# Objektumok kapcsolatai

# Objektum-kapcsolatok fajtái

- Amikor objektumok egymással kommunikálnak (szinkron vagy aszinkron módon egymás metódusait hívják, egyik a másiknak szignált küld, esetleg közvetlenül a másik adattagjain végeznek műveletet), akkor kapcsolat alakul ki közöttük.
- Az objektumok közötti kapcsolatoknak több fajtáját is megkülönböztetjük:
  - Függőség (*dependency*)
  - Asszociáció (*association*) vagy társítás
  - Aggregáció (*aggregation*, *shared aggregation*) vagy tartalmazás
  - Kompozíció (*composition*, *composite aggregation*) vagy szigorú tartalmazás
  - Származtatás vagy öröklődés (*inheritence*)

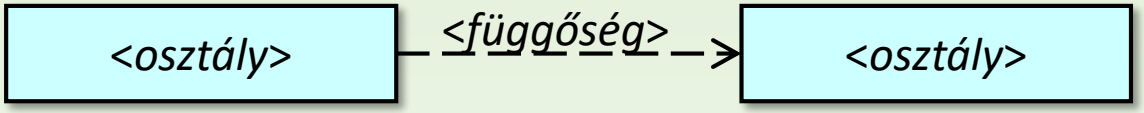
# Fogalmak absztrakciója

- Az objektumokat (*object*) és azok kapcsolatait (*link*) az objektumdiagrammal ábrázolhatjuk, de az objektumok és kapcsolataik tulajdonságait egy magasabb absztrakciós szinten az osztálydiagramm mutatja meg.
- Az osztálydiagramm az objektumdiagramm absztrakciója.
- A programozási nyelvek többnyire sajnos csak az objektumok absztrakcióját, azaz az osztály fogalmát (*class*) ismerik, a kapcsolatok absztrakt leírására – a származtatás kivételével – nem biztosítanak nyelvi eszközöket.

4

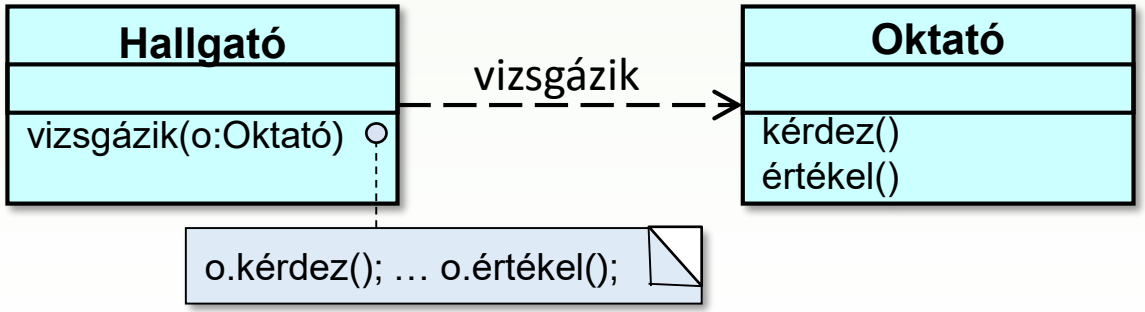
Az objektum-orientáltság ismérve a fogalmi szintű absztrakció.

# Függőség



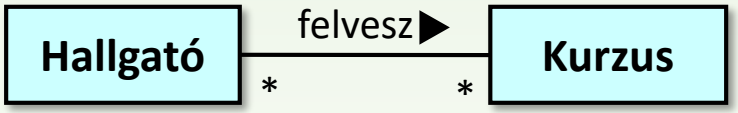

- Amikor egy osztály egyik metódusa epizód szerűen kerül kapcsolatba egy másik osztály objektumával úgy, hogy ez a metódus a másik objektumot paraméterként megkapja vagy lokálisan létrehozza abból a célból, hogy annak egy metódusát meghívja, vagy szignált küldjön neki, vagy magát az objektum hivatkozását adja tovább (pl. kivételdobással).
- Amikor egy osztály egyik metódusa a másik osztály osztályszintű metódusát hívja.

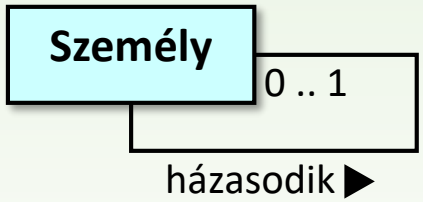
Példa:



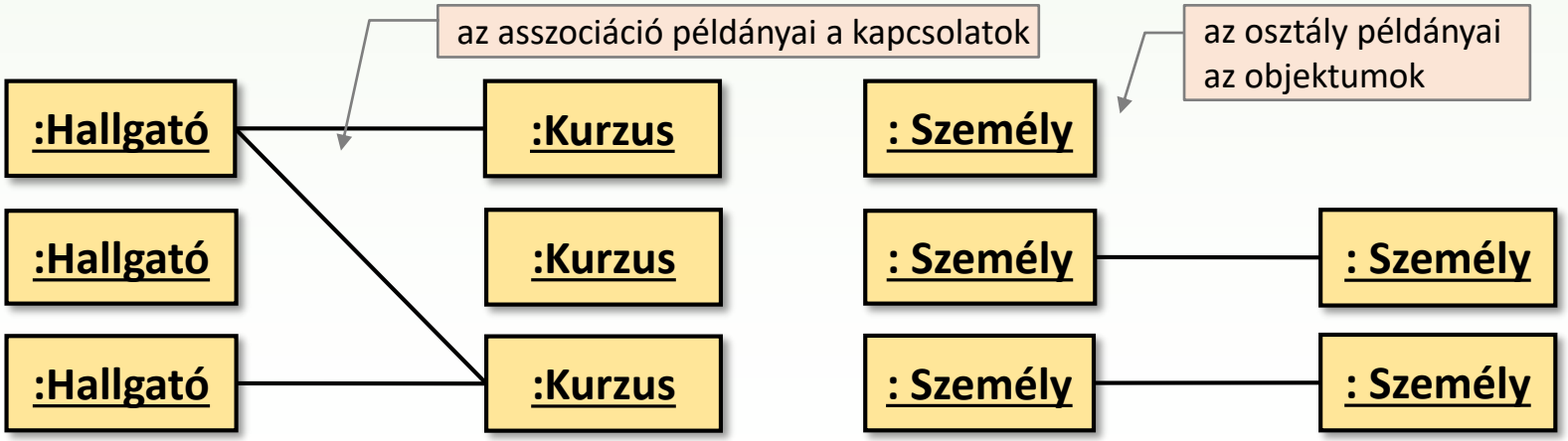
# Asszociáció

- Osztályok közötti asszociáció hosszabb időszakon keresztül fennálló kapcsolatot fejez ki osztályok objektumai között. (Az asszociáció lényegében objektumok között állandósult függőségi kapcsolatot.)
- Egyetlen asszociáció számos objektum-kapcsolatot ír le.




