1DV507, Programming and Data Structures, Spring 2018

Assignment 3: Algorithms, Hashing and BSTs, and GUI (Part 2)

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 3
Inside your Java project named 1DV507, create a new *package* with the name `YourLnuUserName_assign3` 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 instructions, 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 2, create a new sub package named `sort` inside your package `YourLnuUserName_assign3` 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 7 - Algorithms (1)

- **Exercise 1**
  Every Computer Science course involving algorithms should include the *Euclidean algorithm*!
  The Euclidean algorithm is an algorithm to decide the greatest common divisor of two positive integers. The greatest common divisor of N and M, in short GCD(M,N), is the largest integer X such as M and N are evenly dividable with X. Some examples:

  ```
  GCD(18,12) = 6
  GCD(42,56) = 14
  GCD(9,28) = 1
  ```

  Write a program `EuclideanMain` that takes two integers as input and calculates (and presents) their GCD using this algorithm.

  More information about the Euclidean algorithm can be found on the Internet. In Swedish: *Euklides algoritm*

- **Exercise 2 (50% VG Exercise!)**

  This exercise is about sorting arrays of integer and strings. Create one class `SortingAlgorithms` containing four static methods:

```
public int[] insertionSort(int[] in)
public int[] mergeSort(int[] in)         // VG Exercise
public String[] insertionSort(String[] in, Comparator<String> c)
public String[] mergeSort(String[] in, Comparator<String> c)   // VG Exercise
```

All methods return a new sorted array where all elements are sorted in ascending order (lowest first) in the first two integer array methods, as defined by the Comparator in the final two String array methods. The input arrays `in` should not be changed. The methods should of course use the *Insertion Sort* and *Merge Sort* algorithms.

You should also provide a JUnit 5 test class `SortTest` that tests each method.

Notice:

- Our suggestion is that you start to implement (and test) the integer versions.
- Both algorithms are described in the lecture slides and in the textbook by Horstmann. You can also find plenty of information on the Internet.
- The textbook (and Internet) also describes how to implement these algorithms (integer version) in Java. Feel free to take any such implementation as your starting point. However, in this case you must clearly state in your assignments submission which implementation you have used. Provide a web site if you have taken it from the Internet.

Lecture 8 - Hashing and Binary Search Trees

The following five exercises are actually one large exercise named *Count Words*. We have divided Count Words into smaller steps, Exercises 3 - 7, for simplicity. What we want you to do is to count the number of different words in the text HistoryOfProgramming.txt by adding all "words" to a set. We will use four different set implementations: two predefined from the Java library and two that you will implement by yourselves.

Notice: All files related to *Count Words* should be saved in a package named `count_words`.

- **Exercise 3**
  Write a program `IdentyfyWordsMain` that reads a text file(like HistoryOfProgramming) and divide the text into a sequence of words (word = sequence of letters). All non-letters (except whitespace) should be removed. Save the result in a new file (words.txt). Example:

```
Text
====
Computer programming, History of programming
From Wikipedia, the free encyclopedia (081110)

The earliest known programmable machine (that is a machine whose
behavior can be controlled by changes to a
"program") was Al-Jazari's programmable humanoid robot in 1206.

Sequence of words
=================
Computer programming History of programming
From Wikipedia the free encyclopedia
The earliest known programmable machine that is a machine whose
behavior can be controlled by changes to a
program was Al Jazaris programmable humanoid robot in
```

  All exceptions related to file handling shall be handled within the program.

- **Exercise 4**

  Create a class Word, representing a word. Two words should be considered equal if they consist of the same sequence of letters and we consider upper case and lower case as equal. For example hello, Hello and HELLO are considered to be equal. The methods equals and hashCode define the meaning of "equality". Thus, the class Word should look like the following.

```
public class Word implements Comparable<Word> {
   private String word;

   public Word(String str) { ... }
   public String toString() { return word; }

   /* Override Object methods */
   public int hashCode() { ... compute a hash value for word }
   public boolean equals(Object other) { ... true if two words are equal }

   /* Implement Comparable */
   public int compareTo(Word w) { ... compares two words lexicographically }
}
```

  Note:
  - □ If you want, you can add more methods. The methods mentioned above are the minimum requirement.
  - Exercise 5 and onward is based on Exercise 4. Thus, carefully test all methods before proceeding.

