Project 2
Traffic Simulation

Due Date/Time: July 18 at 11:45 pm

This assignment requires you to create a complete Java program consisting of multiple source code files. We are providing much of the actual code. You must electronically submit your source files by the due dates listed according to your section.

You must electronically submit your source files by the due date. Directions for using submit are at the end of this page.

Requirements

Anyone who has waited to pay a toll at the New Jersey Turnpike or the Tappan Zee Bridge understands that traffic flow through toll gates is impacted by the amount of traffic and the number and kind of open toll booths. For this assignment, you must simulate traffic at a toll plaza. The input to the simulation will be the number of toll booths in the plaza and the amount of traffic on the highway. Your simulation must estimate the average amount of time vehicles wait in line and the average amount of time they require to pay the toll.

The problem domain

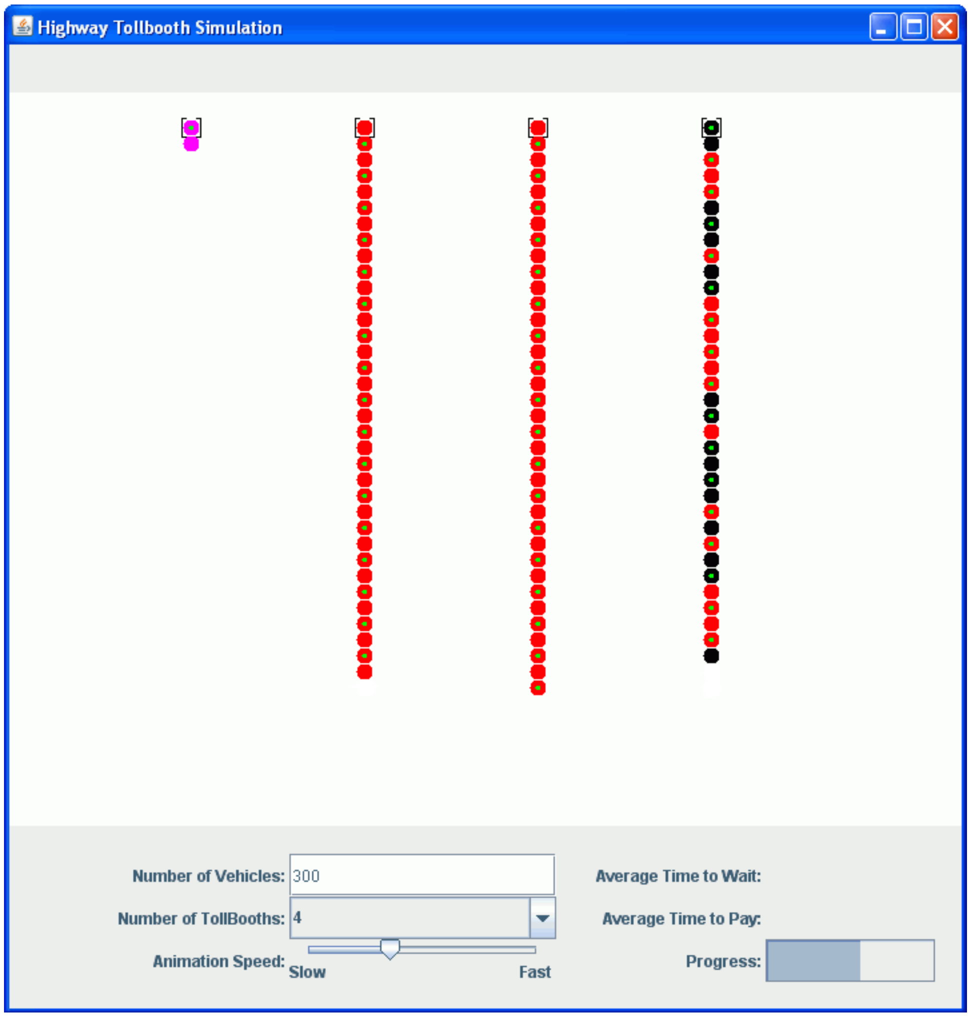
This simulation for this problem models a toll booth plaza with 3 to 9 toll booths. There are 3 kinds of vehicles: cars, cars with EZ-Pass cards, and trucks. The lane a vehicle chooses depends on the kind of vehicle:

