Project 2: Ships and Ports

CMPE 160: Introduction to Object-Oriented Programming, Spring 2021

Instructor: Tuna Tuğcu
TAs: Ersin Başaran, Yiğit Yıldırım
Contacts: nurlandadashov2002@yahoo.com

Due: **June 12, 2021 23.59**

1. Description

Hi, there. Welcome to the second project where you will be learning the fundamentals of OOP. In this project, you are required to implement a simulation of a port management system. The main objective is to analyze the requests provided in input and carry out the necessary actions.

There are ports, and ships are sailing between them. Ships are carrying the following types of containers: basic, heavy, refrigerated, liquid, and each of them should be handled differently. Containers in a port can be loaded to ships, and, conversely, they can be unloaded from a ship to port. Ships need a certain amount of fuel to sail from one port to another.

Note that there is no user interaction during the execution, the program parses the input file composed of the sequential operations.

Also, as a significant remark, the signature of the methods and the field names that you are going to implement should be identical to the ones that are specified in this document. If a specific method signature is not enforced by the project description, you can feel free to implement it in your way (i.e. you can implement extra methods considering your own design choice). However, you should also consider the proper usage of access modifiers for preserving the desired visibility and accessibility throughout the project. In other words, you should not define everything as “public”, which results in a penalty if done so.

2. Class Hierarchy and Details

You are already provided with the following interfaces:

- IShip.java
- IPort.java

Do not modify the code in these files! You are responsible for all the compilation errors that originated from the changes made in any of these classes including the addition or removal of libraries.

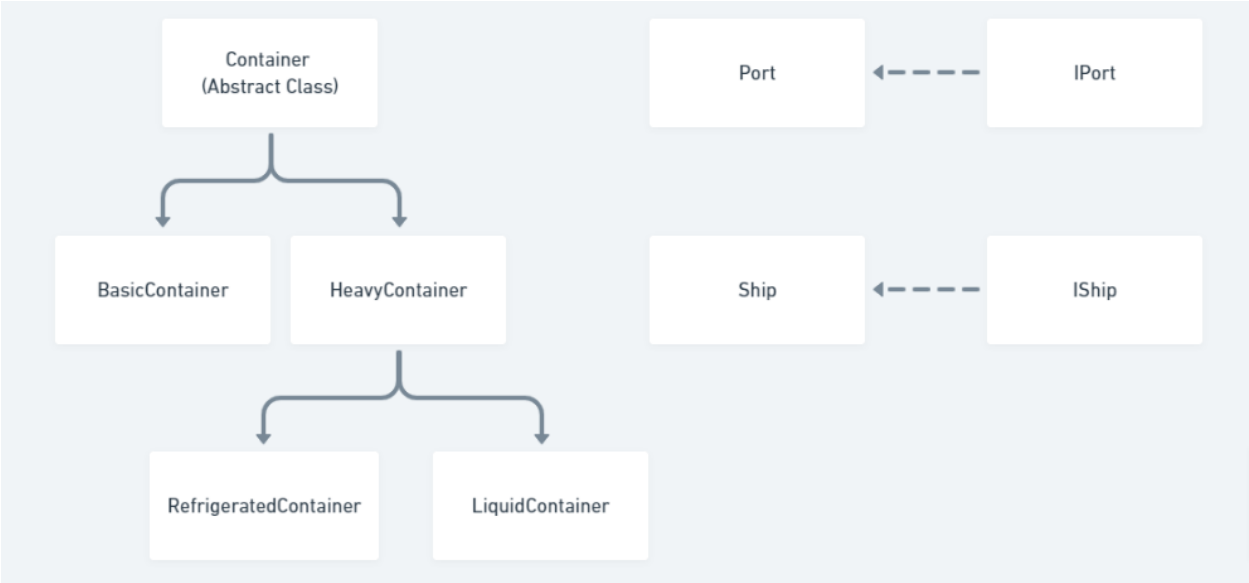
In addition, you are given a part of the Main class. You are required to complete the Main class with the instructions given later in this document. The other files are just empty. Using the fundamentals of the object-oriented programming methodology and the inheritance principles, you are expected to fill in these files in a suitable manner to complete the project. Do not forget that you are free to create new packages, classes, or interfaces. However, the class names and properties that are given below should be exactly the same in your projects. Please note that, if you fail to follow this instruction, you do not get any points as your projects will be automatically graded.

