

# Práctica 1: Simulador de Tráfico

Fecha de entrega: 03 de Marzo 2025, 08:30h

## Detección de copias

Durante el curso se realizará control de copias de todas las prácticas, comparando las entregas de todos los grupos de TP2. Se considera copia la reproducción total o parcial del código de otros alumnos o cualquier código extraído de Internet o de cualquier otra fuente, salvo aquellas autorizadas explícitamente por el profesor. En caso de detección de copia, la calificación en la convocatoria de TP2 en la que se haya detectado la copia será 0.

Si decides almacenar tu código en un repositorio remoto, por ejemplo en un sistema de control de versiones gratuito con vistas a facilitar la colaboración con tu compañero de laboratorio, asegúrate de que tu código no esté al alcance de los motores de búsqueda. Si alguien que no sea el profesor de tu asignatura, por ejemplo un empleador de una academia privada, te pide que facilites tu código, debes negarte.

## Instrucciones generales

Las siguientes instrucciones **son estrictas**, es decir, debes seguirlas obligatoriamente.

1. Lee el enunciado completo de la práctica antes de empezar.
2. Crea un proyecto Java vacío en Eclipse (usa como mínimo JDK17).
3. Descarga lib.zip, descomprímelo en la raíz del proyecto (al mismo nivel de **src**) y añade las librerías al proyecto. Para ello elige **Project -> Properties -> BuildPath -> Libraries**, marca **ClassPath**, pulsa el botón **Add Jars** y selecciona la librería que quieres añadir.
4. Descarga resources.zip y descomprímelo en la raíz del proyecto (al mismo nivel de **src**).
5. Descarga src.zip y reemplaza el directorio **src** del proyecto por este **src** (o copia el contenido).
6. Es muy importante que cada miembro del grupo haga los puntos 2-5, porque en el examen tendrás que hacerlo usando el **src.zip** de tu práctica (no vas a usar un proyecto ya montado).
7. Es necesario usar exactamente la misma estructura de paquetes y los mismos nombres de clases que aparecen en el enunciado.
8. Debes formatear todo el código usando la funcionalidad de Eclipse (**Source->Format**).
9. Todas las constructoras tienen que comprobar la validez de los parámetros y lanzar excepciones correspondientes con mensaje informativo (se puede usar **IllegalArgumentException**).
10. En la corrección tendremos en cuenta el estilo de código: uso de nombres adecuados para métodos/variables/clases, el formato de código indicado

en el punto anterior, etc.

11. Cuando entregues la práctica sube un fichero con el nombre `src.zip` que incluya solo la carpeta `src`. **No está permitido llamarlo con otro nombre ni usar 7zip, rar, etc.**

## Análisis y creación de datos JSON en Java

JavaScript Object Notation (JSON) es un formato estándar de fichero que utiliza texto y que permite almacenar propiedades de los objetos utilizando pares clave-valor y arrays de tipos de datos. Utilizaremos JSON para la entrada y salida del simulador. Una estructura JSON es un texto estructurado de la siguiente forma:

```
{ "key-1": value-1, ..., "key-n": value-n }
```

donde `key-i` es una secuencia de caracteres (que representa una clave) y `value-i` es un valor JSON válido, es decir, un número, un *string*, otra estructura JSON, o un array `[o-1,...,o-k]`, donde `o-i` es un valor JSON válido. Por ejemplo:

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

En el directorio `lib` se ha incluido una librería para analizar JSON y convertirlo en objetos Java fáciles de manipular (ya se encuentra importada en el proyecto). También puedes usar esta librería para crear estructuras JSON y convertirlas en *strings*. Un ejemplo de uso de esta librería está disponible en el paquete `extra.json`.

Para comparar la salida de tu implementación (sobre los ejemplos que se proporcionan) con la salida esperada, puedes usar el siguiente comparador de ficheros en formato JSON, disponible en: <http://www.jsondiff.com>. Además, el ejemplo que aparece en el paquete `extra.json` incluye otra forma de comparar dos estructuras JSON, usando directamente la librería. Observa que dos estructuras JSON se consideran semánticamente iguales si tienen el mismo conjunto de pares clave-valor. No es necesario que las estructuras sean sintácticamente idénticas. También suministramos un programa que ejecuta tu práctica sobre un conjunto de ejemplos, y compara su salida con la salida esperada –ver la sección *Otros Comentarios*.

## Introducción al simulador de tráfico

Una *simulación* permite ejecutar un modelo en un ordenador para poder observar su comportamiento y aplicar este comportamiento a la vida real. Las prácticas de TP2 consistirán en construir un simulador de tráfico, que modelará *vehículos, carreteras y cruces*, teniendo en cuenta la contaminación ambiental. Habrá diferentes políticas en los cruces para permitir el paso de los vehículos, diferentes tipos de carreteras en función del grado de contaminación y vehículos con distintos identificadores ambientales. De esta forma modelaremos una aplicación (usando orientación a objetos) de un problema de la vida real.

Construiremos el simulador utilizando entrada/salida estándar (ficheros y/o consola). El simulador contendrá una colección de *objetos simulados* (vehículos y carreteras conectadas a través de cruces), otra colección de *eventos* a ejecutar y un *contador de tiempo* que se incrementará en cada paso de la simulación. Un paso de la simulación consiste en realizar las siguientes operaciones:

1. *Procesar los eventos*. En particular estos eventos pueden añadir y/o alterar el estado de los objetos simulados.
2. *Avanzar el estado actual de los objetos simulados* atendiendo a su comportamiento.
3. *Mostrar el estado actual de los objetos simulados*.

Los eventos se leen de un fichero de texto antes de que la simulación comience. Una vez leídos, se inicia la simulación, que se ejecutará un número determinado de unidades de tiempo (llamadas *ticks*) y, en cada *tick* se mostrará el estado de la simulación, bien en la consola o en un fichero de texto.

## El modelo

A lo largo de esta sección presentamos la lógica del simulador de tráfico. En la sección *Objetos de la simulación*, se muestran las diferentes clases necesarias para modelar los cruces, carreteras y vehículos. En la sección *Mapa de carreteras* se describe la clase encargada de agrupar todos los objetos de la simulación, es decir, la clase que implementa un *mapa de carreteras*. El proceso de creación de eventos (vehículos, cruces, carreteras y cambio de alguna de sus propiedades) aparece en la sección *Eventos*. Finalmente, la sección *La clase TrafficSimulator* contendrá la descripción de la clase que implementa el simulador de tráfico, que es la clase responsable de controlar la simulación.

## Objetos de la simulación

Tendremos tres tipos de objetos simulados:

- **Vehículos**, que viajan a través de carreteras y contaminan emitiendo CO<sub>2</sub>. Cada vehículo tendrá un itinerario, que consistirá en la secuencia de cruces por los que tiene que pasar.

