Preparatory homework assignment; JAVA, Academic year 2019/2020; FER

Problem 1.

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, **you are not allowed** 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
```