P1

1️⃣ Background

Today, over 400 million people stream music online. But instead of playing songs individually, many people now listen to **music playlists** that contain one or more songs. In this assessment, you will implement a `MusicPlaylist` class that represents a music queue with a notion of history. Listeners can not only get the next song in the queue, but they can also preview all the songs in the queue, and modify the queue's history.

By completing this assignment, students will be able to:

- Implement a well-designed Java class to meet a given specification.
- Manipulate and store data using `Stack`s and `Queue`s
- Follow prescribed conventions for code quality, documentation, and readability

2️⃣ Program Behavior

This program will allow users to manage a music playlist that can queue up songs (put them in the playlist of songs to be played next) and maintain a history of all songs queued. You will use a `Queue` to maintain the playlist of songs and a `Stack` to track the user's listening history. The program will allow a user to add songs to the music playlist, play songs, print out the song history, clear the song history, and delete from the song history.

Sample Program Execution (user input bold and underlined):

Expand to see full sample program execution (user input **bold** and underlined):

Expand

The program should begin with an introductory message that says `Welcome to the CSE 122 Music Playlist!`. After this introduction is printed, there is a menu that allows users to pick options that will control their playlist.

To manage the state of the music playlist, you should use a `Queue`. This will ensure that songs that are added first will be played first (i.e, as `Queue`s are a first-in-first-out structure). To manage the history, you should use a `Stack`. This will ensure that the most recently played songs are at the top of your history (i.e, as `Stack`s are a last-in-first-out structure).

Notice that this behavior is naturally what you might expect when dealing with a playlist of songs in a platform like Spotify!

3️⃣ Main Menu

The program's menu should work properly regardless of the order or number of times its commands are chosen. For example, the user should be able to run each command many times if desired. Users will enter a short string to indicate which option they'd like to execute. When the user's choice is not recognized, the program should just prompt the user for their choice again.

Click "Expand" to see an example of this type of interaction.

Expand

When the user enters "q", your program should end without printing anything else to console output.

This requires you to read user input, which you must do using a `Scanner`. All input should be read using the `nextLine` method in the `Scanner` class, and for integer input you should then call `Integer.parseInt()` (as you did for C0: Todo List Manager).

4️⃣ Music Playlist Implementation

Adding a song

If the user enters "a" (case-insensitively) when prompted with the menu of options, your program should prompt the user for a song to add their music queue, and add that song. Once the song is added to the queue, you should let the user know by printing the statement `Successfully added {song name}`, replacing `{song name}` with the name the user entered. The sample execution below, shows an example of adding a song (user input **bolded** and underlined):

```
Welcome to the CSE 122 Music Playlist!
(A) Add song
(P) Play song
(Pr) Print history
(C) Clear history
(D) Delete from history
(Q) Quit

Enter your choice: A
Enter song name: Bad Romance
Successfully added Bad Romance
...
```

Playing a song

If the user enters "p" (case-insensitively) when prompted with the menu of options, your program should play the song that's at the front of the playlist! You should throw an `IllegalStateException` if the playlist is empty (passing no parameters).

To "play a song", you should remove the song from the playlist, print a message saying `"Playing song: {song name}"` (inserting the song's name), and finally add the song to the history.

The sample execution below, shows an example of adding a song, and then playing it (user input **bolded** and underlined):

```
Welcome to the CSE 122 Music Playlist!
(A) Add song
```

```
(P) Play song
(Pr) Print history
(C) Clear history
(D) Delete from history
(Q) Quit

Enter your choice: A
Enter song name: Just a Friend
Successfully added Just a Friend


(A) Add song
(P) Play song
(Pr) Print history
(C) Clear history
(D) Delete from history
(Q) Quit

Enter your choice: p
Playing song: Just a Friend
...
```

5️⃣ History Implementation

The history tracks all songs that have been played in the past in reverse chronological order. A `Stack` is a great choice for the history, because it will automatically store the songs in this reverse chronological order (so the most recently played songs are stored at the "top" of the `Stack`).

Notice that **songs are only added to the history, once we play that song**. That is, if you don't play any songs, then the history will be empty!

Printing history

If the user enters "Pr" (case-insensitively) when prompted with the menu of options, your program should print out the complete history of all songs played in reverse chronological order. **You are allowed to use either an auxiliary `Queue` or `Stack` to implement this behavior.**

Note that you will want to make sure you *restore the history* (or contents of the history `Stack`) in its original form if you modify the history `Stack` so that the history is maintained. You should throw an `IllegalStateException` if the history is empty.

Expand the section below to see some sample output of printing history (user input **bold** and underlined)!

Expand

So, notice that if you first play Midnight Train to Georgia and then Goodbye Earl, then selecting the "print" option produces the output

Goodbye Earl
Midnight Train to Georgia

Remember that you want to print the songs in reverse chronological order (most recent song played gets printed first). This is tricky! You need to use an **auxiliary (extra)** `Stack` **or** `Queue` within this method to help you do this.

Clearing History

If the user enters "c" (case-insensitively) when prompted with the menu of options, your program should clear the history of all songs played. **You should not create a new object while clearing the history,** but rather delete all the entries in the current history.

