Programozási nyelvek és módszerek

6. ELŐADÁS – OOP ISMÉTLÉS

Kivételkezelés

C++

Kivétel lehet egy objektum, aminek tetszőleges a típusa.

Hasznos, ha definiálunk osztályokat a felhasználó kivételeihez

Például

- `class MyException {};`

C++

Néhány predefinit kivétel:

- `bad_cast`: ha a dynamic_cast operator nem használható egy referenciára,
- `bad_typeid`: ha a typeid operátora egy null pointer dereferenciája.
  (mindkettő a typeinfo.h-ban deklarálva).

C++

Kivétel kiváltása

- `throw [expression] ";"`
  A kifejezés egy időszakos objektumba kerül.
- A normál program lefutás megszakad, a vezérlés a legközelebbi megfelelő kezelőhöz kerül
- Ha nincs kifejezés: csak kivételkezelőben, illetve innen hívott függvényben lehet, az aktuális kivétel újrakiváltása.
- A fordító nem biztos, hogy ellenőrzi, ha `throw` utasítás kivétel-objektum nélkül: `terminate` függvény hívása.

C++

Példák

- `throw 5;`
- `throw "An exception has occurred";`
- `throw MyException();`
- `throw;`

További információk: a kivételosztályban lehetnek nyilvános attribútumok, konstruktor beállíthatja stb.

- ```
  class ExceptionWithParameters {
    public:
     int a,b;
     ExceptionWithParameters(int a0,int b0)
                                  { a=a0; b=b0; }
     };
     // ...
     throw ExceptionWithParameters(0,1);
  ```
  ```

C++

Kivételek kezelése - try blokkal:

- `try compound_statement handler_list`
  `ahol`
  `handler_list = handler{handler}`
- `handler = catch "(" exception_declaration ")"`
  `compound_statement`
  `exception_declaration = type [ident] |"..."`

C++

A dobott kivételeknek megfelel egy catch ág, ha a következő feltételek közül valamelyik teljesül:

- A két típus pont ugyanaz.
- A catch ág típusa publikus bázisosztálya az eldobott objektumnak.
- A catch ág típusa mutató, és az eldobott objektum olyan mutató, melyet valamely standard mutató konverzióval át lehet konvertálni a catch ág típusára.

A kezeletlen kivételek továbbgyűrűződnek a hívó try blokkban, majd az azt hívóban.

- A legkülső try blokk után a *terminate* függvény kerül meghívásra.

C++

Például: class A {};

- ```
  class B: public A {};
  class C: public B {};
  class D: public A {};
  class X {};
  class AX: public A, public X {};
  catch (A) // A,B,C,D, AX típusúakat kap el
  catch (A*) // A*,B*,C*,D*,AX*
  catch (X&) // X és AX
  catch (const B&) // B és C
  catch (char*) // char*
  catch (void*) // tetszőleges pointer típusút
  ```
  ```

C++

Mindig az első kezelőt választja ki

- ha rossz sorrendet írunk, nem hiba!
- ```
  try { // ...
    }catch (A) {/* ... */}
     catch (B) {/* ... */}//soha nem jön ide!
  ```
- ```
  try {// ...
    }catch (B) { /* ... */ }
     catch (A) { /* ... */ } // így OK.
  ```

C++

Lehet a kivételobjektumra is hivatkozni:

- ```
  catch (ExceptionWithParameters e) {
      cout<< e.a; // kiírja
  }
  ```

Lehet ...

- ez bárminek megfelel - utolsó legyen a try-blokkban

A `terminate` hívása általában abort-ot jelent

- A felhasználó a `set_terminate`-tel megadhat mást is
- ez is le kell állítsa a programot.

Miközben a vezérlés átadódik a kivételkezelőnek, destruktort hív minden automatikus objektumra, ami a try-blokkba való belépés óta keletkezett.
  ```

C++

Kivétel specifikációk

- `throw "(" [type {"," type}] ")"`
- Például: `void f(int x) throw(A,B,C);`

Az f függvény csak A,B és C típusú kivételt generál, vagy azok leszármazottait.

- `int f2(int x) throw ()` - f2 nem válthat ki kivételt!
- `void f3 (int x)` - f3 bármit kiválthat!

Ha specifikáltunk lehetséges kivételtípusokat, akkor minden más esetben a rendszer meghívja az `unexpected()` függvényt

- Az alapértelmezett viselkedése a `terminate()` függvény meghívása
- Ez a `set_unexpected` segítségével szintén átállítható.

C++

Példák:

- ```
  class X {};
  class Y: public X {};
  class Z {};
  void A() throw(X,Z) // A X,Y,Z-t dobhat
  { /* ... */ }
  void B() throw(Y,Z)
  {
    A(); // unexpected, ha A X-t dob, ok  Y,  Z típusúra
  }
  void C(); // semmit sem tudunk C-ről
  ```
- ```
  void D() throw()
    {
      C(); // ha  C bármit kivált, unexpected -t hív
           throw 7; // unexpected-t hív
    }
  ```

Java

```
try {
    …
     throw new EgyException (“parameter”);
}
catch (típus változónév) {
     ...
}
finally {
     ...
}
```

Java

A C++ szintaxistól nem sokban különbözik.

- Eltérések a szemantikában.

A továbbiakban csak az eltérések:

- `finally` nyelvi konstrukció, ezzel a Java megbízhatóbb programok írását segíti elő.

Java

Egy függvény által kiváltható kivételek specifikálása:

- `void f (int x) throws EgyikException, MasikException;`

Egy szálon futó program esetén ha a virtuális gép nem talál a kivétel kezelésére alkalmas kódot, akkor a VM és a program is terminál.

- Ha többszálú a program, akkor egy `uncaughtException()` metódus fut le.

Java

Minden kivétel a `java.lang.Throwable` leszármazottja.

Ha olyan kivételt szeretnénk dobni, amely nem a `Throwable` leszármazottja, akkor az fordítási hibát okoz.

A kivételek két nagy csoportba sorolhatóak

- ellenőrzöttek: `Exception` leszármazottjai
- nem-ellenőrzöttek: `Error` leszármazottjai

Java

Azért volt arra szükség, hogy a kivételeket a fenti két csoportba sorolják, mert számos olyan kivétel van, amely előre nem látható és fölösleges lenne mindenhol lekezelni őket.

- Ezeket a kivételeket nevezzük nem-ellenőrzött kivételeknek.
- Például nem ellenőrizzük le minden utasítás végrehajtása előtt, hogy van-e elég memóriánk stb.

Java

Sajnos, a Java a fenti két csoportosítást nem konzisztens módon végzi

- Az `Exception` egyik gyermek osztálya, a `RuntimeException` és leszármazottjai sem ellenőrzöttek.

Az ellenőrzött kivételek esetén fordítási hiba lép fel, ha nincsenek specifikálva vagy elkapva.

- Továbbá ha olyan ellenőrzött kivételt kívánunk elkapni, amely hatókörön kívül van.

Vermes példa

```
class VeremException extends Exception {}
class VeremMegteltException extends VeremException {
      private int utolso;
      public VeremMegteltException (int i) {
            utolso = i;
      }
      public int miVolt () {
            return utolso;
      }
}
```

Vermes példa

```
class Verem {
   final static public int MERET = 10;
   private int tarolo [] = new int [MERET];
   private int mutato = 0;
   public void betesz (int i) {
     try {
       if (mutato < MERET)
          tarolo [mutato++] = i;
       else
          throw new VeremMegteltException (i);
     } catch (VeremMegteltException e) {
        System.out.println (e.miVolt () + “ nem fert be”); throw;}
     finally {
       System.out.println (“finally: mindig lefutok!”);
     }
   }
}
```

Mi a gond ezzel?

Java

Néhány predefinit nem ellenőrzött kivétel

A `RuntimeException` leszármazottjai

- `ArithmeticException`
- `ClassCastException`
- `IndexOutOfBoundsException`
  - `ArrayIndexOutOfBoundsException`
  - `StringIndexOutOfBoundsException`
- `NullPointerException`.

Az `Error` leszármazottjai

- `OutOfMemoryError`
- `StackOverflowError`
- stb.

Java

Predefinit kivételek előfordulása:

```
class A {// ...
}
class B extends A {// ...
}
```

Java

```
class C {
    void X() {
      A a= new A;
      B b= (B)a; // ClassCastException
    }
    void Y() {
      int ia[]= new int[10];
      for (int i=1; i<=10; i++)ia[i]=0;
   /* amikor i==10 lesz,  ArrayIndexOutOfBoundsException  */      }
    void Z() {
      C c=null;
      c.X(); // NullPointerException
    }
  }
```

Java

Kivételek kezelése

```
try {
  // utasítások
}
catch (MyException e) {
  // utasítások MyException kezelésére
}
catch (AnotherException e) {
  // utasítások AnotherException kezelésére
}
finally {
  // mindig végrehajtódó utasítások
}
```

Java

```
class A extends Exception {}

class B extends A {} ...

try {
    // ...
  }
  catch (A a) { /* ... */ }
  catch (B b) { /* ... */ }

void MyMethod() throws MyException {
    // utasítások
    throw new MyException();
    // utasítások
  }
```

Java

Példa

```
class ExcA extends Exception {}
void A() throws ExcA {
    // ...
    throw new ExcA();
    // ...
}
```

Java

```
void B1() {
    A(); // Hiba: ExcA nincs lekezelve B1-ben
}

void B2() {
  try {
     A(); // OK
  } catch (ExcA e) {
     // exception kezelése
  }
}
```

Java

```
void B3() throws ExcA {
    A(); // OK
}

class ExcD extends RuntimeException {}
  void D() throws ExcD {
    // ...
}
```

Java

