CS 171: Introduction to Computer Science II
Assignment #3: Playlist Application

Due: Monday Oct 26 at 11:59 PM on Canvas [Late submission: see syllabus for policy]
[2 points] Mid-Semester Evaluation: Please fill our CS171 mid-semester evaluation survey here: https://forms.gle/1r3kF6QjiAJS7ZZT7. There are no right or wrong answers. Your feedback is important to continuously improve how the course is delivered, especially in these unusual, pandemic-driven circumstance. Your ID will be automatically **detached** from your answers and used only to verify that you have submitted the survey:-)

Problem Description: We are interested in developing a Playlist application that supports adding, deleting, and navigating through different Podcast episodes. To make navigating our Playlist flexible, we looked for a data structure that supports dynamic adding and removing of episodes, while also supporting easy and smooth traversing back and forth between episodes.

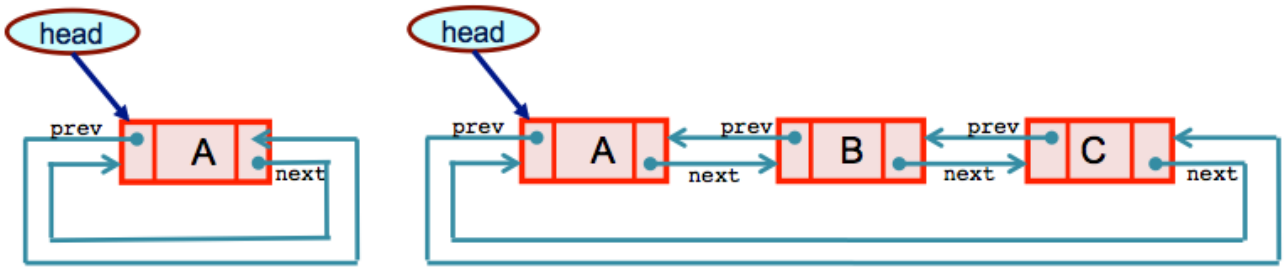
For these reasons, we will be using a **Circular Doubly Linked List** to implement our Playlist. It is circular because the last element in the list points to the first element, thus completing a cycle (circle) of links. And it is doubly because each node has two links, one that refers to the next node and another that referes to the previous node. For our Playlist, this design choice means it will be easy to move forward or backward from any given episode. Starter code is provided; it includes:

(1) Class **Episode** represents an individual Podcast episode. You can think of it as the equivalent of a 'Node' class in our lecture examples, but with more fields. Each episode has a title, length (duration), a link to the next episode, and a link to the previous episode. You must not modify or add anything to this class.

(2) Class **Playlist** represents a circular doubly linked list of Episode objects. The playlist is designed to always maintain a reference to the `head` (i.e. the first episode in the list), and a `size` integer (i.e. the total number of episodes so far). You will implement all the operations supported by this Playlist by filling the code for its member methods. Remember to always make sure that the Playlist's `head` and `size` variables are updated properly after/during each supported operation. **The file Playlist.java is the only file you must submit to Canvas.**

(3) Finally, class **ITunes** represents the application that will test the different features supported in your Playlist. You can modify the code in this class as you wish.

Note that the Playlist may contain no episodes (e.g. when it is first created), a single episode, or multiple episodes. The figure below shows how the nodes should be linked under these different scenarios. Pay attention to how `next` and `prev` are wired. Also, note that de-referencing links in your code more than once is possible; so `episodeObject.prev.next` is valid (assuming `episodeObject` is an object of type Episode).



(a) Example of a single-node circular doubly linked list.

(b) Example of a circular doubly linked list with multiple nodes.

You need to fill in the implementation for the following methods in class Playlist. Do not worry about duplicates for this assignment (i.e. you can assume that no episode will be duplicated and that no two episode titles will be identical).

- **[5 points]** `public void displayPlaylistBackward()`
  Unlike displayPlaylistForward() which is available to you in the starter code, this method prints the episodes in the current Playlist in a reverse order. It starts with printing the last episode, then moves backwards until it reaches the beginning of the playlist. See starter code comments for details on the exact format of the expected output. Your output format must match that expected one for complete marks.

- **[16 points]** `public void addFirst( String title, double length )`
  Create a new Episode using the given title and length parameters, then add this Episode properly at the beginning of the current Playlist.

- **[16 points]** `public void addLast( String title, double length )`
  Create a new Episode using the given title and length parameters, then add this Episode properly at the end of the current Playlist.

- **[16 points]** `public void add( String title, double length, int index )`
  Create a new Episode using the given title and length parameters, then add this Episode properly at the given index. Assume that index zero corresponds to the first node, and so on.

- **[18 points]** `public Episode deleteEpisode(String title)`
  This method should properly delete and return the Episode with the given title in the parameter.

- **[22 points]** `public Episode deleteEveryMthEpisode(int m)`
  One day, a bored manager at Apple decided that iTunes should support a new feature where the user only provides an unlucky number *m*, and the application would go on deleting every *m*-th episode in the user's Playlist (assuming a circular structure), until only one Episode survives. It is possible that this manager was reading about the Josephus problem the night before, which can be summarized as: given a group of n people arranged in a circle under the edict that every *m*-th person will be 'removed' going around the circle until only one remains. For example, if 6, 5, 4, 3, 2, 1, 0 are arranged in circle, with `m` = 3, then the order of removal is 4, 1, 5, 0, 2, 6, with 3 remaining as the survivor. Your job is to implement that feature for our Playlist, given the method parameter `m`. This method should return the last surviving Episode in the Playlist.

Dealing with Exceptions: For all the methods described above, if the user attempts an invalid operation (e.g. deleting from an empty Playlist, etc.), you should throw a proper exception with a meaningful error message. This is similar to the approach we followed in our Linked List code examples in class. Below is an example of throwing a custom exception in Java:

```
throw new RuntimeException("[Error] Cannot delete episode from an empty Playlist!");
```

Read the comments in the starter code carefully as you implement these methods. You can add more methods as needed. **However, do not change the interface of any of the methods described above (i.e. method name, parameters, and return type).** If you make changes for testing and debugging purposes, make sure to comment them out before submitting your file to Canvas. You are only required to submit the file `Playlist.java`.

Important Grading Notes:

- If your program does not compile, you will get 0 points :-(. **Please make sure to remove the local package declarations (if any) from your Java files before the submission.** If your program does compile and run, here's the breakdown of the marks:

- Correctness (the program successfully executes all Playlist operations and does not crash): 95 points
- Code clarity and style: 5 points

Discussion and asking for clarifications: Please utilize our course Piazza page to post questions about the assignment if you need a clarification about any of the methods.

Honor Code: The assignment is governed by the College Honor Code and Departmental Policy. Remember, any code you submit must be your own; otherwise you risk being investigated by the Honor Council and facing the consequences of that. We do check for code plagiarism (across student groups and against online resources). Please remember to have the following comment included at the top of the file.

```
/*
THIS CODE WAS MY OWN WORK, IT WAS WRITTEN WITHOUT CONSULTING
CODE WRITTEN BY OTHER STUDENTS OR COPIED FROM ONLINE RESOURCES.
 _Student_Name_Here_
*/
```