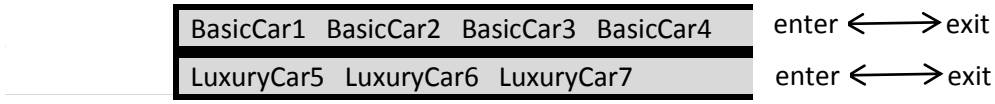
Regis University CC&IS
CS310 Data Structures
Programming Assignment 4: Stacks and Queues

Problem Scenario

The real estate office is located in the middle of Denver, where parking is scarce. Therefore, the office maintains some cars for the realtors to use. Due to limited space in the city, parking for the cars is only available one narrow lot that will hold two lines of parked cars.

Small basic cars are parked in one single line and larger luxury cars are parked in a second single line. There is only one entrance/exit for each line, at the front of lot. Cars are numbered, starting from 1 in the basic line, and continuing the numbering in the luxury line. The lines can currently hold 4 basic cars and 3 luxury cars.

So the basic cars will be numbered 1 to 4, and the luxury cars will be numbered 5 to 7.



This program will use stacks and queues to control use of vehicles in the lots.

As a reward for top sellers, realtors with over $1,000,000 in property listings have access to the luxury cars. When a realtor requests a car, they will be assigned the first available car from the appropriate line. A top seller is assigned cars from the luxury car line, and everyone else gets cars from the basic car line.

If all cars from the luxury line are gone, a top seller may be assigned a car from the basic car line. If all cars from both lines in the lot are out, or the realtor is not a top seller and the basic car line is empty, the realtor must wait until a car is returned. The requesting realtor is placed into a queue. There will be a queue for top sellers, and a queue for the other realtors. Realtors in the top seller queue will be assigned any returned cars ***before*** the realtors in the other queue.

Program Requirements

You will still be using the CS310<lastname>.java file to read an **`assn#input.txt`** input data file (as used for previous assignments), since you will still need to access the realtors, and will need to be able to determine which of them are top sellers.

The current **CS310<lastname>.java** file's **main** method will start as it did in Assn 3, but will skip the initial report, and only print the cleaned report. So the method will read the realtor and property listing data from the input file and create the linked lists, traverse and display the linked lists, clean up the lists, and generate a report of realtors and property listings. Then, for this assignment, you will be building several new implementations, and adding code to the end of the main method to manage the cars.

A second input data file will be used to simulate realtors requesting cars and returning cars. The filename is to be **`carInfo.txt`** and will be located in the **input** folder. Each line of data will be (note that file is NOT comma delimited, like the realtor data file):

```
REQUEST realtorLicenseNumber
```

or

```
RETURN realtorLicenseNumber
```

The cars for each line in the lot will be stored in a **stack**. When a realtor requests a car, the next available car is provided (by popping a car off the stack).

> You will create your own **CarStackImpl** using an array implementation. The data stored in each stack will be an integer, to represent the car number. Cars are numbered, starting from 1 in the basic line, and continuing the numbering in the luxury line (so the basic car stack will hold cars numbered 1 to 4. The luxury car stack will hold cars numbered 5 to 7). Use constants to define the car number ranges, so the program will work if the company decides to expand the available number of cars later.
>
> Each stack should be initialized so that all the cars are in the stack, with the lowest car number at the bottom of the stack and highest car number at the top of the stack. Note that this means the stack will initially be ***full***. When the stack is ***empty***, there are no more available cars available of that type.
>
> The **CarStackImpl** class should implement the ability to **push** and **pop** cars from the stack. You also need to include methods to determine if the stack is **empty** or **full** – and use these methods.

As realtors request and are assigned a car from the lot, you will need to be able to specify which realtor has which car.

> Choose an appropriate data structure to store this information. This data structure (and its methods) should be implemented within a **VehicleUsageImpl** class. As realtors request and return cars, you will need to update the car usage information.

If a realtor requests a car, but there are not any cars available, the realtor will be assigned to a queue to wait for the next available car. If the realtor is a top seller, that realtor is assigned to the top seller queue. If the realtor is not, then the realtor is assigned to standard realtor queue. A new waiting realtor will be added to the rear of the queue's linked list. When a realtor is assigned a returning car, and is removed from the queue, that realtor will be taken from the front of the queue's linked list.

> You will create your own **RealtorQueueImpl** using a singly linked list, with data field references to both the **front** and **back** nodes in the queues (you should be able to use your **RealtorNode** class from last week to build this linked list).
>
> This implementation will be used to create **two separate queues** of RealtorNodes. The same implementation will be used to instantiate each queue (representing lines waiting for cars) – one queue for waiting standard realtors, and one for waiting top selling realtors.
>
> The **RealtorQueueImpl** code should be able to **add** and **remove** realtors from a queue (i.e. the add method should have a **Realtor** as an input parameter, and the remove method should return a **Realtor**).
>
> You will also need to build methods to check if a queue is **empty** or **full**, and use the methods. Be sure to check if a queue is empty, before trying to remove a realtor from it. A queue will only be considered **full** if memory allocation fails when you try to instantiate a new node object.