```
void E() {
    D(); // ok, ExcD leszármazottja
         //RuntimeException-nak, így nem kell jelezni
}

class ExcF extends ExcA {}
    void F() throws ExcA {
      throw new ExcF();   // OK
}
```

Eiffel

Újrakezdés

- A komponens írásakor egy kivétel lehetőségét előre lehet látni és egy alternatív megoldást találni a szerződés betartatására.
- Ekkor a végrehajtás megpróbálja ezt az alternatívát.

Szervezett pánik

- Ha nincs rá mód, hogy teljesítsük a szerződést, akkor az objektumokat egy elfogadható állapotba kell hozni (típusinvariáns helyreállítása), és a felhasználónak jelezni kell a kudarcot.

Eiffel – Újrakezdés

```
try_once_or_twice is
  local
    already_tried : BOOLEAN
  do
    if not already_tried then
      method_1
    else
      method_2
    end
  rescue
    if not already_tried then
      already_tried := true;
      retry
    end
 end -- try_once_or_twice
```

Példa

```
transmit: (p: PACKET)
-- Transmit packet `p'
require
      packet_not_void: p /= Void
local
      current_retries: INTEGER
      r: RANDOM_NUMBER_GENERATOR
do
      line.send (p)
rescue
      if current_retries < max_retries then
            r.next
            wait_millisecs (r.value_between (20, 500))
      current_retries := current_retries + 1
            retry
      end
end
```

Eiffel – Szervezett pánik

```
attempt_transaction(arg : CONTEXT) is
  -- megpróbáljuk a transaction-t arg argumentummal;
  -- ha nem sikerül, az akt. obj. invariánsát visszaállítjuk
  require
     ...
  do
     ...
  ensure
     ...
  rescue
     reset(arg)
 end -- attempt_transaction
```

Eiffel

```
default_rescue is
do
end --default_rescue

class C creation
  make ...
inherit
  ANY
    redefine default_rescue end
  ...
feature
  make, default_rescue is
     -- nincs előfeltétel
    do ...
    end;
  ...
end -- class C
```

C# - .NET

Közös elv alapján a .NET-ben

Nagyon hasonlít a Javához, de van különbség is

Hasonlóság:

- try – catch blokk, (ellenőrzi a jó sorrendet)
- finally lehetősége
- közös ősosztály (System.Exception)

Különbség:

- Nincs exception-specifikáció

C# - .NET

Miért nincs a C#-ban ellenőrzött kivétel?

- Verziókezelés
  - Ha egy következő verzióban egy metódus új kivételt dob, akkor az őt hívó metódust is változtatni kell
  - Ez máshol is probléma, ha le akarjuk kezelni az új kivételt, de legtöbbször nem kezelik
  - 10 az 1-hez a try-finally és a try-catch aránya

Irodalom: http://artima.com/intv/handcuffsP.html

C# - .NET

Miért nincs a C#-ban ellenőrzött kivétel?

- Méretezhetőség:
  - Nagy projektekben, négy-öt alrendszerrel a sok kivétel már kezelhetetlenné válik
  - Ilyenkor keletkezik sok catch {} üresen
- Nem szigorú szabályok kellenek a kezeletlen kivételek ellen, hanem elemző eszközök, amelyek felkutatják a gyanús kódokat, lehetséges réseket

C# - .NET

Példák:

```
using System;
class ExceptionTestClass {
       public static void Main() {
          int x = 0;
                  try {
                        int y = 100/x;
     }
     catch (ArithmeticException e) {
       Console.WriteLine("ArithmeticException Handler: {0}",
                  e.ToString());
     }
     catch (Exception e) {
               Console.WriteLine("Generic Exception Handler: {0}",
                                  e.ToString());
     }
   }
}
```

C# - .NET

```
using System;
class ArgumentOutOfRangeExample {
   static public void Main() {
      …
      try {
      …
      }
      catch (ArgumentOutOfRangeException e) {
       Console.WriteLine("Error: {0}",e);
      }
      finally {
        Console.WriteLine(„It is always executed.");
      }
   }
}
```

C# - .NET

Saját exception-osztály:

```
using System;
public class EmployeeListNotFoundException:
    ApplicationException {
public EmployeeListNotFoundException() { }
public EmployeeListNotFoundException(
            string message) : base(message) { }
public EmployeeListNotFoundException(
            string message, Exception inner) :
    base(message, inner) { }
}
```

Delphi

Eltérő szintaktikára példa

```
begin
  Try
    // The code we want to execute
    ...
  Except
    // Exception handling
    ...
  Finally
    // Finally block
    ...
  end;
end;
```

Delphi

Több kivételtípus kezelése

```
except
    // IO error
    On E : EInOutError do
      ShowMessage('IO error : '+E.Message);
    // Dibision by zero
    On E : EDivByZero do
      ShowMessage('Div by zero error : '+E.Message);
    // Catch other errors
    else
      ShowMessage('Unknown error');
 end;
```

Delphi

Kivétel kiváltása

- `raise object at address`
- Példa
  - `raise EMathError.Create;`
  - `raise Exception.Create at @MyFunction;`

SWIFT

A kivételeket az Error protokolt megvalósító felsorolási típusok segítségével reprezentálja

- SWIFT esetén a protokoll megközelítőleg a Java interfész fogalomnak felel meg

Például

```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

SWIFT

Az egyes esetek paraméterezhetők, amivel a hiba / kivétel specifikusabban megadható.

Kiváltani a throw utasítással lehet

- `throw VendingMachineError.insufficientFunds(coinsNeeded: 5)`

SWIFT

Kivételek kezelése

- Egyrészt, a Java-hoz hasonlóan egy függvénynek jeleznie kell, hogy kivételt dobhat
  - `func canThrowErrors() throws -> String`
  - `func cannotThrowErrors() -> String`
- Ha kritikus függvényt hívunk, akkor `try` kulcsszóval kell tenni
  - `try canThrowErrors()`

SWIFT

A kapott kivételt vagy kezelni kell, vagy jelölni, hogy tovább tud dobódni.

Kivétel kezelés:

```
do {
    try buyFavoriteSnack(…)
} catch VendingMachineError.invalidSelection {
    …
} catch VendingMachineError.outOfStock {
    …
} catch VendingMachineError.insufficientFunds(let c) {
    …
}
```

SWIFT

További lehetőségek

- Értékadásnál, ha nem sikerült, akkor null értéket ad az új változónak
- Ezt később lehet kezelni
  - Optional – formájában

```
let x = try? someThrowingFunction()
```

  - Összeköthető feltétel vizsgálattal is

```
func fetchData() -> Data? {
    if let data = try? fetchDataFromDisk()   { return data }
    if let data = try? fetchDataFromServer() { return data }
    return nil
}
```

SWIFT

További lehetőségek

- Amennyiben biztosak vagyunk, hogy a hívott függvény valóságban nem dobhat kivételt akkor

```
let photo = try! loadImage(atPath: "./…")
```

- Ebben az esetben, ha mégis keletkezik kivétel, akkor a futás megszakad és egy futás idejű hiba keletkezik

