Programozási Ismeretek

Dr. Gergely Tamás
Dr. Ferenc Rudolf, Dr. Jász Judit, Dr. Kiss Ákos

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2024

(v0214)

Tartalom

Tartalom

Inicializálás – konstruktor

- Régebbi nyelvekben a nem biztonságos programozás fő okai:
  - inicializálás hiánya
  - eltakarítás hiánya
- Java-ban egy különleges operáció hívódik meg minden objektum létrehozásakor
  - **konstruktor**
  - neve = az osztály neve
  - garantált inicializálás az objektum létrejöttekor

```
class Alakzat {
    Alakzat() {
        /* inicializáló kód */
    }
}
```

```
Alakzat a = new Alakzat();
```

  - helyette lehetne pl. `initialize()`, de ezt mindig kézzel kellene meghívni, ami hibalehetőség

Inicializálás – konstruktor

(folyt.)

- A konstruktornak is lehetnek paraméterei
  - megadják, hogyan inicializáljunk
  - pl.: csak úgy lehet objektumot létrehozni, hogy megadjuk a pozíciót is

```
class Alakzat {
    Alakzat(int x, int y) {
        /* inicializáló kód */
    }
}
```

```
Alakzat a = new Alakzat(10, 15);
```

- A konstruktornak nincs visszatérési értéke
- Objektumra való hivatkozást kapunk

Operáció-kiterjesztés

statikus polimorfizmus

- Magasabb szintű nyelvekben **neveket** használunk
- Természetes nyelvben is lehet több értelme a szavaknak
  - a szövegkörnyezetből derül csak ki az értelme
- Programozásban ezt nevezzük **kiterjesztésnek** (**overloading**)
  - **statikus polimorfizmus**
  - Nem azonos az overriding-gal (felüldefiniálás-sal), amit *dinamikus polimorfizmus*-nak is hívunk
  - ~~túlterhelés~~
- Régebben, pl. C-ben, minden név egyedi volt

Operáció-kiterjesztés
(folyt.)

- Újabban szükségessé vált
  - pl. a konstruktornak csak egy neve lehet, mégis különböző konstruálásokat szeretnénk
- Metódusok kiterjesztése (nem csak konstruktor)
  - ugyanaz a neve de más a paraméter-lista (akár üres is lehet)
  - ugyanaz a feladat, miért lenne több különböző nevű függvényünk?

```
class Alakzat {
    Alakzat() {
        /* inicializáló kód */
    }

    Alakzat(int x, int y) {
        /* inicializáló kód */
    }
}
```

```
Alakzat a1 = new Alakzat();
Alakzat a2 = new Alakzat(10, 15);
```

Operáció-kiterjesztés
(folyt.)

- A fordító a kiterjesztett metódusokat a paraméterlistájuk alapján, fordítási időben különbözteti meg
- A hívás helyén az aktuális argumentumok típusai határozzák meg, hogy melyik metódus hívódik
- Konvertálható primitív típusoknál, ha nincs pontos egyezés, akkor az adat konvertálódik (de csak nagyobb típusokra!)
- A metódus visszatérési érték **nem** használható a megkülönböztetésre
  - pl.: `f();` hívásnál a visszatérési érték nincs is használva

Inicializálás – konstruktor

Default konstruktor

- **Default constructor**
  - argumentum nélkül hívható (paraméter nélküli) konstruktor
  - alapértelmezett beállításokkal rendelkező objektum létrehozására
- Ha nem definiálunk egy konstruktort sem, akkor (és csak akkor) a fordító készít egy *default*-ot
- Ha készítünk legalább egy konstruktort (akár *default*-ot, akár nem) akkor a fordító már **nem** készít *default*-ot

this

- Egy operáció kódja csak egy példányban van jelen a memóriában
- Honnan tudja a `rajzolj()`, hogy melyik objektumra lett meghívva?

```
class Alakzat {
    public void rajzolj() {
        /* kód */
    }
}
```

```
Alakzat a1 = new Alakzat();
Alakzat a2 = new Alakzat();
a1.rajzolj();
a2.rajzolj();
```

- Egy „titkos” implicit első paramétert generál a fordító
  - hivatkozás az aktuális objektumra
  - `this`-ként hivatkozható

```
struct Alakzat { /*...*/ };
void rajzolj(Alakzat this) {
    /* kód */
}
```

```
rajzolj(a1);
rajzolj(a2);
```

this
(folyt.)

