Programozási nyelvek II.: JAVA

13. gyakorlat

2017. december 11-15.

Az 13. gyakorlat tematikája

- Generikus típusokés öröklődés
- Clone
- Comparator
- Névtelen osztály, és lambda kifejezés

Generikus típus - ismétlés

- "típussal paraméterezett típus"
- példa egyszerű csomagoló osztály generikusok nélkül:

```
public class Box {
        private Object object;

        public void set(Object object)
        {
             this.object = object;
        }
        public Object get() {
            return object;
        }
}
```

Generikus típus - ismétlés

- Problémák:
- akármilyen nem primitív típusú objektumot tárolhat -> ha vissza akarjuk kapni az eredeti típusos objektumunkat castolni kell
- ha tévedésből más típusú objektumot adunk oda neki a castolás futásidejű kivételt eredményez, amit aztán nehéz lenyomozni

Generikus típus - ismétlés

- Megoldás:
- fordítási idejű ellenőrzés - generikusok

```
public class Box<T> {
        // T stands for "Type"
        private T t;

        public void set(T t) { this.t = t; }
    public T get() { return t; }
}
//használat:
 Box<Integer> integerBox = new Box<>(); ‘
```

Generikus metódusok

- hasonlóan az osztályokhoz, a generikus paraméter hatása csak metódusra terjed ki. Lehetnek generikus metódusok static és nem static függvények vagy konstruktorok is.
- Szintaxis: a visszatérédi érték előtt '<...>'-ben jelölni a típusparamétereket
- Példa

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1,
                                         Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}
```

Megszorítások típusparaméterekre

- megkötések generikus osztályunk által kezelt típusokra
- például: szeretnénk egy adott tömbben egy küszöbértéknél "nagyobb elemelet megszámlálni:

```
public interface Comparable<T> {
    public int compareTo(T o);
}

public static <T extends Comparable<T>>
            int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

- Egyszerre több megkötés is alkalmazható: <T extends B1 & B2 & B3> , ekkor a paraméternek az összes paraméter altípusának kell lennie egyszerre.

Generics - praktikus szabályok

- típusmegszorításnál extends
- leszármaztatás megszorított típusparaméterű generikus típusból:

```
pinterface Edge {
    int getSource();
    int getTarget();
}
abstract class Graph<E extends Edge> {...}

class SparseGraph<E extends Edge> extends Graph<E> {...}
```

Beágyazott osztályok

- közelebb vinni a segédosztályt a használat helyéhez
- azt akarjuk hogy az egyik (beágyazott) hozzávérhessen a másik private-nak szánt adattagjaihoz
- osztályok bármilyen blokkon belül definiálhatóak

Névtelen osztályok

- osztály deklarálása és példányosítása egyidőben
- úgy néz ki a mintha interfészt példányosítanánk

Névtelen osztályok

```
public class HelloWorldAnonymousClasses {

    interface HelloWorld {
        public void greet();
        public void greetSomeone(String someone);
    }
  //...
```

Névtelen osztályok

- beágyazott osztály

```
public void sayHello() {

    class EnglishGreeting implements HelloWorld {
        String name = "world";
        public void greet() {
            greetSomeone("world");
        }
        public void greetSomeone(String someone) {
            name = someone;
            System.out.println("Hello␣" + name);
        }
    }

    HelloWorld englishGreeting = new EnglishGreeting();
    englishGreeting.greet();
    //...
```

Névtelen osztályok

- névtelen osztály

```
//...
        HelloWorld frenchGreeting = new HelloWorld() {
            String name = "tout le monde";
            public void greet() {
                greetSomeone("tout le monde");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Salut " + name);
            }
        };
        frenchGreeting.greetSomeone("Fred");
    }
}
```

Lamba kifejezések

- Ha a névtelen osztályunk valami nagyon egyszerű dolgot implementál, például ha mindössze egy metódussal rendelkezik, és a gyakorlatban ezem metódus variálgatására szolgálnak a definiált osztályok, akkor a fentebb mutatott viszonylag tömör típusleírás is feleslegesnek tűnhet.
- tulajdonképpen metódusokat szeretnénk átadni egy másik eljárás paramétereként
- A Lambda kifejezések lényege, hogy lehetővé teszik tehát metódusok paraméterként történő átadását illetve a kód adatként való kezelését.
- Példa: szeretnénk minél szélesebb körben használható szűrési metódust írni amely Person típusú objektumok valamely tulajdonságát vizsgálja, majd a teszten átment objektumokat kiírja.

