Programozási nyelvek Java

Kozsik Tamás előadása alapján
Készítette: Nagy Krisztián

7. előadás

1.) Osztálybetöltés (lusta!) minden függőséggel -

- class loader
  - class fájlt a fájlrendszerről betölti
  - …
- programozható
  - bájtkód verifier
  - dinamikus szerkesztés

2.) static mező inicializáció

1.)-ben 0-ra inicializálódik
2.)-ban utólag a megadott értékekre inicializálódik + osztályszintű inicializátor blokk

new Sub() esetén:

1. x = 0, y = 0, z = 0, u = 0
2. Lefut az Object osztály konstruktora
3. y = 5
4. y = 3
5. z = 1
6. u = 1
7. u = 9

```
class Base{
        int x;
        int y = 5;
        int z;
        Base(int y, int z){
                this.y = y;
                this.z = z;
        }
}
class Sub extends Base{
        int u = 1;
        Sub(int u, int z){
                super(3,z);
                this.u=u;
        }
        Sub(){
                this(9,1);
        }
}
```

begenerálódik egy paraméter nélküli super() hívás

java.lang.Object default konstruktora

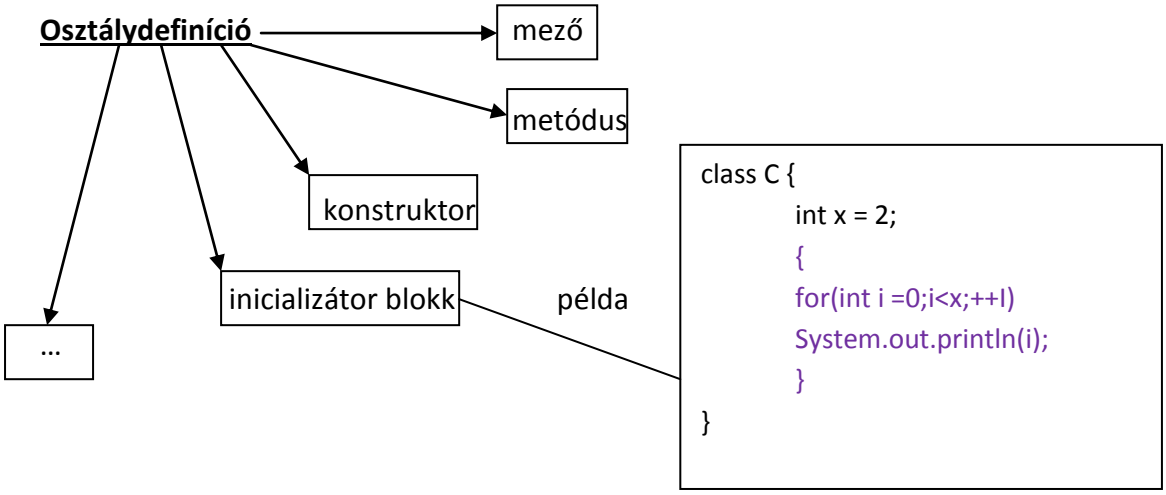
- allokáció
- nuilával való feltöltés
- konstruktor hívás (this-eken, super-eken felfegyek az Object osztály konstruktoráig)
- az utolsó super() hívás után a konstruktor hívása egy pillanatra felfüggesztődik
- explicit mezőinicializáló utasítások és osztályszintű inicizátor blokk (pl.: y = 5)
- folytatódik a konstruktorhívás

- hívások mentén a 0-val való feltöltéseken túl minden utasítás az osztályban (beleértve az inicializációkat az osztályban) támaszkodhat a teljesen inicializált (konstruktor is befejeződött) belső állapotra a bázisokból.

pl.: bázis típusinvariánsa

Megjegyzés: Metódusban(void m()): int x; int y = x; - fordítási hiba! (x inicializálatlan)
A fenti folyamatok az objektum mezőire vonatkoznak!

Összegezve tehát a konstruktor this hívással kezdődik vagy a konstruktor super hívással kezdődik vagy nem az előbbiek, de ekkor generálódik a konstruktor első utasításába egy paraméter nélküli super(). Azaz a legelső tevékenység a super(…) hívás. (Ha csak nem … minimális trükközés, de a compiler igyekszik megakadályozni)



static inicializátor blokk: static {…} – metóduson kívüli blokkutasítás

Láthatóság

- public – csomagon kívül is látható
- protected – leszármazottak esetén is olyasmi, mint a package private
- package private (nem írunk semmit) – csak az adott csomagon belül látható
- private – csak az adott csomagban, az adott osztályleírásban látható

Ha az osztály nyilvános láthatóságú, akkor nyilvános láthatóságú default konstruktor generálódik bele, amennyiben package private, úgy package private:

public class PC{…} ⟶ public PC() { super(); }
class C{…} ⟶ C() { super(); }

final kulcsszó

Csak egy értéket kaphat az adott változó, eltekintve az inicializáció alacsonyszintű részletétől mezők kapcsán. (Majdnem olyan, mint a konstans)

```
class Point{
      int x,y;
      final Color a = Color.BLACK;
}
```

gyakran static

```
class Emp{
      final String name;
      Emp(String name) {
            this.name = name;
      }
}
```

blank final

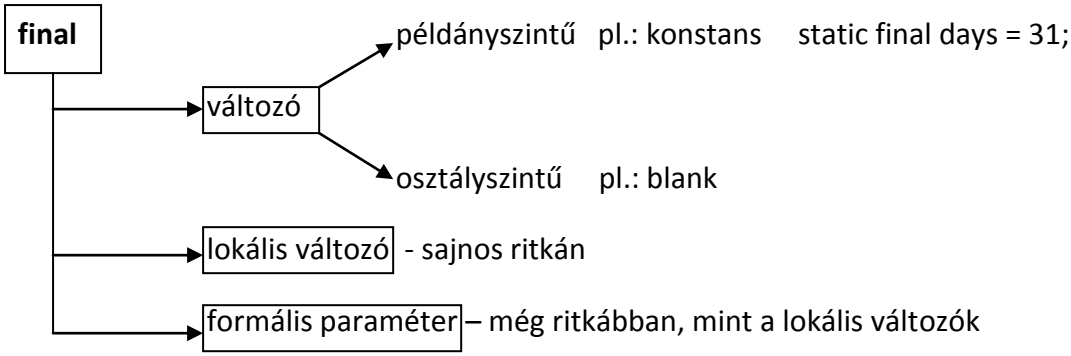
Gyakran az egyszeri értékadásunk a konstruktorra bízzuk, ekkor a final változónk, blank final. Funkcionális programozói stílus.

A párhuzamos programozás ebben a stílusban hatékonyabb, mint az állapot változós programozás.

```
class Point{
      final int x,y;
      Point(int x, int y){
            this.x = x;
            this.y = y;
      }
      Point move(int dx, int dy){
            return new Point(x+dx,y+dy);
      }
      …
}
```

final

funkcionális stílus (immutable) ⟶ nem változtatunk állapotokat



Referenciák?

- példánymetódus – nem definiálható felül
- osztályra – nem lehet használni!

```
class Point{int x; int y; … } Mutable!
class Line{

   final Point p1,p2;
   Line(int x1,int y1, int x2, int y2){

         p1 = new Point(x1,y1);
         p2 = new Point(x2,y2);

   }

   void move(int dx, int dy){

         p1.move(dx,dy);
         p2.move(dx,dy);

   }
   …

}
```

Megjegyzés: p1,p2 nem kap csak egyszer értéket, de a belső állapota a hivatkozott objektumnak változhat!

Ellenőrizni kell még azt, hogy nem esik egybe a két pont!