- Mire jó? Amikor valamire explicite fel akarjuk használni, pl.:
  - ha a metódus formális paraméter neve megegyezik valamelyik attribútum nevével, vagy
  - ha egy metódus visszatér az objektummal

```
/* az első az attribútum,
   a második a paraméter */
this.x = x;
```

```
class Alakzat {
    public void rajzolj() {
        System.out.println("Alakzat");
    }
    public Alakzat kicsinyit() {
        /*...*/
        return this;
    }
}
```

```
Alakzat a1 = new Alakzat();
a1.kicsinyit().kicsinyit();
```

Változók és attribútumok inicializálása

- A lokális változókat kötelező explicit módon inicializálni (legalábbis az első használat előtt értéket adni nekik)
- Adattagok inicializálása
  - A definíció helyén inicializálhatók, vagy
  - ha nincsenek expliciten inicializálva, akkor 0-val ekvivalens értéket kapnak:
    - a primitívek `0`, `0.0`, `'\0'` illetve `false` értéket,
    - a nem primitívek (referenciák) pedig `null` értéket („nincs objektum”).
  - Ezeket felülírhatja az *instance initialization clause*-ban található értékadás.
  - Végül a konstruktorban is megadhatjuk az adattagok kezdőértékeit, ami felülírja az előzőeket.

Attribútumok inicializálása

Példa

```
class A {
    char c;              // default
    int i = 1;           // definíció helyén
    float f = init(); // függvénnyel
    B a = new B();       // nem primitív
    C c1;
    C c2;
    // instance initialization clause:
    {
        c1 = new C(1);
        c2 = new C(2);
    }
    A() {i = 2;}  // default konstruktorral
                  // előbb 1, utána 2
    A(int i) {this.i = i;}
}
```

Tömbök

- Tömb: névvel ellátott (egyetlen azonosítóval kezelt) azonos (primitív vagy objektum) típusú elemek sorozata
  - Java-ban a tömb is egy (Java nyelvi elemekkel támogatott) objektum
  - Deklaráció Java stílusban:

```
int[] a0;
```

  - ~~Deklaráció C stílusban:~~

```
int a0[];
```

  - A méretet a deklarációkor nem lehet megadni
    - A definíciókor (inicializáláskor, létrehozáskor) jön létre a megfelelő méretű tömb
  - Deklaráció után még csak egy referenciánk van tömbre
- Inicializálás
  - Inicializáló kifejezéssel:
  - Tömb másolással:
  - Referencia másolással:

```
int[] a1 = {1, 3, 4};
int[] a2 = a1.clone();
int[] a3 = a2;
```

Tömbök
(folyt.)

- Tömb objektumok mérete a `length` attribútumban tárolódik
  - csak olvasható

```
for (int i = 0; i < a1.length; ++i)
    System.out.println(a1[i]);
```

- Biztonságos, mert túlindexelésnél kivétel lesz dobva
- Inicializálás futás közben
  - A méret a deklarációkor még nem ismert, azt a definíciónál / létrehozásnál kell megadni
  - A lefoglalt terület elemei 0-ra (0-val ekvivalens értékre) lesznek inicializálva

```
int i = /* ... */;
int[] a1 = new int[i];
```

- Ezek az extra funkcionalitások lassíthatják a programot

Tömbök
(folyt.)

- Nem primitív típusok tömbjei esetén
  - csak referenciák tárolódnak, ezért
  - az elemeket egyesével, `new`-val kell lefoglalni, de
  - az inicializálás nagyon bonyolult is lehet.

```
Integer[] a1 = new Integer[3];
a1[0] = new Integer(500);
Integer[] a2 = new Integer[] {
  new Integer(1),
  new Integer(2),
}; // new Integer[] elhagyható
```

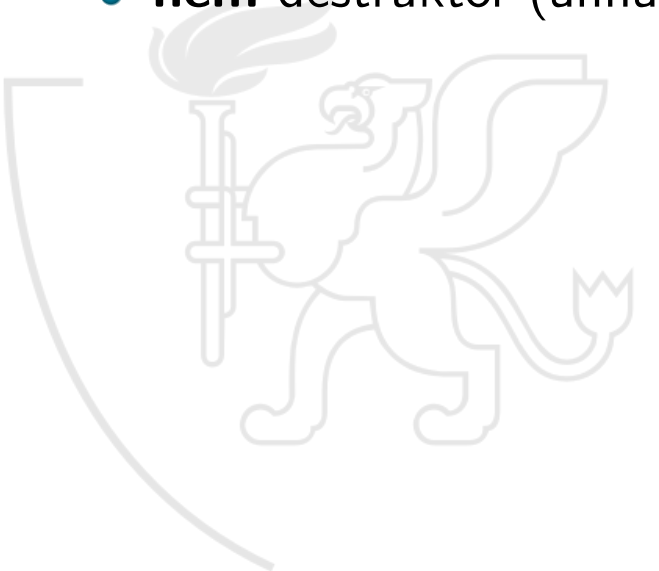
- Többdimenziós tömbök
  - hasonlóan deklarálhatók és használhatók, mint az egydimenziósok

```
int[][] a2d = { {1,2,3}, {4,5,6} };
int[][][] a3d = new int[2][3][5];
```

Tartalom

Kitakarítás

- Nem csak a korrekt inicializálás fontos, hanem az erőforrások helyes kezelése (felszabadítása) is
- Java: **garbage collector**
  - csak memóriával foglalkozik
  - minden kapcsolódó objektum el lesz takarítva
- Segítség: `finalize()`
  - takarítás előtt hívódik
  - **nem** destruktor (annak a végrehajtása szavatolt lenne)!



