2. homework assignment; JAVA, Academic year 2014/2015; FER

First: read page 8. I mean it! You are back? OK. 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 three 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...

Problem 1.

Write an implementation of resizable array-backed collection of objects denoted as `ArrayBackedIndexedCollection` and put it 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**; `null` references **are not allowed**.

You should provide *two* 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. Please implement the first constructor so that it delegates the construction process to second constructor (read section “Delegiranje zadaće konstrukcije objekta” in book, chapter 5).

The class should be equipped with following public methods.

`boolean isEmpty();` which returns `true` if collection contains no objects and `false` otherwise.

`int size();` which returns the number of currently stored objects in collections.

`void add(Object value);` which adds the given object into the 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 (`IllegalArgumentException`). What is the average complexity of this method?

`Object get(int index);` which 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 remove(int index);` which removes the object that is stored in the backing array at position `index`; since the collection must not hold `null` references, the content of the `elements` array which is at positions greater than `index` should be shifted one position down. What is the average complexity of this method?

`void insert(Object value, int position);` which **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`. 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);` which searches the collection and returns the index of the first occurrence of the given `value` or -1 if the `value` is not found. The equality should be determined using the `equals` method. What is the average complexity of this method?

`boolean contains(Object value);` which returns `true` only if the collection contains given `value`, as determined by `equals` method. What is the average complexity of this method?

`void clear();` which removes all elements from the collection. The allocated array is left at current capacity.

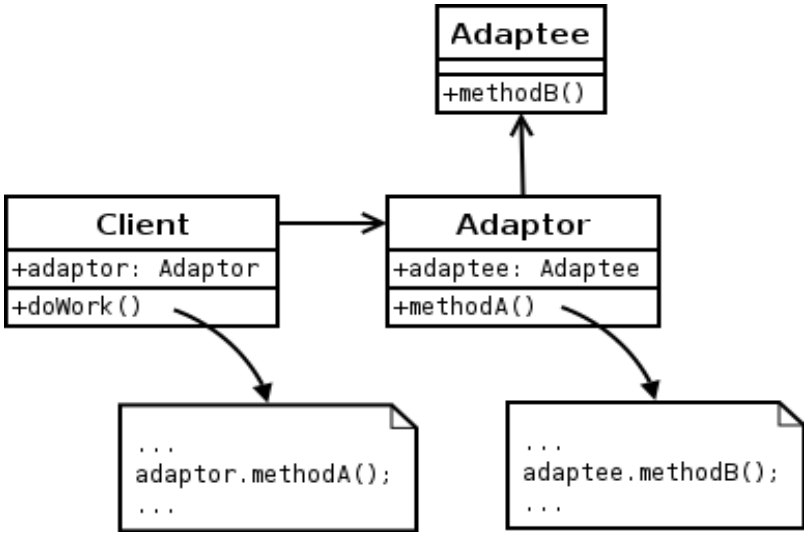
Example of usage:

```
ArrayBackedIndexedCollection col = new  ArrayBackedIndexedCollection(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
```

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 2.

To solve problem 3, you will need an implementation of the *stack* collection. The collection `ArrayBackedIndexedCollection` 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 `ArrayBackedIndexedCollection` 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 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 `ArrayBackedIndexedCollection.isEmpty()`

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

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*. Each `ObjectStack` instance will manage its own

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

private instance of `ArrayBackedIndexedCollection` 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 `ArrayBackedIndexedCollection`. The fact that our implementation of stack internally uses an instance of `ArrayBackedIndexedCollection` 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 and without any consequences for clients of our stack class: we will not have to adjust or modify 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.