- **Carreteras de dirección única**, a través de las cuales viajan los vehículos, y que controlan la velocidad de los mismos para reducir la contaminación, etc.
- **Cruces**, que conectan unas carreteras con otras, organizando el tráfico a través de semáforos. Los semáforos permiten decidir qué vehículos pueden avanzar a la próxima carretera de su itinerario. Cada cruce tendrá asociada una colección de carreteras que llegan a él, a las que denominaremos *carreteras entrantes*. **Desde un cruce sólo se puede llegar directamente a otro cruce a través de una única carretera.**

Cada objeto simulado tendrá un *identificador* único y se podrá actualizar a sí mismo siempre que se lo pida la simulación. Además podrán devolver su estado en formato JSON. La simulación pedirá a cada objeto que actualice su estado exactamente una vez por cada *tick* de la simulación. Todos los objetos de la simulación extienden (directa o indirectamente) a la siguiente clase:

```
package simulator.model;

public abstract class SimulatedObject {

    protected String _id;

    SimulatedObject(String id) {
        if ( id == null || id.isBlank() )
            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 currTime);
    abstract public JSONObject report();
}
```

**Vehículos** Habrá un único tipo de vehículo, que implementaremos en la clase `Vehicle`. Esta clase extenderá a la clase `SimulatedObject`, que se encuentra dentro del paquete `simulator.model`. La clase `Vehicle` debe contener atributos (campos) para almacenar al menos la siguiente información (recuerda que está

prohibido declarar los atributos como `public`):

- *itinerario* (de tipo `List<Junction>`): una lista de cruces, que representa el itinerario del vehículo. La clase `Junction` representa los cruces y se describe en la sección *Cruces*.
- *velocidad máxima* (de tipo `int`): la velocidad máxima a la cual puede viajar el vehículo.
- *velocidad actual* (de tipo `int`): la velocidad actual a la que está circulando el vehículo.
- *estado* (de tipo enumerado `VehicleStatus` – ver paquete `simulator.model`): el estado del vehículo, que puede ser *Pending* (todavía no ha entrado a la primera carretera de su itinerario), *Traveling* (circulando sobre una carretera), *Waiting* (esperando en un cruce), o *Arrived* (ha completado su itinerario).
- *carretera* (de tipo `Road`): la carretera sobre la que el coche está circulando. Debe ser `null` en caso de que no esté en ninguna carretera. La clase `Road` se define en la sección *Carreteras*.
- *localización* (de tipo `int`): la localización del vehículo en la carretera sobre la que está circulando, es decir, la distancia desde el comienzo de la carretera. El comienzo de la carretera es la localización 0.
- *grado de contaminación* (de tipo `int`): un número entre 0 y 10 (ambos inclusive) que se usa para calcular el CO2 emitido por el vehículo en cada paso de la simulación. Es el equivalente a los distintivos medioambientales que actualmente llevan los vehículos en el mundo real.
- *contaminación total* (de tipo `int`): el total de CO2 emitido por el vehículo durante su trayectoria recorrida.
- *distancia total recorrida* (de tipo `int`): la distancia total recorrida por el vehículo.

La clase `Vehicle` tiene una única constructora *package protected*:

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

En esta constructora debes comprobar que los argumentos tienen valores válidos y, en caso de que no los tengan, lanzar la excepción correspondiente. Para hacer dicha comprobación debes tener en cuenta lo siguiente: `maxSpeed` tiene que ser positivo; `contClass` debe ser un valor entre 0 y 10 (ambos incluidos); y la longitud de la lista `itinerary` es al menos 2.

Además, no se debe compartir el argumento `itinerary`, sino que debes hacer una copia de dicho argumento en una lista de sólo lectura, para evitar modificarlo

desde fuera:

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

La clase **Vehicle** tiene los siguientes métodos, cuya declaración debes respetar (recuerda que cuando no aparece modificador de visibilidad significa que el método es *package protected*):

- **void setSpeed(int s):** pone la velocidad actual al valor mínimo entre **s** y la velocidad máxima del vehículo. Lanza una excepción si **s** es negativo.
- **void setContaminationClass(int c):** pone el valor de contaminación del vehículo a **c**. Lanza una excepción si **c** no es un valor entre 0 y 10 (ambos incluidos).
- **void advance(int currTime):** actualiza el estado del vehículo **1 tick**. En concreto, si el estado del vehículo no es *Traveling*, no hace nada. En otro caso:
  - (a) se actualiza su localización al valor mínimo entre (i) la localización actual más la velocidad actual; y (ii) la longitud de la carretera por la que está circulando.
  - (b) calcula la contaminación **c** producida utilizando la fórmula  $c=lf$ , donde **f** es el grado de contaminación y **l** es la distancia recorrida en ese paso de la simulación, es decir, la nueva localización menos la localización previa. Después añade **c** a la contaminación total del vehículo\* y también añade **c** al grado de contaminación de la carretera actual, invocando al método correspondiente de la clase **Road**.
  - (c) si la localización actual (es decir la nueva) es mayor o igual a la longitud de la carretera, el vehículo entra en la cola del cruce correspondiente (llamando a un método de la clase **Junction**). Recuerda que debes modificar el estado del vehículo.
- **void moveToNextRoad():** mueve el vehículo a la siguiente carretera. Este proceso se hace *saliendo* de la carretera actual y *entrando* a la siguiente carretera de su itinerario, en la localización 0. Para salir y entrar de las carreteras, debes utilizar el método correspondiente de la clase **Road**. Para encontrar la siguiente carretera, el vehículo debe preguntar al cruce en el cual está esperando (o al primero del itinerario en caso de que el estado del vehículo sea *Pending*) mediante una invocación al método correspondiente de la clase **Junction**. Observa que la primera vez que el vehículo llama a este método el vehículo no sale de ninguna carretera, ya que el vehículo todavía no ha empezado a circular y que cuando el vehículo abandona el último cruce de su itinerario ya no puede entrar a ninguna carretera dado que ha finalizado su recorrido – no olvides modificar el estado del vehículo.

Este método debe lanzar una excepción si el estado de los vehículos no es *Pending* o *Waiting*. Recuerda que el itinerario es una lista de cruces

*de sólo lectura y por tanto no puedes modificarla.* Es conveniente guardar un índice que indique el último cruce del itinerario que ha sido visitado por el vehículo.

- `public JSONObject report():` devuelve el estado del vehículo en el siguiente formato JSON:

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

donde `id` es el identificador del vehículo; `speed` es la velocidad actual; `distance` es la distancia total recorrida por el vehículo; `co2` es el total de CO2 emitido por el vehículo; `class` es la etiqueta medioambiental del vehículo; `status` es el estado del vehículo que puede ser `PENDING`, `TRAVELING`, `WAITING` o `ARRIVED`; `road` es el identificador de la carretera sobre la que el vehículo está circulando; y `location` es su localización sobre la carretera. Si el estado del vehículo es *Pending* o *Arrived*, las claves `road` y `location` se deben omitir en el informe.

Además, define los siguientes *getters públicos* para consultar la información correspondiente: `getLocation()`, `getSpeed()`, `getMaxSpeed()`, `getContClass()`, `getStatus()`, `getTotalCO2()`, `getItinerary()`, `getRoad()`.