- EZPass cars can enter any lane. They always choose the shortest lane. If two or more lanes are shortest, then an EZPass car always chooses the lane closest to the left. (The leftmost lane is reserved for EZPass holders only.)
- Cars that are not EZPass holders can enter any lane except the leftmost (EZPass only) lane. They always choose the shortest lane among the rest. If two or more of the remaining lanes are shortest, then a car always chooses the one of those that is closest to the left.
- Trucks can enter only truck lanes, which are always at the right of the other lanes. The number of truck lanes varies according to the number of toll booths. To compute the number of truck lanes, take 25% of the number of toll booths and round up. For example, if there are 6, 7, or 8 toll lanes, then the 2 rightmost are truck lanes; if there are 9 toll lanes, then the 3 rightmost are truck lanes. A truck always choose the shortest truck lane. If two or more of the truck lanes are shortest, then a truck always chooses the one that is closest to the left.

Animating a simulation run

The picture below is a snapshot of an animation of a simulation run of 100 vehicles through 4 toll booths. We provide that animation front end for you. The Start button begins the simulation run; the Quit button terminates program execution.

EZ Pass cars are magenta, cars are red, and trucks are black. (The green dots in the middle of every other vehicle in a line help in visualizing the animation.) Note the EZPass only lane is on the left and the truck lane is on the right. There are both cars and trucks in the truck lane. The progress bar shows that this simulation run is more than half completed.



Number of Vehicles: 300

Number of TollBooths: 4

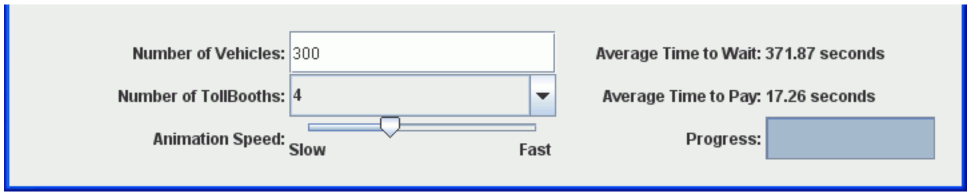
Animation Speed: Slow Fast

Average Time to Wait:

Average Time to Pay:

Progress:

When the simulation run is completed, the bottom part of the screen shows the simulation results of average wait-in-line time and average pay time.



Number of Vehicles: 300

Number of TollBooths: 4

Animation Speed: Slow Fast

Average Time to Wait: 371.87 seconds

Average Time to Pay: 17.26 seconds

Progress:

Simulation events

Simulations are based on *events* that contribute to the underlying problem domain. Do not confuse simulation events with events on graphical user interfaces -- the two concepts are unrelated.

There are two kinds of simulation events for the turnpike traffic simulation, corresponding to the two different locations where things occur:

1. At the highway. The event is the vehicle's leaving the highway. When a vehicle leaves the highway, it enters the rear of its selected toll booth line. Later, the same vehicle will get to the front of the toll booth line, log its information, and leave the simulation.
2. At a toll booth. The event is the vehicle's departure from the front of a toll booth line. As each vehicle departs, it pays its toll, logs its information (how long it stood in line and how long it took to pay its toll), and then leaves the simulation.

An *event calendar* determines the order in which the events occur. Implementing a simulation typically involves creating an event calendar then going through the calendar to process each event in order. Of course, processing one event may create additional events, which are subsequently scheduled on the event calendar.

Design

Your project **must** follow the stringent design guidelines laid out here. Keep in mind that the primary purpose of this assignment is to teach you composition. inheritance, polymorphism, abstract classes, and interfaces.

