Programozási nyelvek és módszerek

OOP – I. (ISMÉTLÉS)

Kivételkezelés

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
- 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

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

Python

Előzőekhez hasonlóan

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

- Az **else** jelentősége, hogy lokális változókhoz hozzáfér, de a **try** által védett kódot már nem tartalmaz

Python

Előredefiniált clean-up kódok lefuttatása

- A következő kód helyett

```
file = open('file_path', 'w')
try:
    file.write('hello world')
finally:
    file.close()
```

- Írható ez is (a try-with-resources mintájára)

```
with open('file_path', 'w') as file:
    file.write('hello world !')
```

Python

A **with** használatára alkalmas osztály készíteni az __enter__ és __exit__ függvények definiálásával lehet

```
class MessageWriter(object):
    def __init__(self, file_name):
        self.file_name = file_name

    def __enter__(self):
        self.file = open(self.file_name, 'w')
        return self.file

    def __exit__(self):
        self.file.close()
```

SWIFT

A kivételeket az `Error` protokollt 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 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( )
```

- 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() {
           // line
        }
        // close(file) hívása itt történik a scope végén
    }
}
```

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

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
```

Kivételkezelés – érdekesség

A C++ nyelvet számos kritika éri, többek között a kivételkezelés kapcsán.

Eredményképpen számos Code-Style tiltja

- Google
- LLVM
- Qt

Objektum Orientált Programozás

Az OO paradigma

Mitől OO egy program?

Objektum

Osztály

Öröklődés

A valós világ modellezése

Az ember a világ megértéséhez modelleket épít

Modellezési alapelvek

- Absztrakció
  - Szemléletmód, amelynek segítségével a valós világot leegyszerűsítjük
    - Csak a lényegre, a cél elérése érdekében feltétlenül szükséges részekre összpontosítunk
  - Elvonatkoztatunk a számunkra pillanatnyilag nem fontos, közömbös információktól
    - Kiemeljük az elengedhetetlen fontosságú részleteket.
- 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

Modellezési alapelvek

- Osztályozás
  - Az objektumokat kategóriákba, osztályokba
    - a hasonló tulajdonságokkal rendelkező objektumok egy osztályba kerülnek.
    - a különböző vagy eltérő tulajdonságokkal rendelkező objektumok külön osztályokba kerülnek.
  - Az objektum-osztályok hordozzák a hozzájuk tartozó objektumok jellemzőit, objektumok mintáinak tekinthetők.
- Általánosítás, specializálás
  - Az objektumok között állandóan hasonlóságokat vagy különbségeket keresünk
    - 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
- Változókkal valósítjuk meg
  - Ezek az adattagok, vagy tulajdonságok.

Kérésre feladatokat hajt végre
