Assignment 1: Physics Simulator

Submission date: 6th of March 2023, 09:00am

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

1. Copy detection

For each of the TP assignments, all the submissions from all the different TP groups will be checked using anti-plagiarism software, by comparing all of them pairwise and by searching to see if any of the code of any of them is copied from other sources on the Internet[1]. 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 TP-course exam session (*convocatoria*) to which the assignment belongs.
- A grade of zero for all the TP-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.
2. Fill in your name(s) in the file "`NAMES.txt`", each member in a separated line.

[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.

3. You have to strictly follow the package structure and class names that we suggest.

4. Submit a **zip** file of the project's directory, including all sub-directories except the **bin** directory. **Other formats (e.g., `7zip`, `rar`, etc.) are not accepted**.

3. Overview of the physics simulator

In this assignment you will develop a simulator for some *laws of physics* in a *2-dimensional* space. Briefly, the simulator has the follow components:

- *Bodies*, which represent physical entities (e.g., planets) that have mass, position, velocity, and a (total) force that is applied on them. They can *move*, when asked, to modify their position according to some motion laws.

- *Force laws*, which apply forces to bodies (e.g., gravitation).

- *Body Groups*, which represent groups of bodies (e.g., galaxies). Each group will have its own force laws.

We will use object oriented design to allow for several kinds of bodies, and several kinds of force laws. We will also implement factories, for both bodies and force laws, using Java generics. A simulation step consists of the following for each group: first resetting the force on each body; apply the force laws in order to change the force applied on each body, and then ask each body to *move*. In this assignment:

- The input is: (a) a `JSON` file describing the groups, force laws for each group, and a list of bodies; (b) the number of simulation steps to be performed.

- The output is a `JSON` structure describing the state of the bodies at the beginning and after each simulation step.

In the directory `resources` you can find some example input files, and some corresponding expected output files (see Section 6.1). You should make sure that your implementation produces a similar output on these examples. See also Section 6.3 for a corresponding viewer that will help you to visualize the output (you will implement a similar one in the next assignment).

4. Some necessary background material

4.1. Motion, Forces, and Gravity

It is recommended to read the following related material on motion and gravity, but you can carry out this assignment without reading it as well:

- `https://en.wikipedia.org/wiki/Equations_of_motion`

- `https://en.wikipedia.org/wiki/Newton%27s_laws_of_motion`

- `https://en.wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation`

Operation	Description	In Java
Addition	$\vec{a} + \vec{b}$ is defined as the new vector $(a_1 + b_1, a_2 + b_2)$	a.plus(b)
Subtraction	$\vec{a} - \vec{b}$ is defined as a new vector $(a_1 - b_1, a_2 - b_2)$	a.minus(b)
Scalar multiplication	$\vec{a} \cdot c$ (or $c \cdot \vec{a}$), where c is a real number, is defined as a new vector $(c * a_1, c * a_2)$	a.scale(c)
Length	the length (or magnitude) of $\vec{a}$, denoted by $\|\vec{a}\|$, is defined as $\sqrt{a_1^2 + a_2^2}$	a.magnitude()
Direction	the direction of $\vec{a}$ is a new vector that goes in the same direction of $\vec{a}$ but its length is 1, i.e., it is defined as $\vec{a} \cdot \frac{1}{\|\vec{a}\|}$	a.direction()
Distance	the distance between $\vec{a}$ and $\vec{b}$ is defined as $\|\vec{a} - \vec{b}\|$	a.distanceTo(b)

Figure 1: Operations on vectors

4.2. Vectors and a corresponding implementation

A vector $\vec{a}$ is a point (a_1, a_2) in an 2-dimensional Euclidean space, where each a_i is a real number (i.e., of type double). A vector can be imagined as a line from the origin $(0, 0)$ to (a_1, a_2). In the package "simulator.misc" there is a class Vector2D that implements such a vector and provides corresponding operations (see Figure 1) that we will use in this assignment. **You are not allowed to modify anything in this class**. Note that the class is immutable, i.e., it is impossible to change the state of an instance after creating it – operations (like addition, subtraction, etc) return new instances.

