Snaek

Shashank Singh

1 Introduction

Please read the entire document and **do not forget to put the collaboration statement at the top of your submission!**

2 Problem Description

In this week's homework, you will be creating a twist on the classic game **Snake**! You will only have to implement the movement of the Snaek. We won't be worrying about having the Snaek grow by eating food. The main focus of this homework will be to write a good implementation for the singly linked list class that will be backing the Snaek. It will also test your understanding of the efficiency of various operations on a singly linked list.

3 Solution Description

Snaek is a JavaFX Application that consists of the following classes:

- **SnaekFX:** extends Application and is the main entry point of the program
- **SnaekController:** contains all the logic for moving and displaying the Snaek
- **SinglyLinkedList:** interface containing methods that SnaekLinkedList must implement
- **SnaekLinkedList:** your implementation of the SinglyLinkedList interface's methods
- **SnaekNode:** a single node in a linked list that stores the position of that cell of the Snaek
- **Position:** represents the position of one cell of the Snaek's body on the game board

3.1 SnaekFX

We've provided you with all of the functionality for this class. You do not need to add anything here. If you want to, you can play around with the game parameters `CELL_SIZE`, `GAME_SIZE`, and `GAME_DELAY` to change the size of each Snaek cell, the number of rows and columns in the game board or the delay in milliseconds between each movement.

3.2 Position

This class has been provided for you. You will still need to know how to use its methods to get and set the position of the cell. This class represents the `x` and `y` position of one Snaek cell where `(0, 0)` is the top left cell of the window.

3.3 SnaekNode

This class has been provided for you. You will still need to know how to use its methods to get and set the reference to the next SnaekNode appropriately. Each SnaekNode holds one Position corresponding to one cell of the Snaek.

3.4 SinglyLinkedList

This interface contains the Linked List methods that you must implement in SnaekLinkedList.

3.5 SnaekLinkedList

This is where you will be doing most of the coding. This class implements SinglyLinkedList so you will have to implement all of the methods present in SinglyLinkedList. The Singly Linked List should have a head reference to the first element and a tail reference to the last element each of which you should update appropriately in the methods below. There is also a size instance variable that you must modify appropriately in each method. We have provided you with the implementations for some of the methods and you have to implement the remaining ones:

- **`SnaekLinkedList()`:** constructor that should initialize the list with size 0
- **`void addFront(Position newPos)`:** add newPos to the front of the list
  - Don't do anything if `newPos` is `null`
- **`void addEnd(Position newPos)`:** add newPos to the end of the list
  - Don't do anything if `newPos` is `null`
- **`Position removeFront()`:** remove and return the Position from the front of the list
  - Return `null` if the list is empty
- **`Position removeEnd()`:** remove and return the Position from the end of the list
  - Return `null` if the list is empty
- **`Position getFront()`:** return the Position at the front of the list
  - Return `null` if the list is empty
- **`Position getEnd()`:** return the Position at the end of the list
  - Return `null` if the list is empty
- **`boolean contains(Position other)`:** check if the list contains the given Position
- **`void clear()`:** empty the list and reset it to size 0
- **`int size()`:** return the size of the list
- **`boolean isEmpty()`:** return true if the list is empty

3.6 SnaekController

This is where the game controls are implemented. We have already provided the setup code as well as the `reset()` method for resetting the game. The game is initialized with a `snaek` of some size by representing it as a SnaekLinkedList of Positions. Your job will be to code the following methods:

- **`void move()`:** move the Snaek one cell based on the current movement direction
  - We are going to move the Snaek by removing a Position from one end of the Snaek, updating the Position's X and Y values and then adding it back to the other end of the Snaek. You will do this by using the SnaekLinkedList methods that you have implemented. There are two ways of doing this: removing from the front and adding to the end or removing from the end and adding to the front. You will have to decide which approach is better based on what you have learnt about running times of various operations.
  - To determine where to add the removed end, you will have to use the `currentDir` enum variable which can be either `U` (up), `D` (down), `L` (left), or `R` (right).
  - If the `currentDir` is `U` then, the removed Position's X and Y should be updated so that they are above the Position at the other end of the Snaek and similarly for the other 3 directions.
  - Before adding the modified Position back to the Snaek, you will have to check if that would cause the Snaek to overlap itself in which case you should reset the game by calling `resetGame()`. Otherwise, you can go ahead and add the Position back to the Snaek at the other end.
  - Finally, as in the classic Snake, if the Snaek goes out of bounds on one side of the screen, it should wrap around from the other side. To do this, you should call the `wrapLocation(int val)` method on your removed Position's new X and Y co-ordinates to adjust them back into the viewable area.
- The `resetGame()` and `wrapLocation(int val)` methods have been provided for you.

