# Homework 05 – Blockbuster

Authors: Archie, Nicolas, Eric
Topics: ArrayList using generics, asymptotics, searching, sorting

## Problem Description

Throwback to the 90s! Olivia lives near the last Blockbuster rental store in Oregon and wants to organize a laid-back media night with her friends. Help Olivia pick the right films and games so she can have a great time!

## Solution Description

You will need to complete and turn in six classes: Genre.java, Media.java, Movie.java, VideoGame.java, Blockbuster.java, and Olivia.java.

**Notes:**

1. All variables should be inaccessible from other classes and must require an instance to be accessed through, unless specified otherwise.
1. All methods should be accessible from everywhere and must require an instance to be accessed through, unless specified otherwise.
3. Use constructor and method chaining whenever possible! Ensure that your constructor chaining helps you reduce code reusage!
4. **Reuse code when possible.** Helper methods with appropriate visibility are optional.
5. Make sure to add all Javadoc comments to your methods and classes!

### *Genre.java*

This enum enumerates the most common genres supported by Blockbuster's movies and video games. The genres this enum should define are: `ACTION`, `COMEDY`, `FANTASY`, `HORROR`, `MYSTERY`, `ROMANCE`, and `SCI_FI` (in this order).

### *Media.java*

This class defines the basic behaviors of media items a customer can check out. This class should implement the Comparable interface using generics such that an instance of Media can only be compared to other Media instances. **This class should never be instantiated.**

**Variables:**

- `Genre genre` – the genre of this media item
- `String name` – the name of this media item
- `int rating` – the audience rating of this media item
- `double rentalPrice` – the price in dollars to rent this media item

**Constructors:**

- A constructor that takes in `genre`, `name`, `rating`, and `rentalPrice`.

- A constructor that takes in `genre`, `name`, and `rating` and sets `rentalPrice` to 7.0.
- Assume all inputs to constructors will be valid (non-null, reasonable values).

**Methods:**

- `toString`
  - This method should properly override the `toString` method from Object.
  - Return a String in the following format (single line, without curly braces, single space following each colon and comma):

    ```
    "Genre: {genre}, Name: {name}, Rating: {rating},
    Rental Price: ${rentalPrice}"
    ```

    - Display floating-point values as rounded to two decimal places.
- `equals`
  - This method should properly override the `equals` method from Object.
  - The implementation must be symmetric.
  - Two media items are considered equal if they have equal `genre`, `name`, `rating`, and `rentalPrice`.
- `compareTo`
  - This method should properly override the `compareTo` method from Comparable.
  - A media item is greater than the other if its genre has a greater ordinal than the other.
    - ***Hint***: Refer to the Enum API for any useful methods!
  - If both media items have the same genre, then the media item with the lexicographically greater name is considered greater.
    - ***Hint***: Refer to the String API for any useful methods!
  - If both media items have the same genre and name, then the media item with the higher rating is considered greater.
  - This means that media items are compared based on genre first, name second, and rating last. You may assume the input will not be null.
- Getters for `name`, `rating`, and `rentalPrice`. **No other getters or setters.**

## *Movie.java*

This class extends from the `Media` class and describes a movie, a specific kind of media item that Olivia can check out.

**Variables:**

- `int runtime` – the runtime of the movie in minutes

**Constructors:**

- A constructor that takes in `genre`, `name`, `rating`, `rentalPrice`, and `runtime`.
- A constructor that takes in `genre`, `name`, and `rating`, and sets `rentalPrice` to 5.0 and `runtime` to 111.
- Assume all inputs to constructors will be valid (non-null, reasonable values).

**Methods:**

- `toString`
  - This method should properly override the `toString` method from Media.

  - Return a String in the following format (single line, without curly braces, single space following each colon and comma):

    ```
    "Genre: {genre}, Name: {name}, Rating: {rating},
    Rental Price: ${rentalPrice}, Runtime: {runtime}"
    ```

    - Display floating-point values as rounded to two decimal places.
- `equals`
  - This method should properly override the `equals` method from Media.
  - The implementation must be symmetric.
  - Two movies are considered equal if they have equal `genre`, `name`, `rating`, `rentalPrice`, and `runtime`.
- **No getters or setters**.

## *VideoGame.java*

This class extends from the `Media` class and describes a video game, a specific kind of media item that Olivia can check out.