The major objects in the traffic simulation from the problem domain are the vehicles (cars,

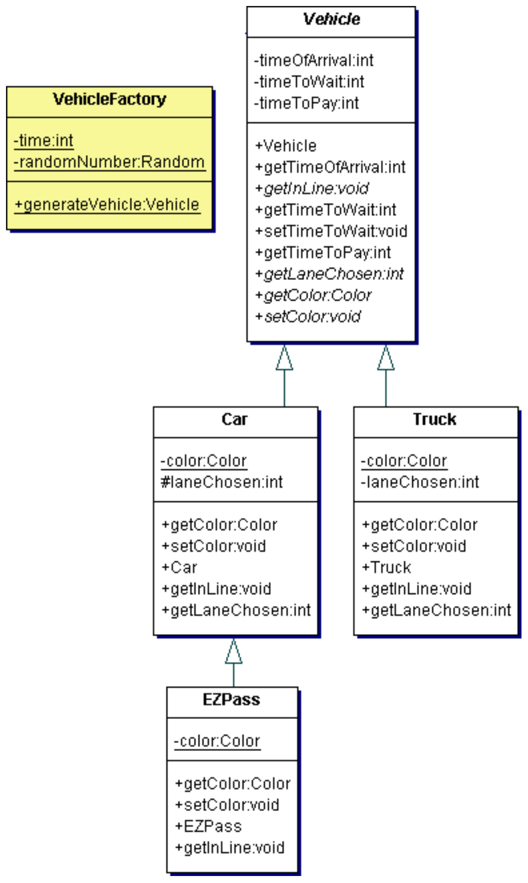
cars with EZPass cards, and trucks), the highway, and the toll booths. Think of the toll booths as queues, which are lists in which items are processed in the order in which they enter the list. A toll booth queue is the line of vehicles waiting to pay their tolls. You should also think of the highway as a queue -- the first vehicles on the highway are the first ones to reach the toll booth plaza. (That's clearly not the way of ordinary traffic, but we do not care how vehicles get down the road. It is simple and sufficient to first generate all the vehicles for the highway, placing them in the highway queue as they are created.)

The other objects that are part of the simulation include a simulator and a log of toll booth traffic. Ancillary objects are queues and a *factory* that generates the vehicles to feed the simulation.

Vehicle hierarchy

There are 3 kinds of vehicles: cars, cars with EZPass cards, and trucks. They choose their toll booth lines by different algorithms. But they all can get in line, have a time of arrival, and have a time to actually pay their tolls. This is a perfect situation to use inheritance. The design calls for an abstract base class, `Vehicle`. It has two children, `Car` and `Truck`. The `Car` class has a child, `EZPass`.

The UML diagram to the right illustrates the `Vehicle` class hierarchy. `VehicleFactory`, which is the yellow class to the side (not actually part of the hierarchy) is one we provide. It is the entity that creates vehicle objects for the highway. Do not change it.



`Vehicle` has one constructor, which has two parameters: the time the vehicle arrives at the toll booth plaza and the time it spends paying the fine. `Vehicle` has three attributes, which are all private:

- `timeOfArrival`. Time when vehicle arrives at the toll booth plaza (when the vehicle enters a toll booth line). Specified by the constructor. Time is time in seconds from the beginning of the simulation.
- `timeToPay`. Number of seconds required to pay the toll (the time at the actual toll booth -- does not include time to wait in line). Specified by the constructor.
- `timeToWait`. Number of seconds the vehicle waits in line before paying. This should be computed when the vehicle first enters a toll booth line.

The abstract `Vehicle` methods are shown in the class diagram in italics. Abstract methods are left to its children to define.

The child classes, `Car` and `Truck`, have two additional data members:

- `color`. A static member used for display. You can pick any colors you want for your classes (client code can change colors if needed). Static members are underlined in UML class diagrams.
- `laneChosen`. Index of the toll booth that the vehicle chooses. The value of this attribute is set when the `getInLine` method executes. The hash mark in front of `laneChosen` of the `Car` class means that this data member is protected (visible to the descendants but nowhere else).

There are getters and both of the data members and a setter for `color`. The class `EZPass` inherits all of the members of `Car`, but it overrides `getInLine` according to its own algorithm for deciding which lane to choose.

