Programozási Ismeretek

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

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2024

(v0214)

Tartalom

Tartalom

Újrafelhasználhatóság

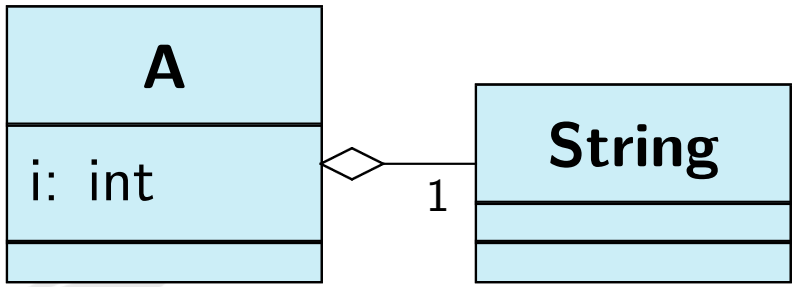
- Az OOP (és a Java nyelv) egyik legvonzóbb tulajdonsága a kód hatékony újrafelhasználásának lehetősége
- **De:** újrafelhasználás $\neq$ egyszerű másolás (klónozás)
- Régebbi nyelvekben a procedurális programozás volt az egyetlen lehetőség
- Ennél jobb az osztály újrafelhasználása
  - A cél: ne kelljen minden osztályt újból megírni, hanem építőkockaként lehessen felhasználni a meglévőket
  - A lényeg: mindezt úgy csinálni, hogy a már meglévő kódot (osztályokat) ne módosítsuk

Újrafelhasználhatóság
(folyt.)

- Kompozíció
  - új osztályban meglévő osztályokból készítünk objektumot
- Öröklődés
  - új osztály létrehozása egy meglévő „altípusaként”
  - új funkcionalitás hozzáadásával
- Ezek a módszerek az alap-építőelemi az OOP-nek

Kompozíció

- *Kompozíció*: összetétel, aggregáció, rész-egész kapcsolat
- Egy osztály egy attribútumának típusa lehet:
  - egy másik osztály, vagy
  - primitív típus.




```
class A {
    private String s;
    private int i;
}
```

  - A fenti példában mindkét adattag inicializálva lesz: `S` kezdőértéke `null`, `i` kezdőértéke pedig 0.

Öröklődés

- Minden OOP nyelvnek szerves része
- Java-ban minden új osztály implicite örököltetve van az `Object`-ből (direkt, vagy indirekt módon)
- Új osztály származtatása meglévőből: „olyan mint a régi”
  - bővíti azt (`extends` kulcsszó)
  - a származtatott az ős minden adatát és metódusát „megkapja” – örökli az őstől
- Öröklődés
  - a származtatott- (al-, gyermek-) osztály egy speciális esete az ős osztálynak
  - az ős- (alap-, szülő-) osztály az általános esete a gyerek osztályoknak

Öröklődés
(folyt.)

- Az ősnek van néhány interfész (publikus) metódusa
- A származtatott mindet tartalmazza
  - az interfész újra fel van használva
- a `surolj()` felüldefiniálja az ősét (overriding)
  - specializálja a viselkedését
  - az őst is hívhatja `super` segítségével (egyébként rekurzív lenne)
- az `aztass()` új metódus, bővítése az ősnek

