COMP 2150 - Spring 2021

Project: Too Much to Watch

Total Points: 100

Due: Friday, Apr. 30, by 2359 CDT

I've provided a data file on eCourseware with all of Netflix's streaming offerings as of early 2021[1]. The file is in CSV (comma-separated values) format and can be opened in either a spreadsheet or a text editor to view its contents. The first line of the file includes information on what each column represents. Each subsequent line is called a **record** and contains info about a single movie or series.

Each record is divided into **fields**, which are separated by commas (thus the name "comma-separated values"). The fields used in this project are:

- Title
- Director
- Cast
- Country

[1]Source: Shivam Bansal, `https://www.kaggle.com/shivamb/netflix-shows`

- Release year
- Rating (G, PG, TV-MA, etc.)
- Duration (runtime in minutes for a movie, or number of seasons for a series)
- Genre (action, sci-fi, etc. — the file calls this `listed_in`)
- Description

Note that some fields are absent from some records; these are indicated with an empty string between the separating commas. Fields that themselves contain commas are placed between double quotes, to prevent confusion with the commas being used to separate fields from one another. Double quotes that are meant to be included as part of a field are represented as two consecutive double quotes (""). For example, consider line 782 of the file:

```
s781,Movie,Beak & Brain: Genius Birds From Down Under,"Volker Arzt, Angelika
Sigl",,Germany,"March 1, 2017",2013,TV-G,52 min,"Documentaries, International
Movies","Whoever came up with the term ""bird brain"" never met these feathered
thinkers, who use their claws and beaks to solve puzzles, make tools and more."
```

Here, the director field is

```
Volker Arzt, Angelika Sigl
```

The cast field is empty (presumably the program stars the birds), and the description field is

```
Whoever came up with the term "bird brain" never met these feathered thinkers,
who use their claws and beaks to solve puzzles, make tools and more.
```

Project Specifications

In this project you'll be writing some software that parses (reads) the data file and allows the user to apply and remove filters to customize the results. That should make it easier to pick from the huge selection of options available! For example, the user may want to see a list of all the movies released in 2000 or later that were directed by Christopher Nolan. This would be expressed using the following three filters:

1. `movie`
2. `year >= 2000`
3. `director nolan`

The user should be able to add as many filters as s/he desires, and the software should allow the user to display a list of the media records from the data file that match all the filters. The user should also be able to remove filters at will. Every time a filter is added or removed, the software should indicate 1) the currently active filters, and 2) how many items from the data file match *all* filters in the filter list.

Your software must support the following filter formats:

- `movie` - matches any movie
- `series` - matches any series
- `title` `____` - matches any title that contains[2] a specific string
- `director` `____` - matches any director that contains a specific string
- `cast` `____` - matches any cast that contains a specific string
- `country` `____` - matches any country that contains a specific string
- `rating` `____` - matches any rating that equals a specific string. Note that unlike the others, this one should be an *exact* (ignoring case) match.
- `genre` `____` - matches any genre that contains a specific string
- Filters that involve the release year. The blank should be a single integer indicating the year.
  - `year <` `____`
  - `year >` `____`
  - `year <=` `____`
  - `year >=` `____`
  - `year =` `____`
- Filters that involve the runtime. The blank should be a single integer indicating the runtime in minutes. These filters work only for movies; they do not match any series.
  - `runtime <` `____`
  - `runtime >` `____`
  - `runtime <=` `____`
  - `runtime >=` `____`
  - `runtime =` `____`

All matching should be done without regard to case. If a filter is entered that doesn't fit any of these formats, the filter should match any media that contains the filter's text in the title, director, cast, country, genre, or description.

[2] An exact match is not necessary. As long as any substring of the title matches the target string, it counts.

Class Design

Your project must include the following classes. I'm purposely a little vague about the class design to give you some freedom to explore and implement the details as you see fit.

1. (20 points) An abstract `Media` class, with concrete `Movie` and `Series` subclasses. These classes should be designed to hold the relevant fields mentioned earlier: title, director, cast, country, release year, rating, duration, genre, description. Include `toString` methods in both of the subclasses; each one should return a string indicating the type of media (movie or series) and all of the instance variables.

2. (30 points) A `DataFileParser` class that handles reading the provided data file. For each record in the file, a new `Media` object (either `Movie` or `Series`) should be created and added to a list of `Media` objects. This class should include a method that returns the finished list.

3. (25 points) A `Filter` class that represents a single filter.[3] Every time the user adds a filter, a new `Filter` object should be created and added to a list. Removing a filter should remove it from this list. The `Filter` class should include a method that determines whether or not the filter matches a specific `Media` object.

