Assignment 1: Physics Simulator

Submission date: 9th of April 2021, 23:00 hrs

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 s physics* in a *2-dimensional* space. Briefly, the simulator has two main 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).

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 in first resetting the force applied on all objects; applying the force laws in order to change the force applied on each body, and then asking each body to *move*. In this assignment:

- The input is a file describing the list of bodies in `JSON` format; the force laws to be used; and 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. The comparison is done using the concept of state comparators as described in Section 5.4. 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 $\lvert\vec{a}\rvert$, 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 $\vec{a}\cdot\frac{1}{\lvert\vec{a}\rvert}$	a.direction()
Distance	the distance between $\vec{a}$ and $\vec{b}$ is defined as $\lvert\vec{a}-\vec{b}\rvert$	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:

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

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

```
{
  "type" : "basic",
  "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).

Basic body

The basic body is a class Body that represents a physical entity. It has (as protected fields) an identifier *id* (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. It also includes following methods:

- public String getId(): returns the body’s 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 $(0, 0)$.
- void move(double t): moves the body for t seconds as follows:
    1. computes the acceleration $\vec{a}$ using Newton’s second law, i.e., $\vec{a} = \frac{\vec{f}}{m}$. Howver, 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 $\vec{v} + \vec{a} \cdot t$.
- public JSONObject getState(): returns the following JSON structure that includes the body’s information:

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

- 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.

Mass losing body

This kind of body is a class MassLossingBody, and it simulates a body that loses mass every now and then as explained below. It extends Body, and has the following attributes:

- lossFactor: a number (double) between 0 and 1 indicating the mass loss factor.
- lossFrequency: a positive number (double) indicating the time interval (in seconds) after which the object loses mass.

Values for these attributes should be received by the constructor.

Method move behaves like the one of Body, but, in addition, *after moving* it checks if lossFrequency seconds passed since the last time the mass was reduced, in such case it reduces the mass *again* by lossFactor, i.e., the new mass is $m*(1-\text{lossFactor})$. This should be done as follows: use a counter c (initially 0.0) to accumulate time (i.e., parameter t of move), and when $c \geq$ lossFrequency apply the reduction and set it again to 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).

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

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

I.e., we compute $\vec{d}_{i,j}$ and scale it by 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.

Technically, for body B_i, method apply adds the force

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

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. 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 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 move of the bodies. The constructor should throw an IllegalArgumentException exception if the value is non-positive.
- *Force laws*: an object of type ForceLaws to be used in the simulation process. The constructor should throw an IllegalArgumentException exception if the value is null.

The class should maintain as well a list of bodies (of type List<Body>) 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) it calls method resetForce of each body; (2) it calls method apply of the force laws; (3) it calls move(dt) of each body where dt is the *real time per step*; and (4) finally increments the current time by dt seconds.
- public void addBody(Body b): adds the body b to the simulator. It should check that there is no other body in the list of bodies 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).
- public JSONObject getState(): returns the following JSON structure that includes the simulator's state:

{ "time": t, "bodies": [$json_1$, $json_2$, ...] }

where t is the current time and $json_i$ is the JSONObject returned by getState() of the i-th body in the list of bodies.

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

5.4. State Comparators

The purpose of this section is to provide a way to check, during the simulation, if two states (returned by getState() of PhysicsSimulator) are equal. All classes/interfaces described in this section should be placed in the package "simulator.control". We will represent a state comparator using the following interface:

```
public interface StateComparator {
 boolean equal(JSONObject s1, JSONObject s2);
}
```

which includes a single method that returns true or false depending on if the states s1 and s2 (obtained from getState() of PhysicsSimulator) are equal or not. We will implement two kinds of comparators that define *state equality* in two different ways.

Mass Equality

This comparator should be implemented as a class called MassEqualStates. Two states s1 and s2 are equal if:

- The value of their "time" key is equal.
- The i-th bodies in the lists of bodies in s1 and s2 must have the same value for keys "id" and "mass".

Epsilon Equality

This comparator should be implemented as a class called EpsilonEqualStates. I has a single constructor that receive a value of type double (below we call it eps) and stores it in a corresponding field .

Two numbers a and b are eps-equal if "Math.abs(a-b) <= eps", and two vectors v1 and v2 are eps-equal if "v1.distanceTo(v2) <= eps". Two states s1 and s2 are equal if:

- The value of their "time" key is equal.
- the i-th bodies in the lists of bodies in s1 and s2 must have equal values for key "id", and eps-equal values for "m", "p", "v" and "f".

This comparator is useful because when performing calculations on data of type double, we might get slightly different results depending on the order in which we apply the operations (because of the use of *floating point arithmetic*). When comparing your output to the expected output, you can allow the values to be slightly different by changing the value of eps.

5.5. Factories

Now that we have defined the different classes for bodies, force laws, and comparators, we will develop factories in order to decouple their creation from the simulator. We will need three factories: one for bodies, one for force laws, and one for state comparators. We will develop these factories using Java generics since they have much in common. Next we describe how to develop these factories step by step. All classes and interfaces should be placed in the package "simulator.factories" immediately (not in sub-packages).

The factory interface

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

- public T createInstance(JSONObject info): receives a JSON structure describing the object to be created (see syntax below), 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 an IllegalArgumentException exception.

- public List<JSONObject> getInfo(): returns a list of JSON objects, which are "templates" for the valid JSON structures that can be passed to createInstance. This is very useful in order to know what are the valid values for a given factory without knowing much about the factory itself. For example, we will use it when listing the possible values for force laws to the user.

The JSON structure that is passed to createInstance includes two keys: *type*, which is a tag describing the type of the object to be created; and *data*, which is a JSON structure that includes all information needed to create the instance. Figure 2 includes a table with the JSON structures that we will use to create instances of bodies, force laws, and comparators.

The elements of the list returned by getInfo are JSON structures as those of Figure 2, with some default values for the keys in the *data* section (or instead of values, we can use strings that describe corresponding keys, if any). In addition, each includes a key *desc* with a string value that describes the template, for example, for Newton's law of universal gravitation we can use:

```
"desc": "Newton's law of universal gravitation"
```

Builders based factory

The concrete factories that we will develop are based on the use of builders. A builder is an object that is able to create an instance of a specific type, i.e., it can handle a JSON structure with a very specific value for key *type*. It is modeled as a generic class Builder<T> with methods:

- public T createInstance(JSONObject info): creates an object of type T (i.e., an instance of a sub-class of T) if it recognizes the information in info, otherwise it returns null to indicate that this builder cannot handle the request. In the case that it recognizes the *type* tag but there is an error in the values provided in the *data* section, it should throw an IllegalArgumentException exception.

- public JSONObject getBuilderInfo(): returns a JSON serving as a template for the corresponding builder, i.e., a valid value for the parameter of createInstance (see getInfo() of Factory<T> as well).

Basic body

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

Mass losing body

```
{
  "type": "mlb",
  "data": {
    "id": "b1",
    "p": [-3.5e10, 0.0e00],
    "v": [0.0e00, 1.4e03],
    "m": 3.0e28,
    "freq": 1e3,
    "factor": 1e-3
  }
}
```

Newton's law of universal gravitation

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

Moving towards a fixed point

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

No force

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

Mass Equal States Comparator

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

Epsilon Equal States Comparator