- Metódusok – melyek hatására állapota megváltozhat

Üzeneteken keresztül lehet megszólítani
- Kommunikál más objektumokkal

Minden objektum egyértelműen azonosítható

Osztály, példány

Osztály (class)

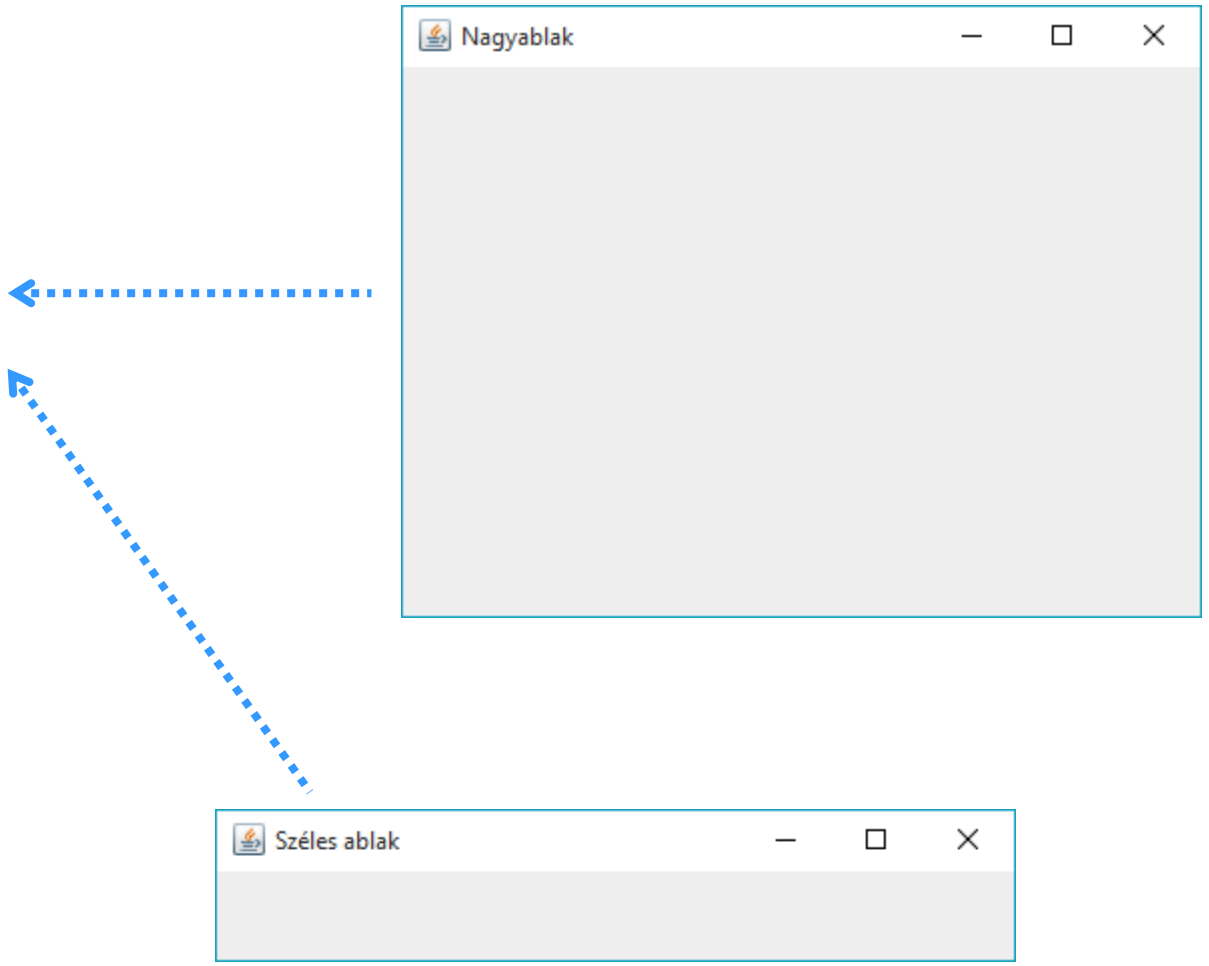
- Olyan objektumminta vagy típus
- Ez alapján példányokat (objektumokat) hozhatunk létre

Példány (instance)

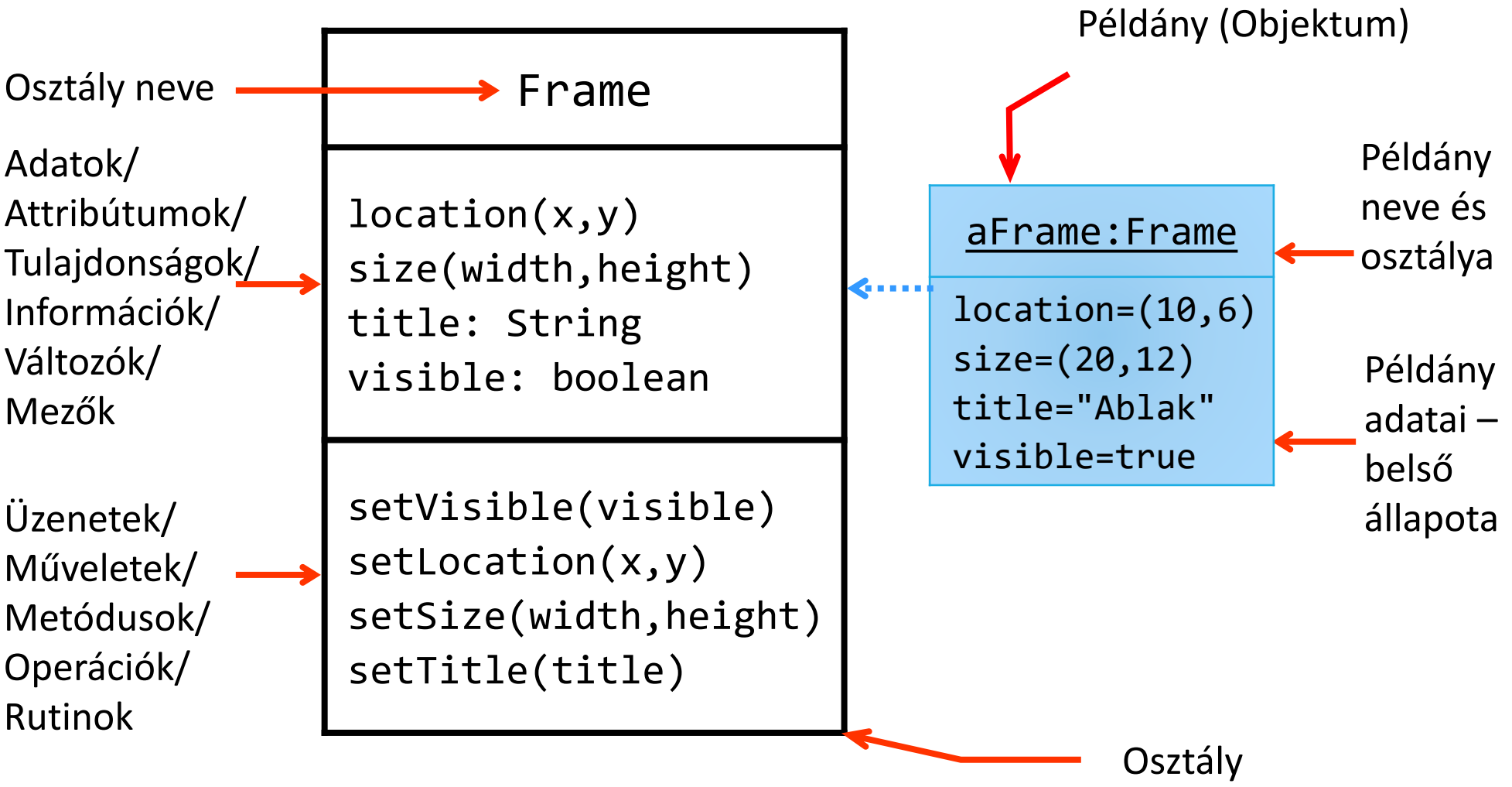
- Egy osztály (minta) alapján létrejött konkrét példány
- Minden objektum születésétől kezdve egy osztályhoz tartozik

Frame osztály és példányai

Frame
location(x,y) size(width,height) title: String visible: boolean
setVisible(visible) setLocation(x,y) setSize(width,height) setTitle(title)



Osztály és példány az UML-ben



Objektum életciklusa

Objektum életciklusa

- Létrejön – „megszületik”
- Állapotváltozásokon esik keresztül – „él”
- Megszűnik – „meghal”

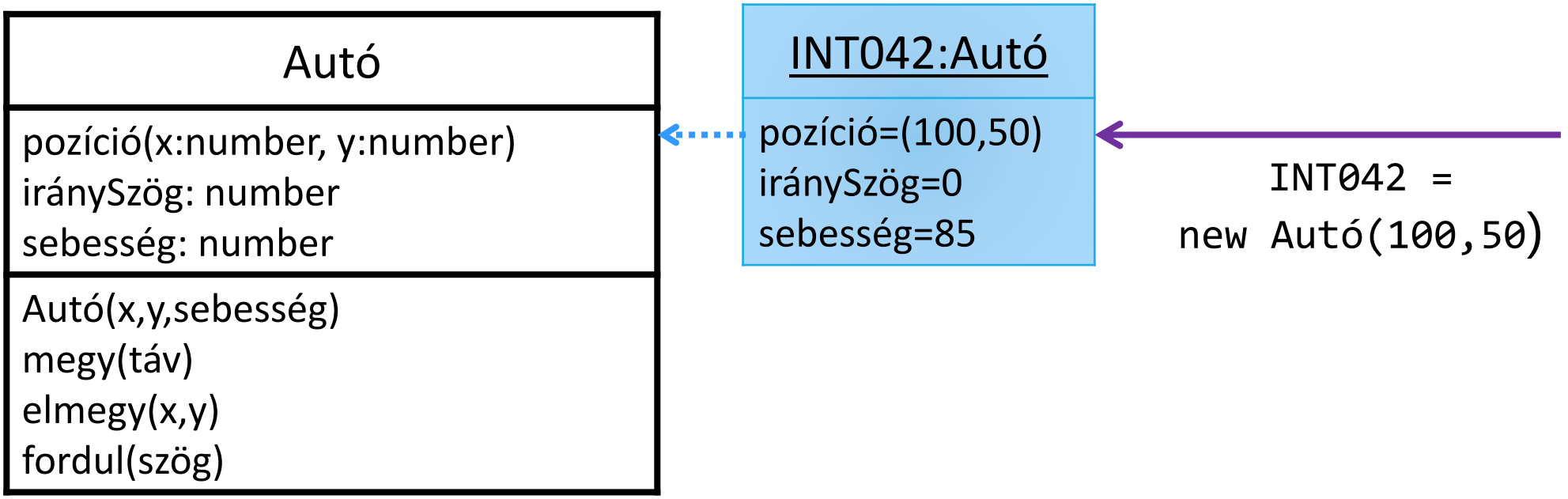
Objektum létrehozása, inicializálása

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 állapotváltozása

Az objektum kérésre feladatokat hajt végre

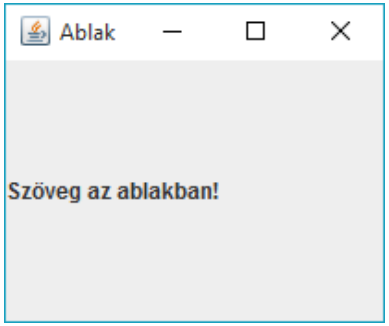
Életciklusa során üzeneteket fogad és feldolgozza

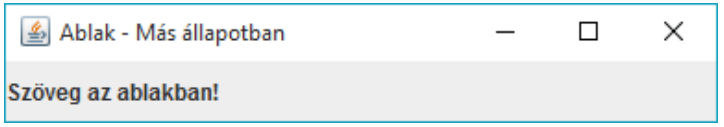
- Hatására az állapot megváltozik
- Ezek során üzeneteket küldhet más objektumoknak

Állapotváltozás

Üzenetek

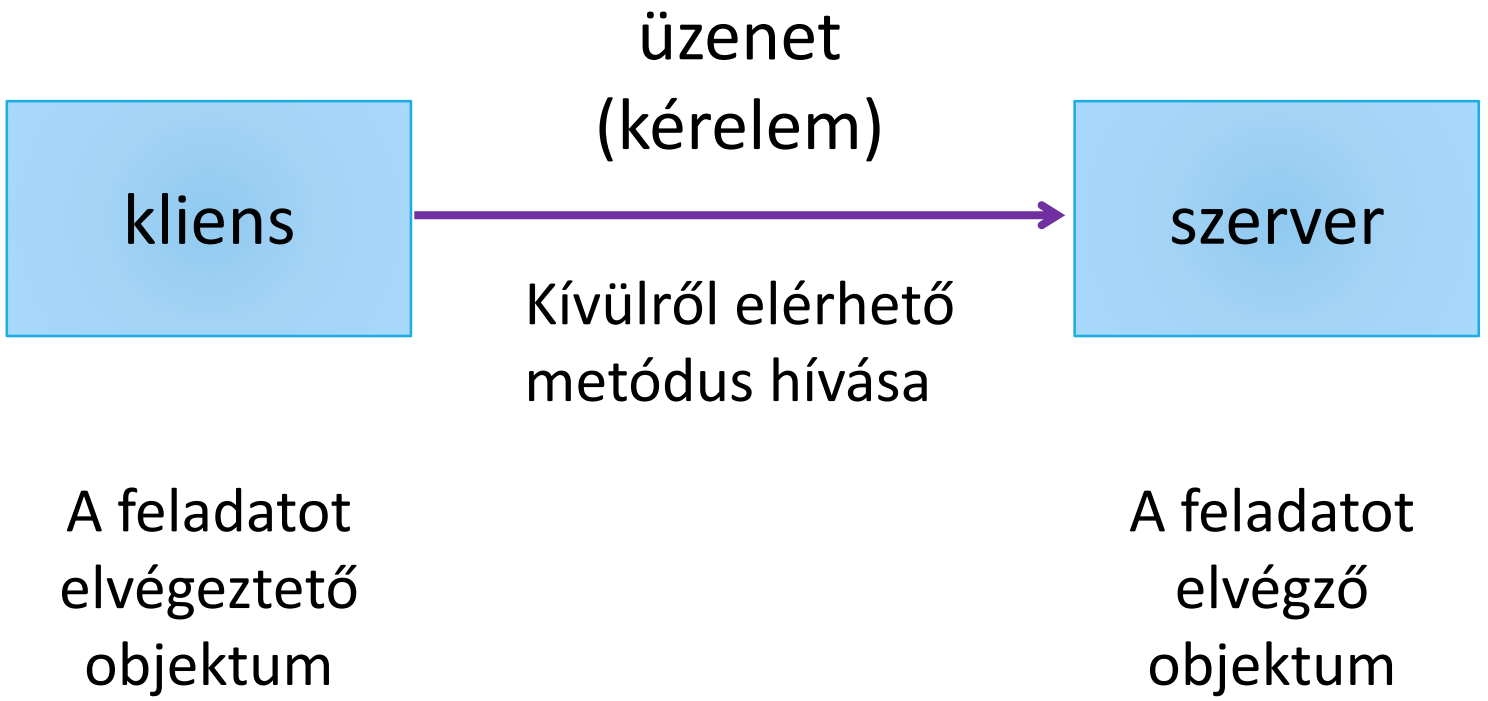
```
setVisible(true)
setLocation(40,8)
   setSize(20,16)
setTitle("Ablak")
```





aFrame:Frame	
(20,16)	location(x,y)
(100,80)	size(width,height)
"Ablak"	title
true	visible

Üzenetküldési modell



Üzenetküldési modell

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

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

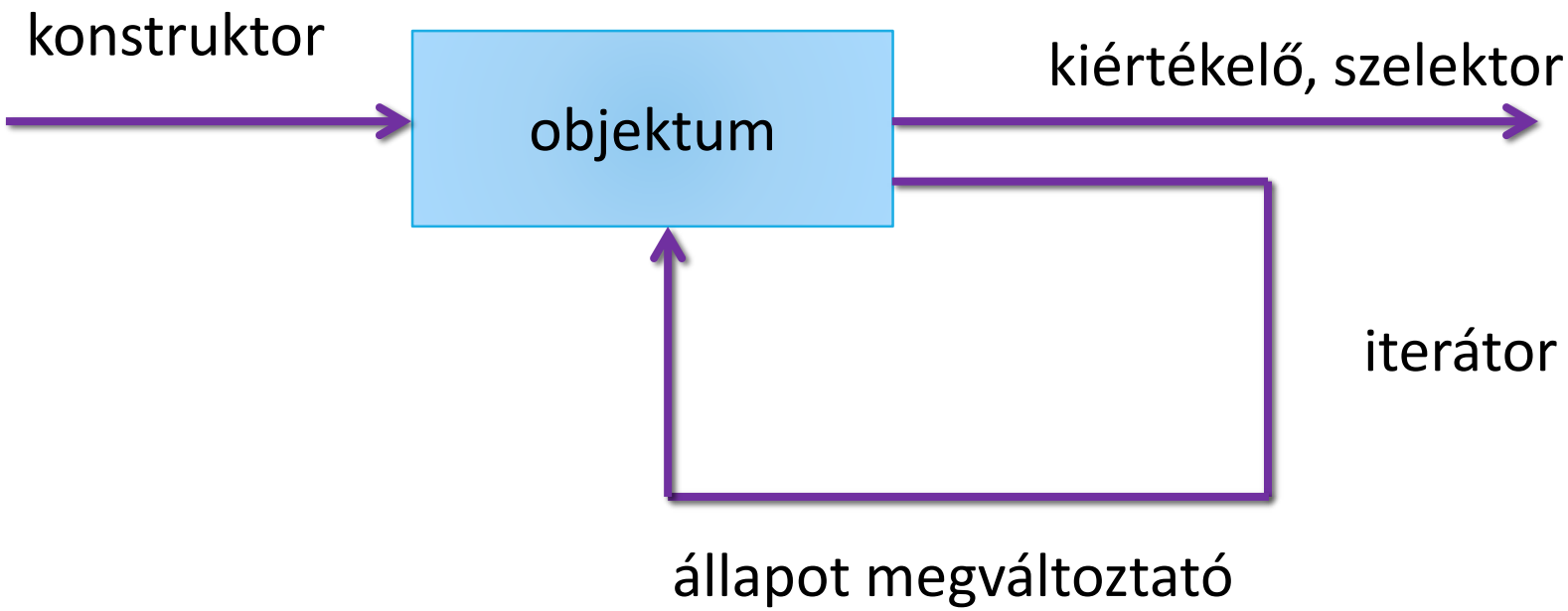
Objektum műveletei

Export műveletek csoportosítása:

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

Objektum műveletei

Export műveletek csoportosítása:



Objektum megszűnése

Az objektum életciklusának végén:

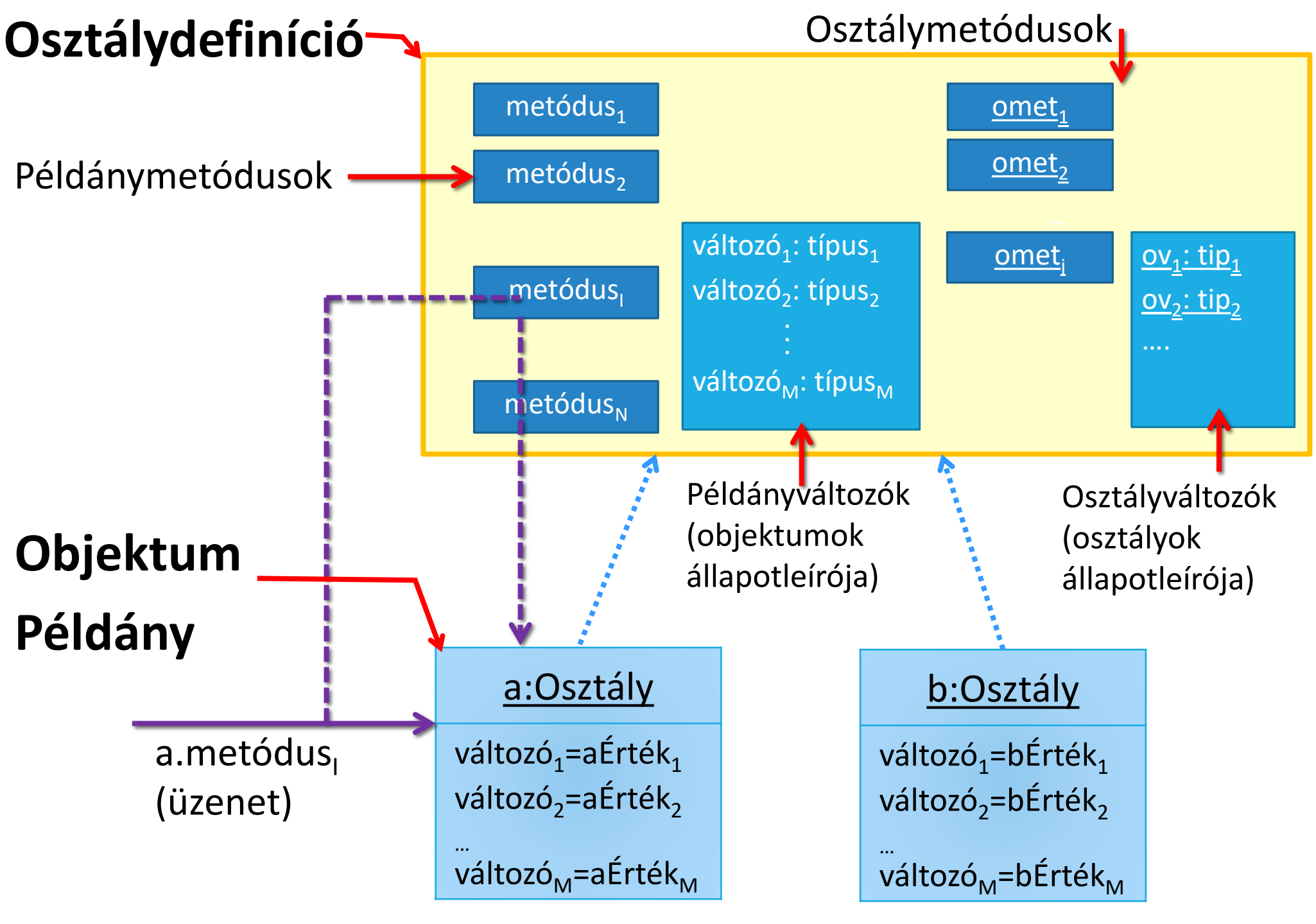
- Erőforrások (memória) felszabadítása
- Adatok, állapot mentése

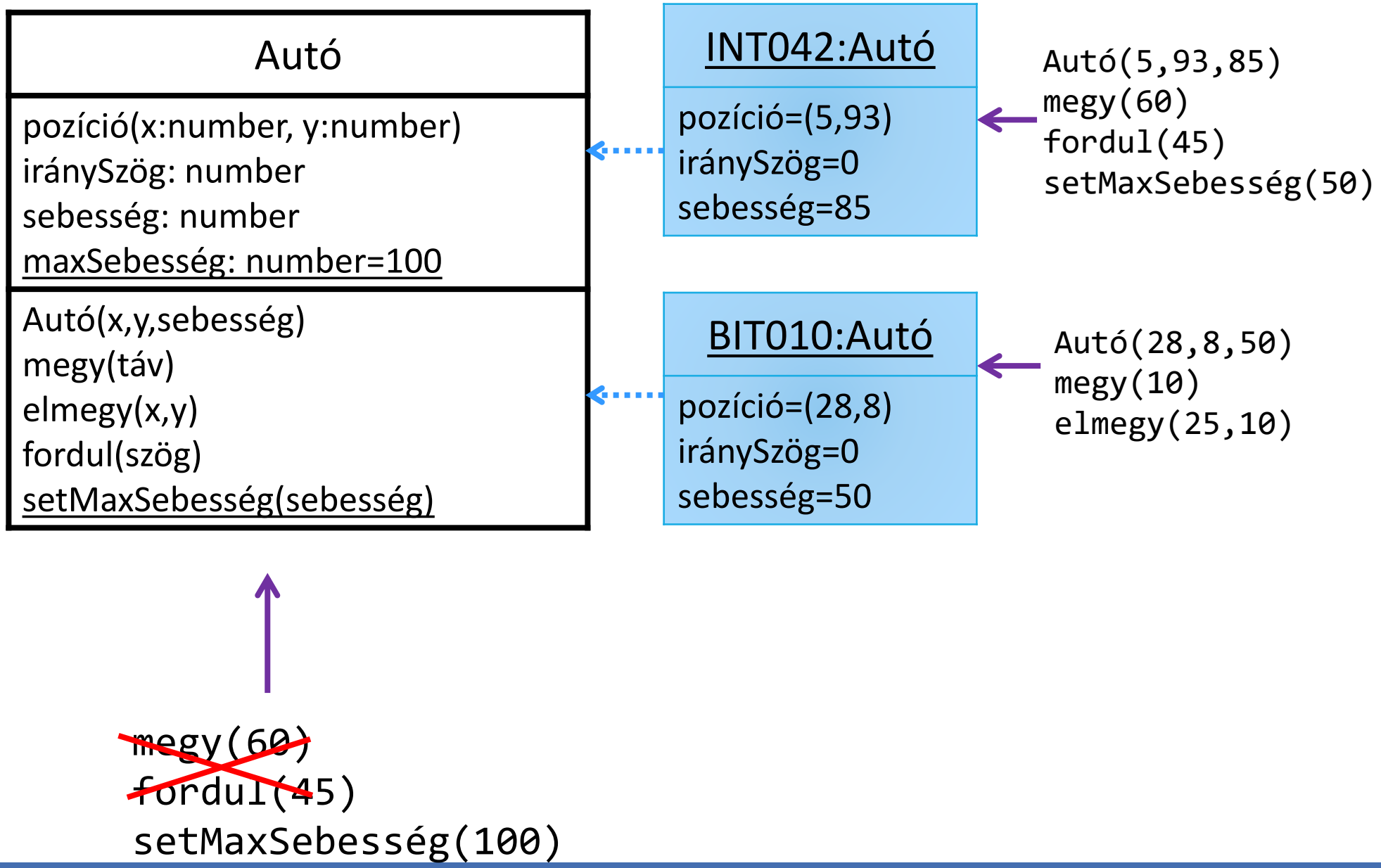
Ha a nyelv támogatja, akkor a destruktor (destructor) hajtja végre.

Osztály, példány

Osztálydefiníció:

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


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


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)
INT042:Autó
pozíció=(5,93)
iránySzög=0
sebesség=85
Autó(5,93,85)
megy(60)
fordul(45)
setMaxSebesség(50)
BIT010: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”

Adott osztályhoz tartozó példányok esetén

- Honnan tudjuk, hogy éppen melyik objektum hívta meg a megfelelő metódust
- Melyik objektum adataival fog dolgozni a metódus?

Szükségünk van egy olyan mutatóra, amely mindig a metódust meghívó objektumpéldányra mutat.

- Ezt szolgálja a „`this`” pszeudóváltozó
  - Néha explicit formális paraméter