A fenti osztálydiagrammok egy lehetséges példányosítása (felpopulálása):

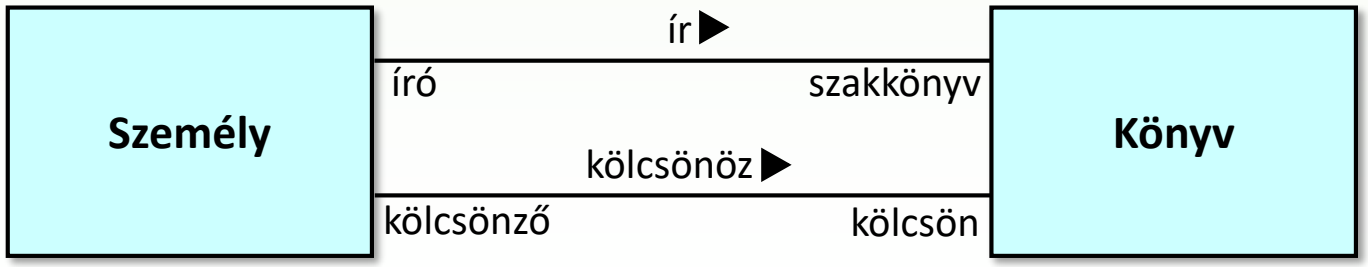


# Asszociáció tulajdonságai


- Egy asszociációnak különféle tulajdonságai vannak, mint például a
  - neve
  - hatásiránya
  - multiplicitása
  - aritása (bináris vagy n-áris asszociációk)
  - navigálhatósága
  - asszociációvégek nevei (szerepnevek)
  - asszociációvégek neveinek láthatósága, tulajdonosa
- Ha valamelyik tulajdonság hiányzik, az általában azt jelzi, hogy ez a tervezés folyamán még nem kristályosodott ki.

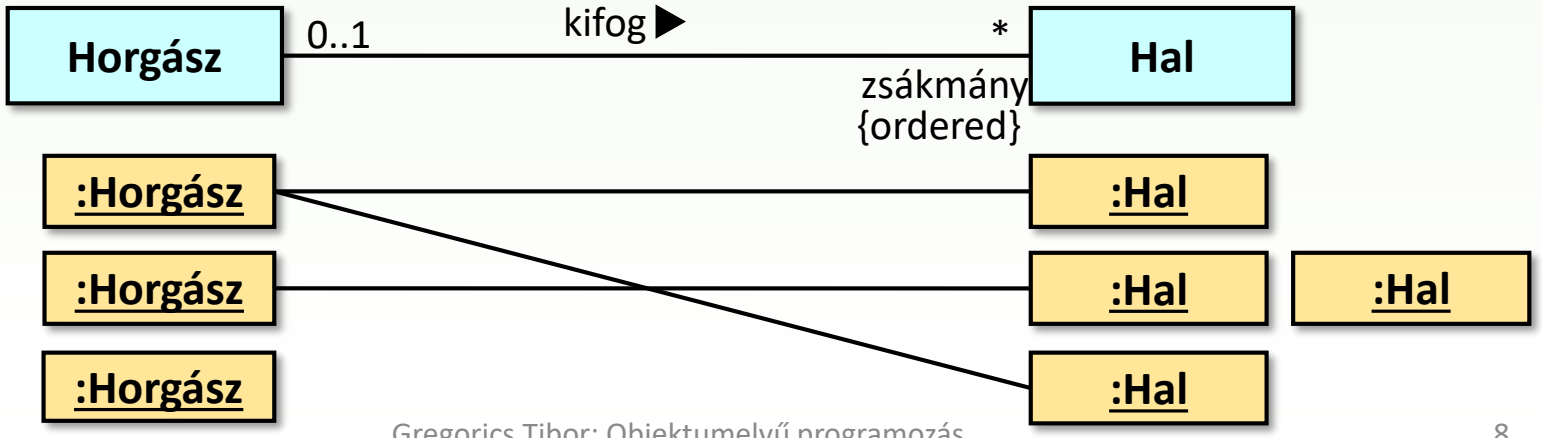
# Asszociáció neve, hatásiránya

- Az asszociációkat gyakran egy egyszerű bővített mondattal fejezzük ki, és ilyenkor ennek a mondatnak az állítmánya az asszociáció neve, a mondat többi eleme pedig az asszociációvégek nevei (amelyeket gyakran szerepneveknek is hívnak).
- A bináris (azaz két objektum kapcsolatát leíró) asszociációkat
  - egy alany-állítmány-tárgy szerkezetű mondattal jellemezhetjük, ahol az asszociáció végek nevei a mondat alanya illetve tárgya.
  - A mondat tárgyára az asszociáció neve mellé rajzolt fekete háromszög hegye mutat: ez a bináris asszociáció hatásiránya.