- **Exercise 5**

  Create a program WordCount1Main doing the following:

  □For each word in the file word.txt

  1. Create an object of the class Word
  2. Add the object to a set of the type java.util.HashSet
  3. Add the object to a set of the type java.util.TreeSet

  Note:

  1. □ The size of the sets should correspond to the number of different words in the files. (Our tests gave 350 words for the file HistoryOfProgramming)
  2. □ An iteration over the words in the TreeSet should give the words in alphabetical order.
  3. Since our defintion of a word is not very precise (similar to the WarAndPeace exercise in Assignment 2), we do not expect all of you to end up with exactly 350 words. But it should be rather close.

- **Exercise 6**

  Given the following interface

```
public interface WordSet extends Iterable {
   public void add(Word word); // Add word if not already added
   public boolean contains(Word word); // Return true if word contained
   public int size(); // Return current set size
   public String toString(); // Print contained words
}
```

Implement the interface using a)□ Hashing, b) Binary Search Tree. In the case of hashing, a rehash shall be performed when the number of inserted elements equals the number of buckets. For the binary search tree, the elements shall be sorted using the method compareTo. The names of the two implementations shall be `HashWordSet` and `TreeWordSet`.

Note: You are not allowed to use any predefined collection classes from the Java library. However, you are allowed to use arrays.

- **Exercise 7**

  Repeat Exercise 5 with the new implementations HashWordSet and TreeWordSet. The program shall be called `WordCount2Main`. The two notes of Exercise 5 should still be valid.

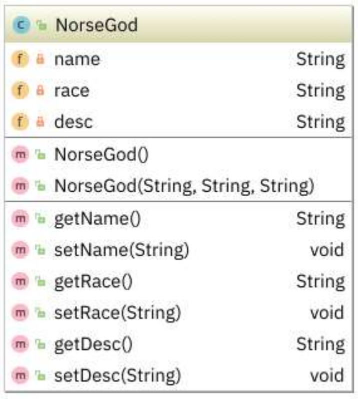
Lecture 9 - JavaFX (Part 2)

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 8**

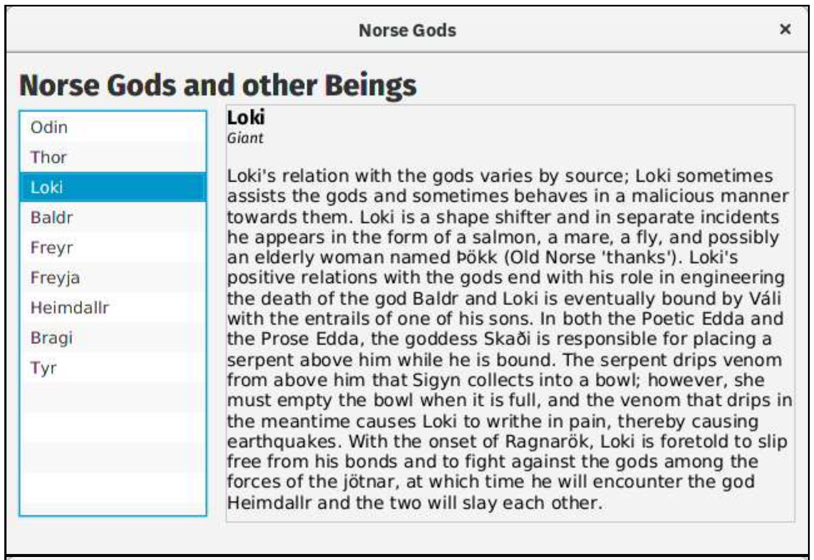
In the picture below there is a class in UML for **NorseGod**s. In this task you are going to create this class as well as a graphical user interface for showing information on the different gods by selecting them from a list.

When the program starts an **ArrayList** of these gods should be contructed and populated with *at least* eight Norse gods (or other beings from the Norse mythology). The information stored is the name of the god, its race (aesir, vanir, giant or other) as well as a description of the god (or being). The information on the gods and beings themselves can be copied and pasted from Wikipedia.



