Homework 9 : Movie Trivia

(Deadline as per Canvas)

For HW9, you may work as a group (no more than 2 students). **Please mention your collaborator's name at the top of each of your code files.**

This homework deals with the following topics:

- More experience with class-based object-oriented programming
- Arrays and ArrayLists
- Test-Driven Development (unit testing)
- Static methods (we'll learn about these in the next lecture)

The Assignment

In this HW, we will deal with representing a movie database using classes and arraylists, with the goal of answering some simple movie trivia questions. For example, "what is the name of the movie that both Tom Hanks and Leonardo DiCaprio acted in?". Or, "what are the movie ratings for Rocky I?"

To represent the movie database, we will use 2 ArrayLists: The first corresponds to information about an actor and all the movies that he/she has acted in. The second corresponds to information about the critics score and the audience score from https://www.rottentomatoes.com/, about different movies.

Given that information, we will then want to answer some typical movie trivia questions.

Data Structures

There are two ArrayLists in this HW: The first ArrayList stores Actors (instances of the Actor class) with the name of the actor and a list of movies the actor has acted in. The second ArrayList stores Movies (instances of the Movie class) with the name of the movie, the critics rating, and the audience rating.

The two ArrayLists are populated with data from two different data files. The ArrayList with the information about actors is loaded from the "moviedata.txt" file. Each row in the file contains an actor's name followed by movies the actor has acted in. For example:

```
Meryl Streep, Doubt, Sophie's Choice, The Post
```

The ArrayList with information about movies is loaded from the “movieratings.csv” file. Each row in the file contains a movie followed by the critics rating and the audience rating. For example:

```
Doubt,79,78
```

The code for loading the data files is provided for you in the MovieDB.java file, but here are some items to note:

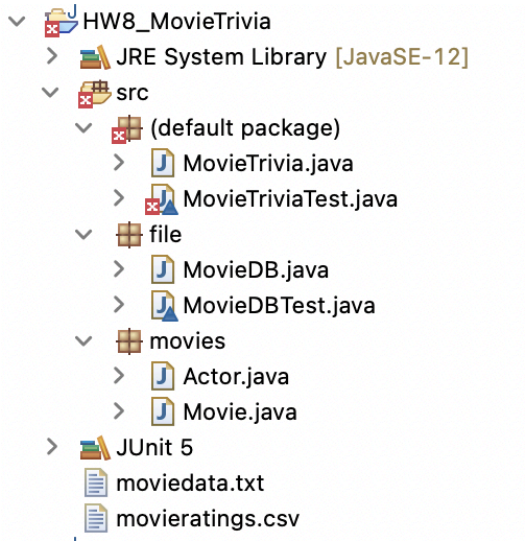
- When loading the data files, the code converts all characters to lowercase and trims the whitespace before and after each column of data before storing the information in an ArrayList. For example:
  - “Meryl Streep” or “ meryl streep ” is converted to “meryl streep”
  - “Sophie’s Choice” or “ sophie’s choice ” is converted to “sophie’s choice”
- The actor names in “moviedata.txt” are unique and the movie names in “movieratings.csv” are unique.
- Actors may have no movies. For example, in “moviedata.txt”, there are no movies listed for “Brandon Krakowsky”. He may not have been casted in a movie (yet!), but we must allow him to keep his dream, right? In this case, the code converts this line into an instance of the Actor class, with a name and an empty list of movies.
- The critics and audience ratings in “movieratings.csv” are between 0 and 100 and always exist.

Starter Code

We have provided you with four main Java class files for the program: MovieDB.java, Actor.java, Movie.java, and MovieTrivia.java. The MovieDB class reads from the two text files, “moviedata.txt” and “movieratings.csv”, and populates two ArrayLists with the data using the Actor class and the Movie class. The MovieTrivia class controls the overall program and will provide methods for asking different kinds of movie trivia questions, and insert new actors and movie ratings into the movie database. We’ve also provided you with MovieDBTest.java to run some very basic tests against MovieDB.java and MovieTriviaTest.java to run some tests against MovieTrivia.java.

