7th homework assignment; JAVA, Academic year 2018/2019; FER

Napravite prazan Maven projekt, kao u 1. zadaći: u Eclipsovom workspace direktoriju napravite direktorij `hw07-0000000000` (zamijenite nule Vašim JMBAG-om) te u njemu oformite Mavenov projekt `hr.fer.zemris.java.jmbag0000000000:hw07-0000000000` (zamijenite nule Vašim JMBAG-om) i dodajte ovisnost prema `junit`. Importajte projekt u Eclipse. Sada možete nastaviti s rješavanjem zadataka.

Problem 1.

When writing non-trivial programs, you are often confronted with the following situation: there is an object that holds some data (let's call it the *object state*) and each time that data changes, you would like to execute a certain action or a set of actions (to process the data, to inform user about the change, to update the GUI, etc). For example, let us consider an object that holds current room temperature (some kind of sensor): each time the temperature changes, you would like to execute one or more actions (update the temperature displayed in some graphical component, write new temperature with timestamp into a log file, etc). Another example: you have an object which monitors some directory in the file system. Each time the user copies some postscript (.ps) file into this directory, you would like to automatically start a program which will convert it into the PDF-form; additionally, you would like to make a backup copy of the original postscript file; you would like to send an e-mail informing some user that a new PS file is available; etc.

If you have a full control of the object, and if there is always the same (single) action, this could easily be coded into the object itself. However, often you will develop the object separately from actions which process the object state. Sometimes, you will even work with objects which are not under your control. For example, the maker of the temperature sensor will make accompanying library for reading sensor data; the maker at the moment of writing library code does not know how the sensor will be used, or what actions should be executed when the temperature changes. So the general idea is to come up with the code organization which would allow you to write (or use) the object which encapsulates the state, and then to develop new actions without ever changing or recompiling the original object code itself.

The *Observer pattern* is a design pattern which can be used in previously described situation. The basic idea is this: your object (denoted as the *Subject* in this design pattern; please remember the terminology here) does not need to know anything about the concrete actions (the *Concrete Observers* in this design pattern) you will develop. However, it will mandate that in order to be able to invoke your action/actions, you must satisfy the following conditions:

1. the object (the *Subject*) will have to be able to "talk" with your actions (*Concrete Observers*) in a way that it expects – this means that the object will prescribe a certain interface your actions will have to implement; this interface is usually called the *Observer* interface (or *Abstract Observer*); the interface will define a method which should be called when the state has changed;
2. the object (the *Subject*) will provide a method that will allow the user to register the developed actions (*Concrete Observers*); these actions implement the *Observer* interface; the object will be responsible for tracking the internal collection of registered observers;
3. the object (the *Subject*) can provide a method for action removal so that you can at any time unregister previously registered actions;
4. every time the objects' state changes, the object will invoke the method prescribed by the *Observer* interface on all of the registered actions.

Lets us consider a concrete example: the temperature sensor library. The maker of the USB attachable sensor will provide a library with a class `TemperatureSensor`. This will be the *Subject*. For modeling actions, the library will define an interface `TemperatureObserver` (the *Observer* interface in Observer design pattern terminology):

```
public interface TemperatureObserver {
  void temperatureChanged(double currentTemperature);
}
```

The *Subject* will implement methods for registering and unregistering *Observers*:

```
public class TemperatureSensor {
  ...
  public void register(TemperatureObserver observer) { … }
  public void unregister(TemperatureObserver observer) { … }
  ...
  private void notifyObservers() { … }
  ...
}
```

Typically, the *Subject* will also have a private method `notifyObservers()` which will be called internally from the *Subject* when the temperature changes; it will iterate through the collection of registered `TemperatureObserver`s and will call `temperatureChanged(...)` with the current temperature on each one.

