Homework 03 – GT Night at Six Flags

Authors: Ariel, Nicolas, Ricky, Daniel
Topics: abstract classes, overriding, equals method, toString

Problem Description

You have been hired by Six Flags to deliver the next big ride tracking system. The roller coasters and trolleys have been bought – your co-workers are just waiting for the code! And so, you set out to amusement parks to learn more about how you should implement the system.

Solution Description

You will create three classes: `Ride.java`, `RollerCoaster.java`, and `Trolley.java`. In this assignment, we use the following definitions:

- A "stop" refers to an individual destination along the ride.
  - For roller coasters, a "stop" is synonymous with a "run".
- A "run" consists of stop(s) and refers to an entire loop on the ride.
  - For roller coasters, a run consists of one stop.
  - For trolleys, a run may consist of multiple stops.

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 minimize code repetition and maximize code reuse!
4. Helper methods with appropriate visibility are optional.
5. Make sure to add Javadoc comments to your methods and classes!
6. **You must not write getters or setters for Ride.java, RollerCoaster.java, and Trolley.java. This assignment can be completed without the use of any getters or setters.**
7. **Constructors must deep copy any mutable arguments to the corresponding instance field.**

Ride.java

This class represents any amusement park ride. Ride should never be instantiated. Instead, it should only serve as a superclass for more specific rides.

Variables:

For fields in Ride, use the most restrictive visibility modifier that allows the fields to be accessible from any subclass.

- `String id` – the identifier of this ride
  - This variable should be **immutable** after construction of an instance.
- `double earnings` – the total amount of money this ride has earned, always initially 0
- `int runsSinceInspection` – the number of runs this ride has completed since the most recent inspection
- `String[] passengers` – the names of passengers aboard this ride.

Constructors:

- A constructor that takes in `id`, `runsSinceInspection`, and `passengers`.
  - **Note:** The `passengers` argument array must be **deep copied** to the corresponding instance field. This means that you should not simply copy the reference of the input array and assign it to the instance field.
- A constructor that takes in `id`, `passengers`, and sets `runsSinceInspection` to 0.
- Assume all constructor inputs are valid.

Methods:

- `canRun`
  - Takes in an `int` number of runs and returns a `boolean` that represents whether the ride can complete that number of runs without needing an inspection.
  - This method should **not** have an implementation in the Ride class.
    - ***Hint:*** *What keyword allows us to declare a method in a class without providing an implementation?*
- `inspectRide`
  - Takes in a `String[]` that represents different components of the ride.
  - Resets `runsSinceInspection` to 0 if the ride passes inspection and returns `true`.
  - Returns `false` if the ride does not pass inspection.
  - This method should **not** have an implementation in the Ride class.
    - ***Hint:*** *What keyword allows us to declare a method in a class without providing an implementation?*
- `costPerPassenger`
  - Takes in an `int` number of stops and returns a `double` representing the cost for the passenger to ride the specified number of stops.
  - This method should **not** have an implementation in the Ride class.
    - ***Hint:*** *What keyword allows us to declare a method in a class without providing an implementation?*
- `addPassengers`
  - Takes in an `int` number of stops and a `String[]` of passenger names.
  - Returns `true` if **all** new passengers can fit on the ride **and** the ride can travel the given number of stops without an inspection, and `false` otherwise.
  - This method should **not** have an implementation in the Ride class.
    - ***Hint:*** *What keyword allows us to declare a method in a class without providing an implementation?*
- `getPassengerList`
  - Returns a list of passengers as a `String` in the specified format.
    - Only one name should be on a line.
    - Only non-null elements should be listed.
    - No leading nor trailing whitespace on each line.
      - Assume that each name does not have trailing or leading whitespace.
  - This method will be *very* helpful when testing the addition and removal of passengers from the ride.