Before CMPE 160, there was a single source file and all the functions were implemented in that file. However, in this project, there will be multiple classes that interact with each other during execution. Therefore, before starting the implementation, you are advised to allocate some time to understand the main logic behind the design and the relation among the code pieces.

Within the scope of this project, you are expected to implement these classes (which will be explained in detail in the rest of this document):

1. Main.java
2. Port.java
3. Ship.java
4. Container.java
5. BasicContainer.java
6. HeavyContainer.java
7. RefrigeratedContainer.java
8. LiquidContainer.java

The class hierarchy that you must consider during the implementation is illustrated in the class diagram below:



Main.java

The main method simply reads an input file that is composed of sequential commands related to the operations. You can use this class to test your code. Scanners for input and output files and the array lists defined below are given for you.

You are required to read from an input file and print to an output file, names of which are passed to your program as arguments. Apart from unit tests to test your code and class structure, we will use the Main class to test your code for various inputs and outputs.

Input Format

In the first line, there is a number **N**, which represents the total number of events that occur during the simulation.

In the following **N** lines, the actions are specified in their customized format as described below. In other words, each distinct line represents an event with a specific action in the port management system. Please note that the test cases will not cover any erroneous input with regards to the format.

The possible actions are encoded through separate numbers:

1. Creating a container
2. Creating a ship
3. Creating a port
4. Loading a container to a ship
5. Unloading a container from a ship
6. Ship sails to another port
7. Ship is refueled

1. When creating a container, three or four inputs are given depending on the type of container. The first input represents the id of the port where that container should be placed initially. Note that there *will not be* a test case where a port of that id has not been created yet. The next input is a *non-zero positive* integer, denoting the weight of that container. Depending on the weight you need to decide the type of container:
   - **Weight <= 3000** : **BasicContainer**
   - **Otherwise: HeavyContainer**

   The special type of *heavy container* is given as **“R”** or **“L”**, which stand for Refrigerated and Liquid, respectively. There may or may not be an “R” or “L” after weight. *It is important to note that Refrigerated and Liquid Containers are special types of HeavyContainer, meaning that input lines that contain “R” or “L” should always create the respective type of container regardless of the weight*. The ID of a container is determined by the simulation and should be unique to each container. *In other words, no two containers can have the same ID, even if one of them is a basic one and the other a heavy/refrigerated/liquid one*. Successive numbers should be assigned as IDs according to the

order of adding them to the simulation by starting indexing from 0, such that the first container must have the ID 0 and the second 1 and so on.

Example:

1 0 500 means that a 500kg BasicContainer(500<=3000) should be placed at the 0th port

1 1 3500 R means that a 3500kg RefrigeratedContainer(3500>3000) should be placed at 1st port

2. When creating a ship six *positive integers* are given:
    - The ID of the port where the ship initially is
    - Maximum weight of all containers in that ship *(nonzero)*
    - Maximum number of all containers in that ship *(counting all types of containers) (maybe zero)*
    - Maximum number of heavy containers in that ship *(counting refrigerated and liquid containers) (maybe zero)*
    - Maximum number of refrigerated containers in that ship *(maybe zero)*
    - Maximum number of liquid containers in that ship *(maybe zero)*

    A nonzero positive double is the 7th input denoting the fuel consumption per km of that ship.

    Example:

    2 0 10000 10 5 3 4 50.00 means that a ship should be created at the 0th port that can hold at most 10 containers (regardless of type), 5 heavy, 3 refrigerated, and 4 liquid containers. This ship also spends 50.00 fuel per km while it travels from one port to another.

    Note that *there will not be* a test case like this: 2 0 10000 10 0 3 4 because if there can be no heavy containers, there cannot be any refrigerated/liquid ones either.

    2 0 10000 10 0 0 0 30.00 means that this ship may contain only BasicContainers (at most 10). This ship also spends 30.00 fuel per km while it travels from one port to another.
3. When creating a port, two *double* inputs are given, which are the x and y coordinates of the port. The IDs of the ports are also determined by the simulation.

    Example:

    3 40.00 50.00 means that a port should be created with the coordinates (40.00, 50.00)

4. When loading a container into a ship, IDs of ship and container are given. A container with that ID will always exist but may or may not be in the port where the ship currently is. Ship with the given ID will always exist. If the container is currently in the port, it may or may not be loaded into the ship depending on the restrictions of that ship.
   Example:
   The ship with ID 0 was created with the following line
   2 0 14000 10 5 4, and currently has 2 BasicContainers, 3 HeavyContainers, 2 RefrigeratedContainers, 0 LiquidContainers.
   The sum of all containers above is 11500 kg.
   4 0 1 means that container 1 should be loaded into the ship 0.
   Consider the following cases about container 1:
   - It is a HeavyContainer: Loading is not possible, as there are already 5 heavy containers (3 + 2 + 0).
   - It is a RefrigeratedContainer: Loading is not possible, as there are already 5 heavy containers (3 + 2 + 0).
   - It is a LiquidContainer: Loading is not possible, as there are already 5 heavy containers (3 + 2 + 0).
   - It is a BasicContainer: Loading possible if weight is *less than or equal to 2500 (14000 - 11500 = 2500)*
5. When unloading a container from a ship, IDs of ship and container are given. A container with that ID will always exist but may or may not be on the ship. Ship with the given ID will always exist. If a container with the given ID exists in a ship, it will be placed into the storage of the port where the ship currently is.
   Example:
   Ship 0 is in port 3 and contains containers 1,2,3.
   5 0 1 means that container 1 should be unloaded from ship 0. In this case, this is possible, as container 1 exists in ship 0. Thus, container 1 will now be located in the storage of port 3.
6. When ships travel from one port to another, IDs of the ship and destination port are given. Fuel consumption of the ship consists of the value that was given in the creation stage and consumption of containers. We will provide more information about how fuel consumption is calculated in the following sections. In short, if a ship has enough fuel it will sail to the destination port.
   Example:
   6 0 1 means that ship 0 should travel to port 1 if it has enough fuel.

7. When fuel is added to a ship, the ID of the ship and amount of fuel is given. The fuel amount is a *nonzero positive double*.
   Example:
   7 0 30.20 means 30.2 amount of fuel should be added to existing fuel of ship 0.

Output Format

You must print the ships in the ports list in the main class with their attributes. Please note that the double values should have 2 digits after the fraction point. The output format should be ordered in ascending order of the port IDs, while in each port, the ships should be also ordered according to their IDs. A port should be printed as “Port ID: (x, y)”, followed by IDs of containers located in the port: “{BasicContainer, HeavyContainer, RefrigeratedContainer, LiquidContainer}: [IDLIST]”. You should also list all ships located in that port as “Ship ID: FUEL_LEFT”. Additionally, containers in each ship should be listed too. Please, refer to example output for a better understanding of output format. Note that port and ship contents are indented with 2 whitespaces. Order of containers types matter and should be as follows: Basic, Heavy, Refrigerated, Liquid.

Example Output:

```
Port 0: (100.00, 200.00)
  BasicContainer: 13 14
  HeavyContainer: 12
  LiquidContainer: 23
  Ship 1: 370.74
    BasicContainer: 1 2
    HeavyContainer: 0
    RefrigeratedContainer: 3
    LiquidContainer: 4
  Ship 2: 150.54
    BasicContainer: 5 6 7
    HeavyContainer: 8 9
    LiquidContainer: 10 11
Port 1: (120.60, 250.00)
  BasicContainer: 15 16 17 18
  LiquidContainer: 20 21
  Ship 0: 260.00
    BasicContainer: 19
    RefrigeratedContainer: 22
```

I have replaced whitespaces with # to illustrate them. The correct output obviously should not contain #s.

```
Port 0: (100.00, 200.00)
##BasicContainer: 13 14
##HeavyContainer: 12
##LiquidContainer: 23
##Ship 1: 370.74
####BasicContainer: 1 2
####HeavyContainer: 0
####RefrigeratedContainer: 3
####LiquidContainer: 4
##Ship 2: 150.54
####BasicContainer: 5 6 7
####HeavyContainer: 8 9
####LiquidContainer: 10 11
Port 1: (120.60, 250.00)
##BasicContainer: 15 16 17 18
##LiquidContainer: 20 21
##Ship 0: 260.00
####BasicContainer: 19
####RefrigeratedContainer: 22
```