SWIFT

Hibától függetlenül lefutó kód

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            …
        }
        // close(file) is called here, at the end of the scope.
    }
}
```

- Tehát a defer részben megadott hívás a blokk befejeződése előtt lefut

OOP I.

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

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

OOP – alapelvek

OOP Alapelvek (Benjamin C. Pierce)

- Dynamic binding (dinamikus kötés)
  - Egy objektum esetén dinamikusan, futási időben dől el, hogy egy metódus melyik implementációja kerül futtatásra
- Encapsulation (egységbe zárás)
  - Adatok és rajtuk végrehajtható műveletek egységet alkotnak
  - Praktikusan ez a modern típus-definícióval konzisztens
- Subtype polymorphism (altípusos polimorfizmus)
  - Egy rögzített típusú változó több a típus altípusának példányára is hivatkozhat
  - Altípus
    - Az eredeti típus megszorításával létrehozott új típus
- Inheritance, vagy delegation (öröklődés, delegáció)
  - Egy adott osztályból lehetőség van képezni egy másik osztályt
    - Ez az ős tulajdonságait megtartja
    - Azonban módosíthatja, bővítheti
- Open recursion (nyílt rekurzió)
  - Speciális változó, amely egy metódus esetén lehetővé teszi az aktuális példány elérését

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

Osztály, példány

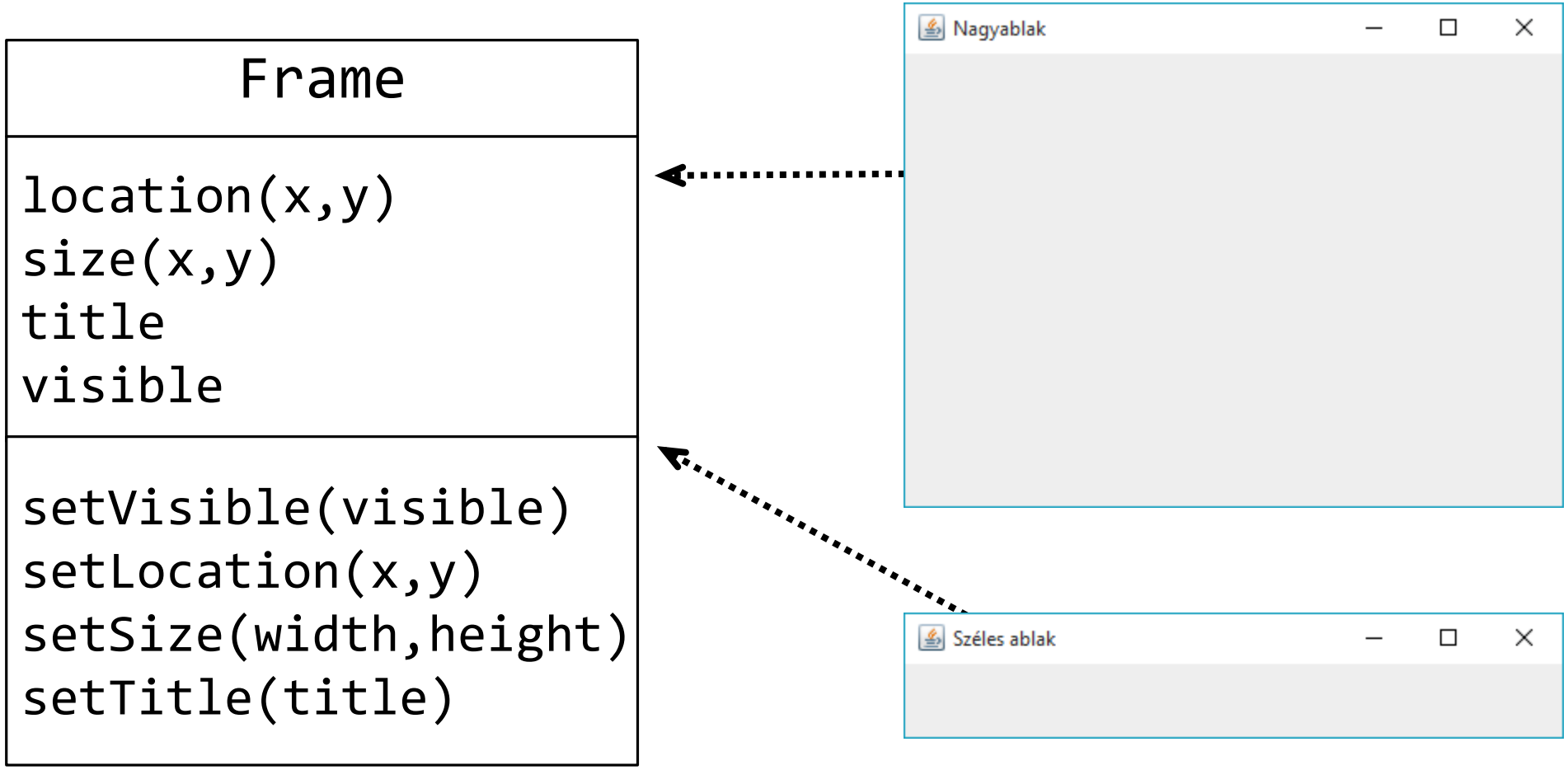
Osztály (class)

- Olyan objektumminta vagy típus, mely alapján példányokat (objektumokat) hozhatunk létre

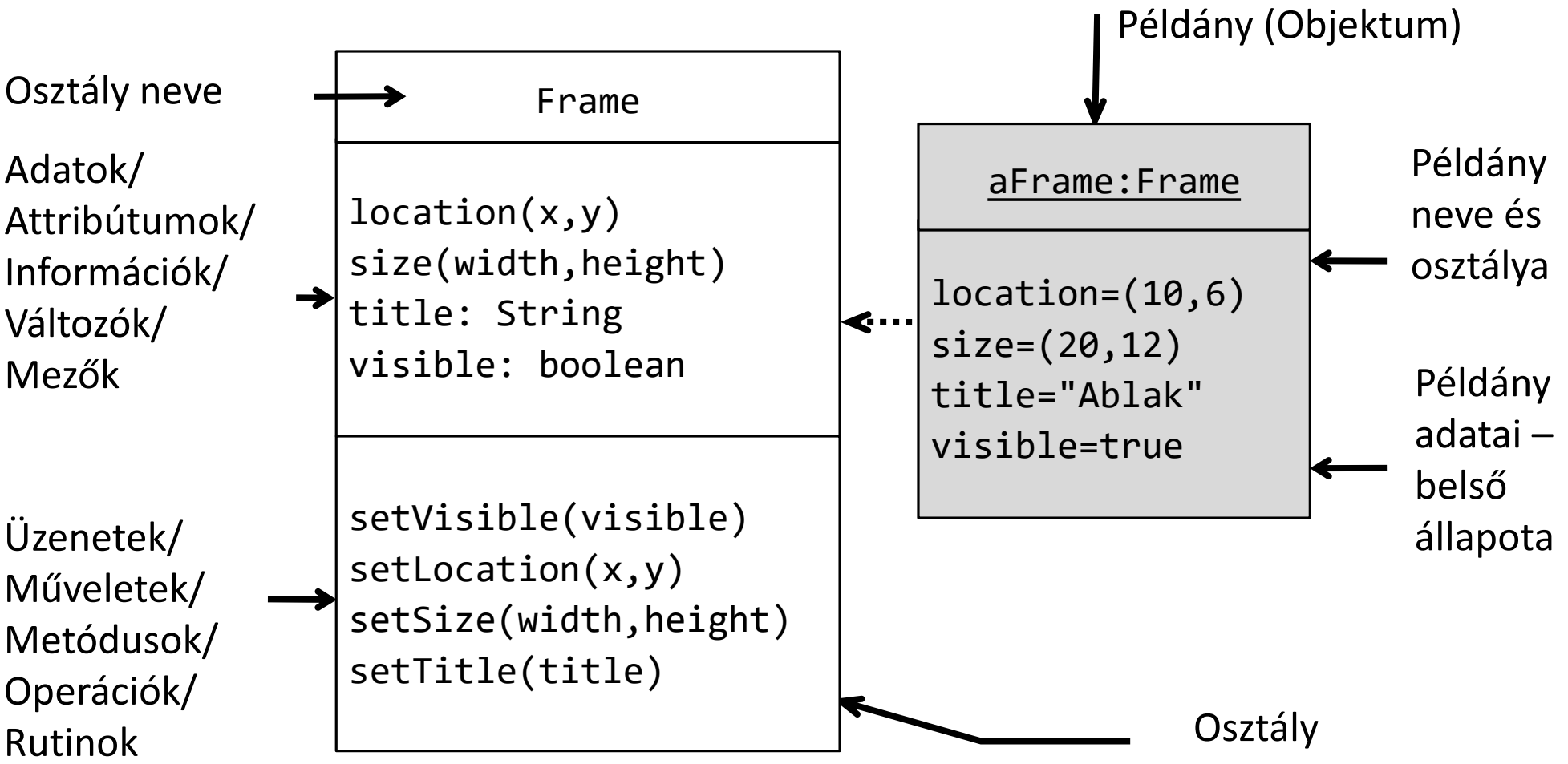
Példány (instance)

- 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

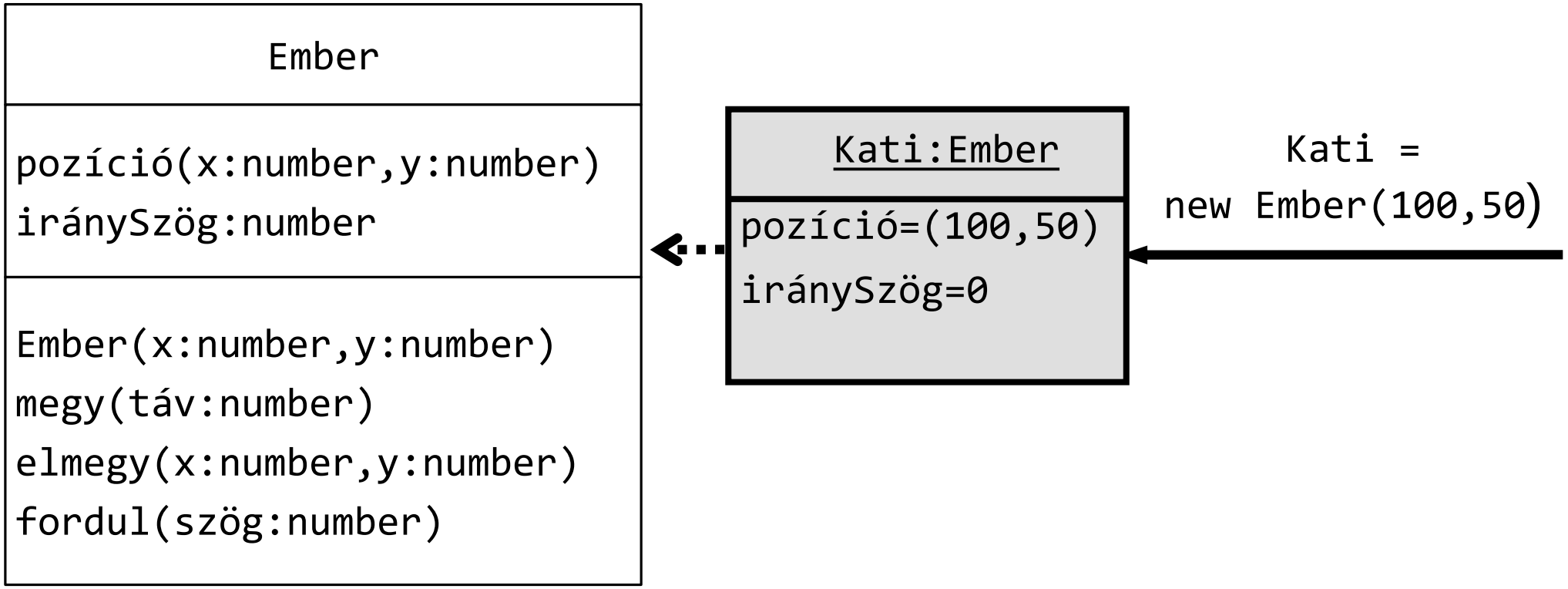
Objektum életciklusa:
„megszületik”, „él”, „meghal”