NorseGod	
name	String
race	String
desc	String
NorseGod()	
NorseGod(String, String, String)	
getName()	String
setName(String)	void
getRace()	String
setRace(String)	void
getDesc()	String
setDesc(String)	void

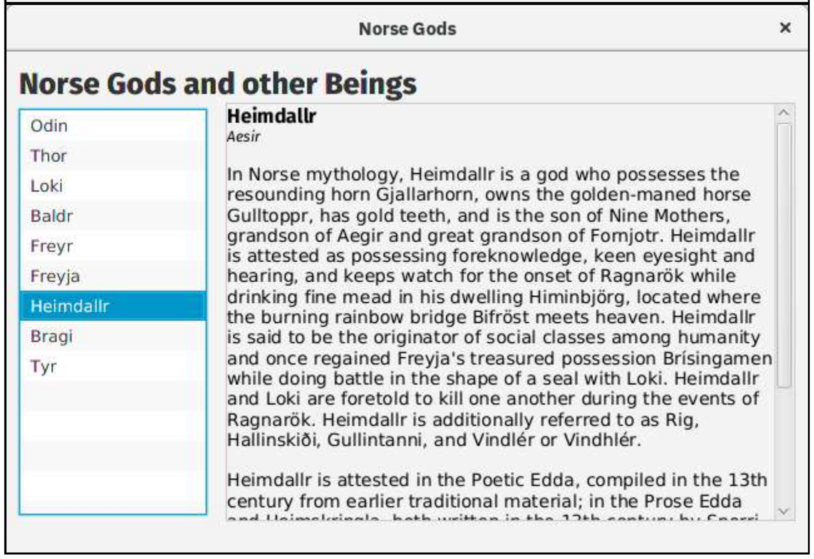
The final program should look something like in the pictures below, but you are free to change the GUI as long as the same functionality exists. The GUI shown has been created using a **BorderPane** for layout and for the text the class **TextFlow** has been used (this is not a requirement, several **Text** or **Label** classes are possible to use as well). A requirement is that the text should be scrollable if too much is shown in the window and therefore you need to look at the class **ScrollPane**.



Norse Gods and other Beings

Loki
Giant

Loki's relation with the gods varies by source; Loki sometimes assists the gods and sometimes behaves in a malicious manner towards them. Loki is a shape shifter and in separate incidents he appears in the form of a salmon, a mare, a fly, and possibly an elderly woman named Þökk (Old Norse 'thanks'). Loki's positive relations with the gods end with his role in engineering the death of the god Baldr and Loki is eventually bound by Váli with the entrails of one of his sons. In both the Poetic Edda and the Prose Edda, the goddess Skaði is responsible for placing a serpent above him while he is bound. The serpent drips venom from above him that Sigyn collects into a bowl; however, she must empty the bowl when it is full, and the venom that drips in the meantime causes Loki to writhe in pain, thereby causing earthquakes. With the onset of Ragnarök, Loki is foretold to slip free from his bonds and to fight against the gods among the forces of the jötnar, at which time he will encounter the god Heimdallr and the two will slay each other.



Norse Gods and other Beings

Heimdallr
Aesir

In Norse mythology, Heimdallr is a god who possesses the resounding horn Gjallarhorn, owns the golden-maned horse Gulltoppr, has gold teeth, and is the son of Nine Mothers, grandson of Aegir and great grandson of Fornjotr. Heimdallr is attested as possessing foreknowledge, keen eyesight and hearing, and keeps watch for the onset of Ragnarök while drinking fine mead in his dwelling Himinbjörg, located where the burning rainbow bridge Bifröst meets heaven. Heimdallr is said to be the originator of social classes among humanity and once regained Freyja's treasured possession Brísingamen while doing battle in the shape of a seal with Loki. Heimdallr and Loki are foretold to kill one another during the events of Ragnarök. Heimdallr is additionally referred to as Rig, Hallinskiði, Gullintanni, and Vindlér or Vindhlér.

