LAB 9: SORTING, JAVA GENERICS & LAMBDA EXPRESSIONS

F27SG – SOFTWARE DEVELOPMENT 3 (10 MARKS)

This is the final lab. Your final chance to get it marked is the lab session in week 12!

The topic of this lab is **sorting** and **Java Generics** and **lambda expressions**. Download Lab9.zip from Vision and import the project into Eclipse:

*File -> Import -> Existing Projects into **Workspace***

Then select the project you downloaded. The project is organized as follows:

- The src directory contains all the source files
  - **ArraySort.java** is where you should implement methods to sort arrays (tasks 2 and 3)
  - **DLinkedList.java** is where you should implement methods to sort doubly linked lists (task 4)
  - **GenericSort.java** is where you should implement the sorting algorithms using Generics (task 5) and a BiPredicate (task 6)
  - **AdvancedJava.java** is where you should call the generic sorting algorithms using anonymous classes (task 5) and lambda expressions (task 6)
- The test directory contains the unit tests for the project.

Note that tasks 1-4 can be completed after the lectures in week 9, while the topics for tasks 5 and 6 will be covered in week 10.

1. WRITE UNIT TESTS FOR SORTING ALGORITHMS (2 POINTS)

In part 2, 3 and 4 you should implement algorithms to sort integers. These are:

- Bubble-sort of an **array** of integers (ArraySort.java)
- Quick-sort of an **ArrayList** of integers (ArraySort.java)
- Insertion-sort of a **Doubly Linked List** of integers (DlinkedList.java)

In **ArraySortTest.java** and **DlinkedListTest.java** empty test methods have been provided for you to complete. For each sorting method you should test sorting of

- an empty collection
- an already sorted (ordered) collection
- an un-ordered collection (i.e. elements are in a random order)

The test should check that the list is ordered and has the same size as before - suitable methods have been provided in **ArraySort.java** and **DlinkedList.java** which you can use for these checks.

2. IMPLEMENT BUBBLE-SORT (2 POINTS)

Bubble-Sort is a comparable sorting algorithm to *Insertion-Sort*. **ArraySort.java** has an empty method skeleton

```
public static void bubbleSort(int[] arr){

      // your code

}
```

Your task is to implement **`bubbleSort`** to sort the given array of integers. The `main` method contains a test case so you can see the program running. The JUnit tests from part 1 should also succeed. You need to do the following in the method:

- Introduce a boolean variable `swaps` initially set to `true`
- While `swaps` is `true`
  - set `swaps` is `false`
  - step through the array from beginning to end (minus the last element)
    - for each step `i`
      - if `arr[i+1]` is smaller than `arr[i]`
        - swap the values of `arr[i+1]` and `arr[i]`
        - set `swaps` to `true`

3. IMPLEMENT QUICK-SORT (2 POINTS)

In **ArraySort.java** there is an empty method

```
public static void quickSort(ArrayList<Integer> S){

      // your code

}
```

Your task is to implement the **`quickSort`** algorithm in this method. Note that we are here using an `ArrayList` and not an array since we do not know the sizes of L, E and G (see below) before running the code. The **`quickSort`** algorithm works as follows:

- If the size of (input) `S` is less than or equal to one then `S` is sorted so you can return (i.e. base case)
- Select an element of `S` to be the `pivot`. You can choose which element this is yourself (e.g. first element of `S`, middle element of `S`, last element of `S`,...)
- Create 3 new `ArrayLists` (of type Integer):
  - L which should store elements of S that are less than the `pivot`
  - E which should store elements of S that equal to the `pivot`
  - G which should store elements of S that are greater than the `pivot`
- While S is not empty

        - **get and delete** the first element and add it to one of L, E and G, according to how it compares with the `pivot`
    - Recursively call `quicksort(L)` and `quicksort(G)`
    - Add all elements back to S in the order:
        - elements of L (in the same order they are in L)
        - elements of E (in the same order they are in E)
        - elements of G (in the same order as they are in G)

Note that an `ArrayLists` has an `addAll` method that will be useful for combining L, E and G into S. The `main` method contains a test case you can use to run your code, and your JUnit tests from part 1 should also succeed.

4. INSERTION-SORT OF DOUBLY LINKED LIST (2 POINTS)

The class **DLinkedList.java** contains an implementation of a Doubly Linked List of integers. This class contains an empty method

```
public  void insertionSort(){


}
```

This method should sort the list using the insertion-sort algorithm discussed in the lecture. A **main** method has been provided for this class so that you can run your code; the JUnit tests from part 1 should also succeed. Two additional methods have been provided in this class to support you

- `int delete(Node n)` deletes node `n` from the doubly linked list and returns its value.
- `insertAfter(Node n,int val)` inserts a new node containing value `v` after node `n`..

5. INSERTION-SORT USING JAVA GENERICS (1 POINT)

You should adapt the insertion-sort algorithm covered in the lecture to use **Generics** instead. This should be implemented by completing the method

```
public void insertionSort(E[] list,Comparator<E> comp){


}
```

in **GenericSort.java**. Next, you should use this method to sort **Integers and Strings** by completing the code in:

```
public void task5()
```

In both cases, you should implement the **Comparator** interface using an **anonymous** class as discussed in the lecture.

6. INSERTION-SORT USING LAMBDA EXPRESSIONS (1 POINT)

For this task you should use lambda expressions for the same problem as in task 5. First the insertion-sort algorithm should use a **BiPredicate** instead of a **Comparator** in **GenericSort.java**:

```
public void insertionSort(E[] list,BiPredicate<E,E> pred){


}
```

Next, you should use this method to sort **Integers and Strings** by completing the code in:

```
public void task6()
```

Here, you should use **lambda expressions** instead of the anonymous classes you used for part 5.

A **main** method is provided which should print both arrays in ordered form for both part 5 and part 6.

ADDITIONAL CHALLENGES

Implement the sorting algorithms from parts 2, 3 and 4 using Generics and lambda expressions.