4.3. 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:

{ "key$_1$": value$_1$, ..., "key$_n$": value$_n$ }

where key$_i$ is a sequence of characters (representing a key) and value$_i$ can be a number, a string, another JSON structure, or an array [o$_1$,...,o$_k$] where o$_i$ is a number, a string, a JSON structure, or an array of JSON structures. Here is an example:

```
{
  "type" : "mv_body",
  "data" : {
             "id" : "planet",
             "p"  : [0.0e00, 4.5e10],
             "v"  : [1.0e04, 0.0e00],
             "m" : 1.5e30
           }
}
```

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

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

use it to create JSON structures and convert them to strings. An example of using this library is available in the package "extra/json".

5. Implementing the physics simulator

In this section we describe the different classes (and interfaces) that you should implement in order to develop the physics simulator. You are strictly required to follow the suggested class and package names. Corresponding UML diagrams are available in resources/uml.

5.1. Bodies

Next we describe the different kinds of bodies that you should implement. All corresponding classes should be placed in the package "simulator.model" immediately (not in sub-packages).

The base class body

The base class is a class Body that represents a physical entity. It has (as **protected** fields) an identifier *id* (String), a group identifier *gid* (String), a velocity vector $\vec{v}$, a force vector $\vec{f}$, a position vector $\vec{p}$ and a mass m (double). All values are received by the constructor, except the force that is set initially to the zero vector. The constructor should though an IllegalArgumentException if (1) any of the parameters is null; (2) the identifier (or group identifier) does not include at least one character that is not a white-space – use s.trim().length()>0; and (3) the mass is not positive.

It also includes following methods:

- public String getId(): returns the body's identifier.
- public String getgId(): returns the body's group identifier.
- public Vector2D getVelocity(): returns the velocity vector.
- public Vector2D getPosition(): returns the position vector.
- public Vector2D getForce(): returns the force vector.
- public double getMass(): returns the mass.
- void addForce(Vector2D f): adds the force f to the force vector of the body (using the method plus of Vector2D).
- void resetForce(): sets the force vector of the body to $(0, 0)$.
- abstract advance(double dt): a method to move the body for dt seconds (the actual implementation is in the subclasses).
- public JSONObject getState(): returns the following JSON structure that includes the body's information:

  { "id": *id*, "m": *m*, "p": $\vec{p}$, "v": $\vec{v}$, "f": $\vec{f}$ }

- public String toString(): returns what is returned by getState().toString().

Note that methods that change the state of the object are *package protected*, this way we guarantee that no class outside the model (i.e., package "simulator.model") can modify the state of the corresponding objects.

Moving body

A moving body is class MovingBody, whose method advance is as follows:

1. computes the acceleration $\vec{a}$ using Newton's second law, i.e., $\vec{a} = \frac{\vec{f}}{m}$. However, if m is zero then set $\vec{a}$ to $(0, 0)$ instead.
2. changes the position to $\vec{p} + \vec{v} \cdot t + \frac{1}{2} \cdot \vec{a} \cdot t^2$
3. changes the velocity to $\vec{v} + \vec{a} \cdot t$.

Stationary Body

A stationary body is a class StationaryBody whose method advance does not do anything, i.e., the body does not move. Its velocity, is always be the vector $(0, 0)$.

Other kinds of bodies

If you like, invent new bodies with different behaviours.

5.2. Force laws

Next we describe the different kinds of force laws that you should implement. All corresponding classes and interfaces should be placed in the package "simulator.model" immediately (not in sub-packages). We will model force laws using an interface ForceLaws that has the follwoing single method:

- public void apply(List<Body> bodies)

This method, in the classes that implement ForceLaws, is supposed to add forces to the different bodies.

Newton's law of universal gravitation

This force law is a class NewtonUniversalGravitation, and it implements *Newton's law of universal gravitation* to change the force applied on each body. It has a single constructor that receives a parameter G that represents the gravitational constant (we will mainly use $G = 6.67 * 10^{-11}$, which is 6.67E−11 in Java syntax, but you should keep it as a parameter). The constructor should though an IllegalArgumentException if G is not positive.

