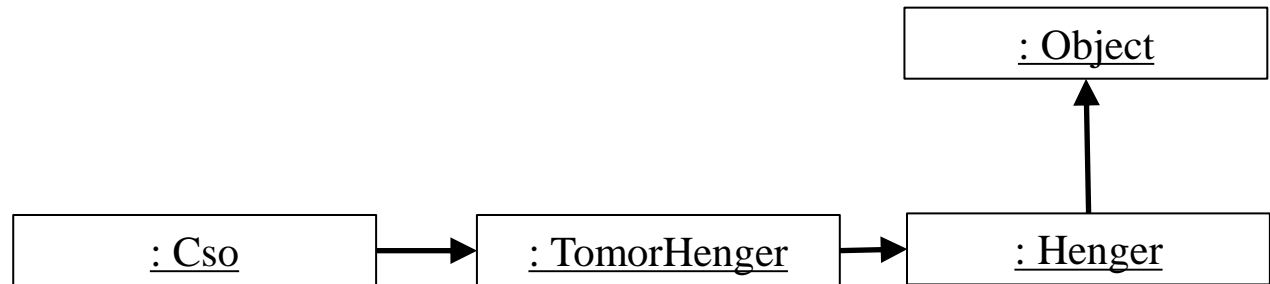


# Objektumreferencia típusai

- Statikus
- Dinamikus



- TomorHenger típusú objektumot azonosíthat:

- Object, Henger, TomorHenger

- Cso típus **NEM**

Object o = th;

Cso cso = th;

- TomorHenger referenciával azonosíthatunk:

- TomorHenger, Cso típusú objektumokat

- Henger vagy Object típusokat **NEM**

th1 = th2;

th = cso

th = o;

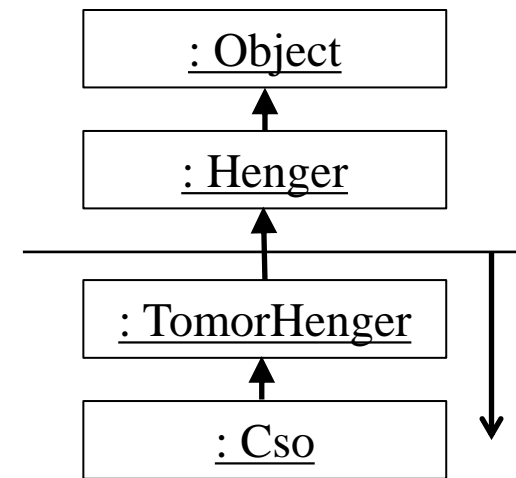
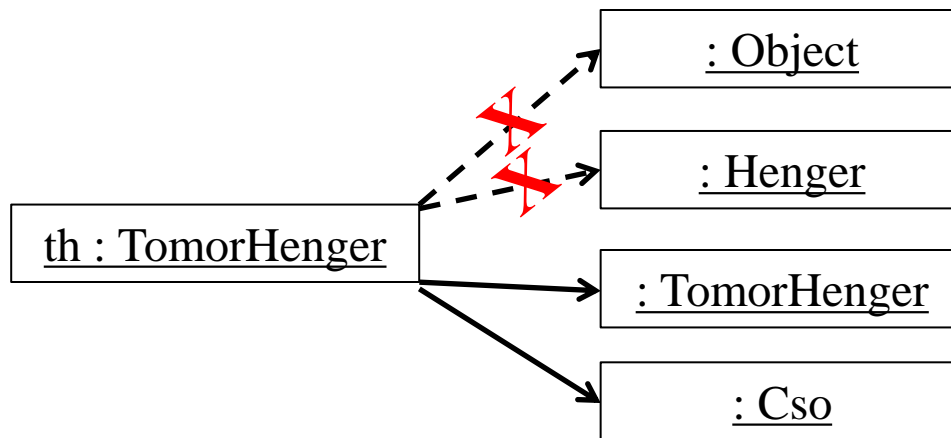
th = h;

# Deklarálások

- Statikus
  - `Object obj; //obj statikus típusa Object`
  - `Henger h1, h2;`
  - `TomorHenger th;`
  - `Cso cso;`
- Dinamikus
  - `obj = new Henger(1,3); //obj dinamikus típusa Henger`
  - `h1 = new TomorHenger(2,2,1.3);`
  - `th = new Cso(1,4,2);`
  - `obj = h1;`
  - `h2 = h1;`

# Objektumreferencia

- A referencia statikus típusa a dinamikussal egyezzen vagy annak őse legyen!
- Különben fordítási hiba!



