CIS 203 Assignment 8 – Assassin*

* The original assignment and its description were developed by Stuart Reges, University of Washington. The assignment has been modified from the original.

This assignment will give you practice with linked lists. You will be creating linked lists that are similar to the built-in `List<E>` in Java. **You are NOT permitted to use the existing Java `List<E>`** because the point of the assignment is for you to understand how linked lists work.

IMPORTANT NOTE: I WILL RUN YOUR CODE THROUGH ANTI-PLAGIARISM SOFTWARE AS THE FIRST STEP OF GRADING ON THIS ASSIGNMENT. IF YOU COPY A SOLUTION FROM ONLINE OR FROM ANOTHER STUDENT, I WILL CATCH YOU AND THERE WILL BE SERIOUS CONSEQUENCES.

You are to write a class `AssassinManager` that allows a client to manage a game of assassin. Each person playing assassin has a particular target that he/she is trying to assassinate. One of the things that makes the game more interesting to play is that initially each person knows only who they are assassinating (they don't know who is trying to assassinate them nor do they know who other people are trying to assassinate). You are working on a program for the "game administrator" who needs to keep track of who is stalking whom and the history of who killed whom.

The game of assassin is played as follows. You start out with a group of people who want to play the game. For example, let's say that we have five people playing whose names are Joe, Sally, Jim, Carol and Chris. A circular chain of assassination targets is established (what is called the "kill ring" in the sample log of execution). For example, we might decide Joe should stalk Sally, Sally should stalk Jim, Jim should stalk Carol, Carol should stalk Chris and Chris should stalk Joe.

```
Joe --> Sally --> Jim --> Carol --> Chris
 ^                                   |

 |                                   V

 +--------<--------<---------<--------+
```

When someone is assassinated, the chain needs to be relinked by "skipping" that person. For example, suppose that Jim is assassinated first (obviously this would have been by Sally). Sally needs a new target, so we give her Jim's target: Carol. Thus, the chain becomes:

```
         +-------->--------+
          ^                |
          |                V
Joe --> Sally     Jim     Carol --> Chris
 ^                                    |
 |                                    V
 +--------<--------<---------<--------+
```

The main program has been written for you and is called `AssassinMain`. It reads a file of names, shuffles the names, and constructs an object of type `AssassinManager`. You are writing the `AssassinManager` class. The main program then asks the user for the names of the victims in order until the game is over (until there is just one player left alive), calling methods of the `AssassinManager` class to carry out the tasks involved in administering the game.

Your class will keep track of two different lists: the list of those currently alive and the list of those who are dead. Each is to be stored in a linked list. We are requiring you to use our node class which is called `AssassinNode`. The `AssassinNode` class has three data fields: one for the name of the person, one for the name of the killer and a "next" field to keep track of the next `AssassinNode` in the list. The next field points to the person that this person is stalking. The `AssassinNode` class appears at the end of this write-up.

The constructor for the `AssassinManager` class will be passed an object of type `List<String>`. You can manipulate this object the same way you would manipulate an `ArrayList<String>`. So you can call methods like `size()`. However, you are not allowed to modify this list. You will need to include the following line at the beginning of your class to have access to List:

```
import java.util.*;
```

Your class should have the following `public` methods.

Method	Description
`AssassinManager (List<String> names)`	This is your method for constructing an assassin manager object. It should add the names from the list into the kill ring in the same order in which they appear in the list. For example, if the list contains {"John", "Sally", "Fred"}, then in the initial kill ring we should see that John is stalking Sally who is stalking Fred who is stalking John (reported in that order). You may assume that the names are nonempty strings and that there are no duplicate names (ignoring case). Your method should throw an `IllegalArgumentException` if the list is empty.

`void printKillRing()`	This method should print the names of the people in the kill ring, one per line, indented four spaces, with output of the form "<name> is stalking <name>". If there is only one person in the ring, it should report that the person is stalking themselves (e.g., "John is stalking John").
void `printGraveyard()`	This method should print the names of the people in the graveyard, one per line, indented four spaces, with output of the form "<name> was killed by <name>". It should print the names in reverse kill order (most recently killed first, then next more recently killed, and so on). It should produce no output if the graveyard is empty.
`boolean killRingContains(String name)`	This should return `true` if the given name is in the current kill ring and should return `false` otherwise. It should ignore case in comparing names.
`boolean graveyardContains(String name)`	This should return `true` if the given name is in the current graveyard and should return `false` otherwise. It should ignore case in comparing names.
`boolean gameOver()`	This should return `true` if the game is over (i.e., if the kill ring has just one person in it) and should return `false` otherwise.
`String winner()`	This should return the name of the winner of the game. It should return `null` if the game is not over.
`void kill(String name)`	This method records the killing of the person with the given name, transferring the person from the kill ring to the graveyard. This operation should not change the kill ring order of `printKillRing()` (i.e., whoever used to be printed first should still be printed first unless that's the person who was killed, in which case the person who used to be printed second should now be printed first). It should throw an `IllegalArgumentException` if the given name is not part of the current kill ring and it should throw an `IllegalStateException` if the game is over (it doesn't matter which it throws if both are true). It should ignore case in comparing names.