Your job in this example would be to define different objects (actions) which process the temperature change events. For example, you might want to write a message to the user each time the temperature changes, and you might want to write the new temperature with timestamp into the log file, so you could later create a fancy temperature-change diagram. In order to to this, you would write two new classes:

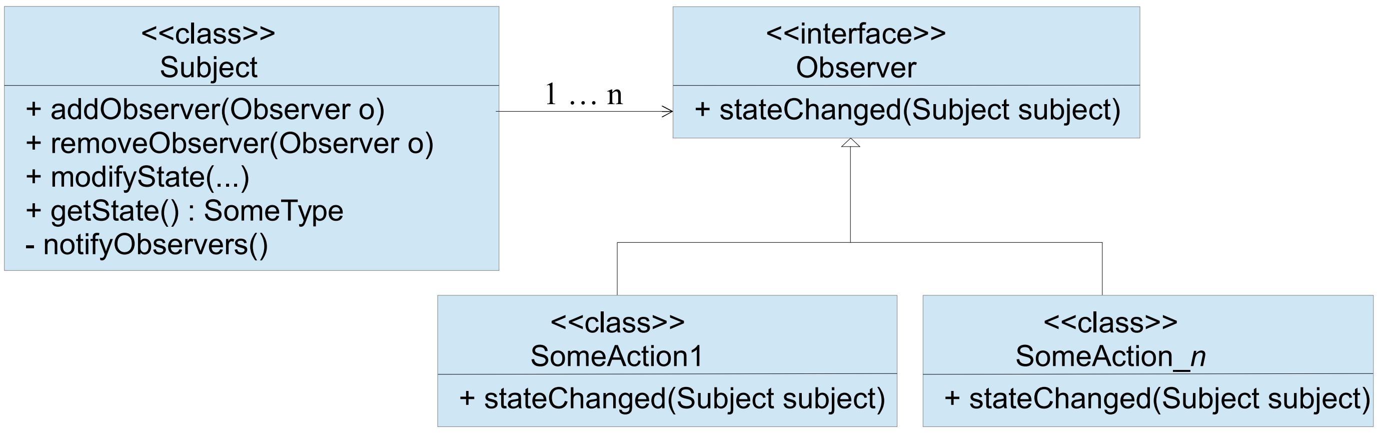
```
public class TemperatureInfo implements TemperatureObserver {
  public void temperatureChanged(double currentTemperature) {
    System.out.println("The temperature now is: " + currentTemperature);
  }
}

public class TemperatureLog implements TemperatureObserver {
  public void temperatureChanged(double currentTemperature) {
    … write currentTemperature to some log file …
  }
}
```

Finally, you would have to create an instance of `TemperatureSensor`, instances of both `TemperatureInfo` and `TemperatureLog`, and wire them together:

```
TemperatureSensor sensor = new TemperatureSensor();
sensor.register(new TemperatureInfo());
sensor.register(new TemperatureLog());
```

Please observe: in this example, in the library created by the sensor maker, you would find the compiled code for `TemperatureSensor` and `TemperatureObserver`. You yourself would be responsible for writing concrete temperature observers, creating an instance of `TemperatureSensor` and creating and registering the actions.

In its simplest form, the Observer pattern can be visualized with the following *structure diagram*. In the diagram, we use class and interface names which are abstract and can represent many different concrete situations. Method names are also deliberately generic. This is the terminology of *Observer* design pattern.



<<class>>
Subject
+ addObserver(Observer o)
+ removeObserver(Observer o)
+ modifyState(...)
+ getState() : SomeType
- notifyObservers()
1 … n
<<interface>>
Observer
+ stateChanged(Subject subject)
<<class>>
SomeAction1
+ stateChanged(Subject subject)
<<class>>
SomeAction_n
+ stateChanged(Subject subject)

In this diagram, the instance of the `Subject` class represents the *object* from previous example. It holds private list of registered observers. The interface `Observer` is the interface that our object expects all actions to implement, and it has a single method:

```
stateChanged(Subject subject);
```

We can implement several different actions (classes `SomeAction1`, `SomeAction2`, …, `SomeAction_n`) that all implement the `Observer` interface. Our object provides usually three methods. Method `addObserver` is used to register a concrete action with our object. Method `getState()` is used to retrieve current state. In example with temperature sensor, the subject itself was responsible for modifying the state (by reading temperature from sensor). In different situations, we would like to allow external user to modify the state: this is the reason why the Subject in previous diagram offers the method `modifyState` – it allows us to modify the object's state. Each time when a state is modified, the subject automatically notifies all registered observers of this change by calling `stateChanged` method on each registered observer. This organization of code will allow us to write the following example:

```
Subject s = new Subject();  // create object with interesting state

Observer observer1 = new SomeAction1();  // create first action
s.addObserver(observer1);                // and register it

Observer observer2 = new SomeAction2();  // create second action
s.addObserver(observer2);                // and register it

s.modifyState(...);  // modify objects state; observer1.stateChanged(s) and
                     // observer2.stateChanged(s) is called as a consequence
s.modifyState(...);  // modify objects state; observer1.stateChanged(s) and
                     // observer2.stateChanged(s) is called as a consequence

s.removeObserver(observer1);

Observer observer3 = new SomeAction3();  // create another action
s.addObserver(observer3);                // and replace the old one with this now

s.modifyState(...);  // modify objects state; now observer2.stateChanged(s)
                     // and observer3.stateChanged(s) is called as a consequence
```

Since this is widely utilized design pattern (for example, it is used throughout graphical user interface libraries in Java), you will practice it on the following example, which you will implement.

