1. homework assignment; JAVA, Academic year 2012/2013; FER

First: read page 9. I mean it! You are back? Now read page 8 and come back here. OK. 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. So, 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 collections,
- `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: 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.

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`).

`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`).

`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.

`void insert(Object value, int position);` which inserts the given `value` at given `position` in array. 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`.

`int indexOf(Object value);` which searches the collection and return the index of the first occurrence of given `value` or -1 if `value` is not found. The equality should be determined using the `equals` method.

`boolean contains(Object value);` which returns `true` only if the collection contains given value, as determined by equals method.

`void clear();` which removes all elements from collection.

Example of usage:

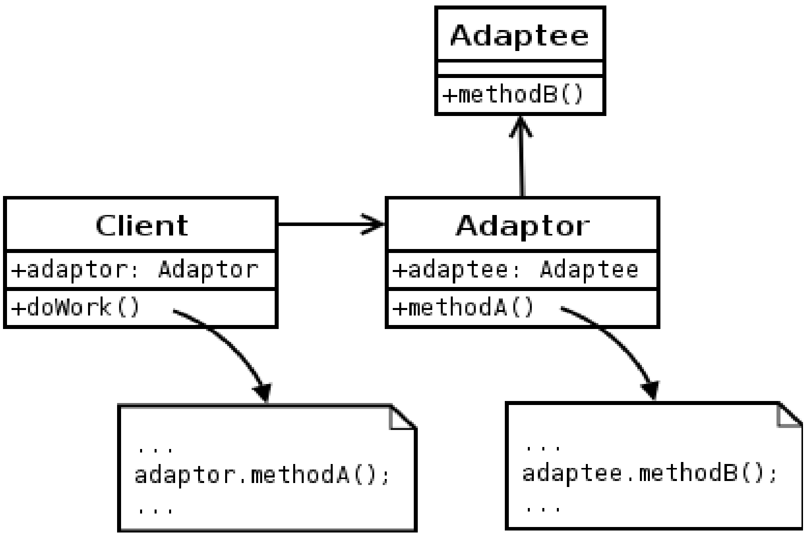
```
ArrayBackedIndexedCollection col = new  ArrayBackedIndexedCollection(2);
col.add(new Integer(20));
col.add("New York");
col.add("San Francisco");  // here the internal array is reallocated
System.out.println(col.contains("New York")); // writes: true
col.remove(1); // removes "New York"; shifts "San 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 as well as the *Lesson: Exception* from the official *Java Tutorial* (see: http://docs.oracle.com/javase/tutorial/essential/exceptions/). We will discuss this during next lecture as well.

Create a separate directory hierarchy for unit tests. Write appropriate tests that will cover all of the implemented methods.

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. 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. Your task will be to write `ObjectStack` class that is the *Adaptor* in used design pattern and which provides methods that are natural for a stack and nothing else. The `ObjectStack` class should provide following

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

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 inherit it from `RuntimeException`).

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

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

Each `ObjectStack` instance should manage its own private instance of `ArrayBackedIndexedCollection` and use it for actual element storage. This way, methods of `ObjectStack` will adapt the interface this class provides toward the user and in the background delegate the actual work to an instance of `ArrayBackedIndexedCollection` of which is final user unaware. Additional benefit of this approach is the fact that actual implementation of element storage can be changed at any time without clients knowledge and without the need to adjust or modify clients.

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

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 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

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.

`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 text 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 $]-th time this message is generated.
[$END$]
[$FOR i 0 10 2 $]
  sin([$=i$]^2) = [$= i i * @sin  "0.000" @decfmt $]
[$END$]
```