- Ez metódushíváskor egyértelműen rámutat azokra az adatokra, amelyekkel a metódusnak dolgoznia kell.

Az objektum a saját magának küldött üzenet esetén a `this.Üzenet(Paraméterek)` formát kell, hogy használja.

- 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)

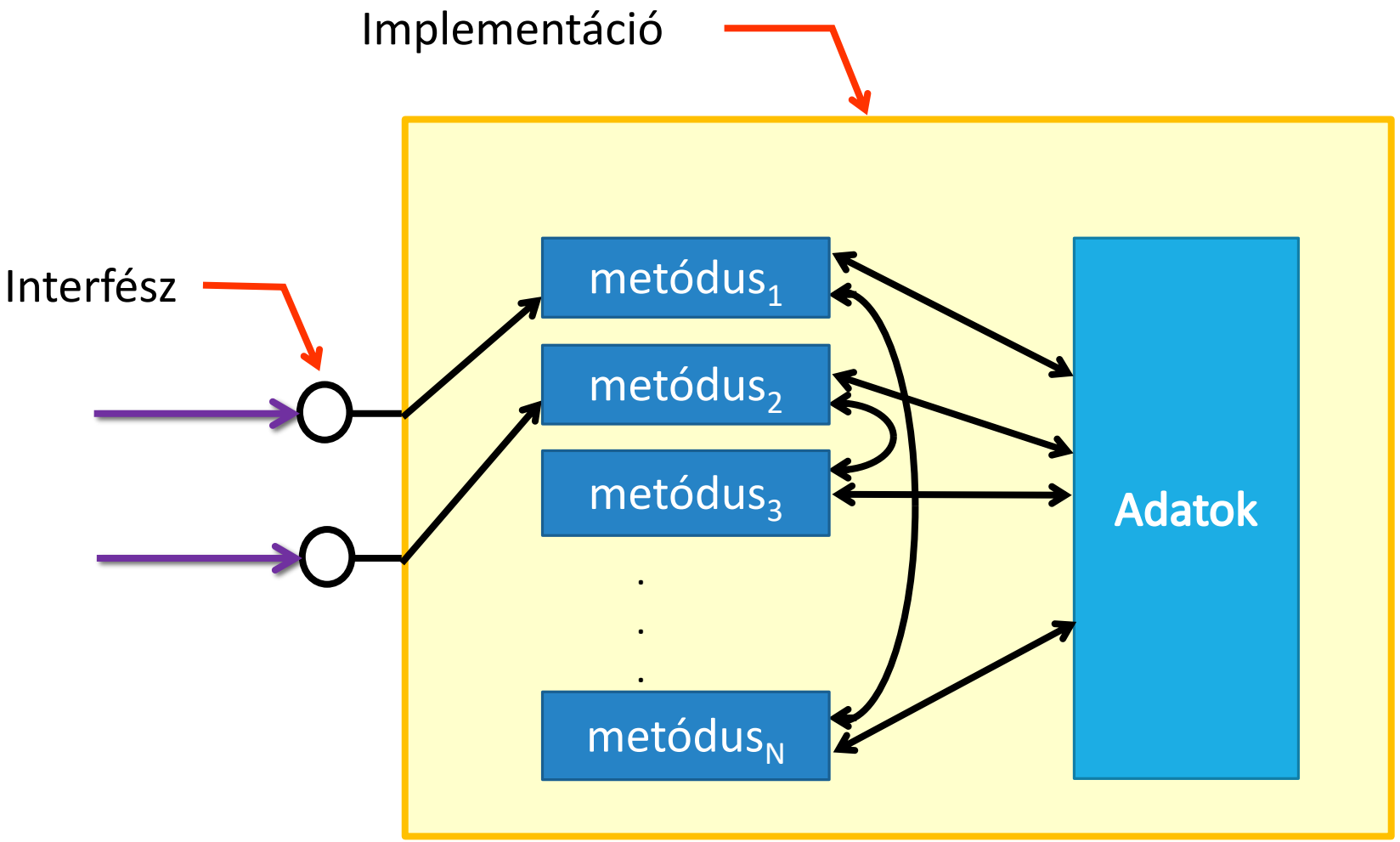
Információ elrejtése (information hiding)

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

Kód újrafelhasználása (code reuse)

- Megírt kód felhasználása példány létrehozásával vagy osztály továbbfejlesztésével

Információ elrejtése



Információ elrejtése

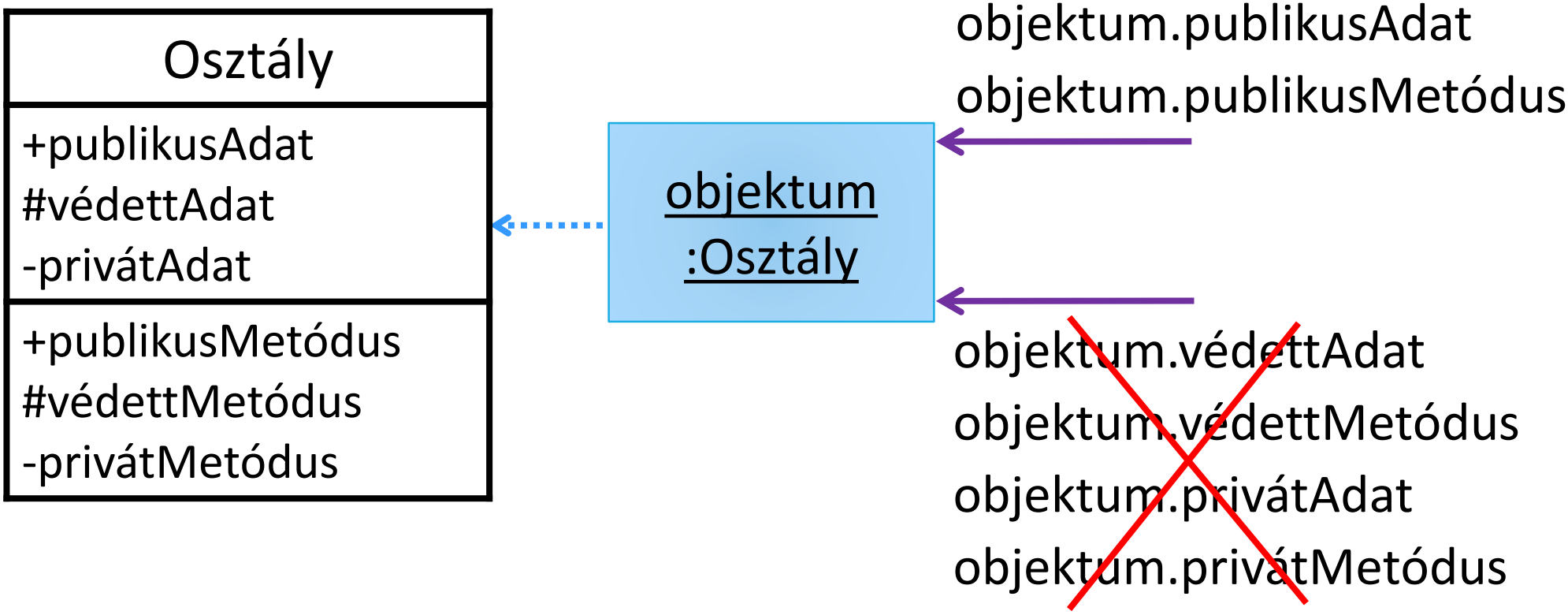
Az objektum adatait csak interfészen keresztül lehet elérni

- Az elv szigorú követése esetén, minden adattag elrejtésre kerül, közvetlenül nem hozzáférhető
  - Dedikált függvények használatával lehetséges a hozzáférés.

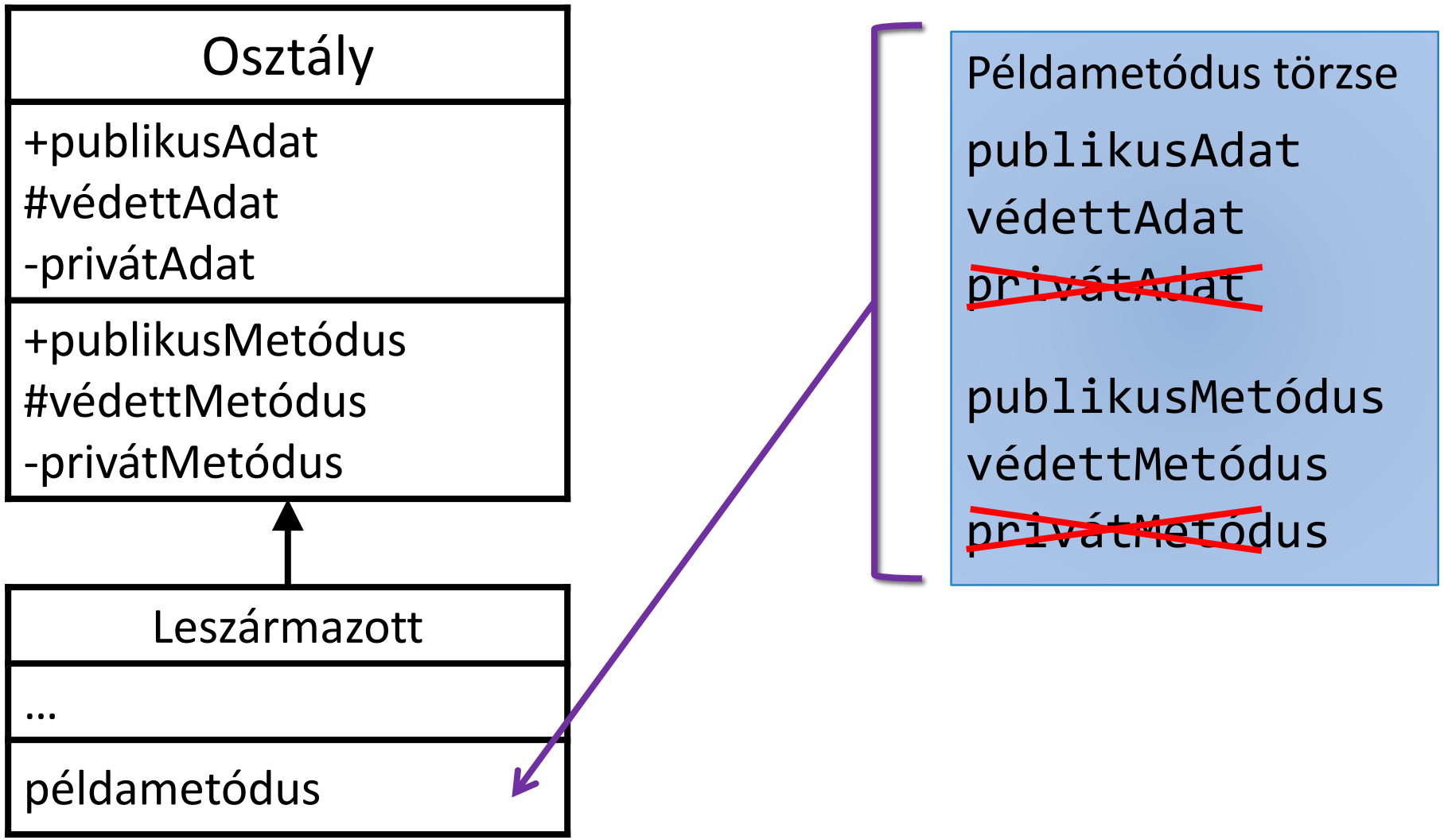
A láthatóság minősítője leggyakrabban

- public
  - a külső felhasználók elérik
- protected
  - csak a leszármazottak érhetik el
- private
  - csak az adott osztály számára elérhető

Láthatóság – Objektum védelme



Láthatóság – Objektum védelme



Öröklés

Az alapgondolat: a gyerekek öröklik őseik metódusait és változóit

Az örököl 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

Lehet egyszeres, vagy többszörös

- Ez a közvetlen ősosztályok számát jelenti

Öröklés

Mit örököl a leszármazott?

- Tulajdonságokat (Adattagokat)
- Metódusokat (Üzeneteket)

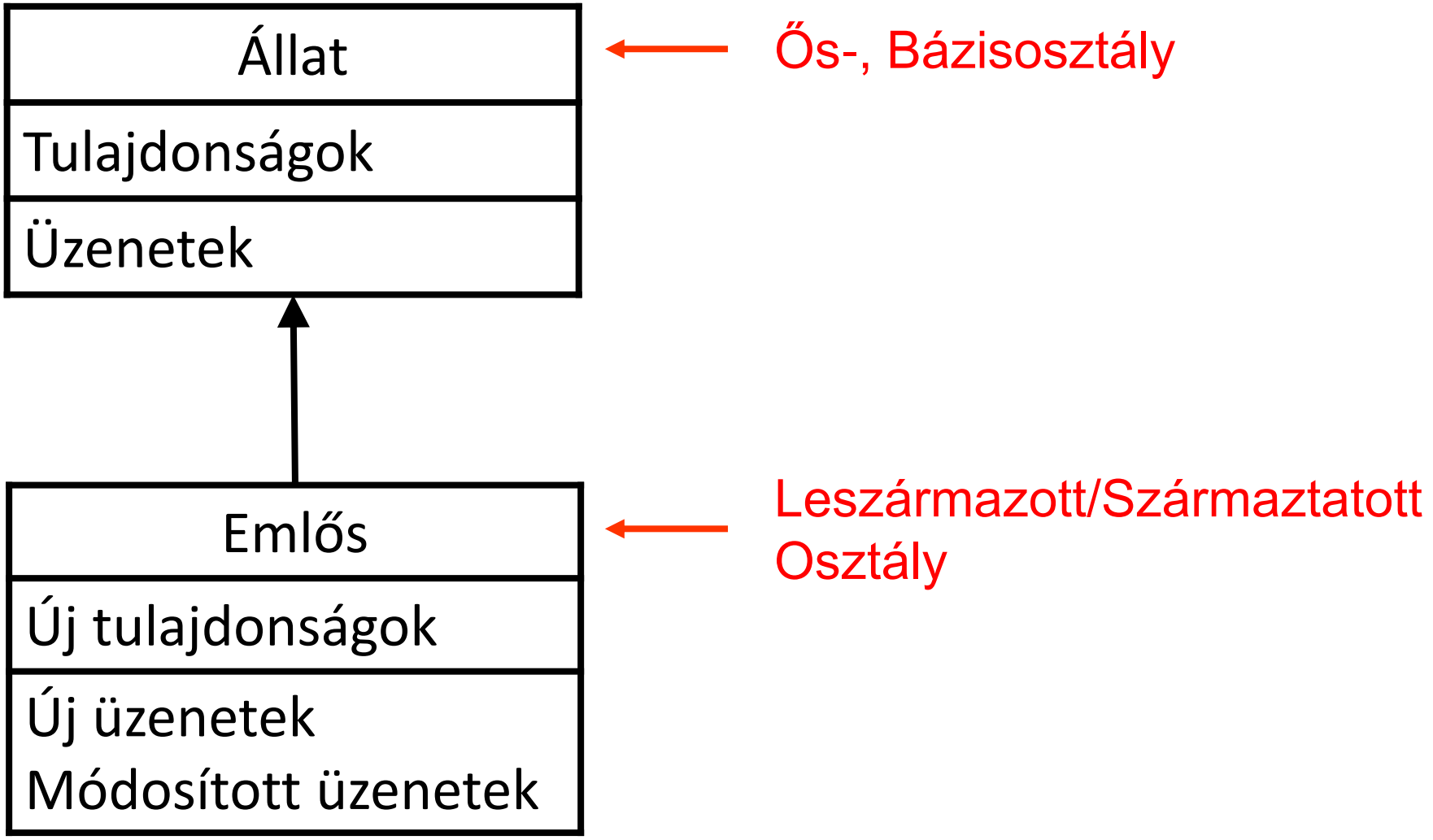
Mit nem örököl a leszármazott?

- Ősosztály konstruktorait, destruktorát
  - Ám tudja használni / meghívni / delegálni
- Ősosztály értékadás operátorát

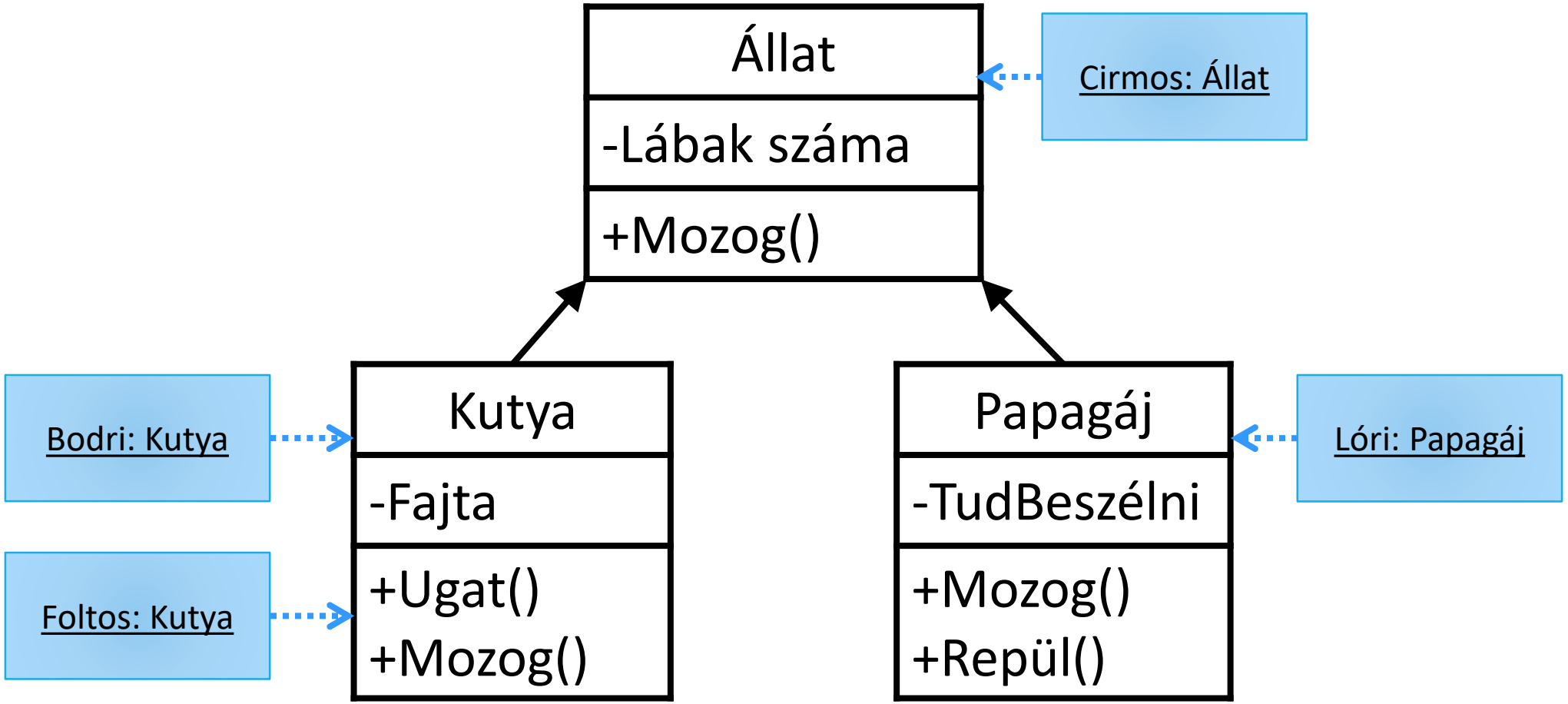
Mit tehet a leszármazott osztály?

- Új tulajdonságokat vezethet be
- Új metódusokat vezethet be
- Felüldefiniálhat, vagy elfedhet már meglévőket
- Új konstruktorokat és destruktort

Öröklés – IS-A reláció



Öröklés – IS-A reláció



Implementáció újrahasznosítás

Alosztályképzés

- Használd egy típus implementációját egy másik típus implementálására!
- Gyakran használjuk az őstípus implementációját az altípus implementálására
- A gyakran használt OO programozási nyelvek keverik az altípus és alosztály fogalmakat.

Felüldefiniálás/Elfedés

Két különböző jelenség

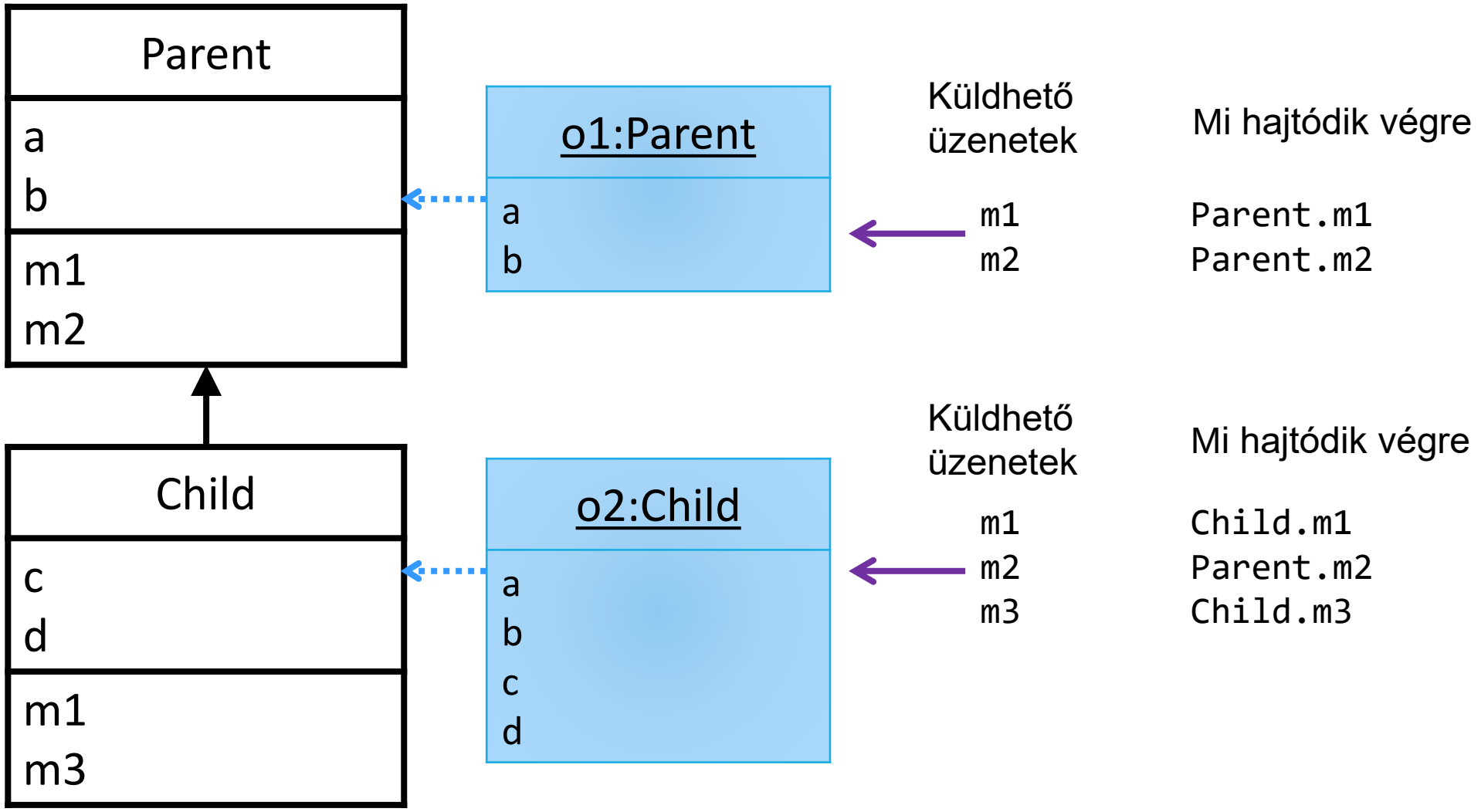
Ami közös a felüldefiniálás (átdefiniálás) illetve elfedés során:

- Az ősosztályban már meglevő, bevezetett függvénynek/üzenetnek ad új definíciót
  - A szignatúra azonos

Ami különböző

- A metóduselérés hogyan történik polimorfizmus esetén
- A két működést kulcsszavakkal lehet szabályozni általában

Küldhető üzenetek/Elérhető adatok



Polimorfizmus

Polimorfizmus (többalakúság)

- Az a jelenség, hogy egy azonosító (név: változó, függvény) nem csak egyfajta „objektumra” hivatkozhat.

A polimorfizmusnak több formája is van

- Altípusos
  - A (változó)név egy szupertípusra és leszármazottjaira hivatkozhat
- Nyers – Duck typing
- Parametrikus
  - Lényegében a sablonok
- Ad hoc
