CSC 250 – Project #3
Fall 2014

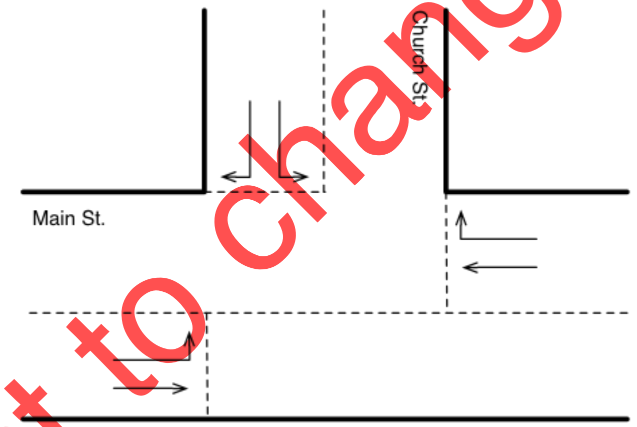
Goals: Design and implement a traffic intersection simulator using a `LinkedQueue<Vehicle>`.

Part 1: Project Summary (Due: 5pm November 19th, 2014; Drop dead date: None)

Simulate the traffic flow of a four-way traffic intersection given specific flow requirements. You will need to use multiple instances of the `LinkedQueue<Vehicle>` collection in order to complete this assignment.

Part 2: Processing

Church Street intersects Main Street in downtown Squaleville, NJ. Church runs north/south, and Main runs east/west. Travel on Church or Main at the intersection is limited to those directions represented in the diagram to the right.



Implementation Requirements

To receive full marks for this project, you must complete the `jsjf` implementation of a `LinkedQueue` that support generic types. Your queues must contain the `Vehicle` objects mentioned below. Your solution must contain a simple Driver class that contains your main method and which instantiates another class of your design (for example, let's call it `Simulator`) and then calls an appropriate method in that class (for example the simulate `method` in the `Simulator` class). You may not use any type of array for this solution without prior approval by the instructor. You may not use any Java API-provided collections without the approval of the instructor.

ANT

Create and fully test an ANT build file (named build.xml). The build file should contain the following targets:

1. **compile** - compiles the project source code
2. **clean** – removes the .class, the META-INF directory and its all of it's contents (if any), and the output file from the build area. This target or its underlying tasks should not fail if the file/dir is not present (output file, META-INF, etc.)
3. **jar** – creates the jar file for uploading (includes the build file, excludes the .class and output files)
4. **author** – prints the name of the program's author to the screen
5. **run** – executes the program, this target depends on the **compile** and **author** targets

Input

The input to this program is the random generation of vehicles and assignment to a street/directional queue codified in the program. There is no reading from the keyboard or other input source. You should create a `Vehicle` class to represent a vehicle travelling in to this intersection.

Each `Vehicle` object should store the following instance data. Note the use of enumerated types. You may need to refresh on this topic via your textbook and/or the Java tutorial.

- `vehicleNumber` (int) – the integer value of the number of the vehicle. The first vehicle generated is #1, the second #2, etc. There is no vehicle numbered 0 or has a negative number.

- `street` (enum) – “Church” or “Main” are the only two possible values here and they represent the street the vehicle is *initially* placed upon. Please note the spelling and capitalization of the values.

- `direction` (enum) – “S”, “E”, or “W” are the only four possible values which represent the direction of movement upon the street where the vehicle is *initially* placed (see the street variable). Please note the capitalization of the values.

- `arrivalTime` (int) – the elapsed time in seconds since the start of the simulation that the vehicle arrived at the intersection.

- `departureTime` (int) – the elapsed time in seconds since the start of the simulation that the vehicle exited the intersection.

Processing

The rules for processing the simulation are as follows:

1. First, pre-populate the intersection (any of the six possible locations) with the random creation of 7 to 12 vehicles (inclusive). Vehicles arriving during this time should have their `arrivalTime` value set to zero (0). This means that you should be instantiating 7-12 Vehicle objects, and randomly assigning them to a direction and lane (thereby defining their position in the intersection, and thereby inserting them into one of the 8 queues).

2. The traffic light that permits the vehicles to move in the southbound direction enables movement for a total of 6 seconds (3 seconds per vehicle). Move the vehicles in the southbound direction on Church St. for a total of 6 seconds. In this direction, up to 4 vehicles can turn left or turn right per movement period, but only up to 2 vehicles can turn left and up to 2 vehicles can turn right prior to the light changing. That is, during the first 3-second interval, one vehicle, if present, can move through the intersection. Depending on the population of the lanes, you may be moving up to 2 vehicles. Then, in the next 3-second interval, up to another 2 vehicles may be moving again.

   A vehicle must be in the lane (each lane is a queue) for it to be eligible to move. Thus, under certain situations, no vehicles will move through the intersection (if none are present waiting to move). Once a vehicle moves through the intersection, its `departureTime` value should be set to the value of the clock (see sample output below).

3. Next, populate the intersection with the random creation of an additional 8-15 vehicles (inclusive). Adding new vehicles to the intersection adds no time to our simulation clock (it really should happen while we are processing the movements in steps #2 and #4, but we’ll overlook this situation). Vehicles are randomly assigned a direction and lane, and can be added to any of the 8 queues at this time as well.