  - **Example:** For a passenger list of `[null, "Ariel", null, "Nicolas", "Ricky"]`, the output should be in the following format (without the angle brackets):

```
Passenger List for <id>:
Ariel
Nicolas
Ricky
```

  - If the ride has no passengers, the list will simply contain the first line in the sample output above (i.e., `"Passenger List for <id>:"`).
- `chargePassenger`
  - Takes in an `int` number of stops and increases `earnings` accordingly.
  - This method does not return anything.
  - This method should not be allowed to be overridden.
    - ***Hint:*** *What keyword prevents a method from being overridden?*
- `removePassenger`
  - Sometimes passengers violate rules or don't meet ride requirements.
  - Takes in a `String` representing the name of a passenger.
  - If the name is not in the passenger array, return `false`.
  - If the name is in the passenger array, remove the passenger by setting that index to `null` and return `true`.
  - **IMPORTANT:** This method should only remove the **first occurrence** of the named passenger, **case-insensitive**.
  - Assume that the input name is not null.
- `equals`
  - Overrides the `equals` method from `Object`.
  - Two Ride objects are equal if they have equal `id` and `runsSinceInspection`.
  - Your implementation of the `equals` method **must** be symmetric.
    - I.e., for two non-null objects `a` and `b`, `a.equals(b) == b.equals(a)` must always be true.
- `toString`
  - Overrides the `toString` method from `Object`.
  - Returns a `String` in the following format, without the angle brackets nor quotation marks, and without leading nor trailing whitespace:

```
"<id> has run <runsSinceInspection> times and has earned
$<earnings>."
```

    - Assume that `id` does not have leading whitespace.
    - Floating-point values should be displayed to two decimal places with rounding.
- **No getters or setters should be written, as they are not necessary to implement RollerCoaster and Trolley.**

RollerCoaster.java

This class represents a roller coaster in the amusement park. It must be a subclass of `Ride`.

Variables:

- `double rate` – the cost of one run
- `double photoFees` – the fees to purchase the mandatory photo package
  - Out of the generosity of our hearts, a passenger may ride the roller coaster multiple times and the fee must only be paid once.
- `int maxNumRuns` - the number of runs that the ride can complete before it must be inspected

Constructors:

- A constructor that takes in `id`, `runsSinceInspection`, `passengers`, `rate`, `photoFees`, and `maxNumRuns`.
- A constructor that takes in `id`, `runsSinceInspection`, and `maxNumRuns`, and defaults `passengers` to an empty String array of length 4, `rate` to 10, and `photoFees` to 15.
- A constructor that takes in `id` and defaults `runsSinceInspection` to 0, `passengers` to an empty String array of length 4, `rate` to 10, `photoFees` to 15, and `maxNumRuns` to 200.
- Assume all constructor inputs are valid.
- Remember that any mutable arguments should be deep copied.
- **Note:** Your use of constructor chaining **must** help you **minimize** code repetition!

Methods:

- `canRun`
  - Takes in an `int` number of runs and returns a `boolean` representing whether the roller coaster can complete that number of runs without needing an inspection.
  - Make sure to factor in the number of times the roller coaster has run already.
  - A ride cannot complete a negative number of runs.
- `inspectRide`
  - Takes in a `String[]` that represents different components of the ride.
    - Assume that the input array is not null and does not contain null elements.
  - The array must contain `"Tracks Clear"` and `"Brakes Ok"` for the roller coaster to pass inspection.
    - The input array may also contain other strings in any order.
    - String comparisons should be **case-insensitive**.
  - Returns `true` and resets `runsSinceInspection` to 0 if the roller coaster passes inspection.
  - Returns `false` if the roller coaster does not pass inspection.
- `costPerPassenger`
  - Takes in an `int` number of stops and returns a `double` representing the cost for the passenger to ride the specified number of stops.
    - ***Hint:*** *Use the descriptions for the `rate` and `photoFees` fields to determine the cost per passenger.*