The `Subject` class here will be `IntegerStorage`.

```
package hr.fer.zemris.java.hw07.observer1;

public class IntegerStorage {

	private int value;
	private List<IntegerStorageObserver> observers;  // use ArrayList here!!!

	public IntegerStorage(int initialValue) {
		this.value = initialValue;
	}

	public void addObserver(IntegerStorageObserver observer) {
		// add the observer in observers if not already there ...
		// … your code …
	}

	public void removeObserver(IntegerStorageObserver observer) {
		// remove the observer from observers if present ...
		// … your code …
	}

	public void clearObservers() {
		// remove all observers from observers list ...
		// … your code …
	}

	public int getValue() {
		return value;
	}

	public void setValue(int value) {
		// Only if new value is different than the current value:
		if(this.value!=value) {
			// Update current value
			this.value = value;
			// Notify all registered observers
			if(observers!=null) {
				for(IntegerStorageObserver observer : observers) {
					observer.valueChanged(this);
				}
			}
		}
	}
}
```

The `Observer` interface will be `IntegerStorageObserver`.

```
package hr.fer.zemris.java.hw07.observer1;

public interface IntegerStorageObserver {
	public void valueChanged(IntegerStorage istorage);
}
```

The main program is `ObserverExample`:

```
package hr.fer.zemris.java.hw07.observer1;

public class ObserverExample {

	public static void main(String[] args) {

		IntegerStorage istorage = new IntegerStorage(20);

		IntegerStorageObserver observer = new SquareValue();

		istorage.addObserver(observer);
		istorage.setValue(5);
		istorage.setValue(2);
		istorage.setValue(25);

		istorage.removeObserver(observer);

		istorage.addObserver(new ChangeCounter());
		istorage.addObserver(new DoubleValue(5));
		istorage.setValue(13);
		istorage.setValue(22);
		istorage.setValue(15);

	}
}
```

Copy these three sources into your Eclipse project. Your task is to implement three concrete observers: `SquareValue` class, `ChangeCounter` class, `DoubleValue` class. Instances of `SquareValue` class write a square of the integer stored in the `IntegerStorage` to the standard output (but the stored integer itself is not modified!), instances of `ChangeCounter` counts (and writes to the standard output) the number of times the value stored has been changed since this observer's registration. Instances of `DoubleValue` class write to the standard output double value (i.e. "value * 2") of the current value which is stored in subject, but only first *n* times since its registration with the subject (*n* is given in constructor); after writing the double value for the *n*-th time, the observer automatically de-registers itself from the subject. To simplify the design, *lets assume that observer objects will not be reused* (registered to one *Subject*, then after some time unregistered and registered to another *Subject*, etc; this way, the `DoubleValue` class can track how many changes have occurred after it was created; generally tracking how many changes occurred after it was registered on some *Subject* is not possible in the described scenario since the *Obeserver* is not notified about the registration events). The output of the previous code should be as follows:

```
Provided new value: 5, square is 25
Provided new value: 2, square is 4
Provided new value: 25, square is 625
Number of value changes since tracking: 1
Double value: 26
Number of value changes since tracking: 2
Double value: 44
Number of value changes since tracking: 3
Double value: 30
```

Now modify method `main` in `ObserverExample` class by adding two more observer registrations: replace the code that removes `SquareValue` observer and registers two observers:

```
		istorage.removeObserver(observer);

		istorage.addObserver(new ChangeCounter());
		istorage.addObserver(new DoubleValue(5));
```

by the following code snippet:

```
            istorage.removeObserver(observer);

            istorage.addObserver(new ChangeCounter());
            istorage.addObserver(new DoubleValue(1));
            istorage.addObserver(new DoubleValue(2));
            istorage.addObserver(new DoubleValue(2));
```

Now run the program and observe how it fails with exception. Try to figure out what is the root cause for this behavior. Then fix it (for observers storage in this homework, you must use `ArrayList`; other implementations or subclasses are not permitted). After that, the previous code should work. Hint: google "*copyonwrite observer pattern*" - go for more efficient implementation under more frequent reading than writing operations.