Lamba kifejezések - metódus átadása

1. Tester osztállyal/lokális osztállyal

```
class CheckWithInnerClass{
        interface CheckPerson { boolean test(Person p );}
        class Checker implements CheckPerson {
            public boolean test(Person p) {
             return p.gender == Person.Sex.MALE
             && p.getAge() >= 18 &&  p.getAge() <= 25;
                }
        }
      public static void printPersons(
      List<Person> roster, CheckPerson tester) {
        for (Person p : roster) {
         if (tester.test(p))
            p.printPerson();
      }
      public static void main(String[] args){
          printPersons(roster, new Checker());
        }
}
```

Lamba kifejezések - metódus átadása

2. Névtelen osztállyal

```
    //...
printPersons(    roster,    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    } );
```

3. Lambda kifejezéssel

```
    //...
printPersons(    roster,    (Person p) ->
 p.getGender() ==    Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25 );
```

Lamba kifejezések - szintaktika

- (<inputTípus1 input1>,...,<inputTípus1 input1>) a megvalósított interfész egyetlen metódusának paraméterei
- a típusnevek lehagyhatók
- zárójel is lehagyható, amennyiben csak egy paraméter lesz (‘(Person p)‘helyett ‘p‘)
- ->‘a nyíl token
- a metódus törzse, melynek vagy egyetlen kifejezésből kell állnia, vagy egy blokknak kell lennie (‘...‘)

Lamba kifejezések - példák

```
p -> p.getGender() == Person.Sex.MALE
&& p.getAge() >= 18
&& p.getAge() <= 25
```

ha blokkot használunk szükség van a ‘return‘használatára

```
p -> {
return p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25;
}
```

Lamba kifejezések - példa

Még egy példa:

```
public class Calculator {
interface IntegerMath {
    int operation(int a, int b);
}
public int operateBinary(int a, int b, IntegerMath op) {
    return op.operation(a, b);
}
public static void main(String... args) {
    Calculator myApp = new Calculator();
    IntegerMath addition = (a, b) -> a + b;
    IntegerMath subtraction = (a, b) -> a - b;
    System.out.println("40␣+␣2␣=␣" +
        myApp.operateBinary(40, 2, addition));
    System.out.println("20␣-␣10␣=␣" +
        myApp.operateBinary(20, 10, subtraction));
}
}
```

Lamba kifejezések -Comparator

- Listák rendezése: java.util.Collections.sort statiku smetódussal:
- 1. Comparable<T> -t megvalósító T osztályokat tároló listák:
- sort(List<T> list)

```
class Person implements Comparable<Person>{...}
//...
List<Person>  l = new ArrayList<>();
//...
Collections.sort(l);
//..
```

Lamba kifejezések -Comparator

- 2. Listákra általában:
- sort(List<T> list, Comparator<? super T> c) - ahol a Comparator egy úgynevezett funkcionális interfész, azaz csak egy meghatározott metódust meglétét deklarálja:

```
int compare(T o1,T o2)
```

- metódus megadása: névtelen osztállyal:

```
class Car{...}
//...
List<Car>  l = new ArrayList<>();
//...
Comparator<Car> byName = new Comparator<Car>() {
    @Override
    public int compare(Car c1, Car c2) {
        return c1.getName().compareTo(c2.getName());
    }
};
Collections.sort(l, byName);
//..
```

Lamba kifejezések -Comparator

- 2. Listákra általában:
- sort(List<T> list, Comparator<? super T> c) - ahol a Comparator egy úgynevezett funkcionális interfész, azaz csak egy meghatározott metódust meglétét deklarálja:

```
int compare(T o1,T o2)
```

- metódus megadása: lambda kifejezéssel:

```
class Car{...}
//...
List<Car>  l = new ArrayList<>();
//...

Collections.sort(l,
(c1,c2)->c1.getName().compareTo(c2.getName())
);
//..
```

Clone

- Objektumok duplikálására szolgál
- azaz új objektumot hozunk létre, amelynek összes adattagja megegyezik az eredeti értékeivel.
- követelmények:

```
i) mc.clone() != mc

ii) mc.getClass() == mc.clone().getClass()

iii) mc.clone().equals(mc)
```

- Hogyan:
  - clone() metódus megvalósítása
  - Cloneable interface implementáló osztály - DE ez egy marker/jelölő interface -> nem kényszeríti ki a clone() implementálását, mindazonáltal jelzi a compiuler és a JVM felé, hogy az osztály clone() metódusát meg lehet hívni
  - CloneNotSupportedException : clone() metódust írhatunk bármilyen osztályhoz, azonban ha az osztály amit klónozni akarunk nincs

