EECS 2030 Winter 2020: Lab 2

Lassonde School of Engineering, York University

INTRODUCTION TO LAB-2

Students in the same lab section are **strongly encouraged** to work in groups of 2 or 3 students. The purpose of this lab is to develop an understanding of Java Classes, Methods, Exceptions, and Utilities. This lab will be graded for style and correctness. This lab is due on **9th Feb 2020**.

STYLE RULES FOR CODING

The style rules are not overly restrictive in EECS2030.

1) Your programs should use the normal Java conventions (class names begin with an uppercase letter, variable names begin with a lowercase letter, public static final constants should be in all caps, etc.).
2) In general, use short but descriptive variable names. There are exceptions to this rule; for example, traditional loop variables are often called i, j, k, etc. Avoid very long names; they are hard to read, take up too much screen space, and are easy to mistype.
3) Use a consistent indentation size. Beware of the TAB vs SPACE problem: Tabs have no fixed size; one editor might interpret a tab to be 4 spaces and another might use 8 spaces. If you mix tabs and spaces, you will have indenting errors when your code is viewed in different editors.
4) Use a consistent and aligned brace style.
5) Insert a space around operators (except the period ".")
6) Avoid using "magic numbers". A magic number is a number that appears in a program in place of a named constant.
7) A good IDE (integrated development environment) such as eclipse will correct many style errors for you. In eclipse, you can select the code that you want to format, right click to bring up a context menu, and choose Source $\rightarrow$ Format to automatically format your code.

LAB EXERCISE 1

Declare a class `ComboLock` that works like the combination lock in a gym locker (Consult the API provided to look for the methods needed). The locker is constructed with a combination - three numbers between 0 and 39. The `reset` method resets the dial so that it points to 0. The `turnLeft` and `turnRight` methods turn the dial by a given number of ticks to the left or right. The `open` method attempts to open the lock. The lock

opens if the user first turned it right to the first number in combination, then left to the second, and then right to the third.

The tester file is provided so that you can check your output on the console. However, you need to write another Test class with at least five Junit test cases.

LAB EXERCISE 2

Write a program that plays tic-tac-toe. The tic-tac-toe game is played on a 3x3 grid. The game is played by two players, who take turns. The first player marks moves with a circle, the second with a cross. The player who has formed a horizontal, vertical, or diagonal sequence of three marks wins. Your program should draw the game board, ask the user for the coordinates of the next mark, change the player after every successful move, and pronounce the winner.

- Implement the private method `wonDiagonal()` which checks if player has won tic-tac-toe along diagonal lines.
- Implement the private method `wonStraightLines()` which checks if player has won tic-tac-toe along straight lines.
- Implement the public method `won()`, which checks if player has won.

If implemented correctly, you can play the game now. But the program has bugs. Try to choose a position that has been chosen (either by the same player or the other player), you will see that the new value replaces the old value, which is not correct. Also try to enter a position that is out of the range, e.g., 3 0, or 5 8 and you will see **ArrayOutOfBoundException** on the screen, which is also not desirable.

- Modify the method `choose()`, such that if the chosen cell has already been occupied, the method throws an exception **UnavailableCellException**.
- Modify the code in `main()` so that it catches both the **UnavailableCellException** and **ArrayOutOfBoundException** exceptions. The program should now run in such a way that if a player chooses a position that has already been occupied, or choose an invalid position, prompts the player to enter again until a empty cell is chosen.

The tester file is provided so that you can check your output on the console as shown in **Appendix A.**

LAB EXERCISE 3

Simulate a car sharing system in which car commuters pick up and drop-off passengers at designated stations. Assume that there are n such stations, one at every mile along a route. At each station, except the last station, there are randomly generated number of passengers, each of them have a desired target station. There are cars at each station except the last station. Each car has a target destination station. The car drive along the stations until it arrives at its destination station. For example, if a car is initially located at station 1 and has destination station 4, then it will drive along station 2 and 3 and finishes at station

4. Each driver picks up a maximum of three passengers (whose destination is either the same as car's destination or on the way to the car's destination), and drops them off where requested, and picks up more if possible. A driver gets paid per passenger per mile.

