Homework 04 – 4 Flags Amusement Park

Authors: Sabina, Chloe, Helen, Ricky, Sang Yoon
Topics: Polymorphism, Dynamic Binding, Interfaces, Comparable

Problem Description

Please make sure to read all parts of this document carefully.

A new amusement park has opened around Tech called "4 Flags." However, with all the different attractions in the park to manage, you decide to use your skills in object-oriented programming to make it easier to maintain the amusement park. In this homework, you will use your knowledge of polymorphism to help with the management of many different attractions.

Solution Description

You will need to complete and turn in 1 interface and 4 classes: Admittable.java, Group.java, Attraction.java, RollerCoaster.java, and TripGuide.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 chaining and reuse methods whenever possible! Ensure that your constructor chaining helps you maximize code reuse!
4. Reuse code when possible and helper methods are optional.
5. Make sure to add Javadoc comments to your methods and classes!

Admittable.java

This will be an `interface` that declares a method to allow for the behavior of admittance to be required for an implementing class.

Methods:

- `admit()`
  - This method should not be implemented in Admittable. It should, when implemented, take in a `String[]` corresponding to the visitor name(s) to be admitted. This method will not return anything.

Group.java

This `class` will represent a group of visitors visiting an attraction at the same time.

Variables:

- `people` – A `String[]` representing a group of visitors that are visiting an attraction at the same time. This field should be **immutable**.

- **Note: Assume the individual elements of the String array will be valid (i.e. non-null, non-blank).**

Constructor:

- A constructor that takes in a `String[]` of `people`.
  - If the reference passed in is `null`, a new array should be created with length 0.
  - If the reference passed in is non-null, a deep copy of the array should be made.

Methods:

- `size()`
  - Returns an `int` representing the size of the group (i.e. the number of people in the group).
- `toString()`
  - This method should properly override `Object`'s `toString()` method.
    - **Note: Specifying `@Override` before the method header allows you to skip the Javadocs for that method and tells the compiler to double check that the method has been properly overridden.**
  - Prints out the names of the visitors in the group with each name separated by a "`/`".
  - For example, a `Group` with three people would print out "`<name1>/<name2>/<name3>`"
  - **Hint**: `substring()` can be used to remove any trailing "`/`" that you might have, depending on the method you use to create this String.
- Any helper methods necessary.
- Getters and setters as necessary

Attraction.java

This class represents an attraction you are visiting and will implement Admittable. Attractions can be compared based on their admission fee and rating and should properly follow the contract set by the Comparable interface using generics.

Variables:

- `name` – A `String` that represents the name of the attraction. In the case we try to instantiate an `Attraction` with a `null` or "blank" string, we default to a name of "`No name`". This field should be **immutable**.
  - Remember that the String library provides a method to test whether a String is "blank" (i.e. composed entirely of whitespace characters).
- `sumRatings` – A `long` containing a sum of all the ratings this attraction has received. It should be initialized to 0.
- `numRatings` – An `int` representing the total number of ratings this attraction has received. It should be initialized to 0.
- `admissionFee` – A `double` representing the fee for going into the attraction. This value cannot be negative. In the case that we try to instantiate an `Attraction` with a negative admission fee, default the admission fee to zero. This field should be **immutable**.
- `visitors` – A `Group[]` representing an array of visitors. `visitors` should always be initialized as an empty array of five `Groups`. All `Groups` in the `visitors` array must remain contiguous.

Constructors:

- A constructor that takes in the `name` and `admissionFee`. These values should conform to the specifications stated above.
- A constructor that takes in `name`. Initialize `admissionFee` to $5.25.

Methods:

- `admit()`
  - This method should properly implement `admit()` from `Admittable` interface.
  - Takes in a `String[]` corresponding to the name(s) of the new visitor(s) to be added.
  - Each visitor should first be added to a `Group` before being added to the `visitors` array.
  - `Groups` have a maximum capacity of size 5. If an array of names larger than five is admitted, `Groups` of five must be created until there are fewer than five names left. The remaining visitors should be placed in a `Group` together. These new `Groups` should be added to the `visitors` array.
  - Adds the `Groups` starting at the first empty element in the `visitors` array.
  - If the `visitors` array is full, resize the array to double the current size. Make sure the existing visitors are copied to the same indices in the resized array.
  - **Hint**: Helper methods would be great to use here! For instance, a helper method could be created to help resize the `visitors` array. Another helper method could be written to create a copy of a range of elements within an array. Making helper methods will make it easier to test your code.
  - **Note: Assume the input will be non-null and the names in the array will be valid (i.e. non-null, non-blank).**
- `rateAndExit()`
  - This method will remove a group of visitors from the attraction and update the attraction's rating based on their response.
  - Takes in an `int` corresponding to the index for a `Group` in the `visitors` array and an int corresponding to the rating the `Group` will give the attraction.
  - Update both `sumRatings` and `numRatings` based on the rating given by the `Group` who left.
    - The rating must be an int in the range [1, 10]. If a number less than 1 is given, then a rating of 1 is used. If a number greater than 10 is given, then a rating of 10 is used.
    - **Note: one group of visitors, regardless of how big the group is, can only yield one rating.**
  - Remove the `Group` from the attraction by removing the `Group` from the array and shifting the remaining `Groups` up (i.e. towards index 0) in the array so that all non-null values in the array are still contiguous (i.e. next to each other).
  - If the index passed in is invalid or no `Group` exists at that index, do not update the rating and print
    "`Could not update rating. Index invalid.`"
- `averageRating()`
  - This method should return a `double` with the average rating rounded to two decimal places (use `Math.round()`). Remember that `sumRatings` and `numRatings` are integer types, but the average rating should include decimals.
  - If the number of ratings is zero, then an average rating of zero should be returned.
- `printVisitors()`