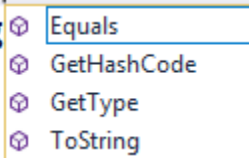
Ha `th` referencia **statikus** típusa **TomorHenger**, akkor **dinamikus** típusa csak **TomorHenger** vagy **leszármazottja** lehet

# Objektumreferencia

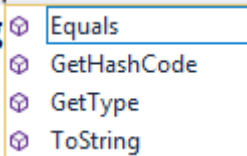
```
class Henger{ public int r, m; }
```

```
Object obj = new Henger();
```

obj.

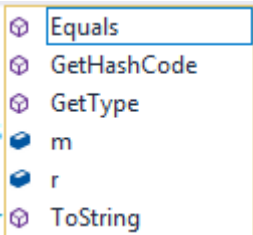
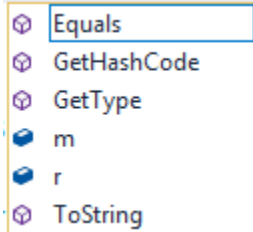


(Henger)obj.



```
Henger h = (Henger)obj;    ((Henger)obj).
```

h.

# Statikus – Dinamikus 1 cast

```

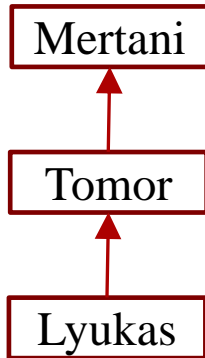
Mertani m1 = new Mertani(1,1);
Tomor t1 = new Tomor(2,2);
Lyukas l1 = new Lyukas(10,3,2);
Cw || sout(m1); //Mertani

m1 = t1; ✓
Cw || sout(m1); //Tomor

m1 = l1; ✓
Cw || sout(m1); //Lyukas
Cw || sout((Tomor)m1); //Lyukas

l1 = (Lyukas)m1;
Cw || sout(l1); //Lyukas

```

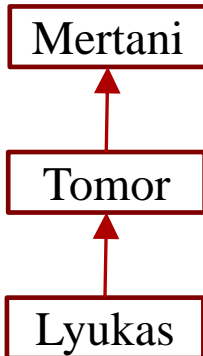


incompatible types:  
Mertani cannot be  
converted to Lyukas  
l1 = m1;

syntax ✓  
l1 = (Lyukas)m1;

runtime exception:  
**ClassCastException**  
**InvalidCastException**

# Statikus – Dinamikus 2 cast



```
Mertani m1 = new Mertani(1,1);
```

```
Tomor t1 = new Tomor(2,2);
```

```
Lyukas l1 = new Lyukas(10,3,2);
```

```
Object o = l1;
```

