

Programozási nyelvek II.: JAVA

12. gyakorlat

2017. december 4-8.

- equals és hashCode metódusok
- Comparable
- Set, Map
- Absztrakt osztály
- Kivétel hierarchia, saját kivétel osztályok
- Ellenőrzött és nem ellenőrzött kivételek, throws

- A Java nyelv objektumok összehasonlítására három alapvető megoldást kínál számunkra:
 - **azonosság** vizsgálata az `==` operátorral. Azt nézzük, hogy két referencia ugyanarra a memóriaterületre mutat-e.
 - **egyenlőség** vizsgálata az `equals` metódussal. Azt nézzük, hogy a két objektum a felhasználásuk szempontjából egyformán viselkedik-e, például mikor két `String` ugyanazt a szöveget reprezentálja.
 - **természetes rendezés** a `Comparable` interfész megvalósításával. Ilyenkor lényegében a kisebb és nagyobb relációkat értelmezzük azokra a típusokra, ahol ennek van értelme.
- Az első kettő minden Java referenciatípusra alkalmazható, a harmadik csak ahol ezt lehetővé tesszük.

Egyenlőségvizsgálat az equals-zal

- Az *equals* metódust az *Object* osztály definiálja, így minden típus rendelkezik vele. (*Emlékeztető*: minden Java osztály őse az *Object*).
- Szignatúrája:

```
public boolean equals(Object obj)
```

- Az *Object*-ben definiált *equals* úgy viselkedik, mint az `==` operátor, azaz azonosságot vizsgál (egy objektumot csak önmagával tekint egyenlőnek). Ha ezt meg szeretnénk változtatni, akkor felül kell definiálnunk ezt a metódust.
- **Figyelem!** Az *equals* metódus paramétere *Object*, azaz két tetszőleges objektumot össze tud hasonlítani (bármilyen Java objektumon meghívhatjuk, és bármilyen Java objektumot kaphat paraméterül). Ezt a felüldefiniálásakor sem szűkíthetjük majd le.

A hashCode metódus

- A *hashCode* metódust szintén az *Object* osztály definiálja.
- Szignatúrája:

```
public int hashCode()
```

- Ez a metódus a hash alapú adatszerkezetek működését teszi lehetővé Java-ban. Egy egész számot rendel minden egyes objektumhoz, mely egyszerűvé teszi sok objektum összehasonlítását.
- A szabály ugyanis: ha két objektum az *equals* szerint egyenlő, akkor a *hashCode* függvényük garantáltan ugyanazt az egész számot adja vissza. (Megfordítva: ha két objektum különböző *hashCode* értéket ad, akkor garantáltan nem egyenlőek, vagyis nincs szükség az összehasonlításukra.)
- Ez azt jelenti számunkra, hogy amikor az *equals* metódust felüldefiniáljuk, a *hashCode*-ot is felül kell definiálni.

Az equals metódus felüldefiniálása – tulajdonságok

- Ha felül szeretnénk definiálni az *equals* metódust, az alábbi 5 tulajdonság teljesülését kell biztosítanunk. (Ezek az *equals* dokumentációjában mind szerepelnek, természetesen angolul.)
 - *Reflexív*: ha x és y nem *null*, és $x == y$, akkor $x.equals(y)$ is igazat kell, hogy adjon.
 - *Szimmetrikus*: ha x és y nem *null*, és $x.equals(y)$, akkor $y.equals(x)$.
 - *Tranzitív*: ha x , y és z nem *null*, valamint $x.equals(y)$ és $y.equals(z)$, akkor $x.equals(z)$.
 - *Konzisztens*: amíg nem módosítunk két objektumon, akkor mindig ugyanazt kell visszaadnia rájuk az *equals*-nak. (Ez gyakorlatilag azt jelenti, hogy nem használhatunk véletlengenerálást az egyenlőség eldöntésére, illetve más külső tényezőket sem vehetünk figyelembe, de nem is lenne értelme.)
 - Ha x nem *null*, akkor $x.equals(null)$ hamisat kell, hogy adjon.
- *Megjegyzés*: Az első három tulajdonság azt mondja ki, hogy az *equals* egy ekvivalenciareláció.
- A fentiek mellett fontos cél továbbá, hogy az *equals* minél gyorsabban lefusson, és ne szálljon el kivétellel.

