CSC461 Basic Java (100pt)

DUE: as stated on D2L

The purpose of this assignment is to give you practice with basic Java syntax and classes, and an introduction on how to code to an API. This must run on IntelliJ and Java with a JDK 17.

There is a checklist that is required to be copied into the top of the District class.

Overview

You will be implementing an API for a number of parking lots in a district. You will be writing three classes: ParkingLot to track the number of vehicles in an individual lot and total funds, FreeParkingLot is a type of ParkingLot but the cost is 0, and District to track the lots in the parking district. This MUST be done in IntelliJ with Java 17.

Since coding to meet a specific API is common in the workforce, and in order to reuse other software or hardware, you are being provided with a testing class that **must** work **without** changes other than the package name. You can add attributes and private methods, but you must implement the methods (with the specified parameters) and you must stay consistent with the below requirements. I will mark anything that is not a match with a >>>>>>>>.

Do not comment out the tier test functions. The tests are designed to stop on the tier that fails. The first tier is getting this compiling **as-is**!

The key operations are

`markVehicleEntry` – called when a vehicle enters a lot, and the requires the number of minutes since the lot opened, and returns an integer ID to identify the car or -1 for a failure

`markVehicleExit` – called when a vehicle exits a lot, and requires the number of minutes since the lot opened, and the ID, if applicable, of the car that left the lot

Since the main class will be overridden, put your man file header comments in the District class. You must also use JavaDoc style comments.

NOTE: Since I reuse this assignment type (due to really good feedback), I will be doing my plagiarism check on this class AND the prior years that used this.

ParkingLot Class

Base Parameters

- IDs should be created starting at 0, and add one for each car that enters.
- Parking lots have `names`. To simplify testing, `ParkingLot` defaults the `name` to `"test"`.
- A lot also has an hourly fee for a car which is collected when the car exits or at the end of the day, that should be set in the constructor.
- The fee can vary, and has a default of $1.00 per hour

- A full constructor call's parameters are in name, lot size, and fee, in that order.
- Since name and fees have default values, it is legal to call a constructor with just an int for the size.
- The method `getVehiclesInLot` returns the number of vehicles in the lot at any one time.
- The method `isClosed` returns whether or not a lot is closed. It will return true is the number of vehicles is at 80% or above of the number of spaces in the lot. When it reaches this point, an electronic sign goes on saying the lot is closed. Drivers can ignore this sign and continue to enter, up to the capacity. After capacity is reached, `markVehicleEntry` should return -1 to signify a failure
- Create a class constant `CLOSED_THRESHOLD` for the 80% threshold. That should be stored in standard user input form (AKA 80). You will need to convert to 0.8 when doing your calculations. Use this constant in your code so that it would be easy to support changes in the policy.
- The method `getClosedMinutes` returns the **total** number of minutes during which the lot has been closed since the start of the day (as defined above). You can assume this method will never need to update the closed minutes between an entry or exit. This is a simplification, otherwise this function would need the current time.
- Add a function called `getProfit()` that will return the current funds collected. Since this is the funds currently collected, this will be accurate even if the cars have not left the lot yet.

Entry/Exit specifications

- It is expected that time never goes backwards in this class. If `markVehicleEntry` or `markVehicleExit` is called with a time that is before some other call, assume the sensor glitched and ignore the event, and return -1 to signify a failure. Otherwise update the current time to the given time.
- If `markVehicleExit` is given a car with an invalid id, simply ignore it as a glitch.
- A car in paid parking car may only pay when exiting. However, the first 15 minute after entry and paying are ignored. This means to charge the full amount of time if they are in the parking lot for **more** than 15 minutes, and nothing otherwise.

Strings

- Add a `toString` method. This method is to return a string of the form "Status for [name] parking lot: [x] vehicles ([p]) Money Collected: $[totalProfit]" where [name] is filled in by the name, [x] by the number of vehicles currently in the lot, and [p] by the percentage of the lot that is occupied. The percentage may have up to 1 decimal place shown, only if needed. If the percentage is at or above the threshold, display "CLOSED" rather than the percentage inside.
- The profit should be rounded to the nearest cent and must display to 2 decimal places.

  Example:
  ```
  Status for red paid parking lot: 1 vehicles (50%) Money Collected: $1.42
  Status for gray paid parking lot: 2 vehicles (66.7%) Money Collected: $2.57
  Status for green paid parking lot: 2 vehicles (50%) Money Collected: $0.73
  ```

FreeParkingLot class

- `FreeParkingLot` is derived from `ParkingLot`
- Its constructor may not allow submitted a cost as the cost must be 0. It must extend the parent's constructor. No copy past of anything that can be reused in the parent.
- If `markVehicleExit` must **extend** the ParkingLot class (no copy paste!) and merely decreases the count in the lot. The ID is not checked, and therefore is not needed. But, to ensure it works right, you must override the original 2 parameter version so the ID is ignored correctly.
- There is no change to `markVehicleEntry. The ID can still be used to indicate successful entry.`
- Change the string output to remove "Money Collected: $[totalProfit]". You may not copy and past the main contents of the toString from its parent.
  - Hint: consider breaking the string construction apart with helper functions.