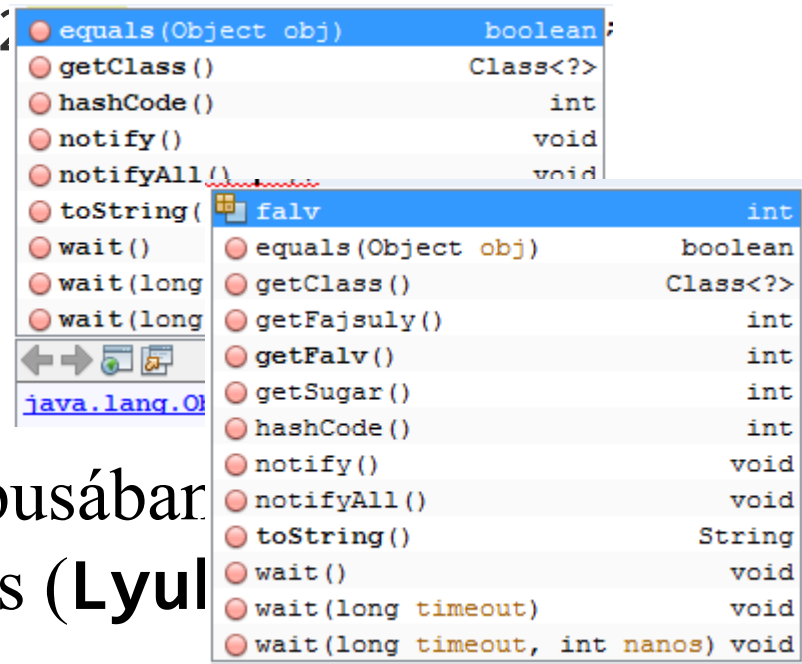
```
Cw || sout(o); //Lyukas
```

```
Object o = l1; Cw || sout(o.
```

**o statikus típusár**  
**dinamikus típus (Lyukas)**

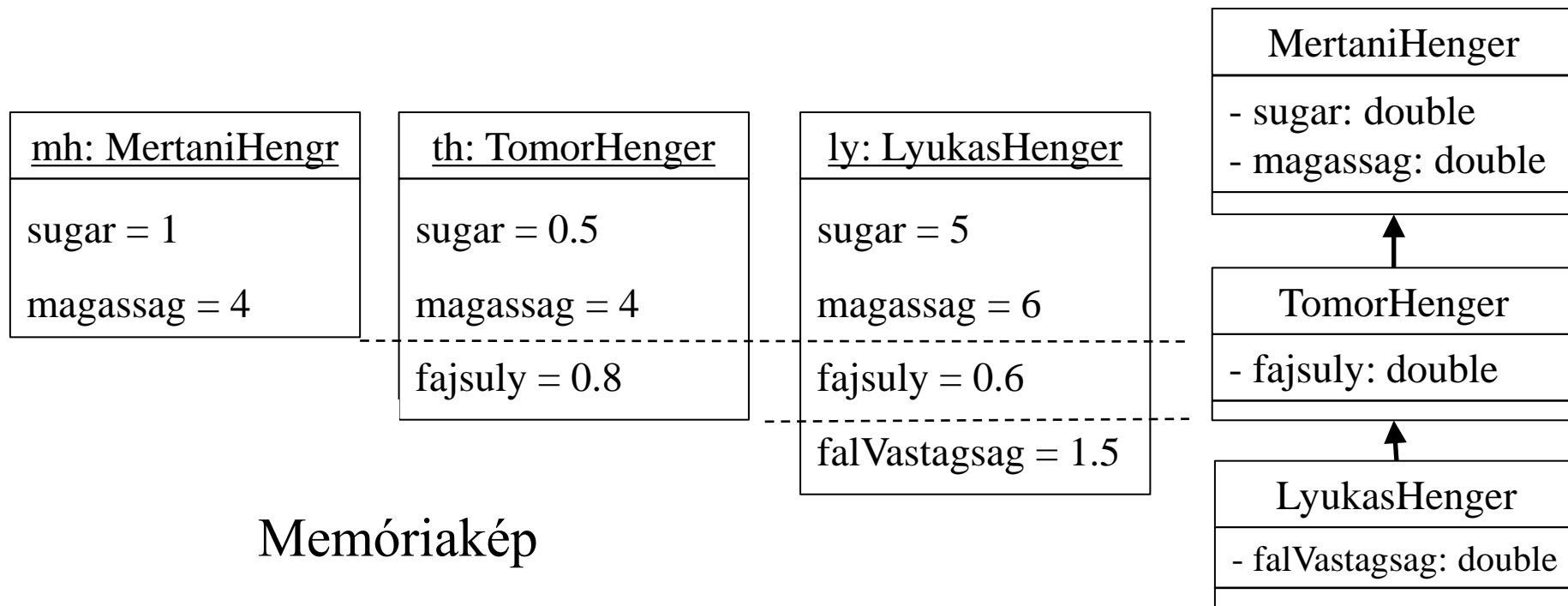
```
Object o = l1; Cw || sout ( ((Lyukas)o). );
```

```
Object o = l1; Cw || sout ( ((Tomor)o). );
```