Az equals metódus felüldefiniálása – sablon

- Mindezen szabályok betartása érdekében egy bevett sablon alapján szoktuk felüldefiniálni az *equals*-t.
- Először ellenőrizzük, hogy a kapott objektum azonos-e a *this*-szel, mert ilyenkor gyors választ tudunk adni a kérdésre. Következő lépésben pedig megvizsgáljuk, hogy nem *null*-t kaptunk-e, mikor is szintén gyors választ tudunk adni.
- Vegyük észre, hogy ez a két vizsgálat még teljesen független attól, hogy mely osztályhoz készül az *equals*.

```
@Override
public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (obj == null) {
        return false;
    }
}
```

...

- Harmadik lépés a típus ellenőrzése. Ehhez az *instanceof* operátort használjuk, mely egy objektumot és egy típust „hasonlít össze”. Ha az objektum dinamikus típusa azonos vagy leszármazottja a megadott típusnak, akkor igazat ad vissza, különben hamisat.
- Ezután castoljuk a paramétert egy megfelelő típusú objektummá, végül pedig sorra ellenőrizzük az adattagok egyenlőségét. A *null* értékekre persze az adattagoknál is figyelniük kell.
- Az alábbiakban egy egyszerű *Person* osztály *equals* metódusát írjuk felül, mely csak a személy nevét és születési évét tárolja.

Person

```
public class Person {
    private String name;
    private int birthYear;

    public Person(String name, int birthYear) {
        this.name = name;
        this.birthYear = birthYear;
    }

    public String getName() {
        return name;
    }

    public int getBirthYear() {
        return birthYear;
    }

    ... // equals és hashCode
}
```

Az equals metódus felüldefiniálása – sablon – 3.

```
@Override
public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if ( !(obj instanceof Person) ) {
        return false;
    }

    Person other = (Person) obj;

    ...
}
```

- *Megjegyzés:* mivel a *null instanceof T* bármilyen *T* típusra hamisat ad, ezért valójában a második *if* elhagyható; anélkül is hamisat kapunk a harmadik miatt *null* esetén.

Az equals metódus felüldefiniálása – sablon – 4.

```
...  
  
// Referenciatípusnál két null-t egyenlőnek kell venni,  
// null-t és nem null-t különbözőnek  
if (name == null) {  
    if (other.name != null) {  
        return false;  
    }  
} else if (!name.equals(other.name)) {  
    return false;  
}  
  
// Primitívnel egyszerű a dolgunk: nem lehet null  
if (birthYear != other.birthYear) {  
    return false;  
}  
return true; // Minden oké, visszatérhetünk igazzal  
}
```

Az equals metódus felüldefiniálása – sablon – 5.

- Egy rövidebb változat, egyben.

```
@Override
public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if ( !(obj instanceof Person) ) {
        return false;
    }
    Person other = (Person) obj;

    return birthYear == other.birthYear &&
        ( name == null ? other.name == null :
          name.equals(other.name) );
}
```

A hashCode metódus felüldefiniálása

- A *hashCode* esetén a legfontosabb követelmény, hogy két olyan objektum, melyek az *equals* szerint egyenlők, mindig ugyanazt az egész értéket adják vissza.
- Ez természetesen azt is jelenti, hogy konzisztensnek is kell lennie: egy objektum ugyanazt az értéket kell, hogy visszaadja, amíg nem változtatunk az állapotán.
- Cél ezúttal is a gyors kód, és hogy ne dobjunk kivételt.
- Nem feltétel, hogy két különböző objektum különböző értéket adjon vissza (nem is mindig lehetséges), de cél, hogy minél változatosabb számokat adjunk vissza, mert úgy válnak hatékonyrá a hash alapú adatszerkezetek.
- Például:

```
@Override
public int hashCode() {
    int hash = (name == null ? 0 : name.hashCode());
    return hash * 1009 + birthYear; // Az 1009 egy prímszám
}
```

A hashCode metódus felüldefiniálása – 2.

