Metropolitan State University
ICS 240 - 50: Introduction to Data Structures
Spring 2022

Homework 01: Object-Oriented Programming
Total points: 40
Out: Tuesday, May 17, 2022
Due: 11:59 PM on Monday May 30, 2022
Last day to submit with penalty: 11:59 PM on Thursday, June 2, 2022

Objective

In this assignment, you will practice solving a problem using object-oriented programming and specifically, you will use the concept of **object composition** (i.e., **has-a** relationship between objects). You will implement a Java application, called `MovieApplication` that could be used in the movie industry. You are asked to implement three classes: `Movie`, `Distributor`, and `MovieDriver`. Each of these classes is described below.

Problem Description

Class names, methods names and parameter types must exactly match the specification for full credit.

`Movie` class

The `Movie` class represents a movie and has the following attributes: `name`(of type `String`), `directorName` (of type `String`), `earnings` (of type `double`), and `genre` (of type `int`). The possible `genre`s are Comedy (0), Action (1) or Fiction (2).

Implement the following for the `Movie` class. Use the names exactly as given (where applicable) and in the other cases, use standard naming conventions.

- A constructor to initialize all instance variables, except `earnings`. A value for each instance variable will be passed as a parameter to the constructor with the exception of `earnings`, which will be set to zero. The parameters will be passed in the order `name`, `directorName`, then `genre`.
- Getters for all instance variables.

- A setter method for each instance variable, except `earnings`.
- `addToEarnings`: a method than takes as input an `amount` (of type `double`) and adds that amount of money to the movie's `earnings`.
- `boolean equals(Object obj)`: a method to test the equality of two movies. Two movies are considered to be equal if they have the same `name`, the same `directorName`, and the same `genre` and the comparison for string attributes must be case insensitive.
- `String toString()`: a method that returns a nicely formatted string description of the movie. All instance variables should be displayed, separated by tabs (use "\t") and the entire `Movie` should be displayed on one line.

`Distributor` class

The `Distributor` class represents a distributor of movies. Every distributor can distribute zero or more movies and, in this simplistic application, a distributor can distribute at most five movies. The `Distributor` class has the instance variables named exactly as follows:

- `name`: a `string` that represents the distributor's name,
- `phone`: a `string` that represents the distributor's phone,
- `movies`: an instance variable of type **array** of `Movie` with length of 5.
- `numMovies`: an `integer` that indicates the number of movies that have been added to the `movies` array. Note that this number can be between 0 and 5.

Implement the following methods for the `Distributor` class. Use the names exactly as given (where applicable) and in the other cases, use standard naming conventions.

- A constructor that takes as input the distributor's name and phone (in that order), and creates the distributor by initializing the fields appropriately, and instantiating the array `movies`. Each `Movie` reference in the array will initially be null by default.
- Getters and setters for `name` and `phone`.
- A getter for movies. Use `Arrays.copyOf` to trim the `movies` array to the exact number of movies and return the trimmed array. The following statement does the trick.

```
return Arrays.copyOf(movies, numberOfMovies);
```

- `addMovie`: a method that takes a single parameter of type `Movie` and returns a `boolean`. The method attempts to add the input movie to the `movies` array. If the `movies` array is full, the movie does not get added and the method returns `false`. Otherwise, the movie is added and the method returns `true`. Note that `movies[0]` must be filled before `movies[1]` and `movies[1]` must be filled before `movies[2]`, etc. The field `numMovies` is updated as necessary.