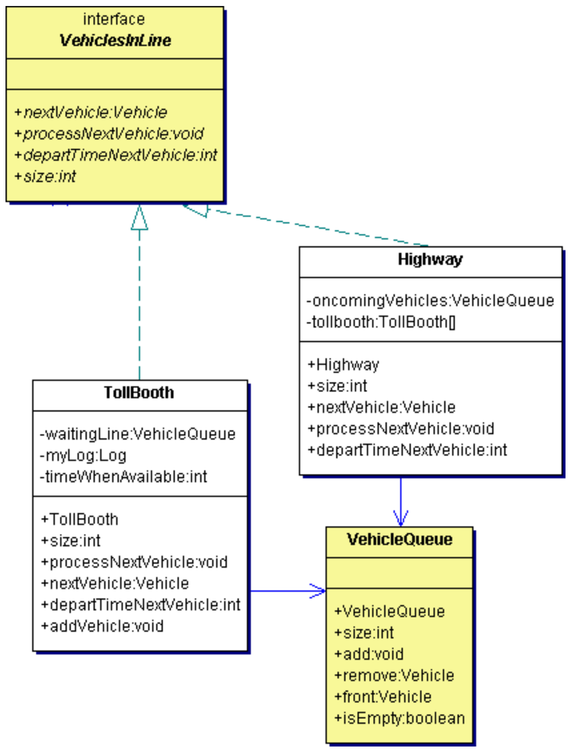
The `getInLine` method for each class has a single parameter, which is an array of TollBooths. In this method, the vehicle enters the back of the line for the toll booth lane it chooses.

The queues

A queue is a first-in first-out list, just like a line at a grocery store checkout or at a theater ticket office. There are two distinct kinds of queues in the simulation design: the highway and a toll booth line, which correspond to `Highway` and `TollBooth` in the design. These two classes must implement the interface `VehiclesInLine`, which we provide.

`VehicleQueue`, which we also provide, maintains all the information on the queue. It has the minimal required queue behaviors: add a vehicle to the end, remove one from the front, show the front vehicle without removing it, and tell whether it is empty. Do not change this class at all. It is a utility class that `Highway` and `TollBooth` can use without knowing how it works!

The UML diagram to the right shows the relationships among the queues. Both `Highway` and `TollBooth` have `VehicleQueue` instance variables (shown by the solid arrows). This is an excellent example of *composition*.



The `Highway` class has 2 data members and 5 methods:

- `oncomingVehicles`. The queue, which is initialized by the constructor. Vehicles are added to the queue in order of their `timeOfArrival`.
- `tollbooth`. An array of `TollBooth` objects, used by `processNextVehicle`.
- `Highway` constructor. Has 2 arguments, the number of vehicles in the simulation and the array of `TollBooths`.
- `size`. Returns the number of vehicles still on the highway.
- `nextVehicle`. Shows the next vehicle ready to leave the highway and enter a toll booth lane.
- `processNextVehicle`. Removes the front vehicle from the highway queue and sends it a `getInLine` message.
- `departTimeNextVehicle`. Tells when the vehicle a the front of the `oncomingVehicles` queue will depart that queue (and subsequently enter a toll booth line).

The `TollBooth` class has 3 data members and 6 methods.