  - Lényegében a függvények túlterhelése

Altípusos polimorfizmus

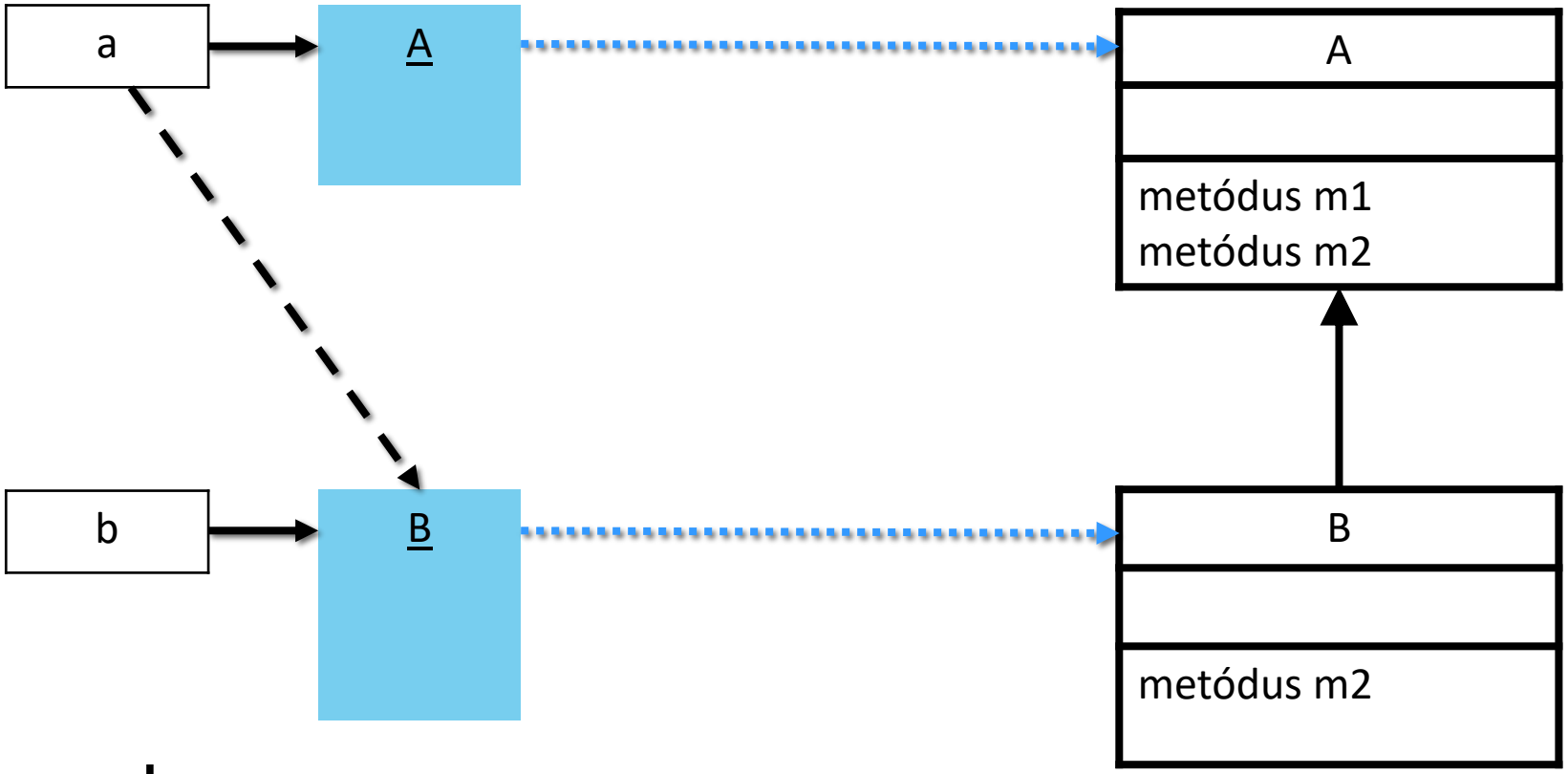
Változó nem csak egyfajta típusú objektumra hivatkozhat, hanem annak altípusaira is

- Statikus típus: a deklaráció során kapja.
- Dinamikus típus: futásidőben, éppen milyen típusú objektumra hivatkozik
  - a statikus típus, vagy annak leszármazottja

`A` típus altípusa `B`, ha a `B` minden olyan helyzetben használható, ahol az `A` is.

- A `Kutya` egyben `Állat` is (IS-A).
- A `Háromszög` az egy `Alakzat`.

Altípusos polimorfizmus



a:=b

Altípusos polimorfizmus

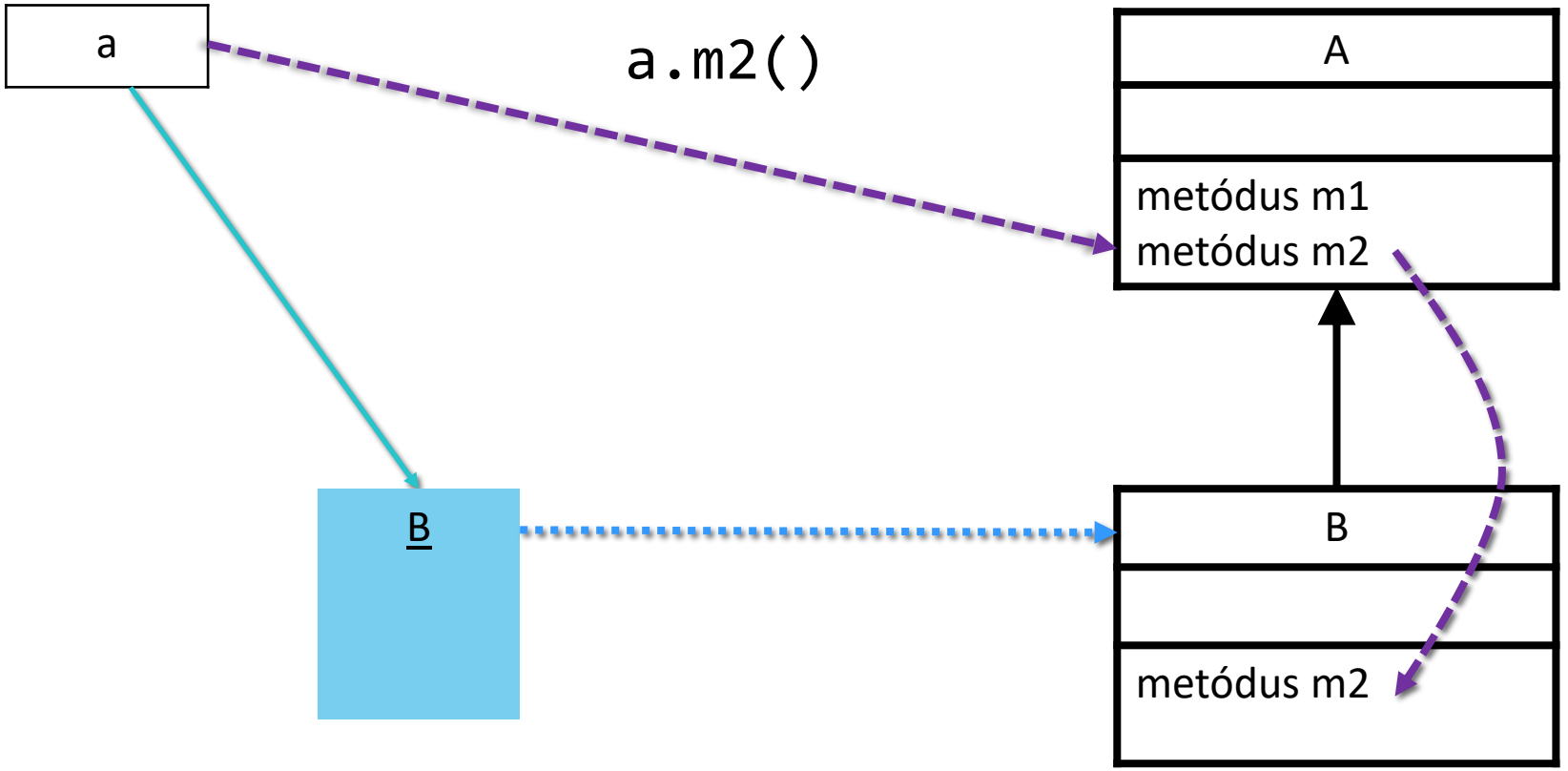
Az előző példában a `a` változó

- Statikus típusa: `A`
- Értékadást követően a dinamikus típusa `B`

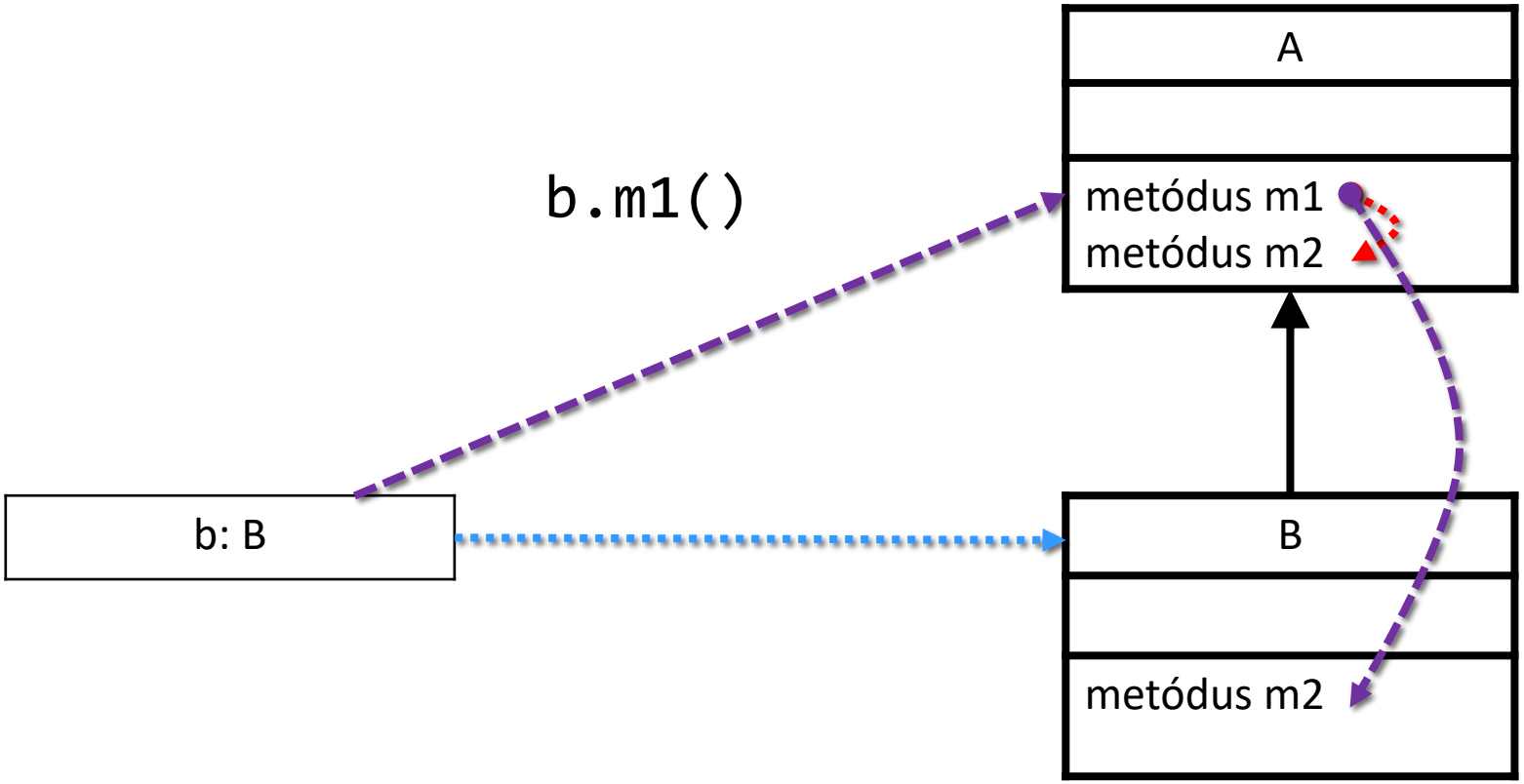
Mi történjen, ha a `B` osztály által felüldefiniált `m2()` függvényt hívjuk az `A` statikus típusú változót használva?

- Egy `B` objektum viselkedése az objektum típusának feleljen meg!
- Ezt a dinamikus összekapcsolás valósítja meg

Dinamikus összekapcsolás



Dinamikus összekapcsolás



Dinamikus összekapcsolás

Run-time fogalom

- Az a jelenség, hogy a változó éppen aktuális dinamikus típusának megfelelő metódus implementáció hajtódik végre
- Felüldefiniált függvények/metódusok esetén történik meg
  - Elfedés esetén nem!

Altípus

Szabályok:

- Reflexív
  - $T \subseteq T$
- Tranzitív
  - Ha: $T1 \subseteq T2$; $T2 \subseteq T3$
  - Akkor: $T1 \subseteq T3$

Altípus – alprogramok esetén

Tegyük fel, hogy Triangle $\subseteq$ Shape

- `fss = proc (Shape) returns (Shape)`
- `fts = proc (Triangle) returns (Shape)`
- `fst = proc (Shape) returns (Triangle)`
- `ftt = proc (Triangle) returns (Triangle)`

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

Hogyan döntsünk?

Egy függvényt A → B alakban írunk, ha A típusú paramétere van és B típusú eredményt ad.

Ha (A’ → B’) <: (A → B), - altípus

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

Tegyük fel, hogy van egy f: A→B típusú függvényünk.

- Ha egy f’ : A’ → B‘ függvényt az f helyén akarunk használni, akkor A-beli argumentumot kell fogadnia és B-beli eredményt adnia.

Hogyan döntsünk?

Mivel az f’ értelmezési tartománya A’, így akkor alkalmazható A-beli elemekre, ha A<:A’, vagyis

- ha a : A tekinthető mint A’-beli, és f’(a) típus-helyes.

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

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

Altípus – Megoldás

- `fss = proc (Shape)    returns (Shape)`
- `fts = proc (Triangle) returns (Shape)`
- `fst = proc (Shape)    returns (Triangle)`
- `ftt = proc (Triangle) returns (Triangle)`

~~`fts ⊆ fss`~~ `fst ⊆ fss` ← Az eredmény lehet speciálisabb (monoton = kovariáns)

~~`ftt ⊆ fss`~~ `fss ⊆ fts`

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

Mikor biztonságos az S $\subseteq$ T altípus reláció?

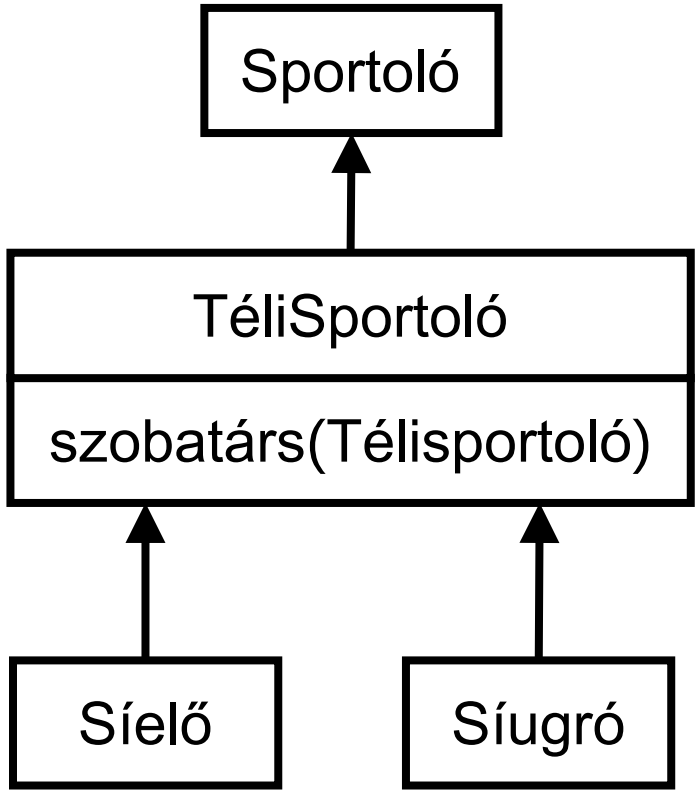
Az altípus metódusai megőrzik az őstípus viselkedését

- Szignatúra