This law states that two bodies B_i and B_j apply gravitational force on each other, i.e., pull each other. In particular, the force applied by body B_j on body B_i is defeind as

$$\vec{F}_{i,j} = \vec{d}_{i,j} \cdot f_{i,j}$$

where $\vec{d}_{i,j}$ is the direction of the vector $\vec{p}_j - \vec{p}_i$ and

$$f_{i,j} = \begin{cases} G * \frac{m_i * m_j}{|\vec{p}_j - \vec{p}_i|^2} & \text{if } |\vec{p}_j - \vec{p}_i|^2 > 0 \\ 0.0 & otherwise \end{cases}$$

I.e., we compute $\vec{d}_{i,j}$ and scale it by $f_{i,j}$. Method apply should add to each body the forces applied on it by other bodies.

Moving towards a fixed point

This force law is a class MovingTowardsFixedPoint, and it simulates a scenario in which we apply a force in the direction of a fixed point $\vec{c}$. The length of this force (i.e., how strong the body is pushed) is related a parameter g and the mass of the body. The values of $\vec{c}$ (Vector2D) and g (double) should be provided as parameters to the constructor. The constructor should though an IllegalArgumentException if $\vec{c}$ is null or g is not positive.

Technically, for body B_i, method apply adds the force

$$\vec{F_i} = m \cdot g \cdot \vec{d_i}$$

where $\vec{d_i}$ is the direction of the vector $\vec{c} - \vec{p_i}$. This will make body B_i moves towards $\vec{c}$ with an acceleration vector of length g.

No force

This force law is a class NoForce, and it simply does nothing, i.e., its apply method is empty. This means that bodies keep moving with a fixed initial velocity.

Other kinds of force laws

If you like, invent new force laws with different behaviours.

5.3. Body groups

A group of bodies is a class BodiesGroup that represents a group of element of type Body. It has (as **private** fields) a group identifier (String), a force laws object (ForceLaws), and a list of bodies (List<Body>). The constructor receives the identifier and the force laws. The constructor should though an IllegalArgumentException if any of the parameters is null, or the identifier does not include at least one character that is not a white-space – use s.trim().length()>0.

It also includes following methods:

- public String getId(): returns the group's identifier.
- void setForceLaws(ForceLaws fl): sets the force laws to fl. It also throws an IllegalArgumentException exception if fl is null.
- void addBody(Body b): adds the body b to the list of bodies. It should check that there is no other body in the group with the same identifier, otherwise it should throw an IllegalArgumentException exception (define the method equal in Body, so you can use contains(b) of the list of bodies). It also throws an IllegalArgumentException exception if b is null.
- void advance(double dt): applies one simulation step for the group as follows: (1) it calls method resetForce of each body; (2) it calls method apply of the force laws; and (3) it calls advance(dt) of each body. It should throw an IllegalArgumentException exception if dt is not positive.
- public JSONObject getState(): returns the following JSON structure that includes the groups's state:

$$\texttt{\{ "id": } id\texttt{, "bodies": [} bb_1\texttt{,...,} bb_n\texttt{] \}}$$

where *id* is the group identifier, and each bb_i is a JSON structure that is returned by calling getState() of the corresponding body.

- public String toString(): returns what is returned by getState().toString().

Note that methods that change the state of the object are *package protected*, this way we guarantee that no class outside the model (i.e., package "simulator.model") can modify the state of the corresponding objects.

5.4. The simulator class

The simulator is a class called PhysicsSimulator. It should be placed in the package "simulator.model" immediately (not in a sub-package). Its constructor receives the following parameters and keeps them in corresponding private fields:

- *Real time per step*: a number of type double representing the actual time (in seconds) that corresponds to a simulation step — also called delta-time — it will be passed to method advance of the groups. The constructor should throw an IllegalArgumentException exception if the value is non-positive.
- *Force laws*: an object of type ForceLaws to be used as a *default force laws* when adding groups. The constructor should throw an IllegalArgumentException exception if the value is null.