# Utódosztály adatai, kapcsolatai

- A példányban helyet foglaló adatok: ősben deklaráltak + saját deklarációk
- Az utód akkor is örökli az ős adatait és kapcsolatát, ha nem látja azokat (private)



# Hivatkozás adatokra

```
Class C1{
    protected static String s = „ÖS”
    protected int a = 5 }
```

```
Class C2 extends C1 {
    public int b = 10
    void kiir(){ Cw||sout(s + „ ” + a + „ ” + b) ) }}
```

obj1 és obj2  
ugyanarra az  
objektumra mutat,  
de obj1 statikus  
típusában C1 ben,  
nincs b

---

```
C2 obj2 = new C2()
```

```
obj2.kiir()      Ös 5 10 //obj2.kiir()
```

```
Cw||sout(obj2.b) 10 //obj2.b
```

```
C1 obj1 = obj2
```

```
Cw||sout(obj1.a) Syntax error //obj1.a csak public elérhető
```

```
Cw||sout(obj1.b) Syntax error //obj1.b
```

**Eredmények?**  
**Miért?**



# Adatok elfedése

- Mint lokális változók
- Ha nem privát, akkor minősítéssel lehet elérni az ős, utódban elfedett adattagját
  - Példányadat esetén: super|base minősítő (a legközelebb felette állót érjük el, nem kell közvetlen ős)
  - Osztályadat esetén: osztály megnevezésével. Ha nincs ilyen adat, közvetlen felette keres

# AdatTakaras.java

Class **Os** {

public static String s = „Ös”;

public int a = 1; int b = 10;

}

Class **Kozepso** extends Os { public int a = 2; }

Class **Utod** extends Kozepso {

public static String s = „Utód”;

public int b = 30;

Utod osztályú objektumban  
lévő adattagok?

---

public void kiir1(){ cw||sout(Os.s + Kozepso.s + s); }

public void kiir2(){ cw||sout(Os.a + Kozepso.a + Utod.a); }

public void kiir3(){ cw||sout(super/base.a + a); }

public void kiir4(){ cw||sout(super/base.b + b); }

}

**Eredmények?**  
**Miért?**

# Eredmények

Kiir1(): Os Os Utod

Kiir2(): syntax error, példányadatot nem lehet osztállyal minősíteni

Kiir3(): 4, mindkettő hivatkozás Kozepso ben deklarált a-t jelenti

Kiir4(): 40, super.b a Os b –je., b pedig a saját

Utod osztályú objektumban lévő adattagok:

Os::a, Os::b, Kozepso::a, Utod::b

**A statikus Os.s és Utod.s adatokat az objektum nem tárolja, az osztályban foglal helyet**

# Metódusok 1


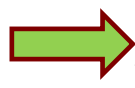
- **Felülírás** / *overriding*: az utód az ősbármely *virtuális* példánymetódusát felülírhatja
  - A szignatúra megegyezik és a típus is (különben **túlterhelés** / *overloading*)
  - Private : nem is látszik, ne írjunk ilyen néven mást
  - Final / sealed: ősből nem módosítható deklaráció
    - Osztályból nem lehet örökölni
  - Static: csak példánymetódust lehet felülírni

# Metódusok 2

- Dinamikus kötés / *dynamic binding*(futás alatti / *runtime binding*, késői kötés / *late binding*): osztályhierarchia bármelyik szintjéről hívunk meg példánymetódust, mindig a megszólított objektum osztályában deklarált metódus fog végrehajtódni

- Fordításkor ez nem derül ki, csak futási időben

C#: láthatóság nem módosítható, Java:

- Láthatóság nem szűkíthető: *public*  *protected*
- Láthatóság bővíthető: *protected*  *public*  
*private* nem is látható...