After you finish this task, copy the content of subpackage `observer1` into `observer2`. You will continue your work there while package `observer1` will preserve your previous solution.

Let us repeat what we have done so far. We have developed our `Subject` to allow a registration of multiple observers. We have defined the `Observer` interface and have developed more than one actual observer (classes that implemented the `Observer` interface). Now you will modify your code from package `observer2` to support the following.

- Change the `Observer` interface (i.e. `IntegerStorageObserver`) so that instead of a reference to `IntegerStorage` object, the method `valueChanged` gets a reference to an instance of `IntegerStorageChange` class (and create this class). Instances of `IntegerStorageChange` class should encapsulate (as read-only properties) the following information: (a) a reference to the `IntegerStorage`, (b) the value of stored integer before the change has occurred, and (c) the new value of currently stored integer.
- During the notifications dispatching, for a single change only a single instance of the `IntegerStorageChange` class should be created, and a reference to that instance should be passed to all registered observers (the order is not important). Since this instance provides only a read-only properties, we do not expect any problems.
- Modify all other classes to support this change.
- Modify the main program so that it registers all developed observers (once) at the beginning of the program and then performs calls to `istorage.setValue(...)`.

Why are we doing this modification? Instead of sending just the *state*, we would like to provide the *Observer* with more detailed information, containing the source of state (reference to the *Subject*) and previous and current state. One way to achieve this would be to increase the number of arguments of the method declared in the *Observer* interface. Better way is to encapsulate all these information into a separate object and send only a reference to that object to the *Observer* – which is the approach taken here, as well as in many popular libraries (i.e. *xxxListeners* in AWT/Swing encapsulate change information in `java.util.EventObject` and subclasses).

Problem 2.

Write a class `PrimesCollection` and put the implementation class in the package `hr.fer.zemris.java.hw07.demo2`. The class must be usable in the following scenario:

```
PrimesCollection primesCollection = new PrimesCollection(5); // 5: how many of them
for(Integer prime : PrimesCollection) {
    System.out.println("Got prime: "+prime);
}
```

The previous snippet should produce output:

```
Got prime: 2
Got prime: 3
Got prime: 5
Got prime: 7
Got prime: 11
```

For implementation of this class you are forbidden to use any multiple element storage: no lists, no array or anything else; a next prime should be calculated only when needed (yes, it is inefficient, but not relevant here). Constructor of this class accepts a number of consecutive primes that must be in this collection. You must write your own support that will allow objects of your class to be used in `for`-loops.

Hint: use nested classes.

Info: to help you to properly design you class, be aware that the following code must also work (and any similarly arbitrarily-deep nested loops) :

```
PrimesCollection primesCollection = new PrimesCollection(2);
for(Integer prime : primesCollection) {
  for(Integer prime2 : primesCollection) {
    System.out.println("Got prime pair: "+prime+", "+prime2);
  }
}
```

which should produce the following output:

```
Got prime pair: 2, 2
Got prime pair: 2, 3
Got prime pair: 3, 2
Got prime pair: 3, 3
```

Place these two demonstration programs in classes `PrimesDemo1` and `PrimesDemo2` in the same package as `PrimesCollection` class.

Problem 3.

