Are you sure you want to submit the exam?

No, continue writing | Yes, submit my exam!

My answers:

1

Napišite klasu `CollectionUtil` koja ima samo generičku statičku metodu `myFilterReduce(Collection<T> coll, T init, Predicate<T> pred, BinaryOperator<T> op)` koja prima kolekciju objekata `Collection<T>`, jednu početnu vrijednost `init` tipa `T`, a vraća agregatni rezultat operacije `BinaryOperator<T>` između `init` i svih objekata koji zadovoljavaju predikat `Predicate<T>`.

Na primjer poziv

```
myFilterReduce(Arrays.asList(2, 3, 6, 3),
               5,
               x->x%2==0,
```

My answer:

```
import java.util.Collection;
import java.util.function.BinaryOperator;
import java.util.function.Predicate;

class CollectionUtil {
    public static <T> T myFilterReduce(Collection<T> coll, T init, Predicate<T> pred, BinaryOperator<T> op) {
        for (T element : coll) {
            if (pred.test(element)) {
                init = op.apply(init, element);
            }
        }
        return init;
    }
}
```

2

Neka su zadane apstraktne klase:

```
abstract class Game {
    private String name;
    private int rating;

    protected Game(String name) {
        this.name = name;
    }
```

My answer:

```
import java.util.*;

class Factory extends Game{
    protected Factory(String name, int rating) {
        super(name);
        super.setRating(rating);
    }

    public static Game createGame(String name, int rating) {
        return new Factory(name, rating);
    }
}

class MyPlayer extends Player implements Iterable<Game>{
    Map<Game, List<Integer>> allScores = new TreeMap<>(Comparator.comparing(Game::getName));

    protected MyPlayer(String name) {
        super(name);
    }

    @Override
    void addGameScore(Game game, int score) {
        allScores.putIfAbsent(game, new LinkedList<>());
        List<Integer> scores = allScores.get(game);
        scores.add(score);
    }

    @Override
    Iterable<Integer> getScores(Game game) {
        return allScores.get(game);
    }

    @Override
    public Iterator<Game> iterator() {
        return allScores.keySet().iterator();
    }
}
```

3

Potrebno je napisati klasu `FibonacciIterator` koja implementirata sučelje `java.util.Iterator<Integer>`. Klasi se u konstruktoru prosljeđuje koliko brojeva će generirati. Ako se pošalje negativan broj potrebno je baciti `IllegalArgumentException`.

Klasa ima vidljivost postavljenu na *package private*.

Primjer korištenja:

```
Iterator<Integer> iterator = new FibonacciIterator(10);
while(iterator.hasNext())
```

My answer:

```
import java.util.NoSuchElementException;

class FibonacciIterator implements Iterator<Integer>{
    private int toGenerate;
    private int num1 = 0, num2 = 1, num3 = 0;
    private int count = 0;

    public FibonacciIterator(int toGenerate) {
        if (toGenerate < 0) {
            throw new IllegalArgumentException();
        }
        this.toGenerate = toGenerate;
    }

    @Override
    public boolean hasNext() {
        return count < toGenerate;
    }

    @Override
    public Integer next() {
        if(!hasNext()) {
            throw new NoSuchElementException();
        }
        if(count == 0) {
            count++;
            return 0;
        }
        num3 = num1 + num2;
        num1 = num2;
        num2 = num3;
        count++;
        return num1;
    }
}
```

4

U ovom zadatku se koristi zapis decimalnih brojeva na "distribuirani" način. Njihove znamenke su zapisane u listi sa oznakama pozicije. Detaljnije:

- Lista se sastoji od **podlista** koje sadrže po dva elementa (**a**,**b**).
- pritom **a** označava redni broj decimalnog mjesta (pozicije) na sljedeći način: 0.1234567. 0. je zadnje mjesto ispred decimalne točke, a potom ostali redni brojevi označavaju pozicije na desno od decimalne točke. Pozicije lijevo od 0. se ignoriraju.
- pritom **b** je znamenka na mjestu označenom sa pripadnim **a**.