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 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 explicite lehet hívni a delete operátorral, vagy implicit hívódik a blokkból való kilépéskor – fordított sorrendben

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

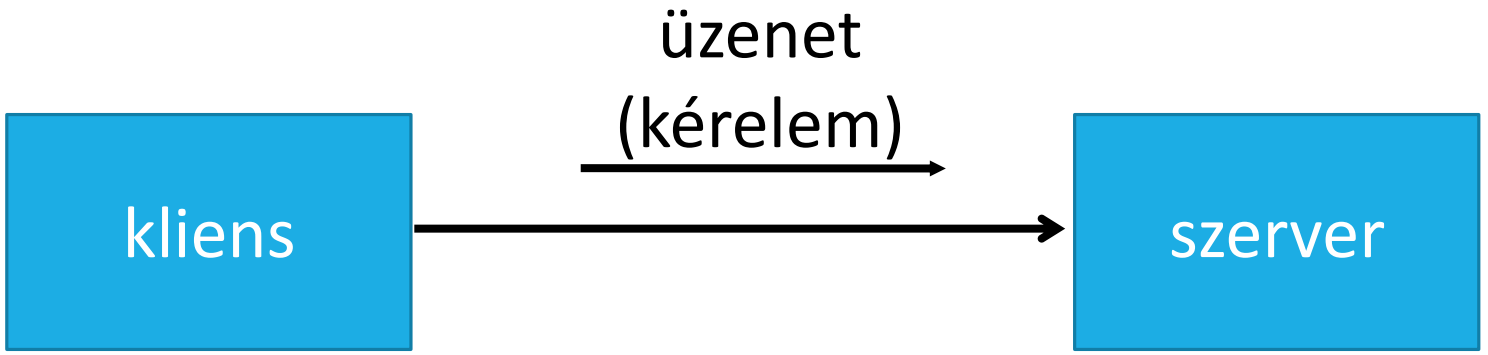
Szerver

- Passzív objektum
- Csak export felülete van
- Másoktól érkező üzenetekre vár, mások szolgáltatását nem igényli

Ágens

- Általános objektum, van export és import felülete

Kliens üzen a szervernek



A feladatot elvégeztető objektum

Kívülről elérhető metódus hívása

A feladatot elvégző objektum

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: $\rightarrow \mathrm{Verem}$
- Állapot megváltoztató
  - Példa verem esetén
    - pop: $\mathrm{Verem} \rightarrow \mathrm{Verem} \times \mathrm{Elem}$
    - push: $\mathrm{Verem} \times \mathrm{Elem} \rightarrow \mathrm{Verem}$

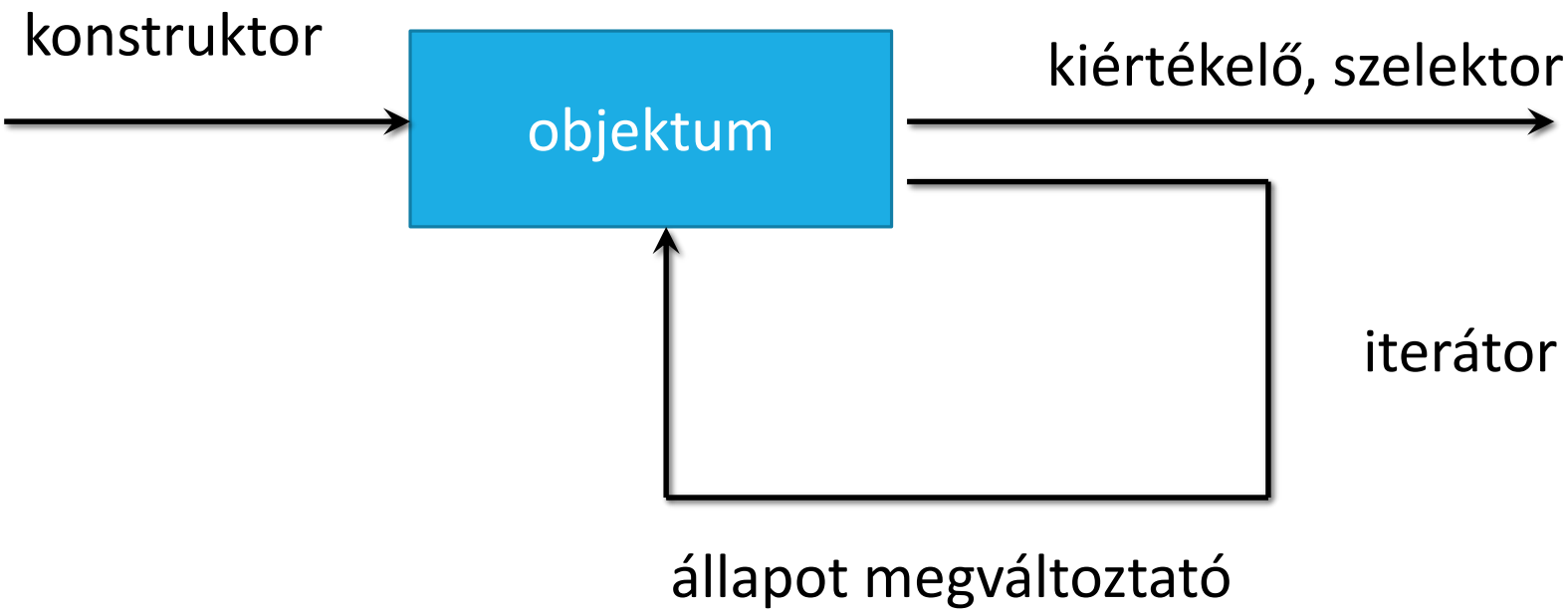
Objektum műveletei

Export műveletek csoportosítása:

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

Objektum műveletei

Export műveletek csoportosítása:



Osztály, példány

Minden objektum?

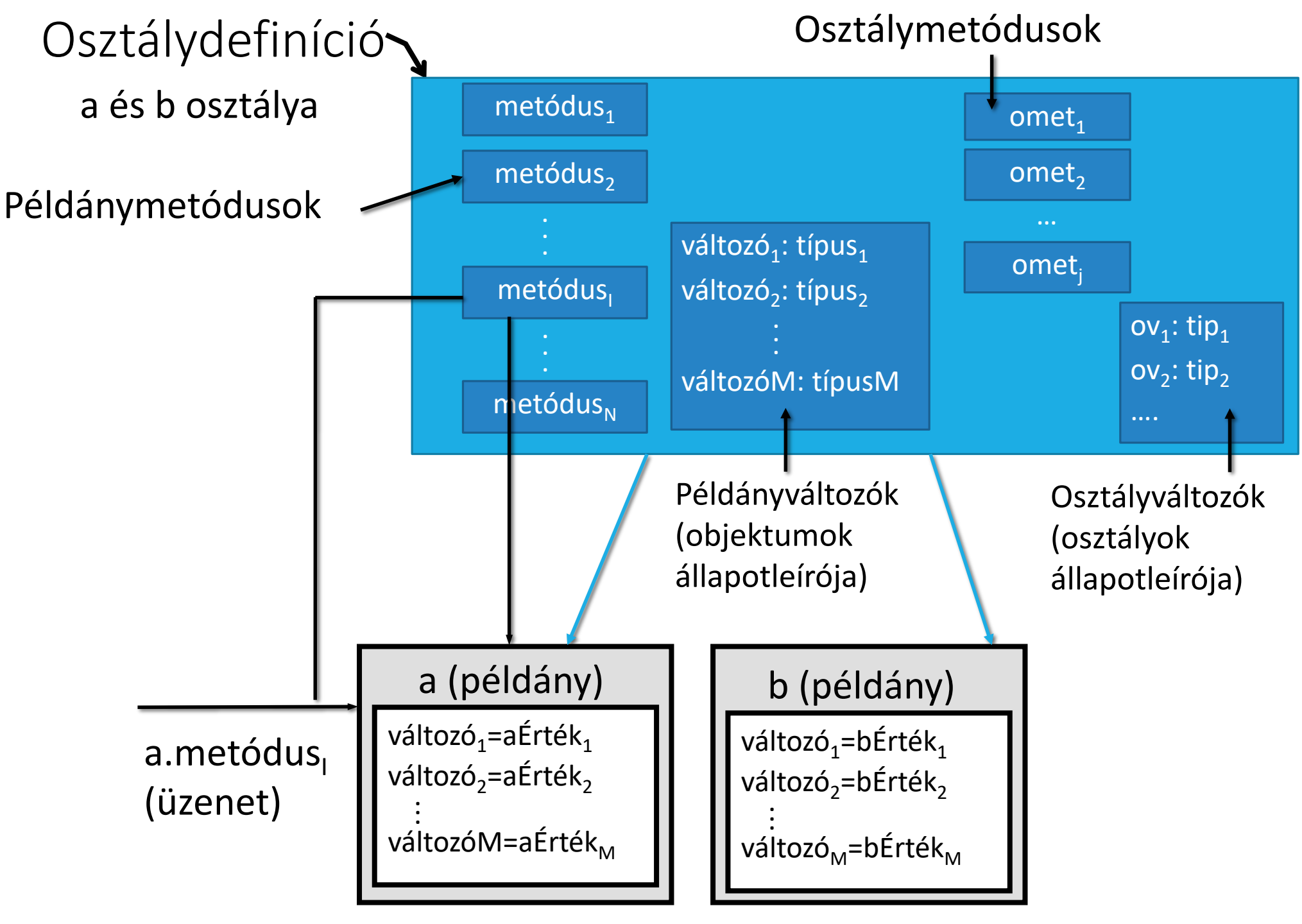
- Akkor az osztályok is …
- Lehet belső állapotuk,
- Küldhetünk üzeneteket neki …
- Minek az objektuma?
  - Metaosztály
    - Singleton objektum

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
a és b osztálya
Példánymetódusok
Osztálymetódusok
metódus1
metódus2
metódusI
metódusN
omet1
omet2
...
ometj
változó1: típus1
változó2: típus2
változóM: típusM
ov1: tip1
ov2: tip2
....
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)

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.

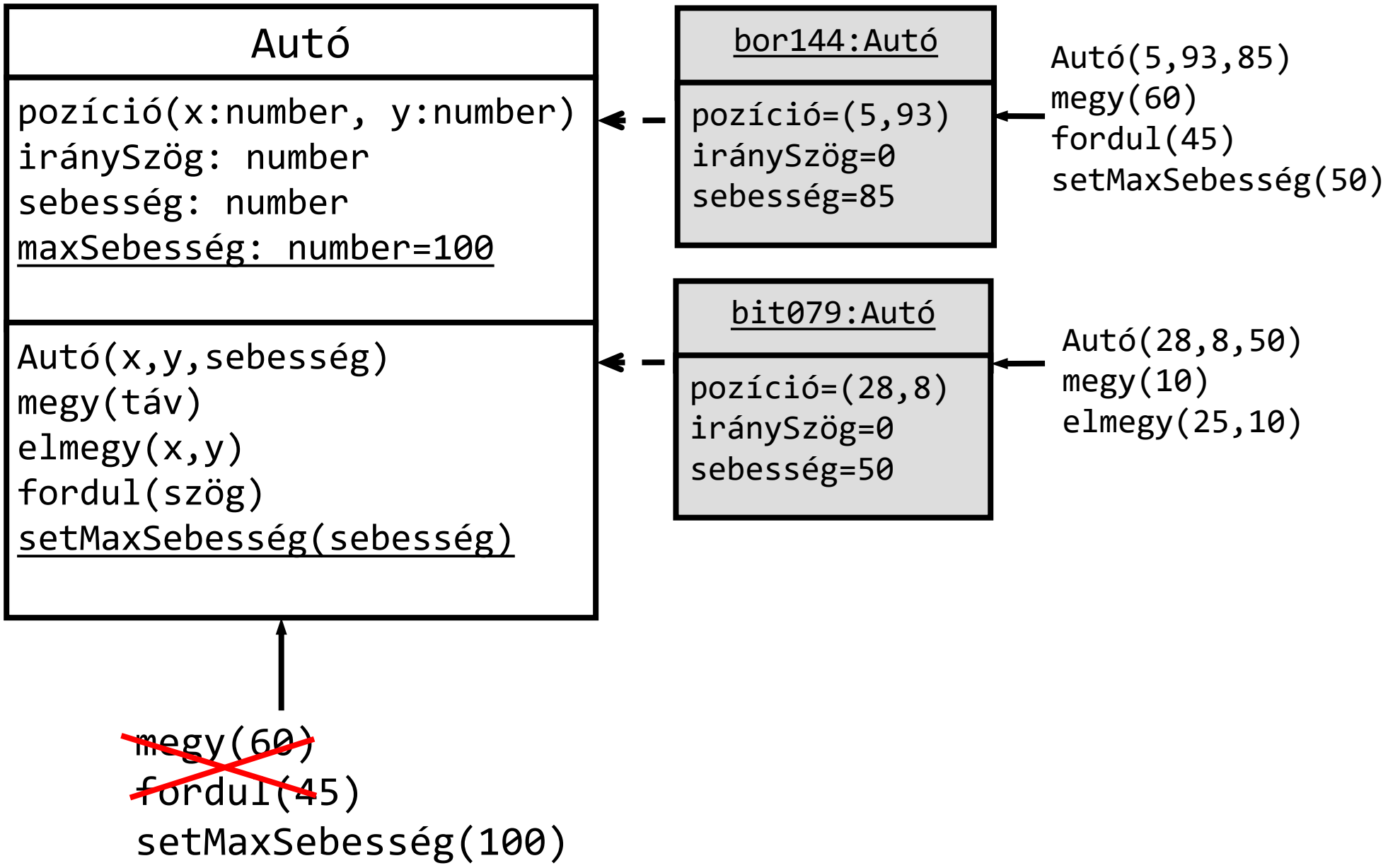
Az előző konstruktor helyett:

```
  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)

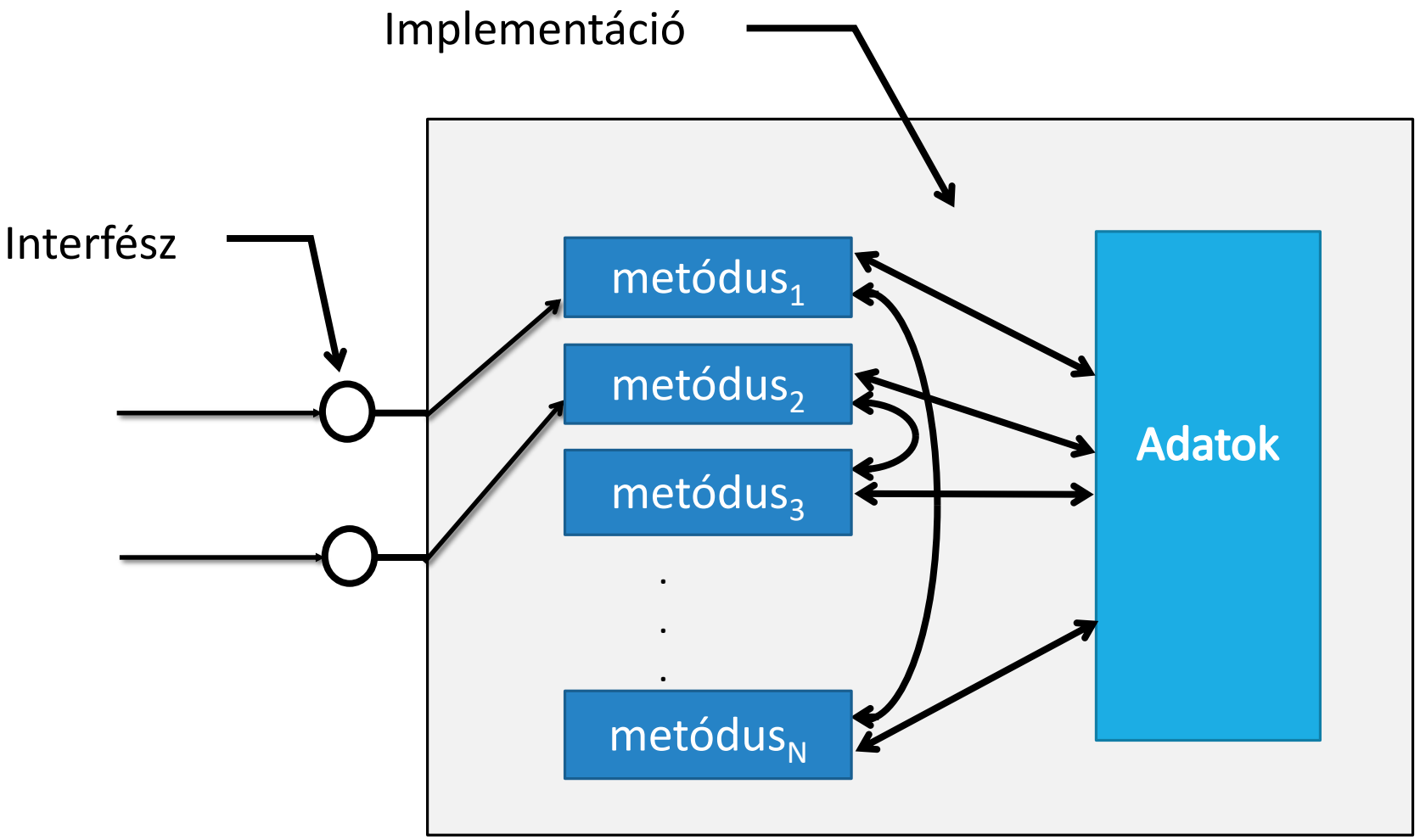
Információ elrejtése (information hiding)

- Az objektum „belügyeit” csak az interfészen keresztül lehet megközelíteni (láthatóságok!)

