CSC461 Basic Java

DUE: Thursday, September 26th, at 8AM

The purpose of this assignment is to give you practice with basic Java syntax and classes, and an introduction to code to an API. This must run on IntelliJ and a JDK 11.

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, `PayParkingLot` is a type of `ParkingLot` but also tracks how much money is gathered, and `District` to track the lots in the parking district. This MUST be done in IntelliJ with Java 11.

Since coding to meet a specific API is common in the workforce, 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.

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

`markVehicleExit` – called when a vehicle exits a lot, and requires the number of minutes since the lot opened, and the ID 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.

ParkingLot Class

- 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.
- Parking lots have `names`. To simplify testing, `ParkingLot` defaults the `name` to `"test"` when no color is specified.
- 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 so drivers do not waste fuel circling the lot to find one of the few remaining spaces. 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. Use this constant in your code so that it would be easy to support changes in the policy.
- The method `getCosedMinutes` returns the number of minutes during which the lot is closed (as defined above). You can assume this method will never need to update the closed minutes while the lot is closed; that is, enough vehicles will have exited that the closed sign has turned off, reopening the lot to new drivers. This is a simplification, otherwise this function would need the current time.
- Add a `toString` method. This method is to return a string of the form "Status for [name] parking lot: [x] vehicles ([p])" 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" for the percentage inside.
  Example:

```
Status for pink parking lot: 17 vehicles (68.2%)
Status for blue parking lot: 28 vehicles (CLOSED)
Status for gray parking lot: 2 vehicles (20%)
```

PayParkingLot class

- `PayParkingLot` is derived from `ParkingLot`
- A paid 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
- It must override the `toString` function to include the current money collected. It should append " Money Collected: $[totalProfit]" to the string produced by the parent class. The profit should be rounded to the nearest cent and must display to 2 decimal places.
- 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.
- If `markVehicleExit` is given a car with an invalid id, simply ignore it as a glitch.
- A car in paid parking car may "pay as they go" similar to refilling a parking meter. The first 15 minute after entry and paying are ignored to allow time to reach the exit. This means other cars can exit before they do. Therefore,
  - Do not charge if they try to exit/pay before they came. This is a glitch like in the ParkingLot class.
  - If they try to exit
    - If it is under 15 since the entry or last pay, do not charge, and do not update their last time of entry/pay.
    - If it is over 15 minutes since the entry or last pay, charge them since the last time of entry/pay, and update their last time of entry/pay.
  - If the car does not exit (the time is already updated to a later time due to another car leaving), leave the car in the lot.

Distract 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.