The class should maintain as well a *map of groups* where the key is a group identifier and the value is a group (Map<String,GroupBodies>), and the *current time* (of type double) which is initially set to 0.0. This class should provide the following methods:

- public void advance(): applies one simulation step as follows:
  1. calls advance(dt) of each group where dt is the *real time per step*; and
  2. increments the current time by dt seconds.

  It should throw an IllegalArgumentException if dt is negative.
- public void addGroup(String id): adds a new group with identifier id to the map. When creating the group, you should use the *default force laws*. It should check that there is no other group in the groups map with the same identifier, otherwise it should throw an IllegalArgumentException exception.
- public void addBody(Body b): add the body b to the group with identifier b.getgId(). It should throw an IllegalArgumentException exception if there is no group with identifier b.getgId().
- public void setForceLaws(String id, ForceLaws fl): changes the force laws of the group with identifier id to fl. It should throw an IllegalArgumentException exception if there is no group with identifier b.getgId().
- public JSONObject getState(): returns the following JSON structure that includes the simulator's state:

{ "time": *t*, "groups": [g_1, g_2, ...] }

Moving body

```
{
  "type": "mv_body",
  "data": {
    "id": "b1",
    "gid": "g1",
    "p": [0.0e00, 0.0e00],
    "v": [0.05e04, 0.0e00],
    "m": 5.97e24
  }
}
```

Stationary Body

```
{
  "type": "st_body",
  "data": {
    "id": "b1",
    "gid": "g2",
    "p": [-3.5e10, 0.0e00],
    "m": 3.0e28,
  }
}
```

Newton's law of universal gravitation

```
{
  "type": "nlug",
  "data": {
    "G" : 6.67e10-11
  }

}
```

Moving towards a fixed point

```
{
  "type": "mtfp",
  "data": {
    "c": [0,0],
    "g": 9.81
   }
}
```

No force

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


}
```

Figure 2: JSON for bodies and force laws

where t is the current time and g_i is the JSONObject returned by getState() of the i-th groups in the map. **The order of the groups should be the insertion order, for this, you should maintain a list of group identifiers (List<String>) because the keys in the map are not necessarily ordered**.

- public String toString(): returns what getState().toString() returns.

5.5. 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);
  public List<JSONObject> getInfo();
}
```

Method createInstance receives JSON structure describing the object to be created (see Figure. 2), 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 include 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.

Method getInfo() returns a list of JSON structures that describe what the factory can create (see below).

There are many ways to define a factory, we will see some during the course. For our purposes, we will design 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 _typeTag;
  protected String _desc;

  public Builder(String typeTag, String desc) {
    if (typeTag == null || desc == null ||
        typeTag.length() == 0 || desc.length() == 0)
          throw new IllegalArgumentException("Invalid type/desc");
  }

  String getTypeTag() {
    return _typeTag;
  }

  public JSONObject getInfo() {
    JSONObject info = new JSONObject();
    info.put("type", _typeTag);
    info.put("desc", _desc);
    return info;
  }

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

The field _typeTag corresponds to the "type" discussed above, and _desc simply describes what kind of objects this builder can create.

Classes that extend Builder<T> are responsible on assigning value to _type and _desc by calling the constructor of class Builder, and also on defining method createTheInstance to create an object of type T (i.e., an instance of a sub-class of T) if all required information is available in data, otherwise it throws an IllegalArgumentException describing with corresponding information on the missing/invalid data.

Method getInfo returns a JSON with two fields that corresponds to _typeTag and _desc. This will be used by method getInfo() of the factory.

Use the class Builder<T> to define the following concrete builders:

- MovingBodyBuilder that extends Builder<Body>, for creating instances of class Body.
- StationaryBodyBuilder that extends Builder<Body>, for creating instance of class StationaryBodyBuilder.

- NewtonUniversalGravitationBuilder that extends Builder<ForceLaws>, for creating instances of class NewtonUniversalGravitation. Key "G" is optional, with default value 6.67E-11.
- MovingTowardsFixedPointBuilder that extends Builder<ForceLaws>, for creating instances of class MovingTowardsFixedPoint. Keys "c" and "g" are optional, with default values $(0, 0)$ and 9.81 respectively.
- NoForceBuilder that extends Builder<ForceLaws>, for creating instances of class NoForce.

