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)

```java
/*
 * Author: Ameya Singh
 * CSE 143 AQ
 * TA: Soham P.
 * Homework 3: AssassinManager
 */

import java.util.*;

/**
 * AssassinManager keeps track of an Assassin game.
 * Manages a "kill ring" of all players still currently in the game. Also
 * manages a "graveyard" of players who were killed.
 */
public class AssassinManager {
    /**
     * Front node of the kill ring.
     */
    private AssassinNode killRing;
    /**
     * Front node of the graveyard.
     */
    private AssassinNode graveyard;


    /**
     * Constructs a new AssassinManager and initializes the kill ring in the
     * same order of the passed List. Players will be hunting the name after
     * them in the passed list.
     *
     * @param names List of names in order of how kill ring should be
     *              constructed. Assumes names are non-empty strings and that
     *              there are no repeated names.
     * @throws IllegalArgumentException Thrown if the passed list is empty.
     */
    public AssassinManager(List<String> names) {
        if (names.isEmpty()) {
            throw new IllegalArgumentException();
        }

        killRing = new AssassinNode(names.get(0));

        AssassinNode current = killRing;
        for (int i = 1; i < names.size(); i++) {
            while (current.next != null) {
                current = current.next;
            }
            current.next = new AssassinNode(names.get(i));
        }
    }

    /**
     * Prints a formatted version of the current kill ring to the console.
     */
    public void printKillRing() {
        AssassinNode current = killRing;

        while (current != null) {
            String next;
            if (current.next == null) {
                next = killRing.name;
            } else {
                next = current.next.name;
            }

            String s = "    "
                    + current.name
                    + " is stalking "
                    + next;
            System.out.println(s);

            current = current.next;
        }
    }

    /**
     * Prints a formatted version of the current graveyard in order of most
     * recently killed to first killed to the console.
     */
    public void printGraveyard() {
        AssassinNode current = graveyard;

        while (current != null) {
            String s = "    "
                    + current.name
                    + " was killed by "
                    + current.killer;
            System.out.println(s);

            current = current.next;
        }
    }

    /**
     * Returns whether or not the passed name is present in the current
     * game.
     *
     * @param name Name to be check for within kill ring. Ignores case when
     *             comparing names.
     * @return Returns true if the name is contained within the kill ring.
     * Returns false if the name is not found in the kill ring.
     */
    public boolean killRingContains(String name) {
        return listContains(name, killRing);
    }

    /**
     * Returns whether or not the passed name has died in the current game.
     *
     * @param name Name to be check for within graveyard. Ignores case when
     *             comparing names.
     * @return Returns true if the name is contained within the graveyard.
     * Returns false if the name is not found in the graveyard.
     */
    public boolean graveyardContains(String name) {
        return listContains(name, graveyard);
    }

    /**
     * Private helper method to search for a specific name within a List of
     * AssassinNodes
     *
     * @param name Name to search for within the list. Case-insensitive.
     * @param list Front node of list to be searched.
     * @return Returns true if list contains name.
     */
    private boolean listContains(String name, AssassinNode list) {
        AssassinNode current = list;
        while (current != null) {
            if (current.name.toLowerCase().equals(name.toLowerCase())) {
                return true;
            }
            current = current.next;
        }
        return false;
    }

    /**
     * Returns true if the game is over (only one person remains in the game).
     *
     * @return True if game over. False if not.
     */
    public boolean gameOver() {
        return killRing.next == null;
    }

    /**
     * Returns the name of the winner of the game.
     *
     * @return Returns the name of the winner. Returns null if game is not over.
     */
    public String winner() {
        if (gameOver()) {
            return killRing.name;
        }
        return null;
    }

    /**
     * Records the killing of the player whose name is passed.
     *
     * @param name Name of player killed. Not case sensitive.
     * @throws IllegalArgumentException Thrown if passed name is not part of the
     *                                  game or not alive.
     * @throws IllegalStateException    Thrown if the game is over.
     */
    public void kill(String name) {
        // Exception handling
        if (!killRingContains(name)) {
            throw new IllegalArgumentException();
        }
        if (gameOver()) {
            throw new IllegalStateException();
        }

        AssassinNode currentKill = killRing;

        // Front case: if killed is the front node.
        if (currentKill.name.toLowerCase().equals(name.toLowerCase())) {
            AssassinNode temp = currentKill.next;
            currentKill.next = null;

            AssassinNode other = temp;
            while (other.next != null) {
                other = other.next;
            }
            currentKill.killer = other.name;

            addGrave(currentKill);

            killRing = temp;
        } else { // All other cases
            while (!currentKill.next.name.toLowerCase().equals(name.toLowerCase())) {
                currentKill = currentKill.next;
            }

            AssassinNode temp = currentKill.next;
            currentKill.next = temp.next;
            temp.next = null;
            temp.killer = currentKill.name;

            addGrave(temp);
        }
    }

    /**
     * Private helper method to add an AssassinNode to the front of the
     * graveyard without creating a new node.
     *
     * @param kill Node to be moved to front of graveyard.
     */
    private void addGrave(AssassinNode kill) {
        AssassinNode currentGrave = graveyard;
        if (currentGrave == null) {
            graveyard = kill;
        } else {
            graveyard = kill;
            graveyard.next = currentGrave;
        }
    }
}

```