# Késői kötés

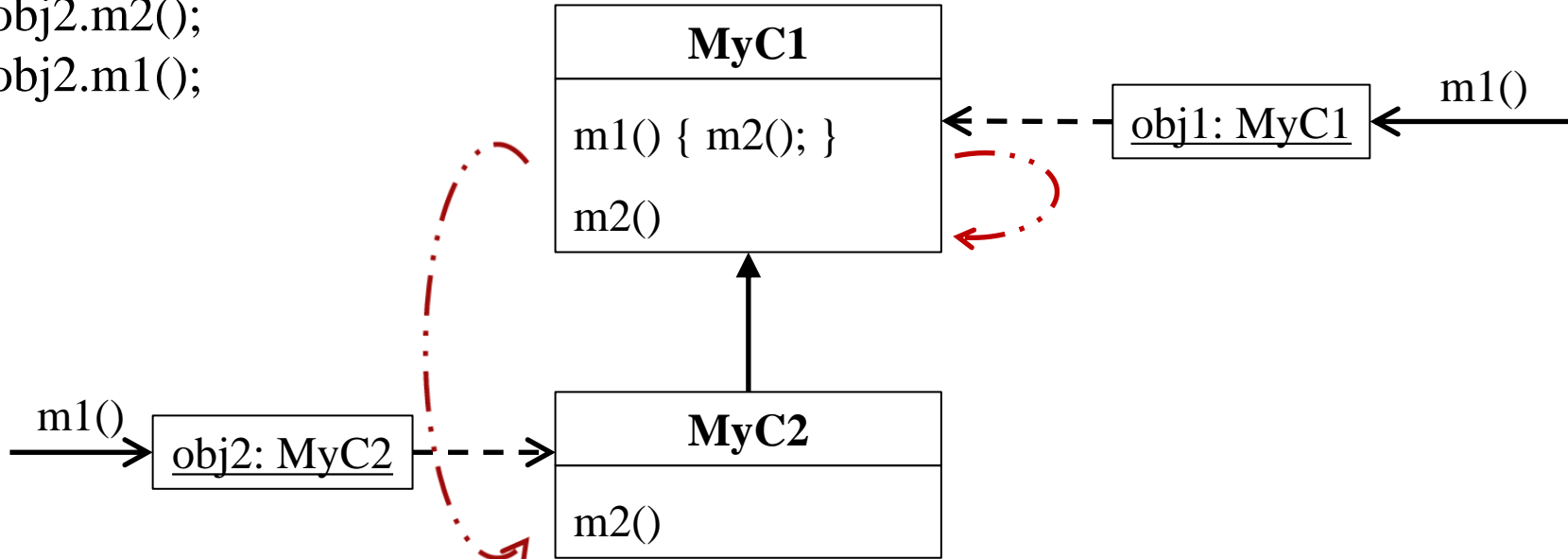
```
MyC1 obj1 = new MyC1();  
MyC1 obj2 = new MyC2();
```

```
obj1.m1();  
obj1.m2();
```

```
obj2.m2();  
obj2.m1();
```

**Implementáld!**

**Debuggerrel megnézni**



## 2.: statikus m2 vel is

```

1  package kesoikotes;
   class C1 {
3      void m1 () {m2 ();}
   static void m2 () {System.out.println("C1.m2");}
5  }
6  class C2 extends C1 {
   static void m2 () { System.out.println("C2.m2");}
8  }
9  public class Kesoikotes {
10     public static void main(String[] args) {
11         C1 obj1 = new C1();
12         C2 obj2 = new C2();
13         obj1.m2 (); //C1.m2
14         obj2.m2 (); //C2.m2 <--
15         obj1.m1 (); //C1.m1
16         obj2.m1 (); // C1.m2 <--
17     }

```

# this, super|base referenciák

**this**: rejtett paraméter, memóriacím, ami a megszólított obj. *referenciája*.

A példánymetódus innen tudja, hogy melyik példányon dolgozik.

