2. homework assignment; JAVA, Academic year 2017/2018; FER

In order to solve this homework, you are expected to read (with understanding) chapters 5 and 6 in book. After that you can proceed with this homework. This homework consists of five problems. During the semester we will return to this code, modify it, polish it and use it to implement some very cool stuff. You will have to reuse the code you write here, so write it smart. Be patient and please, don't panic. Breathe deeply. OK, here we go…

Start by creating a blank Maven project, just as you did for 1. homework assignment: in Eclipse workspace directory create directory `hw02-0000000000` (replace zeros with your JMBAG) and inside setup Maven project `hr.fer.zemris.java.jmbag0000000000:hw02-0000000000` (replace zeros with your JMBAG) and add dependency for `junit:junit:4.12`. Import it into Eclipse. Now you can start solving actual problems.

Problem 1.

All of the following classes should be placed in package `hr.fer.zemris.java.custom.collections`.

Define a class `Processor`. It must define a single method:

```
public void process(Object value);
```

with empty body (its implementation exists but does nothing).

Then define a class `Collection` which represents some general collection of objects. This class should provide only a protected default constructor. Class `Collection` must provide following public methods.

`boolean isEmpty();` Returns `true` if collection contains no objects and `false` otherwise. Implement it here to determine result by utilizing method `size()`.
`int size();` Returns the number of currently stored objects in this collections. Implement it here to always return 0.
`void add(Object value);` Adds the given object into this collection. Implement it here to do nothing.
`boolean contains(Object value);` Returns `true` only if the collection contains given `value`, as determined by `equals` method. Implement it here to always return `false`. It is OK to ask if collection contains `null`.
`boolean remove(Object value);` Returns `true` only if the collection contains given `value` as determined by `equals` method and removes one occurrence of it (in this class it is not specified which one). Implement it here to always return `false`.
`Object[] toArray();` Allocates new array with size equals to the size of this collections, fills it with collection content and returns the array. This method never returns `null`. Implement it here to throw `UnsupportedOperationException`.
`void forEach(Processor processor);` Method calls `processor.process(.)` for each element of this collection. The order in which elements will be sent is undefined in this class. Implement it here as an empty method.

`void addAll(Collection other);` Method adds into the current collection all elements from the given collection. This other collection remains unchanged. Implement it here to define a local processor class (read about *Local classes* in book) whose method `process` will add each item into the current collection by calling method `add`, and then call `forEach` on the other collection with this processor as argument. You must define this new class directly in the method `addAll` (such classes are called local classes).
`void clear();` Removes all elements from this collection. Implement it here as an empty method.

Since this class does not actually have any storage capabilities, you wont be able to test it yet.

Problem 2.

Write an implementation of resizable array-backed collection of objects denoted as `ArrayIndexedCollection` which extends class `Collection` from previous problem.
Put it also in package `hr.fer.zemris.java.custom.collections`. Each instance of this class should manage three *private* variables:

- `size` – current size of collection (number of elements actually stored),
- `capacity` – current capacity of allocated array of object references, and
- `elements` – an array of object references which length is determined by `capacity` variable.

General contract of this collection is: duplicate elements **are allowed**; storage of `null` references **is not allowed**.

You should provide *four* constructors. The default constructor should create an instance with `capacity` set to 16 (this also means that constructor should preallocate the `elements` array of that size). The second constructor should have a single integer parameter: `initialCapacity` and should set the `capacity` to that value, as well as preallocate the `elements` array of that size. If initial capacity is less then 1, an `IllegalArgumentException` should be thrown. Other two constructors are variation of the previous two, but they accept additional parameter (as first argument): reference to some other `Collection` which elements are copied into this newly constructed collection; if the `initialCapacity` is smaller than the given collection size, given collection size should be used for `elements` array preallocation. If the given collection is `null`, `NullPointerException` should be thrown. Please implement the simpler constructors so that they delegate the construction process to the more complex constructors (read section "Delegiranje zadaće konstrukcije objekta" in book, chapter 5).

This class should override empty method definitions inherited from the `Collection` class with an appropriate implementation.

This class should also have all of the methods given below. Please note that some methods are copied from the previous problem but have better specified behavior.