```
{
  "type": "epseq",
  "data": {
     "eps": 0.1
  }
}
```

Figure 2: `JSON` for bodies, force laws, and state comparators

Use this class to define the following 7 concrete builders:

- BasicBodyBuilder that extends Builder<Body>, for creating instances of class Body.
- MassLosingBodyBuilder that extends Builder<Body>, for creating instance of class MassLosingBody.
- 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.
- MassEqualStatsBuilder that extends Builder<StateComparator>, for creating instances of class MassEqualStates.
- EpsilonEqualStatesBuilder that extends Builder<StateComparator>, for creating instances of class EpsilonEqualStates. Key "`eps`" is optional, with default value 0.0.

All builders should throw corresponding exceptions if the input data is not valid (e.g., vectors are not 2-dimensional).

Once the builders are ready we develop a corresponding generic factory, which is a class BuilderBasedFactory<T> that implements Factory<T>. The constructor of this class receives a list of builders to be supported:

public BuilderBasedFactory(List<Builder<T>> builders)

Method createInstance of this factory tries the builders one by one until it succeed to create a corresponding instance — should throw an IllegalArgumentException exception in case of failure. Method getInfo() of this class aggregates the JSON structures returned by getBuilderInfo() of all builders in a list and returns it (*create the list and aggregate the information in the constructor, to avoid creating it every time*).

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 BasicBodyBuilder());
bodyBuilders.add(new MassLosingBodyBuilder());
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 bodies 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 (and compare the output to the expected one if required).

The class receives in its constructor a *physics simulator* (i.e., an object of type PhyicsSimulator) and *bodies factory* (i.e., an object of type Factory<Body>) that will be used to perform the different supported operations. The class should provide the following methods:

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

  { "bodies": [bb_1,...,bb_n] }

  where 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 extracts each bb_i from jsonInupt, creates a corresponding body b using the bodies factory, and adds it to the simulator by calling method addBody.

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

  { "states": [s_0,s_1,...,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.

The 3rd parameter expOut is an InputStream that corresponds to the expected output or null (the same syntax as the output described above). If expOut is not null, first convert it into a JSON structure, and then in each simulation step you should compare the current state of the simulator to the expected one using the state comparator of the 4-th argument, and if the result is not equal throw a corresponding exception with a message that includes the different states and the number of the execution step (better create your own exception class that encapsulate all this information).

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 [-cmp <arg>] [-dt <arg>] [-eo <arg>] [-fl
       <arg>] [-h] [-i <arg>] [-o <arg>] [-s <arg>]
 -cmp,--comparator <arg>       State comparator to be used when comparing
                               states. Possible values: 'epseq'
                               (Espsilon-equal states comparator),
                               'masseq' (Mass equal states comparator).
                               You can provide the 'data' json attaching
                               :{...} to the tag, but without spaces..
                               Default value: 'epseq'.
 -dt,--delta-time <arg>        A double representing actual time, in
                               seconds, per simulation step. Default
                               value: 2500.0.
 -eo,--expected-output <arg>   The expected output file. If not provided
                               no comparison is applied
 -fl,--force-laws <arg>        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/ex1.2body.json -o resources/output/myout.json -s 10000 -dt 3000 -fl nlug
```

would execute the simulator for 10000 steps with delta-time as 3000 seconds, using the force laws nlug (Newton's law of universal gravitation), taking the input from the file resources/examples/ex4.4body.json and writing the output to resources/output/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 parse-ForceLawsOption to understand how "-fl nlug:{G:6.67E-10}" is converted to a corresponding JSONOnject which is passed later to the corresponding factory.
- If we add the option "-eo /path/to/file", then output is compared to the one in the file "/path/to/file" using the default state comparator (epsilon equivalence with eps=0.0). You can change the value of "esp" to 0.5, for example, by using the option "-cmp epseq:{eps=0.5}".

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

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

- Add support for options -o, -eo, and -s. For this you have to understand how we use the commons-cli library — start at method parseArgs.
- Complete method init() to create and initialize the factories (fields _bodyFactory, _forceLawsFactory, and _stateComparatorFactory) – 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, -o, and -eo. Recall that if -o is not provided in the command line, System.out should be used to print on the console.
  - it creates a state comparator as specified in the option -cmp.
  - it creates a controller (instance of Controller), passing it the simulartor and the bodies factory.
  - it adds the bodies to the simulator by calling method loadBodies 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" includes some input examples, and the directory "resources/out" includes some expected output when running the simulator on these examples using different command-line arguments – see "resources/out/README.md" 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.

In “resources/out/README.md” you also have the command-lines that can be used to compare the output using the *epsilon equivalence* state comparator. You can change the value of eps if needed, but do not use large values!.

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.