También puedes definir *setters* siempre que sean privados. **Asegúrate de que la velocidad del vehículo es 0 cuando su estado no es *Traveling*.**

**Carreteras** En esta práctica tendremos dos tipos de carreteras. La diferencia entre ambos tipos radica en cómo gestionan los niveles de contaminación. Primero describimos la clase base `Road` (en el paquete `simulator.model`), que extiende a `SimulatedObject`. Después usaremos herencia para definir los dos tipos de carreteras. La clase `Road` contendrá al menos los siguientes campos o atributos:

- *cruce origen* y *cruce destino* (ambos de tipo `Junction`): los cruces a los cuales la carretera está conectada. Circular por esa carretera es ir desde el cruce *origen* al cruce *destino*.
- *longitud* (de tipo `int`): la longitud de la carretera (en alguna unidad para medir la distancia, por ejemplo metros).

- *velocidad máxima* (de tipo `int`): la velocidad máxima permitida en esa carretera.
- *límite actual de velocidad* (de tipo `int`): un vehículo no puede circular por esa carretera a una velocidad superior al límite establecido. Su valor inicial debe ser igual a la velocidad máxima.
- *alarma por contaminación excesiva* (de tipo `int`): indica un límite de contaminación que una vez superado impone restricciones al tráfico para reducir la contaminación.
- *condiciones ambientales* (de tipo enumerado `Weather` – ver el paquete `simulator.model`): las condiciones atmosféricas en la carretera. Este valor se usa para actualizar la velocidad de los vehículos, estado de contaminación, etc.
- *contaminación total* (de tipo `int`): la contaminación total acumulada en la carretera, es decir, el total de CO2 emitido por los vehículos que circulan sobre la carretera.
- *vehículos* (de tipo `List<Vehicle>`): una lista de vehículos que están circulando por la carretera – **debe estar siempre ordenada por la localización de los vehículos (orden descendente)**. Observa que puede haber varios vehículos en la misma localización. Sin embargo, su orden de llegada a esa localización debe preservarse en la lista. Para eso el vehículo que llega el primero será el primero en circular – recuerda que los algoritmos de ordenación de Java garantizan que elementos iguales no se reordenarán como resultado de la ordenación.

La clase `Road` tiene únicamente una constructora *package protected*:

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

La constructora debe añadir la carretera como carretera saliente a su cruce origen, y como carretera entrante a su cruce destino. En esta constructora debes comprobar que los argumentos tienen valores válidos y, en otro caso, lanzar excepción. Los valores válidos son: `maxSpeed` es positivo; `contLimit` es no negativo; `length` es positivo; `srcJunc`, `destJunc` y `weather` son distintos de `null`. Esta clase tiene los siguientes métodos, cuya declaración debes respetar:

- `void enter(Vehicle v)`: se utiliza por los vehículos para entrar a la carretera. Simplemente añade el vehículo a la lista de vehículos de la carretera (al final). Debe comprobar que la localización del vehículo es 0 y que la velocidad del vehículo también es 0. En otro caso lanzará una excepción.
- `void exit(Vehicle v)`: lo utiliza un vehículo para abandonar la carretera. Simplemente elimina el vehículo de la lista de vehículos de la



carretera.

- `void setWeather(Weather w)`: pone las condiciones atmosféricas de la carretera al valor `w`. Debe comprobar que `w` no es `null` y lanzar una excepción en caso contrario.
- `void addContamination(int c)`: añade `c` unidades de CO2 al total de la contaminación de la carretera. Tiene que comprobar que `c` no es negativo y lanzar una excepción en otro caso.
- `abstract void reduceTotalContamination()`: método abstracto para reducir el total de la contaminación de la carretera. La implementación específica la definirán las subclases de `Road`.
- `abstract void updateSpeedLimit()`: método abstracto para actualizar la velocidad límite de la carretera. La implementación específica la definirán las subclases de `Road`.
- `abstract int calculateVehicleSpeed(Vehicle v)`: método abstracto para calcular la velocidad de un vehículo `v`. La implementación específica la definirán las subclases de `Road`.
- `void advance(int currTime)`: avanza **1 tick** el estado de la carretera de la siguiente forma:
  1. llama a `reduceTotalContamination` para reducir la contaminación total, es decir, la disminución de CO2.
  2. llama a `updateSpeedLimit` para establecer el límite de velocidad de la carretera en el paso de simulación actual.
  3. recorre la lista de vehículos (desde el primero al último) y, para cada vehículo:
    1. pone la velocidad del vehículo al valor devuelto por `calculateVehicleSpeed`.
    2. llama al método `advance` del vehículo.

Recuerda ordenar la lista de vehículos por su localización al final del método.
- `public JSONObject report()`: devuelve el estado de la carretera en el siguiente formato JSON:

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

donde `id` es el identificador de la carretera; `speedlimit` es la velocidad límite actual; `weather` son las condiciones atmosféricas actuales; `co2` es la contaminación total; y `vehicles` es una lista de identificadores de vehículos que están circulando en la carretera (en el mismo orden en el que se han almacenado en la lista correspondiente).

Además, define los siguientes *getters públicos* para consultar la información correspondiente: `getLength()`, `getDest()`, `getSrc()`, `getWeather()`, `getContLimit()`, `getMaxSpeed()`, `getTotalCO2()`, `getSpeedLimit()`, `getVehicles()`. El método `getVehicles()` debe devolver una lista de *solo lectura* (usando `Collections.unmodifiableList(vehicles)`). A continuación describimos las dos clases de carreteras que debes implementar, y que heredan de `Road`.

**Carreteras inter-urbanas** Esta clase de carreteras se utiliza para conectar ciudades y se implementa a través de la clase `InterCityRoad`, que extiende a `Road`. La clase debe colocarse dentro del paquete `simulator.model` y ha de contener, al menos, los siguientes métodos:

- **reduceTotalContamination:** que reduce la contaminación total al valor:  
$$((100-x)*tc)/100$$
donde `tc` es la contaminación total actual y `x` depende de las condiciones atmosféricas: 2 en caso de tiempo `SUNNY` (soleado), 3 si el tiempo es `CLOUDY` (nublado), 10 en caso de que el tiempo sea `RAINY` (lluvioso), 15 si es `WINDY` (ventisca), y 20 si el tiempo está `STORM` (tormentoso).
- **updateSpeedLimit:** si la contaminación total excede el límite de contaminación, entonces pone el límite de la velocidad al 50% de la velocidad máxima (es decir a `maxSpeed/2`). En otro caso pone el límite de la velocidad a la velocidad máxima.
- **calculateVehicleSpeed:** calcula la velocidad del vehículo como la velocidad límite de la carretera. Si el tiempo es `STORM` lo reduce en un 20% (es decir,  $(speedLimit*8)/10$ ).

**Carreteras urbanas** Esta clase de carreteras están dentro de las ciudades y se implementan a través de la clase `CityRoad`, que extiende a `Road` y debe estar dentro del paquete `simulator.model`. Estas carreteras son más restrictivas cuando hay una contaminación excesiva. Nunca reducen la velocidad límite (que siempre es su máxima velocidad), pero calculan la velocidad de los vehículos dependiendo del grado de contaminación. Es más, la reducción de la contaminación no depende de las condiciones atmosféricas como antes. El comportamiento de esta clase viene dado por los siguientes métodos:

- **método reduceTotalContamination:** que reduce el total de la contaminación en `x` unidades de CO<sub>2</sub>, donde `x` depende de las condiciones atmos-

féricas: 10 en caso de WINDY o STORM, y 2 en otro caso. Asegúrate de que el total de contaminación no se vuelve negativo.