Download all of the provided files and put the Java files in the “src” folder of a new Java project, keeping the exact same file and package structure. If there are no packages in your Java project, please create them using “New Package” in Eclipse. Keep the .txt and .csv files

outside of the “src” folder in the Java project. You might also have to add the Junit 5 library to your build path. The final structure of the project in the package explorer should look like this:



Once the project is set up, run MovieDBTest.java to make sure the data files can be loaded and the initial tests pass. If there are any issues with the testing file, please let us know ASAP. You should also run MovieTrivia.java to make sure the main program runs. After running MovieTrivia.java, you should see the actors and movies information printed to the console (see below).

Line 1:

```
[Name: meryl streep Acted in: [doubt, sophie's choice, the post],
Name: tom hanks Acted in: [the post, catch me if you can, cast away],
Name: amy adams Acted in: [doubt, leap year, man of steel, arrival],
Name: brandon krakowsky Acted in: [], Name: robin williams Acted in:
[popeye], Name: brad pitt Acted in: [seven, fight club]]
```

Line 2:

```
[Name: doubt Critic Rating: 79 Audience Rating: 78, Name: arrival
Critic Rating: 94 Audience Rating: 82, Name: jaws Critic Rating: 97
Audience Rating: 90, Name: rocky ii Critic Rating: 91 Audience
Rating: 95, Name: seven Critic Rating: 29 Audience Rating: 29, Name:
popeye Critic Rating: 0 Audience Rating: 0, Name: et Critic Rating:
85 Audience Rating: 86]
```

Suggested Approach

Try doing this homework using the test-driven development (TDD) approach. In particular, once you have run the starter code in MovieTrivia.java successfully, do not worry about writing any additional code in the main method. Instead, implement the code (within the required methods) in MovieTrivia.java so that it passes the test cases in MovieTriviaTest.java. As you write additional test cases in MovieTriviaTest.java, you'll implement additional code in MovieTrivia.java so that it passes the test cases you've created. Once you've passed all test cases, then make sure your method has good style. If not, refactor as needed, then rerun your test cases.

After you finish this process for the first method, do the same for the second method, then the third, etc.

Required Methods:

The "Utility Methods" and "More Fun Methods" described below must be implemented within the MovieTrivia.java class.

Utility Methods

The first step is to create some basic utility methods for interacting with the database. You are allowed (and encouraged) to modify the "moviedata.txt" and "movieratings.csv" files. Feel free to add more movies, actors, ratings, etc. But if you prefer to leave these two input files unaltered, you are free to do that as well. If you do decide to edit "moviedata.txt" or "movieratings.csv", please note that you may also need to modify some test cases in MovieTriviaTest, since the expected values in some of the test cases may no longer apply.

As you'll see, the ArrayList variables are passed to the individual methods as parameters. It is possible that an ArrayList other than the ones created by the MovieDB class will be passed to your methods. In fact, you can be guaranteed that the unit tests we write for checking your methods will use different data. Our unit tests will be in accordance with the formatting specified, but will have different values.

Here is the list of 5 utility methods that we want you to write. Be sure to add javadocs to your methods and comments to your code.

public void insertActor (String actor, String [] movies, ArrayList <Actor> actorsInfo)

- Inserts given *actor* and his/her *movies* into database
    - The *actorsInfo* is the ArrayList that is to be inserted into/updated
    - *actor* is the actor name as a string
    - *movies* is a String array of movie names that the actor has acted in
- Note that while this method is called "insertActor", it should actually do an insert **or** an update
    - If the given *actor* is not already present in the given *actorsInfo* arraylist, this method should append the given *actor* along with their given *movies to the end of actorsInfo*.
    - If the given *actor* is already present in the given *actorsInfo* arraylist, it should append the given *movies* to the end of the *actor* object's *movieCasted* list.
    - This method does not need to insert or update movies in the *moviesInfo* arraylist.
- This method does not return anything. It will work by just modifying the given *actorsInfo* ArrayList passed to it
- Consider different kinds of input when writing your test cases in MovieTriviaTest.java for this method. For example:
    - When inserting a new actor add the new instance of the Actor class to the end of the ArrayList actorsInfo. When the actor already exists, update the actor's movies in the Actor class by adding the movies in the movies Array to the end of their list. When a movie in the movies Array already exists in the actor's list of movies, avoid adding a duplicated one. Otherwise, add this movie to the actor's list.
    - Actor names should not be case sensitive, but spelling must be correct. For example, "Tom Hanks" and "ToM hanKS" should be considered the same actor in the database, so when you try to insert an actor with the same name, but different case, it should recognize it as already existing in the database. The same logic applies to every movie in the movies Array.
    - Try inserting an actor with leading or trailing whitespace (e.g. "Bradley Cooper ") -- the whitespace should not make a difference. For example, "Bradley Cooper " and "Bradley Cooper" should be considered the same actor in the database. The same logic applies to every movie in the movies Array.
    - Keep the format in actorsInfo, for example, when actor is "Brad PITT" and movies are [" Inglourious Basterds ", "ONCE Upon A Time in Hollywood"], when