- `addMovie`: this is an overload of `addMovie` method that takes four input parameters that represent a movie's `name`, `directorName`, `genre`, and `earnings` (in that order) and returns a `boolean`. The method creates a `Movie` object using the input parameters and attempts to add that object to the `movies` array. If the `movies` array is full, the movie does not get added and the method returns `false`. Otherwise, the movie is added and the method returns `true`. Note that `movies[0]` must be filled before `movies[1]` and `movies[1]` must be filled before `movies[2]`, etc. The field `numMovies` is updated as necessary.
- `getNumMovies`: a method that returns as output the number of current number of movies of that distributor.
- `totalEarnings`: a method that returns as output the total earnings for the distributor which is the sum of all earnings for the distributor's movies.
- `comedyEarnings`: a method that returns as output the total earnings of movies of the comedy genre.
- `addEarnings`: a method that takes as input two parameters (of type `String` and `double`) that represent a movie's name and earnings respectively and returns a `boolean` as output. The method searches the `movies` array for a movie with the same name as the first input parameter (case insensitive) and add the input earnings to that movie's earnings if found and return `true`. The method returns `false` if the input movie name is not found.
- `getNumGenre`: a method that takes as input an integer that represents a genre (i.e., 0, 1, or 2) and returns as output the number of distributor's movies of that genre. The method returns -1 if the input is invalid (i.e., less than 0 or greater than 2).
- `calculateTax`: a **static** method that takes two input parameters that represent (1) a `double` that represent the tax rate (e.g., 0.05 for 5% tax) and (2) an object of type `Distributor`. The method returns as output the amount of tax that need to be paid by the input distributor based on the distributor's total earning. For example, if the total earning for a distributor is $1M and the tax rate is 5%, then the method returns 50000.
- `equals`: a method to test the equality of two distributors. Two distributors are considered to be equal if they have the same `name` and the comparison must be case insensitive.
- `toString`: a method that returns a string representation of the distributor that includes the following:
  - distributor's name and phone,
  - number of movies for this distributor,
  - details of all movies, one per line (hint: use `toString` method of Movie)
  - the total earnings for that distributor.

`MovieDriver` class

Your driver class should perform the following actions:

- Create at least 7 `Movie` objects with your choice of movie names, director names, and genres.

- Create 2 `Distributor` objects with your choice of name, address, and phone number.
- Add 5 different movies to the first distributor. Attempt to add the sixth movie to the same distributor and demonstrate that this attempt does not crash the program, but the operation does not succeed.
- Add 2 movies to the second distributor object.
- Print the two distributors.
- Exercise every method of both the `Movie` and `Distributor` classes. Demonstrate that they work.
- Do not use a `Scanner` to read any inputs from the user. Instead, use hard coded values for the different methods' inputs.

Other Requirements

- Draw a UML diagram for your application and make sure to show the relationships between the classes in your solution. You may hand draw or use any tool of your choice but the diagram must be submitted in a format that I can read (one of .doc, .docx, .pdf, .jpg, .txt). If I cannot read it, I cannot grade it.
- Note that automated tools often do not follow the conventions that I require.
- Factor the classes as described above – `Movie`, `Distributor`, and `MovieDriver` are all in separate classes
- Comment the `Movie` and `Distributor` classes with Javadocs style comments (see Appendix H).
- Use good coding style throughout (see Lecture 1's slides and the CodingStandards.pdf in the Syllabus folder of D2L) including:
    - correct private/public access modifiers
    - consistent indenting
    - meaningful variable names
    - normal capitalization conventions
    - other aspects of easy-to-read code

Grading

Your grade in this assignment is based on the following:

- Your submission meets specifications as described above.
- You must use the exact same name (case matched) for all fields and methods as specified in the above description. The method input parameters must be in the same order that is specified above.
- The program is robust with no runtime errors or problems.
- You follow the good programming style as discussed in class (see the file CodingStandards.pdf under the Syllabus folder of D2L).

- Follow the submission instructions given below.
- Here is a broad outline of the rubric:
  - The `Movie`class is coded exactly as specified with respect to name, constructor organization, and methods and their signatures. (7 points)
  - The `Distributor` class is coded exactly as specified with respect to name, constructor, organization, and methods and their signatures. (10 points)
  - The `MovieDriver` class meets the minimum requirements set forth above. (8 points)
  - The program works correctly with my test data. (10 points)
  - The program is coded as per coding standards. (5 points)

Submission Instructions

Follow the following steps to upload your code to D2L:

- Create a java project and call it `<yourLastName>Assignment1`(e.g., mine will be called *`CassidyAssignment1`*)
- Create three .java files to implement the classes as described above.
- Archive the four files into **one zip** file. using Eclipse using the following steps:
  - In Eclipse Project Explorer, right click on the project folder of the project and click on Export.
  - Choose ***General*** then ***Archive File*** and click ***Next***.
  - Use the Browse key to choose a folder to store the archive file on your hard drive and give the file the same name as your project (e.g., `CassidyAssignment1.zip`), then click **Save**, then click **Finish**.
- Upload **only one** **.zip** file to the D2L folder called Assignment 1.
- **It is important that you upload your code in only one zip file. Your assignment will not be graded if you upload individual files to the drop box.**
- You may submit more than once, but then I will grade the most recent submission.