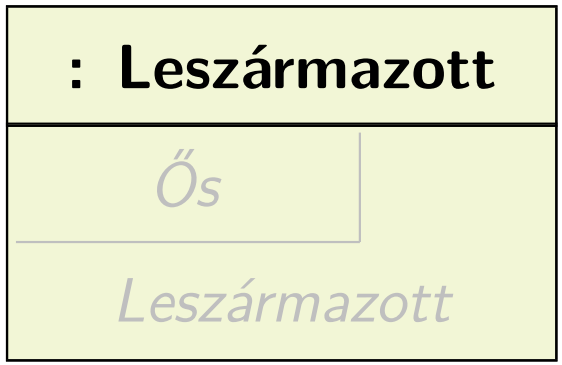
```
class TisztitoSzer {
    public void tisztits() {
        System.out.println("Tisztítószerrel␣tisztít");
    }
    public void surolj() {
        System.out.println("Tisztítószerrel␣súrol");
    }
}
```

```
class MosoPor extends TisztitoSzer {
    // metódus módosítása (felüldefiniálás):
    public void surolj() {
        super.surolj();
        System.out.println("Mosóporral␣súrol");
    }
    // új metódus hozzáadása:
    public void aztass() {
        System.out.println("Mosóporral␣áztass");
    }
}
```

```
public class MosoPorMain {
    public static void main(String[] args) {
        MosoPor mp = new Mosopor();
        mp.aztass();
        mp.tisztits();
        mp.surolj();
    }
}
```

Ős osztály inicializálása

- Öröklődés nem csak az osztály interfészét „másolja”
- A származtatottból létrejövő objektum rész-objektuma lesz az ősnek!
  - Az ős objektum-rész ugyanolyan marad
  - + ki van egészítve a származtatott új elemeivel



- Az őst (őstől örökölt objektum-részt) is inicializálni kell
  - Ki tudja, hogyan kell ezt csinálni?
  - Az ős!
    - „egyszer” már megtette
    - a privát részekhez csak ő fér hozzá

Ős osztály inicializálása
(folyt.)

- A származtatott konstruktorában hívni kell az ős konstruktorát
  - A default ős-konstruktor impliciten meghívódik (ha nincs explicit konstruktorhívás és van default ős-konstruktor)
  - A nem default ős-konstruktort expliciten meg kell hívni a `super` segítségével

```
class Jatek {
    Jatek(int i) { /* ... */ }
}

class TablaJatek extends Jatek {
    TablaJatek(int i) { super(i); /* ... */ }
}

public class Sakk extends TablaJatek {
    Sakk() { super(64); /* ... */ }
}
```

Kompozíció vagy öröklődés?

- Sűrűn használatos a kettő együtt
- Mi a különbség és mikor melyiket használjuk?
  - Az új osztály mindkettőnél rész-objektumokat tárol
  - Az ős osztály inicializálására a fordító kényszerít, de az adattag-objektumokra nekünk kell figyelni!
  - **Kompozíció**: egy meglévő osztály funkcionalitását felhasználjuk, de az interfészét nem
    - a kliens csak az új osztály interfészét látja
    - a beágyazott objektum általában `private` láthatóságú
    - pl. Autó és Kerekek
  - **Öröklődés**: egy meglévő osztály speciális változatát készítjük el (specializálunk)
    - a kliens az ős és az új osztály interfészét is látja
    - pl. Jármű és Autó

Kompozíció vagy öröklődés?
(folyt.)

- Sokszor hajlamosak vagyunk arra, hogy mindenhol az öröklődést használjuk, mert az OOP tulajdonság!
- Nem mindig ez a jó!
- Kezdetben induljunk ki mindig a kompozícióból, és majd ha kiderül, hogy az új osztály mégis egy speciális típusa a másiknak, akkor származtassunk
- Származtatásnak elsődlegesen akkor van létjogosultsága, ha ősre való konvertálás is szükséges lesz
- Túl sok származtatás és mély osztályhierarchiák nehezen karbantartható kódot eredményezhetnek!