- la velocidad límite no cambia, siempre es la velocidad máxima.
- método `calculateVehicleSpeed`: que calcula la velocidad de un vehículo usando la expresión  $((11-f)*s)/11$ , donde **s** es la velocidad límite de la carretera y **f** es el grado de contaminación del vehículo.

**Cruces** Los cruces en la simulación controlan las carreteras entrantes usando semáforos, que pueden estar en *verde* o en *rojo*. Si el semáforo está en verde, entonces se permite que los vehículos pasen, es decir que abandonen el cruce y pasen a la siguiente carretera de su itinerario. Si el semáforo está en *rojo*, entonces los vehículos se queden detenidos. Cada carretera entrante tiene una cola en la cual los vehículos que llegan se almacenan y, van abandonando la cola, cuando el semáforo se pone en verde. En cada paso de la simulación puede haber únicamente una carretera entrante al cruce que tenga un semáforo en *verde*. Tendremos varias clases de cruces, que se diferenciarán en lo siguiente:

1. la forma de elegir la carretera entrante que pondrá su semáforo en verde; y
2. cómo eliminan los vehículos de la cola de la carretera entrante que tiene su semáforo en verde, es decir, qué vehículos de la cola pueden pasar a la siguiente carretera de su itinerario en cada paso de la simulación.

No vamos a utilizar herencia para definir los cruces, sino que los cruces tendrán su propia estrategia. Las estrategias se encapsularán usando una jerarquía de clases. A continuación vamos a describir cómo encapsular las estrategias para cambiar las luces de los semáforos y para eliminar los vehículos de las colas. Después presentaremos la clase `Junction`, que utilizará esta jerarquía.

**Estrategias de cambio de semáforo** Estas estrategias nos servirán para decidir cuál de las carreteras entrantes al cruce pondrá su semáforo en verde. Para implementar las estrategias utilizaremos la siguiente interfaz, que debe colocarse en el paquete `simulator.model`:

```
package simulator.model;

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

El método `chooseNextGreen` recibe como parámetros:

- **roads**: la lista de carreteras entrantes al cruce.
- **qs**: una lista de listas de vehículos, donde las listas internas de vehículos representan *colas*. La cola *i*-ésima corresponde a la cola de vehículos de la *i*-ésima carretera de la lista **roads**. Observa que usamos el tipo

`List<Vehicle>` en lugar de `Queue<Vehicle>` para representar una cola, ya que la interfaz `Queue` en Java no garantiza ningún orden cuando se recorre la colección (y lo que queremos es recorrerla en el orden en el cual los elementos se añadieron).

- **currGreen**: el índice (en la lista **roads**) de la carretera que tiene el semáforo en verde. El valor -1 se utiliza para indicar que todos los semáforos están en rojo.
- **lastSwitchingTime**: el paso de la simulación en el cual el semáforo para la carretera **currGreen** se cambió de *rojo* a *verde*. Si **currGreen** es -1 entonces es el último paso en el cual todos cambiaron a rojo.
- **currTime**: el paso de simulación actual.

El método devuelve el índice de la carretera (en la lista **roads**) que tiene que poner su semáforo a verde – si es la misma que **currGreen**, entonces, el cruce no considerará el cambio. Si devuelve -1 significa que todos deberían estar en rojo.

Tenemos dos estrategias para cambiar los semáforos de color, que son **RoundRobinStrategy** y **MostCrowdedStrategy** (colocadas en el paquete **simulator.model**). Ambas reciben en la constructora un parámetro **timeSlot** (de tipo **int**), que representa el número de “ticks” consecutivos durante los cuales la carretera puede tener el semáforo en verde. A continuación definimos el comportamiento de ambas estrategias.

La estrategia **RoundRobinStrategy** se comporta como sigue:

1. si la lista de carreteras entrantes es vacía, entonces devuelve -1.
2. si los semáforos de todas las carreteras entrantes están rojos (es decir, **currGreen** es -1), entonces pone en verde el primer semáforo de la lista **roads** (es decir, devuelve 0).
3. si **currTime-lastSwitchingTime < timeSlot**, deja los semáforos tal cual están (es decir, devuelve **currGreen**).
4. en otro caso devuelve **currGreen+1** módulo la longitud de la lista **roads** (es decir, el índice de la siguiente carretera entrante, recorriendo la lista de forma circular).

La estrategia **MostCrowdedStrategy** se comporta como sigue:

1. si la lista de carreteras entrantes es vacía, entonces devuelve -1.
2. si todos los semáforos de las carreteras entrantes están en rojo, pone verde el semáforo de la carretera entrante con la cola más larga, empezando la búsqueda (en **qs**) desde la posición 0. Si hay más de una carretera entrante con la misma longitud máxima de cola, entonces coge la primera que encuentra durante la búsqueda.

3. si `currTime-lastSwitchingTime < timeSlot`, entonces deja los semáforos tal cual están (es decir devuelve `currGreen`).
4. en otro caso pone a verde el semáforo de la carretera entrante con la cola más larga, realizando una búsqueda circular (en `qs`) desde la posición `currGreen+1` módulo el número de carreteras entrantes al cruce. Si hay más de una carretera cuyas colas tengan el mismo tamaño maximal, entonces coge la primera que encuentra durante la búsqueda. Observa que podría devolver `currGreen`.

**Estrategias de las colas de las carreteras entrantes** Estas estrategias eliminan vehículos de las carreteras entrantes cuyo semáforo esté a verde. Se modelan a través de la interfaz (que debes colocar en el paquete `simulator.model`):

```
package simulator.model;

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

La interfaz tiene un único método que devuelve una lista de vehículos (extraídos de `q`) a los que habrá que solicitar que se muevan a sus siguientes carreteras. El método no debe modificar la cola `q`, ni pedir a los vehículos que se muevan, ya que esta tarea pertenece a la clase `Junction`. Implementaremos dos estrategias para sacar elementos de la cola (que situaremos en el paquete `simulator.model`):

- `MoveFirstStrategy` devuelve una lista que incluye el primer vehículo de `q`.
- `MoveAllStrategy` devuelve la lista de todos los vehículos que están en `q` (no devuelve `q`). El orden debe ser el mismo que cuando se itera `q`.

