4. homework assignment; JAVA, Academic year 2011/2012; FER

As usual, please see the last page. I mean it! You are back? OK. This homework consists of two problems.

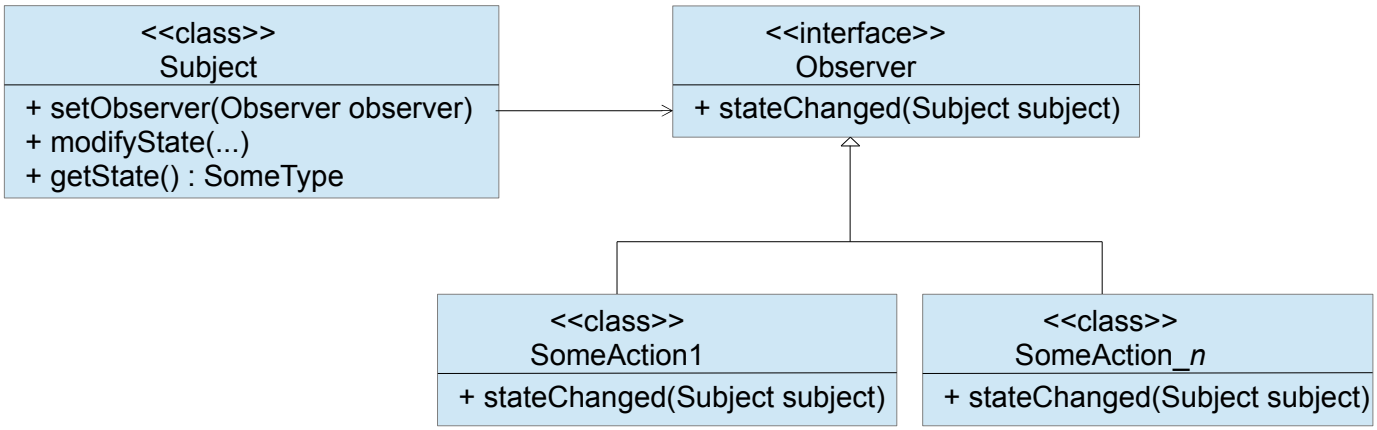
Problem 1.

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 (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 action 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 does not need to know anything about the concrete action you will develop; however, it will mandate that in order to be able to invoke your action, you must satisfy following conditions:

1. the object will have to be able to "talk" with your action in a way that it expects – this means that the object will prescribe a certain interface your action will have to implement; this interface is usually called the Observer;
2. the object will provide you with a method that will allow you to register the action you developed (and which, of course, implements the Observer interface);
3. every time the objects' state changes, the object will invoke the registered action 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. 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 `setObserver` 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. 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.setObserver(observer1);                // and register it

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

Observer observer2 = new SomeAction2();  // create other action
s.setObserver(observer2);                // and replace the old one with this now

s.modifyState(...);  // modify objects state; now observer2.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 a following example.

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

```
package hr.fer.zemris.java.tecaj.hw4.problem1a;

public class IntegerStorage {

	private int value;
	private IntegerStorageObserver observer;

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

	public void setObserver(IntegerStorageObserver observer) {
		this.observer = observer;
	}

	public void clearObserver() {
		this.observer = null;
	}

	public int getValue() {
		return value;
	}

	public void setValue(int value) {
		if(this.value!=value) {
			this.value = value;
			if(observer!=null) {
				observer.valueChanged(this);
			}
		}
	}
}
```

The `Observer` interface will be `IntegerStorageObserver`.

```
package hr.fer.zemris.java.tecaj.hw4.problem1a;

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

The main program is `ObserverExample`:

```
package hr.fer.zemris.java.tecaj.hw4.problem1a;

public class ObserverExample {

	public static void main(String[] args) {

		IntegerStorage istorage = new IntegerStorage(20);

		IntegerStorageObserver observer = new SquareValue();

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

		istorage.setObserver(new ChangeCounter());
		istorage.setValue(13);
		istorage.setValue(22);
		istorage.setValue(15);

	}

}
```

Copy these three sources into your Eclipse project. Your task is to implement two concrete observers: `SquareValue` class and `ChangeCounter` class. Instances of `SquareValue` class write a square of the integer stored in the `IntegerStorage` and instances of `ChangeCounter` counts (and writes) the number of times value stored integer has been changed since the registration. 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
Number of value changes since tracking: 2
Number of value changes since tracking: 3
```

After you finish this task, copy the content of subpackage problem1a into problem1b. You will continue your work here while package1a will preserve your previous solution.

Lets recapitulate what we have done so far. We developed our `Subject` to allow registration of a single observer. We defined the `Observer` interface and developed more that one actual observers (classes that implemented the `Observer` interface). Now you will modify your code from package problem1b to support following.

- Remove method `setObserver(...)` from `IntegerStorage` and append methods that will allow registration and removal of multiple observers. You can select any supporting structure you see fit for holding registered observers. Use the method names: `addObserver` and `removeObserver`. In case of `removeObserver` method, if observer given as argument is not registered, method should do nothing. When you notify observer that integer value has changed, the order in which you will notify them is not important.
- 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. Instances of `IntegerStorageChange` class should encapsulate (as read-only properties) following information: (a) a reference to `IntegerStorage`, (b) the value of stored integer before change occurred, and (c) the 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. 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 both observers at the beginning of the program and then performs calls to `istorage.setValue(...)`.

Problem 2.

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, you are not allowed to use instances of class `Stack`. Instead, you should define your inner 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 behaviour that is needed for this homework.

Methods `pop` and `peek` should throw 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 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 `Integer`s so operation should be performed on `Integer`s 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.

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). Document your code!

In order to solve this homework, create a blank Eclipse Java Project and write your code inside. Once you are done, export project as a ZIP archive and upload this archive on Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted.