CSE 8B Spring 2022

Assignment 4

Objects and classes, and object-oriented thinking

Due Date: Wednesday, April 27, 11:59 PM

Welcome back! Be sure to start this assignment as EARLY as possible! You got this!

Learning goals:

- Implement POJOS (Plain Old Java Object) classes with getters and setters.
- Implement classes that interact with each other.
- Write unit tests using objects and instance methods.

NOTE: This programming assignment must be done individually. Paired programming is NOT allowed for this assignment.

Coding Style (10 points)

For this programming assignment, we will be enforcing the CSE 8B Coding Style Guidelines. These guidelines can also be found on Canvas. Please ensure to have COMPLETE file headers, class headers, and method headers, to use descriptive variable names and proper indentation, and to avoid using magic numbers.

Part 0: Getting started with the starter code (0 points)

1. Make sure there is no problem with your Java coding environment. If there is any, then review Assignment 1, or come to the office/lab hours before you start Assignment 4.
2. Once again, in this PA, you will start with some code that we have written for you.
3. If you are using UCSD Linux Cloud (recommended) for the assignment, download the starter code from Piazza → Resources → Homework by opening a web browser on the Remote Desktop. You will be downloading the following files:
    - Assignment4.java
    - Song.java

- Playlist.java
- MusicPlayer.java

You can open Firefox by going to Applications → Firefox.

4. If you are working on your local machine, then you can download the starter code from a local web browser on your machine.

Part 1: Implement MusicPlayer application (60 points)

Overview

In this assignment, you will implement a very minimalistic simulator version of a music player application like Spotify or Apple Music (obviously with no user interface, audio, or video). With the music player, you can create a single playlist that can store up to a limited number of songs and play songs in the playlist in their order or in shuffle mode.

IMPLEMENTATION TIP: you must NOT change any data field or method signature in the starter code. As such, please observe the starter code and read the instructions below to make sure you understand what each field means before you start to implement.

What's a MusicPlayer without Song.java? (10 points)

In `Song.java`, you will be implementing **1 constructor, 4 instance getter methods, and 2 instance methods**. This class is a POJO (Plain Old Java Object) class without any great functionality of its own but acts as a template to model a 'Song' in a music player.

The Song object should contain the following fields (all are provided in the starter code ; do not change any of these):

1. `private String name`: the name of the song
2. `private String artistName`: the name of the song's artist
3. `private int length`: the length of the song in minutes
4. `private boolean isPlaying`: a field indicating whether the song is currently playing or stopped in the music player

Notice how each member field is declared `private`. This means that the member is only visible *within* the class, not from any other class. In other words, you will need to use accessors (i.e., getter methods) and mutators (i.e., setter methods) to access and modify, respectively, these **`private`** members. **You must also use `this` keyword to modify and access member variables hidden by local variables.**

The Song class should contain the following member methods:

Note: (Please implement and complete the method body but do not change the method headers)

1. `public Song(String name, String artistName, int length)`:
   This is the constructor of the `Song` class. The constructor needs to set the `name`, `artistName`, and the `length` member variables using the constructor parameters. Finally, the constructor should set the `isPlaying` member variable of the newly created song to **`false`**. You must ensure that you use the **`this`** keyword to set the member variables.

2. Four getters/accessors - `public String getName(), public String getArtistName(),public int getLength()` and `public boolean getIsPlaying()`:
   Each getter method should simply return the corresponding `private` field of this `Song` object.

3. Two methods - `public void playSong()` and `public void stopSong()`:
   These two methods are used to modify the value of the `isPlaying` member variable. `playSong()` should set the `isPlaying` attribute to **`true`** whereas `stopSong()` should set the `isPlaying` attribute to **`false`**.

Let's create a Playlist.java! (40 points)

Here, we implement methods inside `Playlist.java` to add songs, delete songs, clear the entire playlist, and get the longest song from the playlist.
The `Playlist` object should contain the following fields (all are provided in the starter code ; do not change any of these):

1. `private String name`: the name of the playlist
2. `private Song[] songList`: an array of `Song` objects to hold songs in the playlist
3. `private int songCount`: the number of songs currently added to the playlist

Notice how each member field is declared `private`. This means that the member is only visible *within* the class, not from any other class. In other words, you will need to use accessors (i.e., getter methods) and mutators (i.e., setter methods) to access and modify, respectively, these **`private`** members. **You must also use the `this` keyword to modify and access member variables hidden by local variables.**

Note: There are two constants **SONG_COUNT** and **SONGLIST_FULL** defined in the beginning of the class. We will describe below how to use them.

