CS 1027 Computer Science Fundamentals II
Assignment 2
Due date: October 31, 11:55 pm

1. Learning Outcomes

In this assignment, you will get practice with:

- Working with single and doubly linked data structures
- Working with exceptions
- Programming according to specifications
- Designing algorithms in pseudocode and implementing them in Java

2. Introduction

One important tool for managing digital media is the Playlist System. A playlist system allows users to create, modify, and use collections of their favorite media.

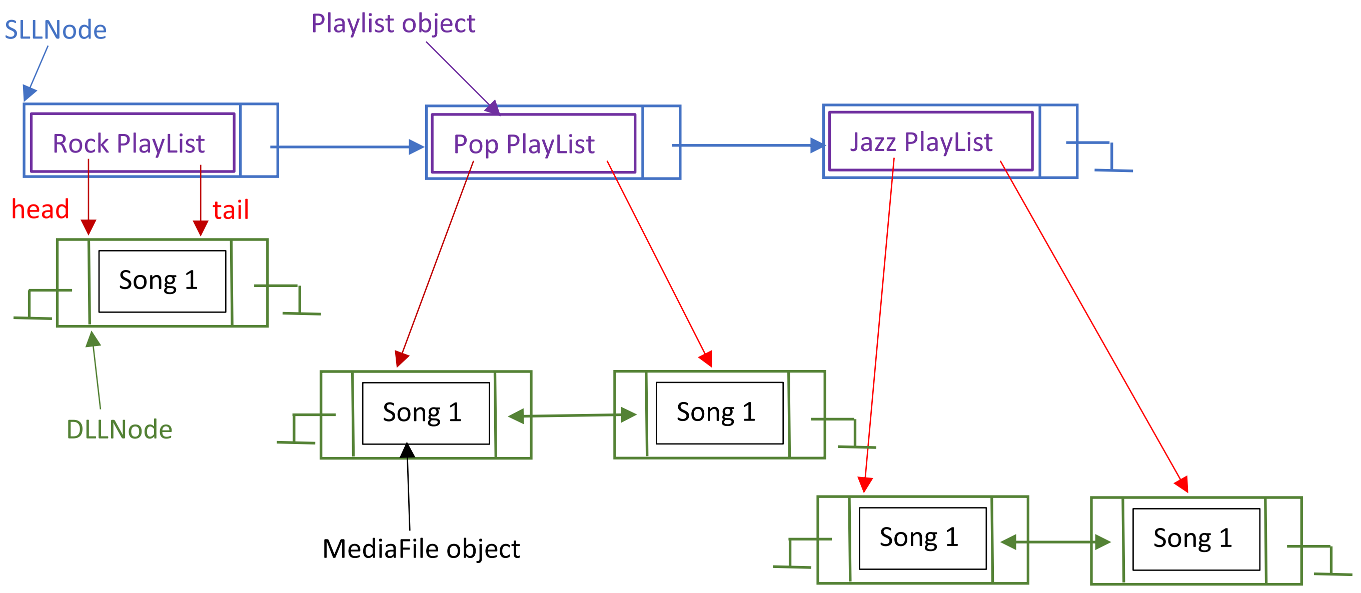
Key aspects of a Playlist System include:

1. **Hierarchical Organization**: Playlists serve as containers for individual media files, allowing for logical grouping and easy access.
2. **Flexible Navigation**: Users can move both forward and backward within a playlist, as well as switch between different playlists.
3. **Metadata Management**: Each playlist and media file contains relevant metadata (e.g., name, duration, artist) for improved organization and search capabilities.
4. **Dynamic Content**: Playlists can be easily modified by adding or removing media files, or by reordering the content.

In this assignment, you will implement a simplified version of a Playlist System using a nested linked list data structure. The outer structure will be a Singly Linked List (SLL) representing the playlists, while each playlist will contain a Doubly Linked List (DLL) of media files.

The following figure shows an example of how the playlist system might look when represented as a nested linked list structure.

In the figure, each node in the outer SLL represents a playlist, with blue arrows indicating the links between playlists. A DLL represents the media files within each playlist node, allowing for bidirectional traversal within the playlist.



3. Classes Provided

- SLLNode.java: A java class representing the nodes in the singly-linked list you will use to implement the playlist structure in the PlaylistManager class.
- DLLNode.java: A java class representing nodes in the doubly-linked list you will use to implement the media file structure within the Playlist class.
- PlayerException.java: A java class representing exceptions that may occur during playlist and media player operations.
- TestAsmt2.java: A java class to help check if your Java classes are implemented correctly.