  - This method should print out the attraction info and the visitor groups currently in the attraction.
  - First, the `toString()` for the `Attraction` should be printed on its own line.
  - The method should then go through the `visitors` array and print out all the `Groups` in the array, each on their own line.
    - Null values in the array should not be printed.
  - Print out the `Groups` in the following format:
    "`Group 1: <Group1>`
    `Group 2: <Group2>`
    `. . .`
    `Group 5: <Group5>`"
- `toString()`
  - This method should properly override `Object`'s `toString()` method.
  - **Note: Specifying `@Override` before the method header allows you to skip the Javadocs for that method and tells the compiler to double check that the method has been properly overridden.**
  - Should return the String:
    "`<name>/<averageRating>/<admissionFee>`"
- `compareTo()`
  - This method should properly override the `Comparable` interface's method of the same name.
  - Attractions should be compared on average rating first, followed by admission fee. Admission fee should only be compared if the ratings are the same.
    - If the **input** `Attraction` is null, return a **negative** result
    - If **this** `Attraction` and the input `Attraction` have the same average rating and admission fee, the result should be **0**.
    - If **this** `Attraction` has a higher rating than the input `Attraction`, then a **negative** result should be returned. Otherwise, a **positive** result should be returned (if the `Attractions` are not equal).
    - If the ratings are the same and this `Attraction` has a lower admission fee than the input `Attraction`, then a **negative** result should be returned. Otherwise, a **positive** result should be returned (if the `Attractions` are not equal).
- Getters and setters as necessary.

RollerCoaster.java

This class represents a roller coaster, a subclass of `Attraction`. Roller coasters have a maximum number of visitors that are allowed to enter.

Variables:

- `maxCapacity` - An `int` representing the capacity for the number of visitors the roller coaster can hold. This value cannot be less than 25. If we try to instantiate a `RollerCoaster` with a `maxCapacity` less than 25, we default to a value of 25. Once instantiated, this value should **not** be changed.
- `occupancy` - An `int` representing the current number of people at the roller coaster. This value should be initialized to 0.

Constructors:

- A constructor that takes in the `name`, `admissionFee`, and `maxCapacity` setting those values according to the above specifications.
- A constructor that takes in the `name`, with a default `admissionFee` of $5.25 and `maxCapacity` of 25.

Methods:

- `admit()`
  - This method should properly override the `admit()` method in `Attraction`.
  - Takes in a `String[]` of visitor(s) and adds it to the `visitors` array.
  - Like `Attraction`, visitors can only be added in `Groups` of five or less.
    - **Hint**: think about how you can reuse code to minimize the code written in this method!
  - `occupancy` should increase by the number of visitors added to `visitors`.
  - If the visitor(s) to be added would cause the `RollerCoaster` to go above its `maxCapacity`, reject all the visitor(s) and print
  - "`RollerCoaster has reached maximum capacity. Please visit another time!`"
- `rateAndExit()`
  - This method should properly override the `rateAndExit()` method in `Attraction`.
  - Like `Attraction`, remove the `Groups` from the `Attraction` and update variables accordingly.
  - `occupancy` should decrease by the number of visitors leaving the `Attraction`.
- `percentOccupancy()`
  - This method should return the percentage of occupancy, using the values of `occupancy` and `maxCapacity`.
  - Return a `double` rounded to two decimal places using `Math.round`.
    - For example, return the double "`12.34`" for the calculated value 12.341%.
- `toString()`
  - This method should properly override the parent class's method of the same name.
    - **Note: Specifying `@Override` before the method header allows you to skip the Javadocs for that method and tells the compiler to double check that the method has been properly overridden.**
  - Should return the `String` with occupancy rounded to two decimal places: "`RollerCoaster: <name>/<averageRating>/<admissionFee>/ <occupancy>%`"
  - **Hint**: Think about what keyword minimizes the code written in this method.

TripGuide.java

This class represents a trip guide that allows you to create and edit your trips. Use this as your driver class to test the functionality of all the code you've implemented!

Methods:

- `main()`
  - Create an `Attraction` array containing:
    - An `Attraction` with a name and admission fee of your choosing
    - An `Attraction` with a different name and same admission fee as the first
    - A `RollerCoaster` with a different name and same admission fee as the first

    - An `Attraction` with a different admission fee
    - A `RollerCoaster` with a different admission fee
    - Any other `Attractions` or `RollerCoaster` of your choosing
  - Create a couple of arrays with different lists of people of varying size (4 people, 5 people, 12 people, 25 people)
  - Use `admit()` to admit these groups of people to all `Attractions`.
    - Call `printVisitors()` after each admittance to observe the changes happening!
  - Call `rateAndExit()` on some `Attractions` so that:
    - Two `Attractions` remove the same `Groups` and thus have the same ratings,
    - Two `Attractions` remove different `Groups` and thus have different ratings,
    - One `Attraction` stays fully occupied and thus its rating remains 0.
  - Call `toString()` and `printVisitors()` for each `Attraction` in the array.
  - Use `compareTo()` on the `Attractions`. Consider scenarios where the rating is different, the rating is the same, but the admission fee is different, and both are the same.
  - Test any edge cases you think your code may run into!

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 25 points.** This means there is a maximum point deduction of 25. 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:

- `Admittable.java`
- `Group.java`
- `Attraction.java`
- `RollerCoaster.java`
- `TripGuide.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

To prevent trivialization of the assignment, you may not import any classes for this assignment.

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.