4. (25 points) A client program with a main method to run. Here's an outline of what the client program needs to do:

   - Read the data file and create a new `Media` object for each record in the file. Store these objects in a list (let's call this `masterList`).
   - Allow the user to add or remove any number of filters. These filters should be stored in a separate list (let's call this `filterList`).
   - Each time the user adds a new filter or removes an existing filter, your code should search the `masterList` to find only the objects that match all the filters in the `filterList`. Copy those objects into another list (let's call this `currentList`). Then, you can simply list the items in the `currentList` to see which items match all filters.

Other Requirements

- User input should happen only in the client program. Implement input validation on *all* user inputs, including exception handling where appropriate. Invalid input should show an error message and prompt the user to try again. Your software should never crash due to user input!

- Use Java's built-in classes (either `java.util.ArrayList` or `java.util.LinkedList`) for the lists of `Media` objects and the list of `Filter` objects.

- How you enter filters in the client program is up to you. The version I'll demonstrate in class allows the user to type in the filters directly, but if you prefer to write a menu-based system that's OK too.

[3] You might be tempted to store the filters as plain strings, but making a class will allow you to more cleanly access the separate parts of each filter.

Hints

- The file reading part is not super difficult, but it is not trivial. Get started on this part early, since you need it for the rest of the project! You will *not* be able to just call `split(",")`, because some of the commas on each line might themselves be part of a CSV field.
- Refer to the `String` class in the Java API for some helpful methods. You're welcome to use any of them in your solution.
- Java does not allow strings to be cast to primitive types. However, you can use the static method `Integer.parseInt` to convert a string into an `int`. If you try passing an argument that can't be read as an `int`, this method throws a `NumberFormatException`.

Code Guidelines

Points can be deducted for not following these guidelines!

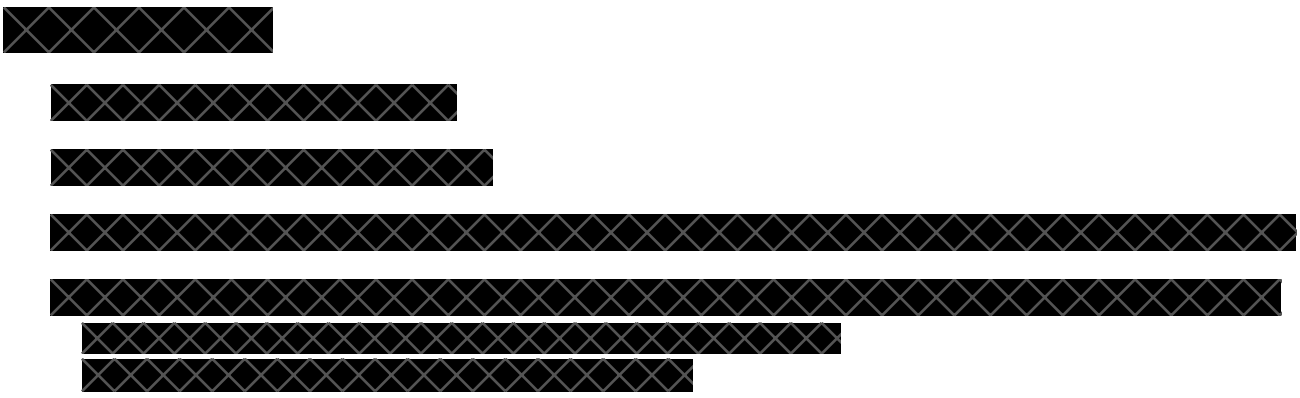
- Most importantly, your code *must* compile and run. Code that does not compile and run may receive zero credit, at the TA's discretion.
- Follow Java capitalization conventions for `ClassNames`, `variableAndMethodNames`, and `CONSTANT_NAMES`.
- Use consistent indentation throughout your code.
- Follow one of these two conventions for curly braces. You can pick either one, but follow it consistently.

  –
  ```
  StyleA {
      // stuff here
  }
  ```

  –
  ```
  StyleB
  {
      // stuff here
  }
  ```

- Include a reasonable amount of comments in your code. "Reasonable" is somewhat subjective, but at the very least include:

  1. A comment at the top of each class summarizing what it does. *Also include your name (and your teammate's name, if applicable) in this section.*
  2. A comment before each method summarizing what it does, its parameters (if any), and its return value (if any).
  3. Comments that indicate the major steps taken by the code. There are generally at least a few of these per class file.



[redacted]

[redacted]

- [redacted]
- [redacted]
- [redacted]