- `addPassengers`
  - Takes in an `int` number of stops and a `String[]` representing the passengers in a group.
    - Assume that the input array is not null and does not contain null elements.
    - Assume that the input array contains at least one passenger.
  - The roller coaster should be able to travel the requested number of runs **without exceeding** `maxNumRuns`, or the passengers cannot be added.
  - For each passenger, add them once to the `passengers` array, starting from the lowest available index.
    - **IMPORTANT:** Only add the passengers if there are enough seats for **ALL** of the input passengers on the roller coaster **AND** if the roller coaster can complete the number of stops specified!
    - A seat is available if the corresponding index in the array does not already have a passenger assigned to it. What value is used to represent an empty seat?
    - Do not assume that passengers already on the roller coaster will always be at the front of the array; empty seats may be spread throughout the array.
      - E.g., the passenger array can be given to the constructor, and its state could be `[null, "Ariel", "Nicolas", null]`.
    - ***Hint**: It would be helpful to write a helper method to count the number of empty seats.*
    - **Note:** Duplicate passenger names are allowed.
  - If all passengers in the group fit on the roller coaster, **charge each passenger** for their ride.
    - Assume that each passenger is riding for the same number of stops.
    - ***Hint**: Reuse code! What method have we already created to charge a passenger?*
  - Increase `runsSinceInspection`. No matter how many passengers there are, `runsSinceInspection` should only be increased once per stop.
  - Returns `true` if the entire group of passengers can fit on the roller coaster and the roller coaster can travel the specified number of stops.
  - Returns `false` if any passenger in the group cannot fit on the roller coaster or the roller coaster cannot travel the given number of stops.
    - **Note:** A roller coaster cannot travel a negative number of stops.
- `equals`
  - Overrides the `equals` method from `Ride`.
  - Your implementation of the `equals` method **must** be symmetric.
  - Two RollerCoaster objects are equal if they have equal `id`, `runsSinceInspection`, `rate`, `photoFees`, and `maxNumRuns`.
- `toString`
  - Overrides the `toString` method from `Ride`.
  - Returns a `String` in the following format, without the angle brackets nor quotation marks, and without leading nor trailing whitespace:

```
"Roller Coaster <id> has run <runsSinceInspection> times
and has earned $<earnings>. It can only run
<maxNumRuns-runsSinceInspection> more times. It costs
$<rate> per ride and there is a one-time photo fee of
$<photoFees>."
```

    - Floating-point values should be displayed to two decimal places with rounding.
- **No getters or setters should be written.**
- **Reuse code to minimize code repetition when possible!**

Trolley.java

This class represents a trolley in the transport network. It must be a subclass of `Ride`.

Variables:

- `String[] stations` – the stations along this trolley's route
  - E.g., ["Midtown", "Atlantic Station", "Downtown", …]
- `int currentStation` – an index to an element in the `stations` array representing the general area of this trolley's current station

Constructors:

- A constructor that takes in `id`, `runsSinceInspection`, `stations`, and `currentStation`, and defaults `passengers` to an empty String array of length 20.
- A constructor that takes in `id`, `stations`, and `currentStation`, and defaults `runsSinceInspection` to 0 and `passengers` to a String array of length 20.
- Assume all constructor inputs are valid.
- Remember that any mutable arguments should be deep copied.
- **Note:** Your use of constructor chaining **must** help you **minimize** code repetition!

Methods:

- `canRun`
  - Takes in an `int` number of runs and returns a `boolean` representing whether the trolley can complete that number of runs.
  - Trolleys are well kept, so they are always ready to run, unless the number of runs is negative.
- `inspectRide`
  - Takes in a `String[]` that represents different components of the ride.
  - The array must contain "`Gas Tank Not Empty`" and "`Brakes Ok`" for the trolley to pass inspection.
    - The input array may also contain other strings in any order.
    - String comparisons should be **case-insensitive**.
  - Returns `true` and resets `runsSinceInspection` to 0 if the trolley passes inspection.
  - Returns `false` if the trolley does not pass inspection.
- `costPerPassenger`
  - Takes in an `int` number of stops and returns a `double` representing the cost for the passenger to travel the specified number of stops.
  - The cost is calculated by taking the number of stops multiplied by 3, and then dividing by the number of stations per loop. In other words, the more stops a trolley makes, the more it'll cost to ride!
    - ***Hint:*** *Be mindful of operand types and the implications of it!*
- `addPassengers`
  - Takes in an `int` number of stops and a `String[]` of passenger names.
    - Assume that the input array is not null and does not contain null elements.
    - Assume that the input array contains at least one passenger.
  - For each passenger, add them once to the `passengers` array, starting from the lowest available index.
    - Only add passenger if the trolley can travel the requested number of stops.
    - A seat is available if the corresponding index in the array does not already have a passenger assigned to it. What value is used to represent an empty seat?