Szemétgyűjtés

garbage collector

- Fontos: a szemétgyűjtés végrehajtása nem szavatolt!
  - pl. ha még sok szabad memória van és így elkerülhető a program fölösleges lassítása
  - kiszámíthatatlan, hogy mikor fog (fog-e egyáltalán) hívódni
- Tehát **nem** a destruktor szerepét tölti be
  - fontos, hogy a mindig végrehajtandó feladatokat ne a `finalize()`-ba tegyük (pl. fájl lezárás)
- Kézzel is indítható

```
System.gc();
System.runFinalization();
```

- Működés: több algoritmus szerint, de ez mind rejtve marad
  - pl. referenciaszámlálás

Tartalom

Implementáció elrejtése

- Osztálykönyvtárak esetén különösen fontos:
  - A kliens kódja változatlan maradhat ha a könyvtár változik,
  - a kívülről elérhető részek nem változnak, de
  - az elrejtett rész (az implementáció) szabadon változhat.
- Egy osztálykönyvár elemeinek elérését hogyan szabályozhatjuk?
- Erre valók az *elérés vezérlők* (access specifiers), amelyek a láthatóságot befolyásolják:
  - `public`
  - `protected`
  - *friendly*, vagy *package private*
  - `private`
- Az elérés vezérlés teljes értelme a csomagok (*package*-ek) használatával jön elő

Csomagok

- Miért van rá szükség?
  - logikai csoportosítás
  - nevek ütközésének elkerülése
- Ha nincs megadva (mint az eddigi példákban), akkor minden új osztály egy ún. *default package* csomagba kerül
- Ha nagy rendszert írunk, célszerű csomagokat használni

Java forráskódok

- Egy-egy Java program írásakor *fordítási egységeket* (compilation unit, translation unit) készítünk
  - `.java` fájl $\rightarrow$ `.class` fájl(ok)
- Minden fordítási egységben
  - lehet egy *publikus osztály* (`public class ...`), amelynek a neve meg kell egyezzen a fájl (`.java` kiterjesztés nélküli) nevével, és
  - lehetnek a fájlban egyéb (rejtett) osztályok is.
- Egy `.java` fájl fordításával `.class` fájlok jönnek létre (minden egyes osztályhoz egy-egy)
- Java-ban nincs linker (a `.class` fájlok önállóak)
  - Be lehet őket tömöríteni egy `.jar` fájlba (ami valójában egy `.zip`)
  - A `.class` fájlokat az interpreter értelmezi és hajtja végre egyenként

Csomagok megadása

- A csomag (package) nem más mint `.class` fájlok halmaza
- Egy osztály melyik csomagba tartozik?
  - Az osztályt tartalmazó forrásfájl első nem-komment sorába kell írni:

```
package mypackage;
```

  - Ez azt jelenti, hogy a benne levő osztályok a megadott csomaghoz fognak tartozni.
- Az egy csomagba tartozó osztályok egy lokális könyvtárba kerülnek
  - Nagyobb rendszerek fordításakor az osztályokat tartalmazó `.java` forrásfájlokat ugyanabban a könyvtárban *célszerű* elhelyezni, és a könyvtár neve egyezzen meg a csomag nevével

Csomagok használata

- Csomagbeli osztály használata teljes hivatkozással:

```
mypackage.MyClass m;
```

- Importálás után elegendő az osztály nevét használni:

```
MyClass m;
```

  - Importálni lehet egész csomagot,

```
import mypackage.*;
```

  - de egyetlen osztályt is.

