CS 143 Lab 4 Program Specifications

Checkpoint Due 11:59PM Friday July 27
Lab Due 11:59PM Monday July 30

Problem Overview: You will implement and test a class named `AssassinManager` that allows a client to manage a game of Assassin. Each person playing Assassin will be assigned another player that they will target for assassination. The `AssassinManager` class will support a game administrator client 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 Chris, Kelly, Jan, Kim and Casey want to play Assassin. Each player must be assigned a target to assassinate, forming a circular chain of targets. For example, it might be decided that Chris will stalk Kelly, who will stalk Jan, who will stalk Kim who will stalk Casey, who will stalk Chris. The resulting "kill ring" would look like the following:

```
Chris --> Kelly --> Jan --> Kim --> Casey
 ^                                    |

 |                                   V
 +--------<--------<---------<--------+
```

When someone is assassinated, the chain needs to be relinked by "skipping" that person. For example, suppose that Jan is assassinated first (this would have been by Kelly). Kelly needs a new target, so we reassign to them Jan's target Kim. Thus, the chain becomes:

```
          +-------->--------+
          ^                 |

          |                V
Chris-->Kelly     Jan      Kim --> Casey
 ^                                    |

 |                                   V
 +--------<--------<---------<--------+
```

Additionally, Jan should be moved to the graveyard where kills are recorded. **The `AssassinManager` class will manage the kill ring and the graveyard using two linked lists.**

Sample I/O Session: The client program has been written in a file named `AssassinManagerClient.java` that is stored in the `Lab4Files` folder posted with this assignment. Be sure to store this file in the same folder as your `AssassinManager.java` file. With the help of your `AssassinManager` class the client program will produce the following sample I/O session, where user input is shown underlined only for easy identification:

```
Name of game players file? players3.txt

Current kill ring:
    Kenny is stalking Bebe
    Bebe is stalking Token
    Token is stalking Kenny
Current graveyard:
```

```
Next victim? TOKEN

Current kill ring:
    Kenny is stalking Bebe
    Bebe is stalking Kenny
Current graveyard:
    Token was killed by Bebe

Next victim? kenny
Game was won by Bebe
Final graveyard is as follows:
    Kenny was killed by Bebe
    Token was killed by Bebe
```

The Client Program: Familiarize yourself with the `AssassinManagerClient` program. It performs the following tasks:

- Ask the user for the name of a file containing player names.
- Read the file, placing the names it contains in a Java `ArrayList<String>` structure.
- Declare/construct an object of type `AssassinManager`.
- Repeatedly ask the user for the name of the next victim until the game is over—that is, until there is just one player left alive.

The program calls methods of the `AssassinManager` class to carry out the tasks involved in administering the game.

Though an `AssassinManagerClient` will be provided for you, you will want to create your own client tester program to test your class in small stages during the development of your class. You could also achieve this incremental testing capability by commenting out the code in the `AssassinManagerClient` file that tests features of the `AssassinManager` class that you have not yet implemented.

`AssassinManager` Class Specifications: Your class will keep track of two different lists: a list of those players still currently alive in the kill ring, and a list of those who are dead in the graveyard. The kill ring and the graveyard will both be implemented as linked list data structures. **These two lists are the only data structures you are allowed to create in your `AssassinManager` class.**

Aside from the checkpoint below, you must decide the stages of implementation and testing that you will use for this lab. In general, one should rarely write more than a method or two before testing them. Long methods should be implemented and tested in stages.

There are several different files containing players' names in the `Lab4Files` folder for you to test your code on. They are named `players0.txt`, `players1.txt`, `players3.txt` and `players5.txt`.

Lab 4 Checkpoint: Declare the data fields, and implement the constructor method, for the `AssassinManager` class described in the table below.

End Lab 4 Checkpoint

`AssassinManager` Class	
Data Field	**Description**

`private static class AssassinNode {`
 	
   `private String player;`
   `private String killer;`
   `private AssassinNode next;`
 	
   `AssassinNode(String name){`
      `this.player = name;`
      `this.killer = null;`
      `this.next = null;`
   `}`
`};`
	This `AssassinNode` class is nested within the `AssassinManager` class.
	
