10 Assignment

Portfolio Problems: Loops

Finish the third part of Lab 10 that deals with loops.

10.1 Eliza

Our goal is to train our computer to be a mock psychiatrist, carrying on a conversation with a patient. The patient (the user) asks a series of questions. The computer-psychiatrist replies to each question as follows. If the question starts with one of the following (key)words: Why, Who, How, Where, When, and What, the computer selects one of the three (or more) possible answers appropriate for that question. If the first word is none of these words the computer replies 'I do not know' or something like that.

1. Start by designing the class *Reply* that holds a keyword for a question, and an *ArrayList* of answers to a the question that starts with this keyword.

2. Design the method *randomAnswer* for the class *Reply* that produces one of the possible answers each time it is invoked. Make sure it works fine even if you add new answers to your database later. Make at least three answers to each question.

3. Design the class *Eliza* that contains an *ArrayList* of *Reply*s.

4. In the class *Eliza* design the helper method *firstWord* that consumes a *String* and produces the first word in the *String*.

   The following code reads the next input line from the user. You will need to find out what was the first word in the patient's question. Look up the documentation for the *String* class (and we gently hint that the methods *trim*, *toLowerCase*, and *startsWith* may be relevant).

   ```
   System.out.println("Type in a question: ");
   s = input.nextLine();
   ```

   Make sure your program works if the user uses all uppercase letters, all lower case letter, mixes them up, etc.

5. In the class *Eliza* design the method *answerQuestion* that consumes the question *String* and produces the (random) answer. If the first word of the question does not match any of the replies, produce an answer *Don't ask me that.* — or something similar. If no first word exists, i.e., the user either did not type any letters, or just hit the return, throw an *EndOfSessionException*.

   Of course, you need to define the *EndOfSessionException* class.

6. In the *Interactions* class design the method that repeats asking questions and providing answers until it catches the *EndOfSessionException* — at which time it ends the game.

10.2 Selection Sort

Selection Sort algorithms works as follows. When the program traverses the list of data for the first time, it finds the location of the smallest item in the list. It then *swaps* the first item in the list with the smallest one (even if the smallest one is already in the first spot).

Next time around, it does the same, but only with the rest of the list, i.e. all items beyond the first one. The third time around, it starts with the third item, because the first two are already in the correct places.

This is hard to do with the recursively constructed lists, but is much easier when we can directly swap two items at specific locations, as is the case when the data is stored in an *ArrayList*.

In the *Algorithms* class design a 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 of the algorithm into one method, but we do not want you to design programs that way. Your program should use the following helper methods:

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

- *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*>.

- the main method, *selectionSort* the implements the algorithm that is described at the beginning.

Variants

You should hand in two different implementation of this algorithm as follows:

You can choose to use any of the loops we have seen (including the *Traversal<T>*, and its implementation for *ArrayList<T>*. However, you should then convert your solutions for *minLoc* and *selectionSort* to use either *while* loop without the *Traversal<T>* or *for* loop without the *Traversal<T>* and hand in both solutions.

If your first solution already used one of these loops (*while* or *for*), your second solution should then use the other loop.

Rename your methods as *minLocV1* and *selectionSortV1*.

Tests

Of course, you need to test your methods. Make a simple class of data, such as a *Book* or *Balloon* we have used in the past — but come up with something different — and define two different *Comparator*s for this class. Then make examples of lists of these data items and make sure your tests use both of the *Comparator*s.

Organize youre tests so that the reader can readily see what is the purpose of each test and what data is used in computing the result and in providing the expected value.