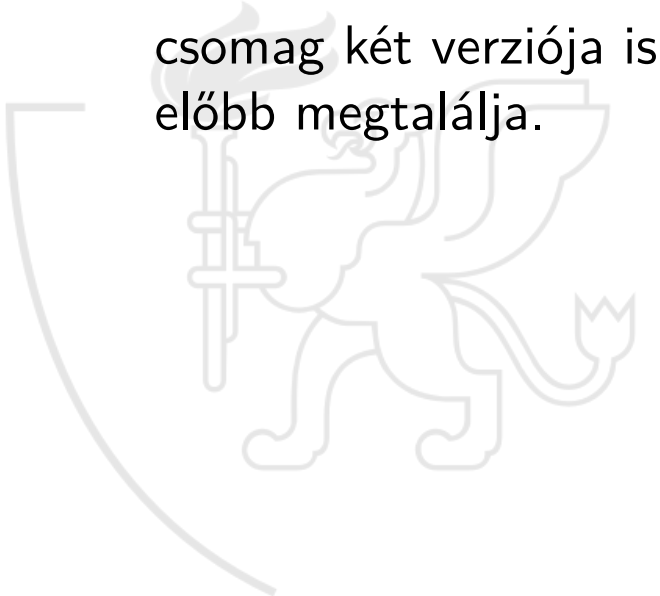
```
import mypackage.MyClass;
```

Csomagok nevei

- Minden csomagnak egyedi teljes neve van:
  - A teljes elérési útvonal bele van kódolva, sőt,
  - konvenció szerint az internet domain név is (fordított sorrendben, pl. `com.sun.java.*`), de
  - ez utóbbinak csak (erősen) javasolt a használata.
- Hogyan találja meg a Java interpreter, virtuális gép (JVM) az osztályokat?
  - Be kell állítani a `CLASSPATH` környezeti változót, ami a keresési könyvtárakat és `.jar` fájlokat tartalmazza
  - A csomagok nevei meg kell, hogy egyezzenek a csomag elemeit tartalmazó könyvtár nevével (hierarchikusan)
  - Az osztályok kódját az ugyanolyan nevű `.class` fáljoknak kell tartalmaznia a megfelelő alkönyvtárban

Csomagok nevei
(folyt.)

- Névütközések
  - Ha két `import valami.*` -szerű általános csomag-import van, és mindkettőben van azonos nevű osztály amit használni is szeretnénk – amíg nem használunk ilyent addig nincs baj –, akkor expliciten ki kell írni a csomag nevét az osztály elé.
- Megtalált osztályok
  - Ha a `CLASSPATH`-ban definiált két különböző keresési helyről ugyanazon csomag két verziója is elérhető, akkor a JVM azt használja, amelyiket előbb megtalálja.



Osztályok elemeinek elérése

Láthatóság megadása

- Az osztály minden egyes tagja elé oda kell írni az elérés vezérlés kulcsszavát:
  - `public`: mindenki bárhonnan elérheti az osztály adott elemét
  - `protected`: csak a csomag többi osztályából és a leszármazottakból hivatkozható
  - ha nincs kiírva, akkor „frendly”, vagy „package private”:
    - csomagon belül mindenhonnan elérhető (public)
    - csomagon kívülről nem elérhető (private)
  - `private`: csak az adott osztályból, annak elemeiből hivatkozható

Osztályok elemeinek elérése

Interfész és implementáció

- Minden osztály saját maga határozza meg, hogy ami elérhető belőle és kinek
  - Az interfész (definíció szerint) mindig `public` (ami publikus, az az osztály interfésze)
  - Az implementáció **ne** legyen publikus:
    - ami nem kell másnak az mindig legyen `private`
    - amit a leszármazottak is használnak, az lehet `protected`
    - a csomagban több osztály által megvalósított funkciókhoz szükséges elemek lehetnek *frendly*-k
- A konstruktornak is megadhatjuk (korlátozhatjuk) az elérését, így lehet például a direkt létrehozást megakadályozni

Osztályok elemeinek elérése

Példa

```java
package sutemeny;

class Suti {
    private int mennyiseg = 100;
    protected void harap() {
        System.out.println("Belémharaptak!");
        mennyiseg -= 25;
    }
    protected boolean maradt() {
        return mennyiseg > 0;
    }
}

public class Ludlab extends Suti {
    public void megesz() {
        while(maradt()) {
            harap();
        }
        System.out.println("Megettek!");
    }
}
```

Másik csomag!

```java
import sutemeny.*;

public class SutiPelda {
    public static void main(String[] args) {
        Ludlab a = new Ludlab();
        a.megesz();
    }
}
```

```java
import sutemeny.*;

public class SutiRosszPelda {
    public static void main(String[] args) {
        Ludlab a = new Ludlab();
        a.harap();                  // nem elérhető
        a.megesz();
        Suti b = new Ludlab();  // nem elérhető
    }
}
```

Elérés vezérlése

Csomagok elemei

- Csomagon belül az egyes osztályok elérését is lehet szabályozni
  - `public` osztály
    - csomagon kívülről is látható, elérhető
    - `.java` fájlonként csak egy publikus osztály lehet (de nem kötelező), az amelyik a fájl nevét viseli (kis- és nagybetű is számít!)
  - A publikuson kívül lehet még *friendly*, ha nincs semmi kiírva
    - csomagon kívülről nem látható
  - `private` és `protected` csak *inner class* lehet