CS 143 Lab 7 Program Specifications

There is no checkpoint for this lab
Lab Due 11:59PM Monday Aug 20

Purpose and Overview: You will get practice using professional API documentation, *iterators*, and major concepts utilized in earlier labs this quarter. You will do this by solving the `AssassinManager` problem employing the following Java language elements:

- *Encapsulation* from Lab 1, and use of JCF classes in Lab 3: The `AssassinNode` class will be a stand-alone class, rather than being nested within the `AssassinManager` class. The `AssassinNode` class will no longer require a `next` field because we will use the JCF `LinkedList` class to manage the nodes.
- *Inheritance* from Lab 2: You will implement a `KillList` class that will *extend* the `LinkedList<AssassinNode>` class.
- Defining `equals()`, `indexOf()` and `toString()` from Lab 3: If you are going to use JCF classes to search and/or display the `AssassinNode` and/or `KillList` objects, you must provide support necessary for the JCF classes to be able to do so.
- *Regular expressions* and the `split()` method from Lab 6: It will be necessary to manipulate the `KillList.toString()` result to achieve the desired display for the needs of the `AssassinNode.printKillRing()` method.

Sample I/O Session: You will edit the `AssassinManagerClient` class, so that it can produce the following sample I/O session, which is identical to that for Lab 4, but with the addition of the last six lines that are the result of code used to test the `graveyardContains()` method:

```
Name of players file? players3.txt

Current kill ring:
    Kenny is stalking Bebe
    Bebe is stalking Token
    Token is stalking Kenny
Current gravyard:

Next victim? TOKEN

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

Next victim? kenny

Game was won by Bebe
Final gravyard is as follows:
    Kenny was killed by Bebe
    Token was killed by Bebe
Enter victim name: (QUIT to quit) KENNY
graveyardContains("KENNY") is true
Enter victim name: (QUIT to quit) token
```

```
graveyardContains("token") is true
Enter victim name: (QUIT to quit) QUIT
graveyardContains("QUIT") is false
```

Preparation: You should download and unzip the compressed `Lab7Files` folder posted with this assignment. It contains the `AssassinManagerClient.java` program that can produce the I/O session above if the `AssassinNode`, `KillList` and `AssassinManager` classes are all implemented correctly, and the additional tester code is added. The folder also contains the `players1.txt`, `players3.txt` and `players5.txt` data files. All files related to Lab 7 should be stored in the `Lab7Files` folder, or a copy thereof.

The `AssassinNode` Class Specifications: The `AssassinNode` class will be an independent class, rather than an inner nested class within the `AssassinManager` class as it was in Lab 4.

`AssassinNode` Class	
Data Fields	**Description**
`private String player;` `private String killer;`	Notice that the `next` field is no longer needed because the JCF `LinkedList` class will be used to manage our list of `AssassinNode` objects.
Method	**Description**
`AssassinNode(String name)`	This constructor should assign the `player` field the value of the `name` parameter, unless the parameter is null or empty, in which case it should `throw` an appropriate exception.

`public boolean equals(Object other)`
This method is defined to facilitate the following:
if(a1.equals(a2))… //a1 and a2 are AssassinNodes
as well as
if(a1.equals("Token"))…
	• Two `AssassinNode` objects are equal if their `player` fields are equal—ignoring case.
	• "This" `AssassinNode` object is equal to a String parameter object if the `player` field and the String object are equal—ignoring case.

`getPlayer()`, `getKiller()`	These are get/accessor methods for the private data fields. Define them appropriately.
`setPlayer()` and `setKiller()`	These are set/mutator methods. Define them appropriately. They should throw an appropriate exception if their parameters are null or empty.

Test the `AssassinNode` class: You are strongly encouraged to write a simple tester client to make sure all the methods of your `AssassinNode` class work correctly.

The `KillList` Class Specifications: The `KillList` class should be a *subclass* of the `LinkedList<AssassinNode>` *superclass*. This means that it will inherit all the non-constructor methods described in related sections of BJP Chapter 11, plus those described in the complete documentation on the following website:
https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html .