  - Az argumentumok kontravarianciája, az eredmény kovarianciája
  - Az m_S kivételei benne vannak az m_T kivételei között
- Metódusok szabálya
  - Előfeltétel: „kontravariancia – altípus gyengébb”
  - Utófeltétel: „kovariancia – altípus erősebb”

Az altípusok megőrzik a szupertípusok tulajdonságait – típusinvariánsukra:

- „kovariancia – altípus erősebb”

Kontra/Ko-Variancia



Hogyan tudja `Síugró` felüldefiniálni `szobatárs`-at?

Kovariancia esetén
- `Síelő.szobatárs (Síelő)`
- `Síugró.szobatárs (Síugró)`

Kontravariancia esetén
- `szobatárs (Sportoló)`

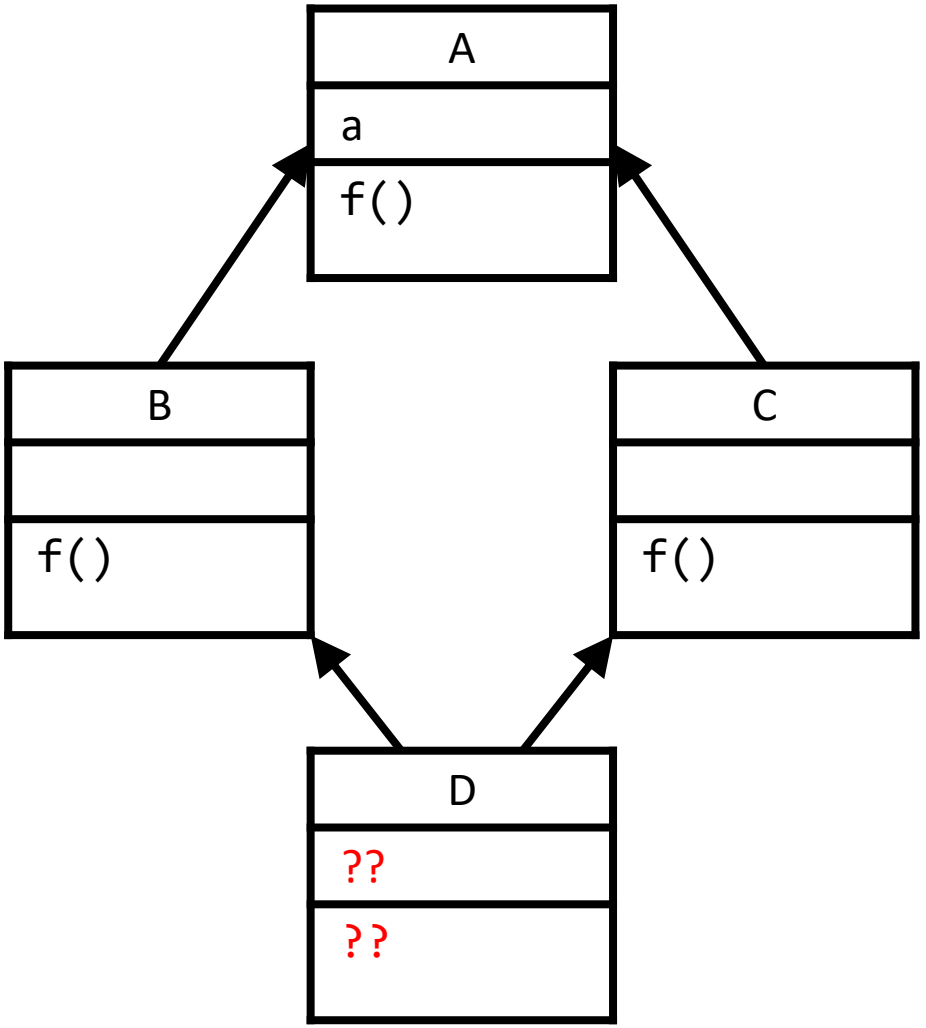
Novariancia esetén
- `szobatárs (TéliSportoló)`

Probléma (kovariáns esetben)
- `t:  TeliSportolo`
- `si: Sielo`
- `sg: Siugro`
- `t := si`
- `t.szobatars(sg)`

A többszörös öröklődés problémái

Diamond problem – A bázisosztály definiál f függvényt és a attribútumot.

- Ezt B és C osztályok függetlenül származtatják – felüldefiniálják
- D osztály B és C osztályokból közvetlenül származik többszörös örökléssel
- Ekkor
  1. A D-beli f melyiket jelentse?
  2. Az „a” attribútum hány példányban jelenjen meg D-ben?



A többszörös öröklődés problémái

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

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

Absztrakt osztály

Tervezés eszköze

Egy felső szinten összefogja a közös tulajdonságokat

A metódusok között van olyan, aminek csak specifikációja van, törzse nincs

Nem hozható létre példánya.

A leszármazott teszi konkréttá.

Interfész

Típusspecifikáció támogatása

„Szolgáltatások”

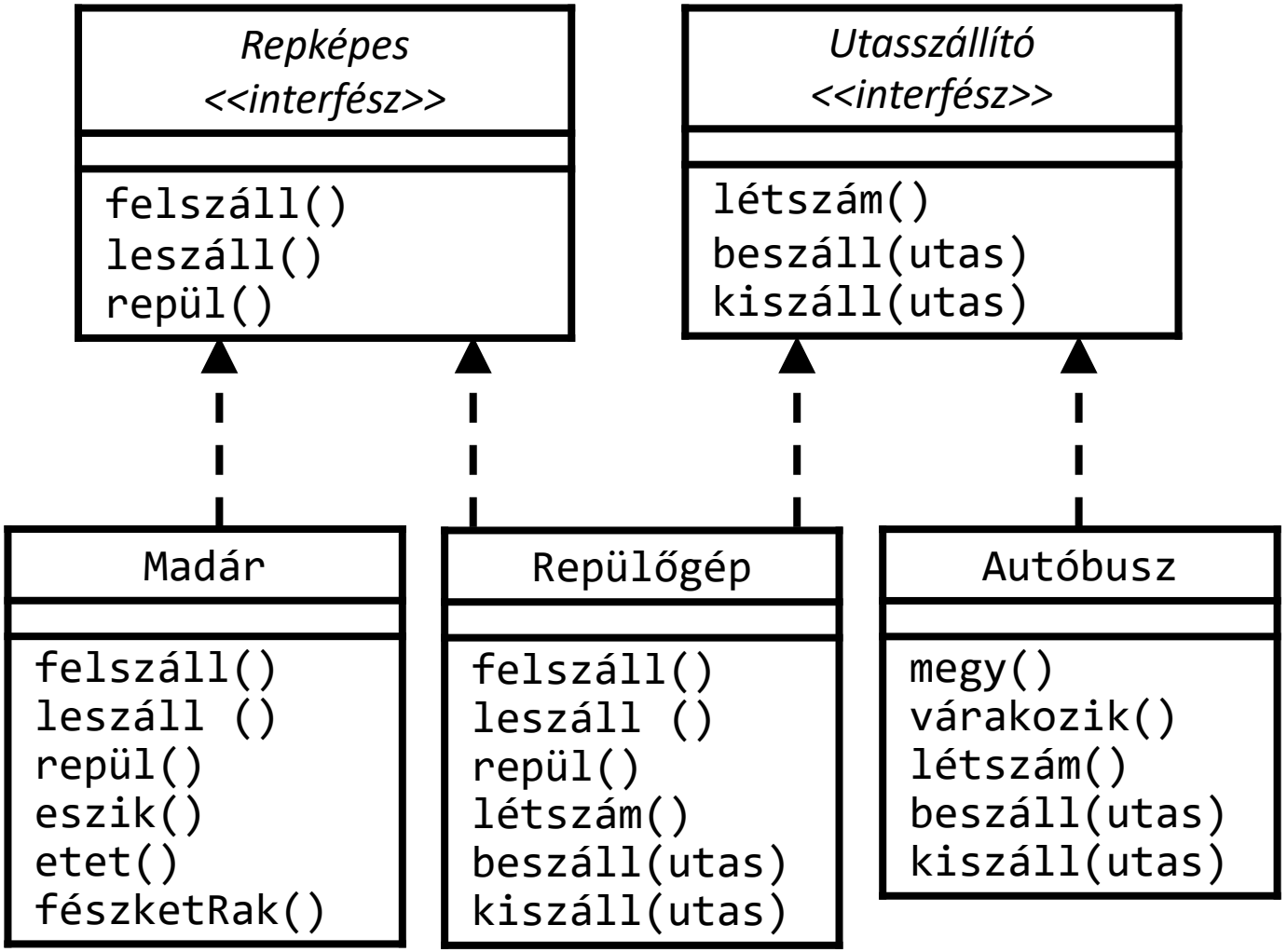
Meg kell valósítani a szolgáltatásait

Többszörös öröklődéssel nincs (annyi) probléma

- összefogja a közös jellemzőket

Interfész ≠ Absztrakt osztály!

Interfész



Interfészek

Implementáló osztályok

Mi az objektumorientált programozás?

A programozó definiálhat altípus kapcsolatokat

A típusszabályok megengedik, hogy az altípus használható legyen a szupertípus helyén

- altípusos polimorfizmus

Típus-vezérelt metódus elérés

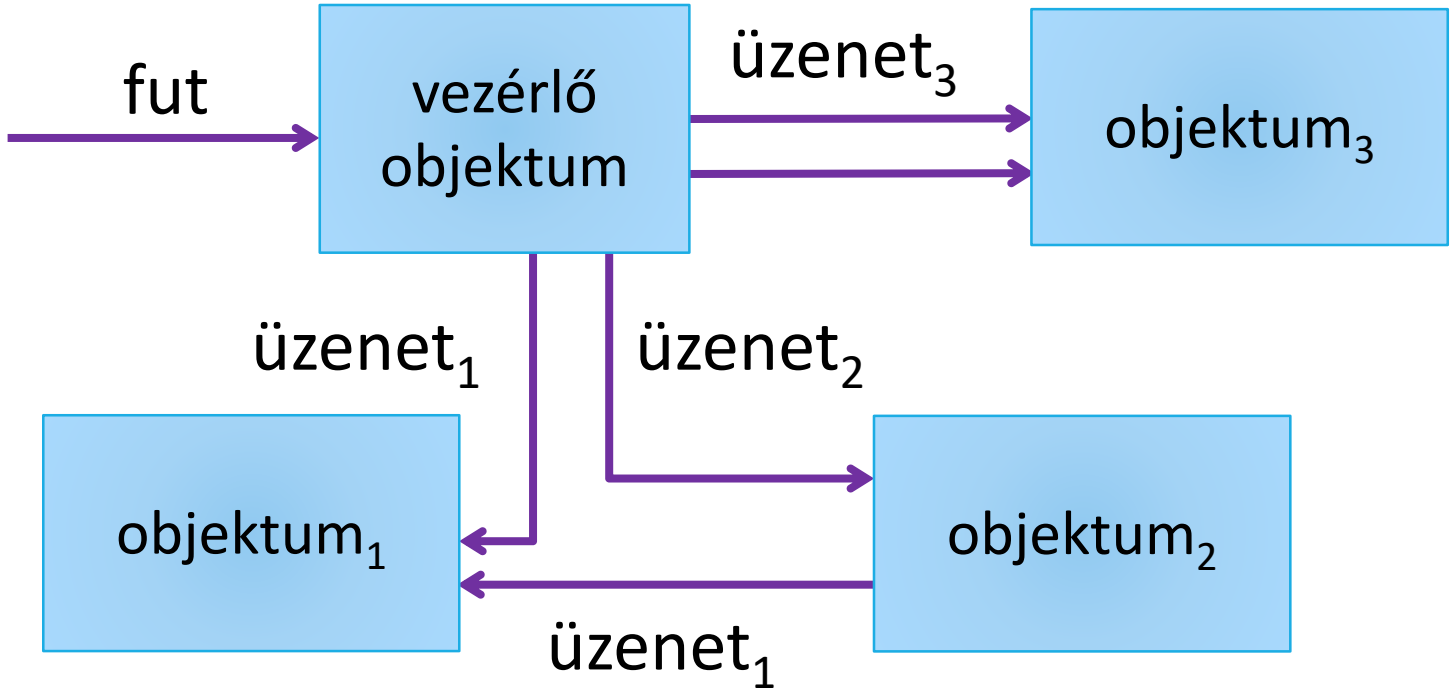
- dinamikus kötés

Implementáció megosztása

- öröklődés

OO program

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



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?

Kérdések

Van-e öröklődés?

Van-e többszörös öröklődés?

Adattagok és metódusok elrejtése hogy van megoldva?

Támogatja-e a polimorfizmust és dinamikus összekapcsolást?

- Esetleg csak mutatók használatával?

Van-e alapértelmezett ősosztály?

- Object?

Kérdések

Van-e absztrakt osztály?

- Nem példányosítható közvetlenül?

Konstruktor

Destruktor?

- Garbage Collector?

Standard objektum könyvtárak?

Osztályszintű attribútumok és metódusok?

C++

Osztálydefiníció

A C++ **`struct`** és **`class`** sok szempontból hasonló

- OOP elvárások tekintetében a **`class`** használata szükséges

Példa egy egyszerű osztály definíciójára

- Osztálydefiníciók egymásba ágyazhatók