Ősre konvertálás

- Mivel a származtatott az ős osztály egy „új típusa”, logikus hogy mindenhol ahol az ős használható, ott a származtatott is használható.
- Ez azt jelenti, hogy ahol ős típusú objektumot vár a program, ott a származtatott egy implicit konverzión megy át az ősbe.

```
class Hangszer {
    public void szolj(Hang h) {
        System.out.println("A␣hangszer␣"
                + h + "-es␣hangon␣szól.");
    }
}
```

```
class Zongora extends Hangszer {
    public void szolj(Hang h) {
        System.out.println("A␣zongora␣"
                + h + "-es␣hangon␣szól.");
    }
}
```

```
class Hangolo {
    public static void hangolj(Hangszer h) {
        h.szolj(Hang.C);
    }
}
```

```
Zongora z = new Zongora();
Hangolo.hangolj(z); // upcasting
```

Tartalom

A `final` kulcsszó

- Több jelentése van, de a lényeg: ami el van látva vele az nem változhat (végleges/végső)
- Miért akarjuk korlátozni?
  - tervezési megfontolásból
  - hatékonyság miatt
- Négy helyen használható
  - adat
  - metódus
  - paraméter
  - osztály

Végső adatok

- Konstans adat létrehozása hasznos lehet
  - ha már fordításkor eldönthető (konstans propagálás)
  - futás közben inicializálva, de onnantól kezdve konstans
- Konstans primitív adat létrehozására egyszerűen ki kell írni a definíció elé (lehet `static` is, ebben az esetben a címe is fix)
- Nem primitív típusra használva nem az objektum lesz konstans hanem a **referencia**
  - inicializálás után másra már nem mutathat, de maga az objektum megváltozhat
  - Java-ban az objektumok nem tehetők konstanssá, csak a viselkedésükkel lehet azt „szimulálni”
- Üres végsők (*blank final*)
  - a konstruktorban inicializálni kell
- Metódus paramétere is lehet final
  - csak olvasható

Végső adatok

Példa

```
class FinalData {
    final int i = 9;
    static final int VAL = 99;          // csak egy van belőle
    final Value v2 = new Value();
    final int j = (int)(Math.random() * 20);      // futás
                                                  // közben!
    final int k;        // blank
    FinalData(int i) {
        k = i;          // a blank finalt a konstruktor(ok)ban
                        // inicializálni KELL!
    }
    void f(final int i) {
        i++;            // ez hiba: a final i nem változhat
    }
}
```

Végső metódusok

- Két dologra használjuk:
  - hogy meggátoljuk a származtatott osztályokat abban, hogy felüldefiniálják és megváltoztassák a viselkedését, és
  - hatékonyság végett.
- Minden `private` metódus impliciten `final` lesz:
  - nem lehet felüldefiniálni mert privát,
  - ha mégis megpróbáljuk, akkor új metódust kapunk.

Végső osztályok

- Osztályra is mondhatjuk hogy legyen végső: `final class ...`
- Azt jelenti, hogy belőle nem lehet új osztályokat származtatni
- Biztonsági vagy hatékonysági megfontolásból használjuk
- Minden metódusa is impliciten `final` lesz
  - nem lehet származtatni ezért felüldefiniálni sem
  - ugyanazok a hatékonysági megfontolások érvényesek lesznek mintha ki lenne írva

Tartalom

Osztálytag

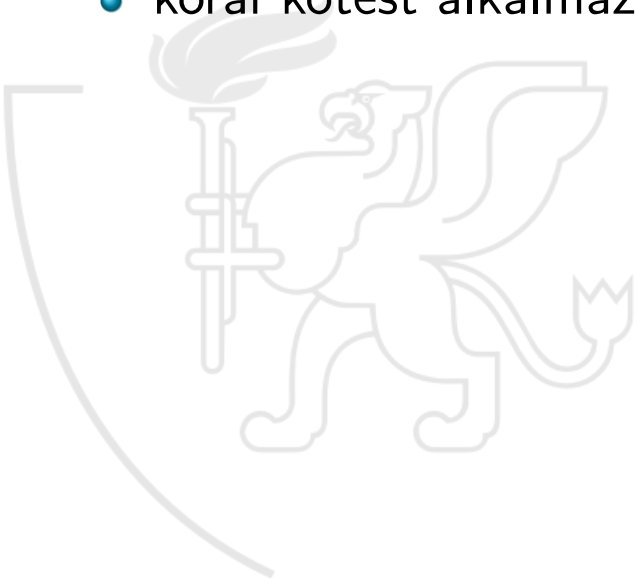
- Az **osztály**hoz tartozik
  - az osztály minden objektumára egyformán érvényes
- Módosítója a `static` kulcsszó
- Osztálytag lehet
  - osztályváltozó (statikus attribútum)
    - egy darab él belőle és statikus (közös állandó) memóriaterületen tárolódik
    - az egyes objektumok osztoznak rajta
  - osztálymetódus (statikus operáció)
    - egy példányban létezik,
    - nem egy konkrét objektumon dolgozik (inkább *függvény* mint *metódus*), így nem hivatkozhat a `this`-re, és
    - csak a többi osztálytagot látja