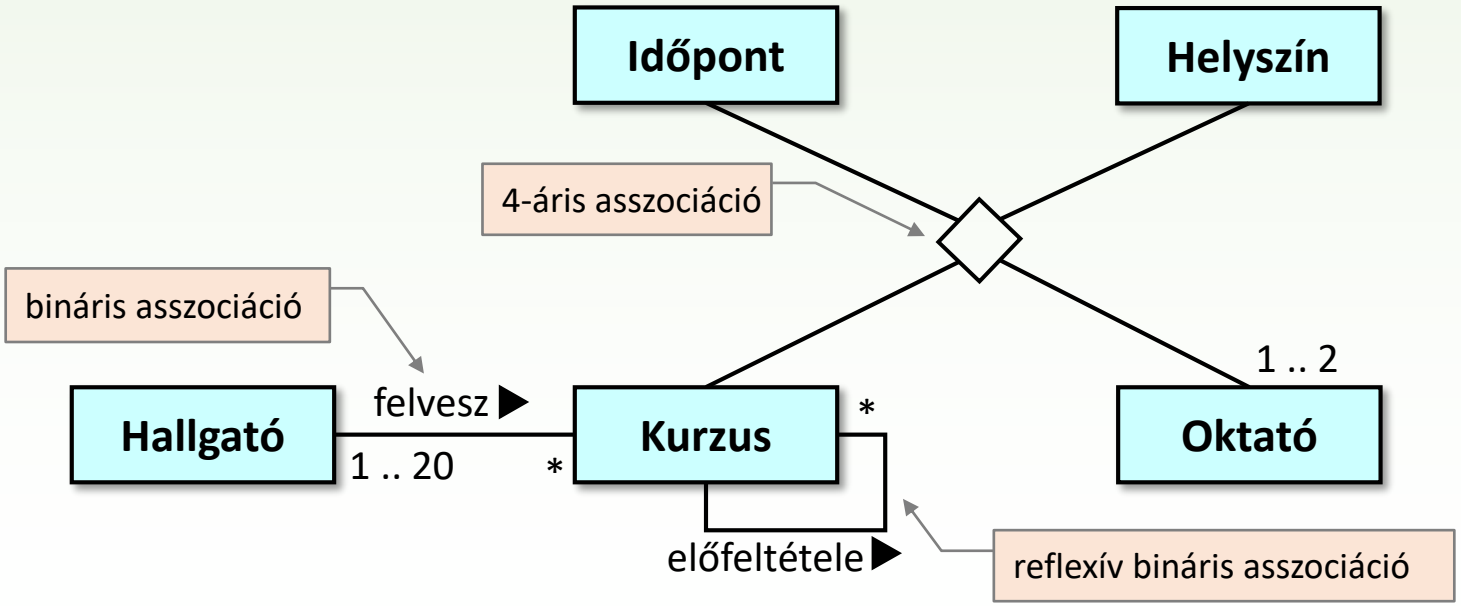
# Multiplicitás

- Az asszociáció multiplicitása azt mutatja meg, hogy a multiplicitással ellátott osztálynak hány (min .. max) objektuma létesíthet egyidejűleg kapcsolatot az asszociáció (többi) másik osztályának egy objektumával.
  - az 1 multiplicitás jelölését gyakran elhagyjuk
  - a 0 .. * helyett a * jelölést használjuk, ahol * tetszőleges természetes szám
- Előírhatjuk egy „sok” multiplicitású asszociáció kapcsolataira, hogy egy objektumhoz kapcsolt „sok oldali” objektumok
  - mind különbözzenek {unique},
  - megadott sorrendben legyenek felsorolhatók {ordered}.



# Asszociáció aritása

- Az asszociáció aritása arra utal, hogy a belőle példányosított kapcsolatok hány objektumot kötnek össze.
- Eddig csak bináris asszociációkkal találkoztunk, ahol két objektum kapcsolódott egymáshoz.
  - Ugyanaz az objektum több kapcsolatban is szerepelhet.
  - Reflexív asszociációban egy osztály két objektuma kapcsolódik össze.



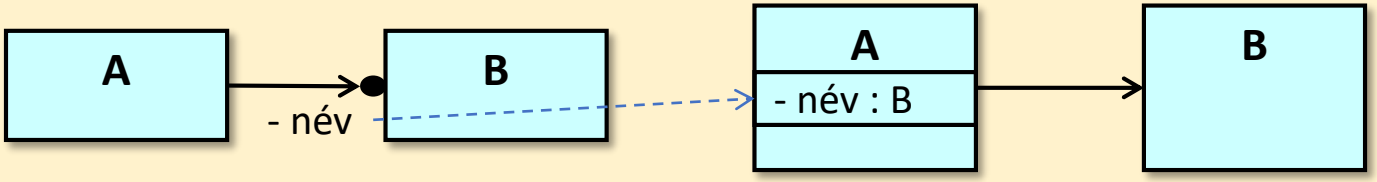
# Navigálhatóság

- A navigálhatóság azt mutatja, hogy egy kapcsolat megvalósítása során melyik objektumnak kell tudnia hatékonyan elérni a másikat.
  - Adott irányban történő hatékony navigálási irányt az asszociáció adott végén elhelyezett nyíl jelöli.
  - Az x a navigálás nem támogatott irányát jelöli.
  - A jelöletlen asszociációvég a nem-definiált navigálhatóságra utal.
- A navigálhatóság iránya és a hatásirány különböző fogalmak, ezért irányításuk különbözhet.


```
class Person {
private:
     IdentityCard *_ic;
public:
     void makeIdentityCard() { … ; _ic = new IdentityCard (…); }
     IdentityCard showIdentityCard() const { return *_ic; }
};
```

# Asszociációvég-név tulajdonosa

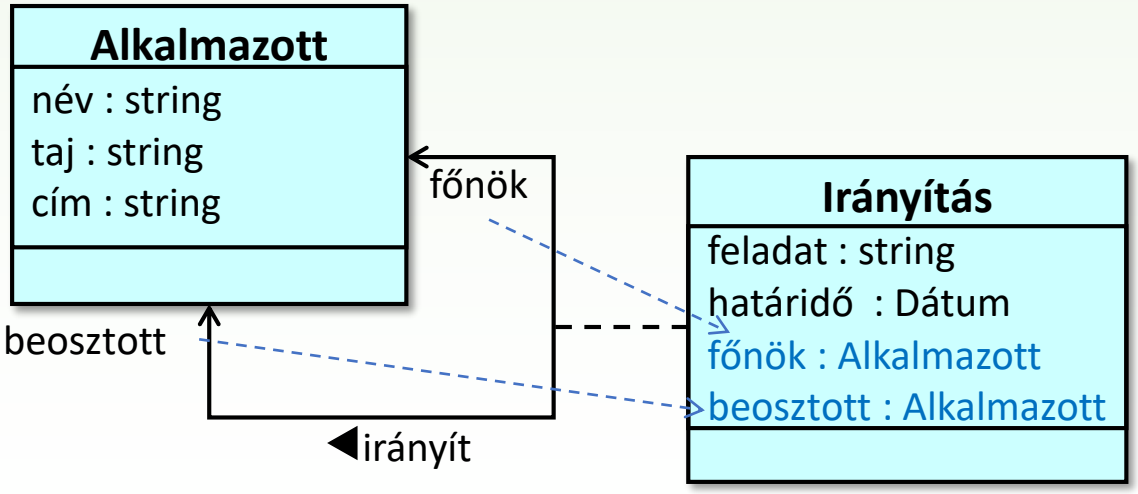
- Egy kapcsolat objektumaira az asszociáció végéhez írt nevekkel (a szerepnevekkel) hivatkozunk. Hol tároljuk ezeket a hivatkozásokat? Ki a szerepnév tulajdonosa?
  - Lehet maga az asszociáció: ekkor az objektumok közötti kapcsolat tárolja a kapcsolatban levő és hatékonyan elérendő objektumok hivatkozását.
  - Lehet az ellentétes oldali osztály: ekkor az ellentétes oldali objektumnak adattagja a szerepnév. Erre az osztálydiagrammban a szerepnévnél feltüntetett *fekete pötty* utal.




- A szerepnév láthatósága (private, protected, public) mutatja, hogy a név publikus, vagy kizárólag csak a tulajdonosa láthatja.

