Java XII. Egyenlőségvizsgálat

Az egyenlőségvizsgálat primitív típusok esetén triviális. Referencia típusoknál azonban már nem ilyen egyszerű a helyzet.

Time t1 = new Time(13,30);

Time t2 = new Time(13,30);

System.out.println( t1 == t2 );

Az eredmény hamis, mivel az == operátor azonosságot vizsgál, azaz csak azt nézi meg hogy a referenciák ugyan arra az objektumra hivatkoznak-e.

A tartalmi egyenlőségvizsgálatra biztosítja a Java az equals metódust, melyet minden objektum megörököl a java.lang.Object-ből.

System.out.println( t1.equals(t2) );

Ez továbbra is hamis, mivel az equals-t felül kell definiálnunk az általunk elvárt működéssel.

# Az equals metódus:

Fontos, hogy a különböző equals implementációkat úgy írjuk meg, hogy azok betartsák a következő szabályokat:

* legyen determinisztikus
* Ekvivalencia-reláció legyen (Reflexív, Szimmetrikus, Tranzitív)
  + Szimmetria egy kicsit lazábban értelmezett:
    - this == null esetén NullPointerException
    - that == null esetén false
* legyen konzisztens a hashCode() metódussal
  + egyenlő objektumok hashCode-ja egyezzen meg
  + (különböző objektumok hashCode-ja jó, ha különböző)

## Alapértelmezett viselkedés:

package java.lang;

public class Object {

...

public boolean equals( Object that ){

return this == that;

}

public int hashCode(){ ... }

}

## Szabályos felüldefiniálás:

public class Time {

...

@Override public boolean equals( Object that ){

if( that != null && getClass().equals(that.getClass()) ){

Time t = (Time)that;

return hour == t.hour && minute == t.minute;

} else return false;

}

@Override public int hashCode(){ return 60\*hour + minute; }

}

Először megvizsgáljuk, hogy a kapott objektum nem null. Ezt követően megvizsgáljuk a this és a that dinamikus típusát, mely, ha megegyezik már explicit átkonvertálhatjuk that-et a megfelelő típusra. Ezután elvégezhetjük a tartalmi összehasonlítást.

Az equals megírása után felül kell definiálnunk a hashCode metódust is.

Gyakran kiegészítjük az equals metódust a this==that szélsőséges eset lekezelésével a gyorsabb működés érdekében:

@Override

public boolean equals( Object that ){

if( this == that ) return true;

if( that != null && ...

Fontos szabály: az equals-t túlterhelni tilos.

class Object {

public boolean equals( Object that ){ ... } ...

}

class Time {

public boolean equals( Time that ){ ... } ...

}

Time t = new Time(11,22); Object o = new Time(11,22);

t.equals(t) t.equals(o) o.equals(t) o.equals(o)

Ez a kód például teljes őrület. Az első hívás a Time-beli definiált equals-t a második a Time-beli örökölt megvalósítást a harmadik és a negyedik az Object-beli implementációt hívja.

# Az öröklődés és az egyenlőségvizsgálat:

public class ExactTime extends Time { ...

@Override public boolean equals( Object that ){

return super.equals(that) && second == ((ExactTime)that).second;

}

@Override public int hashCode(){

return 60\*super.hashCode() + second;

}

}

Az alosztály egyenlőségvizsgálatához felhasználhatjuk a szülőosztály equals metódusát. Hasonlóan járhatunk el a hashCode felüldefiniálásánál is.

Megfigyelhetjük, hogy egy Time és egy ExactTime összehasonlítása mindig false értéket eredményez. A *Liskov helyettesítési elv* és az egyenlőségvizsgálat két összeegyeztethetetlen reláció (részbenrendezés és ekvivalenciareláció), ezért jogosan, de fel kell rúgnunk az LSP-t a helyes működés érdekében.

Próbáljuk meg kiküszöbölni ezt az összeférhetetlenséget:

public class Time {

@Override public boolean equals( Object that ){

if( that instanceof Time ){ ...

Time t = (Time)that;

return hour == t.hour && minute == t.minute;

...

public class ExactTime extends Time {

@Override public boolean equals( Object that ){

if( that instanceof ExactTime ){ ...

ExactTime t = (ExactTime)that;

return super.equals(that) && second == t.second;

...

Time t = new Time(11,22), e = new ExactTime(11,22,33);

t.equals(e) e.equals(t)

Ez az implementáció nem tartja be a szimmetriát. Próbáljunk meg egy másik megoldást.

public class ExactTime extends Time {

@Override public boolean equals( Object that ){

if( that instanceof ExactTime ){ ...

ExactTime t = (ExactTime)that;

return super.equals(that) && second == t.second;

} else if( that instanceof Time ){

return that.equals(this);

} else {

return false;

...

Time t = new Time(11,22);

ExactTime e1 = new ExactTime(11,22,33),

e2 = new ExactTime(11,22,44);

e1.equals(t) t.equals(e2) e1.equals(e2)

Ez az implementáció viszont a tranzitivitást sérti meg. Konklúzió: maradjunk az eredeti elképzelésünknél.

# A final módosítószó:

A példánymetódus nem definiálható felül.

public class Time {

...

@Override public final boolean equals( Object that ){

if( that instanceof Time ){

Time t = (Time)that;

return hour == t.hour && minute == t.minute;

} else return false;

}

}

Ha megpróbáljuk felüldefiniálni fordítási hibát kapunk.