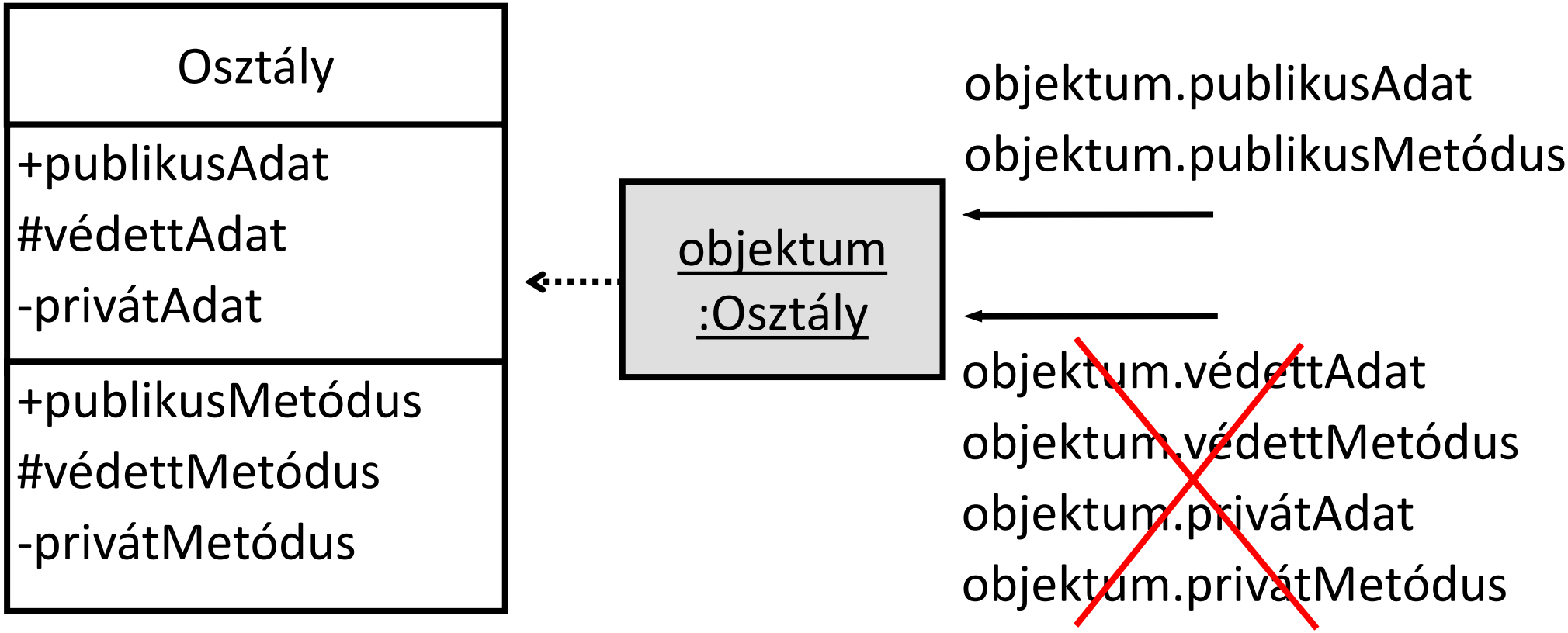
Kód újrafelhasználása (code reuse)

- Megírt kód felhasználása példány létrehozásával vagy osztály továbbfejlesztésével

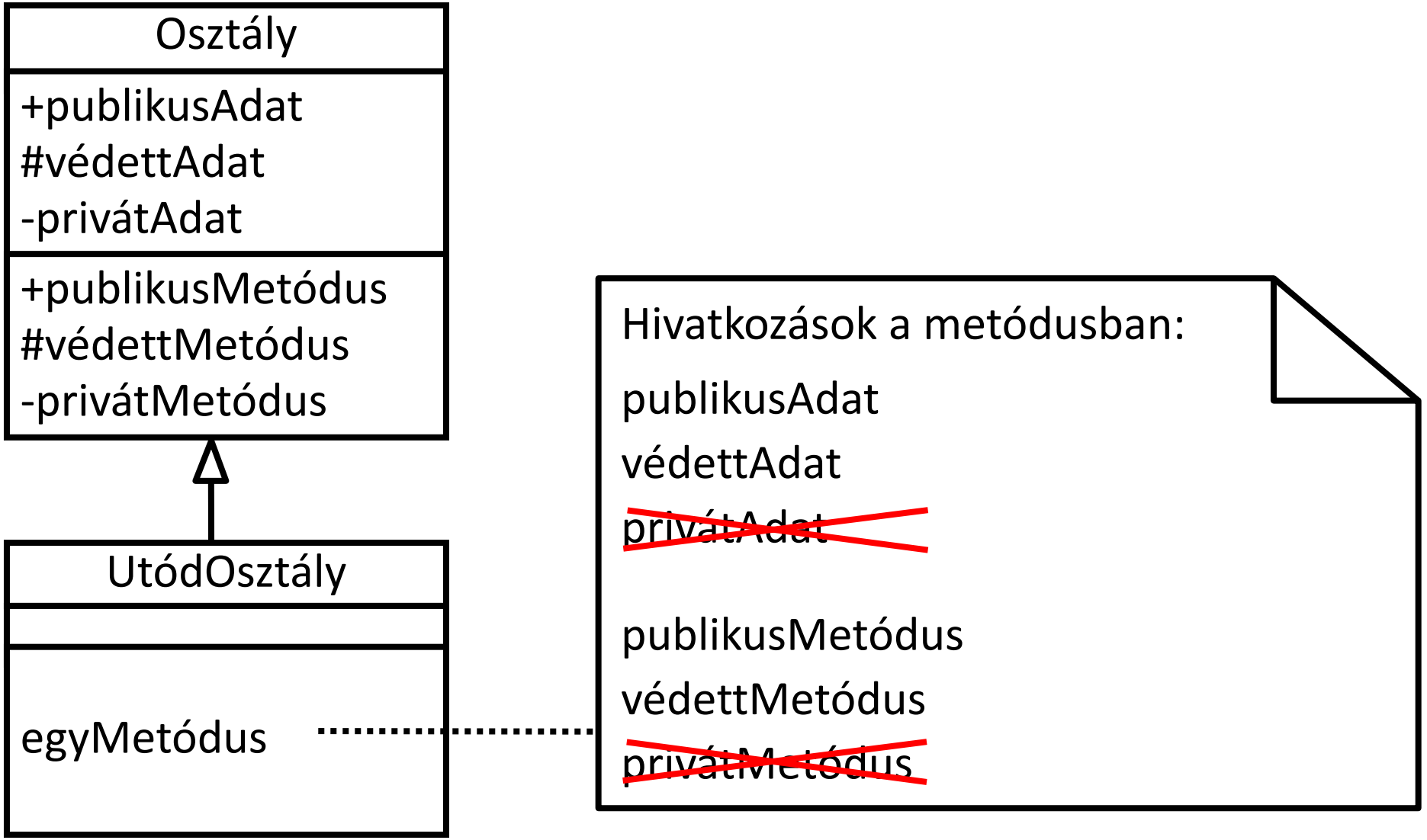
Információ elrejtése



Láthatóság – Objektum védelme



Osztály védelme



C++

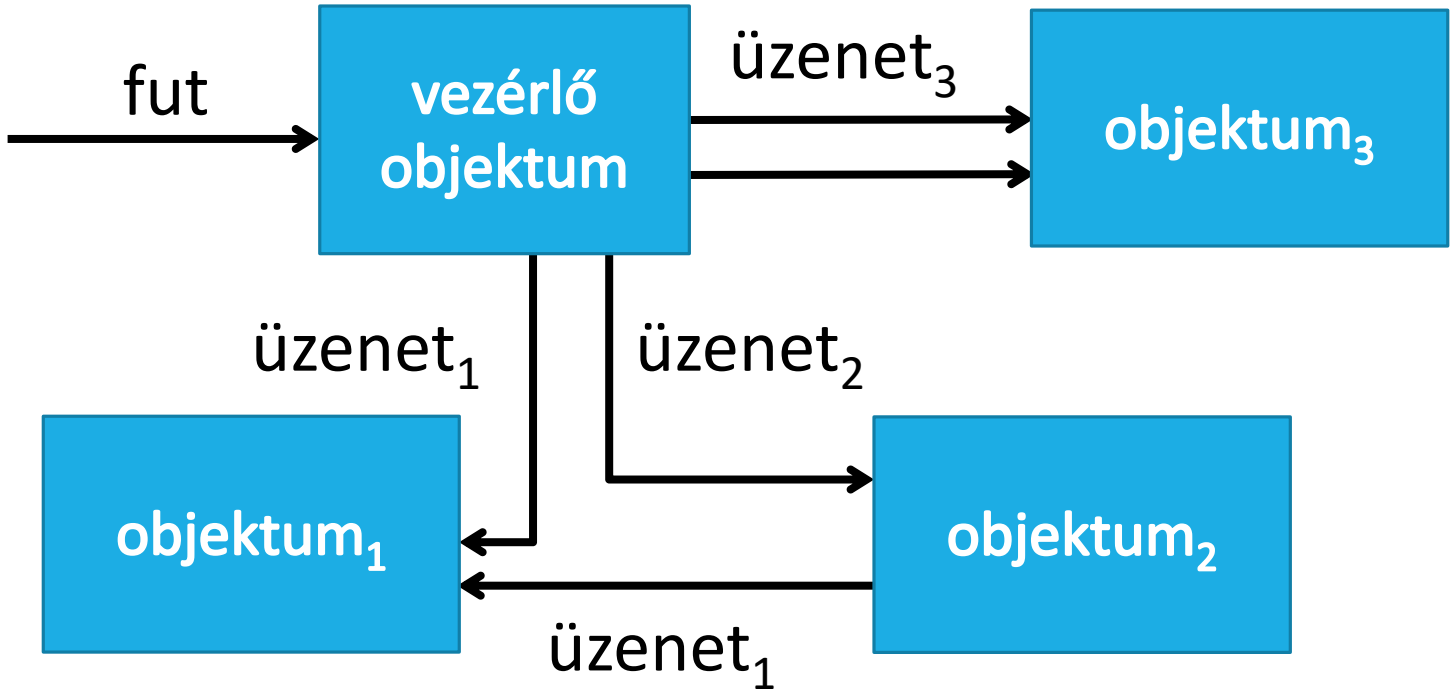
Az adattagok és metódusok elrejtése megoldott