The realty office also wants the program to check for other office workers trying to use the company cars. If a realtor requests a car, but his/her realtor license number is not in your **Realtor** linked list (built using the code from Assn 3), then issue an error message, ignore the request, and move on to the next request/return.

As each request/return action occurs, you will provide an audit trail.

A sample audit trail (for an implementation with only 2 cars per line, basic cars numbered 1 – 2 and luxury cars numbered 3 – 4) would look like the following.

```
Top seller Nick Noonan has been assigned luxury car number 3
Standard realtor Maria Munez has been assigned basic car number 1
Unknown realtor XX9999999 not allowed access to cars. Request ignored.
Standard realtor Joshua Jones has been assigned basic car number 2
Helen Hull waiting in standard realtor queue
Maria Munez has returned car number 1
Standard realtor Helen Hull has been assigned basic car number 1
John Johnson waiting in standard realtor queue
```

After processing the entire **`carInfo.txt`** input file, your application will provide a report with the existing car assignments (which realtor is currently using which car), which cars are available in each of the stacks (the top of the stack is to be listed first, followed by the other cars, in order as they are popped from the stack), and which realtors are still sitting in each of the queues (in the order that they were placed into the queue).

The report should be placed into a **`carUsageReport.txt`** file to be located in the **output** folder, and should be in the following format:

```
CAR USAGE REPORT
Helen Hull is using car number 1
Joshua Jones is using car number 2
Nick Noonan is using car number 3

AVAILABLE CARS
   BASIC CARS
      No basic cars are available

   LUXURY CARS
      Luxury car number 4 is available

TOP SELLER QUEUE
There are no top selling realtors waiting

STANDARD REALTOR QUEUE
John Johnson is waiting
```

The program must follow all **CS310 Coding Standards** from Content section 1.9.

Hints

After creating the linked lists from last week, you will need to create and call several new methods within **your CS310<lastname>** class:

ProcessCarInfo – read and process **carInfo.txt** data file.
The audit trail will be displayed as you read and process the data file.

ProcessCarRequest – handle Realtor request for a car

ProcessCarReturn – handle Realtors returning a car to the lot

You will also add:

A method to the **RealtorLogImpl** class to find a realtor node in the linked list for a given realtor license number, and return the node (or **null** if the realtor is not in the list).

A method to the **PrintImpl** class, which will create the final **carUsage.txt** file.

The new implementation classes for this program are:

CarStackImpl – implementation of stack of cars with car numbers. Note that you will instantiate two different stacks (basic and luxury) from the same **CarStackImpl** class.

VehicleUsageImpl – keeps track of which Realtors are currently using which cars

RealtorQueueImpl – implementation of queue of Realtors waiting for cars. Note that you will instantiate two different queues (standard and topSeller) from the same RealtorQueueImpl class.

Additional Requirements

- Create and/or modify **Javadoc headers**, and generate **Javadoc files**
- Your input data files will still be read from the **input** folder in your project.

  Place all test data files that you create to test your program in the **input** folder of your project, and name them as follows:

  Input files containing Realtors and Properties:

  `assn4input1.txt`
  `assn4input2.txt`

  (i.e. number each data file after the filename of **`assn4input.txt`**)

  and

  Input files containing car requests and car returns:

  `carInfo1a.txt`
  `carInfo1b.txt`
  `carInfo2a.txt`

  (i.e. number each data file to correspond to the **`assn4input.txt`** file used, and add letters for multiple test of the same **`assn4input.txt`** input file)

  Together, all of your test data files should demonstrate that you have tested every possible execution path within your code, including erroneous data which causes errors or exceptions.
- Your output report files will still be written to the **output** folder in your project.
- Add screen shots of **clean compile** of your classes to the documentation folder.

  WARNING: Submittals without the clean compile screenshots will **not** be accepted.
  (This means that programs that do not compile will **not** be accepted)

Program Submission

This programming assignment is due by midnight of the date listed on the **Course Assignments by Week** page of the Content.

- Export your project from NetBeans using the same method as you did for weeks 1 – 3.
    - Name your export file in the following format:
      `CS310<lastname>Assn<x>.zip`
      For example: **`CS310SmithAssn4.zip`**

      NOTE: Save this zip file to some other directory, not your project directory.
- Submit your **.zip** file to the **Prog Assn 4** Submission Folder (located under **Assignments** tab in online course).

  Warning: Only NetBeans export files will be accepted.
  Do not use any other kind of archive or zip utility.

Grading

Your program will be graded using the **rubric** that is linked under **Student Resources** page.

WARNING:
*Programs submitted more than 5 days past the due date will **not** be accepted, and will receive a grade of 0.*