Az osztályoknak is megadhatjuk a final módosítószót, ekkor:

* nem lehet az osztályból származtatni
* nem lehet specializálni, felüldefiniálni, belepiszkálni

Ilyen például a java.lang.String, a java.lang.Class, a csomagoló osztályok stb.

public final class Time {

...

@Override public boolean equals( Object that ){

if( that instanceof Time ){

Time t = (Time)that;

return hour == t.hour && minute == t.minute;

} else return false;

}

}

Ekkor, ha megpróbálunk egy új osztályt származtatni fordítási hibát kapunk.

## Öröklésre tervezés:

Ha szeretnénk, hogy az osztályunk származtatható legyen, akkor figyelnünk kell a következőkre:

* megfelelő equals megírása
* protected láthatóság
* legyen jól dokumentált az, hogy hogyan nézzen ki egy származtatott osztály
* legyen időtálló

Ha nem akarjuk, hogy származtassanak az osztályból, legyen final.

Az öröklődés kiváltása:

Az öröklődés kiváltható kompozícióval:

public class ExactTime {

private final Time time;

private int second;

public ExactTime( int hour, int minute, int second ){

time = new Time(hour,minute);

if( 0 <= second && second < 60) this.second = second;

else throw new IllegalArgumentException();

}

public int getSecond(){ return second; }

public int getMinute(){ return time.getMinute(); }

public void aMinutePassed(){ time.aMinutePassed(); }

...

}

Bár ehhez elég sok *Boilerplate*-kód szükséges.

# Heterogén egyenlőség:

Néha előfordul, hogy el szeretnénk tekinteni mondjuk egy adatszerkezet konkrét megvalósításától, ekkor van értelme heterogén egyenlőséget definiálni.

ArrayList<Integer> aList = new ArrayList<>();

LinkedList<Integer> lList = new LinkedList<>();

aList.add(19);

lList.add(20-1);

aList.equals(lList)

# Adatszerkezetek:

Az ArrayList, a HashSet és a HashMap adatszerkezetek az equals és a hashCode metódusok helyes működésén alapszanak.

# Mutálható objektum az adatszerkezetben:

Mi történik, ha megváltoztatunk egy hasító adatszerkezetben lévő objektumot?

Time t = new Time(5,30);

HashSet<Time> set = new HashSet<>();

set.add(t); set.add(t); System.out.println(set); // [5:30]

set.remove(new Time(5,30)); System.out.println(set); // []

set.add(t);

t.setHour(6);

set.remove(new Time(5,30));

System.out.println(set); // [6:30]

set.remove(new Time(6,30));

System.out.println(set); // [6:30]

A program az 5:30 hashCode-jához tartozó rés láncolt listájában megpróbálja megtalálni az 5:30-at de, mivel mi azt megváltoztattuk az ekkor meghívásra kerülő equals metódus hamis értéket fog adni.

A 6:30-hoz tartozó rés láncolt listája üres ezért nyilvánvalóan sikertelen lesz a törlés.

Ha egy equals/hashCode alapú adatszerkezetben módosítani akarunk egy elemet akkor azt először ki kell vennünk az adatszerkezetből, majd a módosított értéket vissza kell raknunk.

# A Stringek egyenlőségvizsgálata:

A String-ek nagyon érdekes módon tárolódnak. Minden String literál csak egyszer jön létre a heap-en, azaz

String verb = "ring";

String noun = "ring";

valójában ugyan az az objektum. Ezért:

verb.equals(noun) // true

verb == noun // true

Ha viszont new-al allokálunk egy String-et, akkor az már egy új objektum lesz:

String mathematical = new String("ring");

noun.equals(mathematical) // true

noun == mathematical // false

Tanulság: mindig equals-t használjunk.

# Comagolóosztályok

A String-ekéhez nagyon hasonló módon működik a csomagolóosztályok egyenlőségvizsgálata:

Integer nineteen = 19;

Integer twentyButOne = 20-1;

nineteen.equals(twentyButOne) // true

nineteen == twentyButOne // true

Integer dog = -123456;

Integer pup = -123456;

dog.equals(pup) // true

dog == pup // false

A különbség itt csak annyi, hogy csak egy bizonyos intervallumra vonatkozik az, hogy közös objektum tartozik a literálokhoz.

Mindig equals-t használjunk.

# Felsorolási típusok:

A felsorolási típusoknál nem szükséges equals-t használni, minden JVM garantálja az == operátor megfelelő működését.

# Tömbök:

A tömbök egyenlőségvizsgálata nem része az alap nyelvi csomagnak. Vagy nekünk kell definiálni vagy java.Arrays.equals(x,y) metódusát használjuk. Tömbök tömbjére viszont ez már nem működik, de szerencsére rendelkezésünkre áll a java.util.Arrays.deepEquals(x,y).

# Mély vizsgálat:

Ha egy adattag referencia típusú akkor az equals metódusban kötelesek vagyunk az adattag equals metódusát hívni.

public class Interval {

private Time from, to;

...

@Override public boolean equals( Object that ){

if( that != null && getClass().equals(that.getClass()) ){

Interval u = (Interval)that;

return from.equals( u.from ) && to.equals( u.to );

} else return false;

}

...

}