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

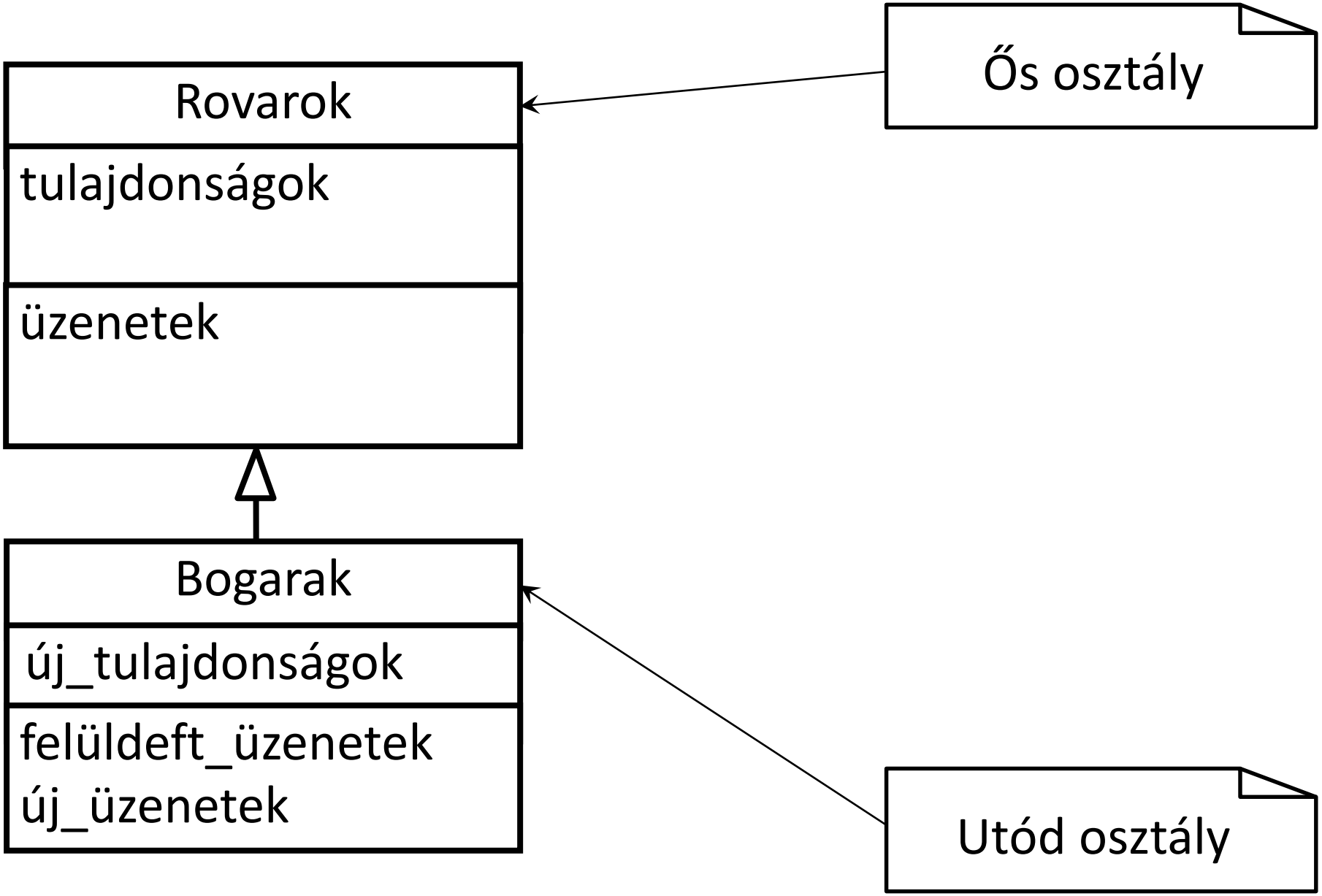
Az alapgondolat: a gyerekek öröklik őseik metódusait és változóit

Az *örököl* terminus azt jelenti, hogy az ősosztály **minden** metódusa és adattagja a gyerekosztálynak is metódusa és adattagja lesz

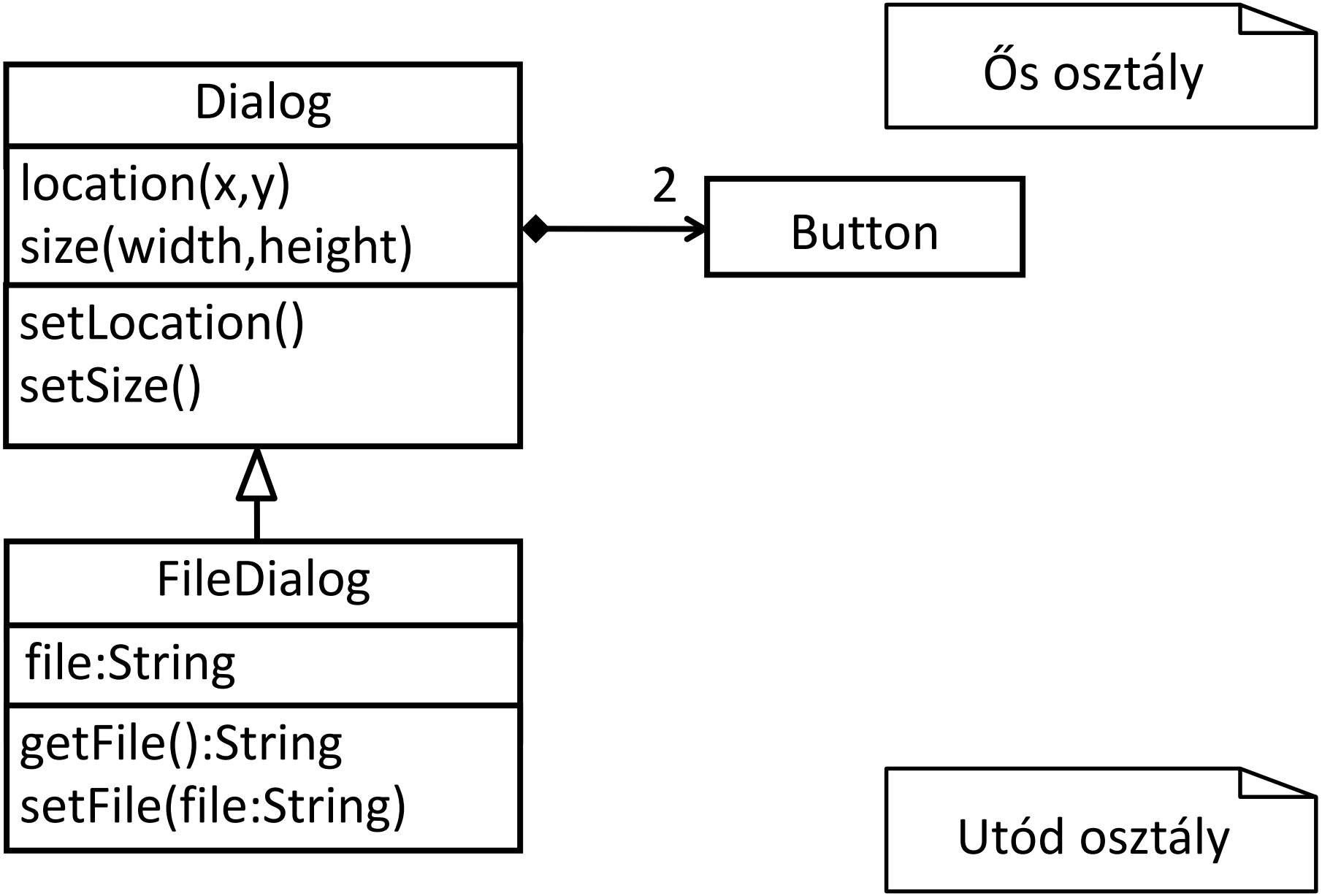
A gyerek minden új művelete vagy adattagja egyszerűen hozzáadódik az örökölt metódusokhoz és adattagokhoz

Minden metódus, amit átdefiniálunk a gyerekben, a hierarchiában felülbírálja az örökölt metódust