added to the database, they should be stored in the format of “brad pitt” and [“inglourious basterds”, “once upon a time in hollywood”]. Please refer to the description in the **Data Structures** section for detailed formatting information.

public void insertRating (String movie, int [] ratings, ArrayList <Movie> moviesInfo)

- Inserts given *ratings* for given *movie* into database
  - The *moviesInfo* is the ArrayList that is to be inserted into/updated
  - *movie* is the movie name as a string
  - *ratings* is an int array with 2 elements: the critics rating at index 0 and the audience rating at index 1
- This method should update the ratings for a movie if the *movie* is already in the database
- Consider different kinds of user input when writing your test cases. For example:
  - Create an instance of the Movie class for a new movie and append to the end of moviesInfo or update the critics and audience scores for an existing movie in moviesInfo
  - The movie should not be case sensitive, but spelling must be correct. For example, “Rocky ii” and “Rocky II” should be considered the same movie in the database.
  - A movie with leading or trailing whitespace (e.g. “ Arrival “)
  - An incorrect number in ratings or missing ratings. If ratings == null or ratings.length != 2, do nothing and return. If the critics rating or audience rating is less than 0 or greater than 100, do nothing and return. Neither of the ratings gets updated.

public ArrayList <String> selectWhereActorIs (String actor, ArrayList <Actor> actorsInfo):

- Given an *actor*, returns the list of all movies
  - *actor* is the name of an actor as a String
  - *actorsInfo* is the ArrayList to get the data from
- Given a non-existent actor, this method should return an empty list
- Consider different kinds of user input when writing your test cases. For example:
  - Non-existent actors. When the *actor* is non-existent, return an empty list.

- Actor names should not be case sensitive, but spelling must be correct. For example, "Tom Hanks" and "ToM hanKS" should be considered the same actor in the database.
- Actor names with leading or trailing whitespace. For example, "Bradley Cooper " and "Bradley Cooper" should be considered the same actor in the database.

public ArrayList <String> selectWhereMovieIs (String movie, ArrayList <Actor> actorsInfo):

- Given a *movie*, returns the list of all actors in that movie
  - *movie* is the name of a movie as a String
  - *actorsInfo* is the ArrayList to get the data from
- Given a non-existent movie, this method should return an empty list
- Consider different kinds of user input when writing your test cases. For example:
  - Non-existent movies
  - Movie names should not be case sensitive, but spelling must be correct. For example, "Rocky ii" and "Rocky II" should be considered the same movie in the database.
  - Movie names with leading or trailing whitespace (e.g. " Arrival "). For example, " Arrival " and "Arrival" should be considered the same movie in the database.

public ArrayList <String> selectWhereRatingIs (char comparison, int targetRating, boolean isCritic, ArrayList <Movie> moviesInfo)

- This useful method returns a list of movies that satisfy an inequality or equality, based on the *comparison* argument and the targeted rating argument
  - *comparison* is either '=', '>', or '< ' and is passed in as a char
  - *isCritic* is a boolean that represents whether we are interested in the critics rating or the audience rating. true = critic ratings, false = audience ratings.
  - *targetRating* is an integer
- Given incorrect input (e.g.) *targetRating* out of range of 0 <= targetRating <= 100) or *comparison* is not '=' or '<' or '>', this method should return an empty list
- Some examples of expected output are:
  - *selectWhereRatingIs('>', 0, true, moviesInfo)* should return every movie in the database, assuming no movie has a critic rating equal to 0.
  - *selectWhereRatingIs('=', 65, false, moviesInfo)* should return every movie that has an audience rating of exactly 65.