Additional tester files will be incorporated into Gradescope's auto-grader; these tests will not be given to you to encourage you to thoroughly test your code.

4. Classes to Implement

For this assignment, you must implement three Java classes: MediaFile, Playlist, and PlaylistManager. Follow the guidelines for each one below.

In these classes, you may implement more private (helper) methods if you want. However, you may not implement more public methods.

You may **not** add instance variables other than the ones specified in these instructions nor change the variable types or accessibility (i.e., making a variable public when it should be private). Penalties will be applied if you use additional instance variables or change the variable types or modifiers from what is described here.

You may not import any Java libraries, such as **Java.util.Arrays** or **Java.util.LinkedList.**

4.1 Class MediaFile.java

This class represents media files storing title, artist, duration in seconds, and file type. The class must have the following private instance variables:

- private String title
- private String artist
- private int durationSeconds
- private String fileType

The class must have the following public methods:

- public MediaFile(String title, String artist, int durationSeconds, String fileType): this is the constructor for the class that initializes the instance variables

  The following four getter methods return the value of the corresponding instance variables:

- public String getTitle()
- public String getArtist()
- public int getDurationSeconds()
- public String getFileType()

- public String toString(): returns a string with the following format:

  Title – Artist (*minutes*:*seconds*)

  where *Title* is the String stored in instance variable title, *Artist* is the String stored in artist. To get minutes and seconds use minutes = durationSeconds / 60 and seconds = durationSeconds % 60 (% is the modulo operator in java which gives the remainder of the division). To convert minutes to a String you can use String.valueOf(minutes); recall that String is the name of the static object for class String.

  The value of seconds must be specified with 2 digits, so if for example title = “My song”, artist = “Me”, minutes = 5 and seconds = 2 method toString must return the String

  “My song - Me (5:02)”

  Hint. If seconds < 10 then convert seconds to a String and then prepend “0” to the String. The operator “+” can be used to concatenate two strings. You can also use method String.format() to create the string with 2 digits (read online the documentation of class String to learn how to use method format()).

4.2 Class Playlist.java

This class represents a playlist, implemented as a doubly-linked list of MediaFile objects.

The class must have the following private instance variables:

- private String name
- private DLLNode<MediaFile> head
- private DLLNode<MediaFile> tail
- private int size

This class must have the following public methods:

- public Playlist(String *name*): the constructor for the class initializes instance variable name (how do you initialize the instance variable if the parameter has the same name?). It also initializes instance variables head and tail to null and size to 0.
- public void addMedia (MediaFile *media*) throws PlayerException:
    - A PlayerException is thrown if *media* is null or if in the doubly linked list referenced by head there is a node storing a MediaFile object with the same title as the title stored in the object referenced by parameter *media*.
      Note. When an exception is thrown **in any** of your methods you must store in the exception a message describing the error that caused the exception to be thrown.
    - If no exception is thrown, then a new DLLNode storing media is added to the rear of the doubly linked list and the value of instance variable size is increased.
- public MediaFile removeMedia (String *title*) throws PlayerException:
    - A PlayerException is thrown if no node in the doubly linked list stores a MediaFile object with the same title as *title*.
    - Otherwise, the node storing the MediaFile with the given *title* is removed from the doubly linked list, the size of the list is updated, and the MediaFile object that was stored in the removed node is returned.
- public MediaFile getNextMedia (String currentTitle) throws PlayerException:
    - A PlayerException is thrown if the doubly linked list is null or if no node of the doubly linked list stores a MediaFile object with title equal to currentTitle.
    - Otherwise,
        - If currentTitle = null return the MediaFile object stored in the first node of the list (the node referenced by head).
        - If the node p storing a MediaFile with title equal to currentTile is not the last node in the list, then return the MediaFile object stored in the node p.getNext().
          Otherwise, return the MediaFile object stored in the first node of the list.
- public MediaFile getPreviousMedia (String currentTitle) throws PlayerException:
    - A PlayerException is thrown if the doubly linked list is null, or if no node of the doubly linked list stores a MediaFile object with title equal to currentTitle.
    - Otherwise,
        - If currentTitle = null return the MediaFile object stored in the last node of the list (the node referenced by tail).
        - If the node p storing a MediaFile with title equal to currentTile is not the first node in the list, then return the MediaFile object stored in the node p.getPrevious().
        - Otherwise, return the MediaFile object stored in the last node of the list.