`void add(Object value);` Adds the given object into this collection (reference is added into first empty place in the `elements` array; if the `elements` array is full, it should be reallocated *by doubling* its size). The method should refuse to add `null` as element by throwing the appropriate exception (`NullPointerException`). What is the average complexity of this method?
`Object get(int index);` Returns the object that is stored in backing array at position `index`. Valid indexes are 0 to `size`-1. If `index` is invalid, the implementation should throw the appropriate exception (`IndexOutOfBoundsException`). What is the average complexity of this method?
`void clear();` Removes all elements from the collection. The allocated array is left at current capacity. Do not just set size to 0; write `null` references into the backing array so that objects which became unreferenced become eligible for garbage collection. Do not allocate new array.
`void insert(Object value, int position);` **Inserts** (does not overwrite) the given `value` at the given `position` in array (observe that before actual insertion elements at `position` and at greater positions must be shifted one place toward the end, so that an empty place is created at `position`). The legal positions are 0 to `size` (both are included). If `position` is

invalid, an appropriate exception should be thrown (`IndexOutOfBoundsException`). Except the difference in position at witch the given object will be inserted, everything else should be in conformance with the method `add`. What is the average complexity of this method?
`int indexOf(Object value);` Searches the collection and returns the index of the first occurrence of the given `value` or -1 if the `value` is not found. Argument can be `null` and the result must be that this element is not found (since the collection can not contain `null`). The equality should be determined using the `equals` method. What is the average complexity of this method?
`void remove(int index);` Removes element at specified `index` from collection. Element that was previously at location `index+1` after this operation is on location `index`, etc. Legal indexes are `0` to `size-1`. In case of invalid index throw an appropriate exception (`IndexOutOfBoundsException`).

You are expected to write junit tests for all methods described in the previous table.

Problem 3.

Write an implementation of linked list-backed collection of objects denoted as `LinkedListIndexedCollection` which extends class `Collection` from previous problem.
Put it also in package `hr.fer.zemris.java.custom.collections`.
This class should define private static class `ListNode` with pointers to previous and next list node and additional reference for value storage.

Each instance of this class should manage three *private* variables:

- `size` – current size of collection (number of elements actually stored; number of nodes in list),
- `first` – reference to the first node of the linked list,
- `last` – reference to the last node of the linked list.

General contract of this collection is: duplicate elements **are allowed** (each of those element will be held in different list node); storage of `null` references **is not allowed**.

You should provide *two* constructors. The default constructor should create an empty collection with `first=last=null`. The second constructor should have a single parameter: reference to some other `Collection` whose elements are copied into this newly constructed collection.

This class should override empty method definitions inherited from `Collection` class with appropriate implementation.

This class should also have all of the methods given below. Please note that some methods are copied from the previous problem but have better specified behavior.

`void add(Object value);`
Adds the given object into this collection at the end of collection; newly added element becomes the element at the biggest index. Implement it with complexity O(1). The method should refuse to add `null` as element by throwing the appropriate exception (`NullPointerException`).

`Object get(int index);`
Returns the object that is stored in linked list at position `index`. Valid indexes are 0 to `size`-1. If `index` is invalid, the implementation should throw the appropriate exception (`IndexOutOfBoundsException`). Implement this method so that it never has the complexity greater than $n/2+1$.

`void clear();`
Removes all elements from the collection. Collection “forgets” about current linked list.

`void insert(Object value, int position);`
Inserts (does not overwrite) the given `value` at the given `position` in linked-list. Elements starting from this position are shifted one position. The legal positions are 0 to `size`. If `position` is invalid, an appropriate exception should be thrown. Except the difference in position at witch the given object will be inserted, everything else should be in conformance with the method `add`. What is the average complexity of this method?

`int indexOf(Object value);`
Searches the collection and returns the index of the first occurrence of the given `value` or -1 if the `value` is not found. `null` is valid argument. The equality should be determined using the `equals` method. What is the average complexity of this method?

`void remove(int index);`
Removes element at specified `index` from collection. Element that was previously at location `index+1` after this operation is on location `index`, etc. Legal indexes are `0` to `size-1`. In case of invalid index throw

an appropriate exception (and document it!).

