Assignment 4: Traffic Simulator

Fecha de entrega: March 19th, 2018, 09:00am

Objetivo: Object Oriented Design, Java Generics, and Collections.

Traffic Simulator Overview

Computers allow us to model the world. Simulations can run these models and help us to understand them better; and, if we are lucky, apply this understanding to the real world. The assignments for the second part of this course will ask you to build and refine a traffic simulator, which will model vehicles, roads and junctions. You will be able to test different junction policies, and most importantly, gain insights into object-oriented modeling of real-world problems.

In this first assignment, you will build a basic simulator with a textual interface. The simulator maintains a collection of *simulated objects* (vehicles on roads connected by junctions); and a discrete time counter that is incremented in a loop while performing the following operations:

1. Process pre-scheduled *events* that can add or alter simulated objects;
2. Advance the state of currently simulated objects, according to their behaviors. For example, vehicles will advance if the road ahead is clear, but will have to stop and wait for red traffic lights.
3. report on the current state of the simulated objects.

Events are read from a file before the simulator starts; and then the simulation loop is run for a given number of time units (called *ticks*), while reports are gathered and written either to a file or to the standard output.

The assignment is split into two parts. The first part describes the basic simulator, with some default behavior for roads, vehicles and junctions; and the second part adds new types of roads, vehicles and junctions (where you should use object-oriented principles). You are strongly recommended to code and test the first part before moving on to the second part.

Simulated Objects

There are 3 types of simulated objects:

- vehicles, which can drive through roads and may break down occasionally;
- one-way roads on which vehicles travel; and
- junctions that connect roads, and manage traffic lights to decide which vehicles can advance to their next road.

All simulated objects have unique identifiers, can update themselves when requested to by the simulation (by calling an `ADVANCE` operation), and can report on their state (by calling a `REPORT` operation). Each object's `ADVANCE` operation will be called exactly once per simulation tick.

Vehicles

Vehicles in the simulation have a *maximum speed*, *current speed*, a *road* that they are traveling on, a *position* on that road (distance from the beginning of the road, the 0 point), and a travel *itinerary*: a list of junctions that they want to travel through. Vehicles can also break down and become *faulty*. While faulty, vehicles stop moving, and make the road slower for any vehicles behind them. Faulty vehicles will be repaired after a pre-set number of simulation ticks.

Vehicles support the following operations:

- `ADVANCE`. Asks the vehicle to advance. When asked, the vehicle first checks if it is in a *faulty* state (their `faulty` counter is positive), and in such case, `faulty` is decreased by 1 and the vehicle does not move. If it is in a *working* state (`faulty` was zero), it advances its location by the current speed: the new location will be the old location plus its current speed. If the new location equals or exceeds the current road length, then it is set to the road length, and the vehicle enters the queue of the corresponding junction (all roads are limited by two junctions). Vehicles waiting in a junction queue cannot advance, and remain on that specific location of the road until the junction calls their `MoveToNextRoad`.
- `MoveToNextRoad`. Asks the vehicle to move on to the next road. In such case, it simply *exits* the current road and *enters* the next road (at location 0) in its itinerary. Note that the first time this operation is applied it does not exit any road as it has not entered any yet. Note also that when the vehicle exits it last road in the itinerary it does not enter any road – it is state should be *arrived*.
- `MakeFaulty`. Puts the vehicle in a faulty state for n ticks. If the vehicle is faulty then n is accumulated to the counter `faulty`.
- `SetSpeed`. Sets the speed of a vehicle (will be used by roads). If the provided speed exceeds the *maximum speed of the vehicle* then the speed is set to the maximum speed.
- `REPORT`. Displays the state of this vehicle, in the following textual format. For more details on formatting, refer to section 3.1.

```
[vehicle_report]
id = v1
time = 5
speed = 20
kilometrage = 60
faulty = 0
location = (r1,30)
```

Where `id` is the vehicle identifier; `time` is the time on which the report was generated, `speed` is the current speed of the vehicle; `kilometrage` is the total distance traveled by the car allured; `faulty` represents remaining fault time (0 if the vehicle is not currently faulty); and `location` is the location of the car which is composed by the road identifier and the location on that road, or simply the keyword `arrived` if the car has arrived to its destination already (see 3.4).

Note that the speed of the vehicle should always be 0 for vehicles that are in a faulty state, are waiting at a junction, or have arrived at their destination.

Roads

Roads in the simulation have a *length* and maximum *speed*, and maintain a list of the vehicles that are currently on them, sorted by vehicles location (location 0 come last). **Note that multiple vehicles can be at the same location**. However, their order of arrival at this location must be preserved (in the list), so that the vehicle that arrives there first will be the first to leave.

Roads provide the following operations:

1. `ENTER`. Used by vehicles to enter a road. Adds the vehicle to the corresponding vehicles list.
2. `EXIT`. Used by vehicles to exit the road. Removes the vehicle from the vehicles list.
3. `ADVANCE`. Used by the simulator to advance the road, including all the vehicles on the road. The road must then
   a) calculates its (default) `base_speed` as $\text{mín}(m, \frac{m}{\text{máx}(n,1)} + 1)$ where m is the maximum speed of the road and n is the number of vehicles on that road (the `base_speed` will be different for the new kind of roads that we will add later). **Note that the division is an integer division**.
   b) For each vehicle:
      1) sets the speed of the vehicle to `base_speed/reduction_factor`, where `reduction_factor` is a function of the number of faulty vehicles (obstacles) in front[1]of this vehicle before any vehicle has advanced. The default `reduction_factor` is 1 if there are no obstacles and 2 otherwise (the `reduction_factor` will be different for the new kind of roads that we will add later).
      2) asks the vehicle to `ADVANCE`.

[1]Vehicle A is considered to be in front of vehicle B when the location of A is strictly larger than the location B.

4. `REPORT`. Displays the status of this road, in the following textual format. For more details on formatting, refer to section 3.1.

```
[road_report]
id = r3
time = 4
state = (v2,80),(v3,67)
```

   Where `id` is the road identifier; `time` is the time on which the report was generated; and `state` is a list with the identifiers and positions of all vehicles in the road (in the same order as they are stored in the vehicles list).

Note that when all vehicles have advanced, if several are at the same location, their relative order in the vehicles list should be the arrival order. **Recall that the list should always sorted by location, with location** 0 **coming last**.

Junctions

Junctions in the simulation manage incoming roads using traffic lights, which can be green to allow traffic to enter (and then leave through any connected outgoing road) or red to make the vehicles from that road wait. At any given time, exactly one incoming road can have a green light.

Junctions provide the following operations:

1. `ENTER`. Used by vehicles to enter an incoming-road queue when they first arrive at the junction. Vehicles in the queues are ordered according to the arrival order (first in first out).

2. `ADVANCE`. Must:

   a) Asks the first vehicle (and **only** the first vehicle, if any) from the queue for the incoming road with the green light to move to its next road, removing it from the queue.

   b) Update the traffic light. By default, junctions lights should go green for one tick for each incoming road, advancing cyclically through incoming roads in the order in which roads were added to the junction, regardless of whether there are any waiting vehicles or not.

   Note that at the beginning, before any call to `ADVANCE`, the all lights are red, and the first one to get green light is the incoming road that was added first to the junction.

3. `REPORT`. Displays the status of this junction, in the following textual format. For more details on formatting, refer to section 3.1.

```
[junction_report]
id = j2
time = 5
queues = (r1,red,[]),(r2,green,[v3,v2,v5]),(r3,red,[v1,v4])
```

Where `id` is the junction identifier; `time` is the time on which the report was generated; and `queues` is a list with the state of all incoming-road queues (in the same order that roads where added). Each queue is described with its road identifier, the state of its traffic light (green or red), and the list of vehicles in the corresponding queue (as they ordered in the queue).

Events

Events allow us to set up and interact with the simulator, by adding vehicles, roads and junctions; and making vehicles faulty. Each event has a time at which it should be executed. However, in textual event representation, the `time` key is always optional. When the event time is not specified, it should be understood to be 0, and the event must therefore be scheduled to execute at the start of the simulation. At each tick t, the simulator executes all events scheduled at time t, in the order in which they were added to the event queue. To execute events, the simulator call their `EXECUTE` operation.

INI Format

Events are loaded from a text file at the start of the simulation. The file uses the ancient `INI` format, [2] which must also be used when generating reports. In this format, there can be multiple sections, each with a section label. Within each section there can be multiple lines, each with a textual key and an associated value.

Together with this assignment, you will receive a package that can parse and generate `INI` files and sections. This package can also compare two `INI` files or sections, to ensure that they are equivalent. You should use it to verify that your output matches the expected outputs for the examples provided with the assignment.

For the `INI` sections used in this assignment, we will adopt the following conventions:

- Simulated **object identifiers** must always be strings composed of only letters, numbers, and underscores.
- **Identifier lists** must only contain valid identifiers separated by commas, without any whitespace.

You should always check that the provided identifiers are valid before executing any event.

Finally, note that the `INI` parser that we provide you has a useful feature (not available in standard `INI`) that allow sections to be ignored. To do so simply add "!" before the section tag. For example, the parser will ignore this section:

```
[!make_vehicle_faulty]
time = 2
vehicles = v1
duration = 3
```

This is useful for experimenting — instead of removing the whole section you can just comment it.

Next we give the details of all events that should be supported.

[2] Popularized by its use in Microsoft Windows until the arrival of the registry in Windows NT. See https://en.wikipedia.org/wiki/INI_file for more details.

New Junction

Adds a new junction to the simulator, with the corresponding parameters.

Textual representation

```
[new_junction]
time = <NONEG-INTEGER>
id = <JUNC-ID>
```

New Road

Adds a new road to the simulator, with the corresponding parameters.

Textual representation

```
[new_road]
time = <NONEG-INTEGER>
id = <ROAD-ID>
src = <JUNC-ID>
dest = <JUNC-ID>
max_speed = <POSITIVE-INTEGER>
length = <POSITIVE-INTEGER>
```

Add Vehicle

Adds a new vehicle to the simulator. The itinerary is a list of junction identifiers and it must have at least two junctions.

Textual representation

```
[new_vehicle]
time = <NONEG-INTEGER>
id = <VEHICLE-ID>
max_speed = <POSITIVE-INTEGER>
itinerary = <JUNC-ID>,<JUNC-ID>(,<JUNC-ID>)*
```

Break Vehicles

Makes all cars that correspond to the vehicle identifiers faulty for the given fault duration (by calling their `MakeFaulty` operation).

Textual representation

```
[make_vehicle_faulty]
time = <NONEG-INTEGER>
vehicles = <VEHICLE-ID>(,<VEHICLE-ID>)*
duration = <POSITIVE-INTEGER>
```

The Simulator

The simulator maintains a list of events, a data structure for storing simulated objects, a time counter (initially set to 0), the event queue, and an `OutputStream` to be used for writing reports. The `OutputStream` can be set to `System.out` to write to the standard output; or to a `new FileOutputStream(filename)` to write to a file.
The simulator should provide the following operations:

- `AddEvent`. Adds an event to the event queue. Events should be ordered by their `time`, and then by their order of arrival. **The event time must be greater than or equal to the simulator's current time.**
- `RUN`. Performs the following pseudocode, where *ticks*, the number of ticks to simulate, is provided as input, and `time` is the simulator's time counter:

```
limit = time + ticks - 1;
while (time <= limit) {
  // 1. execute events scheduled for this time
  // 2. advance roads
  // 3. advance junctions
  // 4. time++;
  // 5. print a report to the current OutputStream
  //    if it is not null
}
```

  The report in the last step above is generated simply by calling the `REPORT` operation of *all simulated objects* in the following order: first reports of all junctions, then reports of all roads, and then reports of all vehicles. In each case, the corresponding simulated objects are reported in the order in which they have been added to the simulator.

The Main class

You will receive, together with this assignment, a skeleton main class, which uses an external library to simplify the parsing of command-line options.

The main class should support the following command lines:

```
> java Main -h
usage: Main [-h] -i <arg> [-o <arg>] [-t <arg>]
 -h,--help           Print this message
 -i,--input <arg>    Events input file
 -o,--output <arg>   Output file, where reports are written.
 -t,--ticks <arg>    Ticks to the simulator's main loop (default
                     value is 10).
```

Examples.

```
java Main -i eventsfile.ini -o output.ini -t 100
java Main -i eventsfile.ini -t 100
```

The first example writes the reports output to a file `output.ini`, while the second writes it the standard output (= the console). In both cases the input events file is `eventsfile.ini` and the ticks is 100.

Part II: Advanced Objects

You should now have a working simulator using basic vehicles, roads and junctions. This second part introduces new vehicle, road and junction types which should be added to the simulator, without breaking any existing functionality. Each new simulated object variant can require additional event parameters, and may have to generate additional output in their reports.

All advanced objects are created with similar events as their corresponding basic versions, extended with a `type` key that identifies the specific variant that must be created. Certain variants also include additional initialization parameters, as described below. Advanced object reports are identical to their corresponding basic objects, but must additionally include their `type` attribute; and, possibly, additional information as specified in the particular variant.

Vehicles

There are 2 new kind of vehicles: Cars and Bikes.

Cars

Cars are vehicles that randomly enter faulty states with a certain probability once they have been driven for long enough. When a car's `ADVANCE` operation is performed, it first checks if the following conditions hold:

- the car is not in a faulty state;
- the car has traveled at least **`resistance_km`** since the last time in which it was in a faulty state (note that it is not the total distance traveled, you should track this information);
- a randomly chosen real number x, between 0 (inclusive) and 1 (exclusive), is smaller than **`fault_probability`**.

... and if all are true, the car sets its **`faulty`** counter to a random integer between 1 and **`max_fault_time`** (both inclusive) and then performs the standard `ADVANCE` operation. Note that the `MakeFaulty` should keep working as before.

Cars are added to the simulation via the following event:

```
[new_vehicle]
time = <NONEG-INTEGER>
id = <VEHICLE-ID>
itinerary = <JUNC-ID>,<JUNC-ID>(,<JUNC-ID>)*
max_speed = <POSITIVE-INTEGER>
type = car
resistance = <POSITIVE-INTEGER>
fault_probability = <NONEG-DOUBLE>
max_fault_duration = <POSITIVE-INTEGER>
seed = <POSITIVE-LONG>
```

Note. The `type = car` keyword-value combination is only valid within this kind of vehicle, and must be included in all its vehicle reports. Key `fault_probability` is a real number between 0 and 1 (both inclusive). Key `seed` is explained below, it is optional and its default value is `System.currentTimeMillis()`.

IMPORTANT. Since this kind of cars has some random behavior, it will be difficult to check that the output produced by your program on the examples that we provide matches the expected results. To solve this problem we will use a *random number generator* with a given *seed* as follows:

- This kind of cars will receive a `seed` (a number of type `long`) as a parameter, and they will store `new Random(seed)` in an instance field, e.g., `_rand`;
- use `_rand.nextDouble()` to generate a random number x between 0 and 1; and
- Use `_rand.nextInt(max_fault_time) + 1` to generate an integer between 1 and `max_fault_time`, including both endpoints.

When testing your program we will provide the same `seed` that we used to generate the output of the provided examples, this way the output should match.

Bikes

Since bikes are less likely to get faulty when they are ridden at low speed, their `MakeFaulty` will not be applied to bikes that travel at a speed that is slower or equal than half of their maximum speed. Bikes are added to the simulation via the following event:

```
[new_vehicle]
time = <NONEG-INTEGER>
id = <VEHICLE-ID>
itinerary = <JUNC-ID>,<JUNC-ID>(,<JUNC-ID>)*
max_speed = <POSITIVE-INTEGER>
type = bike
```

Note. The `type = bike` keyword-value combination is only valid within this kind of vehicle, and must be included in all its vehicle reports.

Roads

There are 2 new kind of roads: Lanes and Dirt.

Lanes

Lane roads are multi-lane roads that support higher traffic than single-lane roads. This results in higher speeds for vehicles even if there are multiple broken cars ahead: their `reduction_factor` is 1 as long as there are more lanes than faulty vehicles ahead (i.e., `numOfLanes>numOfObstacles`) and 2 otherwise. The `base_speed` of the road also takes the number of lanes into account and is defined as $\text{mín}(m, \frac{m*l}{\text{máx}(n,1)} + 1)$ where m is the maximum speed, n is the number of vehicles on the road, and l is the number lanes. **Note that the division is an integer division**.

Lane roads are added using the following event:

```
[new_road]
time = <NONEG-INTEGER>
id = <ROAD-ID>
src = <JUNC-ID>
dest = <JUNC-ID>
max_speed = <POSITIVE-INTEGER>
length = <POSITIVE-INTEGER>
type = lanes
lanes = <POSITIVE-INTEGER>
```

Note. The `type = lanes` keyword-value combination is only valid within this kind of road, and must be included in all its road reports.

Dirt

Dirt roads are just like basic roads, but are even worse to drive through if there are any broken cars in the way, as their `reduction_factor` is calculated as 1 plus *the number of faulty vehicles ahead*. Their `base_speed` is defined as the *maximum speed* of the road.

Dirt roads are added using the following event:

```
[new_road]
time = <NONEG-INTEGER>
id = <ROAD-ID>
src = <JUNC-ID>
dest = <JUNC-ID>
max_speed = <POSITIVE-INTEGER>
length = <POSITIVE-INTEGER>
type = dirt
```

Note. The `type = dirt` keyword-value combination is only valid within this kind of road, and must be included in all its road reports.

Junctions

There are 2 new kind of junctions: Round-robin and Most-crowded.

Round-robin

Round-robin junctions keep green lights on for longer intervals of time, (called *time slices*), instead of incoming switching roads on each tick. Round-robin intersections are configured with the following event:

```
[new_junction]
time = <NONEG-INTEGER>
id = <JUNC-ID>
type = rr
max_time_slice = <POSITIVE-INTEGER>
min_time_slice = <POSITIVE-INTEGER>
```

At the beginning the `time_slice` of each incoming road is set to `max_time_slice` and its corresponding `used_time_slice` to 0. **Note that at the beginning, before**

any call to `ADVANCE`, all lights are red. When the intersection is asked to update its lights, it first checks if the current junction with a green light has consumed its slice (i.e., `used_time_slice` is equal to `time_slice`). Should this be the case, the intersection will:

1. turn the current green light to red;
2. update the `time_slice` of this road as follows:
   a) if the time slice **has been fully used** (for each tick, a vehicle from the corresponding road has passed the junction), then `time_slice` is set to mín(`time_slice`+ 1, `max_time_slice`).
   b) if the time slice **has not been used at all** (no vehicle has passed during any of the ticks that the light was green), then `time_slice` is set to máx(`time_slice`− 1, `min_time_slice`).
   c) otherwise, `time_slice` keeps its current value.
3. sets `used_time_slice` to 0;
4. turns the light of the next incoming road (in the order in which they were added to the junction) to green. Note that when all light are read, the next incoming road is the one that was added first to the junction.

The vehicles in the queue are ordered according to the arrival order (first in first out).

Reports for these junctions are as the default junctions, but for each green light it should also display the remaining time units in green as follows:

```
[junction_report]
id = j2
time = 5
type = rr
queues = (r1,red,[]),(r2,green:3,[v3,v2,v5]),(r3,red,[v1,v4])
```

Most-Crowded

When a junction using this policy is asked to update the lights, the next road to receive a green light is chosen to be the one with the longest queue. If a set of incoming roads have vehicle queues that are all tied for longest length, the order in which roads were added is used to break ties.

At the beginning the `time_slice` and `used_time_slice` is set to 0. When the intersection is asked to update its lights, it first checks if the current junction with a green light has consumed its slice (i.e., `used_time_slice` is equal to `time_slice`). Should this be the case, the intersection will:

1. turns the current green light to red (if any);
2. the incoming road with most vehicles in the queue (and different from the last one with green light, if there are several roads) is given a green with máx($\frac{n}{2}$, 1) as a `time_slice`, where n is the number of vehicles in the corresponding queue, and `used_time_slice` is set to 0. **Note that the division is an integer division**. Recall that if there is more than one incoming road with the same (maximum) number of vehicles in the queue, then the one that was added to the junction first is taken.

The vehicles in the queue are ordered according to the arrival order (first in first out). **Note that at the beginning, before any call to `ADVANCE`, the all lights are red**.

Events that define new most-crowded junctions have the following format:

```
[new_junction]
time = <NONEG-INTEGER>
id = <JUNC-ID>
type = mc
```

The report for this kind of junction is as the one of a round-robin junction, except the value of key `type` which should be `mc`.

Further comments and requirements

- Handle all errors, such as adding simulated objects with the same identifier, refereeing to simulated objects that do not exist, valid values for parameters, etc. Errors should throw appropriate exceptions.
- Pick collections carefully, taking into account how they will be used.
- You will be provided with a set of event file examples with corresponding output that was generated using our implementation. You should check that your output on those example matches the provided output. The skeleton `Main` class includes some `test` methods that can help you to do so.