Homework 06 – Watching Sports

Authors: Sabina, Daniel, David, Eric
Topics: polymorphism, exceptions, file I/O

Problem Description

You and your friends are big sports fans and want to look for some sports games to attend. As you look, you become overwhelmed by your abundance of choices. To help you decide, you create a database that stores information about the sports games you bought tickets for and will watch.

Solution Description

You will need to complete and turn in 5 classes: SportsGame.java, BasketballGame.java, FootballGame.java, InvalidTicketException.java, and Tickets.java.

Notes:

1. All variables should be inaccessible from other classes and require an instance to be accessed through, unless specified otherwise.
2. 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. **When throwing exceptions, a descriptive and specific message should be provided.**
6. Make sure to add Javadoc comments to your methods and classes!

SportsGame.java

This abstract class represents a sports game in general. SportsGame should never be instantiated.

Variables:

- `String venue` – the location of the game
- `String startTime` – the start time of the game
  - Represented in the format "HH:MM" (e.g., 13:31)
- `String startDate` – the start date of the game
  - Represented in the format "MM-DD-YYYY" (e.g., 01-31-1331)
- `int score1` – the score of the first team
- `int score2` – the score of the second team
- `int seatsLeft` – the number of remaining seats at the game

Constructors:

- A constructor that takes in `venue`, `startTime`, `startDate`, `score1`, `score2`, and `seatsLeft`.
  - If the inputted value for `venue`, `startTime`, or `startDate` is blank or null, throw an `IllegalArgumentException`.
  - If the inputted value for `score1`, `score2`, or `seatsLeft` is negative, throw an `IllegalArgumentException`.
  - Leading and trailing whitespace from String inputs should be removed.

Methods:

- `toString`
  - This method should override Object's `toString` method and return a String in the following format (single line, no curly braces, no quotation marks, no spaces preceding/following commas, no leading/trailing whitespace):

```
"{venue},{startTime},{startDate},{score1},{score2},
{seatsLeft}"
```

- `equals`
  - This method should override the `equals` method from Object.
  - Two `SportsGame` objects are equal if they have the same `venue`, `startTime`, `startDate`, `score1`, `score2`, and `seatsLeft`.
- Getters and setters as necessary.

BasketballGame.java

This concrete subclass of SportsGame represents a basketball game.

Variables:

- `String league` – the name of the league that the basketball teams are a part of

Constructors:

- A constructor that takes in `venue`, `startTime`, `startDate`, `score1`, `score2`, `seatsLeft`, and `league`.
  - If the inputted value for `league` is blank or null, throw an `IllegalArgumentException`.
  - Leading and trailing whitespace from String inputs should be removed.

Methods:

- `toString`
  - This method should override SportsGame's `toString` method and return a String in the following format (single line, no curly braces, no quotation marks, no spaces preceding/following commas, no leading/trailing whitespace):

```
"BasketballGame,{venue},{startTime},{startDate},
{score1},{score2},{seatsLeft},{league}"
```

- `equals`
  - This method should override the `equals` method from SportsGame.
  - Two `BasketballGame` objects are equal if they have the same `venue`, `startTime`, `startDate`, `score1`, `score2`, `seatsLeft`, and `league`.
- Reuse as much code as possible!

FootballGame.java

This concrete subclass of SportsGame represents a football game.

Variables:

- `String performer` – the name of the halftime performer at the football game

Constructors:

- A constructor that takes in `venue`, `startTime`, `startDate`, `score1`, `score2`, `seatsLeft`, and `performer`.
  - If the inputted value for `performer` is blank or null, throw an `IllegalArgumentException`.
  - Leading and trailing whitespace from String inputs should be removed.

Methods:

- `toString`
  - This method should override SportsGame's `toString` method and return a String in the following format (single line, no curly braces, no quotation marks, no spaces preceding/following commas, no leading/trailing whitespace):

```
"FootballGame,{venue},{startTime},{startDate},{score1},
{score2},{seatsLeft},{performer}"
```

- `equals`
  - This method should override the `equals` method from SportsGame.
  - Two `FootballGame` objects are equal if they have the same `venue`, `startTime`, `startDate`, `score1`, `score2`, `seatsLeft`, and `performer`.
- Reuse as much code as possible!

InvalidTicketException.java

This class describes a **checked** exception that is thrown when encountering an invalid ticket. Note that this class should be a child of the most generic checked exception class in the `java.lang` package.

Constructors:

- A constructor that takes in a String representing the exception's message.
- A no-argument constructor that has default message "Invalid ticket".

Tickets.java

This class defines various static methods that allow a user to read and write to the database. Any exceptions that you throw from within the methods of this class should propagate to another method. The database is represented by a comma-separated value (CSV) file. We recommend that you open the CSV file in a plain text editor rather than a spreadsheet editor so you can see the delimiting commas. Please refer to the clarification thread on Ed Discussion for a sample CSV file.