4 Important Notes, Tips, and Tricks

- Do **NOT** modify sections of the code that you are not supposed to. This includes changing instance variable names or deleting any of the provided methods. You may, of course, add any methods for your own convenience.
- Avoid repeating yourself and write good, clean code that does not use hardcoded values.

5 Checkstyle

You must run checkstyle on your submission. The checkstyle cap for this assignment is **100** points. Review the Style Guide and download the Checkstyle jar. Run Checkstyle on your code like so:

```
$ java -jar checkstyle-6.2.2.jar *.java
Audit done. Errors (potential points off):
0
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off. The Java source files we provide contain no Checkstyle errors. For this assignment, there will be a maximum of **100** points lost due to Checkstyle errors (1 point per error). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

6 Javadocs

For this assignment you will be commenting your code with Javadocs. Javadocs are a clean and useful way to document your code's functionality. For more information on what Javadocs are and why they are awesome, the online documentation for them is very detailed and helpful. You can generate the javadocs for your code using the command below, which will put all the files into a folder called javadoc:

```
$ javadoc *.java -d javadoc
```

The relevant tags that you need to have are `@author`, `@version`, `@param`, and `@return`. Here is an example of a properly Javadoc'd class:

```
import java.util.Scanner;

/**
 * This class represents a Dog object.
 * @author George P. Burdell
 * @version 13.31
 */
public class Dog {

   /**
    * Creates an awesome dog (NOT a dawg!)
    */
   public Dog(){
      ...
   }

   /**
    * This method takes in two ints and returns their sum
    * @param a first number
    * @param b second number
    * @return sum of a and b
    */
   public int add(int a, int b){
      ...
   }
}
```

Take note of a few things:

1. Javadocs are begun with `/**` and ended with `*/`.
2. Every class you write must be Javadoc'd and the `@author` and `@version` tag included. The comments for a class start with a brief description of the role of the class in your program.

3. Every non-private method you write must be Javadoc'd and the `@param` tag included for every method parameter. The format for an `@param` tag is `@param <name of parameter as written in method header> <description of parameter>`. If the method has a non-void return type, include the `@return` tag which should have a simple description of what the method returns, semantically.

6.1 Javadoc and Checkstyle

You can use the Checkstyle jar mentioned in the following section to test your javadocs for completeness. Simply add -j to the checkstyle command, like this:

```
$ java -jar checkstyle-6.2.2.jar -j *.java
Audit done. Errors (potential points off):
0
```

7 Turn-in Procedure

Submit all of your `.java` files on T-Square as an attachment, including the ones that were provided for you. Do not submit any compiled bytecode (`.class` files), the Checkstyle jar file, or the `cs1331-checkstyle.xml` file. When you're ready, double- check that you have submitted and not just saved a draft.

Keep in mind, unless you are told otherwise, non-compiling submissions on ALL homeworks will be an automatic 0. You may not be warned of this in the future, and so you should consider this as your one and only formal warning. This policy applies to (but is not limited to) the following:

1. Forgetting to submit a file
2. One file out of many not compiling, thus causing the entire project not to compile
3. One single missing semi-colon you accidentally removed when fixing Checkstyle stuff
4. Files that compile in an IDE but not in the command line
5. Typos
6. etc...

We know it's a little heavy-handed, but we do this to keep everyone on an even playing field and keep our grading process going smoothly. So please please make sure your code compiles before submitting. We'd hate to see all your hard work go to waste!

8 Verify the Success of Your Submission to T-Square

Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.

2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.

3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.

4. Recompile and test those exact files.

5. This helps guard against a few things.

   (a) It helps insure that you turn in the correct files.

   (b) It helps you realize if you omit a file or files.[1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)

   (c) Helps find last minute causes of files not compiling and/or running.

[1] Missing files will not be given any credit, and non-compiling homework solutions will receive zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is 8PM Thursday. Do not wait until the last minute!