The `Playlist` class should contain the following member methods:

Note: (Please implement and complete the method body but do not change the method headers)

1. `public Playlist(String name)`:
   This is the constructor of the `Playlist` class. First, the constructor needs to set the `name` member variable using the constructor parameter. Next, you need to initialize the `songList` array and set the size of the array using the **SONG_COUNT** constant provided at the beginning of the class. Lastly, set the `songCount` member variable to 0 indicating that the playlist is new without any songs added yet.

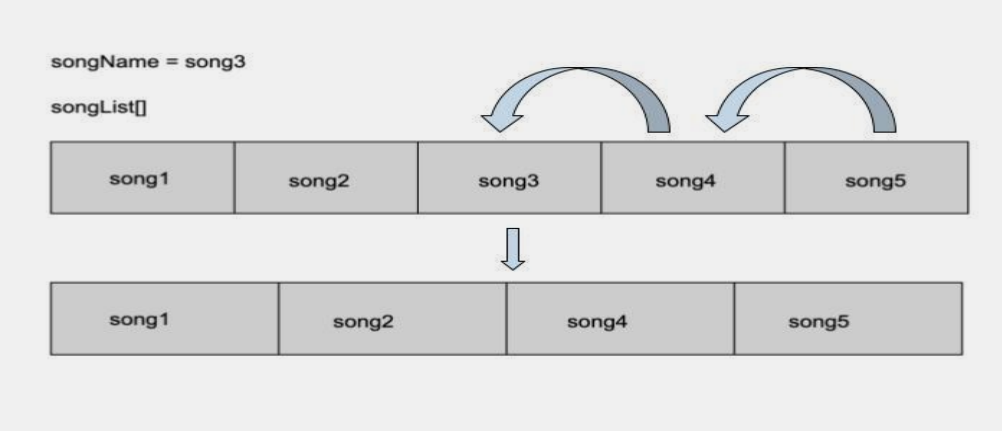
2. Three getters/accessors - `public String getName()`, `public Song[] getSongList()`, and `public int getSongCount()`:
   Each getter method should simply return the corresponding `private` field of this `Playlist` object.

3. One setter/mutator - `public void setName(String name)`:
   The setter method should set the `name` member variable to the name provided in the method argument. **You must also use the `this` keyword to modify the member variable hidden by the local variable.**

4. `public void addSong(Song newSong):`
   This method implements the functionality to add a song `newSong` to the playlist. Use the `songCount` member variable to track the position in the `songList` array where `newSong` would be added. Before adding `newSong`, check if the playlist is full by comparing `songCount` to **SONG_COUNT** constant. If so, print to the console that the `songList` is full using the **SONGLIST_FULL** constant**.** Use `System.out.println()` for this. Make sure to update `songCount` after adding newSong to the array**.**

5. `public void deleteSong(String songName):`
   This method implements the functionality to delete a song corresponding to the `songName` from the playlist. To do this, first identify the index where the song represented by the `songName` is present using an array search in `songList`. If no such song is in the playlist, you do not need to do anything further and just return from the method.

Once the index containing `songName` is found, overwrite the song with the next song in the `songList` if any and continue shifting all the songs that are present in the subsequent positions by one index to the left. The following diagram shows how to perform the delete operation in the `songList` array.



Remember to update the `songCount` member variable once the delete operation is done.

6. `public Song getLongestSong():`
   This method implements the functionality to retrieve the longest song from the playlist. Use the `length` member variable of each song in the `songList` to identify the longest song in the playlist. Return the song corresponding to the maximum song length from the playlist.

7. `public void clear():`
   This method implements the functionality to clear the entire playlist. When this method is invoked, the entire playlist represented by `songList` is cleared. The clear operation is performed by reinitializing the `songList` array to an empty array of songs whose size is equal to **SONG_COUNT**. Remember to reinitialize `songCount` to 0 since the playlist is now cleared.

Finally, we'll play some songs on MusicPlayer.java! (10 points)

Next, we implement methods inside `MusicPlayer.java` to play songs in a playlist in two ways - in order and in shuffle mode. For simplicity, we assume that the music player has only one playlist. We have initialized the playlist for you already in the starter code using the `MusicPlayer()` constructor. For reference, `MusicPlayer()` creates a new `Playlist` object

with name “My Playlist” and adds 5 different songs to it. Please do not modify any of the code inside `MusicPlayer()` constructor or you’re in for some trouble!