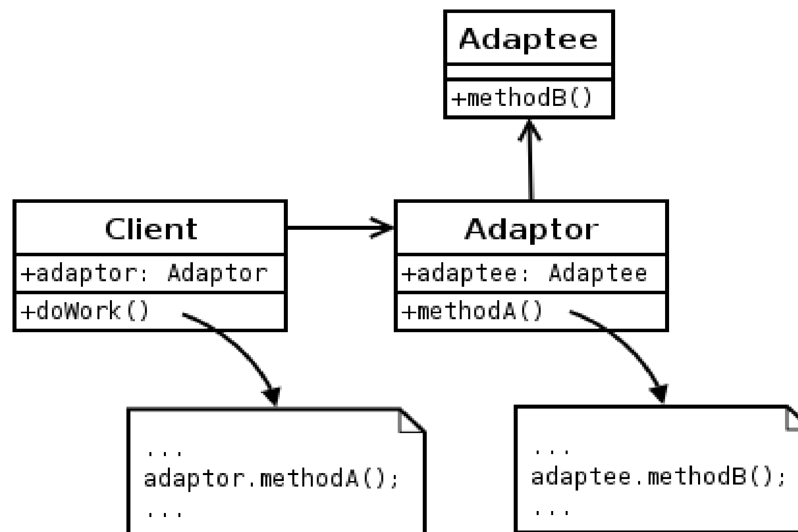
Example of usage for problems 1, 2 and 3 (you will have to `import java.util.Arrays;`):

```
ArrayIndexedCollection col = new ArrayIndexedCollection(2);
col.add(new Integer(20));
col.add("New York");
col.add("San Francisco");  // here the internal array is reallocated to 4
System.out.println(col.contains("New York")); // writes: true
col.remove(1); // removes "New York"; shifts "San Francisco" to position 1
System.out.println(col.get(1)); // writes: "San Francisco"
System.out.println(col.size()); // writes: 2
col.add("Los Angeles");

LinkedListIndexedCollection col2 = new LinkedListIndexedCollection(col);

// This is local class representing a Processor which writes objects to System.out
class P extends Processor {
  public void process(Object o) {
    System.out.println(o);
  }
};

System.out.println("col elements:");
col.forEach(new P());

System.out.println("col elements again:");
System.out.println(Arrays.toString(col.toArray()));

System.out.println("col2 elements:");
col2.forEach(new P());

System.out.println("col2 elements again:");
System.out.println(Arrays.toString(col2.toArray()));

System.out.println(col.contains(col2.get(1))); // true
System.out.println(col2.contains(col.get(1))); // true

col.remove(new Integer(20)); // removes 20 from collection (at position 0).
```

In order to solve this, consult lecture presentation, chapters 5 and 6 in book as well as the *Lesson: Exception* from the official *Java Tutorial* (see: http://docs.oracle.com/javase/tutorial/essential/exceptions/).

Problem 4.

Soon we will need an implementation of the *stack*-like collection. The collection `ArrayIndexedCollection` you already implemented could be used for that purpose; however, the interface (in a sense how users interact with it) of that collection is inappropriate. If the collection is a stack, you would expect it to have methods such as `push`, `pop` and `peek`, and not `insert`, `add` etc. which can be confusing for user. There is well known design pattern that can be employed to solve this mismatch: *Adapter pattern*[1] which is illustrated in the following figure.



In this case the *Adaptee* is the `ArrayIndexedCollection` class with its methods `add`, `insert` etc. It is the class with "wrong" interface toward the user. Your task will be to write `ObjectStack` class (it is top level class – it does not extends `Collection` class we defined previously) that is the *Adaptor* in used design pattern (place the class in the package from previous problem). This class must provide to user the methods which are natural for a stack and hide everything else. The `ObjectStack` class should provide the following methods:

`boolean isEmpty();` – same as `ArrayIndexedCollection.isEmpty()`

`int size();` – same as `ArrayIndexedCollection.size()`

`void push(Object value);` – pushes given value on the stack. `null` value must not be allowed to be placed on stack.

`Object pop();` – removes last value pushed on stack from stack and returns it. If the stack is empty when method `pop` is called, the method should throw `EmptyStackException`. This exception *is not part* of JRE libraries; you should provide an implementation of `EmptyStackException` class (put the class in the same package as all of collections you implemented and let it inherit from `RuntimeException`).

`Object peek();` – similar as `pop`; returns last element placed on stack but does not delete it from stack. Handle an empty stack as described in `pop` method.

`void clear();` – removes all elements from stack.

1 Please see: http://en.wikipedia.org/wiki/Adapter_pattern

The goal that `ObjectStack` should provide for it users appropriate interface but at the same time avoid code duplication will be accomplished by using *delegation* (remember this term). Each `ObjectStack` instance will create and manage its own private instance of `ArrayIndexedCollection` and use it for actual element storage. This way, the methods of `ObjectStack` will be the methods user expects to exist in stack, and those methods will implement its functionality by calling (i.e. delegating) methods of its internal collection of type `ArrayIndexedCollection`. The fact that our implementation of stack internally uses an instance of `ArrayIndexedCollection` is an implementation detail of which the final user is unaware. Additional benefit of this approach is the fact that actual implementation of element storage can be changed at any time (for example, we can decide to use `LinkedListIndexedCollection`) and without any consequences for clients of our stack class: we will not have to adjust or modify any of these clients – they are isolated from this change.