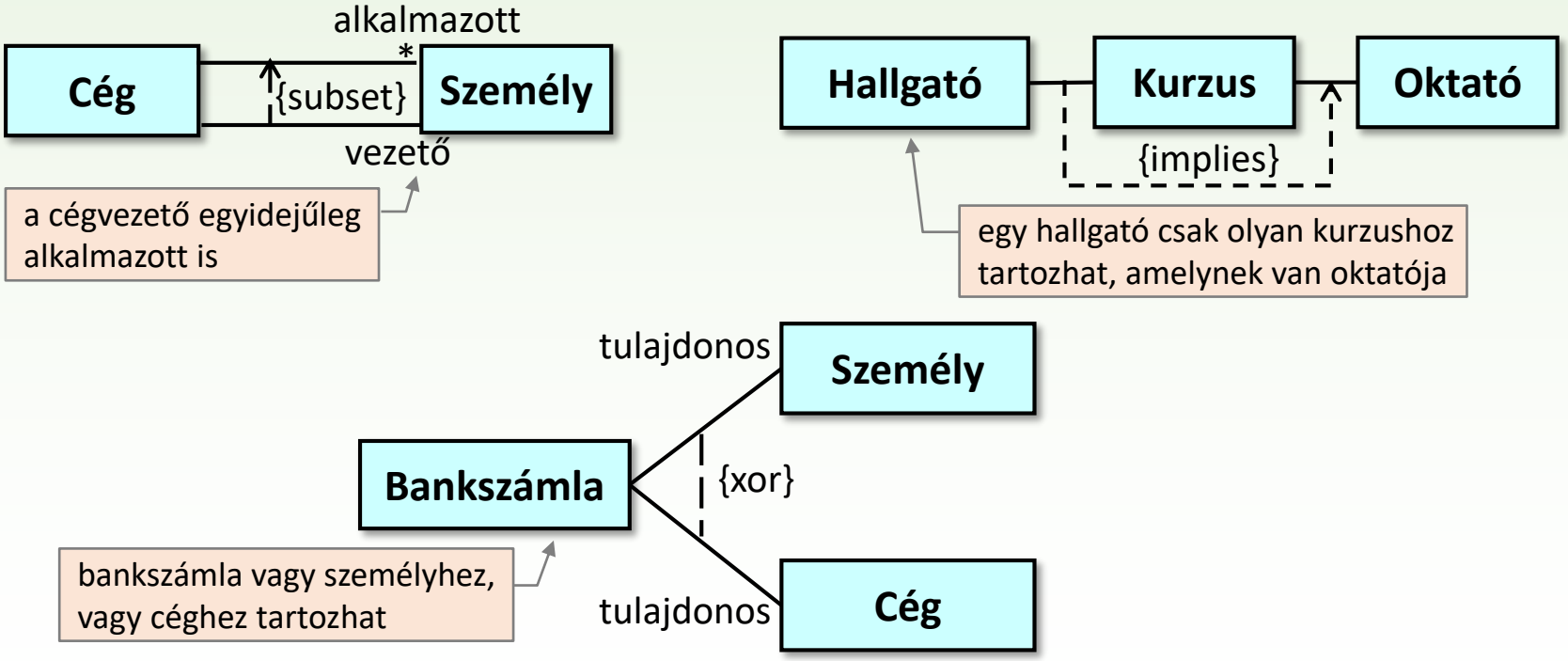
# Asszociációs osztály

- Az UML lehetőséget ad az asszociációhoz tartozó kapcsolatok tulajdonságait leíró osztály definiálására Ennek példányai a kapcsolatok, amelyekhez az általuk összekapcsolt objektumok mind hozzáférnek, és így elérik az abban tárolt információt.
- Amikor egy szerepnévnek maga az asszociáció a tulajdonosa, akkor a szerepnév az asszociációt leíró asszociációs osztálynak lesz az adattagja.
- Ezt a jelentősebb OO nyelvek nem támogatják.

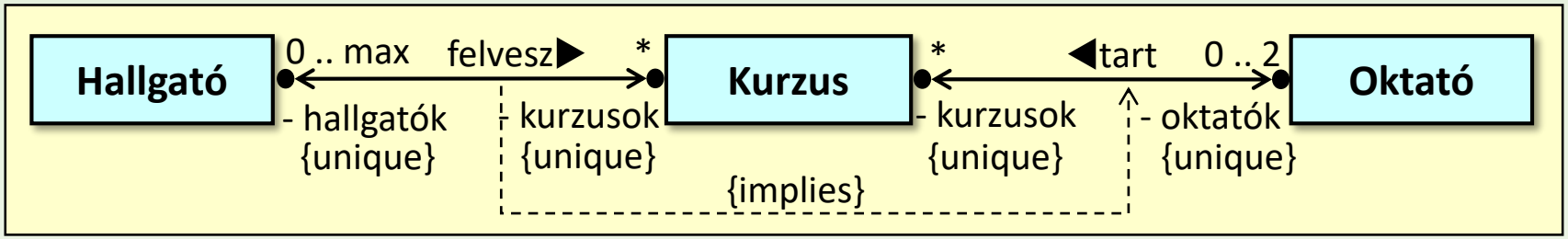


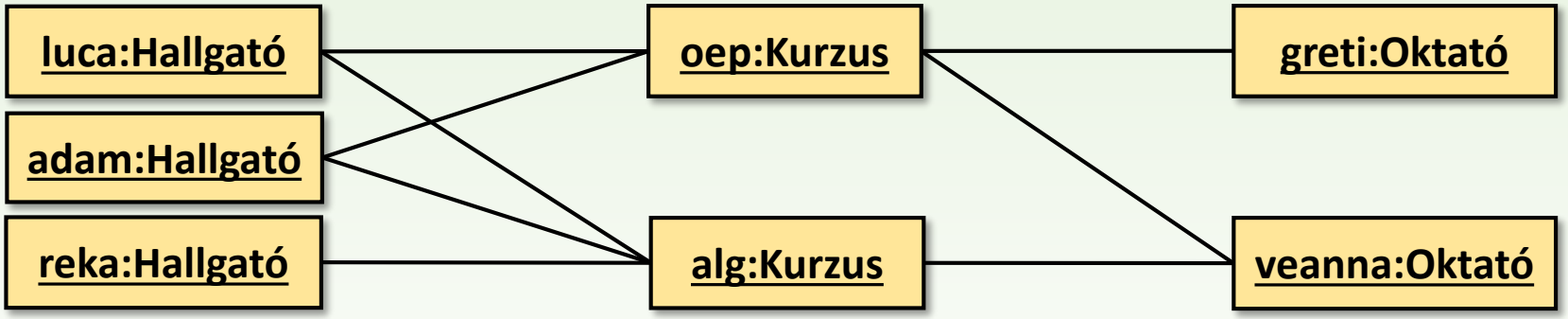
# Asszociációk közötti tulajdonságok

- Megadhatunk az asszociációk között logikai feltételeket (subset, and, or, xor, implies, … ), amelyek ugyanazon objektum különböző asszociációkból létrejött kapcsolataira fogalmazhatnak meg korlátozásokat.



