Assignment 1: Traffic Simulator

PDF generated on: 29-1-2022 19:38:13

Submission date: March 7th, 2022, 09:00am

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

1. Copy detection

For each of the TP2 assignments, all the submissions from all the different TP2 groups will be checked using anti-plagiarism software, by comparing them all pairwise and by searching to see if any of the code of any of them is copied from other sources on the Internet[1] or previous years assignment. Any plagiarism detected will be reported to the Comité de Actuación ante Copias which, after interviewing the student or students in question, will decide whether further action is appropriate and if so, will propose one of the following sanctions:

- A grade of zero for the TP2-course exam session (*convocatoria*) to which the assignment belongs.
- A grade of zero for all the TP2-course exam sessions (*convocatorias*) for that year.
- Opening of disciplinary proceedings (*apertura de un expediente académico*) with the relevant university authority (*Inspección de Servicios*).

2. General instructions

The following instruction are strict, you **MUST** follow them.

1. Download the Java project template that we provide in the Campus Virtual. You must develop your assignment using this template.

[1] If you decide to store your code in a remote repository, e.g. in a free version-control system with a view to facilitating collaboration with your lab partner, make sure your code is not in reach of search engines. If you are asked to provide your code by anyone other than your course lecturer, e.g. an employer of a private academy, you must refuse.

2. Fill in your name(s) in the file "NAMES.txt", each member in a separate line.
3. You have to strictly follow the package structure and class names that we suggest in the assignment description.
4. Submit a **zip** file of the project's directory, including all subdirectories except the **bin** directory. **Other formats (e.g., 7zip, rar, etc.) are not accepted**.

3. Parsing and creating JSON data in Java

JavaScript Object Notation[2] (JSON) is a very common open-standard file format, it uses human-readable text to transmit data objects consisting of attribute-value pairs and array data types. We will use JSON for input and output in the simulator.

Briefly, a JSON structure is a structured text of the form:

$$\{\ \text{"key}_1\text{"}:\ \ \text{value}_1,\ \ldots,\ \text{"key}_n\text{"}:\ \ \text{value}_n\ \}$$

where key_i is a sequence of characters (representing a key) and value_i is a valid JSON value: a number, a string, another JSON structure, or an array $[o_1,\ldots,o_k]$ where o_i is a valid JSON value. Here is an example:

```
{
  "type" : "new_vehicle",
  "data" : {
     "time"      : 1,
     "id"        : "v1",
     "maxspeed"  : 100,
     "class"     : 3,
     "itinerary" : ["j3", "j1", "j5", "j4"]
   }
}
```

In the directory lib we included a library for parsing JSON and converting it into Java objects that are easy to manipulate (it is already imported into the project). You can also use it to create JSON structures and convert them to strings. An example of using this library is available in the package "extra.json".

In order to compare the output of your implementation (on the examples we provide) to the expected output, you can use the following online semantic JSON comparison tool: http://www.jsondiff.com. In addition, the example in package "extra.json" includes another way to compare two JSON structures using the library directly. Note that two JSON structures are considered semantically equal if they have the same set of keys and corresponding values, it is not required that they are syntactically identical. We also provide you with a program that runs your assignment on a set of examples and compares the output to the expected one – see Section 8.

4. Traffic Simulator Overview

Computers allow us to model the world. Simulations can run these models and help us to understand them better;, if we are lucky, apply this understanding to the real world.

[2] https://en.wikipedia.org/wiki/JSON

The assignments of TP2 will ask you to build and refine a traffic simulator, which will model vehicles, roads, and junctions, and manages contamination. You will be able to test different junction policies, different roads with different ways of managing contamination, and vehicles with different contamination classes. 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); a discrete-time counter that is incremented in each simulation step. A simulation step performs the following operations:

1. Process pre-scheduled *events* that operate on the simulated objects;
2. Advance the state of currently simulated objects, according to their behaviors.

Events are read from a file before the simulator starts; 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 the standard output.

The description of the assignment is done as follows:

- in Section 5 we describe the model part, i.e., the components that define the logic of the traffic simulator.
- in Section 6 we describe the control part, i.e., the components that make it possible for a user to interact with the model classes easily.
- in Section 7 we describe the main class.

5. Model

In this part we describe the classes that define the logic of the traffic simulator: in Section 5.1 we describe the different classes required to model junctions, roads, and vehicles; in Section 5.2 we describe a class that will be used to group all simulated objects, that we refer to as *the road-map*; in Section 5.3 we describe the different classes for events (to create vehicles, junction, roads, and change some of their properties); in Section 5.4 we describe the traffic simulator class, which is the one responsible managing the simulation.

5.1. Simulated Objects

There are 3 types of simulated objects:

- vehicles, which can drive through roads and contaminate by emitting CO_2;
- one-way roads on which vehicles travel, and they manage the speed of vehicles to reduce contamination, etc.;
- 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, and can report on their state. Each simulated object will be asked to update its state exactly once per simulation tick. All simulated object extends (directly or indirectly) the following class:

```
package simulator.model;

public abstract class SimulatedObject {

  protected String _id;

  SimulatedObject(String id) {
    if ( id == null || id.length() == 9)
      throw new IllegalArgumentException("the 'id' must be a nonempty
          string.");
    else
      _id = id;
  }

  public String getId() {
    return _id;
  }

  @Override
  public String toString() {
    return _id;
  }

  abstract void advance(int time);
  abstract public JSONObject report();
}
```

5.1.1. Vehicles

In this assignment, we will have only one kind of vehicle, represented by a class called Vehicle that extends SimulatedObject and placed package "simulator.model".