```
class Rectangle {
    double a_side, b_side;
public:
    void print() const {  }
};
```

Példányosítás

```
Rectangle rect(10, 2);
Rectangle *rectpt = &rect;
```

Konstruktorok

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
- C++11 óta lehetséges
  - Mezők inicializálása
  - Konstruktor delegáció
  - Szülő osztály konstruktorának örökítése

Konstruktorok

```
class Rectangle {
    double a_side, b_side;
public:
    Rectangle(double a, double b): a_side(a), b_side(b) {};
    Rectangle() : Rectangle(1.0, 1.0) {};

    void print() const {
        std::cout << "Rectangle " << a_side << " " <<
                                     b_side << std::endl;
    }

    double calculateArea() {
        return a_side * b_side;
    }
};
```

Destruktor

Téglalap esetén nincs kifejezett haszna, de egy általános sokszögnél (illetve példa az inicializációra)

```
class Polygon {
    double *sides {nullptr};
    int nsides {0};

public:
    Polygon(double *data, int ndata): nsides(ndata) {
        sides=new double[nsides];
        std::memcpy(sides, data, sizeof sides);
    }
    ~Polygon() { delete [] sides; }

    void print() const {
        std::cout << "Polygon, number of sides " << nsides
                                                << std::endl;
    }
};
```

Osztályváltozó, osztálymetódus

```
class Circle {
    double radius;
    static double PI;
public:
    Circle(double r): radius(r) {};
    Circle() : Circle(1.0) {};

    void print() const {
        std::cout << "Circle " << radius << std::endl;
    }
};

double Circle::PI = 3.1415926535;
```

Az osztályon kívül definiálni kell!

Osztályváltozó, osztálymetódus

```
class Circle {
 //...
 double calculateArea() {
        return radius * radius * PI;
    }
 //...
};
```

Az osztályon belül elérhető közvetlenül, itt a külső használathoz publikus metódus kell!