Consideration: Since your `KillList` subclass inherits all the public methods used to manage the private linked list structure that is being managed by the `LinkedList<AssassinNode>` superclass, you **do not use *object.method()* notation** when calling those inherited methods. Simply call them directly. If you feel compelled to use dot notation, use `super`.method().

- Remember that *constructors are not inherited*, so you should provide a default (no-argument) constructor that calls the superclass default constructor. (Recall how this is done in BJP Chapter 9.)
- The `KillList` subclass should override the superclass's `indexOf()` method as follows:
  - If the parameter is an `AssassinNode` object, return the superclass's `indexOf()` value for that parameter.
  - If the parameter is a `String` object, return the superclass's `indexOf()` value when it is passed an `AssassinNode` object constructed from the parameter's value.
  - Return false under all other circumstances
- The `KillList` subclass should override the superclass's `toString()` method. It should return the empty string if the list managed by the superclass is empty. Otherwise, suppose the data structure being managed by the superclass looks like the following:

  Kenny → Bebe → Token

The `toString()` method should return the following `String` value:

```
"Kenny#Bebe\nBebe#Token\nToken#Kenney\n"
```

That is, each node in the superclass linked list structure should have a string representation of the following format:

```
"stalker#player\n"
```

If the superclass linked list structure is empty, `toString()` should return the empty string. If the list contained only one node, the stalker and player would have the same value.

Requirement: You must use a `ListIterator<AssassinNode>` object to implement the `KillList.toString()` method. The `ListIterator` class is more flexible than the `Iterator`

class described in BJP 11.1. As the authors stated, you can learn more about the `ListIterator` class on the following Java API and Tutorial pages.

https://docs.oracle.com/javase/7/docs/api/java/util/ListIterator.html

https://docs.oracle.com/javase/tutorial/collections/interfaces/list.html

(You will probably want to skip down to the *Iterators* section of the Tutorial page.)

Test the `KillList` class: You are strongly encouraged to write a simple tester client to make sure `KillList` class working properly.

The `AssassinManager` Class Specifications: The `AssassinManger` class should behave as it did for Lab 4, but should be implemented according to the specs in the table below.

`AssassinManager` Class	
Data Field	**Description**
`private KillList killring = new KillList();`	Empty `killring`
`private LinkedList<AssassinNode> graveyard = new LinkedList<AssassinNode>();`	Empty `graveyard`
NO OTHER DATA FIELDS ALLOWED	The `LinkedList` JCF class has immediate access to the front and rear of the list, and is doubly linked to support forward and backward traversal.
Method	**Description**

`public void printKillRing()`
`___ is stalking ___`
`___ is stalking ___`
`___ is stalking ___`
`Etc.`
	Use the `killring.toString()` method, together with the split() method, to produce the same 4-space indented line of output for each node in the kill ring as was done in Lab 4, and is depicted at the left.

`public void printGraveyard()`
`___ was killed by ___`
`___ was killed by ___`
`___ was killed by ___`
`Etc.`
	Produce the same 4-space indented line of output for each node in the graveyard as was done in Lab 4, and is depicted at the left.

`AssassinManager` Class	
Method	**Description**
`public boolean gameOver()`	This method returns `true` if the size of the kill ring is 1.
`public String winner()`	Return the name of the winner of the game, if the game is over, otherwise return the empty string.
`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`. **Use a `ListIterator` to implement this method.**
`public void kill(String name)`	Throw an appropriate exception if the game is over, or the `name` does not appear in the kill ring. Otherwise, record the killer's name in the kill ring node containing `name`, and remove that node from the kill ring and place them at the front of the graveyard. Accomplish these tasks using JCF `LinkedList` methods.

Style Specifications: Be sure to follow the *Good Style Specifications* described in the document by that name in the Quick Links module on Canvas. Also, for this lab ***the opening comments in the AssassinManager.java file should contain the following information:***

- Programmer name
- Purpose of code in file
- Total hours spent on Assignment 7
- For 1.5 points of the 2-point Lab 7 style grade, the answer to the following question:

What is the relative computing time for the `kill()` method using Big Oh notation? Justify your answer.
.5 pts for Big Oh notation
.5 pts for justification
.5 pts for correctness of answer