# Példa



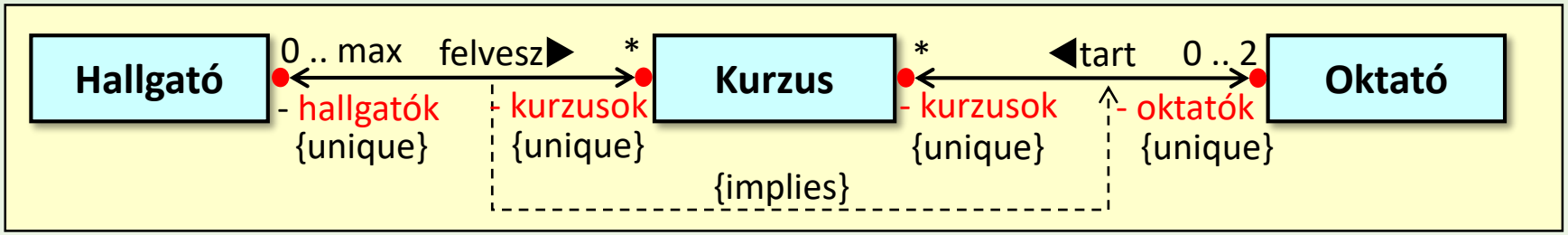




```
Student luca, adam, reka;
Teacher greti, veanna;
Course oep(20), alg(22);

greti.undertakes(&oep);
veanna.undertakes(&oep); veanna.undertakes(&alg);

luca.signs_up(&alg); luca.signs_up(&oep);
adam.signs_up(&alg); adam.signs_up(&oep);
reka.signs_up(&alg);
```

# Példa osztályai

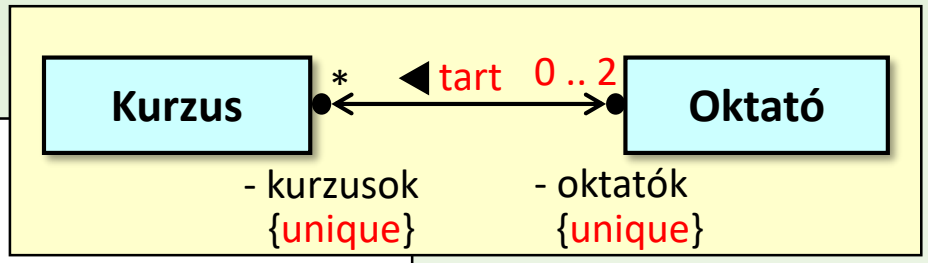



```
class Course {
private:
     unsigned int max;
     std::vector<Teacher*> _teachers; // 0 .. 2
     std::vector<Student*> _students; // 0 .. max
public:
     …
};
```

```
class Student {
private:
     std::vector<Course*> _courses;
public:
     …
};
```

```
class Teacher {
private:
     std::vector<Course*> _courses;
public:
     …
};
```

# Példa metódusai 1.



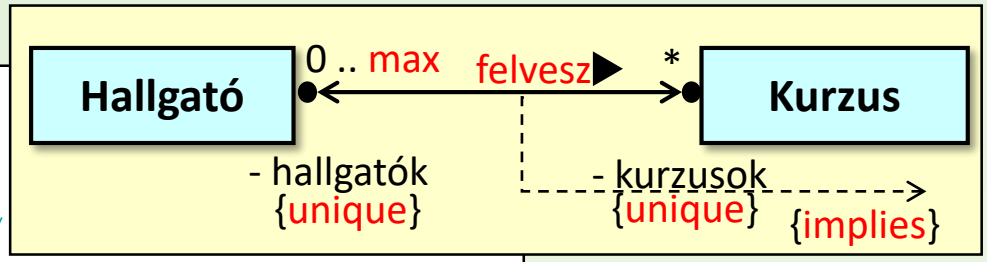

```
class Course {
private:
     unsigned int max;
     std::vector<Teacher*> _teachers; // 0 .. 2
     std::vector<Student*> _students; // 0 .. max
public:
     bool can_lead(Teacher *pt) {
          bool l = false;
          for(Teacher *p : _teachers) {
               if((l=p==pt)) break;
          }
          if(!l && _teachers.size()<2) {
               _teachers.push_back(pt);
               return true;
          }
          else return false;
     }
     …
};
```

ez a lineáris keresés ellenőrzi mindkét unique feltételt

vizsgálja a multiplicitás felső korlátját

```
class Teacher {
private:
     std::vector<Course*> _courses;
public:
     void undertakes(Course *pc){
          if( pc==nullptr ) return;
          if( pc->can_lead(this) ) {
               _courses.push_back(pc);
          }
     }
};
```

# Példa metódusai 2.




```
class Course {
private:
     int max;
     std::vector<Teacher*> _teachers; /
     std::vector<Student*> _students; // 0 .. max
public:
     Course(int a) { max = a; }
     bool can_lead(Teacher *pt) { … }
     bool has_teacher() const { return _teachers.size()>0; }
     bool can_sign_up(Student *ps) {
          bool l = false;
          for(Student *p : _students) {
               if((l = p==ps)) break;
          }
          if( !l && _students.size()<max ){
               _students.push_back(ps);
               return true;
          }
          else return false;
     }
};
```

vizsgálja az implies feltételt

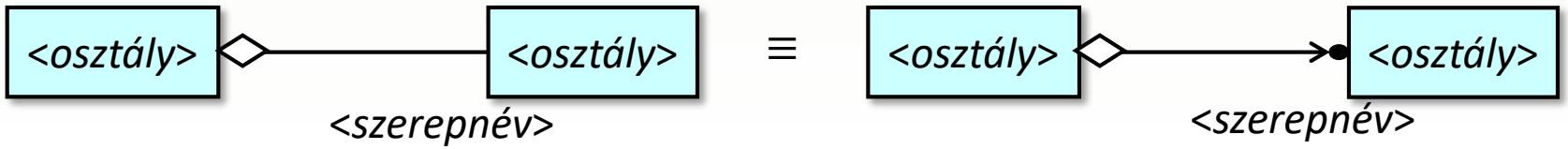
ez a lineáris keresés ellenőrzi mindkét unique feltételt

vizsgálja a multiplicitás felső korlátját

```
class Student {
private:
     std::vector<Course*> _courses;
public:
     void signs_up(Course *pc){
          if( pc==nullptr
               || !pc->has_teacher() )
                    return;
          if( pc->can_sign_up(this) )
               _courses.push_back(pc);
     }
};
```

# Aggregáció

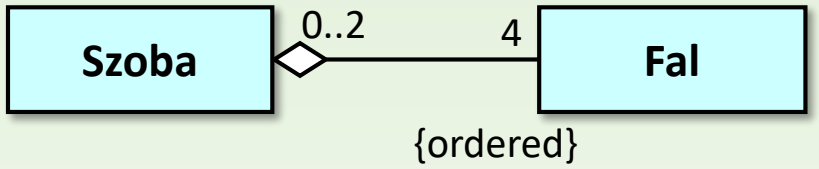
- Az aggregáció egész-rész kapcsolatot kifejező bináris asszociáció, amely akkor valósul meg, ha egy objektumnak része, tulajdona egy másik:
  - Ez a reláció aszimmetrikus, tranzitív, nem reflexív, és irányított kört sem alkothat (azaz egy objektum még közvetett módon sem tartalmazhatja saját osztályának objektumát).
  - Ugyanazon objektum több objektumnak is része lehet – akár egyidejűleg is –, és a tartalmazó objektum megszűnése után a tartalmazott objektum tovább élhet.
- Megállapodunk továbbá abban, hogy külön jelzés hiányában
  - a kapcsolat a tartalmazott osztály irányába navigálható,
  - a tartalmazott osztály szerepneve a tartalmazó osztály tulajdona.