```
static double getPi() { return Circle::PI; }
```

Osztályváltozó, osztálymetódus

Használat:

```
Circle::PI; // HIBA
Circle::getPi();
Circle c1(10);
c1.getPi();
```

Szemantikailag helytelen példányon keresztül elérni, azaz:

- ~~`c1.getPi();`~~
- ~~`return radius * radius * PI;`~~

Helyette

- `return radius * radius * Circle::PI;`

Láthatóság

Az adattagok és metódusok elrejtése megoldott

A láthatóság minősítője C++-ban

- **`public`**
  - a külső felhasználók elérik
- **`protected`**
  - csak a leszármazottak érhetik el
- **`private`**
  - csak az adott osztály és a barátok (**`friend`**) számára elérhető – alapértelmezés

Öröklődés

A téglalap egy specializációja a négyzet:

```
class Square: public Rectangle {
public:
    Square(double a): Rectangle(a, a) { };
    Square(): Square(1) { };

    void print() const {
        std::cout << "Square " << a_side << std::endl;
    }
};
```

Azonban ez nem működik, mert ugyan van saját `a_side` attribútuma a négyzetnek, de azt nem éri el közvetlenül, mivel az ősben ez **`private`**.

Öröklődés

Módosítsuk!

```
class Rectangle {
protected:
    double a_side, b_side;
 //...
};
```

Ekkor

```
rect.print();    // Rectangle 10 2
Square s(5);
s.print();       // Square 5
```

Konstruktor öröklés

Lehetséges C++11 óta

```
class BaseClass {
public:
    BaseClass(int value);
};

class DerivedClass : public BaseClass {
public:
    using BaseClass::BaseClass;
};
```

Altípusos polimorfizmus

Tekintsük az alábbi kódot

```
Rectangle rect(10, 2);
Square s(5);
rect = s;
```

Mi történik?

```
rect.print();
```

- Fontos, hogy itt még nincsen dinamikus kötés!

Altípusos polimorfizmus

Tekintsük az alábbi kódot

```
Rectangle rect(10, 2);
Square s(5);
rect = s;
```

Mi történik?

```
rect.print();
```

- Fontos, hogy itt még nincsen dinamikus kötés!

Altípusos polimorfizmus

Ha B (pl. `Square`) altípusa az A (pl. `Rectangle`) típusnak, akkor B objektumainak referenciái értékül adhatók az A típus referenciáinak.

Azaz:

```
Rectangle* prect = new Rectangle(4, 3);
Square* psqu = new Square(2);

prect->print();
psqu->print();
prect = psqu;
```

Mi történik?

```
prect->print();
```

Dinamikus összekapcsolás

C++ nyelven a dinamikus kötéshez:

- Az **őstípus** `virtual`-nak deklarálja a metódust, amire megengedi a felüldefiniálást
- Ellenkező esetben elfedés lesz!
- Tipikusan az ősbeli **`virtual`-**t a leszármazottban is **`virtual`-**nak jelöljük.

Dinamikus összekapcsolás

Módosítás a `Rectangle` osztályban

```
virtual void print() const {
    std::cout << "Rectangle " << a_side << " "
                              << b_side << std::endl;
}
```

Ekkor az előző példa

```
prect->print();
// Square 2
```

Felüldefiniálás jelzése

Expliciten jelezhetjük, hogy felüldefiniálást szeretnénk

```
class Square: public Rectangle {
    //...
    virtual void print() const override {//...
};
```

- Ellenőrizni fogja a fordító ekkor
- Hibát kapunk ha felüldefiniálandó függvény
  - nem **`virtual`**
  - **`final`**

Felüldefinálás (és leszármazás) megakadályozása

```
virtual void print() const final {//...
```

Virtuális destruktor

A destruktor esetén is fontos a **`virtual`** használata

- Amennyiben az altípusos polimorfizmus használjuk.

Nézzük az alakzatot

```
class Shape {
public:
    virtual void print() const {
        std::cout << "Generic Shape" << std::endl;
    }
};
```

Virtuális destruktor

Ha a Polygon ebből származik le

```
class Polygon: public Shape {
  //...
  ~Polygon() {
    delete [] sides;
    std::cout<< "Deleted" << std::endl;
  }
};
```

Akkor a következő eset memóriaszivárgás

```
Shape *sh = new Polygon(new double[2]{10.0, 10.3}, 2);
delete sh;
std::cout << "^^ Memory leak ^^ " << std::endl;
```

Virtuális destruktor

Helyesen

```
class Shape {
public:
    virtual ~Shape() {};
    virtual void print() const {
        std::cout << "Generic Shape" << std::endl;
    }
};
```

Így már rendben lesz

```
Shape *sh = new Polygon(new double[2]{10.0, 10.3}, 2);
delete sh;
std::cout << "^^ No memory leak ^^ " << std::endl;
```

Automatikusan létrehozott függvények

A fordító több tagfüggvényt létrehoz, ha nem definiáljuk azokat. Dönthetünk az automatikusan létrejövő tagfüggvények

- létrehozásáról (**`default`**)
- elutasításáról (**`delete`**)

```
class Square {
public:
    Square() = default;
    Square(const Square&) = default;
    Square& operator=(const Car&) = delete;
    virtual ~Square() = default;
};
```

Az utolsó sornál fontos a **`virtual`**.

- Ezzel jelezzük, hogy automatikusan létrejövő destruktor virtuális is legyen!

Absztrakt osztály

Pure virtual függvények bevezetése

```
class Shape {
 public:
      virtual ~Shape() {};
      virtual void print() = 0;
      virtual double calculateArea() = 0;
 };
```

Absztrakt osztály

Ilyenkor a leszármazott adja meg a definíciót:

```
class Circle : public Shape {
    double radius;
public:
    Circle(double r): radius(r) {};

    virtual void print() const override {
        std::cout << "Circle radius: " << radius << std::endl;
    }
    virtual double calculateArea() const override {
        return radius * radius * PI;
    }
};
```

Absztrakt osztály típusként

Természetesen nem példányosítható:

```
Shape s;
```

De statikus típusa egy változónak lehet:

```
Circle c;
Shape * pa = &c;
```

De statikus típusa egy változónak lehet és dinamikus kötés működik

```
pa->calculateArea(); // Circle::calculateArea()
```

Többszörös öröklés

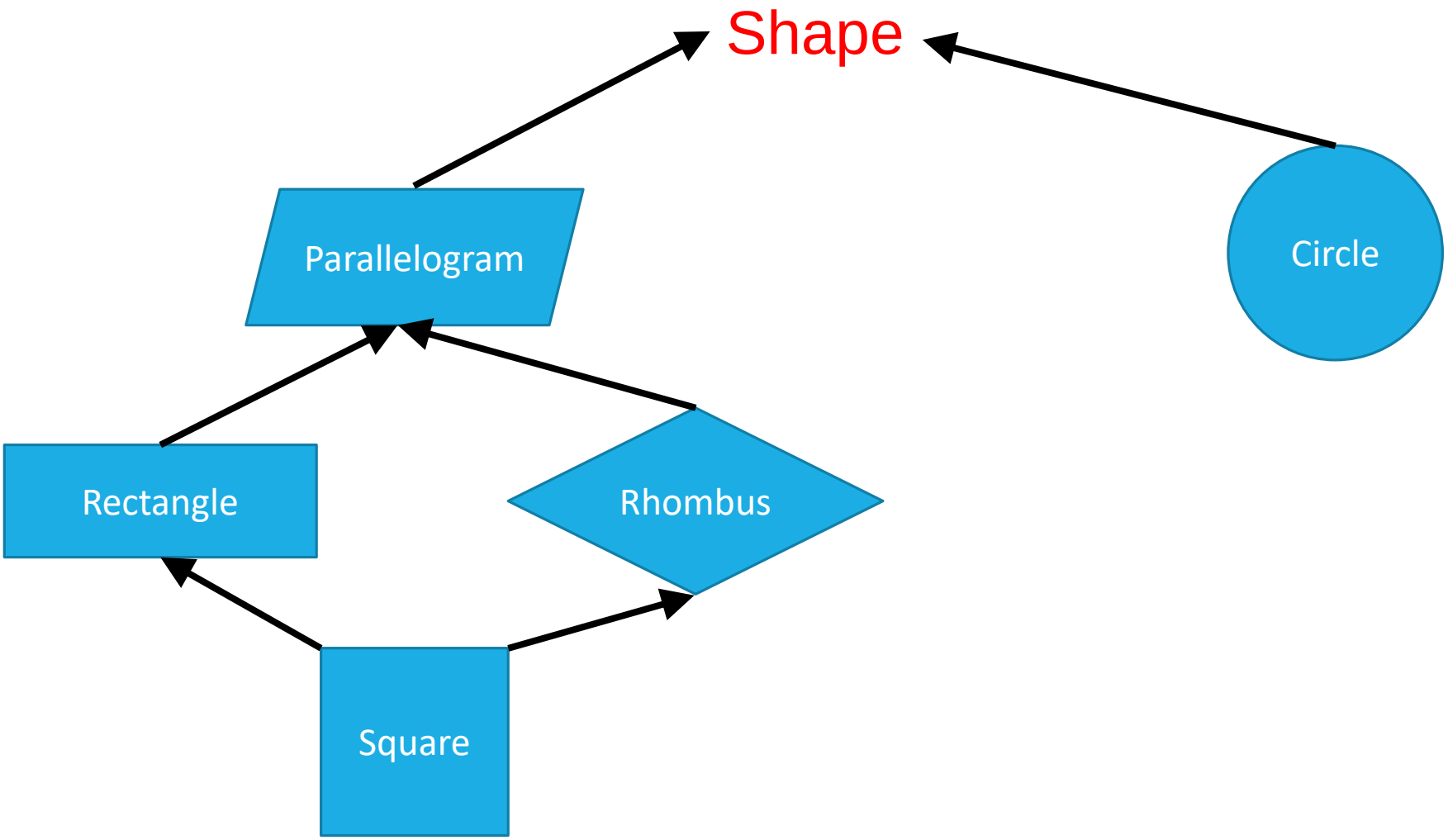
Az osztályhierarchia lehet fa, de lehet általánosabb körmentes gráf is:

- `class Shape`
- `class Circle: public Shape`
- `class Parallelogram: public Shape`
- `class Rhombus: public Parallelogram`
- `class Rectangle: public Parallelogram`
- `class Square: public Parallelogram, public Rhombus`

Problémák:

- Adattagok hányszor?
- Melyik metódus?

Többszörös öröklés



Többszörös öröklés

```
double PI = 3.1415926535;

class Shape {
public:
    virtual ~Shape() {};
    virtual void print() const = 0;
    virtual double calculateArea() const = 0;
};

class Circle : public Shape {
    double radius;
public:
    Circle(double r): radius(r) {};

    virtual void print() const override {
        std::cout << "Circle radius: " << radius << std::endl;
    }
    virtual double calculateArea() const override {
        return radius * radius * PI;
    }
};
```

Többszörös öröklés

```
class Parallelogram : public Shape {
protected:
    double _side_a;
    double _side_b;
    double _delta;
    double _height_a;
public:
    virtual double get_a_side() const final { return _side_a; }
    virtual double get_b_side() const final { return _side_b; }
    virtual double get_angle()  const { return _delta;  }
    virtual bool is_equilateral() const { return _side_a == _side_b; }
    virtual bool is_right() const { return _delta == 90; }

    Parallelogram(double a, double b, double angle): _side_a(a), _side_b(b),
                                                     _delta(angle) {
        _height_a = b * std::sin(_delta*PI / 180.0);
    }

    virtual void print() const override {
        std::cout << "Parallelogram a, b, delta: " << _side_a << ", "
                  << _side_b << ", "
                  << _delta << std::endl;
    }

    virtual double calculateArea() const override {
        return _side_a * _height_a;
    }
};
```

Többszörös öröklés

```
class Rhombus : public Parallelogram {
public:
    Rhombus(double a, double angle): Parallelogram(a, a, angle) {  };

    virtual void print() const override {
        std::cout << "Rhombus side, angle " << _side_a << ", " << _delta << std::endl;
    }

    virtual double calculateArea() const override {
        return _side_a * _side_a * std::sin(_delta*PI / 180.0);
    }

    virtual bool is_equilateral() const override final { return true; }
};

class Rectangle : public Parallelogram {
public:
    Rectangle(double a, double b): Parallelogram(a, b, 90) { _height_a = b; };

    virtual void print() const override {
        std::cout << "Rectangle " << _side_a << ", " << _side_b << std::endl;
    }

    virtual bool is_right() const override final { return true; }
};
```

Többszörös öröklés

```
class Square: public Rectangle, public Rhombus {
public:
    Square(double a): Rectangle(a,a), Rhombus(a, 90) { };

    virtual void print() const override final {
        std::cout << "Square " << _side_a << std::endl;
    }

    virtual double calculateArea() const override final {
        return _side_a * _side_a;
    }
};

int main() {
    Rectangle *s = new Square(10);
    std::cout << s->calculateArea() << std::endl;
    s->print();

    return 0;
}
```

Többszörös öröklés

Ebben a példában a `Square` esetén a `_side_a` mezőre hibát jelez a fordító

- A mezőt két ágon örökli, el kell dönteni, hogy hogyan legyen

1. Lehetséges megadni, például

```
virtual double calculateArea() const override final {
    return Rectangle::_side_a * Rectangle::_side_a;
}
```

- Ez azonban nem oldja meg a következőt

```
Parallelogram *s = new Square(10);
```

Többszörös öröklés

Ebben a példában a `Square` esetén a `_side_a` mezőre hibát jelez a fordító

- A mezőt két ágon örökli, el kell dönteni, hogy hogyan legyen

2. Virtuális öröklés

- Ilyenkor a virtuális ősosztály nem két irányban lesz ős, hanem egy közös ős
- Gondoskodni kell a virtuális ősök inicializációjáról is