        - Do not assume that passengers already on the trolley will always be at the front of the array; empty seats may be spread throughout the array.
            - E.g., the passenger array can be given to the constructor, and its state could be `[null, "Ariel", "Nicolas", null]`.
        - **Note:** Duplicate passenger names are allowed.
    - Not all passengers in the input array need to fit on the trolley. Add the passenger names from the input array to the `passengers` array in order, starting from the front of the array. If there are passengers that cannot fit on the trolley, do nothing with them.
    - Charge each passenger that boards the trolley for the number of stops they are traveling.
        - Assume that each passenger is riding for the same number of stops.
        - ***Hint**: Reuse code! What method have we already created to charge a passenger?*
    - Make sure that `runsSinceInspection` is increased correctly for the trip. Be careful -the number of stops for a trolley is not necessarily equal to the number of runs.
        - ***Hint**: Use `moveTrolley` to update `runsSinceInspection`!*
    - Returns `true` if the trolley can travel the given number of loops and `false` otherwise.
- `equals`
    - Overrides the `equals` method from `Ride`.
    - Your implementation of the `equals` method **must** be symmetric.
    - Two Trolley objects are equal if they have equal `id`, `runsSinceInspection`, `currentStation`, and `stations`.
        - Both trolleys must have the same station names (case-insensitive) in the same order for their `stations` array to be equal.
- `moveTrolley`
    - Takes in the number of stops representing the number of stations to advance.
    - Increments `runsSinceInspection` by the number of loops that will be completed after travelling the requested number of stops.
        - A loop is completed after leaving the last station and returning to the first.
    - Updates `currentStation` to the station where the trolley makes its final stop after moving the requested number of stops. Think carefully and test, test, test!
        - For example, suppose there are five stations along the trolley's route (stations numbered 0, 1, 2, 3, and 4). If we start at station 0 and travel three stops, the current station should now be 3 and no runs have been completed. If we start at station 3 and travel another three stops, the current station should now be 1 and one run has been completed.
    - This method does not return anything.
- `toString`
    - Overrides the `toString` method from `Ride`.
    - Returns a `String` in the following format, without the angle brackets nor quotation marks, and without leading nor trailing whitespace:

        "`Trolley <id> has driven <runsSinceInspection> loops and has earned $<earnings>. This trolley is at <name of the current station>. Next up is <name of the next station>.`"

        - Floating-point values should be displayed to two decimal places with rounding.
- **No getters or setters should be written.**
- **<u>Reuse code to minimize code repetition when possible!</u>**

THIS GRADED ASSESSMENT IS NOT FOR DISTRIBUTION.
ANY DUPLICATION OUTSIDE OF GEORGIA TECH'S LMS IS UNAUTHORIZED

Driver.java

This class will be used to test your code. You do not need to submit your Driver file. We have outlined some suggested tests in the `main` method section to help you get started.

Methods:

- `main`
  - Create 2 `RollerCoaster` objects.
  - Call `addPassengers()` on at least one of the `RollerCoaster` objects.
  - Call `toString()` on both `RollerCoaster` objects.
  - Check to see if the 2 `RollerCoaster` objects are equal.
  - Create 2 `Trolley` objects.
  - Call `addPassengers()` on at least one of the Trolley objects.
  - Call `toString()` on both `Trolley` objects.
  - Check to see if the 2 `Trolley` objects are equal.
  - These tests and the ones on Gradescope are by no means comprehensive, so be sure to create your own!

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 20 points.** This means there is a maximum point deduction of 20. 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:

- `Ride.java`
- `RollerCoaster.java`
- `Trolley.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.