Complete classes `Car, Passenger, Station, Simulation` in your solution.

- Create a class **Passenger** with attributes `String name` and `int destination`.
  - Create a custom constructor `public Passenger(String name, int dest)`.
  - Write down the getter method `getDestination()` to determine the destination of the passenger
  - Write a `toString()` method to print the name and destination of the passenger.
- Create a class **Car**, which has attributes `int idNo, int destination, int location, ArrayList<Passenger> passengers, double faresCollected, int milesDriven, double FAREPERMILE = 1.0, and int MAXPASSENGERS = 3`.
  - Create a custom constructor to initialize `idNo, location, and destination`.
  - Create getter methods `getIdNo, getLocation, getDestination, getFares, getMilesDriven` and `getPassengers.`
  - Write getter methods `public boolean hasArrived()` to know whether the car has arrived its destination and `public boolean hasRoom()` to know whether the number of passengers in a car is within its maximum capacity.
  - Write mutator method `public void drive()`, which drives to the next station. If the car is already at its destination, nothing happens. Note that you should also collect fares. Also print out "Car X drives to station Y", as shown in sample output (see **Appendix B**).
  - Write method `public boolean dropoff()`. This method drops off any passengers who have arrived, and return true or false depending on if someone is dropped. If someone is dropped, also prints "Car X drops off Y at station Z. Now has W passengers", as shown in the sample output (see **Appendix B**).
  - Write method `public boolean add(Passenger p)`. This method adds a passenger if there is a room and return true or false depending on the passenger can be added or not.
  - Write the `toString()` method to print the car id number, location, destination, and passengers.
- Create a class **Station** with attributes `int stationNumber` and an array list `people` of type Passenger. Create a custom constructor `Station(int number)` and getter method `getNumber()`. Also create the following methods:
  - `void add(Passenger p)` that adds a passenger on the station
  - `Passenger get(int index)` that returns the index of a passenger
  - `Passenger remove(int index)` that removes a passenger from the station
  - `ArrayList<Passenger> peopleWaitingList()` that returns the list of people who are waiting at the station

  - `String toString()` that prints the people on a station
- Complete the incomplete method `void loadPassengers (Station s, Car c)` in the **Simulation** program. This method first picks up passengers going to the car's destination ("direct passengers"). Then picks up passengers going on the way ("ride passengers"). The passengers are picked in the order of their position in the waiting queue, which is the order they are added to the station.
  This Simulation program creates 6 stations (station 0-5) including the last station and sets different numbers of passengers at stations 0-4, and one car at each of the station 0-4. Then each time step each car takes turn to perform a sequence of operations "loadPassengers $\rightarrow$ drive $\rightarrow$ dropoff".
- Once you finished, run the **SimulationTester** program. If implemented correctly, you should get the sample output as shown in **Appendix B**.

SUBMIT INSTRUCTIONS FOR STUDENTS NOT WORKING IN A GROUP

Submit your solutions using the submit command. Refer to lab0 (manual) webpage for instructions.

Remember that you first need to find your workspace directory, then you need to find your project directory. In your project directory, your files will be located under different package directories. For example, your **ComboLock.java** should be in the directory **EECS2030_LAB2W20/src/combinationlock**.

Once you are in the directory, issue **submit 2030 lab2 ComboLock.java**

Repeat the above steps for each of the other program files.

At any time and in any directory, you can view the files that you have submitted by issuing **submit -l 2030 lab2**

SUBMIT INSTRUCTIONS FOR STUDENTS WORKING IN A GROUP

If you are working in a group, in addition to submitting program files as shown above, you also need to submit a text file containing EECS login names of your group members.