	←this node's player's name
	←name of this node's player's killer
	←node following this one in a linked list
	
	←`AssassinNode` constructor
	
	The name of the player's killer and the link to the next node in the list will be set to non-null values by other methods in the **`AssassinManager`** class.

`private AssassinNode killring;`	The head of the kill ring linked list.
`private AssassinNode graveyard;`	The head of the graveyard linked list.
NO OTHER DATA FIELDS ALLOWED	If you go online searching for linked list discussions, you may discover references to such helpful items as `tail` or `size` fields—but you will lose partial points if you add such to your **`AssassinManager`** class.
Method	**Description**

`public AssassinManager(ArrayList<String>`
                    `playerNamesList)`
 	
NOTE: Even though the kill ring is depicted as circular in the overview, it will be implemented as a singly-linked, null-terminated linked list.
	This `AssassinManager` constructor creates the kill ring implemented as a linked list.
	• Instead of creating the kill ring, you should throw an appropriate exception if the `ArrayList` is empty or `null`.
	• There will be one node for each name in the `ArrayList` parameter
	• Each name is added to the linked list in the same order that it appears in the `ArrayList`.
	• The last node added to the list will have a `null next` field. (See the note at left.)
	• This method should not alter the `ArrayList` in any way.
	• **This is the only method in this class that will use the `new` operator to construct `AssassinNode` objects.**

`AssassinManager` class specifications continued on next page.

AssassinManager Class	
Method	**Description**
Customized private helper methods of your own design	You are welcome to design and implement any private methods that will help your public methods perform their tasks.
`public void printKillRing()` NOTE: Though the picture of the kill ring in the overview makes the list appear circular, the last player added to the list simply has a `null next` field. That is, it is understood that the first player in the list is targeted for assassination by the last player in the list. You may wish to review BJP 5.2 on *fencepost algorithms* to help you design an elegant implementation for this method that doesn't require if-statements inside your loop.	This method should print the names of the people in the kill ring, from beginning to end of the list, one per line, indented four spaces, with output of the form "??? is stalking ???" as shown in the sample I/O session. If there is only one person in the kill ring, it should report that the person is stalking themselves (e.g., "Ari is stalking Ari"). **It should produce no output if the kill ring is empty or null.**
`public void printGraveyard()`	This method should print the names of the people in the graveyard, from beginning to end of the list, one per line, indented four spaces, with output of the form "??? was killed by ???" as shown in the sample I/O session above. **It should produce no output if the graveyard is empty or null.**
`public boolean killRingContains(String name)`	This method returns `true` if the `name` given by the parameter is in the current kill ring; otherwise it should return `false`. **It should ignore case when comparing names.** It should return `false` if the kill ring is empty or `null`. This method does not display anything.
`public boolean graveyardContains(String name)`	This method returns `true` if the `name` given by the parameter is in the graveyard; otherwise it should return `false`. **It should ignore case when comparing names.** It should return `false` if the graveyard is empty or `null`. This method does not display anything.

AssassinManager class specifications continued on next page.

`AssassinManager` Class	
Method	**Description**
`public boolean gameOver()`	The game is over if there is only one player remaining in the kill ring. This function returns `true` if the game is over, and `false` otherwise.
`public String winner()`	This method should return the name of the winner of the game, if the game is over. It should return the empty string ("") if the game is not over.
`public void kill(String name)`	This method manages the killing of the person with the name given by the parameter (ignore case), by transferring their node from the kill ring to the graveyard. The transfer must adhere to the following specifications: • The method should throw an appropriate exception if the name given in the parameter list is not part of the current kill ring, or if the game is over. • The killer's name should be recorded in the newly killed person's node. • All other players should remain in their current positions in the kill ring. • The killer should be reassigned the newly killed player's target. • *When the newly killed player is added to the graveyard, they are always placed at the beginning of the list.* In this way the graveyard lists kills from most recent to oldest. • The method should call the `killRingContains()` and `gameOver()` methods appropriately to do its job. • **This method does not create any new nodes or any other data structures. The updating of the `killring` and `graveyard` lists is achieved entirely by managing the links in the lists appropriately.**

Style Specifications: Be sure to follow the *Good Style Specifications* described in the document by that name in the Quick Links module on Canvas.