  Example
  ```
  Status for pink parking lot: 1 vehicles (50%)
  Status for blue parking lot: 2 vehicles (66.7%)
  ```

District Class

`District.java` purpose is the manage the individual ParkingLots. If any function needs more than one parking lot, it is best done in `District`.

- You must be able to handle a variable number of parking lots using ONE ArrayList of parking lots
- The methods `markVehicleEntry` and `markVehicleExit` are in `District` as well, and take an additional parameter in with is parking lot index of which the vehicle is entering or exiting.
- The method `isClosed` for `District` returns true if and only if all the parking lots are closed at that time.
- The method `closedMinutes` returns the number of minutes that **all** of the parking lots are closed at the **same time**. This information would be used to determine if more parking lots are needed.
- Add a method `add` that takes in a Parking Lot, and returns its index.
- Add a method `getLot` which returns the `ParkingLot` at the given index. You may assume a valid index.
- Add a method `getVehiclesParkedInDistrict` the returns the total number of parked cars in the district.
- Add a method `getTotalMoneyCollected` that returns the total number money collected (rounded to the nearest cent) in the district.
- Add a method call `isClosed` that returns true is ALL parking lots are closed
- The value returned by `toString` must be a “District status:\n” followed by the toString returned by each lot in the list.

Other restrictions

- You must not copy and paste the contents of the constructor to other constructors, even within the same class. You must use the hierarchy, chain the constructor class, and/or use an initialization function. To ensure this is done correctly, “tag” where you accomplished this as an inline comment **once** in `ParkingLot` with `GRADING: CONSTRUCTION` at the beginning of the chain. Make sure this is copied exactly so a search may find it.
- Your member variables must be private unless they are a constant, or they are only used in the current class and derived classes. In other words, you must use proper encapsulation.
- You MUST NOT rewrite the parent class’s `toString` (or any overridden function) contents if they are used. You must extend them.

REMINDER: When grading, the testing file will be OVERWRITTEN with the original. Before you submit, copy down the test file and paste it in to ensure you did not accidently change it.

Creating the Project

Do the following to set up your IntelliJ project:

1. Download the testing file and save it in a convenient location on your computer.
2. Create a new basic Java project.
3. Right click the src folder, and then New→Package. Name the package your last name followed by your full first name separated by an underscore, all LOWER case. As an example, mine would be rebenitsch_lisa. My script adds the package name based on your D2L recorded name.
4. Right click on the package and choose "Show in Explorer."
5. Copy the test file to the folder that was opened.
6. Open the test file and change the package name to match your package folder, EXACTLY.
7. You can now browse the source files, but it will give you a LOT of error messages until you place the scaffolding/skeleton for the needed functions. Your first time is to provide sufficient stub function to get it compiling.

Grading Tiers

You must reasonable complete the tier below before you can gain the points for higher tiers. "Reasonably complete" means it is reasonable to assume you think you have it working (e.g., works in standard test cases). In the case of this project, the test file has the tests ordered for you already.

The reason for this, is to encourage you to code and test in parts, rather than the typical waterfall method. Scaffolding or skeleton code is NOT the waterfall method, so feel free to add basic code to the functions. However, when you start completing functions, start with the most fundamental objects first to ensure they work. In this assignment, this will be ParkingLost, then FreeParkingLot, then District, and finally a mix of lots with District. Since this is the first assignment, I've given you many built-in tests as a template for how to code in parts. Passing each test is passing another tier.

Submission instructions

1. Copy down the test file and paste it in to ensure that you did not accidentally change it. I will be overwriting it.
2. All of your Java files must use a package named your lastName_firstName (lowercase with no prefixes or suffixes). The name is based on your D2L shown name.
3. Delete the out folders, then zip your **ENTIRE project** into ***ONE*** zip folder. This must be able to be unzipped with a single command.

If you are on Linux, make sure you see "hidden folders" and you grab the ".idea" folder. That folder is what actually lists the files included in the project!

4. Submit to D2L. The dropbox will be under the associated topic's content page.
5. ***Check*** that your submission uploaded properly. No receipt, no submission. Test proper submission by downloading and see if it opens and runs in IntelliJ.

You may upload as many times as you please, but only the last submission will be graded and stored

If you have any trouble with D2L, please email me (with your submission if necessary).

Rubric

All of the following will be graded all or nothing, unless indication by a multilevel score. You must complete the tier below before you can get points for the next tier. Given the way I marked a failed test in this assignment, the only "reasonable complete" and still having errors is in the code structure, not runtime correctness.

Item	Points
Other deductions (on top of other error)	
Got it compiling with test file	5
Additional OOP requirements	10
toString properly extended	5
Constructors properly handled	5
Tier 1: one parking space	8
Tier 2: one parking lot with coming and going	8
Tier 3: Parking lot stress test	12
test closures	4
test refilling a lot	4
test emptying a lot	4
Tier 4: test time errors	6
Tier 5: Test overfilling	6
Tier 6: test tiny district	8
Tier 7: test full district	7
Tier 8: test heavy use	7
Tier 9: test pay parking	8
Tier 10: test pay parking with glitches	8
Tier 11: district with pay parking	7
Total	100