Create a plain text file named **group.txt**. You can do this in eclipse using the menu File $\rightarrow$ New $\rightarrow$ File, or, use any other text editors. Type your login names into the file with each login name on its own line. For example, if the students with EECS login names rey, finn, and dameronp, worked in a group the contents of **group.txt** would be:

rey
finn
dameronp

Submit your group information using submit command. In the directory where the text file is saved, issue **submit 2030 lab2 group.txt**

SUBMIT INSTRUCTIONS FOR STUDENTS OUTSIDE PRISM LAB

It is possible to submit work from outside the Prism lab, but the process is not trivial; do not attempt to do so at the last minute if the process is new to you. The process for submitting from outside of the Prism lab involves the following steps:

1) transfer the files from your computer to the undergraduate EECS server red.eecs.yorku.ca
2) remotely log in to your EECS account
3) submit your newly transferred files in your remote login session using the instructions for submitting from within the lab
4) repeat Steps 1 and 3 as required

Windows users will likely need to install additional software first. Mac users have all of the required software as part of MacOS.

Appendix A

The sample output will look like this.

```
|-----|
| | | |
|-----|
| | | |
|-----|
| | | |
|-----|
Player 1 (X) choose a row and column: 0 0
|-----|
|X| | |
|-----|
| | | |
|-----|
| | | |
|-----|
Player 2 (O) choose a row and column: 1 1
|-----|
|X| | |
|-----|
| |O| |
|-----|
| | | |
|-----|
Player 1 (X) choose a row and column: 0 0
Position Occupied. Try again.
Player 1 (X) choose a row and column: 1 1
Position Occupied. Try again.
Player 1 (X) choose a row and column: 2 2
|-----|
|X| | |
|-----|
| |O| |
|-----|
| | |X|
|-----|
Player 2 (O) choose a row and column: 2 2
Position Occupied. Try again.
Player 2 (O) choose a row and column: 5 0
Invalid Position. Try again.
Player 2 (O) choose a row and column: 4 3
Invalid Position. Try again.
Player 2 (O) choose a row and column: 0 2
|-----|
|X| |O|
|-----|
| |O| |
|-----|
```

```
| | |X|
|-----|
Player 1 (X) choose a row and column: 0 0
Position Occupied. Try again.
Player 1 (X) choose a row and column: 2 1
|-----|
|X| |O|
|-----|
| |O| |
|-----|
| |X|X|
|-----|
Player 2 (O) choose a row and column: 3 0
Invalid Position. Try again.
Player 2 (O) choose a row and column: 2 0
|-----|
|X| |O|
|-----|
| |O| |
|-----|
|O|X|X|
|-----|
Player 2 wins!
```