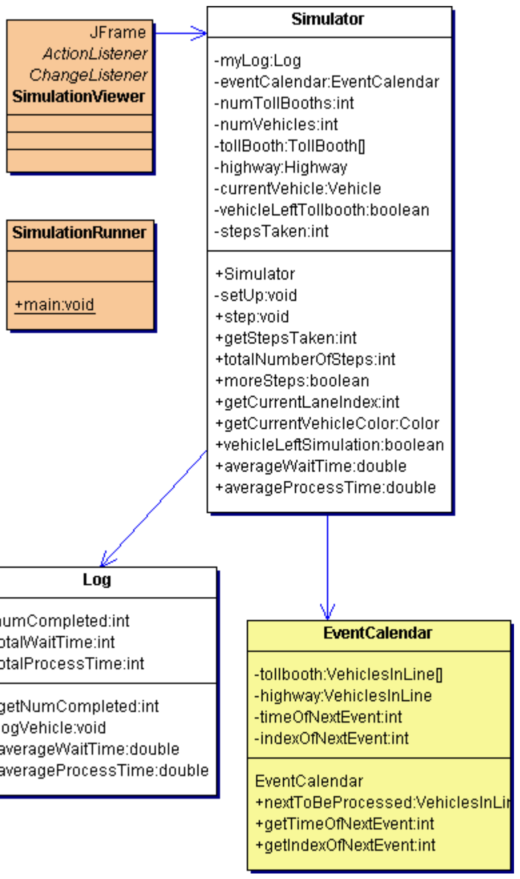
- `waitingLine`. The queue of vehicles in line.
- `myLog`. The departing vehicle logs its information here. (Logging is explained below. Since a `TollBooth` *has-a* `Log`, the final UML diagram will have an arrow from `TollBooth` to `Log` to indicate the relationship.)
- `timeWhenAvailable`. The time when this toll booth will finally clear out all the vehicles currently in line.
- `TollBooth` constructor. Has one parameter, which is of type `Log`. This initializes `myLog`.
- `size`. The number of vehicles in line.
- `nextVehicle`. Shows the front vehicle in line.
- `processNextVehicle`. Removes the front vehicle from the queue, logging its information in the process.
- `departTimeNextVehicle`. Tells when the vehicle currently at the front of the line (paying its toll) will finish paying its toll and leave the simulation.
- `addVehicle`. Adds a vehicle to the end of the line, updating the vehicles time to wait in the process.

The interface `VehiclesInLine` specifies the behaviors that lines of vehicles must provide in order to be part of the event calendar. We have created this interface and documented it for you. You are not allowed to change `VehiclesInLine`.

Simulation bookkeepers and recorders

The Simulator, EventCalendar, and Log classes control running the simulation and keeping track of the results. EventCalendar, which we provide for you, determines when events occur. Log keeps count of the statistics -- how long it took vehicles to pay their tolls and how long they waited in line prior to paying. Simulator runs the simulation and provides step-by-step information to any viewer that needs it.

The UML class diagram to the right shows how those three classes are related. It also shows two other classes, SimulationViewer and SimulationRunner, which you can use to test your code. We provide those classes too. You are allowed but not encouraged to change them.



Read the EventCalendar documentation to understand where it fits in the design. You are not allowed to change EventCalendar.

The Log class is straightforward. It has 3 data members.

- numCompleted. The number of vehicles that have logged their information.
- totalWaitTime. The sum of all wait times logged by vehicles so far.
- totalProcessTime. The sum of all times that vehicles took to pay their tolls (not including wait time)

The log method of the Log class takes care of updating the three data members. The cumulative statistics are available through the remaining methods.

Simulator runs the simulations step-by-step. Steps correspond to events -- there is a step whenever a vehicle leaves the highway to join a toll booth lane and there is a step whenever a vehicle leaves a toll booth lane. Simulator can report on the results of each step as it goes along.

The public Simulator constructor has two parameters, the number of vehicles and the number of toll lanes, which it uses to create a Highway and an array of TollBooths. The constructor also creates the Log to keep track of the traffic statistics and an EventCalendar to control the order of events.