Primjer: broj 3.141 zapisan na distribuirani način jest: ((0,3), (1,1), (2,4), (3,1))

My answer:

```
import java.util.*;
import java.util.function.Predicate;

class Solution{

    public static Predicate<List<List<Integer>>> allDigitsMatch(double exemplar){
        String number = String.valueOf(exemplar);
        char[] charDigits = number.toCharArray();
        List<Integer> digits = new ArrayList<>();
        for (char c : charDigits) {
            if(c != '.') {
                digits.add(Integer.parseInt(String.valueOf(c)));
            }
        }
        return lists -> {
            for (List<Integer> list : lists) {
                if(!digits.contains(list.get(1))) {
                    return false;
                }
            }
            return true;
        };
    }


    public static Predicate<List<List<Integer>>> allDigitsDefined(){
        return lists -> {
            Set<Integer> indexes = new TreeSet<>();
            for (List<Integer> list : lists) {
                indexes.add(list.get(0));
            }
            for (int i = 0; i < indexes.size(); i++) {
                if(!indexes.contains(i)) {
                    return false;
                }
            }
            return true;
        };
    }


    // this is from allDigitsMatch and it does... something...
    //        String number = String.valueOf(exemplar);
    //        char[] charDigits = number.toCharArray();
    //        List<Integer> digits = new ArrayList<>();
    //        for (char c : charDigits) {
    //            if(c != '.') {
    //                digits.add(Integer.parseInt(String.valueOf(c)));
    //            }
    //        }
    //        List<List<Integer>> distributed = new ArrayList<>();
    //        for (int i = 0; i < digits.size(); i++) {
    //            List<Integer> onePosition = List.of(i, digits.get(i));
    //            distributed.add(onePosition);
    //        }
    //        return lists -> {
    //            lists.sort(Comparator.comparing(o -> o.get(0)));
    //            for (int i = 0; i < lists.size(); i++) {
    //                if(!lists.get(i).equals(distributed.get(i))) {
    //                    return false;
    //                }
    //            }
    //            return true;
    //        };
}
```

5

Pretpostavite da korisnik vašeg programa preko tipkovnice unosi podatke o studentima i ostvarenim bodovima na ispitu iz OOP-a, i to u sljedećem formatu:

```
Ime1#bodovi
Ime2#bodovi
```

Vaš zadatak je da dovršite klase Zapis i LabTask

```
class Zapis {
```

My answer:

```
import java.util.*;

class LabTask {
    public static Collection<Zapis> readData() {
        // Dovršiti
        Collection<Zapis> col = new TreeSet<>();
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String line = sc.nextLine();
            if(line.equals("quit")) {
                break;
            }
            String[] data = line.split("#");
            Zapis entry = new Zapis(data[0], Integer.parseInt(data[1]));
            col.removeIf(element -> element.equals(entry));
            col.add(entry);
        }
        return col;
    }
}

class Zapis implements Comparable<Zapis>{
    String ime;
    Integer bodovi;

    public String toString() {
        return this.ime + ":" + this.bodovi;
    }

    // Dovršiti
    public Zapis(String ime, Integer bodovi) {
        this.ime = ime;
        this.bodovi = bodovi;
    }
    @Override
    public int compareTo(Zapis o) {
        return this.ime.compareTo(o.ime);
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        if(!(obj instanceof Zapis other)) return false;
        return this.ime.equals(other.ime);
    }
}
```

6

Potrebno je napisati klasu `FactorialIterator` koja implementirata sučelje `java.util.Iterator<Integer>`. Klasi se u konstruktoru prosljeđuje koliko brojeva će generirati. Ako se pošalje negativan broj potrebno je baciti iznimku.

Klasa ima vidljivost postavljenu na *package private*.

Primjer korištenja:

```
Iterator<Integer> iterator = new FactorialIterator(10);
while(iterator.hasNext())
```

My answer:

```
import java.util.Iterator;
import java.util.NoSuchElementException;

class FactorialIterator implements Iterator<Integer> {

    private int toGenerate;
    private int generated = 0;
    private int last = 1, next;

    public FactorialIterator(int toGenerate) {
        if (toGenerate < 0) {
            throw new IllegalArgumentException();
        }
        this.toGenerate = toGenerate;
    }

    @Override
    public boolean hasNext() {
        return generated < toGenerate;
    }

    @Override
    public Integer next() {
        if(hasNext()) {
            generated++;
            int temp = last;
            next = last * generated;
            last = next;
            return temp;
        }
        else{
            throw new NoSuchElementException();
        }
    }
}
```