NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
PRACTICAL ASSESSMENT 1 FOR
Semester 2 AY2023/2024

CS2030S Programming Methodology II

March 2024 Time Allowed 120 minutes

---

## INSTRUCTIONS TO CANDIDATES

1. The total mark is 40. We may deduct up to 4 marks for violating of style.
2. This is a CLOSED-BOOK assessment. You are only allowed to refer to one double-sided A4-size paper.
3. Only the files directly under your home directory will be graded. Do not put your code under a subdirectory.
4. Write your student number on top of EVERY FILE you created or edited as part of the `@author` tag. Do not write your name.
5. To compile your code, run `javac -Xlint:unchecked -Xlint:rawtypes *.java`. You can also compile the files individually if you wish.
6. To check the style of your code, run `./checkstyle.sh`. You can also check the style of individual file with `java -jar checkstyle.jar -c pe1_style.xml <FILENAME>`.

**IMPORTANT:** Any code you wrote that cannot be compiled will result in 0 marks will be given for the question or questions that depend on it. Make sure your program compiles by running

```
javac -Xlint:unchecked -Xlint:rawtypes *.java
```

before submission.

**API:** The class `Exception` has the following two constructors:

```
Exception()            // Constructs a new exception with null as its detail message.
Exception(String msg)  // Constructs a new exception with the specified detail message.
```

A new SoC bus service is introduced to run between stops COM0, COM1, COM2, . . . and COM$k$ in a loop. At every stop, the bus will pick up and drop off passengers. For example, for $k = 3$, the bus will go from COM0 to COM1, then to COM2, then back COM 0, and then repeat.

Every stop has an id (e.g., for COM1, the id is 1) and, at every stop, there is a queue of passengers. When a bus reaches a stop, it lets all passengers in the bus who have reached the destination alight. It then lets as many passengers queueing up at the bus stop board the bus as possible until the bus reaches its capacity. SoC has limited space so there is a limit to the size of the queue at each stop.

The school is interested to know if this bus service meets the demand for our students. Knowing that you have helped a bank with building their simulation before, the school has tasked you to build a discrete event simulation for SoC buses.

1. (5 points) Implement a class named `Passenger` which represents the individual passengers taking the bus. Each `Passenger` has a unique id, starting from 0. The id of a `Passenger` cannot be changed. Each `Passenger` has a destination to reach, represented by the stop id.

   The `Passenger` constructor takes in an `int` value as the parameter, representing the `destination` stop of the passenger. The `Passenger` class has a method `hasReachedDestination` with an `int` stop id as its parameter. The method returns `true` if the given stop is the destination of this `Passenger`, and returns `false` otherwise. The `Passenger` class also overrides the `toString` method. Write your class so that it behaves exactly as follows:

   ```
   jshell> /open Passenger.java

   jshell> Passenger p = new Passenger(1);
   p ==> P0->COM1
   jshell> p.getDestination();
   $.. ==> 1
   jshell> p.hasReachedDestination(1);
   $.. ==> true
   jshell> p.hasReachedDestination(3);
   $.. ==> false

   jshell> Passenger p = new Passenger(4);
   p ==> P1->COM4
   ```

   You may add additional fields or methods to `Passenger` as necessary.

2. (3 points) Implement a checked exception `CannotBoardException` with a given error message.

   ```
   jshell> /open CannotBoardException.java

   jshell> throw new CannotBoardException("Testing 1 2 3")
   |  Exception REPL.$JShell$11$CannotBoardException: Testing 1 2 3
   |        at (#3:1)
   ```

3. (10 points) Now, implement the `Bus` class that represents the bus. The `Bus` constructor has two arguments of type `int` – the bus capacity, and the number of stops in its loop. The capacity of the bus and the total number of stops cannot be changed once initialized. Every bus has a unique id, starting with 0.

   Passengers alight from the bus in the same order in which they board. To ensure this, the `Bus` class maintains an array of `Queue<Passenger>`, indexed by the id of the destination stop. When implementing this field, you are permitted to use one `@SuppressWarnings` on the smallest scope of the declaration.

   Implements `Bus` so that it provides the following methods:

   - `board` takes in a passenger as an argument and adds the passenger to the bus. If the bus is full, a `CannotBoardException` with the message "`Bus is full`" is thrown. A passenger may go to an invalid stop.
   - `alight` takes no argument and removes all passengers that have reached their destination from the bus.

- `move` moves the bus to the next stop. At the beginning, the bus is at COM0 (Stop 0).

All three methods return the `this` reference of the target `Bus`.

The `Bus` class also overrides the `toString` method. An example string representation of `Bus` is in the form

```
B0@COM2 passengers: [ ] [ P1->COM1 ] [ ]
```

`B0` refers to the bus with id 0. `@COM2` refers to the fact that the bus is currently stopping at COM2. After the string "passengers: ", the three queues are printed. They correspond to the queues of passengers on the bus for destination Stops 0, 1, and 2 respectively.

