1DV507, Programming and Data Structures, Spring 2018

Assignment 2: Data Structures, JUnit, and JavaFX (1)

Problems?
Do not hesitate to ask your teaching assistant at the practical meetings (or Jonas at the lectures) if you have any problems. You can also post a question in the assignment forum in Moodle.

Prepare Eclipse for course 1DV507 and Assignment 2
Inside your Java project named 1DV507, create a new *package* with the name `YourLnuUserName_assign2` and save all program files for this assignment inside that package. Later on, when submitting your assignment, you should submit a zipped version of this folder/package.

General Assignment Rules

- Use English! All documentation, names of variables, methods, classes, and user instrutions, should be in English.
- Each exercise that involves more than one class should be in a separate package with a suitable (English!) name. For example, in Exercise 1, create a new sub package named `queue` inside your package `YourLnuUserName_assign2` and save all .java files related to this exercise inside this package.
- All programs asking the user to provide some input should check that the user input is correct and take appropriate actions if it is not.

Lecture 4 - Simple Data Structures

- **Exercise 1**

  A Queue is a FIFO (first in, first out) data structure. Consider the following queue interface:

```
public interface Queue {
   public int size();                     // current queue size
   public boolean isEmpty();              // true if queue is empty
   public void enqueue(Object element);   // add element at end of queue
   public Object dequeue();               // return and remove first element.
   public Object first();                 // return (without removing) first element
   public Object last();                  // return (without removing) last element
   public String toString();              // return a string representation of the queue content
   public Iterator<Object> iterator();    // element iterator
}
```

  The iterator iterates over all elements of the queue. Operations not allowed on an empty queue shall generate an unchecked exception.

  Tasks:
  - Create a *linked* implementation `LinkedQueue.java` of the interface Queue. Use the *head-and-tail* approach.
  - Write also a program `QueueMain.java` showing how all methods work.
  - Create Javadoc comments in the code and generate good-looking and extensive HTML documentation for the interface and the class. All public class members shall be documented.

Notice:
- The implementation shall be linked, i.e. a sequence of linked nodes where each node represents an element.
- You are not allowed to use any of the predefined collection classes in the Java library.
- In the report, the HTML pages generated by the classes `Queue` and `LinkedQueue` shall be attached. Attach no other HTML pages!

- **Exercise 2 (VG Task)**
  A straight forward arrray based implementation of the Queue interface above would use an Object array (that grows on demand) and two indices `first` and `last` to keep track of the array positions where to remove an element on `dequeue` (return and increase position `first`), and where to add an element on `enqueue` (insert at and increase position `last`). The problem with this approach is that after (say) 100 `dequeue` we will have that `first = 100` and 100 non-used elements (positions 0 to 99) that never will be used again. That is, a waste of memory.

  Your task is to provide an array based Queue implementation (`ArrayQueue`) that avoids this problem by treating the array like a circular array in which array indices larger than the array size ``wrap around'' to the beginning of the array. That is,

  - When, after a number of enqueues, index `last` reaches `array.length`, you should move `last` to position 0 and start to reuse the first part of the array.
  - Later on, after an even larger number of dequeues, you will reach the point where index `first` reaches `array.length`. Then, move `first` to position 0 (and you have returned to the initial configuration where all the queue elements are stored between `first` and `last`).
  - Finally, the array is full when `last`, after one or more ``wrap arounds'', reaches `first`. In that case you should resize the array and restore the order such that `first` equals 0.

  This approach is called a queue implementation based on *circular arrays*. Look it up on the Internet to get all details.

Lecture 5 - JUnit Testing

Important: You must use the new JUnit 5. Solutions based on older versions will not be accepted.

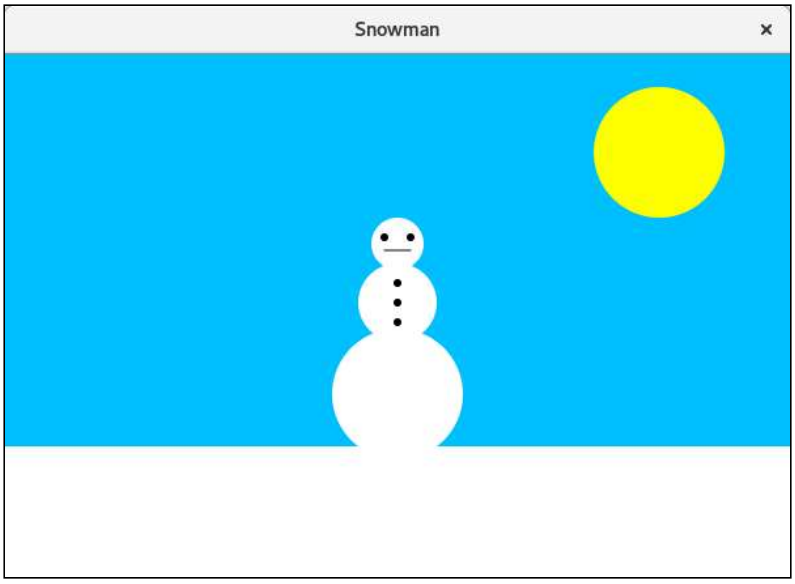
- **Exercise 3**
  Write a JUnit test for the class `LinkedQueue` in Exercise 1.