- A *hashCode* megírásakor gyakran használunk prímszorzókat, mert azok többnyire hatékony eredményt adnak. De lehetne szorozni, hatványozni, bármit, ami változatos értékeket ad.
- Egy egyszerű megoldás sok adattag esetén is a *java.util.Objects* osztály *hash* metódusa. Ennek a metódusnak bármennyi paramétert adhatunk, és ugyanazokat a paramétereket ugyanolyan sorrendben megkapva mindig ugyanazt a hash értéket állítja elő.

```
@Override
public int hashCode() {
    return java.util.Objects.hash(name, birthYear);
}
```

- Objektumok természetes rendezését (kisebb és nagyobb reláció) teszi lehetővé a *Comparable* interfész. Ha egy típusra definiálni szeretnénk ezeket a relációkat, meg kell valósítania ezt az interfészt.
- A *Comparable* egy generikus interfész, a típusparaméterének azt a típust kell átadnunk, amelyik épp megvalósítja az interfészt (ezzel fogunk összehasonlítani).

```
public class Person implements Comparable<Person> {  
  
    ...  
}
```

- A *Comparable* interfész egyetlen metódusa a *compareTo*, mely az összehasonlítást végzi. Paramétere a típusparaméternek megfelelő, tehát éppen az a típus, amelyiket összehasonlítjuk.

```
@Override
public int compareTo(Person other) {
    ...
}
```

- A visszatérési érték egy egész szám, mégpedig a következőképpen:
 - Ha a két objektum egyenlő, akkor 0-val térünk vissza.
 - Ha a *this* objektum kisebb, negatívval térünk vissza.
 - Ha a *this* objektum nagyobb, pozitívval térünk vissza.

- Ennél a metódusnál a paraméter sosem lehet *null* (pontosabban a dokumentációja szerint *null* paraméter esetén *NullPointerException*-t dobhatunk).
- A két összehasonlítandó objektum adattagjai elvileg lehetnek *null*-ok, de a példánkban feltehetjük, hogy a személy neve sosem *null*, mikor ezt a metódust meghívjuk.
- Először név szerint rendezünk (alfabetikusan, azaz ábécé szerint), másodsor pedig a születési év szerint. Természetesen lehetne másképp is.

compareTo – 3.

```
@Override
public int compareTo(Person other) {
    int compare = this.name.compareTo(other.name);
    if (compare != 0) {
        return compare;
    }
    // Ha már itt eldőlt, hogy valamelyik objektum kisebb,
    // térjünk vissza. Különben vizsgálódunk tovább.

    if (this.birthYear < other.birthYear) {
        compare = -1;
    } else if (this.birthYear > other.birthYear) {
        compare = 1;
    }
    return compare;
}
```

- Kicsit egyszerűbben:

```
@Override
public int compareTo(Person other) {
    int compare = this.name.compareTo(other.name);
    if (compare != 0) {
        return compare;
    }

    return this.birthYear - other.birthYear;
}
```

- Ne feledjük, hogy nem kötöttük ki, hogy kisebb esetén milyen negatív, nagyobb esetén milyen pozitív számmal térünk vissza. Általában -1-et és 1-et adunk vissza, de ez nem szabály, és **nem is feltételezhetjük, még egy általunk írt osztálynál sem.**
- Azaz ha meghívunk egy *compareTo* metódust, nem tehetjük fel, hogy ezeket a konkrét értékeket használja, mert ez nem dokumentált viselkedés, bárki bármikor megváltoztathatja. (Nem használunk ki rejtett invariánsokat.)