Methods:

- `retrieveGames`
  - Takes in a `String` representing the path name of the file to read from.
  - Throws a `FileNotFoundException` if the inputted path name is blank or null, or the path name is valid but is not the path to a file.
  - If the path name represents the path to an existing file, assume the file is readable.
  - Returns an `ArrayList` of `SportsGame` objects created from the tokens.
  - Each line of the file will contain information about a sports game.
    - File line tokens will be in the following format on each line, without curly braces:

      ```
      {GameType},{venue},{startTime},{startDate},{score1},
      {score2},{seatsLeft},{league/performer}
      ```
    - Iterate through the file and create a `SportsGame` object from each token.
      - **Note**: There can be any number of lines in the file, but no lines will be empty (unless the file is empty).
    - Add each `SportsGame` object to an `ArrayList`.
    - If the game type in the token is not BasketballGame or FootballGame, throw an `InvalidTicketException`. You may assume other game details in the line are valid.
    - For example, an exception should be thrown given the following token:

      ```
      "SoccerGame,MBS,13:31,01-31-1331,3,0,6,MLS"
      ```
- `processInfo`
  - This is a private helper method that will be used to process a line of text from the CSV into a `BasketballGame` or `FootballGame` object.
  - Takes in a String representing the line of info to process.
  - Returns a `SportsGame` object that is either a BasketballGame or FootballGame.
  - If the game type in the token is not BasketballGame or FootballGame, throw an `InvalidTicketException`.
  - **IMPORTANT**: Do **NOT** use a second Scanner object to process this line as it will break the autograder.
  - ***Hint***: The String `split` method can be used to split a String on a specified delimiter into an array of tokens. Think about what character you can split the line apart with to separate the information.

- `purchaseTickets`
  - Takes in a `String` representing the path name of the file to write to and an `ArrayList` of `SportsGame` objects. Returns nothing.
  - If the inputted path name is blank or null, throw an `IllegalArgumentException`.
  - If a file with the inputted path name does not exist, create a file with the given name.
    - You may assume the directory is writable.
  - If a file with the inputted path name exists, you may assume it is readable and does not contain empty lines (unless the file is empty).
  - Iterate through the ArrayList and write each SportsGame to a new line.
    - Keep the lines in the file contiguous (i.e., there should be no gaps between the lines in the file).
  - If a SportsGame has no remaining seats, do not write it to the file.
  - **Note**: If the file exists and there is already information contained in the given file, it should not be overwritten. Instead, you should add onto the end of the existing file.
    - ***Hint***: How can you reuse code to read the file?
  - To implement this, you should follow a *read-modify-write* design pattern. This pattern can be used to change the contents of a file in the following three steps:
    - Read data from a file into an intermediate data structure (e.g., an ArrayList).
    - Perform any necessary modifications to that data in the intermediate structure.
    - Write the data in the intermediate data structure back out to the same file.

    This allows us to use familiar tools like Scanner and PrintWriter to change the contents of a file without modifying the contents of the file *directly* (which is much more difficult).
- `findTickets`
  - Takes in a `String` representing the path name of the file to read from and a `SportsGame` object.
  - Throws a `FileNotFoundException` if the inputted path name is blank or null, or the path name is valid but is not the path to a file.
  - Returns an `ArrayList` of `Integer` objects.
  - Iterate through the file and find all occurrences of the SportsGame object.
    - When you find an occurrence of the inputted SportsGame, add its corresponding line number to the ArrayList. The first line of a file is line 0.
    - ***Hint***: How can you reuse code to read the file?
  - If the inputted SportsGame object is not found, throw an `InvalidTicketException`.
  - **Example:** If the SportsGame object is found on lines 1, 3, and 9, return an ArrayList containing Integers 1, 3, and 9.
- `attendGame`
  - Takes in a `String` representing the path name of the file to write to and a `SportsGame` object. Returns nothing.
  - Throws a `FileNotFoundException` if the inputted path name is blank or null, or the path name is valid but is not the path to a file.
  - Iterate through the file and find all instances of the SportsGame object.
    - ***Hint***: How can you reuse code by calling some of the methods already written?
  - Remove each occurrence of the SportsGame from the file, keeping the lines in the file contiguous (i.e., there should be no gaps between the lines in the file).
  - If the inputted SportsGame is not found, throw an `InvalidTicketException`.

Driver.java

You will use this class to test your code.

Methods:

- `main`
  - Create two BasketballGame and two FootballGame objects.
  - Write the objects into a file called "TestTickets.csv" using `purchaseTickets`.
    - **Note:** It is recommended to open the CSV file using a text editor rather a spreadsheet editor when viewing or editing the file.
  - Create another BasketballGame object and add it to "TestTickets.csv".
  - Read the CSV file into an ArrayList using `retrieveGames` and print each object to a new line.
  - Call `attendGame` on a SportsGame currently in the CSV file to remove each occurrence from the file.
  - **Note:** This is to help you get started with testing your code, and it is not comprehensive. It is *strongly* recommended that you create more test cases.

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 35 points.** This means there is a maximum point deduction of 35. 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:

- `SportsGame.java`
- `BasketballGame.java`
- `FootballGame.java`
- `InvalidTicketException.java`
- `Tickets.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`
- `java.util.Scanner`
- `java.io.File`
- `java.io.FileNotFoundException`
- `java.io.IOException`
- `java.io.PrintWriter`

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.