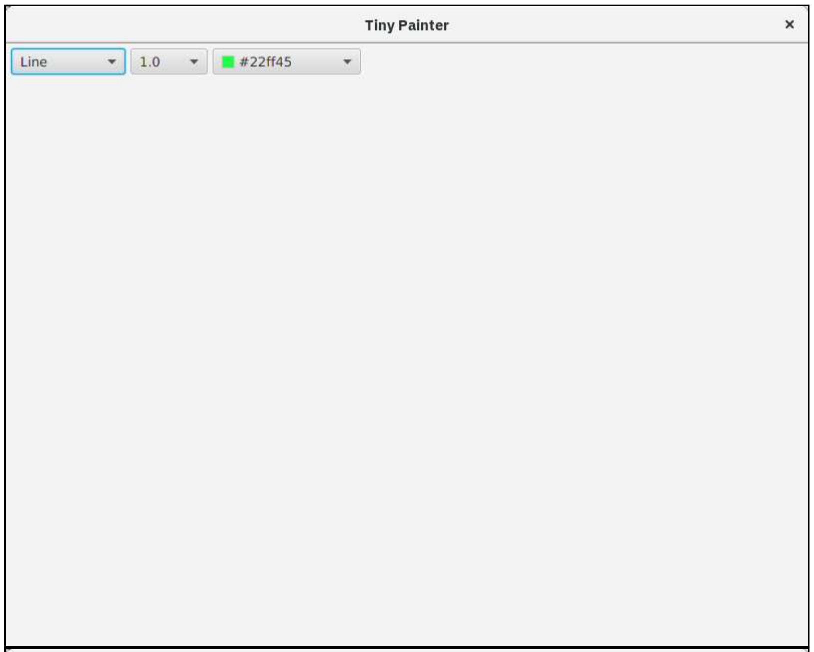
Heimdallr is attested in the Poetic Edda, compiled in the 13th century from earlier traditional material; in the Prose Edda

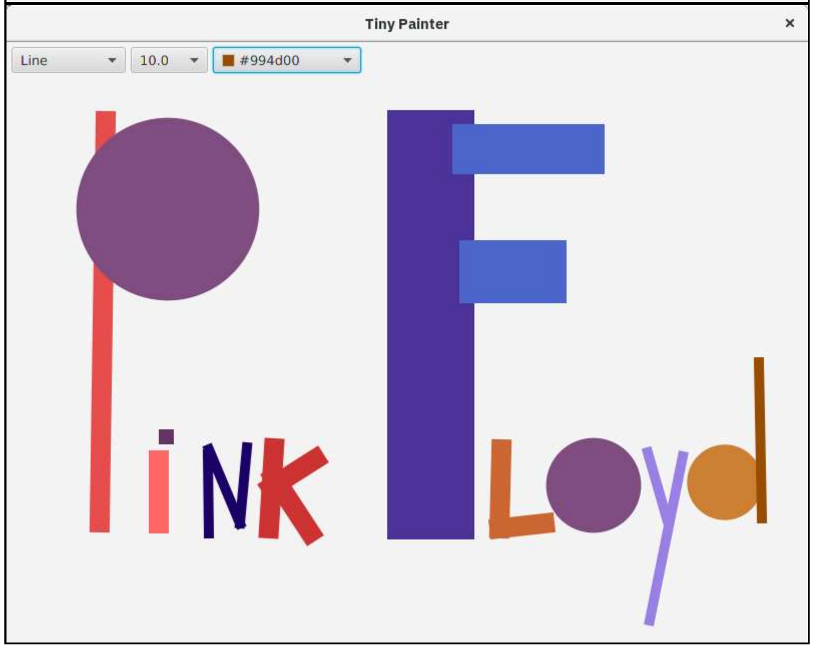
- **Exercise 9**

Create a JavaFX program called `TinyPainter` that will be a simple paint application. At the top of the application there should be a row of drop down menus for (in order) shape, size and colour. The shapes are limitied to Line, Dot, Rectangle and Circle and for Line and Dot a size is important, which is the second menu. The sizes should be ranging from 1 to 40 in suitable sizes (not all need to be included). The shapes Rectangle and Circle need no size but the menu can be available anyway.The last menu is for colour. Use the component `ColorPicker` to select a colour.