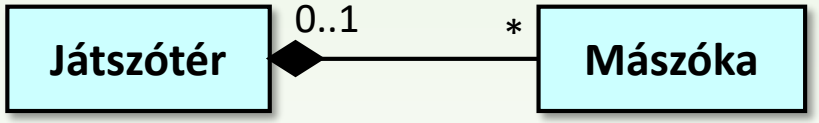
# Kompozíció

- A kompozíció egy speciális aggregáció, ahol a tartalmazott objektum csak egy másik objektum része lehet.
  - Ez a reláció is aszimmetrikus, tranzitív, nem reflexív, és nem alkothat irányított kört.
  - Több szigorúbb értelmezése is ismert: a tartalmazott objektum
    - tartalmazó objektum nélkül, önmagában nem létezhet: mindig része egy tartalmazó objektumnak
    - végig ugyanannak a tartalmazó objektumnak a része: létrehozása és megszüntetése a tartalmazó objektum feladata
    - élettartama azonos a tartalmazó objektuméval: annak konstruktora példányosítja, destruktora törli.

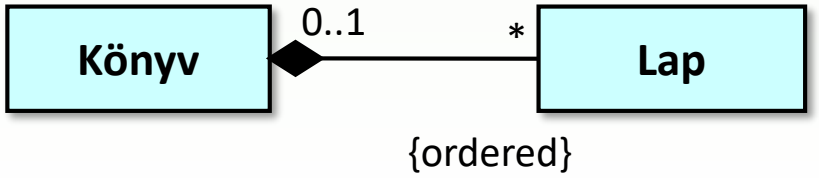
# Példák




Ugyanaz a fal egyszerre két szobához is tartozhat. A fal magában is állhat.

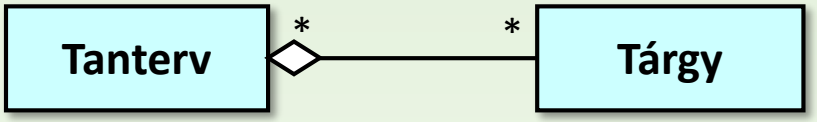



A mászókák a játszótér részei, de mindig csak egy játszótéré. Át lehet vinni őket egy másik játszótérre, és önmagukban, játszótér nélkül is használhatók. (Ez itt a kompozíció legmegengedőbb megítélése.)

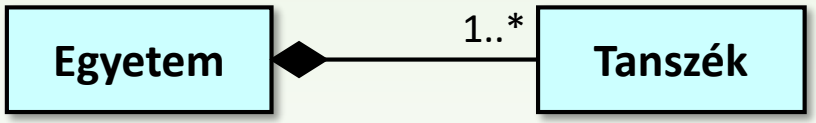



A könyv lapjai egyszerre csak egy könyvhöz tartoznak. Egy lap kieshet a könyvből, és ezután is használható, akárcsak a könyv. (Egy könyvtáros mást mondana erről.)

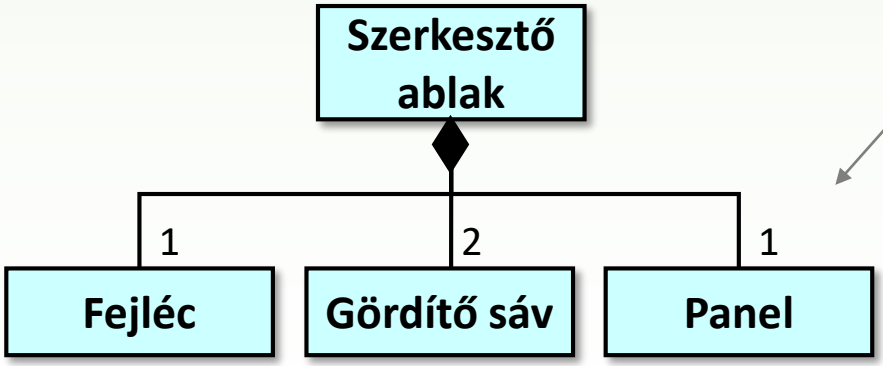
# Példák




Egy tantárgy egyszerre több tanterv része is lehet, de tanterven kívüli is lehet.



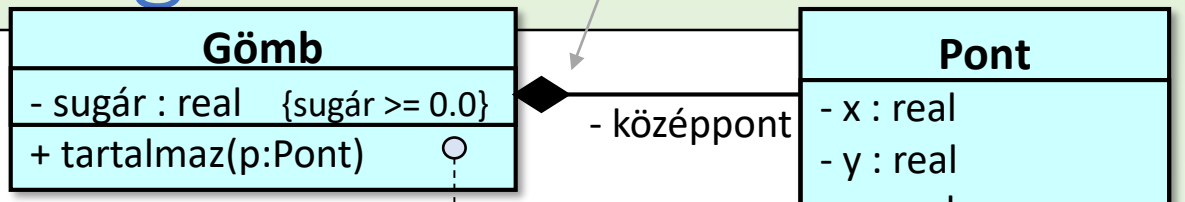

A tanszékek egy egyetemhez tartoznak. A tanszékek élettartama nem nyúlik túl az egyetem élettartamán (ha bezárják az egyetemet, akkor tanszékei megszűnnek). Egy tanszék élettartama lehet rövidebb az egyeteménél.




Egy ún. szerkesztő ablak létrehozásakor az ablakkal együtt születnek meg annak tartozékai, amelyek majd csak az ablak megszűnésekor törlődnek.