- Ha meg akarjuk jegyezni, hogy mikor kell negatívát és mikor pozitívát visszaadni, gondoljunk arra, hogy $x > y$ helyett $x.compareTo(y) > 0$ -t kell írunk.

```
x.compareTo(y) < 0 // x kisebb-e, mint y
x.compareTo(y) <= 0 // x kisebb vagy egyenlő-e, mint y
x.compareTo(y) == 0 // x egyenlő-e y-nal
x.compareTo(y) >= 0 // x nagyobb vagy egyenlő-e, mint y
x.compareTo(y) > 0 // x nagyobb-e, mint y
```

- Egy gyakran adatszerkezet, mely kihasználja az objektumok összehasonlítását, a *java.util.Set*, azaz halmaz típus. Fő jellemzője, hogy egyenlő objektumokból csak egyet tartalmazhat.
- Akárcsak a *List*, a *Set* is egy generikus interfész. Típusparamétere az elemei típusa, két leggyakoribb megvalósítása a *java.util.HashSet* és a *java.util.TreeSet*. Előbbi hash alapú (fontos a helyes *hashCode* metódus!), utóbbi pedig csak rendezhető típusokat tud kezelni, azaz meg kell valósítanunk a *Comparable* interfészt (van más mód is rendezés megadására, de ez nem a mostani gyakorlat anyaga).

```
Set<Person> set = new HashSet<Person>();  
set.add(new Person("Aladar", 2000));  
set.add(new Person("Aladar", 2000));  
// Egy egyenlő objektum már van bent, nem csinál semmit  
  
System.out.println(set.size()); // 1  
  
set.remove(new Person("Aladar", 2000));  
// A törlés is equals alapján történik  
  
System.out.println(set.size()); // 0
```

- Egy másik gyakori és hasznos adatszerkezet adatszerkezet a *java.util.Map*, más néven asszociatív tömb, mely kulcs-érték párokat tárol.
- Olyan, mintha egy tömbben az értékeket speciális kulcsokkal indexelnénk egész számok helyett.
- Ha a kulcs egész szám, akkor sem muszáj „sorban haladni”, lehet a kulcs negatív, és társíthatunk rögtön egy értéket az 1000-hez, akkor is, ha nincs kisebb kulcs.
- Szintén generikus interfész, két típusparamétere a kulcsok és értékek típusa.
- Két leggyakoribb megvalósítása a *Set*-hez hasonlóan a *java.util.HashMap* és a *java.util.TreeMap*, melyekre hasonló szabályok is igazak.
- Értelemszerűen két egyenlő kulcs sosem lehet egy *Map*-ben, hiszen indexek. Két azonos érték megengedett.

Map – példa

```
Map<String, Person> map =  
    new HashMap<String, Person>();  
  
Person aladar = new Person("Aladar", 2000);  
Person balazs = new Person("Balazs", 1999);  
  
map.put("a", aladar);  
map.put("b", aladar); // Oké  
  
System.out.println(map.size()); // 2  
  
map.put("b", balazs); // Felülírjuk  
  
System.out.println(aladar == map.get("a")); // true  
  
System.out.println(aladar == map.get("b")); // false  
System.out.println(balazs == map.get("b")); // true  
  
System.out.println(map.get("c")); // null  
// Nem létező indexre null-t ad vissza
```

- Eddig beszéltünk interfészekről és osztályokról. A kettő közt álló konstrukció az absztrakt osztály.
 - Absztrakt, tehát nem példányosítható.
 - Absztrakt, tehát lehet absztrakt metódusa.
 - Osztály, tehát csak egyszeresen örökíthető.
 - Osztály, tehát lehet adattagja, metódusa, konstruktora, tetszőleges láthatóságokkal, akárcsak eddig az osztályoknál megszokhattuk
- Absztrakt osztályt és absztrakt metódust az *abstract* kulcsszóval azonosítunk. Absztrakt metódusnak ezenfelül nincs törzse, a kapcsos zárójelek helyett egy pontosvesszőt teszünk.
- Absztrakt osztályból ugyanúgy származtathatunk, mint konkrét osztályból, csak mindenképp felül kell definiálnunk az absztrakt metódusait.

```
abstract class Rectangle {
    private double a;

    public Rectangle(double a) {
        this.a = a;
    }

    public double getA() { return a; }

    public abstract double getB();

    public double getArea() {
        return getA() * getB();
    }
}
```

- Absztrakt osztályban nincs automatikus publikus láthatóság a metódusoknak, mint interfészben! Ugyanúgy lehetnek bármilyen láthatóságúak.