Class Vehicle maintains (at least) the following information as *instance fields* in its state (recall that it is forbidden to declare fields as public):

- *itinerary* (of type List<Junction>): a list of junctions representing the vehicle's itinerary.
- *maximum speed* (of type int): the maximum speed at which the vehicle can travel.
- *current speed* (of type int): the current speed at which the vehicle is traveling.
- *status* (of enum type VehicleStatus – see package "simulator.model"): the status of the vehicle, which can be *Pending* (did not enter the first road yet), *Traveling* (on a road), *Waiting* (at a junction), or *Arrived*.
- *road* (of type Road): the road on which it is currently traveling, which should be null if it is not on any road.
- *location* (of type int): the vehicle's position on the road on which it is traveling (the distance from the beginning of the road, i.e., point 0).
- *contamination class* (of type int): a number between 0 and 10 (both inclusive) that is used to calculate the CO_2 emitted by the vehicle in each simulation step.

- *total contamination* (of type int): the total CO_2 emitted by the vehicle so far.
- *total traveled distance* (of type int): the total distance traveled by the vehicle.

Class Vehicle has only one *package protected* constructor:

```
Vehicle(String id, int maxSpeed, int contClass, List<Junction>
    itinerary) {
  super(id);
  // TODO complete
}
```

In this constructor, you should check that the arguments have valid values and throw corresponding exceptions otherwise: (a) id is nonempty string; (b) maxSpeed should be positive; (c) contClass should be between 0 and 10 (both inclusive); (d) the length of the list itinerary should be at least 2. Besides, do not store list itinerary as it is received by the construction, but rather copy it into a new *read-only* list (to avoid modifying it from outside) as follows:

```
Collections.unmodifiableList(new ArrayList<>(itinerary));
```

Class Vehicle has the following methods (you should maintain visibility modifiers as described below – no modifier means package protected):

- void setSpeed(int s): sets the current speed to the minimum between (i) s; (ii) the vehicle's maximum speed. It should throw a corresponding exception if s is negative.
- void setContaminationClass(int c): sets the contamination class of the vehicle to c. It should throw a corresponding exception if c is not between 0 and 10 (both inclusive).
- void advance(int time): if the vehicle's status is not *Traveling*, it does nothing, otherwise:
  (a) its location is updated to the minimum between (i) *the current location* plus *the current speed*; (ii) the length of the current road;
  (b) calculate the contamination produced as $c = l * f$ where f is the contamination factor and l is the distance traveled in this simulation step, i.e., the new location minus the previous location, and then add c to the vehicle's *total contamination* and also to the current road contamination by calling the corresponding method of class Road.
  (c) if the current location (i.e., the new one) equals to the road length, the vehicle enters the queue of the corresponding junction (by calling the corresponding method of class Junction). Remember to modify the vehicle's status.
- void moveToNextRoad(): moves the vehicle to the next road. This is done by *exiting* the current road and *entering* the next road at location 0 of its itinerary – exiting and entering roads is done by calling the corresponding methods of class Road. To find the next road, it should ask the junction at which the vehicle is waiting now (or the first one of the itinerary in case of a *Pending* status) by calling the corresponding method. Note that the first time this method is called it does not exit any road as it has not entered any yet and that when the vehicle exits the last junction of its itinerary it does not enter any road – recall modifying the vehicle's status. The

method should throw a corresponding exception if the vehicle's status is not *Pending* or *Waiting*. **Recall that the itinerary list is *read-only*, and thus you cannot modify it**. It is convenient to keep an index indicating the last junction that has been visited in the itinerary.

- public JSONObject report(): returns the vehicle's state in the following `JSON` format:

```
{
  "id" : "v1",
  "speed" : 20,
  "distance" : 60,
  "co2": 100,
  "class": 3,
  "status": "TRAVELING",
  "road" : "r4",
  "location" : 30
}
```

where "`id`" is the vehicle's identifier; "`speed`" is its current speed; "`distance`" is the total distance traveled by the vehicle; "`co2`" is the total CO_2 emitted by the vehicle; "`class`" is the contamination class of the vehicle; "`status`" is the status of the vehicle which can be "`PENDING`", "`TRAVELING`", "`WAITING`", or "`ARRIVED`"; "`road`" is the identifier of the road on which it is traveling; "`location`" is its location on that road. If the status of the vehicle is *Pending* or *Arrived*, keys "`road`" and "`location`" should be omitted from the report.

Besides, define the following getters to retrieve corresponding information: getLocation(), getSpeed(), getMaxSpeed(), getContClass(), getStatus(),getTotalCO2(), getItinerary(), getRoad(). You can define additional setters as far as they are *private*. **Make sure that the speed of the vehicle is** 0 **when its status is not *Traveling***.

5.1.2. Roads

In this assignment, we will have two kinds of roads. The difference between the kinds of roads is in the way they deal with contamination. We will first describe a base-class Road (in package "simulator.model"), that extends SimulatedObject, and then use inheritance to define two kinds of roads.

Class Road maintains (at least) the following information as *instance fields* in its state (recall that it is forbidden to declare fields as public):

- *source junction* and *destination junction* (both of type Junction): the junctions to which the road is connected. Traveling is in one direction, from *source* to *destination*.
- *length* (of type int): the length of the road (in some distance units, let us say meters).
- *maximum speed* (of type int): the maximum speed allowed by this road.
- *current speed limit* (of type int): the current speed limit, i.e., a vehicle cannot travel on this road at speed higher than this limit. Its initial value should be equal to the maximum speed.

- *contamination alarm limit* (of type int): indicates a limit after which restrictions may be applied to reduce contamination.
- *weather conditions* (of enum type Weather – see package "simulator.model"): the weather conditions on this road. This value is used for calculating how contamination vanishes, calculating the speed limit, etc.
- *total contamination* (of type int): the total accumulated contamination of the road. I.e., the total CO_2 emitted by the vehicles that traveled on the road so far.
- *vehicles* (of type List<Vehicle>): a list of the vehicles that are currently traveling on the road – **should always be sorted by vehicle location (descending order)**. 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 – recall that the sorting algorithms of Java guarantee that equal elements will not be reordered as a result of the sort.

Class Road has only one *package protected* constructor:

```
Road(String id, Junction srcJunc, Junction destJunc, int maxSpeed,
    int contLimit, int length, Weather weather) {
  super(id);
  // ...
}
```

The constructor should add the road as an incoming road to its destination junction, and as an outgoing road of its source junction. In this constructor, you should check that arguments have valid values and throw corresponding exceptions otherwise: maxSpeed is positive; contLimit is non-negative; length is positive; srcJunc, destJunc and weather is not null.

Class Road has the following methods (you should maintain visibility modifiers as described below – no modifier means package protected):

- void enter(Vehicle v): is used by vehicles to enter a given road. It simply adds the vehicle to the corresponding vehicles list (at the end). It should check that following hold and throw a corresponding exception otherwise: the vehicle's location is 0; the vehicle's speed is 0.
- void exit(Vehicle v): used by vehicles to exit the road. It simply removes the vehicle from the vehicles list.
- void setWeather(Weather w): sets the road's weather to w. It should check that w is not null and throw a corresponding exception otherwise.
- void addContamination(int c): add c units of CO_2 to the total contamination of the road. It should check that c is non-negative and throw a corresponding exception otherwise.
- abstract void reduceTotalContamination(): an abstract method for reducing the total contamination. The specific behavior is defined in subclasses.
- abstract void updateSpeedLimit(): an abstract method for updating the speed limit of the road. The specific behavior is defined in subclasses.

- abstract int calculateVehicleSpeed(Vehicle v): an abstract method for calculating the speed for vehicle v. The specific behavior is defined in subclasses.
- void advance(int time): advances the state of the road as follows:
  (1) calls reduceTotalContamination to reduce the total contamination, i.e., some CO_2 vanish.
  (2) calls updateSpeedLimit to set the speed limit for the current simulation step.
  (3) traverses the list of vehicles (from first to last), and for each vehicle: (a) sets the vehicle's speed to the value returned by calculateVehicleSpeed; (b) calls the vehicle's advance method. Recall sorting the list of vehicles by location at the end.
- public JSONObject report(): returns the road's state in the following JSON format:

```
{
 "id" : "r3",
 "speedlimit" : 100,
 "weather" : "SUNNY",
 "co2" : 100,
 "vehicles" : ["v1","v2",...],
}
```

  where "id" is the road's identifier; "speedlimit" is the current speed limit; "weather" is the current weather conditions; "co2" is the total contamination; "vehicles" is a list of vehicles identifiers traveling on the road (in the same order as store in the corresponding list).

Besides, define the following *public getters* to retrieve corresponding information: getLength(), getDest(), getSrc(), getWeather(), getContLimit(), getMaxSpeed(), getTotalCO2(), getSpeedLimit(), getVehicles(). Note that getVehicles() should return the list of vehicles as a *read-only* list (i.e., it returns Collections.unmodifiableList(vehicles) where vehicles is the list of vehilces). You can define additional setters as far as they are *private*.

Next, we describe the two kinds of roads that should be implemented.

5.1.2.1 Inter-City Road

This kind of road is used to connects cities, and is implemented by a class called InterCityRoad that extends Road and placed in the package "simulator.model". Its behavior is as follows:

- method reduceTotalContamination reduces the total contamination to the value of "((100 - x) * tc) / 100)" where tc is the current total contamination and x depends on the weather conditions: 2 in case of SUNNY weather, 3 in case of CLOUDY weather, 10 in case of RAINY weather, 15 in case of WINDY weather, and 20 in case of a STORM.
- method updateSpeedLimit sets the speed limit to 50% of the maximum speed (i.e., to "maxSpeed*/2") if the total contamination exceeds the contamination alarm limit, and to the maximum speed otherwise.
- method calculateVehicleSpeed calculates the speed of a vehicle as to the speed limit of the road but reduced by 20% (i.e., "(speedLimit*8)/10") in case of a STORM.

5.1.2.2 City Road

This kind of road models roads inside cities, and is implemented by a class called CityRoad that extends Road and placed in the package "simulator.model".

It is more restrictive when it comes to contamination. It never reduces the speed limit (it is always the maximum speed), but rather calculates the speed of each vehicle depending on its contamination class. Moreover, the reduction of contamination does not depend much on the weather as before. Its behavior is as follows:

- method reduceTotalContamination reduces the total contamination by x CO_2 units where x depends on the weather conditions: 10 in case of WINDY weather or a STORM, and 2 otherwise. Make sure that total contamination does not become negative.
- the speed limit does not change, it is always the maximum speed.
- method calculateVehicleSpeed calculates the speed of a vehicle using the expression "((11-f)*s)/11", where s is the speed limit of the road and f is the contamination class of the vehicle.

5.1.3. Junctions

Junctions in the simulation manage incoming roads using traffic lights, which can be *green* to allow traffic to enter (and then leave to the corresponding next road) or *red* to make the vehicles from that road wait. Each incoming road will have a queue in which the vehicles arriving from that road wait until it gets a green light. At any given time, exactly one incoming road can have a *green* light.

There are several kinds of junctions, and they can be different in (a) how they decide which incoming road gets a green light; (b) how they remove vehicles from the queue of the road with green lights, i.e., which vehicles from the queue move to their next road in each simulation step.

We will not use inheritance to define junctions, but rather we will use composition which can be seen as composing an object with an algorithm (also knows as a strategy) to perform a specific task. These algorithms are encapsulated in a different class hierarchy. In what follows: we describe how to encapsulate algorithms for switching traffic lights; how to encapsulate algorithms for removing vehicles from the queues; finally describe the class Junction that uses these algorithms.

5.1.3.1 Light Switching Strategies

This strategy deals with deciding which incoming road gets a green light. It is modeled by the following interface in the package "simulator.model":

```
package simulator.model;

public interface LightSwitchingStrategy {
  int chooseNextGreen(List<Road> roads, List<List<Vehicle>> qs, int
      currGreen, int lastSwitchingTime, int currTime)
}
```

which has a single method chooseNextGreen that receives:

- roads: the list of the incoming roads (of a junction).
- qs: a list of lists of vehicles, where the (inner) lists of vehicles represent queues. The i-th queue corresponds to the i-th road in the list roads. Note that we use the type List<Vehicle> and not Queue<Vehicle> to represent a queue since the interface Queue does not guarantee any order when the collection is iterated (and we want to iterate them in the order in which the elements were added).
- currGreen: the index (in list roads) of the road with a green light. The value -1 is used to indicate that all are red.
- lastSwitchingTime: the simulation time at which the light for the road currGreen was switched from *red* to *green*. If currGreen is -1 then it is the last time all switched to red.
- currTime: the current simulation time.

The method returns the index of the road (in the list roads) to be switched to *green* – if it is the same as currGreen then, the junction will not consider it as switching. If it returns -1 it means all should be red.

We have two light switching strategies RoundRobinStrategy and MostCrowdedStrategy (in package "simulator.model"). Both receive in the constructor a single parameter called timeSlot (of type int), which is the number of consecutive ticks during which a road can have a green light. Next, we describe the behavior of these two strategies.

Strategy RoundRobinStrategy behaves as follows:

- if the list of roads is empty, return -1.
- if the lights of all incoming roads are red (i.e., currGreen is -1), it gives green to the first one of in list of roads (i.e., it returns 0); otherwise
- if "currTime-lastSwitchingTime $<$ timeSlot", it keeps the lights as they are (i.e., it returns currGreen); otherwise
- it returns currGreen+1 modulo the length of the list roads (i.e., the index of the next incoming road circularly traversing the list).

Strategy MostCrowdedStrategy behaves as follows:

- if the list of roads is empty, return -1.
- if the lights of all incoming roads are red, it gives green to the incoming road with the largest *queue*, starting the search (in qs) from position 0. If there is more than one with the same maximal size, it picks the first one that it finds during the search.
- if the "currTime-lastSwitchingTime $<$ timeSlot", it keeps the lights as they are (i.e., it returns currGreen); otherwise
- it gives green to the incoming road with the largest *queue*, starting a circular search (in qs) from position currGreen+1 modulo the number of incoming roads. If there is more than one with the same maximal size, it picks the first one that it finds during the search. Note that it might return currGreen.

5.1.3.2 Dequeuing Strategies

This strategy deals with removing vehicles from the queue of the road with a green light. It is modeled by the following interface in the package "simulator.model":

```
package simulator.model;

public interface DequeuingStrategy {
  List<Vehicle> dequeue(List<Vehicle> q);
}
```

which has a single method that returns a list of vehicles (from q) that should be asked to move to their next roads. The method should not modify the queue q nor ask the vehicles to move, this is the task of the class Junction.

We have two dequeuing strategies (in the package "simulator.model"):

- Strategy MoveFirstStrategy returns a list that include the first vehicle of q.
- Strategy MoveAllStrategy returns a list of all vehicles that are in q (do not return q) itself. The order should be the same as when iterating q.

5.1.3.3 Class Junction

The functionality of a junction is implemented by a class Junction, that extends SimulatedObject and is placed in the package "simulator.model".

Class Junction maintains (at least) the following information as *instance fields* in its state (recall that it is forbidden to declare fields as public):

- *list of incoming roads* (of type List<Road>): a list of all roads that enter the junction, i.e., the junction is their destination.
- *map of outgoing roads* (of type Map<Junction,Road>): a map of outgoing roads, i.e., if (j,r) is a pair of key-value in the map, then the current junction is connected to junction j by road r. This is used to know which road to take to arrive at junction j.
- *list of queues* (of type List<List<Vehicle>>): a list of queues for incoming roads – the i-th queue (represented as List<Vehicle>) corresponds to the i-th road in the *list of incoming roads*. It is also recommended to keep a road-queue map (of type Map<Road,List<Vehciles>) to make the search for the queue of a given road more efficient.
- *green light index* (of type int): the index of the incoming road (in the list of incoming roads) that currently has a green light. The value -1 is used to indicate that all incoming roads have a red light.
- *last switching time* (of type int): the time at which the *green light index* has been changed value. Its initial value is 0.
- *light Switching strategy* (of type LightSwitchingStrategy): a strategy for switching lights.
- *dequeuing strategy* (of type Dequeuing strategy): a strategy for removing vehicles from queues.

- *x and y coordinates* (of type int): coordinates that will be used for drawing the junction in the next assignment.

Class Junction has only one *package protected* constructor:

```
Junction(String id, LightSwitchStrategy lsStrategy, DequeingStrategy
    dqStrategy, int xCoor, int yCoor) {
  super(id);
  // ...
}
```

Note that it receives strategies as parameters. It should check that lsStrategy and dqStrategy are not null, and that xCoor and yCoor are non-negative and throw corresponding exceptions otherwise.

Class Junction has the following methods (you should maintain visibility modifiers as described below – no modifier means package protected):

- void addIncommingRoad(Road r): adds r at the end of the list of incoming roads, and creates a queue (an instance of LinkedList for example) for r and adds it at the end of the list of queues. In addition, if you have used a road-queue map then add a corresponding pair to the map. It should check that road r is indeed an incoming road, i.e., its destination junction is equal to the current junction, and throw a corresponding exception otherwise.
- void addOutGotingRoad(Road r): adds a pair (j,r) to the map of outgoing roads, where j is the destination junction of road r. It should check that no other road goes to j, and that road r is indeed an outgoing road and throw corresponding exceptions otherwise.
- void enter(Vehicle v): adds vehicle v to the queue of road r, where r is the road on which v is travelling.
- Road roadTo(Junction j): returns the road that goes from the junction to junction j. For this you should look in the map of outgoing roads.
- void advance(time): advances the junction's state as follow:
  1. uses the *dequeuing strategy* to compute a list of vehicles that should be advanced, and then ask these vehicles to move to their next roads and remove them from the corresponding queue.
  2. uses the *light switching strategy* to compute the index of the next road to get a green light. If it is different from the current *green light index*, then change the value of the *green light index* to the new one and set the *last switching time* to the current time (i.e., the value of the parameter time).
- public JSONObject report(): returns the junction's state in the following JSON format:

```
{
 "id" : "j3",
 "green" : "r1",
 "queues" : [Q1,Q2,....]
}
```

where "id" is the junction's identifier; "green" is the identifier of the road with a green light ("none" if all are red); "queues" is a list of queues of the incoming roads, where each Qi is in the following JSON format

```
{
 "road"  : "r3",
 "vehicles" : ["v1","v2",....]
}
```

where "road" is the road identifier, and "vehicles" is the list of vehicles in the corresponding queue (the order should be the same as when iterating the queue).

5.2. Road-Map

The purpose of this class is to group all the simulated objects. This facilitates the work of the simulator. It is implemented as a class RoadMap and is placed in the package "simulator.model".

Class RoadMap maintains (at least) the following information as *instance fields* in its state (recall that it is forbidden to declare fields as public):

- *list of junctions* of type List<Junction>.
- *list of roads* of type List<Road>.
- *list of vehicles* of type List<Vehicle>.
- *junctions map* (of type Map<String,Junction>): a map from junction identifiers to the corresponding objects.
- *roads map* (of type Map<String,Road>: a map from road identifiers to the corresponding objects.
- *vehicles map* (of type Map<String,Vehicle>): a map from vehicle identifiers to the corresponding objects.

Note that we keep both lists and maps to: use the maps to search for an object in a more efficient way; and use the lists of to traverse the corresponding objects in the same order in which they have been added.

Class RoadMap has a single *package protected* constructor without arguments, it simply initializes the data-structures mentioned above.

Class RoadMap has the following methods (you should maintain visibility modifiers as described below – no modifier means package protected):

- void addJunction(Junction j): adds junction j at the end of the list of junctions, and modifies the junction's map accordingly. It should check that there is no other junction with the same identifier.
- void addRoad(Road r): adds road r at the end of the list of roads, and modifies the road's map accordingly. It should check that the following hold and throw corresponding exceptions otherwise: (i) there is no other road with the same identifier; (ii) the junctions that connect the road exists in the road-map.

- void addVehicle(Vehicle v): adds vehicle v at the end of the list of vehicles, and modifies the vehicle's map accordingly. It should check that the following hold and throw corresponding exceptions otherwise: (i) there is no other vehicle with the same identifier; (ii) the itinerary is valid, i.e., there are roads that connect each consecutive junctions of the itinerary.
- public Junction getJunction(Srting id): returns the junction with identifier id, and null if no such junction exists.
- public Road getRoad(Srting id): returns the road with identifier id, and null if no such road exists.
- public Vehicle getVehicle(Srting id): returns the vehicle with identifier id, and null if no such vehicle exists.
- public List<Junction> getJunctions(): returns a *read-only* version of the list of junctions.
- public List<Roads> getRoads(): returns a *read-only* version of the list of roads.
- public List<Vehicle> getJunctions(): returns a *read-only* version of the list of vehicles.
- void reset(): clears all lists and maps.
- public JSONObject report(): return the road-map's state in the following `JSON` format:

```
{
  "junctions" : [J1Report,J2Report,...],
  "road" : [R1Report,R2Report,...],
  "vehicles" : [V1Report,V2Report,...],
}
```

where `JiReport`, `RiReport`, and `ViReport`, are the reports of the corresponding simulated objects. The order in the `JSON` lists should be like the order of the corresponding lists of RoadMap.

5.3. Events

Events allow us to set up and interact with the simulator, by adding vehicles, roads, and junctions; changing the weather conditions of roads; changing the contamination class of vehicles. Each event has a time at which it should be executed. 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. Executing an event is done by calling a corresponding method.

We first define an abstract base-class Event (in package "`simulator.model`" to model an event:

```
package simulator.model;

public abstract class Event implements Comparable<Event> {

  protected int _time;
```

```
  Event(int time) {
    if ( time < 1 )
      throw new IllegalArgumentException("Invalid time: "+time);
    else
      _time = time;
  }

  int getTime() {
    return _time;
  }

  @Override
  public int compareTo(Event o) {
    // TODO complete the method to compare events according to their _time
  }

  abstract void execute(RoadMap map);
}
```

Field "_time" is the time at which this event should be executed, and method execute is the one called by the simulator to execute the event. The functionality of this method is defined in subclasses.

In what follows we describe the supported events, all extend class Event and should be placed in the package "simulator.model". The constructor of each event receives some data to perform an operation when requested, which is stored in fields and used in method execute to perform the corresponding functionality.

5.3.1. New Junction Event

This event is implemented by a class NewJunctionEvent. It has a single constructor:

```
public NewJunctionEvent(int time, String id, LightSwitchingStrategy
    lsStrategy, DequeuingStrategy dqStrategy, int xCoor, int yCoor) {
  super(time);
  // ...
}
```

Method execute of this event creates a corresponding junction and adds it to the road-map (the parameter of method execute).

5.3.2. New Roard Events

There are two events for creating roads: NewCityRoadEvent and NewInterCityRoadEvent. Each has a single constructor:

```
public NewCityRoad(int time, String id, String srcJun, String
    destJunc, int length, int co2Limit, int maxSpeed, Weather weather)
     {
  super(time);
  // ...
}
```

```
public NewInterCityRoad(int time, String id, String srcJun, String
    destJunc, int length, int co2Limit, int maxSpeed, Weather weather)
     {
  super(time);
  // ...
}
```

Method execute of these events creates a corresponding road and adds it to the road-map.

These two events have much in common, and thus you will end up duplicating code. After implementing and testing these two events, refactor them by introducing a superclass NewRodEvent that includes the common parts, etc.

5.3.3. New Vehicle Event

This event is implemented by a class NewVehicleEvent. It has a single constructor:

```
public NewVehicleEvent(int time, String id, int maxSpeed, int
    contClass, List<String> itinerary) {
  super(time);
  // ...
}
```

Method execute of this event creates a corresponding vehicle and adds it to the road-map, and then calls its moveToNext method to start its journey.

5.3.4. Set Weather Event

This event is implemented by a class SetWeatherEvent. It has a single constructor:

```
public SetWeatherEvent(int time, List<Pair<String,Weather>> ws) {
  super(time);
  // ...
}
```

It should check that ws is not null and throw a corresponding exception otherwise. Method execute traverses the list ws, and for each element w sets the weather of the road with identifier w.getFirst() to w.getSecond() (see package "simulator.misc" for the code of class Pair). It should throw an exception if the road does not exist in the road-map.

5.3.5. Set Contamination Class Event

This event is implemented by a class NewSetContClassEvent. It has a single constructor:

```
public NewSetContClassEvent(int time, List<Pair<String,Integer> cs) {
  super(time);
  // ...
}
```

It should check that cs is not null and throw a corresponding exception otherwise. Method execute traverses the list cs, and for each element c sets the contamination class of the vehicle with identifier c.getFirst() to c.getSecond(). It should throw an exception if the vehicle does not exist in the road-map.

5.4. The Simulator Class

The simulator class is the one responsible for performing the simulation. It is implemented by a class TrafficSimulator in the package "simulator.model".

Class TrafficSimulator maintains (at least) the following information as *instance fields* in its state (recall that it is forbidden to declare fields as public):

- *road-map* of type (of type RoadMap): a road-map in which all simulated objects are stored.
- *list of events* (of type List<Event>): a list of events to be executed, the list is sorted by the time of the events. If two events have the same time, the one that was added first goes before in the list – to guarantee this use the class SortedArrayList that was explained in class, and is provided in package "simulator.misc".
- *simulation time* (of type int): the simulation time, initialized to 0.

Class TrafficSimulator has only one public constructor without any parameter, it simply initializes the fields.

Class TrafficSimulator has the following methods (you should maintain visibility modifiers as described below – no modifier means package protected):

- public void addEvent(Event e): adds the event e to the list of events. Recall that the list of events should be maintained sorted as described above.
- public void advance(): it advances the state of the simulation as follows (**the order is important!**):
  1. increments the *simulation time* field by one.
  2. executes all events whose time is as current the *simulation time* and removes it from the list.
  3. calls method advance of all junctions.
  4. calls method advance of all roads.
- public void reset(): clears the *road-map* and the *list of events*, and sets the *simulation time* to 0.
- public JSONObject report(): return the simulator's state in the following `JSON` format:

```
{
  "time"  : 3,
  "state" : {
              "junctions" : [...],
              "road" : [...],
              "vehicles" : [...]
            }
}
```

  where "`time`" is the current *simulation time*, and "`state`" is what is returned by calling method report() of the *road-map*.

6. Control

Now that we have defined the different classes that form part of the logic of the simulator, we can start testing it by writing a method that creates some events, adds them to the simulator and then calls method advance of the simulator several times to perform the simulation. Although this is adequate for testing the application, it is still not an easy way for users to use the application. The control part aims to provide an easy way for using the application, in particular it allows loading events from text files, automatically creating corresponding events and adding them to the simulator, and perform a given number of simulation steps.

We will use JSON structures to describe events, and we will use factories to parse these structures and convert them to actual objects. In Section 6.1 we describe the factories required to facilitate creating events from JSON structures; in Section 6 we describe the controller, which is the class that allows loading events from an InputStream and executing the simulator for a given number of steps.

6.1. Factories

Since we have several factories, we will use Java generics to avoid duplication of code. Next, we describe how to develop all factories step by step. All classes and interfaces should be placed in the package "simulator.factories".

We will model a factory by a generic interface Factory<T>:

```
package simulator.factories;

public interface Factory<T> {
  public T createInstance(JSONObject info);
}
```

Method createInstance receives JSON structure describing the object to be created, and returns an instance of a corresponding class — an instance of a sub-type of T. If it does not recognize what is described in info, it should throw a corresponding exception.

For our purposes, we require the JSON structure that is passed to createInstance to includes two keys:

- *type*, which is a string describing the type of the object to be created;
- *data*, which is a JSON structure that includes all information needed to create the instance, e.g., what is passed to the corresponding constructor.