- *selectWhereRatingIs('<', 30, true, moviesInfo)* should return every movie that has a critic rating less than 30. Basically the ones the critics hated!
- Consider different kinds of user input when writing your test cases. For example:
  - when *comparison* is non-existent (e.g. '?'), return an empty list.
  - when *targetRating* that is out of range, return an empty list.

More Fun Methods

Outside of the 5 utility methods above, we would like to be able to answer some other interesting movie trivia questions and hence these other 5 methods are required. **Remember code reuse when you write these methods!**

public ArrayList <String> getCoActors (String actor, ArrayList <Actor> actorsInfo):

- Returns a list of all actors that the given *actor* has ever worked with in any movie except the *actor* herself/himself. You may think of how to use the 5 basic utility methods implemented previously.
  - *actor* is the name of an actor as a String
  - *actorsInfo* is the ArrayList to search through
- Consider different kinds of user input when writing your test cases. For example:
  - Existing actors and non-existent actors (return an empty ArrayList)
  - Actor names should not be case sensitive, but spelling must be correct. For example, "AMY ADAms" and "Amy Adams" should be considered the same actor in the database.
  - Actor names with leading or trailing whitespace (e.g. " Sylvester Stallone "). For example, "Bradley Cooper " and "Bradley Cooper" should be considered the same actor in the database.

public ArrayList <String> getCommonMovie (String actor1, String actor2, ArrayList <Actor> actorsInfo):

- Returns a list of movie names where both actors were cast. You may think of how to use the 5 basic utility methods implemented previously.
  - *actor1* and *actor2* are actor names as Strings
  - *actorsInfo* is the ArrayList to search through

- In cases where the two actors have never worked together, this method returns an empty list
- Consider different kinds of user input when writing your test cases. For example:
    - Existing actors and non-existent actors. When *actor1* or *actor2* is non-existent, there should be no movies in common.
    - Actor names should not be case sensitive, but spelling must be correct. For example, “AMY ADAms” and “Amy Adams” should be considered the same actor in the database.
    - Actor names with leading or trailing whitespace (e.g. “ Sylvester Stallone “) For example, “Bradley Cooper “ and “Bradley Cooper“ should be considered the same actor in the database.
    - *actor1* and *actor2* can be the same, in this case, return an ArrayList of movie names that this actor was cast in.

public ArrayList <String> goodMovies (ArrayList <Movie> moviesInfo):

- Returns a list of movie names that both critics and the audience have rated above 85 (>= 85).

public ArrayList <String> getCommonActors (String movie1, String movie2, ArrayList <Actor> actorsInfo)

- Given a pair of movies, this method returns a list of actors that acted in both movies. You may think of how to use the 5 basic utility methods implemented previously.
    - *movie1* and *movie2* are the names of movies as Strings
    - *actorsInfo* is the actor ArrayList
- In cases where the movies have no actors in common, it returns an empty list
- When writing test cases for this method, consider different kinds of user input. For example:
    - Existing movies and non-existent movies. When *movie1* or *movie2* is non-existent, there should be no actors in common.

- Movie names should not be case sensitive, but spelling must be correct. For example, “Rocky ii” and “Rocky II” should be considered the same movie in the database.
- Movie names with leading or trailing whitespace (e.g. “ Arrival “). For example, “ Arrival “ and “Arrival“ should be considered the same movie in the database.
- *movie1* and *movie2* can be the same. In this case, return an ArrayList of actor names that were cast in this movie.

public static double [] getMean (ArrayList <Movie> moviesInfo)

- Given the *moviesInfo* DB, this static method returns the mean value of the critics ratings and the audience ratings.
  - Returns the mean values as a double array, where the 1st item (index 0) is the mean of all critics ratings and the 2nd item (index 1) is the mean of all audience ratings

Testing

You have now seen how Test Driven Development works. For this homework, you are required to write unit tests for each of the methods in MovieTrivia.java. While it might seem annoying at first, I highly recommend doing this using a proper test-driven approach. First, write the unit tests and then, implement the methods to pass your unit tests. Be thoughtful about writing a good suite of tests to ensure proper behavior for the methods. Test typical inputs and edge cases.

