

Homework Turnin

Name: Ameya Singh
Account: ameyas (ameyas@uw.edu)
Student ID: 1868457
Section: AQ
Course: CSE 143 18au
Assignment: a3
Receipt ID: 8038ab92aeec4094aa76eb59abebaf19

Turnin Successful!

The following file(s) were received:

AssassinManager.java (6562 bytes)

```
1. /*
2.  * Author: Ameya Singh
3.  * CSE 143 AQ
4.  * TA: Soham P.
5.  * Homework 3: AssassinManager
6.  */
7.
8. import java.util.*;
9.
10. /**
11.  * AssassinManager keeps track of an Assassin game.
12.  * Manages a "kill ring" of all players still currently in the game. Also
13.  * manages a "graveyard" of players who were killed.
14.  */
15. public class AssassinManager {
16.     /**
17.      * Front node of the kill ring.
18.      */
19.     private AssassinNode killRing;
20.     /**
21.      * Front node of the graveyard.
22.      */
23.     private AssassinNode graveyard;
24.
25.     /**
26.      * Constructs a new AssassinManager and initializes the kill ring in the
27.      * same order of the passed List. Players will be hunting the name after
28.      * them in the passed list.
29.      *
30.      * @param names List of names in order of how kill ring should be
31.      *               constructed. Assumes names are non-empty strings and that
32.      *               there are no repeated names.
33.      * @throws IllegalArgumentException Thrown if the passed list is empty.
34.      */
35.     public AssassinManager(List<String> names) {
36.         if (names.isEmpty()) {
37.             throw new IllegalArgumentException();
38.         }
39.
40.         killRing = new AssassinNode(names.get(0));
41.
42.         AssassinNode current = killRing;
43.         for (int i = 1; i < names.size(); i++) {
44.             while (current.next != null) {
45.                 current = current.next;
46.             }
47.             current.next = new AssassinNode(names.get(i));
48.         }
49.     }
```

```

50.     }
51.
52.     /**
53.      * Prints a formatted version of the current kill ring to the console.
54.      */
55.     public void printKillRing() {
56.         AssassinNode current = killRing;
57.
58.         while (current != null) {
59.             String next;
60.             if (current.next == null) {
61.                 next = killRing.name;
62.             } else {
63.                 next = current.next.name;
64.             }
65.
66.             String s = "      "
67.                 + current.name
68.                 + " is stalking "
69.                 + next;
70.             System.out.println(s);
71.
72.             current = current.next;
73.         }
74.     }
75.
76.     /**
77.      * Prints a formatted version of the current graveyard in order of most
78.      * recently killed to first killed to the console.
79.      */
80.     public void printGraveyard() {
81.         AssassinNode current = graveyard;
82.
83.         while (current != null) {
84.             String s = "      "
85.                 + current.name
86.                 + " was killed by "
87.                 + current.killer;
88.             System.out.println(s);
89.
90.             current = current.next;
91.         }
92.     }
93.
94.     /**
95.      * Returns whether or not the passed name is present in the current
96.      * game.
97.      *
98.      * @param name Name to be check for within kill ring. Ignores case when
99.      * comparing names.
100.     * @return Returns true if the name is contained within the kill ring.
101.     * Returns false if the name is not found in the kill ring.
102.     */
103.     public boolean killRingContains(String name) {
104.         return listContains(name, killRing);
105.     }
106.
107.     /**
108.      * Returns whether or not the passed name has died in the current game.
109.      *
110.      * @param name Name to be check for within graveyard. Ignores case when
111.      * comparing names.
112.      * @return Returns true if the name is contained within the graveyard.
113.      * Returns false if the name is not found in the graveyard.
114.      */
115.     public boolean graveyardContains(String name) {
116.         return listContains(name, graveyard);
117.     }
118.
119.     /**
120.      * Private helper method to search for a specific name within a List of
121.      * AssassinNodes
122.      *
123.      * @param name Name to search for within the list. Case-insensitive.
124.      * @param list Front node of list to be searched.
125.      * @return Returns true if list contains name.
126.      */
127.     private boolean listContains(String name, AssassinNode list) {
128.         AssassinNode current = list;
129.         while (current != null) {
130.             if (current.name.toLowerCase().equals(name.toLowerCase())) {

```

```

131.         return true;
132.     }
133.     current = current.next;
134. }
135. return false;
136. }
137.
138. /**
139.  * Returns true if the game is over (only one person remains in the game).
140.  *
141.  * @return True if game over. False if not.
142.  */
143. public boolean gameOver() {
144.     return killRing.next == null;
145. }
146.
147. /**
148.  * Returns the name of the winner of the game.
149.  *
150.  * @return Returns the name of the winner. Returns null if game is not over.
151.  */
152. public String winner() {
153.     if (gameOver()) {
154.         return killRing.name;
155.     }
156.     return null;
157. }
158.
159. /**
160.  * Records the killing of the player whose name is passed.
161.  *
162.  * @param name Name of player killed. Not case sensitive.
163.  * @throws IllegalArgumentException Thrown if passed name is not part of the
164.  *                                     game or not alive.
165.  * @throws IllegalStateException Thrown if the game is over.
166.  */
167. public void kill(String name) {
168.     // Exception handling
169.     if (!killRingContains(name)) {
170.         throw new IllegalArgumentException();
171.     }
172.     if (gameOver()) {
173.         throw new IllegalStateException();
174.     }
175.
176.     AssassinNode currentKill = killRing;
177.
178.     // Front case: if killed is the front node.
179.     if (currentKill.name.toLowerCase().equals(name.toLowerCase())) {
180.         AssassinNode temp = currentKill.next;
181.         currentKill.next = null;
182.
183.         AssassinNode other = temp;
184.         while (other.next != null) {
185.             other = other.next;
186.         }
187.         currentKill.killer = other.name;
188.
189.         addGrave(currentKill);
190.
191.         killRing = temp;
192.     } else { // All other cases
193.         while (!currentKill.next.name.toLowerCase().equals(name.toLowerCase())) {
194.             currentKill = currentKill.next;
195.         }
196.
197.         AssassinNode temp = currentKill.next;
198.         currentKill.next = temp.next;
199.         temp.next = null;
200.         temp.killer = currentKill.name;
201.
202.         addGrave(temp);
203.     }
204. }
205.
206. /**
207.  * Private helper method to add an AssassinNode to the front of the
208.  * graveyard without creating a new node.
209.  *
210.  * @param kill Node to be moved to front of graveyard.
211.  */

```

```
212.     private void addGrave(AssassinNode kill) {
213.         AssassinNode currentGrave = graveyard;
214.         if (currentGrave == null) {
215.             graveyard = kill;
216.         } else {
217.             graveyard = kill;
218.             graveyard.next = currentGrave;
219.         }
220.     }
221. }
222.
```