- public int getSize(): returns the number of nodes in the doubly linked list.

- public String getName(): returns instance variable name.
- public String toString(): returns a String with the following format:

  “Playlist: name_of_playlist

  1. MediaFile1

  2. MediaFile2

  …

  k. MediaFilek”

  where name_of_playlist is the String stored in instance variable name, MefiaFile1 is the String obtained by invoking method toString on the MediaFile object stored in the first node of the doubly linked list, MediaFile2 is the String obtained by invoking toString on the MediaFile object stored in the second node of the list, and so on. In the above example we assume that the number of nodes in the doubly linked list is k.

  To separate the String into lines, use the end-of-line character: \n. So, for example this String: “Playlist: name_of_playlist\n1. MediaFile1\n2. MediaFile2\n … \nk. MediaFilek” will be displayed as shown above.

4.3 Class PlaylistManager.java

This class represents the entire media player system, implemented as a singly-linked list of Playlist objects.

The class must have the following private instance variables:

- private SLLNode<Playlist> head
- private int numPlaylists

The class must have the following public methods:

- public PlaylistManager(): the constructor for the class the initializes head to null and numPlaylists to 0.
- public void addPlaylist(Playlist newPlaylist) throws PlayerException:
  - A PlayerException is thrown if playlist is null or if there is already a Playlist object stored in the singly linked list with the same name as newPlayList.
  - Otherwise, a new SLLNode storing newPlaylist is added to the end of the singly linked list.
- public Playlist removePlaylist(String listName) throws PlayerException:
  - A PlayerException is thrown if no Playlist object with name equal to listName is stored in any of the nodes of the list.
  - Otherwise, the node p storing a Playlist object with the same name as listName is removed from the singly linked list and the Playlist object stored in p is returned.

- public Playlist getPlaylist(String listName) throws PlayerException:
    - A PlayerException is thrown if no Playlist object with name equal to listName is stored in any of the nodes of the list.
    - Otherwise, the Playlist object stored in the singly linked list with name equal to listName is returned.
- public void addMediaToPlaylist(String playlistName, MediaFile *media*) throws PlayerException:
    - A PlayerException is thrown if the singly linked list does not store a Playlist object with name equal to playlistName.
    - A PlayerException is also thrown if a Playlist object L with name playlistName is in the singly linked list, but L stores a MediaFile object with the same title as *media*.
    - Otherwise, a DLLNode storing *media* is added to the Playlist object with the same name as playlistName that is stored in the singly linked list.
- public MediaFile removeMediaFromPlaylist(String playlistName, String mediaTitle) throws PlayerException:
    - A PlayerException is thrown if there is no Playlist object with name equal to playlistName stored in the singly linked list.
    - A PlayerException is also thrown if the Playlist object with name playlistName does not store a MediaFile object with the same title as mediaTitle.
    - Otherwise, the DLLNode p storing a MediaFile object with title mediaTitle is removed from the doubly linked list and the MediaFile object stored in p is returned.
- public String toString(): returns a String with the following format:

  ```
  "Playlist Manager:
  Play_list1
  Play_list2
      …
  Play_listq"
  ```

  where Play_list1 is the String returned by the toString() method invoked from the Playlist object stored in the first node of the singly linked list, Play_list2 is the String returned by the toString() method invoked from the Playlist object stored in the second node of the singly linked list, and so on.

5. Examples

Example 1: Creating a Playlist and Adding Media

Code:

```
PlaylistManager manager = new PlaylistManager();
Playlist rockPlaylist = new Playlist("Rock Classics");
manager.addPlaylist(rockPlaylist);
MediaFile song1 = new MediaFile("Stairway to Heaven", "Led Zeppelin", 482, "mp3");
MediaFile song2 = new MediaFile("Bohemian Rhapsody", "Queen", 354, "mp3");
manager.addMediaToPlaylist("Rock Classics", song1);
manager.addMediaToPlaylist("Rock Classics", song2);
System.out.println(manager.toString());
```

Program Output:
```
Playlist Manager:
Playlist: Rock Classics
1. Stairway to Heaven - Led Zeppelin (8:02)
2. Bohemian Rhapsody - Queen (5:54)
```