In the provided MovieTriviaTest.java file, we’ve created the test methods for you. We’ve also created **some of the test cases**. We’ve also created a setUp method, which by default, is called before each and every test method, and can be used to re-initialize objects. In our setUp method, we initialize the MovieTrivia object to be used by each test method in the testing file.

Note, the Actor and Movie classes do not need to be tested. Also, the setUp, printAllActors, and printAllMovies methods in the MovieTrivia class also do not need to be tested. Please do not modify the code in these methods or in the Actor, Movie, or MovieDB classes.

You are expected to write additional test cases for each of the test methods given in the MovieTriviaTest.java. Please make sure you have a total of at least 4 distinct, valid "test case scenarios" for each test method.

Hint: when testing getMean, insert a new rating, then get mean again.

Note, a "test case scenario" can include multiple test cases that test a single scenario. For example, see how we test the "insertActor" method below. First we initially call the "insertActor" method. Then we have a test case that tests the new size of the *actorsInfo* ArrayList, a test case that tests the name of the new actor in the *actorsInfo* ArrayList, a test case that tests the size of the actor's list of movies, and a test case that tests the name of one of the actor's movies. This is considered a complete "test case scenario".

```
mt.insertActor("test123", new String [] {"testmovie1", "testmovie2"},
mt.actorsInfo);
assertEquals(mt.actorsInfo.size(), 4);
assertEquals(mt.actorsInfo.get(mt.actorsInfo.size() - 1).getName(),
"test123");
assertEquals(mt.actorsInfo.get(mt.actorsInfo.size() -
1).getMoviesCast().size(), 2);
assertEquals(mt.actorsInfo.get(mt.actorsInfo.size() -
1).getMoviesCast().get(0), "testmovie1");
```

You are not required to test helper methods that you write, but you are welcome to test them.

Comments & Style

Write comments using // for any non-trivial lines of code. In general, all of the style conventions from Python also apply in Java. The main differences are in naming conventions (lowercase and underscores versus camelCase) and the syntax for comments (/* */ versus ''' ''' and // versus #). The content of your comments should be very similar and the length of your lines should not go off the right hand edge of the editing window.

Javadocs

If not already added, please add javadocs to all class definitions, methods, and instance variables.

What to Submit

Please submit all the classes in your entire Java project. Make sure it includes everything in your "src" folder, as well as any data files outside of the "src" folder. Do not remove the package declarations in each file and keep the files in their respective package subdirectory.

If you're working as part of a team, only one student from your team needs to submit the files. Include the members of your team as part of the @author tag in the Javadocs for the MovieTrivia class. If Brandon was working with Sarah, their Javadocs and @author tag would look something like:

```
6
7  /**
8   * Movie trivia class providing different methods for querying and updating a movie database.
9   * @author Brandon Krakowsky & Sarah Broomall
10  */
11 public class MovieTrivia {
12
```

Evaluation

1. Does your code function? Does it do what the specifications require? (10 pts)
    a. Did you implement the methods in the MovieTrivia class exactly as they have been defined in this document?
    b. We will run our own unit tests in addition to the ones provided and the ones you write.
2. Did you have a total of at least 4 distinct, valid test case scenarios for each test method? (10 pts)
    a. Did you pass your own test cases?
    b. Did you include general test case scenarios as well as edge cases?
3. Did you follow good programming practices? (7 pts)
    a. Did you reuse code to avoid repetition? When applicable, did you call some methods inside other methods to cut down on code repetition? (+4)
    b. Did you name additional variables and methods descriptively with camelCase? (+1)
    c. Did you add javadocs to classes, instance variables, and methods, and comments to all non-trivial code? (+2)

4. Did you set up the files correctly? Does it compile and is everything named correctly? (3 pts)