Create an implementation of `ObjectMultistack`. You can think of it as a `Map`, but a special kind of `Map`. While `Map` allows you only to store for each key a single value, `ObjectMultistack` must allow the user to store multiple values for same key and it must provide **a stack-like abstraction** (do not use `java.util.Stack` literally!!!). Keys for your `ObjectMultistack` will be instances of the class `String`. Values that will be associated with those keys will be instances of class `ValueWrapper` (you will also create this class). `ObjectMultistack` is not parameterized. Let me first give you an example.

```
package hr.fer.zemris.java.custom.scripting.demo;

import hr.fer.zemris.java.custom.scripting.exec.ObjectMultistack;
import hr.fer.zemris.java.custom.scripting.exec.ValueWrapper;

public class ObjectMultistackDemo {

	public static void main(String[] args) {
		ObjectMultistack multistack = new ObjectMultistack();

		ValueWrapper year = new ValueWrapper(Integer.valueOf(2000));
		multistack.push("year", year);

		ValueWrapper price = new ValueWrapper(200.51);
		multistack.push("price", price);

		System.out.println("Current value for year: "
		                              + multistack.peek("year").getValue());
		System.out.println("Current value for price: "
		                              + multistack.peek("price").getValue());

		multistack.push("year", new ValueWrapper(Integer.valueOf(1900)));
		System.out.println("Current value for year: "
		                              + multistack.peek("year").getValue());

		multistack.peek("year").setValue(
		            ((Integer)multistack.peek("year").getValue()).intValue() + 50
		);
		System.out.println("Current value for year: "
		                              + multistack.peek("year").getValue());

		multistack.pop("year");
		System.out.println("Current value for year: "
		                              + multistack.peek("year").getValue());

		multistack.peek("year").add("5");
		System.out.println("Current value for year: "
		                              + multistack.peek("year").getValue());
		multistack.peek("year").add(5);
		System.out.println("Current value for year: "
		                              + multistack.peek("year").getValue());
		multistack.peek("year").add(5.0);
		System.out.println("Current value for year: "
		                              + multistack.peek("year").getValue());

	}

}
```

This short program should produce the following output:

```
Current value for year: 2000
Current value for price: 200.51
Current value for year: 1900
Current value for year: 1950
Current value for year: 2000
Current value for year: 2005
Current value for year: 2010
Current value for year: 2015.0
```

Your `ObjectMultistack` class must provide the following methods:

```
package hr.fer.zemris.java.custom.scripting.exec;

public class ObjectMultistack {

      public void push(String keyName, ValueWrapper valueWrapper) {...}
      public ValueWrapper pop(String keyName) {...}
      public ValueWrapper peek(String keyName) {...}
      public boolean isEmpty(String keyName) {...}

}
```

Of course, you are free to add any private method you need. The semantic of methods `push`/`pop`/`peek` is as usual, except they are bound to “virtual” stacks, each defined by given name. In a way, you can think of this collection as a map that associates `String`s (keys) with stacks of `ValueWrapper`s (values). And this virtual stacks for two different string are completely isolated from each other.

Your job is to implement this collection. However, **<u>you are not allowed</u>** to use instances of existing class `Stack` from Java Collection Framework. Instead, **you must define your inner static class** `MultistackEntry` that acts as a node of a single-linked list. Then privately use some implementation of interface `Map` to map names to instances of `MultistackEntry` class (making `ObjectMultistack` an Adapter; remind yourself of *Adapter* design pattern). Using `MultistackEntry` class you can efficiently implement simple stack-like behavior that is needed for this homework. Your `Map` object will then map each key to first list node (represented by `MultistackEntry` object which will store a reference to `ValueWrapper` object and a reference to next `MultistackEntry` object). Said differently, you use `MultistackEntry` objects to form linked lists which represent individual stacks; each linked list is one stack. Internally, you use one `Map` instance to associate each key with a head-node of linked list, i.e. stack. Use the combination of the two to implement for each key a stack with O(1) `put`/`peek`/`pop`.

Methods `pop` and `peek` should throw an appropriate exception if called upon empty stack.

Finally, you must implement `ValueWrapper` class whose structure is as follows.

- It must have a read-write property `value` of type `Object`. It must accept objects of any type, or `null`.
- It must have a single public constructor that accepts initial value.
- It must have four arithmetic methods:
  - `public void add(Object incValue);`
  - `public void subtract(Object decValue);`
  - `public void multiply(Object mulValue);`
  - `public void divide(Object divValue);`
- It must have additional numerical comparison method:
  - `public int numCompare(Object withValue);`

All four arithmetic operation modify the current value. However, there is a catch. Although instances of `ValueWrapper` do allow us to work with objects of any types, if we call arithmetic operations, it should be obvious that some restrictions will apply **at the moment of the method call** (e.g. we can not multiply a network socket with a GUI window). So here are the rules. Please observe that we have to consider three elements:

1. what is the type of currently stored value in `ValueWrapper` object,
2. what is the type of argument provided in method, and
3. what will be the type of the new value that will be stored as a result of invoked operation.

