10 Sorting, Stacks, Queues, and more...

Etude: Lab 10 — Words

For the etude finish the lab parts 10.2 and 10.3.

Part 1: Eliza

In the lab you started working on the *Eliza* program that allows the computer to interact with the user by providing replies to a series of questions. As part 1 of this assignment, finish the program. Include in your portfolio a sample transcript of the user-computer interaction.

Part 2: Selection Sort

The main goal is to practice working with loops by designing a well-known sorting algorithm. You will need data and *Comparator*s for your tests.

1. To test the algoritms you design here use the *City* classes we defined in **Assignment 8**. Define three different *Comparator*s: by name, by zip code, and by latitude.

In the Algorithms class design a static method *SelectionSort* that consumes an *ArrayList*<*T*> and an instance of a class that implements *Comparator*<*T*> and mutates the *ArrayList*<*T*> so that it is sorted in the order given by the *Comparator*<*T*>.

It is possible to combine all parts into one method, but you must use the following helper methods:

3. *swap* that swaps in the given *ArrayList*<*T*> the elements at the two given locations.

4. *findMinLoc* finds in the given *ArrayList*<*T*> the location of the minimum element among all elements at the location greater than or equal to the given location. Of course, it also consumes the *Comparator*<*T*>.

5. *selectionSort* method that is described at the beginning.

Variants

5. You can choose to use any of the loops we have seen (including the *Traversal<T>*, and its implementation for *ArrayList<T>*. However, as the second part of the problem you must convert your solutions for *minLoc* and *selectionSort* to use either *while* loop without the *Traversal<T>* or *for* loop without the *Traversal<T>*.

   If you already used one of these, convert the code to using the other loop. Rename your methods as *minLocV1* and *selectionSortV1*.

Part 3: Stacks, Queues, and Priority Queues

In our final project we will need to keep track of accumulated values — places we should visit next when searching for a path from one place to another on a map. However, the way how we add/remove items from this accumulator will depend on our choice of search strategy. Therefore, we start with a common interface, and design three different implementations of this interface.

The *Accumulator* interface is defined as follows:

```
/**
 * <P>An interface that represents a container for accumulated collection of
 * data elements. The implementation specifies the desired add and remove
 * behavior.</P>
 * <P>The expected implementations are Stack, Queue, and Priority Queue.</P>
 */
public interface Accumulator<T>{


  /**
   * Does this <CODE>{@link Accumulator}</CODE> contain any data elements?
   * @return true is there are no elements in this
   * <CODE>{@link Accumulator}</CODE>.
   */
  public boolean isEmpty();


  /**
   * Change the state of this <CODE>{@link Accumulator}</CODE> by adding
   * the given element to this <CODE>{@link Accumulator}</CODE>.
   * @param t the given element
   */
  public void add(T t);
```

```
    /**
     * Change the state of this <CODE>{@link Accumulator}</CODE> by removing
     * the given element to this <CODE>{@link Accumulator}</CODE>.
     * Produce the removed element.
     * @return the removed element
     */
    public T remove();
}
```

1. Design the class *MyStack<T>* that implements the *Accumulator<T>* interface by always removing the most recently added element.

2. Design the class *MyQueue<T>* that that implements the *Accumulator<T>* interface by always removing the least recently added element.

3. Design the class *MyPriorityQueue<T>* that contains an instance of a *Comparator<T>* and implements the *Accumulator<T>* interface by always removing the element that has the highest priority as determined by its *Comparator<T>*.

4. Design the classes *IllegalStackOperation IllegalQueueOperation* and *IllegalPriorityQueueOperation* that extend the class *Exception* in the *java.lang* package. Modify the methods that implement the *Stack*, *Queue*, and the *PriorityQueue* so that they *throw* the appropriate exceptions.

   Explore the Java documentation and in online tutorials to see how to *throw* and *catch* an *Exception* that is not a subclass of the *RuntimeException*.

Note: You can decide on your own what will be the class of data that will provide the elements to use in testing these classes.

Part 4: Extra Credit: Animations

Design an animated representation of a queue and a stack as follows:

The world consists of a queue and a stack of points. The queue is shown as a set of lines from the first point to the next, till the end. The stack is not seen. On tick, one point is removed from the stack and added to the queue. The effect is an animated display of a route from the beginning to the end.

The Documentation: a concise summary

You may have noticed that the style in which we write documentation for this assignment has changed. When written in the well formatted *javadoc* style, the comments can used to generate web pages of documentation with cross-references and browsing capabilities. There are a few basic rules, the rest you should learn on your own, gradually, as you become more and more skilled Java programmers.

Here are comments to specify the name of the file, and the class definition:

```
/*
 * @(#)Word.java   17 November 2006
 *
 */

/**
 *
 *  <P><CODE>Word</CODE> represents one word and its
 * number of occurrences counted in the
 * <CODE>{@link WordCounter WordCounter}</CODE> class.</P>
 *
 * @see Comparable
 *
 * @author Viera K. Proulx
 */
public class Word implements Comparable {
```

The *@author* and *@see* identify the author and provide a cross-reference to other classes as specified.

Each field in the class has its own comment:

```
/**
 * the frequency counter
 */
public int counter;
```

Each method has a comment that includes a separate line for each parameter as well as for the return value:

```
    /**
     * Compare two <CODE>Object</CODE>s for equality
     *
     * @param obj the object to compare to
     * @return true if the two objects have the same contents
     */
    public boolean equals(Object obj){
```

The *@param* has to be followed by the identifier used for that parameter. The <CODE> and < /CODE> tags specify the formatting for the document to be the teletype font for representing the code.

Eclipse helps you to write the documentation. If you start the comment line with /∗∗ and hit the return, the beginnings of remaining comment lines are generated automatically, and you only need to add the relevant information.

When you have finished all the documentation, select the item **Generate Javadoc...** in the **Project** menu. To see your web pages, just open the tab *doc* in the **Package Explorer** window under your project and double click on the *index.html*.