CSE 143: Computer Programming II — Autumn 2016

HW3: AssassinManager (due Thursday, October 20th, 2016 11:30pm)

This assignment focuses on implementing a linked list. Turn in the following file using the link on the course website:

- `AssassinManager.java` – A class that manages a game of Assassin.

You will need the support files `AssassinMain.java` and `names.txt`; place these in the same folder as your program or project. If you are using EZclipse, the files will be automatically downloaded for you in the correct location. Do not modify the provided files. The code you submit must work properly with the unmodified versions.

Program Details

"Assassin" is a game often played on college campuses. Each person playing has a particular target that he/she is trying to "assassinate." Generally "assassinating" a person means finding them on campus in public and acting on them in some way (e.g. saying "You're dead," squirting them with a water gun, or touching them). One of the things that makes the game more interesting to play in real life 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 whom the other people are trying to assassinate.

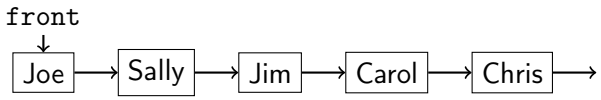
Assassin Rules

- You start out with a group of people who want to play the game
- A circular chain of assassination targets (called the "kill ring" in this program) is established.
- When someone is assassinated, the links need to be changed to "skip" that person.

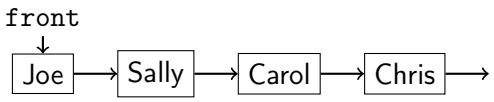
Example Game of Assassin

Let's walk through an example with five people playing: Carol, Chris, Jim, Joe, Sally. We might decide Joe should stalk Sally, Sally should stalk Jim, Jim should stalk Carol, Carol should stalk Chris, and Chris should stalk Joe. In the actual linked list that implements this kill ring, Chris's next reference would be null. But, conceptually we can think of it as though the next person after Chris is Joe, the front person in the list.

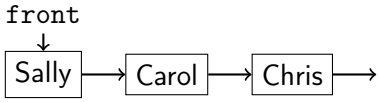
Here is a picture of this "kill ring":



Then, suppose Sally assassinates Jim. Sally needs a new target, so we give her Jim's target: Carol. The kill ring becomes:



If the first person in the kill ring is assassinated, the front of the list must adjust. If Chris kills Joe, the list becomes:



In this assignment, you will write a class `AssassinManager` that keeps track of who is stalking whom and the history of who killed whom. You will maintain two linked lists:

- a list of people currently alive (the "kill ring") and
- a list of those who are dead (the "graveyard").

As people are killed, you will move them from the kill ring to the graveyard by rearranging links between nodes. The game ends when only one node remains in the kill ring, representing the winner.

A client program called `AssassinMain` has been written for you. It reads a file of names, shuffles the names, and constructs an object of your class `AssassinManager`. This main program then asks the user for the names of each victim to kill until there is just one player left alive (at which point the game is over and the last remaining player wins). `AssassinMain` calls methods of the `AssassinManager` class to carry out the tasks involved in administering the game.

AssassinManager

You must use the `AssassinNode` class which we have provided to you as a private inner class. If you download `AssassinManager.java` from the course website or use Ezclipse, the inner class will be included in the file for you.

Our class has the following implementation:

```
1  private static class AssassinNode {
2     public final String name; // this person's name
3     public String killer; // name of who killed this person (null if alive)
4     public AssassinNode next; // next node in the list
5     public AssassinNode(String name) { ... }
6     public AssassinNode(String name, AssassinNode next) { ... }
7  }
```

You cannot change final variables and fields!

`AssassinManager` should have the following fields:

- a reference to the front node of the kill ring
- a reference to the front node of the graveyard (null if empty)

Do **NOT** add a size field!

Note that a requirement of this assignment is that you have *exactly* these two fields and *no others*.

`AssassinManager` should have the following constructor:

`public` **`AssassinManager`**`(List<String> names)`
This constructor should initialize a new assassin manager over the given list of people. Note that you should not save the list parameter itself as a field, nor modify the list. Instead, you should build your own kill ring of list nodes that contains these names in the same order. If the list is null or empty, you should throw an `IllegalArgumentException`.
For example, if the given list contains `["John", "Sally", "Fred"]`, your initial kill ring should represent that John is stalking Sally who is stalking Fred who is stalking John (in that order). You may assume that the names are non-empty, non-null strings and that there are no duplicates.

Do not change the list that is passed in.

`AssassinManager` should also implement the following methods:

`public void printKillRing()`

This method should print the names of the people in the kill ring, one per line, indented by four spaces, as "X is stalking Y". If the game is over, then instead print "X won the game!".

For example, using the names in the example game above, the output is:

```
>>    Joe is stalking Sally
>>    Sally is stalking Jim
>>    Jim is stalking Carol
>>    Carol is stalking Chris
>>    Chris is stalking Joe
```

X and Y are names of the players

Indent the output using four spaces, not tabs!

`public void printGraveyard()`

This method should print the names of the people in the graveyard, one per line, with each line indented by four spaces, with output of the form "name was killed by name". It should print the names in the opposite of the order in which they were killed (most recently killed first, then next more recently killed, and so on). It should produce no output if the graveyard is empty.