Since the `ValueWrapper` object is allowed to wrap objects of any classes, we prescribe that the allowed values for the current content of the `ValueWrapper` object and for the argument of the arithmetic or comparison method, at the moment the method is called, are `null` or the instances of `Integer`, `Double` and `String` classes. If this is not the case, throw a `RuntimeException` with appropriate explanation.

```
ValueWrapper vv1 = new ValueWrapper(Boolean.valueOf(true));
vv1.add(Integer.valueOf(5));  // ==> throws, since current value is boolean

ValueWrapper vv2 = new ValueWrapper(Integer.valueOf(5));
vv2.add(Boolean.valueOf(true));  // ==> throws, since the argument value is boolean
```

The rules which follow will explain how to perform arithmetic operations (possibly by promoting data temporarily into different types). Here are the rules.

Rule 1: If either current value or argument is `null`, you should treat that value as being equal to `Integer` with value 0.

Rule 2: If current value and argument are not `null`, they can be instances of `Integer`, `Double` or `String`. For each value that is `String`, you should check if `String` literal is decimal value (i.e. does it have somewhere a symbol `'.'` or `'E'`). If it is a decimal value, treat it as such; otherwise, treat it as an `Integer` (if conversion fails, you are free to throw `RuntimeException` since the result of operation is undefined anyway).

Rule 3: Now, if either current value or argument is `Double`, operation should be performed on `Double`s, and the result should be stored as an instance of `Double`. If not, both arguments must be `Integer`s so the operation should be performed on `Integer`s and the result stored as an `Integer`.

If you carefully examine the output of program `ObjectMultistackDemo`, you will see this happening!

Please note: you have four methods that must somehow determine on which kind of arguments they will perform the selected operation and what will be the result – please do not *copy&paste* appropriate code four times; instead, isolate it into one (or more) private methods (or classes?) that will prepare what is necessary for these four methods to do its job. *Strategy* design pattern can be your best friend here!

Rules for the `numCompare` method are similar. This method does not perform any change in the stored data. It performs numerical comparison between the currently stored value in the `ValueWrapper` object and given argument. The method returns an integer less than zero if the currently stored value is smaller than the argument, an integer greater than zero if the currently stored value is larger than the argument, or an integer 0 if they are equal.

- If both values are `null`, treat them as equal.
- If one is `null` and the other is not, treat the `null`-value being equal to an integer with value 0.

- Otherwise, promote both values to same type as described for arithmetic methods and then perform the comparison.

If done correctly, the whole class `ValueWrapper` with all required functionality can be written in less than 100 lines (even counting the empty lines between methods and in methods, but without javadoc).

You are required to add unit tests for classes described in this problem. Especially test the behavior of the `ValueWrapper` object under the arithmetic and comparison operations. Here you have a lot of different cases to cover.

Here are several examples of expected behavior.

```
ValueWrapper v1 = new ValueWrapper(null);
ValueWrapper v2 = new ValueWrapper(null);
v1.add(v2.getValue());     // v1 now stores Integer(0); v2 still stores null.

ValueWrapper v3 = new ValueWrapper("1.2E1");
ValueWrapper v4 = new ValueWrapper(Integer.valueOf(1));
v3.add(v4.getValue());     // v3 now stores Double(13); v4 still stores Integer(1).

ValueWrapper v5 = new ValueWrapper("12");
ValueWrapper v6 = new ValueWrapper(Integer.valueOf(1));
v5.add(v6.getValue());     // v5 now stores Integer(13); v6 still stores Integer(1).

ValueWrapper v7 = new ValueWrapper("Ankica");
ValueWrapper v8 = new ValueWrapper(Integer.valueOf(1));
v7.add(v8.getValue());     // throws RuntimeException
```

Problem 4.

Zadatak rješavate u paketu `hr.fer.zemris.java.hw07.demo4`.

U datoteci `studenti.txt` (skinite je s Ferka – repozitorij, Dodatci 7. domaćoj zadaći) nalaze se bodovi za 500 studenata jednog kolegija. U datoteci su redom:

- jmbag,
- prezime,
- ime,
- broj bodova na međuispitu,
- broj bodova na završnom ispitu,
- broj bodova na laboratorijskim vježbama te
- ocjena.

Učitajte sadržaj te datoteke u memoriju pozivom `Files.readAllLines` koja vraća listu redaka (ovu smo metodu već koristili – podsjetite se iz prethodnih zadaća kako).

