6. homework assignment; JAVA, Academic year 2015/2016; FER

First: read last page. I mean it! You are back? OK. This homework consists of five problems.

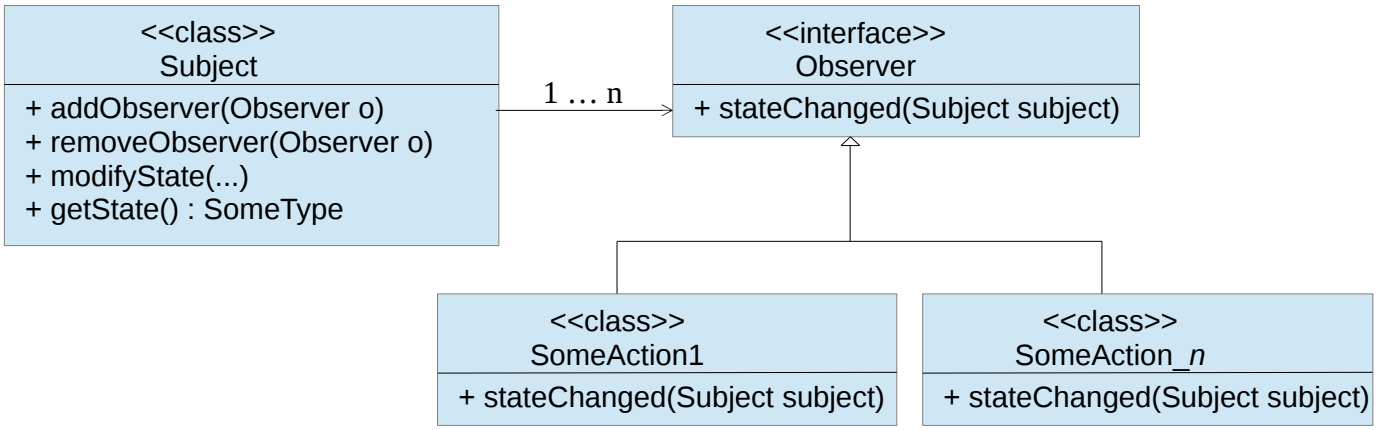
Problem 1.

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 even more than one action (to process the data, to inform user about the change, to update the GUI, etc). If you have full control of the object and if there is always the same (single) action, this can be easily coded into the object itself. However, often you will develop the object separately (or will be provided with one that is not under your control), and you will want to be able to change the actions when necessary, and to develop new actions without ever changing or recompiling the object itself.

The *Observer pattern* is an appropriate solution that can be utilized in previously described situation. The basic idea is this: your object (denoted as the *Subject* in this design pattern) 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 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*);
2. the object (the *Subject*) will provide you with a method that will allow you to register actions (*Concrete Observers*) you developed, and which, of course, implements the *Observer* interface;
3. the object (the *Subject*) can provide you a method for action removal so that you can at any time de-register previously registered actions;
4. every time the objects' state changes, the object will invoke all of the registered actions by using methods of prescribed interface.

At its simplest case, the observer pattern can be depicted as following picture shows.



In this picture, 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 and method `modifyState` allows us to modify the objects 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.

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

```
package hr.fer.zemris.java.tecaj.hw6.observer1;

public class IntegerStorage {

	private int value;
	private List<IntegerStorageObserver> observers;

	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.tecaj.hw6.observer1;

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

The main program is `ObserverExample`:

```
package hr.fer.zemris.java.tecaj.hw6.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(2));
		istorage.setValue(13);
		istorage.setValue(22);
		istorage.setValue(15);

	}

}
```

Copy these three sources into your Eclipse project. Your task is to implement four 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, instances of `ChangeCounter` counts (and writes to the standard output) the number of times value stored integer has been changed since the registration. Instances of `DoubleValue` class write to the standard output double value of the current value which is stored in subject, but only first *n* times since its registation with 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 deregistered and registered to another Subject, etc). 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
```

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());
```

by the following code snippet:

```
istorage.removeObserver(observer);

istorage.addObserver(new ChangeCounter());
istorage.addObserver(new DoubleValue(2));
istorage.addObserver(new DoubleValue(1));
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. After that, the previous code should work.

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

Lets recapitulate 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 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) following information: (a) a reference to `IntegerStorage`, (b) the value of stored integer before the change has occurred, and (c) the new value of currently stored integer.
- During the dispatching of notifications, for a single change only a single instance of `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(...)`.

