Homework 3

Authors: Vince, Will, Melanie, Aqdas, Nick, Andrew, and Jack

Purpose

We at 1331 Incorporated are starting an ambitious new project: a music subscription service. Due to our poor taste in music and general lack of research in the market, the coding portion has been left to you (we're more of the *idea* people). To complete this assignment, you will apply your knowledge of Polymorphism, Interfaces, and Arrays.

Solution Description

For this assignment, you will be simulating playlists, the songs they contain, and the users that create them. You will also have to include JavaDocs and adhere to proper Checkstyle.

`Playable.java`

This file contains the interface for any object that contains audio data that can be played back to a listener (e.g. song, podcast, audiobook, etc.). `Playable` must be an interface.

- Methods
  - A `play` method that takes no parameters and returns nothing.

`HipHopSong.java`

This file represents a hip hop song and must implement the `Playable` interface

- Variables (all of the fields below should follow the rules of encapsulation)
  - `name` – a String that holds the name of the hip hop song
  - `artist` – a String that holds the name of artist of the hip hop song
  - `producer` – a String that holds the name of the hip hop song's producer
- Constructors
  - A constructor that takes values in the following order- `name, artist, producer.` The constructor should initialize the fields of the object with the corresponding values passed in.
- Methods
  - A `play` method that prints "`[producer] on the track! Twitter fingers turn to coding fingers`" followed by a newline character
  - An override of Object's `toString` method that returns a `String` representation of a `Song` object:

    "`[name] by [artist] – woo!`"

  - Note: Usually good class design would dictate also overriding the `equals` and `hashCode` methods. However, we are omitting them for the sake of brevity.

`ClassicalSong.java`

This file represents a classical song and must implement the interface `Playable`

- Variables (all the fields below should follow the rules of encapsulation)
  - `name` – a String that holds the name of the song
  - `artist` – a String that holds the name of the composer of the classical song

  - `length` – an int that holds how long the song is, in seconds
- Constructors
  - A constructor that takes values in the following order- `name, artist, length.` The constructor should initialize the fields of the object with the corresponding values passed in.
- Methods
  - A `play` method that prints "`weewoo` " `length` number of times followed by a newline character after the last "`weewoo` "
    - If length was 5, this would print "`weewoo weewoo weewoo weewoo weewoo` "
  - An override of Object's `toString` method that returns a `String` representation of a `Song` object:

    ```
    "[name] by [artist] pub"
    ```

  - Note: Usually good class design would dictate also overriding the `equals` and `hashCode` methods. However, we are omitting them for the sake of brevity.

User.java

This file represents a user of our music service. `User` must be an abstract class and will be used to as the foundation to create concrete user types (e.g. free users, paid users, premium users, etc.)

- Variables (all the fields below should follow the rules of encapsulation)
  - `name` - a String that holds the name of the user
- Constructors
  - A constructor that takes in `name` and initializes the corresponding field in the object
- Methods
  - A `giveRecommendation` abstract method. It should take in a `PlayList` object and not return anything
  - An override of Object's `equals` method dependent on the `name`
  - An override of Object's `hashCode` method dependent on the `name.` This method should return the hash code of that field.
  - An override of Object's `toString` method that returns the `name` of the user

FreeUser.java

This file represents a free user of our music service. It should inherit from the `User` abstract class.

- Constructors
  - A constructor that takes in `name` and initializes the corresponding field in the object
- Methods
  - A `giveRecommendation` method. It should take in a `PlayList` object and print out "I recommend the `[toString of the first song in the playlist]".`
    - If the `PlayList` is empty, print "`The playlist is empty`" instead.

PlayList.java

This file represents a playlist of songs

- Variables (all the fields below should follow the rules of encapsulation)
  - `songs` – an `Array` that holds all the `Playable` objects on the playlist
  - `name` – a String that holds the name of the playlist

    - `currentlyPlaying` – an int holding the index of the element of the current song playing, -1 if no song is playing
    - `owner` – the `User` that owns the playlist
- Constructors
    - A constructor that takes values in the following order: `songs`, `name`, `owner`. `currentlyPlaying` should be initialized to -1
- Methods
    - An override of Object's `toString` method that returns "`[name] owned by [toString of owner], currently playing [toString of Playable]`"
        - If `currentlyPlaying` is –1, return "`[name] owned by [toString of owner], currently playing nothing`"
    - An override of Object's `equals` method dependent on the `name` and the `owner`
    - An override of Object's `hashcode` method dependent on the `name` and the `owner`. This method should return the sum of the hash codes for each of said fields.
    - A `startPlaying` method that iterates through `songs` and calls the `play` method of each `Playable.` Start iterating from index `currentlyPlaying`, including the `Playable` at `currentlyPlaying`, and stop at the end of the `songs`. `currentlyPlaying` should be set to –1 at the end of this process. If `currentlyPlaying` is not within the bounds of the `songs` array (i.e. it is negative or greater than the last index of the array), start at index 0.

For all classes, create accessor and mutator methods as you see fit.

Import Restrictions:

You may not import anything.

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our auto grader. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`
- Creating your own, named packages

Collaboration

- *Collaboration Statement*
- To ensure that you acknowledge collaboration and give credit where credit is due, **we require that you place a collaboration statement as a comment at the top of at least one java file that you submit**. That collaboration statement should say either:

  I worked on the homework assignment alone, using only course materials.

  – or –

  In order to help learn course concepts, I worked on the homework with [give the names of the people you worked with], discussed homework topics and issues with [provide names of people], and/or consulted related material that can be found at [cite any other materials not provided as course materials for CS 1331 that assisted your learning].
- Recall that comments are special lines in Java that begin with `//`.

Turn-In Procedure

Submission

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- `PlayList.java`
- `Playable.java`
- `HipHopSong.java`
- `ClassicalSong.java`
- `User.java`
- `FreeUser.java`

Make sure you see the message stating "HW03 submitted successfully". From this point, Gradescope will run a basic autograder on your submission as discussed in the next section.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the homework. We will only grade your last submission: be sure to submit every file each time you resubmit.

Gradescope Autograder

For each submission, you will be able to see the results of a few basic test cases on your code. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

The Gradescope tests serve two main purposes:

- Prevent upload mistakes (e.g. forgetting Checkstyle, non-compiling code)
- Provide basic formatting and usage validation

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or Checkstyle your code; you can do that locally on your machine.

Other portions of your assignment can also be graded by a TA once the submission deadline has passed, so the output on Gradescope may not necessarily reflect your grade for the assignment.

Checkstyle and Javadocs

You must run Checkstyle on your submission. The Checkstyle cap for this assignment is **25 points**. If you don't have Checkstyle yet, download it from Canvas. Place it in the same folder as the files you want Checkstyle. Run Checkstyle on your code like so:

```
$ java -jar checkstyle-8.28.jar yourFileName.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be points we would take off (limited to the amount we specified above). The Java source files we provide contain no Checkstyle errors. In future homeworks, we will be increasing this cap, so get into the habit of fixing these style errors early!

Along with this, you will also JavaDoc your code.

Run the following to only check your JavaDocs.

```
$ java -jar checkstyle-8.28.jar -j yourFileName.java
```

Run the following to check both JavaDocs and Checkstyle.

```
$ java -jar checkstyle-8.28.jar -a yourFileName.java
```

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit `.class` files.
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- Check on Piazza for all official clarifications