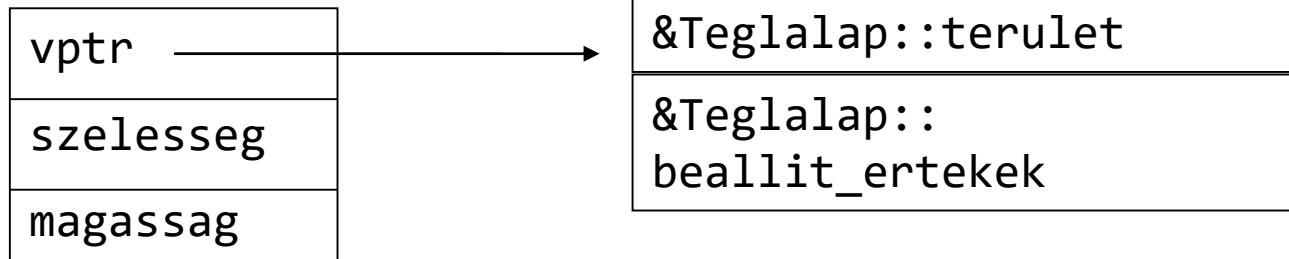
```
Sikidom* a2 = &t;  
Sikidom* a3 = &h;
```

```
a2->terulet(); //Teglalap::terulet()  
a3->terulet(); //Haromszog::terulet()
```

C++

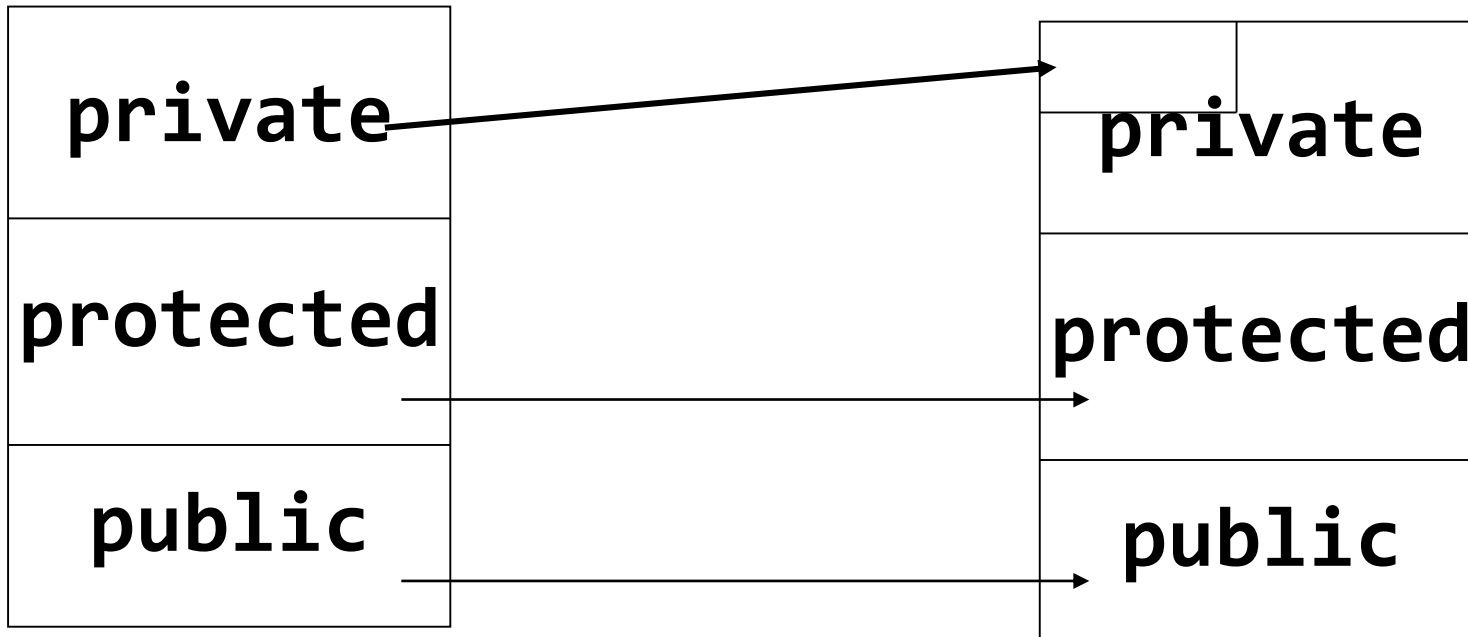
- Megvalósítás:
Teglalap t;

Teglalap osztály virtuális
metódustáblája



C++

- public öröklődés:

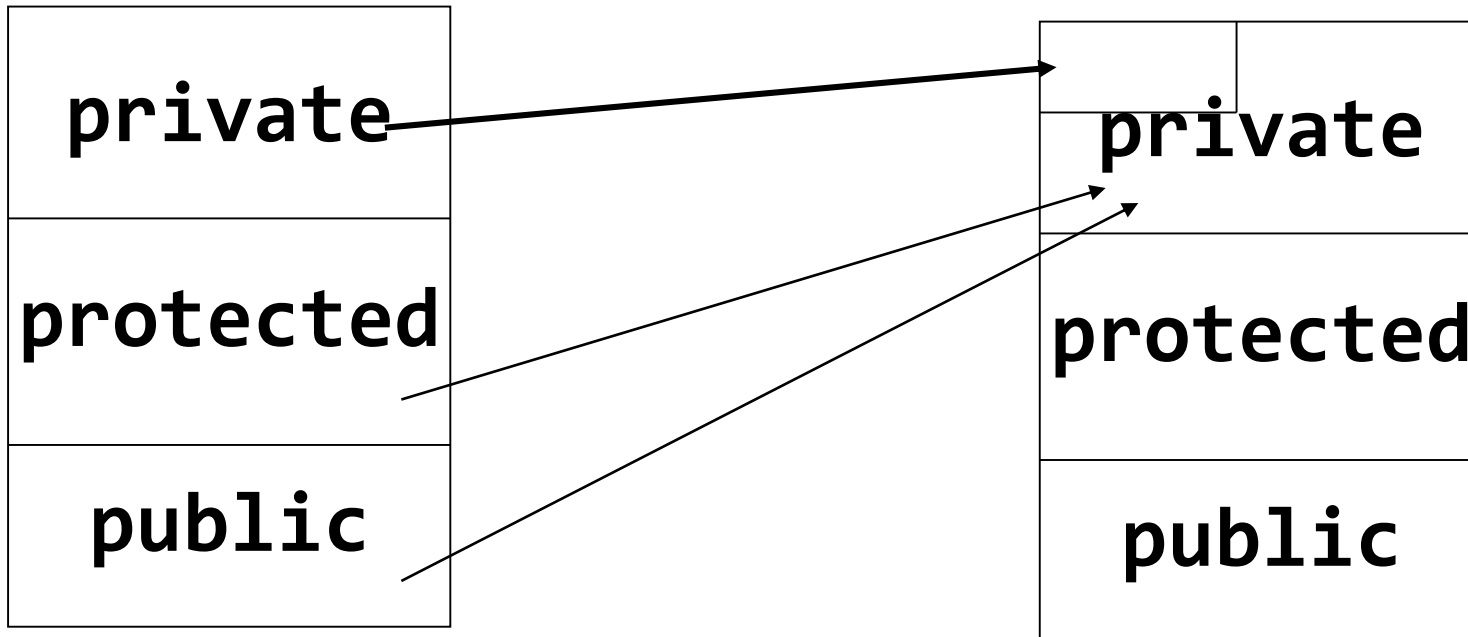


ős

leszármazott

C++

- private öröklődés:

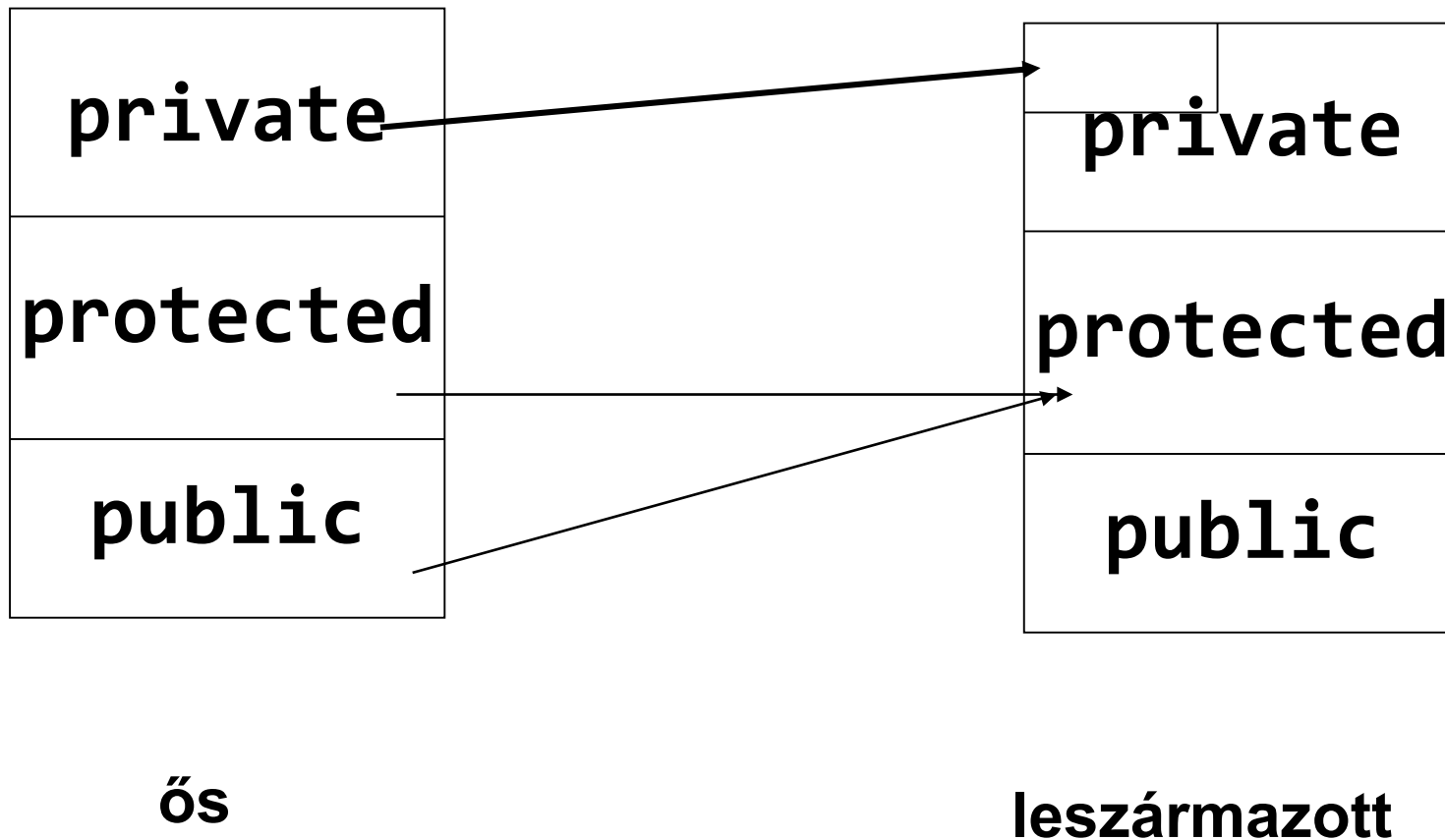


ős

leszármazott

C++

- protected öröklődés:



C++

- Mit örököl a leszármazott?
 - Adattagokat
 - Metódusokat
- Mit nem örököl a leszármazott?
 - Ősosztály konstruktorait, destruktort
 - Ősosztály értékadás operátort
 - Ősosztály barátait

C++

- Mit vezethet be a leszármazott osztály?
 - Új adattagokat
 - Új metódusokat
 - Felüldefiniálhat már meglévőket
 - Új konstruktorokat és destruktort
 - Új barátokat

C++

- Egy általános metódus deklarációja a következőket jelenti:
 1. A metódus elérheti a privát mezőket is
 2. Az osztály scope-ját használja
 3. A metódus egy konkrét objektumra hívódik meg, ezért birtokolja a 'this' pointert
- Statikus metódus csak az 1, 2 - vel rendelkezik,
- Ha egy függvényt friend-nek deklarálunk, akkor csak az 1. jogunk lesz (friend mechanizmus)

friend példa:

```
    typedef double Angle;
class Complex {
public:
    Complex(double r=0, double i=0){ R = r; I = i;}
    Complex operator =(Complex z){R = z.R; I = z.I; return *this; }
    Complex operator +(Complex z) {return Complex(R+z.R,I+z.I);}
    Complex operator +(double x) { return Complex(R+x,I);}
    Complex operator *(Complex);
        Complex operator *(double);
        Complex operator -(Complex);
        Complex operator -(double);
        Complex operator /(Complex);
        Complex operator /(double);
        double Re(); double Im();
        double Abs(); Angle Phi();
private:
    double R; double I;
};
```

C++

```
Complex operator + (double x, Complex z){ return z+x;}
```

Vagy: osztály belsejében:

```
friend Complex operator+(double, Complex);
```

```
Complex operator+(double p1, Complex p2)
{
    Complex temp;
    temp.R = p1+p2.R;
    temp.I = p2.I;
    return (temp);
}
```

C++

- Még egy (tipikus) friend példa

```
class Point {  
    friend ostream &operator<<( ostream &, const Point &);  
    public:  
        Point( int = 0, int = 0 );           // default constructor  
        void setPoint( int, int );         // set coordinates  
        int getX() const { return x; }     // get x coordinate  
        int getY() const { return y; }     // get y coordinate  
    protected:                             // accessible by derived classes  
        int x, y;                           // x and y coordinates of the Point  
}; // end class Point
```