Deleting from history

Users can also delete songs from history. They can choose to either delete the most recently played songs, or the songs that were played earliest in the history.

When the users enter how many songs they would like to delete from history (`n`), a *positive* number indicates that you should delete the `n` *most recently-played songs* from the history. If the user enters a *negative* number `n`, you should delete the *chronologically-first* `-n` *songs that were played.*

Note that if `n` is negative, then `-n` is a *positive* number.

Notice that if the user enters `0`, they are indicating that they don't want to delete any songs so your program doesn't have to do anything!

For example, suppose the current history is this (as if the user selected option 3; remember they are listed in reverse chronological order):

`Livin' on a Prayer`

`Summertime Sadness`

`Africa`

`Sugar, We're Goin' Down`

`Call Me Maybe`

If the user enters `3` when prompted for how many songs to delete, your program should delete the 3 *most recently-played songs* from history (so it would delete `Livin' on a Prayer`, `Summertime Sadness`, and `Africa`).

If the user enters `-3` when prompted for how many songs to delete, your program should delete the *chronologically-first 3 songs that were played* (so it would delete `Call Me Maybe`, `Sugar, We're Goin' Down`, and `Africa`).

If the user enters "d" when prompted with the menu of options, your program should first print out a message explaining how they should indicate whether they want to delete from most-recent history or from the beginning of history:

A positive number will delete from recent history.
A negative number will delete from the beginning of history.

Your program should then print out a prompt the user for an input `n` and delete `Math.abs(n)` entries from history (either from the most-recent history or from the beginning of history, depending on whether `n` is positive or negative).

You should throw an `IllegalArgumentException` if the size of the history is less than `Math.abs(n)`.

`Math.abs()` is a method that calculates and returns the *absolute value* of the given number. So if `n` is non-negative, `Math.abs(n)` will return `n`. If `n` is negative, `Math.abs(n)` will return `-n`.

You are allowed to use either an auxiliary `Queue` or `Stack` to implement this behavior.

Expand the section below to see a sample output of this command (user output **bolded** and underlined).

Expand

6️⃣ Development Strategy

The best way to write code, especially for a big project like this one, is in *stages*. If you attempt to write everything at once, it will be significantly more difficult to debug, because it's harder to narrow down what part of your code introduced the bug.

If you write a small chunk of code, then test (and potentially debug and fix) your code until you feel confident in its correctness before moving on to the next part of code, any bugs you encounter in the future are very likely caused by the code you wrote most recently! The technical term for this development style is "iterative enhancement" or "stepwise refinement."

It is also important to be able to test the correctness of your solution at each different stage.

We suggest that you work on your assessment in four stages:

- **Part 1** - First, work on writing the menu loop in `main` and creating the data structures that you use to maintain the music playlist and history.
- **Part 2** - Next, work on adding a song and playing that song. You should only move on to the behavior of playing a song once you're confident that you're able to add a song to the music playlist successfully.
- **Part 3** - Next, work on printing the history and then clearing the history. Before starting these tasks, make sure you understand how the history is stored and updated.
    - Make sure not to *destroy* the current history when printing the history! (i.e., the contents of the history `Stack` should be the same before and after printing the history out.)
- **Part 4** - Finally, work on deleting from the history.

7️⃣ Development Notes

Capturing Structure

Your `main` method in this program may have more code than it has in previous assessments. In particular, you may include a limited amount of output and some control flow constructs (e.g. a loop to drive the menu) in main. However, your main method must remain a concise summary of your program's structure, and you must still utilize methods to both capture structure and eliminate redundancy.

The `MovieFavorites.java` example from class is a good example to reference!

User Input with `Scanner`

- You should read ***all*** user input from your console Scanner using line-based method `.nextLine()` . Using both token-based and line-based processing with the same Scanner object can lead to odd behavior and bugs so you should avoid calling anything but `.nextLine()` for this assignment.
- Since you should read all user input using `.nextLine()`, you cannot read integer input with `.nextInt()`. When prompting for integers, read user input as a `String` with `.nextLine()` and convert it to an `int` in the following way:

  ```
  String input = console.nextLine();
  int num = Integer.parseInt(input);
  ```

8️⃣ Assignment Requirements

For this assignment, you should follow the Code Quality Guide when writing your code to ensure it is readable and maintainable. In particular, you should focus on the following requirements:

- For this part of the assignment, you should only be using a single `Stack` to manage the history and a single `Queue` to manage the music queue.
- You should use interface types where appropriate and properly use type parameters when specifying the types of `Stack`s and `Queue`s.
- You should comment your code following Commenting Guide. You should write comments with basic info (a header comment at the top of your file), a class comment, and a comment for every method other than main.
  - In particular, some of the methods you will write will need to **throw Exceptions**. Your method comments should describe the Exception it may throw and under what circumstances, in addition to the normal required details (behavior, returns, and parameters).
- If you investigate `Stack`s and `Queue`s on your own, you may discover that other methods exist. However, we are restricting you to only the methods that have been introduced in class! You should only use these methods in your solution.
- As a general rule, you have learned everything necessary to implement a solution to an assignment by the time it is released. If you are tempted to use any concepts or tools that have *not* been covered in class, check the **Forbidden Features** section of the Code Quality Guide first to make sure they do not appear there!