Assignment 1

CS 494 - Principles of Concurrent Programming - S20

Description

In this assignment, you will build a warehouse management utility. Your submission should implement the following interface:

```
public interface Warehouse <S extends Shelf, I extends Item> {
    public S createShelf(int size);
    public I createItem(String description);
    public boolean addItems(S s, Set<I> items);
    public boolean removeItems(S s, Set<I> items);
    public boolean moveItems(S from, S to, Set<I> items);
    public Set<I> getContents();
    public Set<I> getContents(S s);
    public List<Action<S>> audit(I i);
    public List<Action<I>> audit(S s);
}
```

Each operation (method) behaves as follows:

- **createShelf**: Creates a shelf that holds `size` items, and adds it to the warehouse.
- **createItem**: Creates a unique item that can be put on a shelf.
- **addItems**: Adds a set of `items` to the shelf `s`.
    - This operation either adds all the items, if the shelf has enough room for all the items, or none.
    - For instance, attempting to add two items to a shelf that only has room for one should not change the contents of the shelf.
    - If all the items are added, this operation returns `true`. If the shelf remains unchanged, this operation returns `false`.
- **removeItems**: Removes a set of `items` from the shelf `s`.
    - This operation behaves as **addItems**: Either all items are removed, or the shelf remains unchanged. The return is also similar to **addItems**.
    - Trying to remove an item that is not on the target shelf results in failure of that remove operation.
- **moveItems**: Moves a set of `items` from a shelf `from` to another shelf `to`.
    - This operation behaves as **addItems**: Either all items are moved, or both shelves remain unchanged. The return is also similar to **addItems**.
    - Trying to move items that are not on the shelf `from` results in failure of that move operation.

- **getContents**: Gets the items in the warehouse or on a shelf `s`.
    - Without arguments, this operation gets all the items that are inside the warehouse
    - With a shelf argument `s`, this operation gets all the items that are held by that shelf `s`.
- **audit**: Returns an audit log that tracks items or shelves.
    - With an item argument, returns a list of all the shelves in which that item was.
        - Note that the order of the list matters
        - Items should be added to a shelf before being removed from that shelf
    - With a shelf argument, returns a list of all the items that were in that shelf, and the order in which those items were added/removed.
        - If a add/remove/move operation moves many items at once, the order among those items is unspecified.
        - However, all those items should be on the list after preceding operations and before later operations.

You can assume that all operations **createShelf** precede the first **addItems** operation.

Your implementation should keep the following properties at all times:

1. **getContents** operations never list more items for a shelf `s` than `s`'s capacity.
2. **getContents** operations never list more items for a warehouse than the sum of the sizes of all the shelves.
3. Adding items to a shelf successfully should result in those items being listed in later **getContents** operations until those items are removed.
4. Removing items from a shelf successfully should result in those items not being listed in later **getContents** operations.
5. Each item is listed in one shelf at most by **getContents** operations.
6. Items are never "in transit" due to move operations (i.e., **getContents** operations not listing items removed from the `from` shelf and still not added to the `to` shelf).
7. The current contents of any shelf can be explained by following the entries in the audit log, by the order in which they are on 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 warehouse properties must be violated. To that end, your implementation can limit the concurrency inside the warehouse as much as needed.

Entry Point

You should change method Warehouse.createWarehouse so that it creates an instance of your solution. You should not change any other part of the code that is provided to you.

```
public interface Warehouse <S extends Shelf, I extends Item> {
    public static Warehouse createWarehouse() {
        throw new Error("Not implemented");
    }
}
```

Due Date and Late Policy

- This assignment is due on **February 8** (Saturday) by **5pm CST**.
- Submissions delivered by **February 9** (Sunday) by **5pm CST** will have a 10% penalty.
- Submissions delivered by **February 10** (Monday) by **5pm CST** will have a 25% penalty.
- **No submission will be accepted after February 10 past 5pm CST (Monday)**

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

Submission and Grading

This assignment should be submitted through Github, and has an automatic grade component of **80%**. You can check your current grade at any point by submitting your code and checking Travis. The automatic grade is determined by 8 tests, that will check if your project outputs the expected result. Each test is worth 10%.

This assignment also has a question component, which should be completed on **Blackboard**. There are two questions about your implementation, focusing on concurrency/multi-threading, each worth 10%:

1. How does your implementation ensure Properties 1 and 2 with regards to concurrent **add** and **move** operations?
2. How does your implementation ensure Properties 5 and 6 with regards to concurrent **move** and **remove** operations?

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 tagged with the **Assignment 1** label and non-anonymous to the instructors to count towards the bonus.

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.