Modelirajte zapis o studentu prikladnim razredom `StudentRecord` koji sadrži sve podatke (jednog retka). Razredu definirajte metodu toString() koja generira string identičnog formata prema kojem su podatci upisani u tekstovnu datoteku (znači: jmbag, tab, …, tab, ocjena). Pretočite učitanu datoteku u listu objekata tipa `StudentRecord`. Evo pseudokoda:

```
List<String> lines = Files.readAllLines(… "./studenti.txt" …)
List<StudentRecord> records = convert(lines);
```

Stazu do datoteke ostavite kako je prikazano (relativnu s obzirom na Vaš trenutni projekt, odnosno direktorij iz kojeg se program pokreće).

Smjestite ovaj kod u program `StudentDemo` u metodu `main`. Potom uporabom novog tokovnog API-ja namijenjenog obradi podataka iz kolekcija ostvarite sljedeće zadatke. Svaki zadatak mora biti jedna "ulančana" naredba.

1. Odrediti broj studenata koji u sumi MI+ZI+LAB imaju više od 25 bodova

```
long broj = records.stream()….;
```

2. Odrediti broj studenata koji su dobili ocjenu 5:

```
long broj5 = records.stream()….;
```

3. Pripremiti listu studenata koji su dobili ocjenu 5 (redosljed u listi nije bitan):

```
List<StudentRecord> odlikasi = records.stream()….;
```

4. Pripremiti listu studenata koji su dobili ocjenu 5 pri čemu redosljed u listi mora biti takav da je na prvom mjestu student koji je ukupno ostvario najviše bodova a na zadnjem mjestu onaj koji je ukupno ostvario najmanje (dakle, sortirano po bodovima):

```
List<StudentRecord> odlikasiSortirano = records.stream()….;
```

5. Pripremiti listu JMBAG-ova studenata koji nisu položili kolegij, sortiranu prema JMBAG-u (od manjeg prema većem; uočite, svi su JMBAGovi stringovi s 10 znamenaka).

```
List<String> nepolozeniJMBAGovi = records.stream()….;
```

6. Pripremiti mapu čiji su ključevi ocjene a vrijednosti liste studenata s tim ocjenama (hint: pogledati `Collectors.groupingBy`).

```
Map<Integer, List<StudentRecord>> mapaPoOcjenama = records.stream()….;
```

7. Pripremiti mapu čiji su ključevi ocjene a vrijednosti broj studenata s tim ocjenama. Za ovo iskoristite sljedeću metodu razreda `Collectors`:

```
Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,
                                Function<? super T, ? extends U> valueMapper,
                                BinaryOperator<U> mergeFunction)
```

```
Map<Integer, Integer> mapaPoOcjenama2 = records.stream()…;
```

8. Pripremiti mapu s ključevima `true`/`false` i vrijednostima koje su liste studenata koji su prošli (za ključ `true`) odnosno koji nisu prošli kolegij (za ključ `false`). Hint: koristite `Collectors.partitioningBy`.

```
Map<Boolean, List<StudentRecord>> prolazNeprolaz = records.stream()….;
```

Svaku od prethodnih točaka potrebno je ostvariti u zasebnoj metodi koja vraća propisani podatak a prima samo listu `List<StudentRecord> records`. Imena tih metoda moraju redom biti:

1. vratiBodovaViseOd25
2. vratiBrojOdlikasa
3. vratiListuOdlikasa
4. vratiSortiranuListuOdlikasa
5. vratiPopisNepolozenih
6. razvrstajStudentePoOcjenama
7. vratiBrojStudenataPoOcjenama
8. razvrstajProlazPad

Povratni tip možete vidjeti iz primjera. Glavni program treba pozvati svaku od tih metoda te pripremiti ispis u sljedećem (okvirnom formatu):

```
Zadatak 1
=========
xx
Zadatak 2
=========
xx
…
```

Pri tome “xx” u zadatcima koji traže broj predstavlja izračunati broj, kod zadataka koji traže jednostavnu kolekciju ispis elemenata kolekcije (jedan po retku), a kod zadataka koji traže mapu ispis u kojem su za svaki ključ ispisane pridružene vrijednosti (točan format nije bitan).

Please note. You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries for this homework (whether it is yours old code or someones else). You can use Java Collection Framework classes (unless explicitly prohibited in specific problem). Document your code!

The consultations are at standard times.

All source files must be written using UTF-8 encoding. All classes, methods and fields (public, private or otherwise) must have appropriate javadoc.

You are expected to write tests for problem 3.
You are encouraged to write tests for other problems.

When your homework is done, pack it in zip archive with name `hw07-0000000000.zip` (replace zeros with your JMBAG). Upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is April 25th 2019. at 11:59 PM.