This is meant to be an exercise in implementing linked lists that are like the built-in `LinkedList<E>` (you are not allowed to use the built-in structure because you are implementing your own). In implementing this structure, you will be required to adhere to the following rules:

- You must use our `AssassinNode` class for your lists. **You are not allowed to modify it**.
- You may not construct any arrays or `ArrayList`s or other data structures to solve this problem. You must solve it using linked sequences of `AssassinNode` objects. You can examine the list of `String`s passed to the constructor, but you are not allowed to modify it.
- If there are *n* names in the list of `String`s passed to your constructor, you should ask for a new `AssassinNode` exactly *n* times. This means that as people are killed, you have to move their node from the kill ring to the graveyard without creating any new nodes.

The main effect of the rules above is that your constructor will create an initial set of nodes (the initial kill ring) and then your class will not create any more nodes for the rest of the program execution. That means that you need to solve the problem of moving people from the kill ring to the graveyard by rearranging references, not by creating new nodes. You are allowed to declare local reference variables of type `AssassinNode` (like "current" and "prev", for example) because otherwise you can't solve the problem at all. Declaring local reference variables of type `AssassinNode` is not the same as constructing node objects and, therefore, doesn't count against the limit of *n* nodes.

For this assignment, we are specifying what data fields you should have in your class. **You should have exactly two data fields: a reference to the front of the kill ring and a reference to the front of the graveyard. You are not allowed to have any other data fields.**

In lecture, we have been looking at nodes of type `ListNode` that have just two fields: a field called data of type `int` and a field called next that points to the next value in the list. The `AssassinNode` class has three fields. The first two are fields for storing data called name and killer (they are used to store the name of a player and the name of the person who killed that player). The third field is called next and it serves the same purpose as the next field in the `ListNode` class.

For this particular program, it is intuitive to store the kill ring in what is known as a "circular" linked list. Normally lists have the value "null" stored in the next field of the last node of the list. Such lists are known as "null terminated" lists. In a circular list, the final element stores a reference to the first element in the list. But most novices find it difficult to work with a circular list, especially since all of our examples involve null-terminated lists. There is no need to use a circular list to solve the problem, so you are encouraged to solve it with a null-terminated list. If you feel strongly that you want to attempt the circular list, you are allowed to do so, but it is likely to make the program harder to write.

You will want to write your own test client program (`AssassinMain`, for example, never generates any of the exceptions you have to handle). When your class is in good shape, you can use the `AssassinMain` program to make sure it works properly. A log of execution for `AssassinMain` appears at the end of this write-up. Your program should exactly reproduce the format and general behavior demonstrated in the log, although you won't exactly recreate this scenario because of the "shuffling" of the names that `AssassinMain` performs if you choose to shuffle the names.

In terms of correctness, your class must provide all of the functionality described above and satisfy all of the constraints mentioned in this writeup. In terms of style, we will be grading on your use of comments, good variable names, consistent indentation and good coding style to implement these operations.

You should name your file `AssassinManager.java` and should turn it in electronically. A collection of files needed for the assignment is included on Moodle, including some input files for testing. You will need to have `AssassinNode.java, AssassinMain.java`, and whatever names of files of names you are using all in the same directory as your `AssassinManager.java` in order to run `AssassinMain`.

Submission

You should name your file `AssassinManager.java` and you should submit electronically and in hard copy.

Grading
Correct Behavior – 45 pts
Comments, Indentation, Variable Names, Code Style – 15 pts
Program logic – 10 pts

Total – 70 pts

Log of execution (user input underlined)

```
Welcome to the Assassin Manager

What name file do you want to use this time? names3.txt
Do you want the names shuffled? (y/n)? n

Current kill ring:
     Athos is stalking Porthos
     Porthos is stalking Aramis
     Aramis is stalking Athos
Current graveyard:

next victim? Aramis

Current kill ring:
     Athos is stalking Porthos
     Porthos is stalking Athos
Current graveyard:
     Aramis was killed by Porthos

next victim? Athos

Game was won by Porthos
Final graveyard is as follows:
     Athos was killed by Porthos
     Aramis was killed by Porthos
```