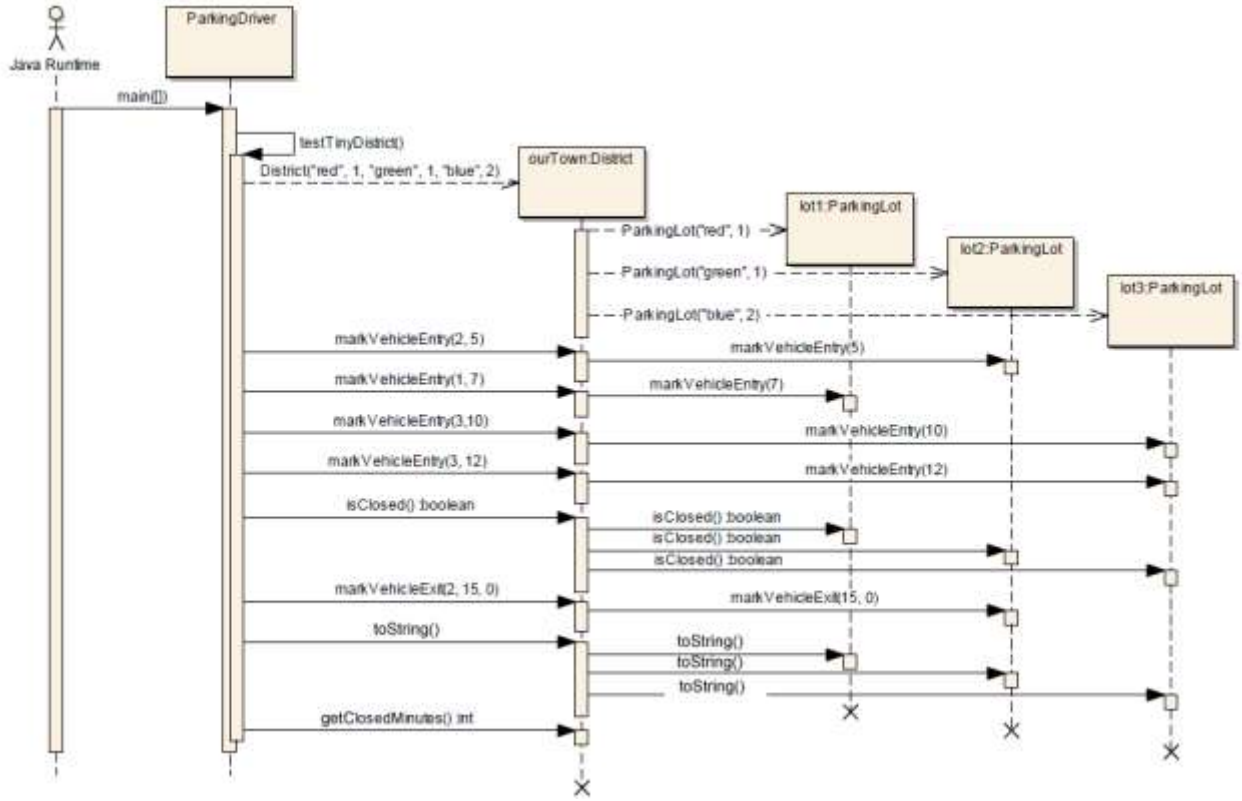
  You will need to use something similar to `if( lot instanceof PayParkingLot)` for this to work properly, since your regular and paid parking lots are in the same ArrayList. There is an OOP pattern that can avoid `instanceof` (which is considered bad practice), but for one task, this is *far* simpler.

- Add a method call `isClosed` that returns true is ALL parking lots are closed
- The value returned by `toString` must be a string with newline characters embedded in it. For example, `toString` could return a string such as

```
Status for purple parking lot: 8 vehicles (CLOSED)
Status for gold parking lot: 20 vehicles (33.3%)
```

Example Sequence Diagram

A sequence diagram has been made for one of the tests below. Note the general pattern: a message is sent to `ourTown`, and this generates messages to one or all of the parking lots. The first horizontal line is shorthand for adding all three parking lots.



Formatting number the easy way

Java has another formatting option other than printf for formatting numbers call DecimalFormat. Decimal format asks for how you want your number to output described in a string with # and 0. # means "at most this many digits," and 0 means "there must be a digit here". As an example, the following will output π to the third decimal place

```
DecimalFormat formater = new DecimalFormat("0.000");
String result = formater.format(Math.PI);
```

Other restrictions

- You must not copy and paste the contents of the constructor to other constructors. You must use the hierarchy or an initialization function.
- You member variable must be private unless they are a constant, or they are only used in the current class and derived classes only. 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.
- You must also use JavaDoc style comments.

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 first name separated by an underscore, all LOWER case. As an example, mine would be rebenitsch_lisa.
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 message until you place the scaffolding/skeleton for the needed functions.

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 ParkingLot, then District, and then PayParkingLot, then PayParkingLots with Districts. Since this first assignment, I've given you a large number of built in tests as a template for how to code in parts later. Passing each test is passing another tier.

Submission instructions

1. Check the coding conventions before submission.
2. Copy down the test file and paste it in to ensure you did not accidently change it
3. If anything is NOT working from the following rubric, please put that in the bug section of your main file header. This helps with feedback and potentially partial credit.
4. I will be overwriting the ParkingTest.java file with the original when I grade.
5. All of your Java files must use a package named your lastName_firstName (lowercase with no prefixes or suffixes)
6. Delete the out folders, then zip your **ENTIRE project** into ***ONE*** zip folder named your lastName_firstName (lowercase with no prefixes or suffixes). Make sure this matches your package name!

   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!

7. Submit to D2L. The dropbox will be under the associated topic's content page.
8. *Check* that your submission uploaded properly. No receipt, no submission.

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 grades all or nothing, unless indication by a multilevel score. You must reasonably complete the tier below before you can get points for the next tier.

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