Problem 2.

Write a generic class `LikeMedian` and put the implementation class in the package `hr.fer.zemris.java.tecaj.hw6.demo2`. The class must be usable in the following scenario:

```
LikeMedian<Integer> likeMedian = new  LikeMedian<Integer>();
likeMedian.add(new Integer(10));
likeMedian.add(new Integer(5));
likeMedian.add(new Integer(3));
Optional<Integer> result = likeMedian.get();
System.out.println(result);

for(Integer elem : likeMedian) {
  System.out.println(elem);
}
```

and it must have `java.lang.Object` as its direct parent (so do not extend any other class; implementing interfaces is fine). The previous program should result with:

```
5
10
5
3
```

The class should be able to work with any type which has defined natural ordering (remember the last lecture?); think carefully what implication this has on generic parameter bounds. If user adds an odd number of elements, the `get()` method must return median element. If user provides an even number of elements, the `get()` method must return the smaller from the two elements which would usually be used to calculate median element. If no elements are given, the method `get()` must not throw an exception but return an appropriate `Optional` object – this is exactly what they are used for (see the book, version 2015-09-30, end of page 265, start of page 266).

When completed correctly, this should also work:

```
LikeMedian<String> likeMedian = new  LikeMedian<String>();
likeMedian.add("Joe");
likeMedian.add("Jane");
likeMedian.add("Adam");
likeMedian.add("Zed");
String result = likeMedian.get();
System.out.println(result); // Writes: Jane
```

Place these two demonstration programs in classes `MedianDemo1` and `MedianDemo2` in the same package as `LikeMedian` class.

Problem 3.

Write a class `PrimesCollection` and put the implementation class in the package `hr.fer.zemris.java.tecaj.hw6.demo3`. The class must be usable in 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: please 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 4.

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. 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). 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").increment("5");
		System.out.println("Current value for year: "
					+ multistack.peek("year").getValue());
		multistack.peek("year").increment(5);
		System.out.println("Current value for year: "
					+ multistack.peek("year").getValue());
		multistack.peek("year").increment(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 following methods:

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

public class ObjectMultistack {

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

}
```

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 bounded to "virtual" stack defined by given name. In a way, you can think of this collection as a map that associates strings with stacks. 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 use some implementation of interface `Map` to map names to instances of `MultistackEntry` class. 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).

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 have a single public constructor that accepts initial value.
- It must have four arithmetic methods:
  - `public void increment(Object incValue);`
  - `public void decrement(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 method call (e.g. we can not multiply a network socket with a GUI window). 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, and
3. what will be the type of new value that will be stored as a result of invoked operation.

We define that allowed values for current content of `ValueWrapper` object and for argument are `null` and instances of `Integer`, `Double` and `String` classes. If this is not the case, throw a RuntimeException with explanation.

Further, if any of current value or argument is `null`, you should treat that value as being equal to `Integer` with value 0.

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).

Now, if either current value or argument is `Double`, operation should be performed on `Double`s, and result should be stored as an instance of `Double`. If not, both arguments must be `Integers` so operation should be performed on `Integers` and 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 it will perform the selected operation and what will be the result – please do not copy&paste appropriate code four times; instead, isolate it in one (or more) private methods that will prepare what is necessary for these four methods to do its job.

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

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.

Problem 5.

Zadatak rješavate u paketu `hr.fer.zemris.java.tecaj.hw6.demo5`.

U datoteci `studenti.txt` (skinite je s Ferka – repozitorij, Dodatci 6. 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). Pretočite učitanu datoteku u listu objekata tipa `StudentRecord`. Evo pseudokoda:

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

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().…;
```

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). However, you are free to use classes which are part of Java Standard Edition platform and which were covered on lectures. Document your code!

In order to solve this homework, create a blank Eclipse Java Project and write your code inside. Create additional source folder tests/multistack (you can create other if you wish). This is for junit tests of problem 4 which are required. Junit tests for other problems are optional.

You must name your project's main directory (which is usually also the project name) `HW06-`*yourJMBAG*; for example, if your JMBAG is 0012345678, the project name and the directory name must be `HW06-0012345678`. Once you are done, export the project as a ZIP archive and upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is April 21th 2016. at 08:00 AM.