There are many ways to define a factory, we will see some during the course. For our purposes, we will use what we call a *builder based factory*, which allows extending a factory with more options without actually modifying its code.

The basic element in a *builder based factory* is the *builder*, which is a class that can handle one case of those provided by the factory, i.e., create an instance of a specific type. We can model it as a generic class Builder<T>:

```
package simulator.factories;

public abstract class Builder<T> {
  protected String _type;
```

```
  public Builder(String type) {
    if ( type == null )
      throw new IllegalArgumentException("Invalid type: "+type);
    else
      _type = type;
  }

  public T createInstance(JSONObject info) {

    T b = null;

    if (_type != null && _type.equals(info.getString("type"))) {
      b = createTheInstance(
            info.has("data") ? info.getJSONObject("data") : new
               JSONObject());
    }

    return b;
  }

  protected abstract T createTheInstance(JSONObject data);
}
```

As can be seen, its method createInstance receives a JSON object, and if it has key "type" whose value is equal to the field _type, it calls the abstract method createTheInstance with the value of key "data" to create the actual object, otherwise it returns null to indicate the it does not recognize this JSON structure. Classes that extend Builder<T> are responsible on assigning value to _type by calling the constructor of class Builder, and also on defining method createTheInstance to create the instance. Later we will describe several builders that we need, but let us first describe how these builders can be used to create a factory.

A *builder based factory* is class that has a list of builders, and when asked to create an object from a corresponding JSON structure it traverses all builders until it finds one the can create the instance:

```
package simulator.factories;

public class BuilderBasedFactory<T> implements Factory<T> {

  private List<Builder<T>> _builders;

  BuilderBasedFactory(List<Builder<T>> builders) {
    _builders = new ArrayList<>(builders);
  }

  @Override
  public T createInstance(JSONObject info) {
    if (info != null) {
      for (Builder<T> bb : _builders) {
        T o = bb.createInstance(info);
        if (o != null) return o;
      }
    }

    throw new IllegalArgumentException("Invalid value for
       createInstance: "+info);
```

```
  }
}
```

Note that the list of builders is received as a parameter by the constructor, which means we can extend the factory by adding more builders to this list.

Next, we describe three factories that we need in this assignment. Builders should return null (or throw corresponding exceptions) if some data is missing in the JSON structure or some keys have invalid values.

6.1.1. Light Switching Strategies Factory

For this factory we need two builders, RoundRobinStrategyBuilder and MostCrowdedStrategyBuilder, both extend Builder<LightSwitchingStrategy> since they create instances of classes that implement LightSwitchingStrategy.

Builder RoundRobinStrategyBuilder creates an instance of RoundRobinStrategy from the following JSON structure:

```
{
   "type" : "round_robin_lss",
   "data" : {
              "timeslot" : 5
            }
},
```

Key "timeslot" is optional, its default value is 1.

Builder MostCrowdedStrategyBuilder creates an instance of MostCrowdedStrategy from the following JSON structure:

```
{
   "type" : "most_crowded_lss",
   "data" : {
              "timeslot" : 5
            }
}
```

Key "timeslot" is optional, its default value is 1.

In both cases above, key "timeslot" is optional, its default value is 1.

Once the above builders are defined, we can use them to create a corresponding factory as follows:

```
List<Builder<LightSwitchingStrategy>> lsbs = new ArrayList<>();
lsbs.add( new RoundRobinStrategyBuilder() );
lsbs.add( new MostCrowdedStrategyBuilder() );
Factory<LightSwitchingStrategy> lssFactory = new BuilderBasedFactory
    <>(lsbs);
```

This factory will be used later to parse strategies for junctions (see the junction event builder below).

6.1.2. Dequeuing Strategies Factory

For this factory we need two builders, MoveFirstStrategyBuilder and MoveAllStrategyBuilder, both extend Builder<DequeuingStrategy> since they create instances of classes that implement DequeuingStrategy.

Builder MoveFirstStrategyBuilder creates an instance of MoveFirstStrategy from the following JSON structure:

```
{
   "type" : "move_first_dqs",
   "data" : {}
},
```

Key "data" can be omitted since it does not include any further information.

Builder MoveAllStrategyBuilder creates an instance of MoveAllStrategy from the following JSON structure:

```
{
   "type" : "most_all_dqs",
   "data" : {}
},
```

Key "data" can be omitted since it does not include any further information.

Once the above builders are defined, we can use them to create a corresponding factory as follows:

```
List<Builder<DequeuingStrategy>> dqbs = new ArrayList<>();
dqbs.add( new MoveFirstStrategyBuilder() );
dqbs.add( new MoveAllStrategyBuilder() );
Factory<DequeuingStrategy> dqsFactory = new BuilderBasedFactory<>(
    dqbs);
```

6.1.3. Events Factory

For this factory we need a builder for each kind of event that we have described in Section 5.3, all extend Builder<Event> since they create instances of classes that extend Event.