Absztrakt osztály – példa – 2.

```
class GeneralRectangle extends Rectangle {
    private double b;

    public GeneralRectangle(double a, double b) {
        super(a);
        this.b = b;
    }

    @Override
    public double getB() { return b; }
}

class Square extends Rectangle {
    public Square(double a) {
        super(a);
    }

    @Override
    public double getB() { return getA(); }
}
```

- A Java kivételek valójában közönséges osztályok.
- A kivételhierarchiát az egymásból való származtatásuk adja. A hierarchia csúcsán a *Throwable* osztály áll, de ezzel nem foglalkozunk.
- Az *Exception* osztály és a belőle származó *RuntimeException* osztály az általunk használt kivételek ősei.
- Egy új (saját) kivétel létrehozásához egyszerűen származtatunk az *Exception* vagy a *RuntimeException* osztályból. Metodust, adattagot és konstruktort nem muszáj megadni.
- Gyakran használjuk a *String* paraméteres konstruktorukat, ez egy hibaüzenetet jelenít meg a konzolon, ha a kivétel a főprogramot is eléri, és nem kerül lekezelésre.

```
class UnnamedPersonException extends Exception {  
}  
  
class EmptyStackException extends RuntimeException {  
    EmptyStackException() {  
        super("The accessed stack is empty, " +  
            "no element can be retrieved.");  
    }  
}
```

- A kivételek egymásból is származtathatók. A *catch* ágak nem csak a megadott kivételeket kapják el, hanem azok leszármazottait is. Ugyanígy a *throws* ágban elég a szülőosztályt megadni, és dobható a leszármazottja is.

```
try {  
    ...  
} catch (Exception e) {  
    // bármilyen kivételt lekezel. Ha tehetjük, nem használjuk  
}
```

- A *RuntimeException*-ből származó kivételek úgynevezett nem ellenőrzött kivételek. (A nem *RuntimeException*-ből származó kivételeket pedig ellenőrzött kivételeknek nevezzük.)
- Abban speciálisak, hogy nem muszáj őket a *throws* részben feltüntetni, ha nem kezeljük le a metódusban (de feltüntethetjük).

```
public T getNextStackElement() {  
    if (list.isEmpty()) {  
        throw new EmptyStackException(); // Oké  
    }  
    return list.get(0);  
}
```

- Olyan kivételeket szoktunk *RuntimeException*-ként definiálni, melyek a kódban szinte bárhol előfordulhatnak, illetve amelyekről gyakran tudhatjuk a kód írásakor, hogy biztosan nem váltódnak ki, mert valamilyen korábbi ellenőrzés ezt biztosítja számunkra.
- Jól ismert példák ilyen kivételekre: *NullPointerException*, *ArrayIndexOutOfBoundsException*, *IllegalArgumentException*.

Feladatok



- Elérhető ezen a [LINKEN](#) egy Java forrásfájl, mely a következő feladatot volna hivatott elvégezni. Javítsd ki a benne található hibákat! Javasolt, hogy egy egyszerű *Main* osztályt írjunk, mely példányosítja a hibás osztályt, és kipróbálja a működését.
- Az osztály egy filmet ábrázol, tárolva annak címét, rendezője nevét és a megjelenése évét. Ellenőrzést nem végzünk.
- A konstruktor ezt a három paramétert várja a fenti sorrendben, és letárolja.
- Mindegyik adattaghoz legyen egy getter metódus.
- Írja felül az osztály a *hashCode* és *equals* metódusokat. Az *equals* ellenőrizze mindhárom adattagról, hogy azonosak-e a paraméterül kapott objektuméval, amennyiben az is egy *Movie*.
- Valósítsa meg az osztály a *Comparable* interfészt. Elsőre cím szerint, másodjára a megjelenési év szerint, harmadjára pedig a rendező neve szerint rendezzen. Ebben a metódusban feltehetjük, hogy az aktuális objektum és a paraméter egyik adattagja sem null.