Implicit paraméterként minden konstr. és pldmetódus megkapja

Osztálya mindig a megszólított obj. Osztálya

Ekvivalens hívások:

- `kiir();`
- `this.kiir();`



# this, super|base referenciák

**Super|base:** az éppen működő objektum lebutított referenciája, típusa a közvetlen ős.

NINCS: `super.super`

```
Public double terfogat(){
    Henger belso = new Henger(getSugar() – falVastagsag,
    getMagassag() )
    return super.terfogat() – belso.terfogat();}
```

Osztálysztintű elérésre nincs `this`, `super`. Osztály nevével kell hivatkozni

```
Henger.getHengerDarab()
```

# Polimorfizmus

- Ugyanarra az üzenetre különböző típusú objektumok különbözőképpen reagálnak. Minden obj. a saját metódusával
- A `terfogat()` üzenet kül. objektumok esetén más és más alakot ölt
- `toString()` is polimorfikus
- A statikus referencia osztályában már szerepeljen a metódus, a dinamikus típustól függően más és más metóduslánc fog végrehajtódni

# Polimorfizmus

- Az utódban felülírt metódus fog végrehajtódni
- Polimorfizmus létezik futás alatti kötés nélkül is

```
double osszTerfogat = 0 ;  
Henger henger;  
for(int i = 0; i < hengerek.size(); i++){  
    henger = (Henger) ( hengerek.get(i) );  
    osszTerfogat += henger.terfogat();  
}
```

Mikor kell?

ArrayList hengerek;

Mikor **NEM** kell?

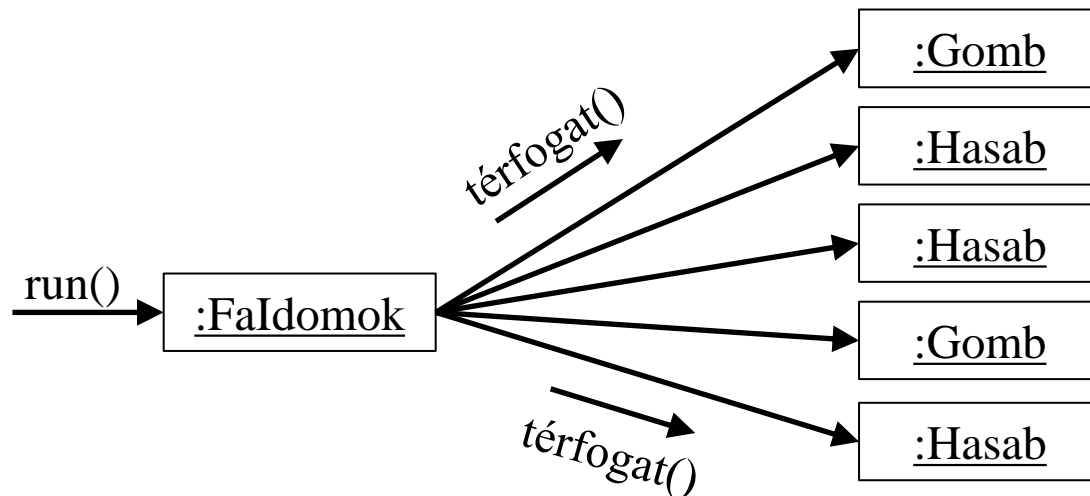
ArrayList<**Henger**> hengerek;

# Absztrakt metódus, osztály

- Bizonyos metódusok nem lettek implementálva
- Ekkor absztrakt a metódus, amit absztrakt osztály tartalmazhat
- Abstr. osztályban nem muszáj abstr. Metódus
- Abstr. Osztály nem lehet final, nem példányosítható
- Abstr. Metódusnak nincs blokkja

# Feladat

- 2 féle *idom*: **gömb** és **hasáb**, különböző méretekben
- A program tudja:
  - Idomok összes súlyát
  - Gömbök összes súlyát
  - Legkisebb és legnagyobb térfogatú idom típusát, és adatait



**UML (absztrakt)?**

**FaIdomok program**

**Együttműködési diagram**

# Falandomok UML