Port.java

Port class should have the following variables, exactly named as below:

- int ID
- double X
- double Y
- ArrayList<Container> containers
- ArrayList<Ship> history : keeps track of every ship that has visited
- ArrayList<Ship> current : keeps track of the ships currently here

Port must implement the IPort interface and the methods it requires. The class should have the following methods, exactly as named below:

- A constructor with three parameters, ID, X, and Y.
- A method that calculates the distance between the object itself and another Port, double getDistance(Port other)

Additionally, the choice of defining the variables as private, protected, or public may require additional getter and setter methods. This applies to all other classes as well, unless specified otherwise.

Ship.java

Ship class should have the following variables, exactly named as below:

- int ID
- double fuel
- Port currentPort

Ship must implement the IShip interface and the methods it requires. The class should have the following methods, exactly as named below:

- public Ship(int ID, Port p, int totalWeightCapacity, int maxNumberOfAllContainers, int maxNumberOfHeavyContainers, int maxNumberOfRefrigeratedContainers, int maxNumberOfLiquidContainers, double fuelConsumptionPerKM)
- ArrayList<Container> getCurrentContainers() : should return the list of all containers currently in the ship *sorted by ID*.

Container.java

Container is an abstract class and should have the following fields:

- int ID
- int weight

It should have the following methods:

- A constructor with parameters ID, weight
- double consumption() : should return fuel consumption required by the container
- boolean equals(Container other) : check type, ID and weight of a container. If they are the same, return true, otherwise return false.

BasicContainer.java and HeavyContainer.java

They should extend the Container class. They should have a constructor with two inputs like a Container.

- Weight of a BasicContainer <= 3000
- Weight of a HeavyContainer : otherwise

Fuel consumption is as follows:

- BasicContainer : 2.50 per unit of weight
- HeavyContainer : 3.00 per unit of weight

RefrigeratedContainer.java and LiquidContainer.java

They are special types of HeavyContainer and should extend the HeavyContainer class. They should have a constructor with two inputs like a HeavyContainer.

Fuel consumption is as follows:

- RefrigeratedContainer : 5.00 per unit of weight
- LiquidContainer : 4.00 per unit of weight

3. Interface Details

IPort.java

You are already provided with this interface. Do not modify the code in this file! It contains the following method:

- void incomingShip(Ship s) : should add this ship to *current* ArrayList.
- void outgoingShip(Ship s) : should add this ship to *history* ArrayList.

Note that there *should not be any duplicates* in the *current* and *history* ArrayLists *(i.e. do not add the same ship to history if it has already visited that port before)*

IShip.java

You are already provided with this interface. Do not modify the code in this file! It contains the following method:

- `boolean sailTo(Port p) : returns true if a ship successfully sailed to the destination port`
- `void reFuel(double newFuel) : adds fuel to a ship`
- `boolean load(Container cont) : returns true if a container was successfully loaded to a ship`
- `boolean unLoad(Container cont) : returns true if a container was successfully unloaded from a ship`

3. Some Remarks

- ➔ The method signatures and the field names should be exactly the same as the ones that are specified by this document. However, you can implement additional methods as you desire.
- ➔ 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. For example, it should not be possible to change IDs of objects, X and Y coordinates of Ports, etc. The usage of appropriate access modifiers and other Java keywords (super, final, static, etc.) play an important role in this project.
- ➔ There will also be a partial credit for the code documentation. You need to document your code in Javadoc style including the class implementations, method definitions (including the parameters, return if available, etc.), and field declarations. You do not need to create and submit a documentation file generated by Javadoc as the software documentation.
- ➔ 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 all the possible criteria and implement the code accordingly.
- ➔ Even though the input format is definite and no erroneous format will be utilized for testing, you should consider the necessary validity checks for the actions. For example, the input file may include an action for a ship, which does not have container 1, to unload container 1 to a port. Your code should not do anything in this case.

For questions you can contact the student assistant Nurlan Dadashov by mail: nurlandadashov2002@yahoo.com