The `MusicPlayer` class must contain the following member methods:

Note: (Please implement and complete the method body but do not change the method headers)

1. `public void playSongsInOrder():`
   This method implements the functionality to play the songs in the `playlist` in their order. To do this, iterate over all the songs one by one, keeping track of the last played song (song in the previous index), stopping it and playing the song in the current index. Remember to stop the last played song after all the iterations to ensure no songs are playing on the music player at the end of all the playing.

   Thus, perform the following two steps in each iteration:
   a) To stop the last played song, use the `stopSong()` member method on the `playlist` object. Also, notify the user that the song is stopped by printing to the console *"Stopping song: <insert song name of last played song>"* (use the System.out.println() method with the **`STOPPING`** constant here). **Note that for the first iteration, there would be no last played song and you can skip this step.**

   b) To play the song in the current index, use the `playSong()` member method on the `playlist` object. Again, notify the user that the song is playing by printing to the console *"Playing song: <insert song name of song in current index>"* (use the System.out.println() method with the **`PLAYING`** constant here).

2. `public void playSongsInShuffle():`
   This method implements the functionality to play the songs in the `playlist` in a shuffle (randomized) mode. Before implementing this method, let’s first implement a helper function `randomWithRange()` that has already been defined in the starter code.

   `private int randomWithRange():`
   In this method, use the Math.random() library method to generate a random number in the range **[0, playlist.getSongCount())**, i.e., 0 *inclusive* and playlist.getSongCount() *exclusive*. Remember that **Math.random()** generates a random number (double) greater than or equal to 0.0 and less than 1.0. You should use this to generate a number in the required range. Typecast the double value to an integer and return the generated

random number as an int from the method. Note that this method is marked `private`. This means it cannot be accessed from any other class and its scope is limited to the current class.

Now, let's continue with our implementation of `playSongsInShuffle()`.In each iteration, we stop the last played song and play a randomly chosen song.

To do this, iterate over `songCount` number of iterations keeping track of the last played song. In each iteration, first, stop the last played song and notify the same to the user. Then, generate a random number `randomNum` in the range from 0 to `songCount` using the helper method `randomWithRange()` that we just implemented. Play the song present in the playlist at `randomNum` and notify the user of the same. Remember to stop the last played song after all the iterations to ensure no songs are playing on the music player at the end of all the playing.

a) To stop the last played song, use the `stopSong()` member method. Also, notify the user that the song is stopped by printing to the console *"Stopping song: <insert song name of last played song>"* (use the System.out.println() method with the **`STOPPING`** constant here). **Note that for the first iteration, there would be no last played song and you can skip this step.**

b) To play the song in `randomNum` index, use the `playSong()` member method. Again, notify the user that the song is playing by printing to the console *"Playing song: <insert song name of song in randomNum>"* (use the System.out.println() method with the **`PLAYING`** constant here).

Look up the member methods of the `playlist` object to get the `songList` and `songCount` of the `playlist` wherever necessary. Similarly, take a look at the member methods of the `Song` class wherever required.

3. One getter/accessor - `public Playlist getPlaylist()`:
The getter method should simply return the `private playlist` field of this `Playlist` object.

Part 2: Compile, Run and UnitTest Your Code (10 points)

Just like in previous assignments, **in this part of the assignment, you need to implement your own test cases in the method called `unitTests` in the `Assignment4.java` class. We will be writing unit tests to test only `Playlist.java`.**

In the starter code, a test case is already implemented for you. You can regard it as an example to implement other cases. Recall, the general approach is to come up with different inputs and manually give the expected output, then call the method with that input and compare the result with expected output.

You are encouraged to create as many test cases as you think to be necessary to cover all the edge cases. The `unitTests` method should `return true` only when all the test cases are passed. Otherwise, it should `return false`. **To get full credit for this section, you should create at least five test cases that cover different situations (including the one we have provided) for four methods - addSong(), deleteSong(), getLongestSong() and clear().** In other words, you will need to create at least **four** more tests that test `addSong`, `deleteSong`, `getLongestSong` and `clear`, with at least **one test for each.**

To compare arrays by the equality of contents, you must use **Arrays.equals()**. See the given unit tests for examples.

If a test is not passing, try temporarily printing the result of your method(s) and comparing them to the expected output.

You can compile all the files present in the starter code and run your unit tests from main() using the following commands: (Make sure you are in the correct directory, else navigate to the starter code using `cd`)

```
> javac *.java
> java Assignment4
```

The first command `javac *.java` compiles all the files in the folder with a .java extension, which is what is required. After the .java files compile and .class files are generated, we run the main() in Assignment4 using `java Assignment4`. The below screenshot shows the same:

```
(base) ➜  PA4 ls
Assignment4.java MusicPlayer.java Playlist.java    Song.java
(base) ➜  PA4 javac *.java
(base) ➜  PA4 java Assignment4
```

Part 3: Implement main() method (20 points)

In Part 3, you will be completing `Assignment4.java`'s main method to create an instance of `MusicPlayer` class and run some methods with the help of user's input.