# Példa: pont a gömbben

Egy gömbnek része a középpont, amely a gömbbel együtt születik és szűnik meg.




```
class Point {
private:
     double _x, _y, _z;
public:
     Point(double a, double b, double c) : _x(a),_y(b),_z(c){}
     double distance(const Point &p) const {
          return sqrt(pow(_x-p._x,2) + pow(_y-p._y,2) + pow(_z-p._z,2));
     }
};
```

**return** középpont.távolság(p) ≦ sugár

**return** sqrt((x–p.x)²+(y–p.y)²+(z–p.z)²)

```
class Sphere{
private:
     Point _centre;
     double _radius;
public:
     enum Errors{ILLEGAL_RADIUS};
     Sphere(const Point &c, double r): _centre(c), _radius(r) {
          if (_radius<0.0) throw ILLEGAL_RADIUS;
     }
     double contains(const Point &p) const {
          return _centre.distance(p) <= _radius;
     }
};
```

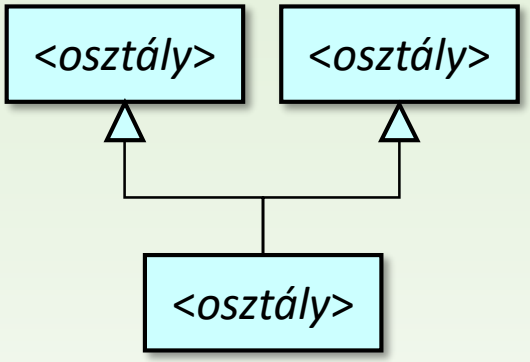
kompozíció

A c pontot az automatikusan létrejövő középpont számára másolja le: így külön objektum lesz a c-nek átadott pont és a gömb középpontja.

```
Point  p(-12.0, 0.0, 23.0);
Point  c(-12.3, 0.0, 23.4);
Sphere g1(c, 1.0);
cout << g1.contains(p) << endl;
cout << c.distance(p) << endl;
```

# Származtatás, öröklődés

- Ha egy objektum más objektumokra hasonlít (azokkal megegyező adattagjai és metódusai vannak), akkor az osztálya a hasonló objektumok osztályaiból származtatható: örökli azok tulajdonságait, de módosíthatja, és ki is egészítheti azokat.




- A modellezés során kétféle okból végezhetünk származtatást:
  - Általánosítás: már meglévő, egymáshoz hasonló osztályoknak a közös tulajdonságait leíró ősosztályát (szuperosztály) hozzuk létre.
  - Specializálás: egy osztályból származtatással hozunk létre egy alosztályt.

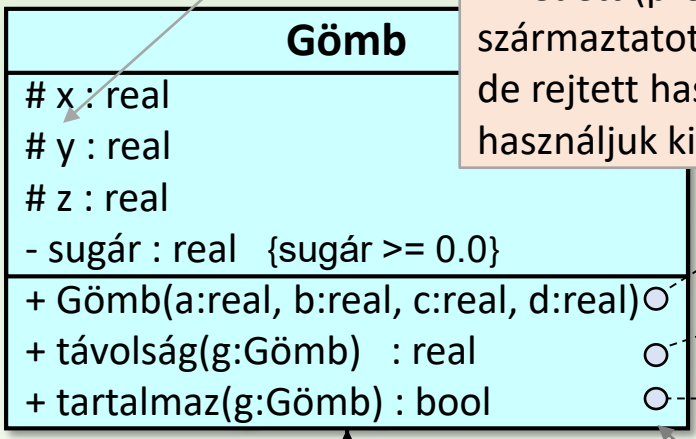
5

Az objektum-orientáltság legtöbbet emlegetett ismérve az öröklés: osztályok származtathatóak már meglévő osztályokból.
Ősosztály változójának értékül adható az alosztályának objektuma.

# Példa: gömbből pont

**S**ingle responsibility
**O**
**L**iskov substitution
**I**
**D**

A védett (protected) láthatóság lehetőséget ad a származtatott osztálynak egy adattag közvetlen, de rejtett használatára, még ha ezt most nem is használjuk ki.

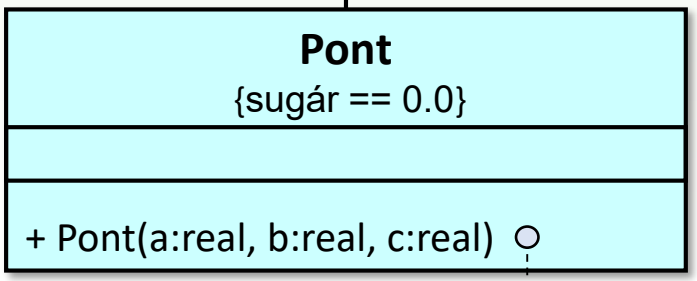



x, y, z, sugár := a, b, c, d

**return** sqrt((x–g.x)²+(y–g.y)²+(z–g.z)²) – this.sugár – g.sugár

**return** távolság(g) + 2 · g.sugár ≦ 0

Egy ősosztály típusú változónak értékül adható az alosztály egy objektuma.




Az öröklődés miatt a távolság() és a tartalmaz() metódus meghívható gömb helyett pontra is, sőt a paraméterük is lehet gömb helyett pont:

```
Gomb g1, g2; Pont p1, p2;
… g1.tartalmaz(g2) …
… g1.tartalmaz(p2) …
… p1.tartalmaz(g2) …
… p1.tartalmaz(p2) …
```

Ezek itt mind helyesen is működnek.

Gömb(a, b, c, 0.0)

Az ősosztály konstruktora nem öröklődik ugyan, de a leszármazott konstruktora meghívja.

típus invariáns

# Származtatás és láthatóság

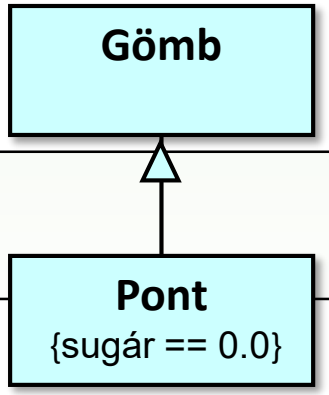
- Az alosztályban hivatkozhatunk az ősosztályban definiált publikus és védett tagokra, de nem érjük el az ősosztály privát tagjait, azokra csak indirekt módon, az ősosztálytól örökölt metódusokkal tudunk hatni.
- Maga az származtatás módja is lehet publikus, védett vagy privát
  - Publikus (*public*), ha az ősosztály publikus és védett tagjai az ősosztályban definiált láthatóságukkal együtt öröklődnek. (Az UML szerint ez a default, de a C++ nyelvben nem.)
  - Védett (*protected*), ha az ősosztály publikus és védett tagjai védettként öröklődnek.
  - Privát (*private*), ha az ősosztály publikus és védett tagjai privátként öröklődnek.

