Overview: The Assassin Game

“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 tagging 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. That is, the person who was assassinated passes their target on to the person who assassinated them.

Example Game of Assassin

Let’s walk through an example with five people playing: Carol, Chris, Jim, Joe, Sally. We might start with Joe stalking Sally, Sally stalking Jim, Jim stalking Carol, Carol stalking Chris, and Chris stalking 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”:

```
front
Joe ─── Sally ─── Jim ─── Carol ─── Chris
```

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

```
front
Joe ─── Sally ─── Carol ─── Chris
```

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

```
front
Sally ─── Carol ─── Chris
```
Program Behavior

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

- a list of people currently alive (the “kill ring”) and
- a list of those who have been assassinated (the “graveyard”).

As people are assassinated, 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 assassinate 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

To implement your lists, you must use our `AssassinNode` class provided on the course website without modification. The class is summarized here:

```java
public class AssassinNode {
    public final String name; // this person's name
    public String killer; // name of who killed this person (null if alive)
    public AssassinNode next; // next node in the list

    public AssassinNode(String name) { ... }
    public AssassinNode(String name, AssassinNode next) { ... }
}
```

In class section 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 assassinated that player). The third field is called next and it serves the same purpose as the next field in the `ListNode` class.
Your **AssassinManager** class must have exactly 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)

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

Your **AssassinManager** class should have the following constructor:

```java
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 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.

Your **AssassinManager** class should also implement the following methods:

```java
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 is stalking X”.

For example, using the kill ring from the example game on the first page of this spec, the output is:

- Joe is stalking Sally
- Sally is stalking Jim
- Jim is stalking Carol
- Carol is stalking Chris
- Chris is stalking Joe

If the game is over and Chris is the winner, so Chris is the only name in the kill ring, the output is:

- Chris is stalking Chris

```java
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 “X was killed by Y”. It should print the names in the opposite of the order in which they were assassinated (most recently assassinated first, then next most recently assassinated, and so on). It should produce no output if the graveyard is empty.

For example, using the kill ring 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

```java
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”.

---

Page 3 of 6
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 gameOver()
This method should return true if the game is over (i.e. the kill ring contains exactly 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 yet.

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 stalking 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.

Your method 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.

The kill method is the hardest to complete, so we strongly suggest you write it last. Use the jGRASP 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.

Implementation Guidelines
The learning objectives for this assessment are explicitly related to manipulating linked lists. To that end, you are limited in how you may implement the operations required for AssassinManager. Specifically, you must adhere to the following rules:

- You may not construct any arrays, ArrayLists, LinkedLists, Stacks, Queues, or other data structures; you must use instances of AssassinNode and manipulate them yourself.

- If there are n names in the list of Strings passed to your constructor, you must create exactly n new AssassinNode objects in your constructor. You may not create any additional node objects, and you may not create node objects in any other methods. In addition, you may not modify the name field of nodes after they have been created. As people are assassinated, you will move the existing node from the kill ring to the graveyard by changing references. You must not create any new node objects or change the name field of the nodes.

- You may declare as many references to AssassinNode objects (i.e. local variables of type AssassinNode as you like. AssassinNode references are not node objects; they don’t count against the node limit.

- Your constructor should be “efficient” in the sense that it should not use any nested loops to construct the initial kill ring. (We will learn in class that this is called O(n) time, where n is the number of names in the list.)
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). It is significantly more difficult to write bug-free code using a circular list. There is no need to use a circular list for this assessment, 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 we believe it will make the program significantly more difficult to write, and we strongly discourage it. We will not provide assistance in office hours to help you implement the circular list solution.

jGRASP Debugger
We strongly recommend that you use the jGRASP debugger for this assessment, even if you are primarily working in another IDE. The jGRASP debugger has a structure viewer to see what your list looks like by dragging one of your fields from the debug window outside the window. By default the viewer won’t show you the name in each node (it will show a “?” instead). Fix this by clicking the wrench icon, then in the “Value Expressions” box, type: _node_.name, Click OK, and you should see the names in the nodes. You can also drag the width scrollbar to see the names better.

You can also find a video tutorial here.

Code Quality Guidelines
In addition to producing the behavior described above, your code should be well-written and meet all expectations described in the General Style Deductions, Style Guide, and Commenting Guide. For this assessment, pay particular attention to the following elements:

Avoid Redundancy
If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code. All helper methods should be declared as private.

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.

Exceptions
The specified exceptions must be thrown correctly in the specified cases. Exceptions should be thrown as soon as possible, and no unnecessary work should be done when an exception is thrown. Exceptions should be documented in comments, including the type of exception thrown and under what conditions.

Commenting
Each method should have a header comment including all necessary information as described in the Commenting Guide. Comments should be written in your own words (i.e. not copied and pasted from this spec) and should not include implementation details.
Running and Submitting

While developing your code, you should work locally on your computer in jGRASP. To test your code, you should run the provided test files and copy the printed logs into the Output Comparison Tool to check against the expected output.

If you believe your behavior is correct, you can submit your work by uploading your files and clicking the "Mark" button in the Ed assessment. You will see the results of some automated tests along with tentative grades. **These grades are not final until you have received feedback from your TA.**

You should treat Ed as a final check for your correctness, not as a debugging tool. If you aren’t passing the tests, you should go back to jGRASP to debug your work.

You may submit your work as often as you like until the deadline; we will always grade your most recent submission. If you submit a version that you later decide you do not want to have graded, **you must warn your TA not to grade that version and to wait for a later submission from you.**

Getting Help

If you find you are struggling with this assessment, make use of all the course resources that are available to you, such as:

- Reviewing relevant examples from class
- Reading the textbook
- Visiting the IPL or Taylor’s office hours
- Posting a question on the Ed discussion board

Collaboration Policy

Remember that, while you are encouraged to use all resources at your disposal, including your classmates, **all work you submit must be entirely your own.** In particular, you should **NEVER** look at a solution to this assessment from another source (a classmate, a former student, an online repository, etc.). Please review the full policy in the syllabus for more details and ask the course staff if you are unclear on whether or not a resource is OK to use.