```
class Rhombus: public virtual Parallelogram
class Rectangle: public virtual Parallelogram
Square(double a): Parallelogram(a, a, 90),
                  Rhombus(a, 90), Rectangle(a,a) { };
```

- Így már jó

```
Parallelogram *s = new Square(10);
```

Többszörös öröklés függvény példa

```
class A{
public:
    virtual void f() { std::cout << "A.f()" << std::endl; };
};

class B : public A {
public:
    void f() { std::cout << "B.f()" << std::endl; };
};

class C : public A{
public:
    void f() { std::cout << "C.f()" << std::endl; };
};

class D : public B, public C {
public:
    void g() { std::cout << "D.g()" << std::endl; };
};
```

Többszörös öröklés függvény példa

Használatkor

- `D d;`
  `d.f();`
- Hibaüzenet: `request for member „f” is ambiguous`

Itt is döntés kell

- `using C::f;`
- Azonban ilyenkor!
- `B *b = new D();`
  `b->f();  // B.f()`

Friend

A barát hozzáfér a privát adatokhoz, függvényekhez

- Tipikusan olyan operátorok készítésére használjuk, amelyek a modellezés szerint nem részei a típusnak, de szorosan kötődnek hozzá
  - Kiírás

```
class Point {
    friend ostream &operator<<( ostream &, const Point &);
public:
    Point( int = 0, int = 0 );
    void setPoint( int, int );
    int getX() const { return x; }
    int getY() const { return y; }
protected:
    int x, y;
};
```

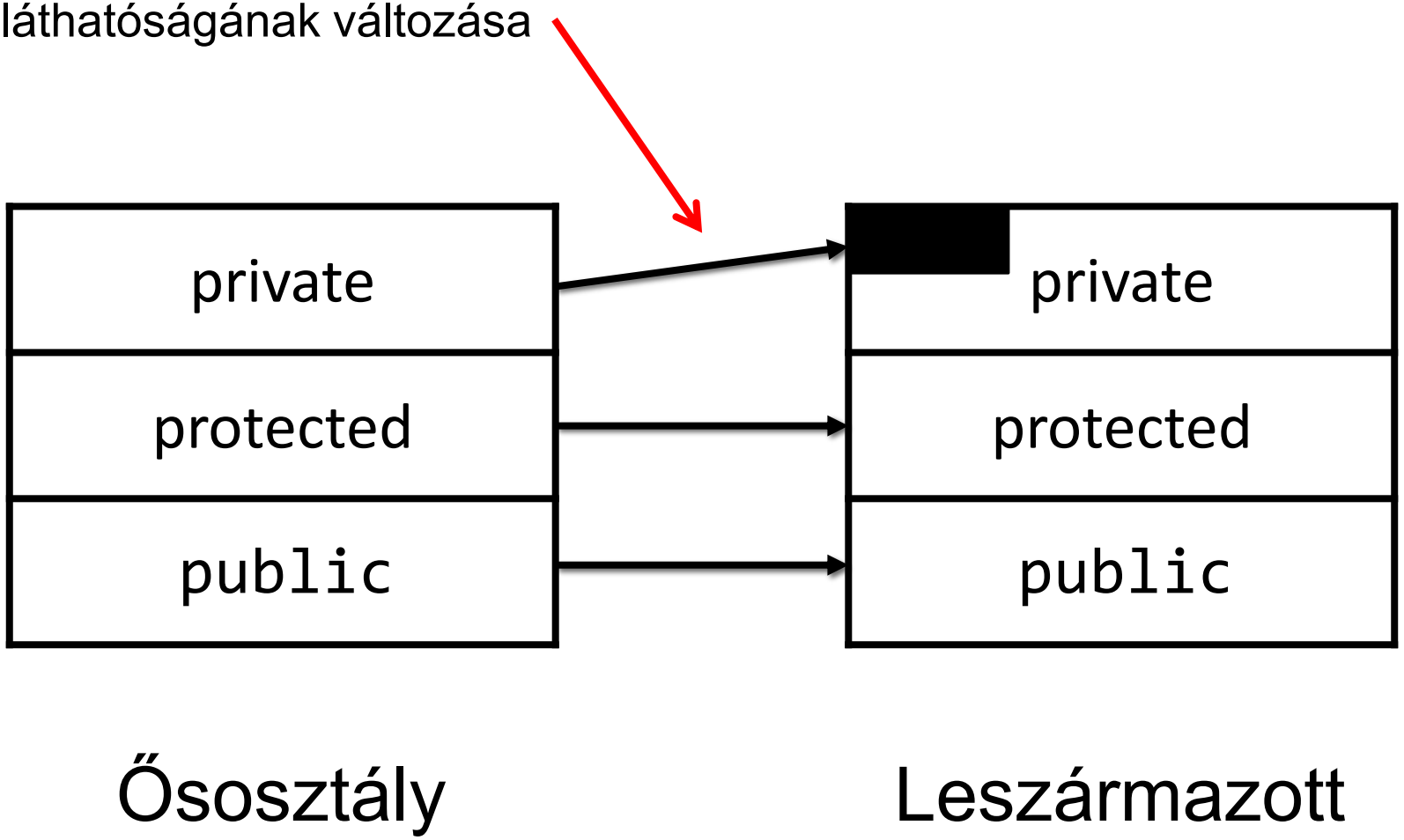
Alosztályképzés

C++ - implementációs öröklés altípus létrehozása nélkül

- **`private`**, **`protected`** öröklés

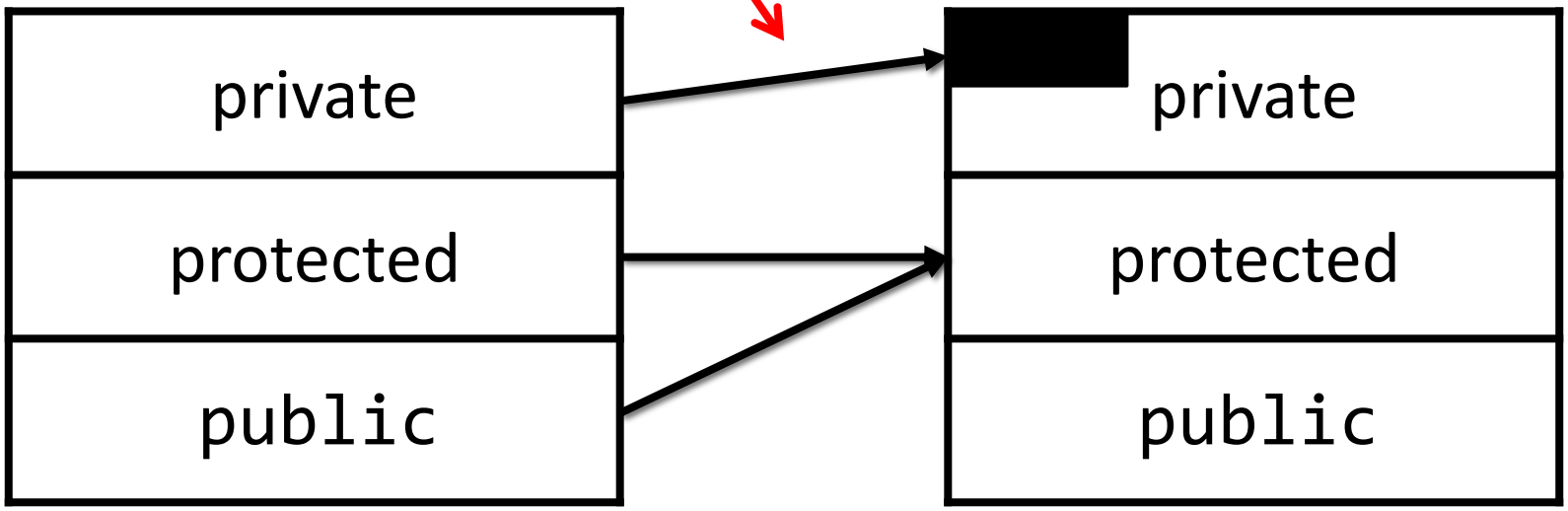
Public öröklés

Tagok láthatóságának változása



Protected öröklés

Tagok láthatóságának változása

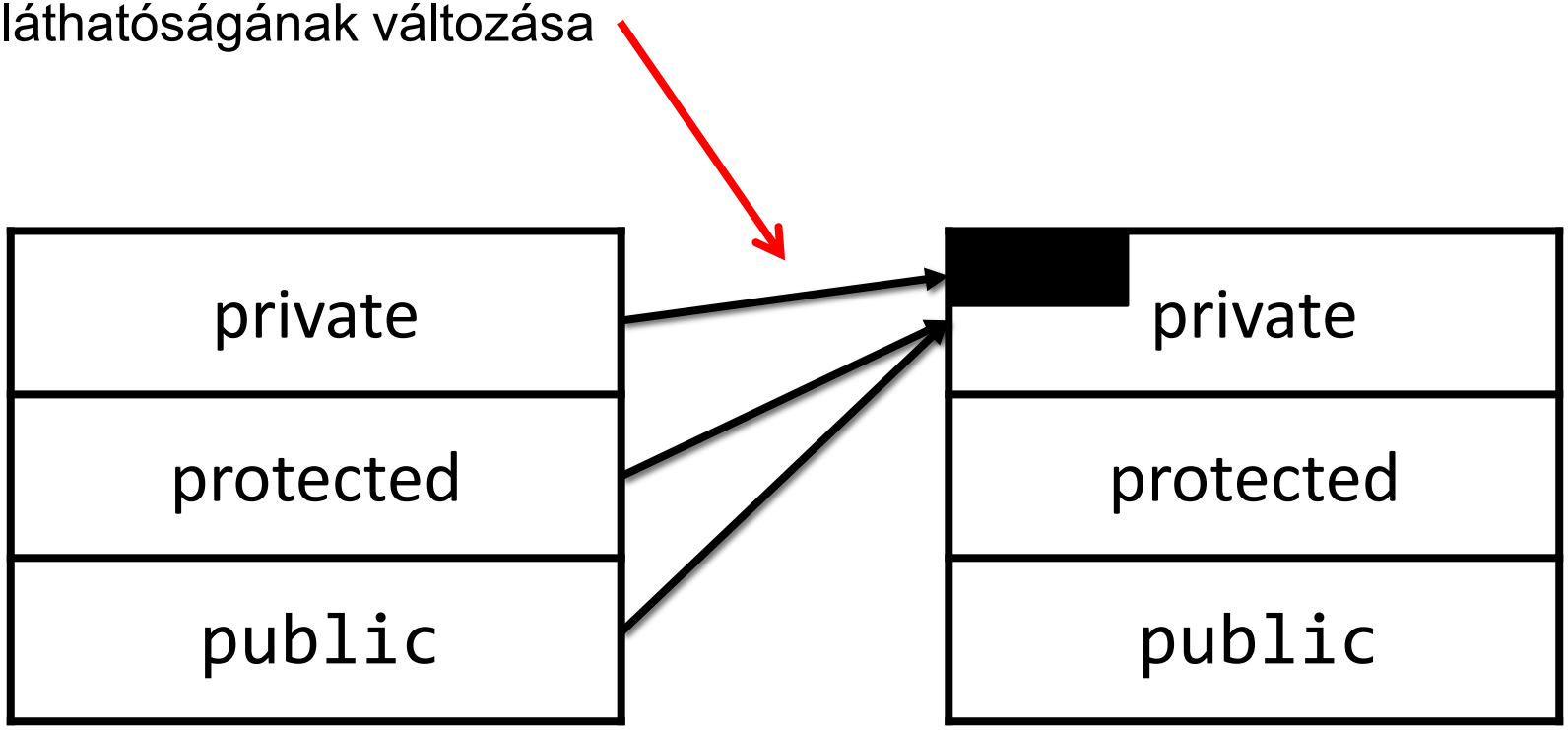


Bázisosztály

Származtatott osztály

Private öröklés

Tagok láthatóságának változása



Bázisosztály

Származtatott osztály

Kovariáns metódusok

Lehet bevezetni kovariáns és kontravariáns metódusokat az altípusban

- ezek elfedik (túlterhelik) az eredeti metódust, nem felüldefiniálják

```
class A {
public:
    int data;
};

class B : public A {
public:
    virtual void f(B *p)  { std::cout << "B.f(B) " << this->data << " " << p->data << std::endl; };
};

class C : public B {
public:
    // Kontravariáns
    // void f(A *p)  { std::cout << "C.f(A) " << this->data << " " << p->data << std::endl; };

    // Novariáns
    // void f(B *p) override { std::cout << "C.f(B) " << this->data << " " << p->data << std::endl;
};

    // Kovariáns
    void f(C *p)  { std::cout << "C.f(C) " << this->data << " " << p->data << std::endl; };
};
```

Kovariáns metódusok

Az osztálydefiníciókkal a következő kód nem fordul

- ```
  A *a = new A(); a->data=1;
  B *b = new B(); b->data=10;
  C *c = new C(); c->data=100;
  C *test = c;
  test->f(b);
  ```

A problémát az okozza, hogy a C osztályban van egy f() implementáció, amely C paramétert vár.

- Ezt a fordító megtalálja, ám a szignatúra nem megfelelő.
- Nem keres tovább, mert az elfedés miatt
- Megoldás: `using B::f;`
  ```

Kis kitérő – `struct`

Struct esetén minden tag **`public`** alapértelmezés szerint.

- Ez az információ elrejtést nem segíti, úgy mint az osztályoknál
- Azonban célszerű „sima” adatrekordok tárolására (rekord típus)
  - Plain Old Data Structure (POD)
  - C-vel kompatibilis

C++ – Összefoglalás

Mit örököl a leszármazott?

- Adattagokat
- Metódusokat

Mit nem örököl a leszármazott?

- Ősosztály konstruktorait, destruktorát
  - Konstruktor örökíthető
  - Leszármazottban delegálható rá, használható
- Ősosztály értékadás operátorát
- Ősosztály barátait

C++ – Összefoglalás

Mit vezethet be a leszármazott osztály?

- Új adattagokat
- Új metódusokat
- Felüldefiniálhat már meglévőket (virtual)
- 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)

OOP folytatás

KÖVETKEZŐ ALKALOMMAL