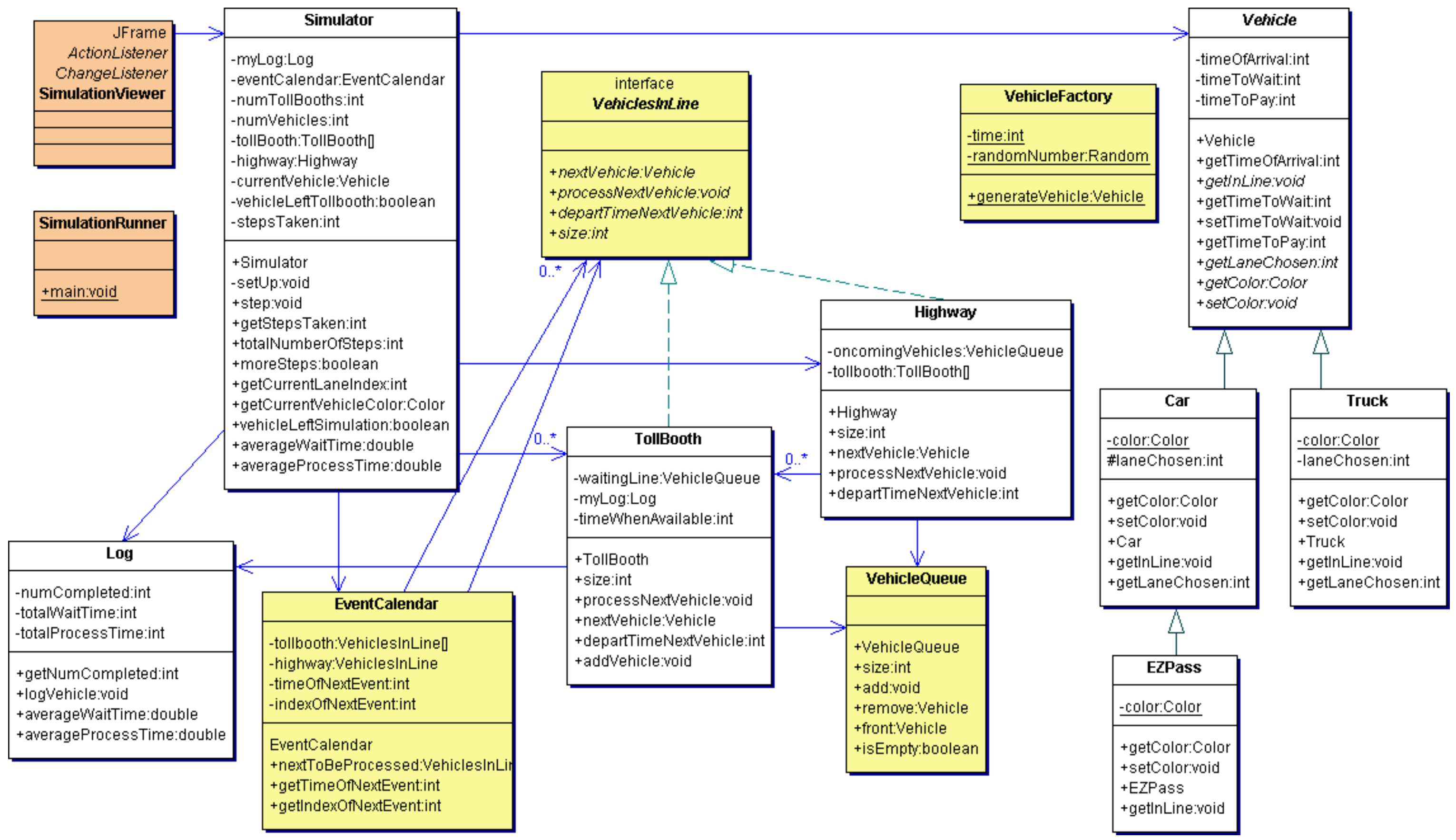
Simulator must contain the following public methods:

- step. Process the next event from the EventCalendar.
- getStepsTaken. Number of steps taken so far.
- totalNumberOfSteps. Total number of steps in the simulation.
- moreSteps. true if the simulation hasn't finished, false if it has.
- averageWaitTime. Average number of seconds vehicles had to wait in line prior to paying their tolls.
- averageProcessTime. Average number of seconds vehicles spent paying their tolls.
- getCurrentLaneIndex. Index of tollbooth selected by the most recently processed vehicle.
- getCurrentVehicleColor. Color of most recently processed vehicle.
- vehicleLeftSimulation. true if the most recently processed vehicle paid its toll and left the toll booth line; false if the most recently processed vehicle left the highway to join a toll booth line.

We also wrote a private Simulator method, setUp, to do some of the preliminary work. This method was called by the constructor.

The entire UML diagram

The following UML diagram shows the relationships among the classes in the instructor solution. You must construct the white classes. We provide the ones that are colored. You are not allowed to change the yellow ones.



Implementation

All of your classes must be in a package named `csc216project2`.