Write your class so that it behaves exactly as follows:

```
jshell> /open Passenger.java
jshell> /open CannotBoardException.java
jshell> /open Queue.java
jshell> /open Bus.java

jshell> Bus bus = new Bus(2, 3);
bus ==> B0@COM0 passengers: [ ] [ ] [ ]

jshell> // Two passengers get on
jshell> bus.board(new Passenger(2));
$6 ==> B0@COM0 passengers: [ ] [ ] [ P0->COM2 ]
jshell> bus.board(new Passenger(1));
$7 ==> B0@COM0 passengers: [ ] [ P1->COM1 ] [ P0->COM2 ]

jshell> // Third passenger tried to get 0
jshell> bus.board(new Passenger(1));
|  Exception REPL.$JShell$12$CannotBoardException: Bus is full
|        at Bus.board (#4:35)
|        at (#8:1)

jshell> // Move to COM 1, alight a passenger
jshell> bus.move();
$9 ==> B0@COM1 passengers: [ ] [ P1->COM1 ] [ P0->COM2 ]
jshell> bus.alight();
$10 ==> B0@COM1 passengers: [ ] [ ] [ P0->COM2 ]

jshell> // Move to COM 2, alight a passenger
jshell> bus.move();
$11 ==> B0@COM2 passengers: [ ] [ ] [ P0->COM2 ]
jshell> bus.alight();
$12 ==> B0@COM2 passengers: [ ] [ ] [ ]

jshell> // Back to COM 0
jshell> bus.move();
$13 ==> B0@COM0 passengers: [ ] [ ] [ ]
```

4. (4 points) Implement a `Stop` that encapsulates a stop id and a `Queue` of `Passenger` objects. The class `Queue<T>` has been given to you. `Stop` provides the following methods and behavior:

   - `addPassenger` takes in a passenger as an argument and adds the passenger into the queue at the stop. The method returns `false` if the queue is full, and returns `true` if the passenger successfully queue up for the bus.
   - `removePassenger` takes no argument and returns the first passenger at the front of the queue. It returns `null` if no passenger is in the queue.
   - `isFull` returns `true` if the queue at the stop is full. It returns `false` otherwise.

The `Stop` class also overrides the `toString` method. Write your class so that it behaves exactly as follows:

```
jshell> /open Queue.java
jshell> /open Passenger.java
jshell> /open Stop.java

jshell> // Create stop COM1 with queue capacity of 2
jshell> Stop s = new Stop(1, 2)
s ==> COM1 [ ]

jshell> // Add two passengers
jshell> s.addPassenger(new Passenger(4))
$5 ==> true
jshell> s.addPassenger(new Passenger(2))
$7 ==> true
jshell> s
s ==> COM1 [ P0->COM4 P1->COM2 ]

jshell> // Add third passengers but queue is full
jshell> s.addPassenger(new Passenger(0))
$9 ==> false
jshell> s
s ==> COM1 [ P0->COM4 P1->COM2 ]

jshell> // Remove two passengers
jshell> s.removePassenger()
$11 ==> P0->COM4
jshell> s
s ==> COM1 [ P1->COM2 ]
jshell> s.removePassenger()
$13 ==> P1->COM2
jshell> s
s ==> COM1 [ ]

jshell> // Remove passengers when queue is empty
jshell> s.removePassenger()
$15 ==> null
```

5. (18 points) Now we are ready to implement our simulation. We have provided you with the class `Event`, `Simulator`, `Simulation` and `BusSimulation`. The first three classes are the same as what we provided in your exercises.

   To initialize the `BusSimulation`, the class `BusSimulation` reads the following the standard input:

   - three positive integers $N_s$, $N_b$, and $N_p$ that corresponds to the number of stops, the number of buses, and the number of passengers.
   - The next $N_s$ values are integers $c_0, c_1, \ldots c_{N_s-1}$, representing the queue capacity for Stop 0, 1, and so on.
   - The next $N_b$ lines contain two values each. Each line represents a bus. The two values represent the time each bus first leaves COM0 (a `double`) and its capacity (an `int`) respectively.
   - The remaining $N_p$ lines contain three values each. Each line represents a passenger. The three values correspond to the arrival time of the passenger (a `double`), the id of the stop the passenger is arriving at to wait for the bus, and the id of the passenger's destination.

   You can assume that the input is correctly formatted.

   The code to read these inputs has been given to you in `BusSimulation`, but the code to initialize the events has not. You should add the necessary code in `BusSimulation` and necessary classes to make the simulation run properly.

After initializing the simulator with the events specified in the input files above, the simulation runs as follows:

- The first time a bus leaves COM0, it picks up any passengers at COM0.
- A bus takes 1 unit of time to move from one stop to the next.
- Boarding and alighting is instantaneous and takes 0 unit time.
- A bus always lets passengers alight first, before picking up new passengers.
- The simulation stops after 10 units of time.

We are interested in simulating and printing out three events:

- Arrival of a passenger
- Departure of a passenger (due to full queue at a stop)
- Bus stopping at a stop

A sample expected output is as follows:

```
1.000 P0->COM1 arrives at COM0 [ ]
2.000 P1->COM1 arrives at COM0 [ P0->COM1 ]
3.000 P2->COM1 arrives at COM0 [ P0->COM1 P1->COM1 ]
3.000 P2->COM1 departs
4.000 B0@COM0 passengers: [ ] [ ]  Stop: COM0 [ P0->COM1 P1->COM1 ]
5.000 B0@COM1 passengers: [ ] [ P0->COM1 ]  Stop: COM1 [ ]
```

In the example above, Lines 1 - 3 are output from the arrival events. It prints the passenger followed by "arrives at" followed by the stop. Line 4 corresponds to the departure event. It prints the passenger, followed by the string "departs". Finally, the last two lines print the bus followed by the stop. Recall that, in `Simulator`, the event is printed first before it is simulated.

You may add new classes and methods as needed.

You can run the simulation using the main program `PE1`. You can find some test cases in `inputs` and the expected output in `outputs`. A test script `test.sh` has been given for testing.

You can test your code with

```
./test.sh
```

END OF PAPER