When a shape, size and colour is selected, the user should be able to draw the shape in the window. Drawing is done by pressing a mouse button and then drag the mouse to the suitable position in the window. As long as the mouse key is pressed the shape should follow the mouse, but as soon as it is released, the shape should be finalised in the window. For a Line, the start position is where the mouse key is first pressed and end position where it is released. A rectangle is drawn from upper left corner to lower right corner while a Dot simply is a square in the size selected. A circle is drawn with the center at the position where the mouse key is pressed and with a radius to the mouse pointer.

To simplify somewhat, it is acceptable if the shapes can overwrite the buttons at the top. See images below for examples of what the finished program can look like.

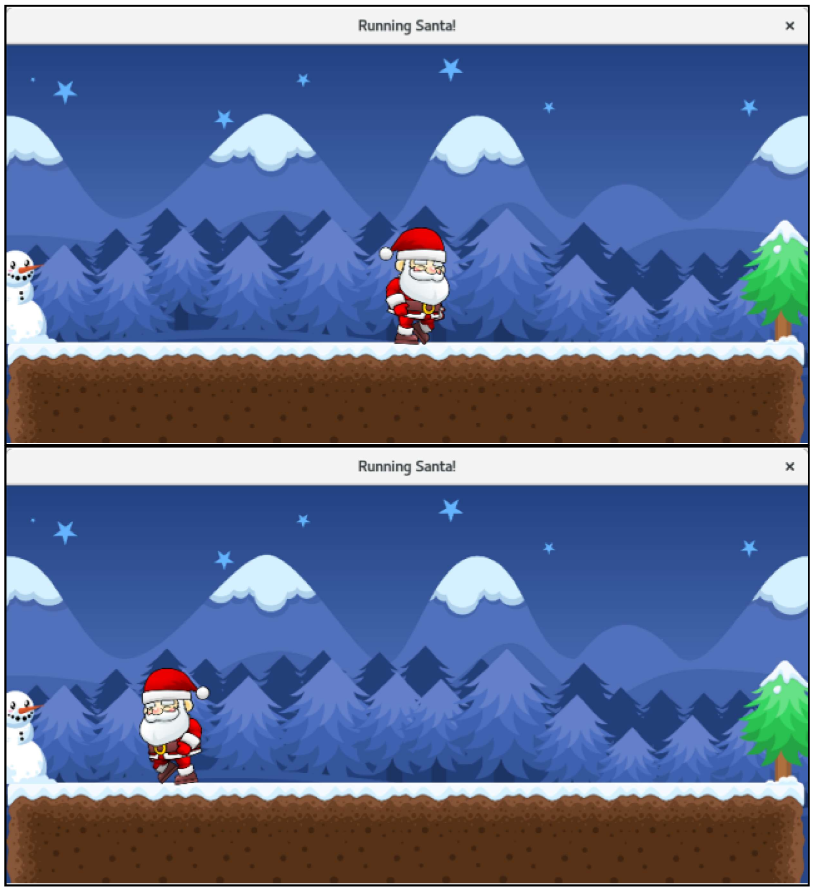




- **Exercise 10 (VG)**

For this task you will need some images that you can get at Game Art 2D. The task is to animate a character as shown in the images below over a flat surface. On the page referred to you will find so called *sprites*, characters in different positions. For the example, a santa claus has been used which has eleven characters for running (in the zip-file you download they have the name `Run (x).png`, where x is 1 to 11). There are also several backgrounds and tiles to place on the backgrounds that you can combine using a paint tool. In the example a snowman and a tree has be placed on the background at the start and end of the image.

The task is to have a sprite running between the start (snowman) and end (tree) markers with JavaFX animation. You can use any set of sprites and backgrounds from the Game Art 2D site as long as the character is running (unless you use the zombie set, then a slither movement is quite okay). We recommend that you use a keyframe animation on a timeline but as long as the character is nicely moving over the surface, it is okay.



Submission

Please note that all exercises apart from the VG exercises (the merge sort part of Exercise 2, and Exercise 10) are mandatory to pass. Also, we are only interested in your .java files. Hence, zip the directory named `YourLnuUserName_assign3` (inside directory named `src`) and submit it using the Moodle submission system.