Példánytag

- Az **objektum**hoz tartozik
  - minden egyes objektumra különbözhet
- Minden tag, amelynek nincs `static` módosítója
- Példánytag lehet
  - példányváltozó
    - minden objektumban külön szerepel
    - értéke az objektum állapotára jellemző
  - példánymetódus
    - ez is csak egy példányban létezik, de
    - mindig egy adott objektumon (`this`) dolgozik, és
    - az osztály- és példánytagokat egyaránt látja

Osztálytag és példánytag

- Statikus (és csak statikus) metódus meghívható anélkül, hogy az osztályból objektumot hoznánk létre (pl.: a `main`)
- Statikus metódus csak az osztály statikus változóit és metódusait éri el
  - nem statikus metódusokat csak élő objektumokra lehet meghívni
- Statikus metódusokat nem lehet felüldefiniálni
  - nem működik rá a polimorfizmus
  - korai kötést alkalmaz a fordító a függvényhívásokra



Tartalom

Hivatkozási kör

- A metódus blokkjából hivatkozni lehet az osztály bármely tagjára
  - osztálymetódusból csak osztálytagra
- Egy adattag csak a fizikailag előtte deklarált adattagokra hivatkozhat
  - osztályváltozó csak osztályváltozóra
- Lokális változóra csak az őt deklaráló metóduson belül lehet hivatkozni
- Adattagnak és lokális változónak vagy metódus attribútumának lehet ugyanaz a neve
- A lokális változók és a metódusok attribútumai eltakarják az ugyanolyan nevű osztály- illetve példányváltozókat. Ilyenkor
  - az osztályváltozót az osztály nevével minősítve
  - a példányváltozót a `this` referenciával minősítve

  lehet elérni

Objektum referencia típusa

- Egy objektumra bármely saját- és ősosztály típusú referenciával hivatkozhatunk
  - fordítva: egy adott osztály típusú referenciával bármely vele azonos vagy származtatott osztályú objektumra hivatkozhatunk
- Az objektum referencia `statikus` típusa a deklaráláskor megadott osztály
- Az objektum referencia `dinamikus` típusa az adott pillanatban hivatkozott objektum konkrét osztálya

Referenciák típuskonverziói

- Csak öröklődési láncon belül lehet konvertálni
- Ősosztály típusú referencia mindig értékül kaphat leszármazott osztály típusú referenciát
  - leszármazottról ősre a konverzió implicit
  - biztonságos, mert amit a referencia alapján „ismer” az objektum, azt az objektum valóban „tudja” (örökölte)
- Leszármazott osztály típusú referencia típuskényszerítéssel értékül kaphat ősosztály típusú referenciát
  - ősről a leszármazottra a konverzió explicit kell legyen (soha nem automatikus)
  - veszélyes lehet, mert ahhoz képest, amit a referencia alapján „ismer” az objektum, mi többet „feltételezünk” róla
  - a rossz típusra való konverzió (ha az objektum mégsem az, amit hittünk) hibát (kivételt) eredményez