The methods `push` and `pop` should be implemented so that they have *o*(1) average complexity (except when the underlying array in used collection is reallocated).

Now create class `StackDemo` in subpackage `demo`. This should be command-line application which accepts a single command-line argument: expression which should be evaluated. Expression must be in postfix representation. When starting program from console, you will enclose whole expression into quotation marks, so that your program always gets just one argument (`args.length` should be `1` and the `args[0]` should be the whole expression).

Example 1: "8 2 /" means apply / on 8 and 2, so 8/2=4.
Example 2: "-1 8 2 / +" means apply / on 8 and 2, so 8/2=4, then apply + on -1 and 4, so the result is 3.

In expressions, you can assume that everything is separated by one (or more) spaces.
Each operator takes two preceding numbers and replaces them with operation result. You must support only +, -, /, * and % (remainder of integer division). All operators work with and produce integer results. So it is expected that 3/2=1. The calculation process can be solved by using the stack you just developed. Split the expression by spaces, and then do the following:

```
stack = empty
for each element of expression
    if element is number, push it on stack and continue
    else pop two elements from stack, perform operation and push result back on stack
end for
if stack size different from 1, write error
else syso stack.pop()
```

Ensure that you terminate the evaluation if user tries to divide by zero (write appropriate message to user; do not dump a stack trace on user). Also, if expression is invalid, write appropriate message to user.

Usage example:

```
D:\java> java -cp . hr.fer.zemris.java.custom.collections.demo.StackDemo "8 -2 / -1 *"
Expression evaluates to 4.
```

Please observe: the whole argument is enclosed in quotes so that it is given to your program as a single argument. It is your responsibility to split it.

Problem 5.

Implement a support for working with complex numbers. Your task is to create a class `ComplexNumber` which represents an unmodifiable complex number. Place the class in the package `hr.fer.zemris.java.hw02`. Each method which performs some kind of modification must return a new instance which represents modified number. This class must have:

- public constructor which accepts two arguments: *real* part and *imaginary* part (use `double` for both),
- public static factory methods:
  - `fromReal(double real): ComplexNumber`,
  - `fromImaginary(double imaginary): ComplexNumber`,
  - `fromMagnitudeAndAngle(double magnitude, double angle): ComplexNumber`,
  - `parse(String s): ComplexNumber` (accepts strings such as: "3.51", "-3.17", "-2.71i", "i", "1", "-2.71-3.15i"),
- public instance methods for information retrieval (the function should be clear for method names):
  - `getReal(): double`
  - `getImaginary(): double`
  - `getMagnitude(): double`
  - `getAngle(): double` (angle is in radians, from 0 to 2 Pi)
- public instance methods which allow calculations:
  - `add(ComplexNumber c): ComplexNumber`,
  - `sub(ComplexNumber c): ComplexNumber`,
  - `mul(ComplexNumber c): ComplexNumber`,
  - `div(ComplexNumber c): ComplexNumber`,
  - `power(int n): ComplexNumber`; n>=0,
  - `root(int n): ComplexNumber[]`; n > 0,
- public method for conversion to string:
  - `toString(): String`.

Create a subpackage `demo` and place inside program `ComplexDemo` with the following code in the `main` method.

```
ComplexNumber c1 = new ComplexNumber(2, 3);
ComplexNumber c2 = ComplexNumber.parse("2.5-3i");
ComplexNumber c3 = c1.add(ComplexNumber.fromMagnitudeAndAngle(2, 1.57))
                   .div(c2).power(3).root(2)[1];
System.out.println(c3);
```

Where needed, throw an appropriate exception. For parsing decimal numbers always use `Double.parseDouble(…)` so that decimal symbol is always dot ('.') and never comma (',').

You are expected to write junit tests for all methods of ComplexNumber class.

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). Additionally, for this homework you can not use any of Java Collection Framework classes which represent collections or its derivatives (its OK to use Arrays class if you find it suitable). Document your code!

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

You must write junit tests for problem 2 and problem 5. Junit tests for other problems are encouraged but not mandatory.

When your homework is done, pack it in zip archive with name `hw02-0000000000.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 March 22th 2018. at 07:00 AM.