**Variables:**

- `int maxPlayers` – the maximum number of players that can play the game at once
- `boolean needsConsole` – whether the video game needs a gaming console to play

**Constructors:**

- A constructor that takes in `genre`, `name`, `rating`, `rentalPrice`, `maxPlayers`, and `needsConsole`.
- A constructor that takes in `genre`, `name`, and `rating`, and sets `rentalPrice` to 5.0, `maxPlayers` to 2, and `needsConsole` to false.
- Assume all inputs to constructors will be valid (non-null, reasonable values).

**Methods:**

- `toString`
  - This method should properly override the `toString` method from Media.
  - Return a String in the following format (single line, without curly braces, single space following each colon and comma):

    ```
    "Genre: {genre}, Name: {name}, Rating: {rating},
    Rental Price: ${rentalPrice}, Players: {maxPlayers},
    {does/does not} need console"
    ```

    - The `{does/does not}` depends on the value of `needsConsole`.
    - Display floating-point values as rounded to two decimal places.
- `equals`
  - This method should properly override the `equals` method from Media.
  - The implementation must be symmetric.
  - Two video games are considered equal if they have equal `genre`, `name`, `rating`, `rentalPrice`, `maxPlayers`, and `needsConsole`.
- Getter for `needsConsole`. **No other getters or setters**.

## *Blockbuster.java*

This class represents a Blockbuster store. The store has both video games and movies for Olivia to check out.

**Variables:**

- `ArrayList inventory` – an ArrayList **of type Media** representing the store's inventory

**Constructors:**

- A no-argument constructor that initializes `inventory` to an `ArrayList` of type `Media`.
    - **Note**: Depending on your implementation, this constructor may not be necessary. However, you must write this constructor for grading purposes.

**Methods:**

- `addMedia`
    - This method takes in a non-null media item and adds it to the end of `inventory`.
    - This method should not return anything.
    - This method should run in $O(1)$ time.
- `removeMedia`
    - This method takes in a non-null media item and removes the first occurrence of it.
    - Return the media item that is removed or `null` if the media item is not found.
        - **Note**: If a media item is removed, you should return the media item that was removed from the store rather than returning the inputted media item.
    - Use the media item's `equals` method to identify the first occurrence of the inputted media item in the store.
    - This method should run in $O(n)$ time.
- `sortMedia`
    - This method sorts the store's inventory in **ascending** order based on genre first, name second, and rating last.
        - ***Hint***: What method compares two media items based on genre first, name second, and rating last?
    - This method does not return anything.
    - This method should run in $O(n^2)$ time.
- `findMedia`
    - This method takes in a non-null media item and finds and returns the item in the store that has the same `genre`, `name`, and `rating`.
        - **Note**: Assume that if the item can be found in the store, then that item will also be equal to the inputted item according to the `equals` method.
        - **Note**: If the media item is found, you should return the media item that is in the store rather than returning the inputted media item.
    - Assume that the items in the store are sorted based on genre first, name second, and rating last.
    - If a media item with the same `genre`, `name`, and `rating` is not found, return `null`.
    - Assume that the store has **unique** media items based on genre, name, and rating (i.e., no two media items in the store have the same genre, name, and rating).
    - This method should run in $O(\log n)$ time.

- `getMostPopularMovie`
    - This method returns the most popular movie based on audience rating.
    - If there is a tie among movies in audience rating, the movie whose name lexicographically precedes the others is considered more popular.
    - Assume that the names of the movies in the store are unique.
    - If the store does not have any movies, return `null`.
    - This method should run in $O(n)$ time.
- **No getters or setters.**

## *Olivia.java*

This class defines Olivia's states and behaviors. All states and behaviors should be static. Notice that you have already implemented most of the code necessary for Olivia's behaviors (hint: reuse code).

**Variables:**

- `double budget` – the total amount of money Olivia can spend on media rentals
- `ArrayList cart` – the media items in Olivia's cart
- `boolean canUseConsole` – whether Olivia's console at home is currently working

**Methods:**

- `addToCart`
    - This method takes in a non-null `Media` item Olivia wants to rent and a non-null `Blockbuster` store she is shopping at.
    - If Olivia has enough money in her budget to rent the media item **AND** a media item with the same genre, name, and rating is available in the store, remove that item from the store, add that item to Olivia's cart, and return true.
        - **Note**: Assume that if the item can be found in the store, then that item will also be equal to the inputted item according to the `equals` method.
        - **Note**: You should add the media item that was removed from the store to Olivia's cart rather than adding the inputted media item.
        - Assume that the store has **unique** media items based on genre, name, and rating (i.e., no two media items in the store have the same genre, name, and rating).
        - Assume that the items in the store are sorted based on genre first, name second, and rating last.
    - ***Hint***: What method in `Blockbuster` searches for media items in a sorted store?
    - ***Hint***: What method in `Blockbuster` removes an item from the store?
    - Update `budget` accordingly if necessary.
    - If the media item is a video game that needs a console, Olivia will only add the item to her cart if her console at home is currently working.
    - If Olivia could not find the media item in the store, does not have enough money in her budget, or if the media item was a video game and she cannot play it on her console, do not add the media item to the cart and return false.
    - Assume that the input `Blockbuster` store's `inventory` will have unique `Media` items.
    - This method should run in $O(n)$ time.