> **Important.** You should wrap your head around the design of this project before you begin coding. The most difficult part of this assignment is getting an in-depth understanding of the design.

Vehicle classes

You cannot get a working version of your program right away, but you can start with the `Vehicle` class hierarchy. Look at `VehicleFactory` to see what parameters are required by the constructor.

The method `getInLine` chooses the line for this vehicle base on the algorithm described in the Requirements section above. It has the array of `TollBooths` for a parameter.

Highway and TollBooth

The `Highway` constructor has two parameters, the number of vehicles in the simulation and an array of `TollBooths`. The constructor should use `VehicleFactory` to construct the amount of vehicles given by the parameter and place them on the highway (add them to the highway queue). For `departTimeNextVehicle`, if the queue of vehicles still on the highway is empty, set the return value to `Integer.MAX_VALUE`.

The `TollBooth` constructor has a single parameter of type `Log`, which it uses to initialize its `Log` instance variable. The method `processNextVehicle` should remove the vehicle from the front of its line and log its information.

The only subtle `TollBooth` method is `addVehicle`. It must take the current vehicle and add it to its waiting line. But it must also compute `timeWhenAvailable`, which is the earliest time when the *next* vehicle that joins it will be able to pay its toll.

Log and Simulator

The `Log` class has instance variables that keep running totals of wait times, pay times, and number of vehicles logged. `Log` has a null constructor. Its log method has a `Vehicle` parameter, which it uses to get the vehicle's statistics.

The `Simulator.step` method does the work of running the simulation. The remaining `Simulator` methods provide information to any client code (`SimulationViewer` and

`SimulationRunner`). The `step` method works like this:

1. Find out which queue (highway or tollbooth line) has the vehicle for the next event.
2. Get the next vehicle from that queue.
3. What is the length of the vehicle's selected line?
4. Send a message to the line to process the vehicle.
5. Keep track of what's happening to that vehicle's tollbooth line -- did this event make it shorter (did the vehicle leave the line)?.

This intermediate bookkeeping information is eventually used when the `SimulationViewer` animates the simulation.

Testing

This project is difficult to test. Even unit testing is difficult because of the high coupling of the classes. (High coupling, by the way, is considered undesirable. Now you know one of the reasons.)

> So you should look at your code carefully before trying to run it. And run it first with the `SimulationRunner`. You should get the following results when executing the `SimulationRunner`.

# vehicles	# tollbooths	Average Wait Time	Average Pay Time
100	4	120.20 seconds	17.21 seconds
5000	9	7.59 seconds	17.35 seconds
1000	3	2430.06 seconds	17.46 seconds
10000	8	257.22 seconds	17.41 seconds
10000	6	3693.91 seconds	17.41 seconds

When the numbers are coming out right, try out the `SimulationViewer`. If you specify a large number of vehicles (say more than 500), set the speed to Fast. Also, note that the random number generator gives slightly different numbers for each simulation that is started for a single execution of `SimulationViewer`. So the final results for two different simulations with the same number of vehicles and toll booths will differ slightly.

Deployment

For this class, deployment means submitting your work for grading. Before you submit your work, make sure your program:

1. Satisfies the style guidelines.
2. Behaves as specified in this document. You should test your code thoroughly. Be sure to know what messages should be displayed for each major scenario.
3. Satisfies the gradesheet for this assignment.

You must submit the Java source code files that you created:

- `Vehicle.java`
- `Car.java`
- `Truck.java`
- `EZPass.java`
- `Highway.java`
- `TollBooth.java`
- `Log.java`
- `Simulator.java`

We will use our own copies of the code that we provided when we execute your code.

You should submit your program through wolfware:

http://courses.ncsu.edu/csc216/

The name of the assignment is Project_2. Submit your source code files only (the .java files). Do not attempt to submit .class files.

The electronic submission deadline is precise. Do not be late. You should count on last minute system failures (your failures, ISP failures, or NCSU failures). You are able to make multiple submissions of the same file (the later submissions overwrite the earlier ones with the same file name). To be on the safe side, start submitting your work as soon as you have completed a substantial portion.