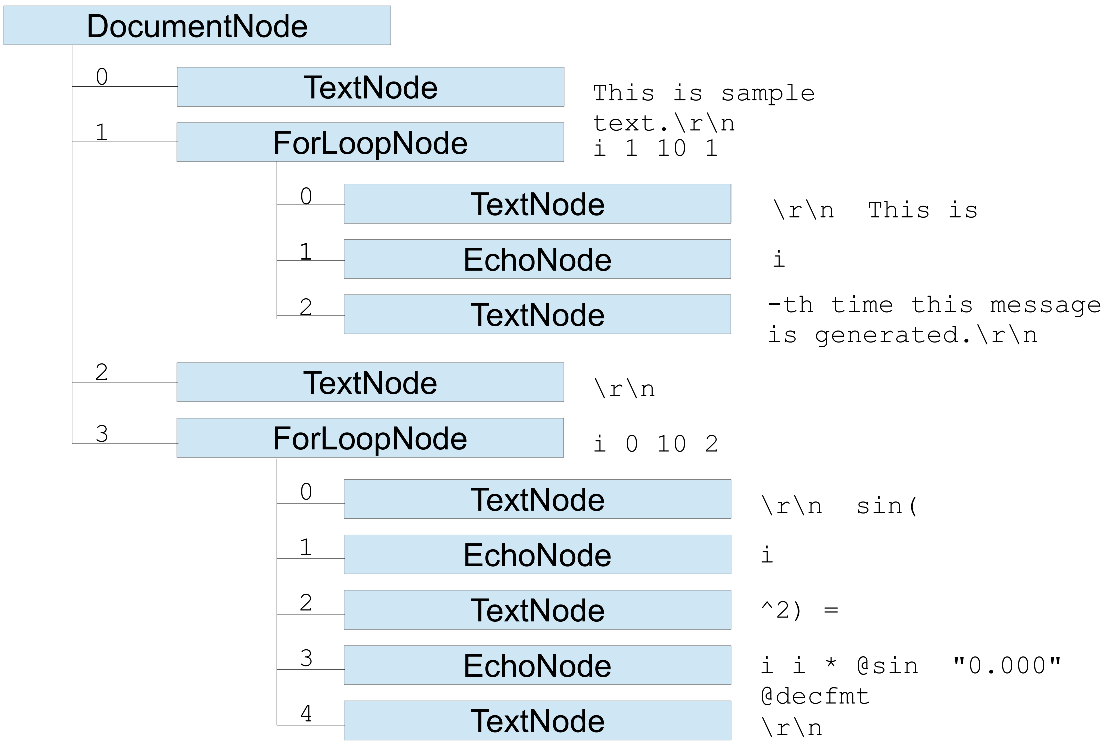
This document consists of tags (bounded by `[$` and `$]`) and rest of the text. Reading from top to bottom we have:

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 … $]` or similar. A one or more spaces can be included before tag name, so all of the following is also OK: `[$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 a collection only when needed.

`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.

Valid name of variable 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.

In strings (*and only in strings*!) parser must accept following escaping:
`\\` treat as `\`
`\"` treat as `"`
`\n`, `\r` and `\t` have its usual meaning (ascii 10, 13 and 9).
For example, `"Joe \"Long\" Smith"` represents a single string whose value is `Joe "Long" Smith`.

In text (i.e. outside of tags) parser must accept following escaping:
`\[` treat as `[`

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
```

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.

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.hw1`. In the main method put the above-shown scriptlet; as `docBody` use document from the example in this document. Implement all needed methods in order to ensure that the program works.

Important: you do not have to develop 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.

Important notes

Solve all of the problems in a single Eclipse project. Configure Eclipse to use two source directories: `src/main/java` for your source files and `src/test/java` for sources files of unit tests.

You are required to write the adequate number of unit tests for all of the classes developed in problem 1 and problem 2.

You must equip your project with `build.xml` script so that the project can be build from the command line. In that script, you must integrate all of the quality-checking tools which I have described in the book. It should be possible to run at least the following targets: `init`, `compile`, `compile-tests`, `run-tests`, `quality`, `reports`, `clean`.

Target `quality` must run unit tests and all of the quality checks (i.e. *checkstyle*, *pmd*, *findbugs*). Unit tests must be run with the code coverage analysis. Target `reports` is a wrapper that will run all of the unit tests, the quality checks and the javadoc generation.

All of the classes in all three problems should have appropriate javadoc.

Considering source quality reports, you can ignore warnings that after `if`, `for` etc. a space is mandatory; you can write in your source: `if(` and you don't have to write `if (`. Also, you can ignore the warnings that method arguments and some variables should be declared final. It is OK to have code written as follows:

```
public int successorSquare(int n) {
   int succ = n+1;
   return succ*succ;
}
```

To make quality-check tools happy, you could rewrite it as follows (but you do not have to):

```
public int successorSquare(final int n) {
   final int succ = n+1;
   return succ*succ;
}
```

Configuring ant build script

In order to get portable build script, you should not define paths to additional tools directly in `build.xml` file (for example, variables such as `checkstyle.home`, `pmd.home` etc). Instead, create a new file `config.properties` in the same directory as `build.xml`. It must be a text file which defines all of the necessary home variables. Here is a sample content:

```
checkstyle.home=d:/usr/checkstyle-5.6
pmd.home=d:/usr/pmd-bin-5.0.2
findbugs.home=d:/usr/findbugs-2.0.2
junit.home=d:/usr/junit-4.11
jacoco.home=d:/usr/jacoco-0.6.3
xalan.home=d:/usr/xalan-j_2_7_1
```

Then delete in `build.xml` declarations for those variables (*and only that!*) and just add the following line:

```
<property file="config.properties"/>
```

Now, ant will read file `config.properties` and will define variables as defined in that file.

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. 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.