# C++ : gömbből pont

```
Point  p(0,0,0);
Sphere s(1,1,1,1);

cout << s.contains(p) << endl;
cout << s.distance(p) << endl;
cout << s.contains(s) << endl;
cout << s.distance(s) << endl;
cout << p.contains(s) << endl;
cout << p.distance(s) << endl;
cout << p.contains(p) << endl;
cout << p.distance(p) << endl;
```

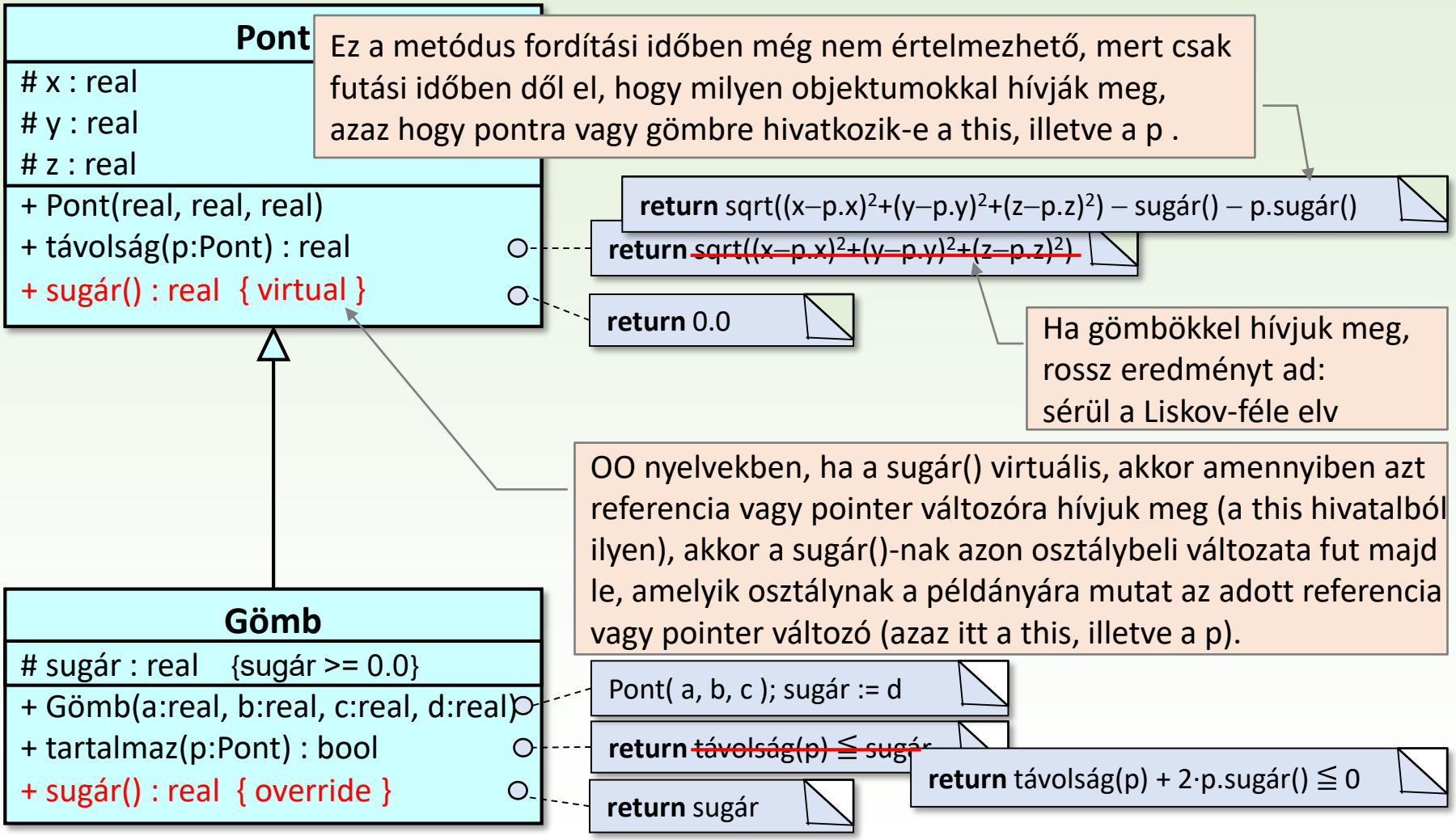
```
class Sphere{
protected:
    double _x, _y, _z;
private:
    double _radius;
public:
    enum Errors{ILLEGAL_RADIUS};
    Sphere(double a, double b, double c, double d):
        _x(a),_y(b),_z(c),_radius(d) { if(_radius<0.0) throw ILLEGAL_RADIUS; }
    double distance(const Sphere g) const
        { return sqrt(pow((_x-g._x),2) + pow((_y-g._y),2) + pow((_z-g._z),2))
                -_radius - g._radius; }
    bool contains(const Sphere g)  const
        { return distance(g) + 2 * g._radius <= 0; }
};
```




publikus származtatás

```
class Point : public Sphere {
public:
    Point(double a, double b, double c) : Sphere(a, b, c, 0.0) {}
};
```

# Példa: pontból gömb



# Polimorfizmus és dinamikus kötés

- Ha egy ősosztály metódusát a leszármazott osztályban felülírjuk (override), akkor ugyanaz a metódus több alakkal is rendelkezik (polimorf).
- Egy ősosztály típusú változónak értékül adható egy alosztálynak egy példánya, ezért csak futási időben derülhet ki, hogy a változó az ősosztály példányára vagy egy alosztályának példányára hivatkozik. (késői vagy dinamikus kötés).
- Ha egy objektumra hivatkozó referencia vagy pointer változóra az ősosztály egy polimorf virtuális metódusát hívjuk meg, akkor attól függ, hogy a metódus melyik változata fut majd le, hogy változó éppen melyik osztály objektumára hivatkozik (futási idejű polimorfizmus).

6

Az objektum-orientáltság jellegzetes ismérve a futási idejű polimorfizmus és az ezzel párban járó dinamikus kötés.

# C++ : pontból gömb

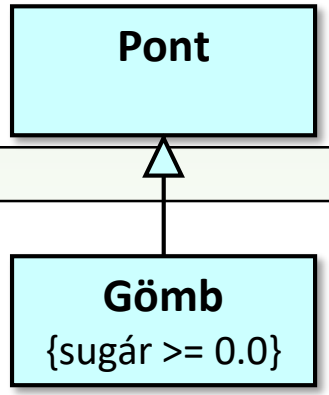
```
Point  p(0,0,0);
Sphere g(1,1,1,1);

cout << p.distance(p) << endl;
cout << g.distance(p) << endl;
cout << p.distance(g) << endl;
cout << g.distance(g) << endl;

cout << g.contains(p) << endl;
cout << g.contains(g) << endl;
```

```
class Point {
protected:
     double _x, _y ,_z;
public:
     Point(double a, double b, double c) : _x(a), _y(b), _z(c) {}
     double distance(const Point &g) const
          { return sqrt(pow((_x-g._x),2)+pow((_y-g._y),2)+pow((_z-g._z),2))
                    - radius() - g.radius();}
     virtual double radius() const { return 0.0; }
};
```

virtuális metódus




```
class Sphere : public Point {
private:
     double _radius;
public:
     enum Errors{ILLEGAL_RADIUS};
     Sphere(double a, double b, double c, double d) : Point(a,b,c), _radius(d)
          { if(_radius<0.0) throw ILLEGAL_RADIUS; }
     bool contains(const Point &p) const { return distance(p)+2*p.radius()<=0; }
     double radius() const override { return _radius; }
};
```

felülírt metódus