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 precceding 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.
```

Problem 3.

Write two hierarchies of classes: *tokens* and *nodes*. Place the classes into packages `hr.fer.zemris.java.custom.scripting.tokens` and `hr.fer.zemris.java.custom.scripting.nodes` respectively. *Nodes* will be used for representation of structured documents. *Tokens* will be used to for the representation of expressions.

`Token` hierarchy

`Token` – base class having only a single public function: `String asText();` which for this class returns an empty String.

`TokenVariable` – inherits `Token`, and has a single read-only[2] `String` property: `name`. Override `asText()` to return the value of `name` property.

`TokenConstantInteger` – inherits `Token` and has single read-only `int` property: `value`. Override `asText()` to return string representation of `value` property.

`TokenConstantDouble` – inherits `Token` and has single read-only `double` property: `value`. Override `asText()` to return string representation of `value` property.

`TokenString` – inherits `Token` and has single read-only `String` property: `value`. Override `asText()` to return `value` property.

`TokenFunction` – inherits `Token` and has single read-only `String` property: `name`. Override `asText()` to return `name` property.

`TokenOperator` – inherits `Token` and has single read-only `String` property: `symbol`. Override `asText()` to return `symbol` property.

`Node` hierarchy

`Node` – base class for all graph nodes.

`TextNode` – a node representing a piece of textual data. It inherits from `Node` class.

`DocumentNode` – a node representing an entire document. It inherits from `Node` class.

`ForLoopNode` – a node representing a single for-loop construct. It inherits from `Node` class.

`EchoNode` – a node representing a command which generates some textual output dynamically. It inherits from `Node` class.

Lets assume that we work with following text document:

```
This is sample text.
{$ FOR i 1 10 1 $}
  This is {$= i $}-th time this message is generated.
```

2 If class has property Prop, this means that it has private instance variable of the same name and the public getter method (getProp()) and the public setter method (setProp(value)). If property is read-only, no setter is provided. If property is write-only, no getter is provided. For read-only properties, use constructor to initialize it.

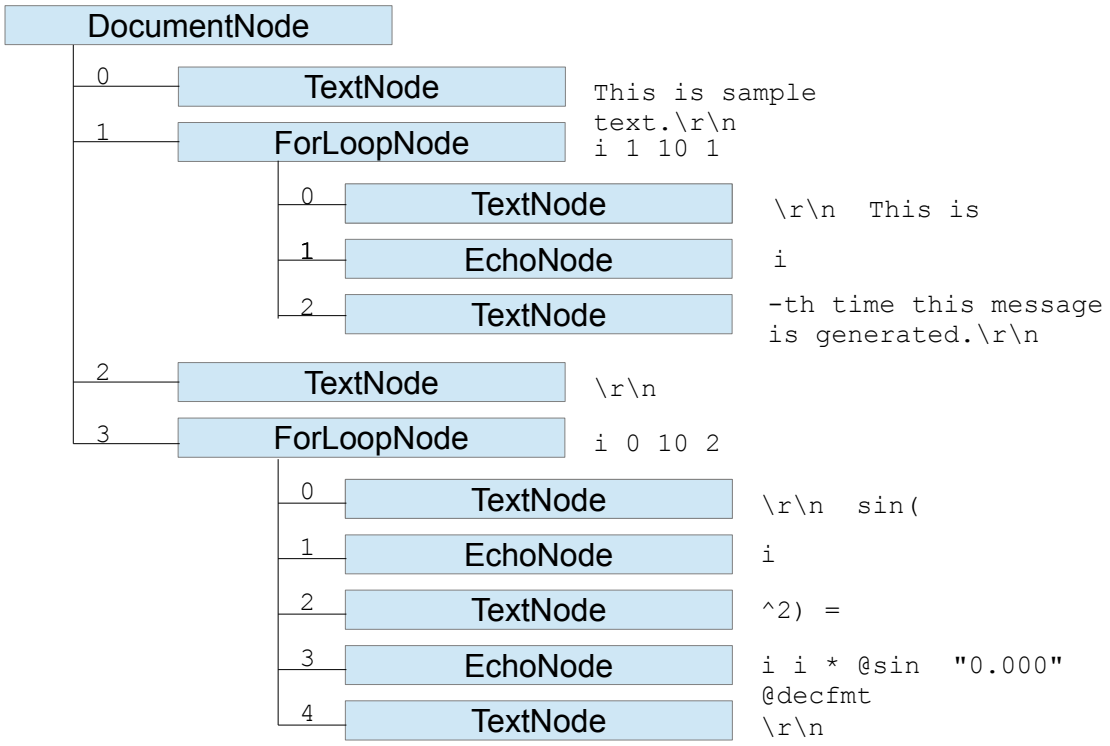
```
{$END$}
{$FOR i 0 10 2 $}
  sin({$=i$}^2) = {$= i i * @sin  "0.000" @decfmt $}
{$END$}
```

This document consists of tags (bounded by `{$` and `$}`) and rest of the text. Reading from top to bottom we have:

text	`This is sample text.\r\n`	1
tag	`{$ FOR i 1 10 1 $}`	2
text	`\r\n  This is`	3
tag	`{$= i $}`	4
text	`-th time this message is generated.\r\n`	5
tag	`{$END$}`	6
text	`\r\n`	7
tag	`{$FOR i 0 10 2 $}`	8
text	`\r\n  sin(`	9
tag	`{$=i$}`	10
text	`^2) =`	11
tag	`{$= i i * @sin  "0.000" @decfmt $}`	12
text	`\r\n`	13
tag	`{$END$}`	14

Observe that spaces in tags are ignorable; `{$END$}` means the same as `{$ END  $}`. Each tag has its name. The name of `{$ FOR … $}` tag is `FOR`, and the name of `{$= … $}` tag is `=`. Tag names are case-insensitive. This means that you can write `{$ FOR … $}` or `{$ For … $}` or `{$ foR … $}` or similar. A one or more spaces (tabs, enters or spaces – we will treat them equally) can be included before tag name, so all of the following is also OK: `{$FOR … $}` or `{$ FOR … $}` or `{$ FOR … $}`. `=`-tag is an empty tag – it has no content so it does not need closing tag. `FOR`-tag, however, is not an empty tag. Its has content and an accompanying `END`-tag must be present to close it. For example, the content of the `FOR`-tag opened in the line 2 in above table comprises two texts and a tag given in lines 3, 4 and 5. Since `END`-tag is only here to help us close nonempty tags, it will not have its own representation.

The Document model built from this document looks as follows.



Class `Node` defines methods:

`void addChildNode(Node child);` – adds given `child` to an internally managed collection of children; use an instance of `ArrayBackedIndexedCollection` class for this. However, create this collection only when actually needed (i.e. create an instance of the collection on demand).

`int numberOfChildren();` – returns a number of (direct) children. For example, in above example, instance of `DocumentNode` would return 4.

`Node getChild(int index);` – returns selected child or throws an appropriate exception if the index is invalid.

All other node-classes inherit from `Node` class.
Class `TextNode` defines single additional read-only String property `text`.

Class `ForLoopNode` defines several additional read-only properties:

- property `variable` (of type `TokenVariable`)
- property `startExpression` (of type `Token`)
- property `endExpression` (of type `Token`)
- property `stepExpression` (of type `Token`, which can be `null`)

Class `EchoNode` defines a single additional read-only `Token[]` property `tokens`.

As you can see, `ForLoopNode` and `EchoNode` work with instances of `Token` (sub)class. Lets take a look on =-tag from our example:

```
{$= i i * @sin  "0.000" @decfmt $}
```

Arguments (parameters) of this tag are:

- two times `TokenVariable` with `name="i"`
- once `TokenOperator` with `symbol="*"`
- once `TokenFunction` with `name="sin"`
- once `TokenString` with `value="0.000"`
- once `TokenFunction` with `name="decfmt"`

Implement a parser for described structured document format. Implement it as single class `SmartScriptParser` and put it in the package `hr.fer.zemris.java.custom.scripting.parser`. The parser should have a single constructor which accepts a string that contains document body. The constructor should then delegate the parsing to separate method (in the same class) that will perform actual job. This will allow us to later add different constructors that will retrieve documents by various means and delegate the parsing to the same method. Create a class `SmartScriptParserException` (derive it from `RuntimeException`) and place it in the same package as `SmartScriptParser`. If any exception occurs during parsing, parser should catch it and rethrow an instance of this exception.

Please observe that tag `ForLoopNode` can have three or four parameters (as specified by user): first it must have one `TokenVariable` and after that two or three `Token`s. If user specifies something which does not obeys this rule, throw an exception. Here are several good examples:

```
{$ FOR i -1 10 1 $}
{$ FOR sco_re "-1" 10 "1" $}
{$ FOR year 1 last_year $}
```

and here are several bad examples (for which an exception should be thrown):