- **Exercise 4** (VG Task)
  Modify the Queue test in Exercise 3 so that it can also handle the `ArrayQueue` class from Exercise 2. We do not want two separate tests. We want one test that can be used for any implementation of the `Queue` interface *with just a minimum of modifications*.

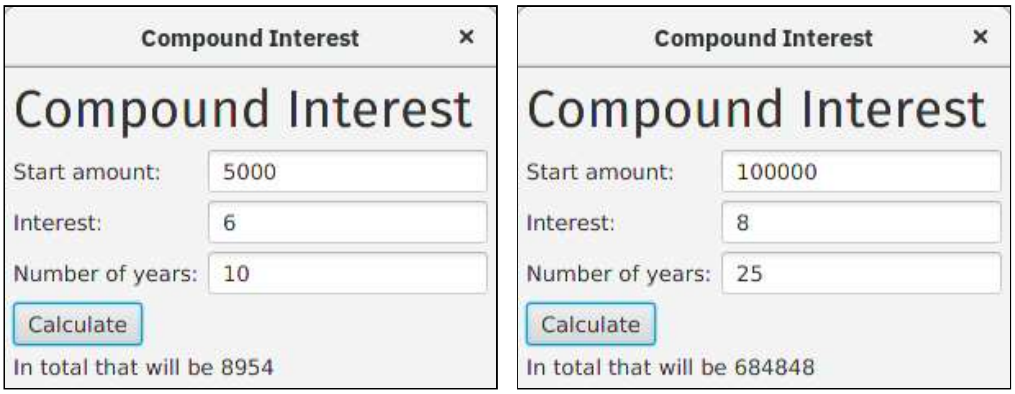
Lecture 6 - JavaFX (Part 1)

Important: You are not allowed to use any GUI builder tools in these assignments. All your code should be written by you, not generated by a tool.

- **Exercise 5**
  Write a program `Snowman.java` which creates a JavaFX Application window containing a drawing of a snowman. It should be drawn by you using the vector graphics primitives in JavaFX and not a static image. The snowman must not look *exactly* as in the image below, but it must consist of three differently sized circles with three buttons, stand in snow, have a face and a sun in the top. You are allowed to add to the snowman, for example a nice hat, a happy smile or a carrot nose.



- **Exercise 6**
  Create a class `CompoundInterest.java` that calculates a compound interest, for example for a loan or saving. You can read more about compound interest on Wikipedia. The program should allow the user to input an amount of money (in no particular currency), an interest and a number of years. At the press of the button it should calculate the total sum of money that corresponds to with compound interest.



- **Exercise 7**
  Write a GUI program `Yahtzee.java` that will be the *beginning* of a game of Yahtzee. The program should display five dice (you can find graphics on the Internet) and allow the user to roll the dice three times. For each throw, the user should be able to hold one or more of the dice, i.e. only rethrow the others. Do this by displaying check boxes below the each die, and it should only be possible to tick the box once the first throw has been made (the check boxes should be set to disabled when the program starts). The program only needs to work once, so when three throws are done, you need to restart the program to try again. When all three thows are done, the program should display if the final dice are showing one of the following: Yahtzee, Four of a kind, Three of a kind, Full House, Large Straight, Small Straight or Pair. Check the dice in that order, only one of them needs to be displayed. An example run of the game is shown below.

Figure 1

Figure 2

Figure 3

Figure 4

Submission

All exercises should be handed in and we are only interested in your .java files. (Notice that the VG exercises 2 and 4 are not mandatory.) Hence, zip the directory named `YourLnuUserName_assign2` (inside directory named `src`) and submit it using the Moodle submission system. Make sure to also include any images or icons that you might have used in the JavaFX exercises.