For example, using the names from above, if Jim is killed, then Chris, then Carol, the output is:

```
>>    Carol was killed by Sally
>>    Chris was killed by Carol
>>    Jim was killed by Sally
```

`public boolean killRingContains(String name)`

This method should return true if the given name is in the current kill ring and false otherwise. It should ignore case in comparing names; so, "salLY" should match a node with a name of "Sally".

`public boolean graveyardContains(String name)`

This method should return true if the given name is in the current graveyard and false otherwise. It should ignore case in comparing names; so, "CaRoL" should match a node with a name of "Carol".

`public boolean isGameOver()`

This method should return true if the game is over (the kill ring has one person) and false otherwise.

`public String winner()`

This method should return the name of the winner of the game, or null if the game is not over.

`public void kill(String name)`

This method should record the assassination of the person with the given name, transferring the person from the kill ring to the front of the graveyard. This operation should not change the relative order of the kill ring (i.e. the links of who is killing whom should stay the same other than the person who is being killed). This method should ignore case in comparing names.

A node remembers who killed the person in its killer field, and you must set the value of this field.

You should throw an `IllegalStateException` if the game is over, or throw an `IllegalArgumentException` if the given name is not part of the kill ring. If both of these conditions are true, the `IllegalStateException` takes precedence.

Exceptions should be thrown as soon as possible in the method.

Try to write simple code, and use inline comments to clarify anything complex.

The kill method is the hardest one, so write it last. Use the debugger and `println` statements liberally to debug problems in your code. You will likely have a lot of `NullPointerException` errors, infinite loops, etc. and will have a very hard time tracking them down unless you are comfortable with debugging techniques.


Do NOT use nested loops on this assignment!

You will be graded in part by how efficient your code is, especially your `kill` method. In particular, you may **NOT** use nested loops in any method in order to get full credit on this assignment.

Note that there is nothing inherently wrong with nested loops in general, and using nested loops does not necessarily mean your code is inefficient. We will discuss in more detail later this quarter what exactly it means for a program to be "efficient", but for now, all you need to know is that an efficient implementation of `AssassinManager` will not use nested loops any way.

Constraints

This is meant to be an exercise in linked list manipulation. As a result, you must adhere to the following rules while implementing `AssassinManager`:

- You may not construct any arrays, `ArrayLists`, `LinkedLists`, `Stacks`, `Queues`, or other data structures; you must use list nodes. You may not modify the list of Strings passed to your constructor.
- If there are n names in the list of Strings passed to your constructor, you should create exactly n new `AssassinNode` objects in your constructor. As people are killed, you have to move their node from the kill ring to the graveyard by changing references, without creating any new node objects.
- Your constructor will create the initial kill ring of nodes, and then your class may not create any more nodes for the rest of the program. You are allowed to declare as many local variables of type `AssassinNode` (like `current` from lecture) as you like. `AssassinNode` variables are not node objects and therefore don't count against the limit of n nodes.

You should write some of your own testing code. `AssassinMain` requires every method to be written in order to compile, and it never generates any of the exceptions you have to handle, so it is not exhaustive.

Sample Log of Execution


Make sure to match the output *exactly*.

Your program should reproduce the format and behavior demonstrated in this log. Note that you may not exactly recreate this scenario because of the shuffling of the names that `AssassinMain` performs.

```
>> Welcome to the CSE143 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
```

Circular Lists

Some students try to store the kill ring using a "circular" linked list (where the list's final element stores a next reference back to the front). We discourage you from implementing the program this way; instead, you should follow the normal convention of having null in the next field of the last node. It is significantly more difficult to write bug-free code using a circular list. There is no need to use a circular list, because you can always get back to the front via the fields of your `AssassinManager`. If you feel strongly that you want to use a circular list, you may, but it will make the program significantly more difficult to write.

Style Guidelines and Grading

Unless otherwise specified, your solution should use only material covered so far. Part of your grade will come from appropriately utilizing linked lists and nodes as described previously.

Factor out any redundancy in your methods.

Avoid Redundancy

Create "helper" method(s) to capture repeated code. Any helper methods you create must be `private` so outside code cannot call them. If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code.

Data Fields

Properly encapsulate your objects by making data your fields `private`. Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place. Fields should always be initialized inside a constructor or method, never at declaration.

Java Style Guidelines

Appropriately use control structures like loops and if/else statements. Avoid redundancy using techniques such as methods, loops, and factoring common code out of if/else statements. Properly use indentation, good variable names, and types. Do not have any lines of code longer than 80 characters.

Commenting

You should comment your code with a heading at the top of your class with your name, section, and a description of the overall program. All method headers should be commented as well as all complex sections of code. Make sure you describe complex methods inside methods. Comments should explain each method's behavior, parameters, return values, and assumptions made by your code, as appropriate. The `ArrayIntList` class from lecture provides a good example of the kind of documentation we expect you to include. You do not have to use the pre/post format, but you must include the equivalent information– including the type of exception thrown if a precondition is violated. Write descriptive comments that explain error cases, and details of the behavior that would be important to the client. Your comments should be written in your own words and not taken verbatim from this document.