CmpE 160 - Introduction to Object Oriented Programming

Project #2 - Cargo Delivery Simulation
Due Date: 01.05.2019 23:55 PM

1. Introduction

In this project you are going to implement a cargo delivery simulation. You are asked to use several data structures together to simulate the delivery of cargo packages between a number of stations. The task, concepts and the classes you are expected to implement are explained in detail in the following sections.

1.1 Concepts

There are 4 concepts to be simulated in this scenario:

1. Cargo package
2. Carriage
3. Train
4. Train station

1.1.1 Cargo Package:

A cargo package is the item to be transported. Normally, each is kept in some station and is requested in some other station. Given that none would be larger than the size of a carriage, cargo packages can be of different sizes.

1.1.2 Carriage:

Cargo packages are carried within carriages. Each carriage has the same capacity throughout an individual run of your program. This means, once you read in the capacity of the carriage from the input file, you can fix it for the remainder of that run. Another run with a different input file may require you to use a different value for carriage capacity, of course.

The important thing about the carriage concept is that they are loaded following the last in first out (LIFO) principle. The first cargo package that is loaded into the carriage comes at the end, during the unloading.

1.1.3 Train:

A train consists of a number of carriages that are connected to each other, in some order. This order is preserved using a linked list structure. The objective of a train is to carry the connected carriages from one station to another.

1.1.4 Train Station:

Either the source or the destination of a train. A station processes the unloaded packages in a first in first out (FIFO) manner, using a queue.

1.2 Task

The main requirement of this project is to deliver each cargo package in each train to the right train station. Of course, while delivering these packages, there are certain rules which must be taken into consideration.

The scenario is as follows: a train initially starts loading the cargo packages at *station 0*. According to the input, it fills its carriages with the cargo packages. After the loading ends, it moves to the next station. Unloads the packages that are sent to this station and loads back the others, and this process resumes until the entire task finishes.

When the train comes into a station, it must unload all of the cargo packages into the queue of the train station, starting from the first carriage, which is the head of the list. The process must continue until the tail of the train. In other words, the first carriage in the train must be unloaded firstly, the second carriage secondly, and so on.

The unloading of carriages must follow LIFO principles. On the other hand, all of the unloaded packages must be enqueued in the package queue of the station so that they could be loaded back to the train, following the FIFO principles. Note that, this is the point that you are expected to select the packages sent to that station and push only the other packages to the carriages.

2. Class Diagram and Implementation Details

You are provided with 5 empty class files.
You have to implement `Train, Carriage, Station, Cargo` and `Main` classes.
`Main` class takes two arguments: input and output files.
Station index starts with 0.

Train.java:

`Train.java` represents the train objects. It holds the carriage capacity in `carCapacity`, the length of the train as an integer in `length`, and the head and the tail carriages of the train in `head` and `tail` respectively.

This class has the constructor with the signature `Train(int length, int carCapacity)` and the methods with the following sigantures `load(Queue<Cargo> cargos)` and `unload(Queue<Cargo> cargos)`.

`Train(int length, int carCapacity)`:
Takes the length of the train (number of total carriages) and the capacity of each carriage in the train as integers and initializes the necessary fields.

`load(Queue<Cargo> cargos)`:
This method will load all of the cargos into the train starting from the first carriage. Loading cargo packages into the carriages follows a special procedure. Every time you want to push a package into a carriage, you have to consider the size of the cargo and the space left in the carriage. That is, for instance, if the carriage you are loading has 10 units of space left at some point and next package to be loaded is bigger than 10 units, you should try the next carriage. You should check each carriage starting from the first one at each step of loading, since carriages could have empty space. If all the carriages in the train are full, add necessary amount of carriages to the train. Look at the following example:

```
carCapacity: 100
Sizes of the cargo packages to be loaded: 40, 40, 60, 50, 10, 50
carriages to be formed:
   1. c0: 40, 40, 10
   2. c1: 60
   3. c2: 50, 50
```

After the loading procedure at some station, if some of the carriages are empty, you should leave those empty carriages at the current station before leaving (by removing them from the train).

`unload(Queue<Cargo> cargos)`:
Train will unload all of its cargo to cargos starting with the first carriage until the train is completely empty.