- `changeMind`
    - This method takes in a non-null `Media` item that is in Olivia's `cart` and a non-null `Blockbuster` store she is shopping at.
    - Olivia has changed her mind about renting the specified media item and wants to remove the first occurrence of the item from her cart and add it back to the store.
        - **Note**: You should add the media item that was removed from the cart to the store rather than adding the inputted media item.
        - **Note**: Use the media item's `equals` method to identify the first occurrence of the inputted media item in Olivia's cart.
    - Update `budget` accordingly.
    - This method should not return anything.
    - This method should run in $O(n)$ time.

## Checkstyle

You must run Checkstyle on your submission (to learn more about Checkstyle, check out cs1331-style-guide.pdf under the Checkstyle Resources module on Canvas). **The Checkstyle cap for this assignment is 30 points.** This means there is a maximum point deduction of 30. If you don't have Checkstyle yet, download it from Canvas → Modules → Checkstyle Resources → checkstyle-8.28.jar. Place it in the same folder as the files you want to run Checkstyle on. 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 the number of points we would take off (limited by the Checkstyle cap). In future assignments we will be increasing this cap, so get into the habit of fixing these style errors early!

Additionally, you must 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
```

For additional help with Checkstyle see the CS 1331 Style Guide.

## Turn-In Procedure

### *Submission*

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

- `Genre.java`
- `Media.java`
- `Movie.java`

- `VideoGame.java`
- `Blockbuster.java`
- `Olivia.java`

Make sure you see the message stating the assignment was submitted successfully. From this point, Gradescope will run a basic autograder on your submission as discussed in the next section. **Any autograder tests are provided as a courtesy to help "sanity check" your work and you may not see all the test cases used to grade your work.** You are responsible for thoroughly testing your submission on your own to ensure you have fulfilled the requirements of this assignment. If you have questions about the requirements given, reach out to a TA or Professor via the class forum for clarification.

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

## *Gradescope Autograder*

If an autograder is enabled for this assignment, you may be able to see the results of a few basic test cases on your code. Typically, tests will correspond to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue. **We reserve the right to hide any or all test cases, so you should make sure to test your code thoroughly against the assignment's requirements.**

The Gradescope tests serve two main purposes:

- Prevent upload mistakes (e.g., 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.

## *Burden of Testing*

You are responsible for thoroughly testing your submission against the written requirements to ensure you have fulfilled the requirements of this assignment.

Be **very careful** to note the way in which text output is formatted and spelled. Minor discrepancies could result in failed autograder cases.

If you have questions about the requirements given, reach out to a TA or Professor via the class forum for clarification.

## Allowed Imports

- `java.util.ArrayList`

## Feature Restrictions

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

- `var` (the reserved keyword)
- `System.exit`
- `System.arraycopy`

## Collaboration

Only discussion of the assignment at a conceptual high level is allowed. You can discuss course concepts and assignments broadly; that is, at a conceptual level to increase your understanding. If you find yourself dropping to a level where specific Java code is being discussed, that is going too far. Those discussions should be reserved for the instructor and TAs. To be clear, you should never exchange code related to an assignment with anyone other than the instructor and TAs.

The only code you may share are test cases written in a Driver class. If you choose to share your Driver class, they should be posted to the assignment discussion thread on the course discussion forum. We encourage you to write test cases and share them with your classmates, but we will not verify their correctness (i.e., use them at your own risk).

## 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 Ed Discussion for a note containing all official clarifications and sample outputs.**

It is expected that everyone will follow the Student-Faculty Expectations document and the Student Code of Conduct. The professor expects a **positive, respectful, and engaged academic environment** inside the classroom, outside the classroom, in all electronic communications, on all file submissions, and on any document submitted throughout the duration of the course. **No inappropriate language is to be used, and any assignment deemed by the professor to contain inappropriate offensive language or threats will get a zero.** You are to use professionalism in your work. Violations of this conduct policy will be turned over to the Office of Student Integrity for misconduct.