Appendix B

```
A Commuter-car-sharing simulation: 1 runs.

add passenger: P#0A->4 at station 0
add passenger: P#0B->4 at station 0
add passenger: P#0C->5 at station 0
add car: Car[idNo=1000, location=0, destination=1, passengers=[]] at station 0

add passenger: P#1A->4 at station 1
add passenger: P#1B->5 at station 1
add passenger: P#1C->4 at station 1
add car: Car[idNo=1001, location=1, destination=3, passengers=[]] at station 1

add passenger: P#2A->3 at station 2
add passenger: P#2B->3 at station 2
add passenger: P#2C->4 at station 2
add passenger: P#2D->5 at station 2
add passenger: P#2E->4 at station 2
add car: Car[idNo=1002, location=2, destination=4, passengers=[]] at station 2

add passenger: P#3A->4 at station 3
```

```
add passenger: P#3B->4 at station 3
add passenger: P#3C->4 at station 3
add passenger: P#3D->5 at station 3
add passenger: P#3E->5 at station 3
add passenger: P#3F->4 at station 3
add car: Car[idNo=1003, location=3, destination=5, passengers=[]] at station 3

add passenger: P#4A->5 at station 4
add passenger: P#4B->5 at station 4
add passenger: P#4C->5 at station 4
add passenger: P#4D->5 at station 4
add passenger: P#4E->5 at station 4
add car: Car[idNo=1004, location=4, destination=5, passengers=[]] at station 4


 ------------ timestep 1------------
Car 1004 loads direct passenger P#4A->5 at station 4. Car now has 1 passengers
[P#4A->5]
Car 1004 loads direct passenger P#4B->5 at station 4. Car now has 2 passengers
[P#4A->5, P#4B->5]
Car 1004 loads direct passenger P#4C->5 at station 4. Car now has 3 passengers
[P#4A->5, P#4B->5, P#4C->5]
Car 1004 drives to station 5
Car 1004 arrives at station 5.
Car 1004 drops off P#4C->5 at station 5. Car now has 2 passengers
Car 1004 drops off P#4B->5 at station 5. Car now has 1 passengers
Car 1004 drops off P#4A->5 at station 5. Car now has 0 passengers
Car[idNo=1004, location=5, destination=5, passengers=[]] finishes at
destination station 5. Miles: 1. Fare 3.000000

Car 1003 loads direct passenger P#3D->5 at station 3. Car now has 1 passengers
[P#3D->5]
Car 1003 loads direct passenger P#3E->5 at station 3. Car now has 2 passengers
[P#3D->5, P#3E->5]
Car 1003 loads ride passenger P#3A->4 at station 3. Car now has 3 passengers
[P#3D->5, P#3E->5, P#3A->4]
Car 1003 drives to station 4
Car 1003 arrives at station 4. Waiting queue: [P#4D->5, P#4E->5]
Car 1003 drops off P#3A->4 at station 4. Car now has 2 passengers

Car 1002 loads direct passenger P#2C->4 at station 2. Car now has 1 passengers
[P#2C->4]
Car 1002 loads direct passenger P#2E->4 at station 2. Car now has 2 passengers
[P#2C->4, P#2E->4]
Car 1002 loads ride passenger P#2A->3 at station 2. Car now has 3 passengers
[P#2C->4, P#2E->4, P#2A->3]
Car 1002 drives to station 3
Car 1002 arrives at station 3. Waiting queue: [P#3B->4, P#3C->4, P#3F->4]
Car 1002 drops off P#2A->3 at station 3. Car now has 2 passengers
```

```
Car 1001 has no one to load at station 1
Car 1001 drives to station 2
Car 1001 arrives at station 2. Waiting queue: [P#2B->3, P#2D->5]
Car 1001 has no one to drop off at station 2

Car 1000 has no one to load at station 0
Car 1000 drives to station 1
Car 1000 arrives at station 1. Waiting queue: [P#1A->4, P#1B->5, P#1C->4]
Car 1000 has no one to drop off at station 1
Car[idNo=1000, location=1, destination=1, passengers=[]] finishes at
destination station 1. Miles: 1. Fare 0.000000


 ------------ timestep 2------------
Car 1003 loads direct passenger P#4D->5 at station 4. Car now has 3 passengers
[P#3D->5, P#3E->5, P#4D->5]
Car 1003 drives to station 5
Car 1003 arrives at station 5.
Car 1003 drops off P#4D->5 at station 5. Car now has 2 passengers
Car 1003 drops off P#3E->5 at station 5. Car now has 1 passengers
Car 1003 drops off P#3D->5 at station 5. Car now has 0 passengers
Car[idNo=1003, location=5, destination=5, passengers=[]] finishes at
destination station 5. Miles: 2. Fare 6.000000

Car 1002 loads direct passenger P#3B->4 at station 3. Car now has 3 passengers
[P#2C->4, P#2E->4, P#3B->4]
Car 1002 drives to station 4
Car 1002 arrives at station 4. Waiting queue: [P#4E->5]
Car 1002 drops off P#3B->4 at station 4. Car now has 2 passengers
Car 1002 drops off P#2E->4 at station 4. Car now has 1 passengers
Car 1002 drops off P#2C->4 at station 4. Car now has 0 passengers
Car[idNo=1002, location=4, destination=4, passengers=[]] finishes at
destination station 4. Miles: 2. Fare 6.000000

Car 1001 loads direct passenger P#2B->3 at station 2. Car now has 1 passengers
[P#2B->3]
Car 1001 drives to station 3
Car 1001 arrives at station 3. Waiting queue: [P#3C->4, P#3F->4]
Car 1001 drops off P#2B->3 at station 3. Car now has 0 passengers
Car[idNo=1001, location=3, destination=3, passengers=[]] finishes at
destination station 3. Miles: 2. Fare 1.000000

------------ Commuter-car-sharing simulation done ----------
```