```
{$ FOR 3 1 10 1 $}
{$ FOR * "1" -10 "1" $}
{$ FOR year @sin 10 $}
{$ FOR year 1 10 "1" $}
{$ FOR year $}
{$ FOR year 1 10 1 3 $}
```

Valid variable name starts by letter and after follows zero or more letters, digits or underscores. If name is not valid, it is invalid. This variable names are valid: `A7_bb`, `counter`, `tmp_34`; these are not: `_a21`, `32`, `3s_ee` etc.

Valid function name starts with `@` after which follows a letter and after than can follow zero or more letters, digits or underscores. If function name is not valid, it is invalid.

Valid tag names are "=", or variable name. So = is valid tag name (but not valid variable name).

In strings (*and only in strings*!) parser must accept following escaping:
`\\` sequence treat as a single string character `\`
`\"` treat as a single string character `"` (and not the end of the string)
`\n`, `\r` and `\t` have its usual meaning (ascii 10, 13 and 9).
Every other sequence which starts with `\` should be treated literally as is written.
For example, `"Some \\ test \* X"` should be interpreted as string with value `Some \ test \* X`.
Another example: `"Joe \"Long\" Smith"` represents a single string with value `Joe "Long" Smith`.

In document text (i.e. outside of tags) parser must accept only the following two escaping:
`\\` treat as `\`
`\{` treat as `{`
Every other sequence which starts with `\` should be treated literally as is written.

For example, document whose content is following:

```
Example \{$=1$}. Now actually write one {$=1$}
```

should be parsed into only three nodes:

```
DocumentNode
  *
  *- TextNode with value Example {$=1$}. Now actually write one
  *- EchoNode with one token
```

Implementation hint. As help for tree construction use `ObjectStack`. At the beginning, push `DocumentNode` to stack. Then, for each empty tag or text node create that tag/node and add it as a child of `Node` that was last pushed on the stack. If you encounter a non-empty tag (i.e. `FOR`-tag), create it, add it as a child of `Node` that was last pushed on the stack and than push this `FOR`-node to the stack. Now all nodes following will be added as children of this `FOR`-node; the exception is `{$END$}`; when you encounter it, simple pop one entry from the stack. If stack remains empty, there is error in document – it contains more `{$END$}`-s than opened non-empty tags, so throw an exception.

During the tag construction, you do not have to consider whether the provided tags are meaningful. For example, in tag:

```
{$= i i * @sin  "0.000" @decfmt $}
```

you do not have to think about is it OK that after two variables `i` comes the `*`-operator. You task for now is just to build the accurate document model which represents the document **as provided by the user**. At some later time we will consider whether that which user gave us is actually legal or not.

Developed parser should be used as illustrated by the following scriptlet:

```
String docBody = "....";
SmartScriptParser parser = null;
try {
  parser = new SmartScriptParser(docBody);
} catch(SmartScriptParserException e) {
  System.out.println("Unable to parse document!");
  System.exit(-1);
} catch(Exception e) {
  System.out.println("If this line ever executes, you have failed this class!");
  System.exit(-1);
}
DocumentNode document = parser.getDocumentNode();
String originalDocumentBody = createOriginalDocumentBody(document);
System.out.println(originalDocumentBody); // should write something like original
                                          // content of docBody
```

Create a main program named `SmartScriptTester` and place it in package `hr.fer.zemris.java.hw2`. In the main method put the above-shown scriptlet. Let this program accepts a single command-line argument: path to document. You can read the content of this file by following code:

```
import java.nio.file.Files;
import java.nio.charset.StandardCharsets;
import java.nio.file.Paths;
String docBody = new String(
```

```
                    Files.readAllBytes(Paths.get(filepath)),
                    StandardCharsets.UTF_8
);
```

In your project create directory examples and place inside at least doc1.txt which contains the example given in this document. You are free to add more examples.

Implement all needed methods in order to ensure that the program works.

The method `createOriginalDocumentBody` does not have to reproduce the exact original documents, since this is impossible: after the parsing is done you have lost the information how the tokens were separated (by one or more spaces, tabs, etc. and similar). But it must reproduce something which will after parsing again result with the same document model! So this is the actual test:

```
String docBody = "....";
SmartScriptParser parser = new SmartScriptParser(docBody);
DocumentNode document = parser.getDocumentNode();
String originalDocumentBody = createOriginalDocumentBody(document);
SmartScriptParser parser2 = new SmartScriptParser(originalDocumentBody);
DocumentNode document2 = parser2.getDocumentNode();
// now document and document2 should be structurally identical trees
```

Very important: you *do not have to* develop an engine that will “execute” this document (iterate for-loop for specified number of iterations etc). All you have to do at this point is write a piece of code that will produce a document tree model.

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 or its derivatives. Document your code!

In order to solve this homework, create a blank Eclipse Java Project and write your code inside. You must name your project's main directory (which is usually also the project name) `HW02-`*yourJMBAG*; for example, if your JMBAG is 0012345678, the project name and the directory name must be `HW02-0012345678`. 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 March 25nd 2015. at 11:59 PM.