The corresponding JSON can be seen in Figure 2. All builders should throw corresponding IllegalArgumentException exceptions if the input data is not valid (e.g., vectors are not 2-dimensional or some key is missing).

Once the builders are ready we develop a corresponding generic *builder based factory*. Its a 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 Map<String,Builder<T>> _builders;
  private List<JSONObject> _buildersInfo;

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

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

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

  public void addBuilder(Builder<T> b) {
    // add and entry '' b.getTag() -> b'' to _builders.
    // ...

    // add b.getInfo() to _buildersInfo
    // ...
  }

  @Override
  public T createInstance(JSONObject info) {
    if (info == null) {
      throw new IllegalArgumentException("Invalid value for
          createInstance: null");
    }

    // Search for a builder with a tag equals to info.getString("type"), call its
    // createInstance method and return the result if it is not null. The value you
```

```
    // pass to createInstance is :
    //
    //     info.has("data") ? info.getJSONObject("data") : new getJSONObject()

    // If no builder is found or thr result is null ...
    throw new IllegalArgumentException("Invalid value for
        createInstance: " + info.toString());
  }

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

}
```

We will use this class to create two factories, one for bodies and on for force laws. The following is an example of how we can create a factory of bodies using the classes that we have developed:

```
ArrayList<Builder<Body>> bodyBuilders = new ArrayList<>();
bodyBuilders.add(new MovingBodyBuilder());
bodyBuilders.add(new StationaryBodyBuilder());
Factory<Body> bodyFactory = new BuilderBasedFactory<Body>(bodyBuilders);
```

5.6. The controller

The controller should be implemented as a class called Controller. It should be placed in the package "simulator.control" immediately (not in a sub-package). It is responsible on (1) reading the data from a given InputStream and adding them to the simulator; and (2) executing the simulator a predetermined number of steps and printing the different states to a given OutputStream.

The class receives in its constructor a *physics simulator* (i.e., an object of type PhyicsSimulator), *force laws factory* (i.e., an object of type Factory<ForcwLaws>), and a *bodies factory* (i.e., an object of type Factory<Body>). The class should provide the following methods:

- public void loadData(InputStream in): we assume that the inpit stream in includes a JSON structure of the form

  { "groups": [g_1,...], "laws": [l_1,...], "bodies": [bb_1,...] }

  where

  - each g_i is a group identifier.
  - each l_1 is a JSON of the form

    { "id": *id*, "laws": *laws*}

    where *id* is a group identifier, and *laws* is a JSON describing the corresponding force laws (see Figure 2).
  - each bb_i is a JSON structure defining a corresponding body (see Figure 2).

  This method first converts the input JSON into a JSONObject using

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

and then (1) calls addGroup of the simulator for each g_i; (2) calls setForceLaws of the simulator for each l_i (after transforming l_i to an objectpo of type ForceLaws using the corresponding factory); and calls addBody of the simulator for each bb_i (after transforming bb_i to an objectpo of type Body using the corresponding factory).

- public void run(int n, OutputStream out): it runs the simulator n steps, and prints to the different states to out in the following JSON format:

$$\{\ \texttt{"states":}\ \ [s_0, s_1, \ldots, s_n]\ \}$$

where s_0 is the state of the simulator before executing any step, and each s_i with $i \geq 1$ is the state of the simulator immediately after executing the i-th simulation step. Note that state s_i is obtained by calling getState() of the simulator. Note also that when calling this method with $n < 1$, the output should include s_0. See Section 6.2 for a convenient way to print into a OutputStream.

5.7. The main class

In the package "simulator/launcher" you can find an incomplete Main class, that you are required to complete. It is the main entry to the simulator, i.e., it should process the command-line arguments and start the simulation accordingly. The class already parses some command-line arguments using common-cli (a library that is included the directory lib and already imported into the project), however, you will extend it to parse some more arguments. You should first study this class to understand how to define and parse command-line arguments.

Executing Main with the command-line argument -h (or --help) would print something like the following on the console, but without options -o, -eo, and -s that you will be asked to implement:

```
usage: simulator.launcher.Main [-dt <arg>] [-fl <arg>] [-h] [-i <arg>] [-o
       <arg>] [-s <arg>]
 -dt,--delta-time <arg>   A double representing actual time, in seconds,
                          per simulation step. Default value: 2500.0.
 -fl,--force-laws <arg>   Default force laws to be used in the simulator.
                          Possible values: 'nlug' (Newton's law of universal
                          gravitation), 'mtfp' (Moving towards a fixed
                          point), 'nf' (No force). You can provide the
                          'data' json attaching :{...} to the tag, but
                          without spaces.. Default value: 'nlug'.
 -h,--help                Print this message.
 -i,--input <arg>         Bodies JSON input file.
 -o,--output <arg>        Output file, where output is written. Default
                          value: the standard output.
 -s,--steps <arg>         An integer representing the number of simulation
                          steps. Default value: 150.
```

This help text describes how to execute the simulator. For example, the following command-line arguments

```
-i resources/examples/input/ex1.json -o resources/tmp/myout.json -s 1000 -dt 3000 -fl nlug
```

would execute the simulator for 1000 steps with delta-time as 3000 seconds, using the default force laws nlug (Newton's law of universal gravitation), taking the input from the file resources/examples/input/ex1.json and writing the output to resources/tmp/myout.json. Let us some other useful options:

- If we replace the parameter "-fl nlug" by "-fl nlug:{G:6.67E-10}", it would use the gravitational constant 6.67E-10 instead of the default one — see method parseForceLawsOption to understand how "-fl nlug:{G:6.67E-10}" is converted to a corresponding JSONObject which is passed later to the corresponding factory.

See "resources/examples/expected_output/README.md" for the command-line arguments that were used to generate all output files in "resources/examples/expected_output" from the examples in "resources/examples/input".

The content of the class Main that we provide is not complete, you should complete it as follows:

- Add support for options -o and -s. For this you have to understand how we use the commons-cli library — start at method parseArgs.
- Complete method initFactories() to create and initialize the factories (fields _bodyFactory and _forceLawsFactory) – see the end of Section 5.5 for an example code.
- Complete method startBatchMode() such that:
  - it creates a simulator (instance of PhyicsSimulator), passing it a corresponding force laws and delta-time as specified by the options -fl and -dt.
  - it creates corresponding input and output streams as specified by the options -i and -o. Recall that if -o is not provided in the command line, System.out should be used to print on the console.
  - it creates a controller (instance of Controller), passing it the simulartor and the bodies factory.
  - load the data into the simulator by calling method loadData of the controller.
  - it starts the simulation, by calling method run of the controller and passing the corresponding argument.

6. Extras

6.1. Example input and output files

The directory "resources/examples/input" includes some input examples, and the directory "resources/examples/expected_output" includes some expected output when running the simulator on these examples using different command-line arguments – see "resources/examples/expected_ to understand which command-line arguments where used. Your implementation should produce a similar output, i.e., one that is accepted by the *epsilon equivalence* state comparator with a relatively small value for eps.

6.2. How to print to an OutputStream

Suppose out is a variable of type OutputStream, we can write to it in a convenient way using a corresponding PrinterStream as follows:

```
PrintStream p = new PrintStream(out);

p.println("{");
p.println("\"states\": [");

// run the sumulation n steps, etc.

p.println("]");
p.println("}");
```

6.3. Output visualization

As you have seen above, the output of your assignment is a `JSON` structure describing the different simulation states, which is not easy to read. In assignment 1 you will be asked to develop a graphical user interface in order to, among other things, visualize the states graphically with animation.

In the meantime, you can use resources/viewer/viewer.html to visualize the output of your program. It is an HTML file that uses JavaScript to perform the visualization, open it in an web-browser like Firefox, Safari, Internet Explorer, Chrome, or the one of Eclipse.