ASSIGNMENT 1

CS 494: Principles of Concurrent Programming / Spring 2021

Description

In this assignment, you will build a management utility to track vaccine production and distribution between a laboratory and many clinics. Your submission should extend the following abstract class:

```
abstract class Lab {
  abstract Clinic createClinic(int capacity);

  abstract VaccineDose createVaccineDose(int id);

  abstract boolean addVaccineDoses(Clinic clinic, Set vaccineDoses);

  abstract boolean administerVaccineDoses(Clinic clinic, Set vaccineDoses);

  abstract boolean discardVaccineDoses(Clinic clinic, Set vaccineDoses);

  abstract boolean moveVaccineDoses(Clinic from, Clinic to, Set movedDoses);

  abstract Set getVaccineDoses();

  abstract Set getVaccineDoses(Clinic clinic);

  abstract List<Action<Clinic>> audit(VaccineDose dose);

  abstract List<Action<VaccineDose>> audit(Clinic clinic);
}
```

Each operation (method) behaves as follows:

- **createClinic**: Creates a clinic that can store and administer vaccine doses. Due to cold storage requirements, each clinic has a limit in the number of doses it can store – `capacity`
- **createVaccineDose**: Creates one vaccine dose with a given `id`. The `id` is unique across the same `Lab`.
- **addVaccineDoses**: Adds previously created `vaccine doses` to the provided `clinic`.
    - This operation either adds all the doses, if the clinic has enough capacity, or none.
    - For instance, attempting to add two doses to a clinic that only has room for one should not change the contents of the clinic.

    - If all the doses are added to the clinic, this operation returns `true`. If the clinic remains unchanged, this operation returns `false`.
- **administerVaccineDoses**: Administers the `vaccine doses` provided, which should be present on the given `clinic`.
    - Similarly to **addVaccineDoses**, this operation either administers all the vaccine doses or none.
    - Trying to administer a dose that is not in the current clinic results in failure of the whole operation.
    - If all the doses are administered, this operation returns `true`. If any dose cannot be administered, then no doses are administered and this operation returns `false`.
- **discardVaccineDoses**: Doses have a limited lifetime and must be discarded once expired. This operation discards the given `vaccine doses` on the provided `clinic`.
    - This operation behaves like **administerVaccineDoses**, in that either all the doses are discarded or no dose is changed.
    - Discarding doses on a clinic that does not have them results in the whole operation failing, and returning `false`. If the operation is successful, it returns `true`.
- **moveVaccineDoses**: Moves `vaccine doses` current present in the `from` clinic to the `to` clinic.
    - If there is not enough room in the `to` clinic, this operation should fail and return `false`.
    - If any dose is not present in the `from` clinic, this operation should fail and return `false`.
    - This operation returns `true` if it succeeds in moving all the doses between clinics.
- **getVaccines**: Gets the vaccines that are <u>ready to be administered</u> (i.e., added to a clinic, and not discarded or administered).
    - Without arguments, this operation lists all the vaccines that the lab produced that are ready to be administered.
    - With a `clinic` argument, this operation lists all the vaccines currently in that clinic that are ready to be administered.
- **audit**: Returns an audit log that tracks vaccine doses and clinic contents.
    - With a `vaccineDose` argument, returns a list of all the clinics in which the dose was
        - The order of the list matters
        - When moving, vaccine doses should be removed from one clinic before being added to another clinic
    - With a `clinic` argument, returns a list of all the vaccine doses that passed by that clinic
        - The order of the list matters
            - If an operation changes many doses at once, the order between those doses does not matter.
            - However, all those doses should be on the list after preceding operations and before later operations

Besides the `Lab` interface, your solution should also implement the `VaccineDose` interface for each individual dose:

```
interface VaccineDose {
    enum Status { READY, USED, DISCARDED }
    Status getStatus();
}
```

Each vaccine dose should behave as follows:

- All doses are created as `READY`
- A `READY` dose can be administered, in which case it becomes `USED`; or discarded, in which case it becomes `DISCARDED`
- Once a dose is used or discarded, it cannot become `READY` again or be used/discarded again

Correctness Requirements

Your implementation should keep the following properties at all times:

1. **getVaccineDoses** operations never list more doses than a clinic's capacity
2. **getVaccineDoses** operations never list more items for the whole lab than the sum of the capacity of all the clinics.
3. Adding vaccines to a clinic successfully results in those doses being listed in later **getVaccineDoses** operations.
4. Using or discarding doses from a clinic successfully results in those doses not being listed in later **getVaccineDoses** operations.
5. Each vaccine is listed in one clinic at most by **getVaccineContents** operations.
6. Doses are never "in-transit" due to move operations (i.e., **getVaccineContents** operations not listing doses removed from the `from` clinic and still not added to the `to` clinic).
7. Once the status of a dose is observed to be `USED` or `DISCARDED`, it cannot be observed to be anything else from that point on.
8. The current contents of any clinic can be explained by following the entries in the audit log, by the order in which they appear in the log.
9. The current state and location of any dose can be explained by following the entries in the audit log, by the order in which they appear in the log.

Concurrency Requirements

Your submissions should be **thread-safe**. That is, if multiple threads call any combination of methods in any order, none of the correctness properties must be violated. To that end, your implementation can limit the concurrency inside as much as needed.

Entry Point

You should create a new class, on a new file, where you will implement your solution. You should change method Lab.createLab so that it creates an instance of the class you added. You cannot change any other part of the code that is provided to you.

```
abstract class Lab {
    static Lab createLab() {
        throw new Error("Not implemented");
    }
}
```

Due Date and Resubmission Policy

This assignment is due on **February 6 2021** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

To resubmit this assignment, your **original grade** (as defined by the autograder) should be **equal to or higher than 30%**. You can resubmit your assignment until **February 13 2021** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **10% bonus points**, which you can earn by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment1` label, and non-anonymous to the instructors to count towards the bonus.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 70%. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 7 tests, that will check if your submission outputs the expected result. Each test is worth 10%.

Together with the code, you should submit a video screen-cast (**through Gradescope**) that answers the three questions below by explaining how your code works. The questions focus on concurrency/multi-threading and are worth 10% each. You can record such a video without installing any software by using the following website: https://screenapp.io/#/

1. How does your implementation ensure Properties 1 and 2 in the presence of concurrent operations?
2. How does your implementation ensure Properties 5 and 6 in the presence of concurrent operations?
3. How does your implementation ensure Property 7 in the presence of concurrent operations?

The maximum length for the video is 5 minutes, instructors will stop watching at the 5 minute mark (nothing past that point in the video will be graded). This video should be a screencast of your IDE open on the code submitted, and you should highlight the code. Note that longer videos are not better videos, and you should record a video as short as needed to show all the expressions and answer the questions above.

The final grade for the assignment will be the grade of the original submission, for assignments without a resubmission; or the average between the original grade and the resubmission grade, for assignments with a resubmission. The grade of the original submission includes any bonus points.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `System.out.println("expected result").`

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.