Carriage.java:

`Carriage.java` represents the carriages of the train. Each carriage of the same train has the same capacity. Each carriage object will contain an integer capacity field as `emptySlot` and a stack of cargos as `cargos`. Also, there will be pointers for the next and previous carriages as `next` and `prev`.

This class will contain the methods (and the constructor) below:

`Carriage(int capacity)`:
Initializes the necessary fields.

`isFull()`:
Checks if the carriage is full, returns true if it is full and false otherwise.

`push(Cargo cargo)`:
Pushes a cargo into the cargo stack of the carriage.

`Cargo pop()`:
Pops a cargo from the cargo stack of the carriage.

Station.java:

`Station.java` holds `id` as the station id, `cargoQueue` as the queue of cargos and a print stream object called `printStream` to give the necessary output.
This class also contains the `process(Train)` method.

`process(Train train)`:
Handles the events from the train's arrival to the leaving of the train.This method also takes care of all the output operations. In this method train unloads into `cargoQueue` and prints the cargos that have reached their destination without changing the order in the `cargoQueue`. Then it loads rest of the queue into the train also without changing the order. Lastly prints the length of the train.

Cargo.java:

`Cargo.java` has mainly 3 fields. These fields are `id`, `loadingStation`, `size` and `targetStation`.This class also overrides `toString()` method to print the output respective to the fields.

Main.java:

`Main.java` consists of two methods:

First, write a method named `readAndInitialize`, reading the inputs and initializing necessary classes and fields, and also placing cargos to their initial station.
Second, write a method named `execute` that will start the train from the first station, it will move the train and perform the necessary operations until train leaves the last station.

INPUT FORMAT:

First line of the input consists of three integer values side by side (with spaces in between):

1. Initial number of carriages the train has
2. The size capacity of each carriage (all carriages will have equal size capacities)
3. The number of stations the train will stop.

Rest of the lines have four integers for each line, side by side (with spaces in between):

1. Id of the cargo itself
2. Id of the station the cargo is waiting
3. Id of the target station.
4. Size of the cargo

Example input file:

10 100 2
0 0 1 20
1 0 1 60

According to the first line, the train has 10 carriages in the beginning, each carriage has 100 cargo size capacity, and the train will stop at 2 stations.

Second and third lines are different cargo packages:

- ID of the first cargo package is 0, the station ID that the cargo will be loaded is 0,the ID of the target station that the cargo will be unloaded is 1, and the size of the cargo is 20 units.
- ID of the second cargo package is 1, the station ID that the cargo will be loaded is 0, the ID of the target station that the cargo will be unloaded is 1, and the size of the cargo is 60 units.

Therefore, in this scenario, there are two stations and two cargos and both cargos are loaded into the train from station 0 and both are unloaded into station 1.

OUTPUT FORMAT:

When the train stops on each station, there are 5 integers to be printed for each cargo package during unloading and loading: (side by side, with spaces in between, each operation will be printed in its own line):

1. Id of the cargo
2. Id of cargo's initial station
3. Id of cargo's target station
4. Size of the cargo
5. 0 for unloading and 1 for loading
6. Number of carriages the train has after unloading or loading

Each cargo will be printed respective to the order of unloading and loading.

Example output file:

(Comments are written here to explain the meaning of the output numbers and they should not be included in your output file)

```
0 0 1 1 10      //cargo 0, which is going from station 0 to 1 is loaded and there are 10 carriages
1 0 1 1 10      //cargo 1, which is going from station 0 to 1 is loaded and there are 10 carriages
1 0 1 0 10      //cargo 1, which is going from station 0 to 1 is unloaded and there are 10 carriages
0 0 1 0 10      //cargo 0, which is going from station 0 to 1 is unloaded and there are 10 carriages
```

3. Some Remarks

- Do not use the **LinkedList** class of Java explicitly, you are expected to write the necessary linked list operations yourself.
- Please keep in mind that providing the necessary accessibility and visibility is important: you should not implement everything as public, even though the necessary functionality is implemented. The usage of appropriate access modifiers and other Java keywords (super, final, static etc.) play an important role in this project since there will be a partial credit, specifically for the software design.
- Please do not make any assumptions about the content or size of the scenarios defined by the input test files. Your project will be tested through different scenarios, so you need to consider the all the possible criteria and implement the code accordingly.