**Clase `Junction`** La funcionalidad de un cruce se implementa a través de la clase `Junction`, que extiende a `SimulatedObject`, y que debe estar colocada en el paquete `simulator.model`. La clase `Junction` contiene al menos los siguientes atributos (que no pueden ser públicos):

- *lista de carreteras entrantes* (de tipo `List<Road>`): una lista de todas las carreteras que entran al cruce, es decir, el cruce es su destino.
- *mapa de carreteras salientes* (de tipo `Map<Junction,Road>`): un mapa de carreteras salientes, es decir, si  $(j,r)$  es un par clave-valor del mapa, entonces el cruce está conectado al cruce  $j$  a través de la carretera  $r$ . El mapa se usa para saber qué carretera seleccionar para llegar al cruce  $j$ .
- *lista de colas* (de tipo `List<List<Vehicle>>`): una lista de colas para las carreteras entrantes – la cola  $i$ -ésima (representada como

`List<Vehicle>`) corresponde a la  $i$ -ésima carretera en la *lista de carreteras entrantes*. Se recomienda guardar un mapa de “carretera-cola” (de tipo `Map<Road, List<Vehicles>`) para hacer la búsqueda, en la cola de una carretera dada, de forma eficiente.

- *índice del semáforo en verde* (de tipo `int`): el índice de la carretera entrante (en la lista de carreteras entrantes) que tiene el semáforo en verde. El valor -1 se usa para indicar que todas las carreteras entrantes tienen su semáforo en rojo.
- *último paso de cambio de semáforo* (de tipo `int`): el paso en el cual el *índice del semáforo en verde* ha cambiado de valor. Su valor inicial es 0.
- *estrategia de cambio de semáforo* (de tipo `LightSwitchingStrategy`): una estrategia para cambiar de color los semáforos.
- *estrategia para extraer elementos de la cola* (de tipo `DequeuingStrategy`): una estrategia para eliminar vehículos de las colas.
- *coordenadas  $x$  e  $y$*  (de tipo `int`): coordenadas que se usarán para dibujar el cruce en la próxima práctica.

La clase `Junction` tiene únicamente la constructora *package protected*:

```
Junction(String id, LightSwitchStrategy lsStrategy, DequeuingStrategy dqStrategy, int xCoord,
    super(id);
    // ...
}
```

Observa que la constructora recibe las estrategias como parámetros. Debe comprobar que `lsStrategy` y `dqStrategy` no son `null`, y que `xCoord` y `yCoord` no son negativos, y lanzar una excepción en caso contrario.

La clase `Junction` tiene los siguientes métodos (debes respetar los modificadores de visibilidad tal cual se describen):

- `void addIncommingRoad(Road r)`: añade `r` al final de la lista de carreteras entrantes, crea una cola (una instancia de `LinkedList` por ejemplo) para `r` y la añade al final de la lista de colas. Además, si has usado un mapa carretera-cola, entonces debes añadir el correspondiente par al mapa. Debes comprobar que la carretera `r` es realmente una carretera entrante, es decir, su cruce destino es igual al cruce actual y, lanzar una excepción en caso contrario.
- `void addOutGoingRoad(Road r)`: añade el par  $(j, r)$  al mapa de carreteras salientes, donde  $j$  es el cruce destino de la carretera `r`. Tienes que comprobar que ninguna otra carretera va desde `this` al cruce  $j$  y, que la carretera `r`, es realmente una carretera saliente al cruce actual. En otro caso debes lanzar una excepción.
- `void enter(Vehicle v)`: añade el vehículo `v` a la cola de la carretera `r`, donde `r` es la carretera actual del vehículo `v`.

- `Road roadTo(Junction j)`: devuelve la carretera que va desde el cruce actual al cruce `j`. Para esto debes buscar en el mapa de carreteras salientes.
- `void advance(int currTime)`: avanza **1 tick** el estado del cruce como sigue:
  1. utiliza la *estrategia de extracción de la cola* para calcular la lista de vehículos que deben avanzar y después les pide a los vehículos que se muevan a sus siguientes carreteras, eliminándolos de las colas correspondientes.
  2. utiliza la *estrategia de cambio de semáforo* para calcular el índice de la siguiente carretera a la que hay que poner su semáforo en verde. Si es distinto del índice actual, entonces cambia el valor del índice al nuevo valor y pone el último paso de cambio de semáforo al paso actual (es decir, el valor del parámetro `currTime`).
- `public JSONObject report()`: devuelve el estado del cruce en el siguiente formato JSON:

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

donde `id` es el identificador del cruce; `green` es el identificador de la carretera con el semáforo en verde (`none` si todos están en rojo); y `queues` es la lista de colas de las carreteras entrantes, donde cada `Qi` tiene el siguiente formato JSON:

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

donde `road` es el identificador de la carretera y `vehicles` es la lista de vehículos en el orden en que aparecen en la cola (el orden debe ser el mismo que el usado para recorrerla).

## Mapa de carreteras

El propósito de esta clase es agrupar todos los objetos de la simulación. Esto facilita el trabajo del simulador. Se implementa a través de la clase `RoadMap` que debe estar colocada dentro del paquete `simulator.model`. La clase `RoadMap` tiene al menos los siguientes atributos, que no pueden ser públicos:

- *lista de cruces* de tipo `List<Junction>`.
- *lista de carreteras* de tipo `List<Road>`.

- *lista de vehículos* de tipo `List<Vehicle>`.
- *mapa de cruces* de tipo `Map<String,Junction>`: un mapa de identificadores de cruces a los correspondientes cruces.
- *mapa de carreteras* de tipo `Map<String,Road>`: un mapa de identificadores de carreteras a las correspondientes carreteras.
- *mapa de vehículos* de tipo `Map<String,Vehicle>`: un mapa de identificadores de vehículos a los correspondientes vehículos.

Recuerda tener actualizadas las listas y los mapas para usar los mapas en pro de la eficiencia de la búsqueda de un objeto; y usar las listas para recorrer los objetos en el mismo orden en el cual han sido añadidos.

La clase `RoadMap` tiene una única constructora *package protected* por defecto, para inicializar los atributos mencionados a sus valores por defecto. Además contiene los siguientes métodos (que deben tener los modificadores de visibilidad descritos abajo):

- `void addJunction(Junction j)`: añade el cruce `j` al final de la lista de cruces y modifica el mapa correspondiente. Debes comprobar que no existe ningún otro cruce con el mismo identificador.
- `void addRoad(Road r)`: añade la carretera `r` al final de la lista de carreteras y modifica el mapa correspondiente. Debes comprobar que se cumplen lo siguiente:
  - (i) no existe ninguna otra carretera con el mismo identificador; y (ii) los cruces que conecta la carretera existen en el mapa de carreteras. En caso de que no se cumplan el método lanza una excepción.
- `void addVehicle(Vehicle v)`: añade el vehículo `v` al final de la lista de vehículos y modifica el mapa de vehículos en concordancia. Debes comprobar que los siguientes puntos se cumplen: (i) no existe ningún otro vehículo con el mismo identificador; y (ii) el itinerario es válido, es decir, existen carreteras que conecten los cruces consecutivos de su itinerario. En caso de que no se cumplan (i) y (ii), el método debe lanzar una excepción.
- `public Junction getJunction(String id)`: devuelve el cruce con identificador `id`, y `null` si no existe dicho cruce.
- `public Road getRoad(String id)`: devuelve la carretera con identificador `id`, y `null` si no existe dicha carretera.
- `public Vehicle getVehicle(String id)`: devuelve el vehículo con identificador `id`, y `null` si no existe dicho vehículo.
- `public List<Junction> getJunctions()`: devuelve una versión de *solo lectura* de la lista de cruces.
- `public List<Roads> getRoads()`: devuelve una versión de *solo lectura* de la lista de carreteras.



- `public List<Vehicle> getVehicles():` devuelve una versión de *solo lectura* de la lista de vehículos.
- `void reset():` limpia todas las listas y mapas.
- `public JSONObject report():` devuelve el estado del mapa de carreteras en el siguiente formato JSON:
 

```

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

donde `JiReport`, `RiReport` y `ViReport`, son los informes de los correspondientes objetos de la simulación. El orden en las listas JSON debe ser el mismo que el orden de las listas correspondientes de `RoadMap`.

## Eventos

Los eventos nos permiten inicializar e interactuar con el simulador, añadiendo vehículos, carreteras y cruces; cambiando las condiciones atmosféricas de las carreteras; y cambiando el nivel de contaminación de los vehículos. Cada evento tiene un tiempo en el cual debe ser ejecutado. En cada tick `t`, el simulador ejecuta todos los eventos correspondientes al paso `t`, en el orden en el cual fueron añadidos a la cola de eventos. Primero vamos a definir una clase abstracta `Event` (dentro del paquete `simulator.model`) para modelar un evento:

```

package simulator.model;

public abstract class Event implements Comparable<Event> {

    private static long _counter = 0;

    protected int _time;
    protected long _time_stamp;

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

    int getTime() {
        return _time;
    }
}

```

```

    }

    @Override
    public int compareTo(Event o) {
        // TODO complete the method to compare events according to their _time, and when
        // _time is equal it compares the _time_stamp;
    }

    abstract void execute(RoadMap map);
}

```

El campo `_time` es el tiempo (o paso) en el cual este evento tiene que ser ejecutado, el campo `_time_stamp` se usará para saber que evento se ha creado antes cuando su campo `_time` es igual, y el método `execute` es el método al que el simulador llama para ejecutar el evento. La funcionalidad de este método se define en las subclases.

En lo que sigue vamos a describir los tipos de eventos que existen en el simulador. Todos ellos extenderán a la clase `Event` y deben estar colocados dentro del paquete `simulator.model`. La constructora de cada evento recibe algunos datos para ejecutar una operación cuando sea requerido, que se almacenan en los campos y se usan en el método `execute` para ejecutar la funcionalidad correspondiente.

**Evento “New Junction”** Este evento se implementa en la clase `NewJunctionEvent`, que extiende a `Event`. Tiene la siguiente constructora:

```

public NewJunctionEvent(int time, String id, LightSwitchingStrategy lsStrategy, DequeueingStrategy dsStrategy) {
    super(time);
    // ...
}

```

El método `execute` de este evento crea el cruce correspondiente y lo añade al mapa de carreteras (el parámetro de `execute`).

**Evento “New Road”** Existen dos eventos para crear carreteras: `NewCityRoadEvent` y `NewInterCityRoadEvent`. Cada evento tiene una constructora de la forma:

```

public NewCityRoadEvent(int time, String id, String srcJun, String destJunc, int length, int width) {
    super(time);
    // ...
}

public NewInterCityRoadEvent(int time, String id, String srcJun, String destJunc, int length, int width) {
    super(time);
    // ...
}

```

El método `execute` de estos eventos crea una carretera y la añade al mapa de carreteras. Estos dos eventos tienen mucho en común, y por lo tanto tendrás que duplicar código. Después de implementarlos y comprobar dichos eventos, refactorízalos incorporando una super clase `NewRoadEvent` que incluye las partes comunes a ambas.

**Evento “New Vehicle”** Este evento se implementa en la clase `NewVehicleEvent`. Tiene la siguiente constructora:

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

El método `execute` de este evento crea un vehículo en función de sus argumentos y lo añade al mapa de carreteras. Después llama a su método `moveToNext` para comenzar su viaje.

**Evento “Set Weather”** Este evento se implementa en la clase `SetWeatherEvent`. Tiene como constructora a:

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

Se debe comprobar que `ws` no es `null` y lanzar una excepción en caso contrario. El método `execute` recorre la lista `ws`, y para cada elemento `w` pone las condiciones atmosféricas de la carretera con identificador `w.getFirst()` a `w.getSecond()` (ver el paquete `simulator.misc` para el código de la clase `Pair`). Debe lanzar una excepción si la carretera no existe en el mapa de carreteras.

**Event “Set Contamination Class”** Este evento se implementa en la clase `SetContClassEvent`. Tiene como constructora:

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

Debe comprobar que `cs` no es `null` y lanzar una excepción en caso contrario. El método `execute` recorre la lista `cs`, y para cada elemento de `c`, pone el estado de contaminación del vehículo con identificador `c.getFirst()` a `c.getSecond()`. Debe lanzar una excepción si el vehículo no existe en el mapa de carreteras.

### La clase “TrafficSimulator”

La clase del simulador es la única responsable de ejecutar la simulación. Se implementa en la clase `TrafficSimulator` dentro del paquete `simulator.model`.

Esta clase tiene al menos los siguientes atributos, que no pueden ser públicos:

- *mapa de carreteras* (de tipo `RoadMap`): un mapa de carreteras en el cual se almacenan todos los objetos de la simulación.
- *cola de prioridades de eventos* (de tipo `Queue<Event>`): una cola de eventos a ejecutar. La prioridad es por el tiempo de los eventos, y si dos eventos tienen el mismo tiempo, el que fue añadido antes sale antes – esta funcionalidad hay que implementarla en el método `compareTo` de la clase `Event`. Usar la clase `PriorityQueue<Event>` para instanciar este atributo.
- *tiempo (paso) de la simulación* (de tipo `int`): el paso de la simulación, que inicialmente será 0.

La clase `TrafficSimulator` tiene sólo una constructora pública por defecto, que inicializa los campos a sus valores por defecto. Además contendrá los siguientes métodos (debes mantener los modificadores de visibilidad como se describen a continuación):

- `public void addEvent(Event e)`: añade el evento `e` a la lista de eventos. Recuerda que la lista de eventos tiene que estar ordenada como se describió anteriormente.
- `public void advance()`: avanza el estado de la simulación de la siguiente forma (**¡el orden de los puntos que aparecen a continuación es muy importante!**):
  1. incrementa el tiempo de la simulación en 1.
  2. ejecuta todos los eventos cuyo tiempo sea el tiempo actual de la simulación y los elimina de la lista.
  3. llama al método `advance` de todos los cruces.
  4. llama al método `advance` de todas las carreteras.
- `public void reset()`: limpia el *mapa de carreteras* y la *lista de eventos*, y pone el *tiempo de la simulación* a 0.
- `public JSONObject report()`: devuelve el estado del simulador en el siguiente formato JSON:

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

donde `time` es el *tiempo de la simulación* actual, y `state` es lo que devuelve el método `report()` del *mapa de carreteras*.

## Controlador

Ahora que hemos definido las diferentes clases que forman parte de la lógica del simulador, podemos empezar a testearlo escribiendo un método que cree algunos eventos, los añada al simulador y después llame al método `advance` del simulador varias veces para ejecutar la simulación. Aunque esto es adecuado para testear la aplicación, no es fácil para los usuarios utilizarla. La misión del controlador es ofrecer una manera sencilla de utilizar la aplicación. En particular permite cargar los eventos de ficheros de texto, y crear de forma automática los eventos para añadirlos al simulador, ejecutando la simulación un número determinado de pasos.

Vamos a usar estructuras JSON para describir eventos, y utilizaremos factorías para parsear estas estructuras y transformarlas en objetos de la simulación. En la sección *Las Factorías* describimos las factorías necesarias para facilitar la creación de eventos a partir de las estructuras JSON; y en la sección *El Controlador* describiremos el controlador, que es la clase que permite cargar eventos desde un `InputStream` y ejecutar el simulador un número concreto de pasos.

## Las Factorías

Todas las clases/interfaces de este apartado tienen que ir en el paquete `simulator.factories`.

Como en la práctica tenemos varias factorías vamos a usar genéricos para evitar duplicar código. A continuación detallamos cómo implementarlas paso a paso.

**La Interfaz `Factory<T>`** Una factoría se modela con la interfaz genérica `Factory`:

```
public interface Factory<T> {
    public T create_instance(JSONObject info);
    public List<JSONObject> get_info();
}
```

El método `create_instance` recibe una estructura JSON que describe el objeto a crear, y devuelve una instancia de la clase correspondiente — una instancia de un subtipo de `T`. En caso de que `info` sea incorrecto, entonces lanza la excepción correspondiente. En nuestro caso, la estructura JSON que se pasa como parámetro al método `create_instance` incluye dos claves:

- **type**: es un string que describe el objeto que se va a crear;
- **data**: que es una estructura JSON que incluye toda la información necesaria para crear el objeto, por ejemplo, los argumentos necesarios en el correspondiente constructor de la clase, etc.

El método `get_info` devuelve una lista de objetos `JSON` que describen qué puede ser creado por la factoría, ver detalles a continuación. Este método lo vamos a usar en la segunda práctica.

Existen muchas formas de definir una factoría, que veremos durante el curso (o en la asignatura Ingeniería de Software). Nosotros la vamos a diseñar utilizando lo que se conoce como *builder based factory*, que permite extender una factoría con más opciones sin necesidad de modificar su código. Es una combinación de los patrones de diseño *Command* y *Factory*.

**La Clase `Builder<T>`** El elemento básico en una *builder based factory* es el *builder*, que es una clase capaz de crear una instancia de un tipo específico. Podemos modelarla como una clase genérica `Builder<T>`:

```
public abstract class Builder<T> {
    private String _type_tag;
    private String _desc;

    public Builder(String type_tag, String desc) {
        if (type_tag == null || desc == null || type_tag.isBlank() || desc.isBlank())
            throw new IllegalArgumentException("Invalid type/desc");

        _type_tag = type_tag;
        _desc = desc;
    }

    public String get_type_tag() {
        return _type_tag;
    }

    public JSONObject get_info() {
        JSONObject info = new JSONObject();
        info.put("type", _type_tag);
        info.put("desc", _desc);
        JSONObject data = new JSONObject();
        fill_in_data(data);
        info.put("data", data);
        return info;
    }

    protected void fill_in_data(JSONObject o) {
    }

    @Override
    public String toString() {
        return _desc;
    }
}
```

```

    }

    protected abstract T create_instance(JSONObject data);
}

```

El atributo `_type_tag` coincide con el campo `type` de la estructura JSON correspondiente, y el atributo `_desc` describe que tipo de objetos pueden ser creados por este builder (para mostrar al usuario). Las subclases tienen que sobrescribir `fill_in_data` para rellenar los parámetros en `o` si es necesario.

Las clases que extienden a `Builder<T>` son las responsables de asignar valores a `_type_tag` y `_desc` llamando a la constructora de la clase `Builder`, y también de definir el método `create_instance` para crear un objeto del tipo `T` (o de cualquier instancia que sea subclase de `T`) en caso de que toda la información necesaria se encuentre disponible en `data`. En otro caso genera una excepción de tipo `IllegalArgumentException` describiendo que información es incorrecta o no se encuentra disponible.

El método `get_info` devuelve un objeto JSON que será utilizado por el método `get_info()` de la factoría. Este objeto incluye dos campos correspondientes a `_type_tag` y `_desc`, y otro `data` con la información que requiere dicho “builder”. Por defecto, `data` no incluye ninguna información, y si queremos añadir más información tenemos que sobrescribir `fill_in_data` para rellenarla. Utilizaremos este método en la segunda práctica, así que podéis dejar las implemenetación de `fill_in_data` en las subclases para más adelante.

**La Clase `BuilderBasedFactory<T>`** Una vez que los builders están preparados, implementamos una builder based factory genérica. Es una clase que tiene un mapa de builders, de tal forma que cuando queramos crear un objeto a partir de una estructura JSON, encuentra el builder con el que poder generar la instancia correspondiente:

```

public class BuilderBasedFactory<T> implements Factory<T> {
    private Map<String, Builder<T>> _builders;

    public BuilderBasedFactory() {
        // Create a HashMap for _builders, and a LinkedList _builders_info
        // ...
    }

    public BuilderBasedFactory(List<Builder<T>> builders) {
        this();

        // call add_builder(b) for each builder b in builder
        // ...
    }
}

```

```

public void add_builder(Builder<T> b) {
    // add an entry "b.getTag() |-> b" to _builders.
    // ...
    // add b.get_info() to _buildersInfo
    // ...
}

@Override
public T create_instance(JSONObject info) {
    if (info == null) {
        throw new IllegalArgumentException("'info' cannot be null");
    }

    // Look for a builder with a tag equals to info.getString("type"), in the
    // map _builder, and call its create_instance method and return the result
    // if it is not null. The value you pass to create_instance is the following
    // because 'data' is optional:
    //
    // info.has("data") ? info.getJSONObject("data") : new JSONObject()
    // ...

    // If no builder is found or the result is null ...
    throw new IllegalArgumentException("Unrecognized 'info':" + info.toString());
}

@Override
public List<JSONObject> get_info() {
    return Collections.unmodifiableList(_builders_info);
}
}

```

**Factoría para las estrategias de cambio de semáforo** Para esta factoría necesitamos dos “builders”, `RoundRobinStrategyBuilder` y `MostCrowdedStrategyBuilder`, ambos extendiendo a `Builder<LightSwitchingStrategy>`, ya que crean instancias de las clases que implementan a `LightSwitchingStrategy`.

La clase `RoundRobinStrategyBuilder` crea una instancia de `RoundRobinStrategy` a partir de la siguiente estructura JSON:

```

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

```



La clave `timeslot` es opcional, y su valor por defecto es 1.

La clase `MostCrowdedStrategyBuilder` crea una instancia de `MostCrowdedStrategy` a partir de la siguiente estructura JSON:

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

La clave `timeslot` es opcional y su valor por defecto es 1.

En las dos estructuras anteriores, la clave `timeslot` es opcional, con un valor por defecto de 1. Una vez que se definan las clases anteriores, se pueden usar para crear la factoría correspondiente de la siguiente forma:

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

Esta factoría se usará más tarde para parsear las estrategias asociadas a los cruces (ver el “builder” para un cruce que se explica más abajo).

### **Factoría para definir las estrategias de extracción de la cola**

Para esta factoría necesitamos dos “builders”, `MoveFirstStrategyBuilder` y `MoveAllStrategyBuilder`, ambas extendiendo a `Builder<DequeuingStrategy>`, ya que ambos “builders” necesitan crear instancias de las clases que implementan a `DequeuingStrategy`.

La clase `MoveFirstStrategyBuilder` crea una instancia de `MoveFirstStrategy` a partir de la siguiente estructura JSON:

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

La clave `data` se puede omitir ya que no incluye ninguna información.

La clase `MoveAllStrategyBuilder` crea una instancia de `MoveAllStrategy` a partir de la estructura JSON:

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

La clave `data` se puede omitir ya que no incluye ninguna información.

Una vez definidas las clases anteriores, podemos utilizarlas para crear la factoría correspondiente de la siguiente forma:

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

**Factoría de eventos** Para esta factoría necesitamos un “builder” para cada clase de evento utilizado en la práctica, definidos en la sección *Eventos*. Todos los “builders” deben extender a `Builder<Event>`, ya que deben crear instancias de clases que extienden a `Event`.

La clase `NewJunctionEventBuilder` crea una instancia de `NewJunctionEvent` a partir de la siguiente estructura JSON:

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

La clave `coord` es una lista que contiene las coordenadas `x` e `y` (en este orden). Observa que su sección `data` incluye estructuras JSON para describir las estrategias que se deben usar. Asumimos que las factorías para estas estrategias se suministran a la constructora de este “builder”, es decir, su constructora debe declararse como sigue (más tarde veremos como pasar estas factorías a la constructora):

```
public NewJunctionEventBuilder(Factory<LightSwitchingStrategy> lssFactory, Factory<Dequeuing
```

La clase `NewCityRoadEventBuilder` crea una instancia de `NewCityRoadEvent` a partir de la siguiente estructura JSON:

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

```

        "weather" : "SUNNY"
    }
}

```

La clase `NewInterCityRoadEventBuilder` crea una instancia de `NewInterCityRoadEvent` a partir de:

```

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

```

Observa que las clases `NewCityRoadEventBuilder` y `NewInterCityRoadEventBuilder` tienen parte de su código igual, por ello podrías considerar hacer refactorización introduciendo una superclase.

La clase `NewVehicleEventBuilder` crea una instancia de `NewVehicleEvent` a partir de:

```

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

```

La clase `SetWeatherEventBuilder` crea una instancia de `SetWeatherEvent` a partir de la estructura JSON:

```

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

```

```

        ]
    }
}

```

La clase `SetContClassEventBuilder` crea una instancia de `SetContClassEvent` a partir de:

```

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

```

**Cómo Crear e Inicializar Las Factorías** Una vez implementadas las clases anteriores, podemos usarlas para crear la factoría correspondiente, utilizando el siguiente código:

```

// initialize the events factory
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);

```

### El controlador

El controlador se implementa en la clase `Controller`, dentro del paquete `simulator.control`. Esta clase es la responsable de:

- leer los eventos de un `InputStream` y añadirlos al simulador; y
- ejecutar el simulador un número determinado de pasos, mostrando los diferentes estados en un `OutputStream`.

La clase `Controller` contiene al menos los siguientes atributos (no públicos):

- *traffic simulator* (de tipo `TrafficSimulator`): utilizado para ejecutar la simulación.
- *events factory* (de tipo `Factory<Event>`): que se usa para parsear los eventos suministrados por el usuario.

Tiene sólo la constructora pública:

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

En la constructora debes comprobar que los valores de los parámetros no son null, y lanzar una excepción en caso contrario. Además esta clase tendrá los siguientes métodos:

- `public void loadEvents(InputStream in)`: asumimos que `in` incluye (el texto de) una estructura JSON de la forma:

```
{ "events": [e-1,...,e-n] }
```

donde cada `e-i` es a su vez una estructura JSON asociada a un evento. Este método primero convierte la entrada JSON en un objeto `JSONObject` utilizando:

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

y después extrae cada `e_i` de `jo`, crea el evento correspondiente `e` utilizando la *factoría de eventos*, y lo añade al simulador invocando al método `addEvent`. Este método debe lanzar una excepción si la entrada JSON no encaja con la de arriba.

- `public void run(int n, OutputStream out)`: ejecuta el simulador `n` ticks, llamando al método `advance` exactamente `n` veces, y escribe los diferentes estados en `out` utilizando el siguiente formato JSON:

```
{ "states": [s-1,...,s-n] }
```

donde `s-i` es el estado del simulador **después de** ejecutar el paso `i`. Observa que el estado `s_i` se obtiene llamando al método `report()` del simulador de tráfico.

- `public void reset()`: invoca al método `reset` del simulador de tráfico.

## Clase Main

Te facilitaremos, junto con la práctica, un esqueleto de la clase `Main`, que usa una librería externa para simplificar el parseo de las opciones de línea de comandos. La clase `Main` debe aceptar las siguientes opciones por línea de comandos:

```
> 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
```

El primer ejemplo escribe la salida en un fichero `output.json`, mientras que el segundo escribe la salida en la consola. En ambos casos, el fichero de entrada, conteniendo los eventos, es `eventsfile.json` y el número de ticks es 100. Simplemente tienes que completar los métodos `initFactories` y `startBatchMode`, y añadir código para procesar la opción `-t` (estudiando como se ha hecho para otras opciones). Observa que es un argumento opcional, que en caso de no ser proporcionado, tendrá como valor 10.

## Pruebas unitarias

El archivo `tests.zip` incluye pruebas de JUnit que tu código debe pasar. **No está permitido modificar las pruebas.**

## Otros comentarios

- El directorio `resources/examples` incluye un conjunto de ejemplos, junto con su salida esperada, que puedes usar para comprobar tu práctica (ver `resources/examples/README.md` para más detalles). Posiblemente añadiremos más ejemplos antes de la fecha de entrega. Tu práctica debe pasar todos los tests suministrados.
- Para convertir un `String s` a su correspondiente valor `enum`, por ejemplo de tipo `Weather`, usa `Weather.valueOf(s)` – o mejor usa `s.toUpperCase()` para evitar problemas de mayúsculas/minúsculas, asumiendo que todos los valores del tipo `enum` están definidos usando mayúsculas (esto es cierto para `Weather` y `VehicleStatus`).
- Para escribir fácilmente en un `OutputStream out`, primero crea un `PrintStream` utilizando `PrintStream p = new PrintStream(out);` y después usa comandos como `p.println("...")`, `p.print("...")`, etc.
- Para transformar un `JSONObject jo` a `String`, utiliza `jo.toString()` o `jo.toString(3)`. El primero es compacto, no escribe espacios en blanco. El segundo muestra la estructura JSON, donde el argumento es el número de espacios que añade a cada nivel de indentación.