Clone

- Hogy is néz ez ki?

```
public class CloneTrial implements Cloneable {
    int i;
    XYZ xyz = new XYZ();
    public CloneTrial clone() throws CloneNotSupportedExcep
        CloneTrial ct =  (CloneTrial)super.clone();
        return ct;
    }
    public static void main(String[] x) throws CloneNotSupp
        CloneTrial ct = new CloneTrial();
        ct = ct.clone();
    }
}
class XYZ{
    int x,y,z;
}
```

Sekély másolás/shallow copy

- Mi történik mikor klónozunk?
- "Bitenkénti másolás", azaz minden mező lemásolódik ugyanazzal az értékkel, ami az eredetiben található.

```
//..
public static void main(String[] x) throws CloneNotSupporte
        CloneTrial ct = new CloneTrial();
        CloneTrial ct1 = ct.clone();
        ct.xyz.x = 1;
        System.out.println(ct1.xyz.x);
    }
    // OUTPUT : 1
```

- Következmény: a referenciák is bitenként másolódnak, azaz ugyanarra az objektumra fognak mutatni.
- Ezt hívjuk sekély másolásnak

Mély másolás/deep copy

- Azt akarjuk,hogy a logikailag az objektumhoz tartozó másik objektumok állapota védve legyen, azaz a beágyazott objektumok is lemásolódjanak.
- Ehhez osztályonként definiálnunk kell a metódust, mit akarunk érték és mit referencia szerint átadni.

```
//..
public static void main(String[] x)
    throws CloneNotSupportedException {
        CloneTrial ct = new CloneTrial();
        CloneTrial ct1 = ct.clone();
        ct.xyz.x = 1;
        System.out.println(ct1.xyz.x);    // OUTPUT : 1
    }
```

- Következmény: a referenciák is bitenként másolódnak, azaz ugyanarra az objektumra fognak mutatni.
- Ezt hívjuk sekély másolásnak

Mély másolás/deep copy

- Az előző program mély másolással : XYZ típusú objektum is lemásolva
- Ehhez osztályonként definiálnunk kell a metódust, mit akarunk érték és mit referencia szerint átadni.
- 1. próba

```
public class CloneTrial implements Cloneable{
    int i;
    XYZ xyz = new XYZ();
    public CloneTrial clone()
        throws CloneNotSupportedException {
        CloneTrial ct =  (CloneTrial)super.clone();
         ct.xyz = (XYZ)ct.xyz.clone();
        return ct;
    }
}
class XYZ {    int x,y,z; }
```

- fordítási hiba: a clone() protectednek van deklarálva az Objectben(XYZ)
- definiáljuk felül!

Mély másolás/deep copy

- 2. próba - clone felüldefiniálása

```
//...
class XYZ {
    int x,y,z;
    public XYZ clone() throws CloneNotSupportedException {
        return (XYZ)super.clone();
    }
}
```

- CloneNotSupportedException : nincs Cloneable-nek definiálva XYZ

Mély másolás/deep copy

- 3. próba - Cloneable interfész

```
//...
class XYZ implements Cloneable{
    int x,y,z;
    public XYZ clone() throws CloneNotSupportedException {
        return (XYZ)super.clone();
    }
}
```

- Most már rendben van: ouput :0

Feladat - hibajavítás - clone()

javítsuk ki a következő hibás programot!
bathroom.zip

- javítsuk ki a fordítási hibákat!
- implementáljunk a mély másolást a BathRoom.java clone() metódusának kijavításával, azaz érjük el, hogy a klónozás során a BathTub objektum is lemásolásra kerüljön!

Feladat 1

Induljuk ki a következő programból:
Ember.java BeolvasEmber.java input.txt

- a).Rendezzük az emberek listáját név szerint.
  - a rendezéshez használd a Collections.sort metódust
  - az ‘Ember‘osztály valósítsa meg a Comparable<Ember>‘interfészt
- b). Rendezzük az emberek listáját azonosító szerint (de úgy, hogy név szerint továbbra is rendezhető maradjon)
  - I. megoldás: Comparator osztály és a Comparator-paraméteres Collections.sort metódus használatával
  - II. megoldás: ugyanez névtelen Comparator‘osztállyal
  - II. megoldás: ugyanez lambda függvénnyel

Feladat 2

Gyakoroljuk a lambda függvényeket! Készítsünk egy List objektumot, amit töltsünk fel 10 véletlenszerű számmal. Lambda függvények segítségével végezzük el a listán a következő feladatokat:

- a). list.forEach segítségével:
  Írjuk ki a lista elemeit Tegyük át a lista elemeit egy halmazba
  Végezzük el mindkét feladatot egyszerre
- b). list.removeIf segítségével: távolítsuk el a listából a 10-nél nagyobb számokat.
- c). list.replaceAll segítségével: emeljük négyzetre a számokat.