Builder NewJunctionEventBuilder creates an instance of NewJunctionEvent from the following JSON structure:

```
{
  "type" : "new_junction",
  "data" : {
    "time" : 1,
    "id"   : "j1",
    "coor" : [100,200],
    "ls_strategy" : { "type" : "round_robin_lss", "data" : {"timeslot" : 5} },
    "dq_strategy" : { "type" : "move_first_dqs",  "data" : {} }
  }
}
```

The "coor" key is a list that contains the x and y coordinates (in this order). Note that its data section includes JSON structures that describe the strategies that should be used. We assume that factories for these strategies are provided to the constructor of this builder, i.e., its constructor should have the following signature (later we will see how to pass these factories to this constructor):

```
public NewJunctionEventBuilder(Factory<LightSwitchingStrategy>
    lssFactory, Factory<DequeuingStrategy> dqsFactory)
```

Builder NewCityRoadEventBuilder creates an instance of NewCityRoadEvent from the following JSON structure:

```
{
  "type" : "new_city_road",
  "data" : {
    "time"     : 1,
    "id"       : "r1",
    "src"      : "j1",
    "dest"     : "j2",
    "length"   : 10000,
    "co2limit" : 500,
    "maxspeed" : 120,
    "weather"  : "SUNNY"
  }
}
```

Builder NewInterCityRoadEventBuilder is supposed to create an instance of NewInterCityRoadEvent from the following JSON structure:

```
{
  "type" : new_inter_city_road
  "data" : {
    "time"     : 1,
    "id"       : "r1",
    "src"      : "j1",
    "dest"     : "j2",
    "length"   : 10000,
    "co2limit" : 500,
    "maxspeed" : 120,
    "weather"  : "SUNNY"
  }
}
```

Note that builder NewCityRoadEventBuilder and NewInterCityRoadEventBuilder have some common code, so you might consider doing some refactoring by introducing a corresponding super-class.

Builder NewVehicleEventBuilder creates an instance of NewVehicleEvent from the following JSON structure:

```
{
  "type" : "new_vehicle",
  "data" : {
    "time"      : 1,
    "id"        : "v1",
    "maxspeed"  : 100,
    "class"     : 3,
    "itinerary" : ["j3", "j1", ...]
  }
}
```

Builder SetWeatherEventBuilder creates an instance of SetWeatherEvent from the following JSON structure:

```
{
  "type" : "set_weather",
  "data" : {
    "time"     : 10,
    "info"     : [ { "road" : r1, "weather": "SUNNY" },
                   { "road" : r2, "weather": "STORM" },
                   ...
                 ]
  }
}
```

Builder SetContClassEventBuilder creates an instance of SetWeatherEvent from the following JSON structure:

```
{
  "type" : "set_cont_class",
  "data" : {
    "time"     : 10,
    "info"     : [ { "vehicle" : v1, "class": 3 },
                   { "vehicle" : v4, "class": 2 },
                   ...
                 ]
  }
}
```

Once the above builders are defined, we can use them to create a corresponding factory as follows:

```
List<Builder<Event>> ebs = new ArrayList<>();
ebs.add( new NewJunctionEventBuilder(lssFactory,dqsFactory) );
ebs.add( new NewCityRoadEventBuilder() );
ebs.add( new NewInterCityRoadEventBuilder() );
// ...
Factory<Event> eventsFactory = new BuilderBasedFactory<>(ebs);
```

6.2. The Controller

The controller is implemented by s class Controller, and should be placed in the package "simulator.control". It is responsible for

- reading the events from a given InputStream and adding them to the simulator;
- executing the simulator a given number of steps, and printing the different states to a given OutputStream.

Class Controller maintains (at least) the following information as *instance fields* in its state (recall that it is forbidden to declare fields as public):

- *traffic simulator* (of type TrafficSimulator): used to perform the simulation.
- *events factory* (of type Factory<Event>): used to parse the events provided by the user.

Class Controller has only one public constructor:

```
public Controller(TrafficSimulator sim, Factory<Event> eventsFactory)
    {
  // TODO complete
}
```

In the constructor, you should check that the values of the arguments are not null and throw corresponding exceptions otherwise.

Class Controller has the following methods:

- public void loadEvents(InputStream in): we assume that in includes (the text of) a JSON structure of the form

  $$\{\ \texttt{"events":}\ \ [e_1,\ldots,e_n]\ \}$$

  where each e_i is a JSON structure that corresponds to an event. This method first converts the input JSON into a JSONObject using

  JSONObject jo = new JSONObject(new JSONTokener(in));

  and then extracts each e_i from jo, creates a corresponding event e using the *events factory*, and adds it to the simulator by calling method addEvent. The method should throw a corresponding exception if the input JSON does not match the one above.

- public void run(int n, OutputStream out): runs the simulator n ticks, by calling advance exactly n times, and prints to the different states to out in the following JSON format:

  $$\{\ \texttt{"states":}\ \ [s_1,\ldots,s_n]\ \}$$

  where s_i is the state of the simulator **after** executing step i. Note that state s_i is obtained by calling method report of the traffic simulator object.

- public void reset(): calls method reset of the traffic simulator.

7. 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-line options:

```
> 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.json -o output.json -t 100
java Main -i eventsfile.json -t 100
```

The first example writes the output to a file `output.json` and the second to the standard output (= the console). In both cases the input events file is `eventsfile.json` and the simulation is executed for 100 ticks.

You will have to complete methods initFactories and startBatchMode, and add a support for the command-line option -t (by studying how it is done already for other options). Note that it is an optional argument, i.e., if it is not provided in the command-line then its value should be 10.

8. Other Comments

- Directory resources/examples includes a set of examples, and corresponding expected output, that you can use to test your assignment (see resources/examples/README.md for more details). We might provide more examples before the deadline. Your assignment must pass all these tests.
- To convert a String s to a corresponding enum value, e.g., of type Weather, use Weather.valueOf(s) – or better use s.toUpperCase() to avoid case-sensitivity problems, assuming all enum values are defined using upper-case letter (this is true for Weather and VehicleStatus).
- In order to easily print into an OutputStream out, create first a corresponding PrintStream using "PrintStream p = new PrintStream(out);" and then use commands like p.println("..."), p.print("..."), etc.
- To convert a JSONObject jo to a String use jo.toString() or jo.toString(3). The first is compact, it does not print white-spaces. The second pretty-prints the `JSON` structure, where the argument is the number of spaces to add to each level of indentation.