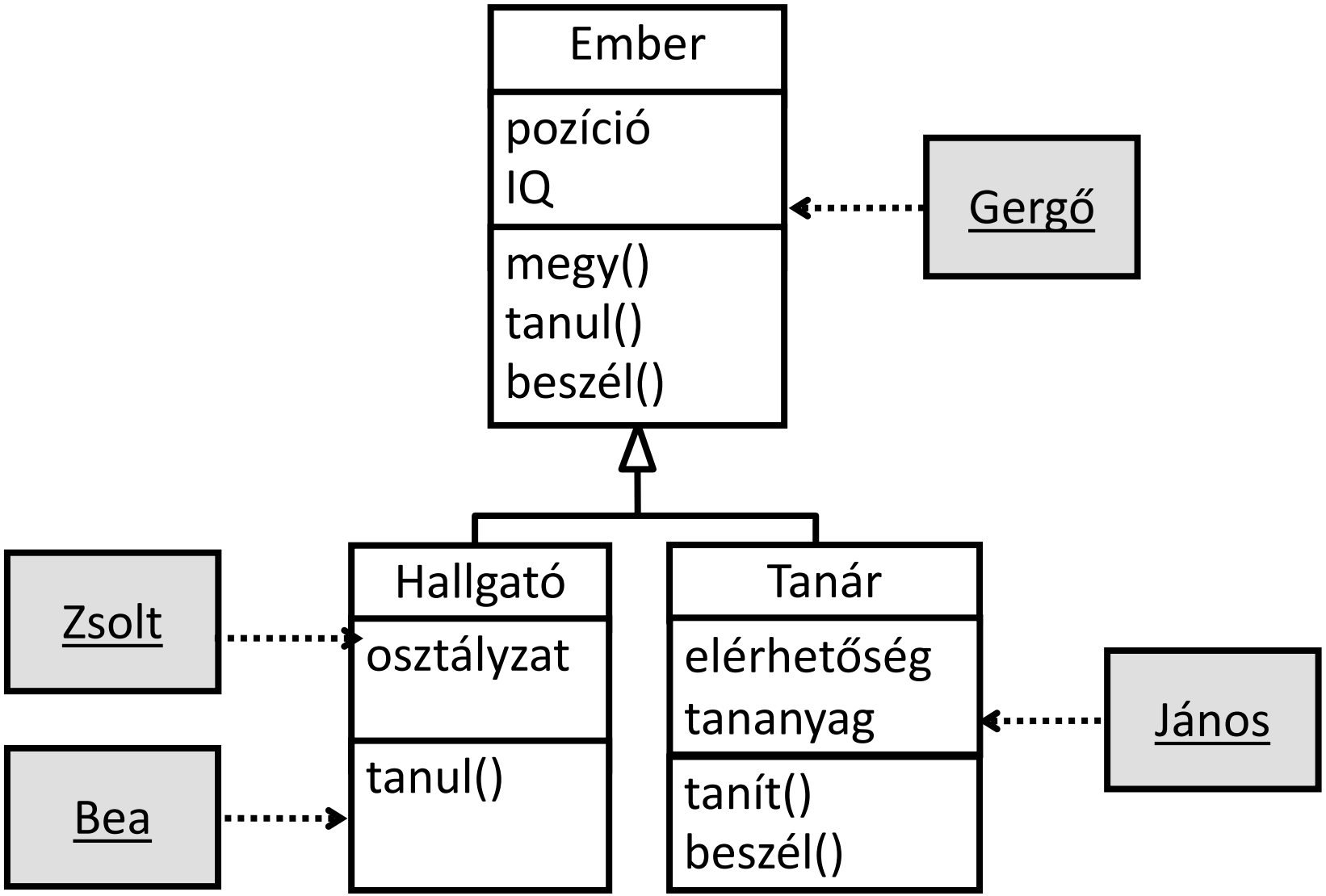
Öröklődés – IS-A reláció



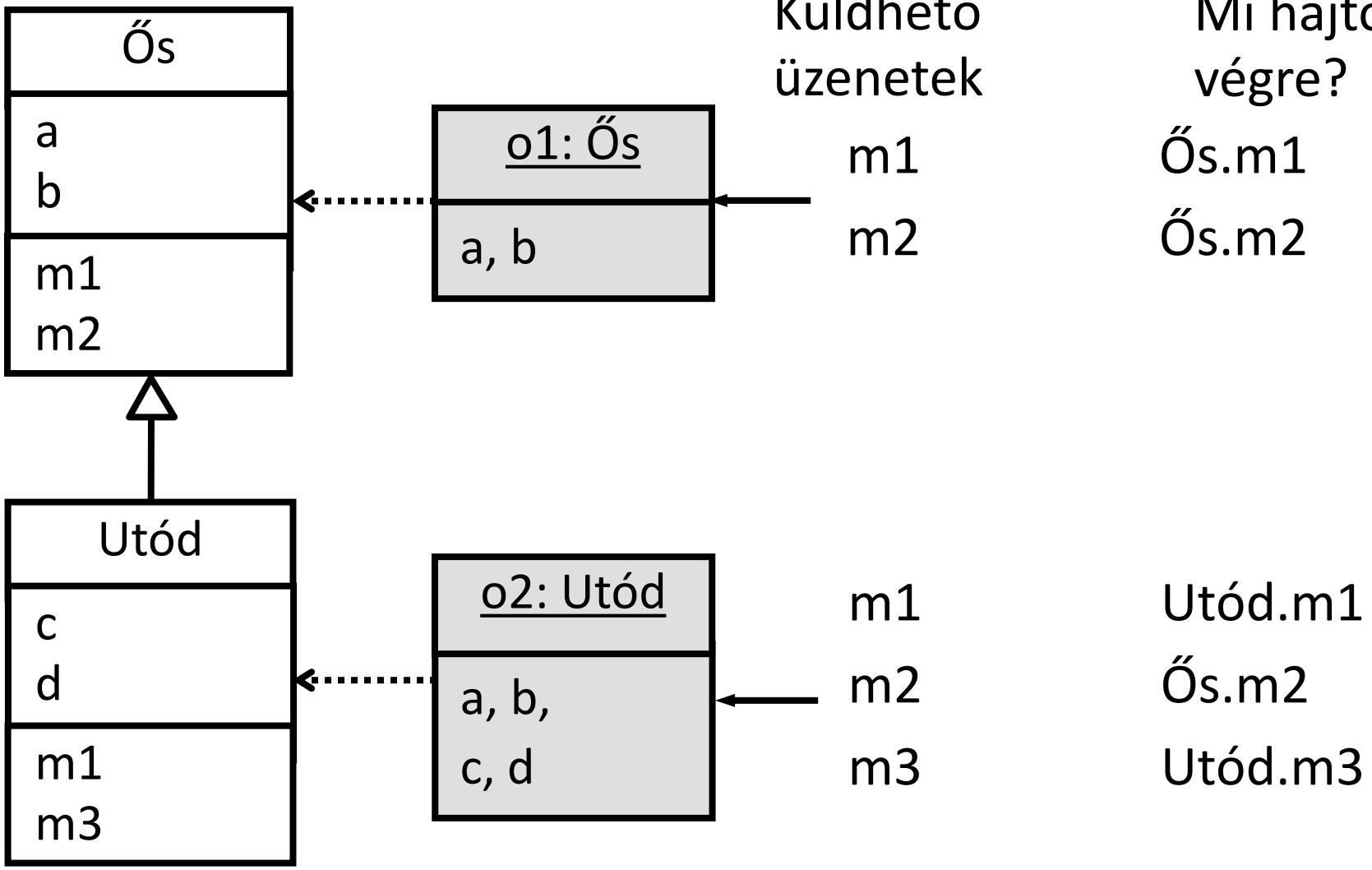
Öröklődés


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

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

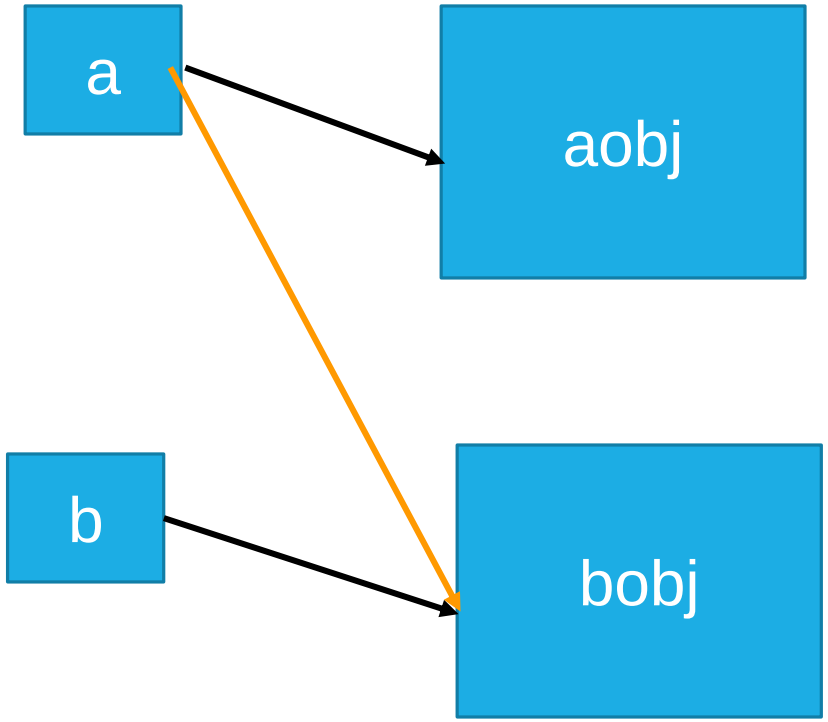
```
Employee emp ("Kati","Fekete",3);
Manager m("Jozsef", "Kovacs", 3,2);
emp = m;
```

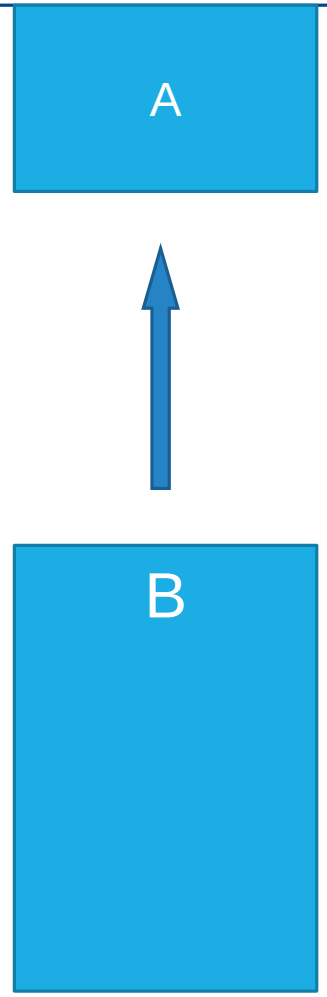
Megengedett

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

Mi történik?

Hogy lenne jó?





a:=b

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`?

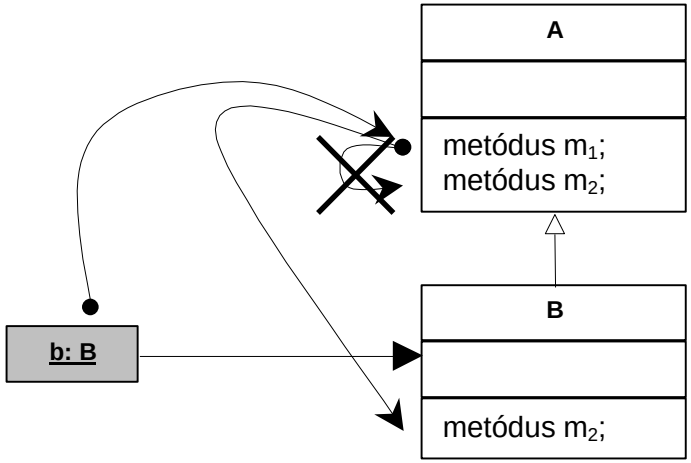
Dinamikus összekapcsolás

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';//…
   }
 …
}
```