- Készítsük el a *Circle* osztályt, mely tárolja egy kör középpontjának x és y koordinátáját és a sugarát (három lebegőpontos szám). A konstruktor fogadja ezeket a paramétereket, és tárolja le.
- Legyen mindegyikhez egy lekérdező getter művelet.
- Definiáljuk felül a kör osztály *hashCode* és *equals* metódusát. Az *equals* vizsgálja meg, hogy két kör egybevágó-e, azaz hogy a sugaruk egyenlő-e. A középpontot ne vegye figyelembe! (Figyelem! Ekkor a *hashCode* sem szabad, hogy függjön a középpont koordinátáitól, mert akkor két egyenlő objektumra adhatna különböző értéket.)
- Valósítsa meg a kör osztály a *Comparable* interfészt, két kör között a sugaruk alapján tegyen különbséget.

- Legyen egy *CheckedSet*<*T*> generikus osztályunk, mely egy halmazt valósít meg.
- Legyen egy *Set*<*T*> adattagja, melyet a nulla paraméteres konstruktor megfelelően feltölt.
- Lehessen lekérdezni a halmaz aktuális méretét.
- Legyen egy *add*(*T element*) metódusa, mely dobhat *AlreadyContainedException* ellenőrzött kivételt (származzon az *Exception*-ből). Ehhez hozzuk létre ezt a kivételosztályt, és dobjuk akkor, ha a halmaz már tartalmaz a megadottal egyenlő elemet. Különben tegyük bele a halmazba az új elemet.
- Legyen egy logikai visszatérési értékű *contains*(*T element*) metódusa, mely ellenőrzi, hogy a halmaz tartalmazza-e a megadott elemet.
- Egy főprogramban teszteljük le az új osztályt, rakjunk bele különböző sugarú köröket, és azonos sugarú, de más középpontú körrel is próbálkozzunk.

- Legyen egy $Bag<T>$ generikus osztályunk, mely egy zsákot valósít meg. A zsák olyan halmaz többször tartalmazhatja ugyanazt az elemet.
- Legyen egy $Map<T, Integer>$ adattagja, melyet a nulla paraméteres konstruktor megfelelően feltölt.
- Legyen egy $add(T\ element)$ metódusa. Ellenőrizze, hogy van-e már ilyen kulcs a zsákban, ha nincs, tegye bele 1 értékkel. Ha van, kérdezze le az aktuális értéket, és tegye bele az eggyel megnövelt értéket. (Azt tároljuk a map -ben, hogy melyik objektum hányszor van a zsákban.)
- Legyen egy egész visszatérési értékű $countOf(T\ element)$ metódusa, mely megadja, hogy hányszor van az elem a zsákban. Ha nincs az elemhez mint kulcshoz rendelve semmilyen érték a map -ben, adjon vissza 0-t.
- Legyen egy $remove$ metódusa is egy elem kivételére. Csökkentse eggyel a megadott elem darabszámát a zsákban. Ha 0-ra csökken a darabszám, vegye ki a megfelelő kulcs-érték párt a map -ből, hogy ne tároljunk feleslegesen adatokat. (Ehhez keressünk megfelelő metódust)

- Hozzuk létre a *Fraction* osztályt, mely törtszámokat ábrázol. Legyen két *int* adattagja, a számláló és a nevező.
- Legyen egy konstruktora, mely feltölti az adattagokat, és lehessen is őket lekérdezni getterekkel.
- Definiálja felül az *equals* és a *hashCode* metódusokat, valamint valósítsa meg a *Comparable* interfészt, mely a matematikai kisebb és nagyobb relációnak megfelelően működjön a törtszámokra.
- Lehessen két törtet összeadni, kivonni, összeszorozni. Figyeljünk a műveletek helyes elvégzésére.