4. Move the vehicles in the east/west-bound direction. In this direction, up to a total of 12 can move through the intersection. The traffic light that permits the vehicles to move in the east/west-bound direction enables movement for 9 seconds. Here again, each vehicle takes 3 second to move through the intersection. This means that a maximum of 8 vehicles may turn during this change in the light, but the number could be lower (see the rationale in step #2).

5. Populate the intersection again with the random creation of 3-15 additional vehicles (inclusive) and return to processing step #2 (above).

6. Continue this processing sequence until you have depleted the queues of vehicles.

In addition to these processing requirements, note the following:

1. When populating new vehicles in the simulation, add no more than 120 total vehicles. Once the threshold of 120 vehicles has been reached, skip all further attempts to add additional vehicles. Check your output that only 120 vehicles were instantiated, assigned a vehicle number, and moved through an intersection. You will lose points in your simulation if you under- or over-populate the simulation.

2. The simulation should continue until no vehicles exist in any queue and all 120 vehicles had been placed in a queue and been processed through the intersection.

3. Once a vehicle moves through the intersection (turns right or continues straight), it can be effectively removed from the intersection. However, you will need to print information about the vehicle to the output file (see below). You can choose to do this once the movement is completed, or once the queues are empty, just prior to the end of the simulation.

4. When checking the flow of traffic, time elapses from the simulation regardless if there are vehicles present or not.

Output
Output for this project will consist of a listing of alternating state changes (light changes) and each vehicle's data in the order the vehicle proceeded through the intersection. Output should be written to a file named "output.txt". The format of the output is shown below. Follow this format explicitly, including blank lines, capitalization, numbering, the use of square brackets, punctuation, etc.

Example
```
---Start of simulation, time set to 0.
---Light changed. Now processing southbound traffic---
[Time 03] Vehicle #1 (southbound) turned right and headed westbound. Total wait time 03 seconds.
[Time 03] Vehicle #3 (southbound) turned left and headed eastbound. Total wait time 03 seconds.
[Time 06] Vehicle #7 (southbound) turned right and headed westbound. Total wait time 06 seconds.

---Light changed. Now processing east/west-bound traffic---
[Time 09] Vehicle #4 (eastbound) continued straight. Total wait time 09 seconds.
[Time 09] Vehicle #8 (eastbound) turned left and headed northbound. Total wait time 09 seconds.
[Time 09] Vehicle #2 (westbound) continued straight. Total wait time 09 seconds.
[Time 12] Vehicle #6 (eastbound) turned left and headed northbound. Total wait time 12 seconds.

---Light changed. Now processing southbound traffic---
<simulation output continues>
```

Part 4: Specifics for the delivery of this project:

To submit your project, jar up only the source code file(s) and your ANT build file and submit it to Canvas.

The task of organizing and creating the jar file for each deliverable will be up to you. You should always check your work, as this file is what I will be grading. You want to ensure that only the files I request are contained in the jar file.

Be sure to fully check the operation of the Ant build file as well, which may include moving your jar file to a clean (empty) directory, unjarring it and performing the ant commands as I will be doing.

You may submit the project any number of times until the due date. Only the last submission will be downloaded, examined, and graded by me.

Late points will be deducted for projects turned in starting 1 minute after the due date (that's why they are called due dates!) Following the drop-dead date for a project, no solutions will be graded for the project. At such a point in time, you will receive a zero grade for the work if you have not uploaded a solution to Canvas.

At a bare minimum, your program should compile. I expect that in this class you are minimally capable of delivering a barely functional program that can be compiled using the command line tools and/or Ant. Failure to deliver a compliable program will result in an automatic 50 score and further analysis of your submission will cease.

Part 5: Advice and Disclaimer

Plan your schoolwork and life accordingly. Many students don't adequately plan their work schedules and attempt to finish programs at the last minute. Doing so usually introduces bugs/problems into the solution. Consider how much time you will require to adequately test your solution for boundary cases and bugs. Failure to plan on your part does not constitute an emergency on my part!

I do not give individual extensions for projects, and rarely give class extensions for work. You have ample time to complete this project. I suggest you start your design as soon as possible. Consider what you need to build to solve the problem at hand. Programming involves designing a solution, implementing your solution, and testing your solution.

Part 6: Do You Need Help?

Are you stuck? Ask yourself if you have planned the design of your project. Have you asked questions in class? Did you attend the lectures, and labs? Have you come to see me? Like a textbook or web site, I am a resource that I expect you to use throughout the course of the semester. If you are stuck with your project, I strongly suggest you make an appointment to come see me in person. Generally, I can offer some guidance or help to get you back on the path.

When you come to see me, plan on bringing your source code to me on a memory stick or CD-Rom. Do not email me your source code, I do not provide email-based help for coursework. I prefer that you make time to visit me in my office where we can sit down and discuss your source code together. I can ask you questions and you can tell me why you coded something the way you did. We can use my whiteboard to draw large diagrams and talk about good solutions and designs. This is not something that can be done easily through email.

Part 7: Code Commenting and Formatting

Please refer to the separate document provided on Canvas for the details of the commenting and formatting guidelines.