Example 2: Navigating Through a Playlist
Code:
```
    PlaylistManager manager = new PlaylistManager();
    Playlist popPlaylist = new Playlist("Pop Hits");
    manager.addPlaylist(popPlaylist);
    MediaFile song3 = new MediaFile("Shape of You", "Ed Sheeran", 233, "mp3");
    MediaFile song4 = new MediaFile("Uptown Funk", "Mark Ronson ft. Bruno Mars", 270, "mp3");
    manager.addMediaToPlaylist("Pop Hits", song3);
    manager.addMediaToPlaylist("Pop Hits", song4);
    Playlist currentPlaylist = manager.getPlaylist("Pop Hits");
    MediaFile currentSong = currentPlaylist.getNextMedia(null); // Get first song
    System.out.println("Now playing: " + currentSong.getTitle());
    currentSong = currentPlaylist.getNextMedia(currentSong.getTitle());
    System.out.println("Next song: " + currentSong.getTitle());
    currentSong = currentPlaylist.getPreviousMedia(currentSong.getTitle());
    System.out.println("Previous song: " + currentSong.getTitle());
```

Program Output:
```
Now playing: Shape of You
Next song: Uptown Funk
Previous song: Shape of You
```

Example 3: Removing Media and Playlists
Code:
```
    PlaylistManager manager = new PlaylistManager();
    Playlist rockPlaylist = new Playlist("Rock Classics");
    manager.addPlaylist(rockPlaylist);
    MediaFile song1 = new MediaFile("Stairway to Heaven", "Led Zeppelin", 482, "mp3");
    MediaFile song2 = new MediaFile("Bohemian Rhapsody", "Queen", 354, "mp3");
    manager.addMediaToPlaylist("Rock Classics", song1);
    manager.addMediaToPlaylist("Rock Classics", song2);

    manager.removeMediaFromPlaylist("Rock Classics", "Stairway to Heaven");
    System.out.println(manager.getPlaylist("Rock Classics").toString());
    manager.removePlaylist("Pop Hits");
    try {
        manager.getPlaylist("Pop Hits");
    } catch (PlayerException e) {
        System.out.println("Error: " + e.getMessage());
    }
```

Program Output:

```
Playlist: Rock Classics
1. Bohemian Rhapsody - Queen (5:54)
Error: Playlist 'Pop Hits' not found.
```

6. Non-Functional Specifications

- **Assignments are to be done individually and must be your own work**. **Software will be used to detect cheating**.
- You must properly document your code by adding comments where appropriate. Add comments at the beginning of your classes indicating who the author of the code is and a giving a brief description of the class. Add comments to methods to explain what they do and to instance variables to explain their meaning and/or purpose. Also add comments to explain the meaning of potentially confusing parts of your code.
- Use Java coding conventions and good programming techniques:
    - Use meaningful variable and method names.
    - Use consistent conventions for naming variables, methods, constants, and classes.
- Readability. Use indentation and white spaces in a consistent manner to improve the readability of your code.

7. Submitting your Work

- You **MUST SUBMIT ALL THE JAVA FILES THAT YOU WROTE through the Gradescope link in OWL**.
- **DO NOT** put a ``package'' line at the top of your java files.
- **DO NOT** submit a compressed file (.zip, .tar, .gzip, ...); **SUBMIT ONLY** .java files.
- Do not submit your .class files. If you do this and do not submit your .java files, your assignment cannot be marked.
- Submit your assignment on time. Late submissions will receive a penalty as specified in the course outline.

Files to submit

- MediaFile.java
- Playlist.java
- PlaylistManager.java

Remember **you must do** all the work on your own. **Do not copy** or even look at the work of another student. All submitted code will be run through similarity-detection software.

8. Marking

What You Will Be Marked On:

- Functional specifications:
  - Does the program behave according to specifications?
  - Does it produce the correct output?
  - Are your classes implemented properly?
  - Are you using appropriate data structures?
- Non-functional specifications: as described above.
- The assignment has a total of 20 marks.

8.1 Marking Rubric

- Program Design and Implementation.
  - MediaFile Class Implementation (1 mark)
  - Playlist Class Implementation (3 marks)
  - PlaylistManager Class Implementation (3 marks)

- Testing. Program produces the correct output for all tests: 11 marks.
- Programming Style: 2 marks
  - Meaningful names for variables and constants: 0.5 mark
  - Code is well designed (simple to follow, no redundant code, no repeated code, no overly complicated code, …) and readable (good indentation): 0.5 mark
  - Code comments: 1 mark