Once your unit tests pass and you are confident that the above methods are working, it's time for you to put everything together so that we can play songs from our music player using the command line!

Similarly to previous assignments, all of this command line functionality will be implemented in the `main` method. We have defined several `String` constants for you (at the top of `Assignment4.java`) to use for printing. Assume that every `String` constant needs to be printed with the cursor thrown to the next line after (Think about print vs println).

MusicPlayer - SETUP

First, complete the method - `public static MusicPlayer setUpMusicPlayer():`
In this method, create an object of `MusicPlayer` and return the same. This will instantiate a playlist with songs added to it since the constructor of `MusicPlayer` does this for us. Check `MusicPlayer()` in `MusicPlayer.java` for more details about the songs added. Note that this is a `static` method unlike all the other methods we have implemented till now. This is because we would be calling this from our `main()` in a `static` context. For more details about the usage of static, revisit Lecture 7 slides!

MusicPlayer IN ACTION!

The following steps outline the logic of the code to be written in `main`.
Hint: You will be using all of the instance methods you wrote, and you will be implementing only a one-time interaction (no loops).

1. Set up the music player using the `setUpMusicPlayer()` method we just implemented.

2. Ask the user for what playlist order he chooses to play songs in using the prompt provided. Scan the user's choice as an integer. Make sure to use the constants provided at the top of the class.
3. Play songs in either in order or shuffle mode based on the user's choice. If the user's choice is invalid, print out an invalid message to the console.
4. We are done playing songs! Let's clear our playlist now. Go ahead and clear the playlist of your music player.

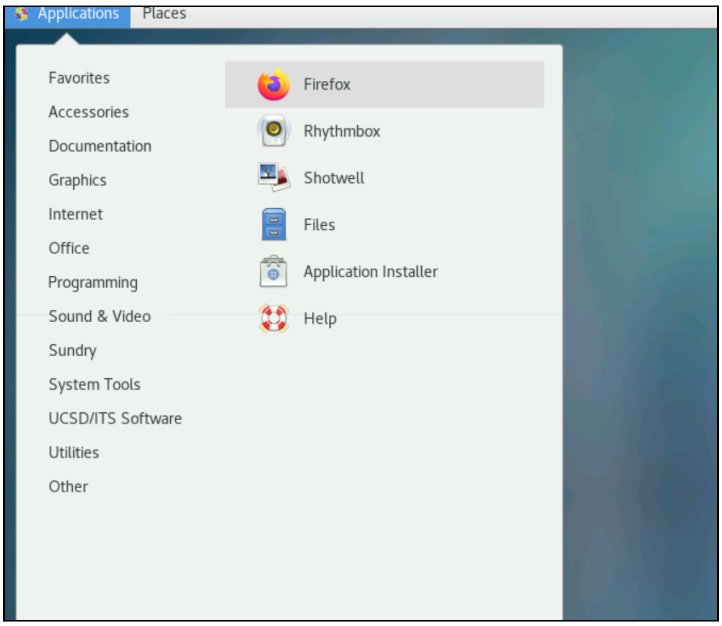
Optional (but very helpful) suggestions:

- View the **4/19** Discussion section to see the command line `MusicPlayer` in action. Your output will be compared to the reference solution being run by the TA. Line spacing and other formatting of the output should be **exact** for full points in this section.
- Once you are done with writing the code for this section, start playing your music player in the command line to test out whether everything works as expected.

Submission

You're almost there! Please follow the instructions below carefully and use the **exact submission format**. Because we will use scripts to grade, **you may receive a zero** if you do not follow the same submission format.

1. Open a web browser on the Remote Desktop. You can open Firefox by going to Applications → Firefox.



2. Open Gradescope in Firefox and login. Then, select this course → PA4.
3. Click the DRAG & DROP section and directly select the required files **Song.java, Playlist.java, MusicPlayer.java** and **Assignment4.java**. Drag & drop is fine. Please make sure you don't submit a zip, just the separate files in one Gradescope submission. Make sure the names of the files are correct.
4. You can resubmit unlimited times before the due date. Your score will depend on your final (most recent) submission, even if your former submissions have higher scores.
5. Your submission should look like the below screenshot. If you have any questions, feel free to post on Piazza!