Práctica 1: Simulador Físico

Fecha de entrega: 9 de abril de 2021 a las 23:00

Objetivo: Diseño Orientado a Objetos, Genéricos en Java y Colecciones.

1. Control de copias

Durante el curso se realizará control de copias de todas las prácticas, comparando las entregas de todos los grupos de TP. 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 se informará al Comité de Actuación ante Copias que citará al alumno infractor y si considera que es necesario sancionar al alumno propondrá una de las tres medidas siguientes:

- Calificación de cero en la convocatoria de TP a la que corresponde la práctica o examen.
- Calificación de cero en todas las convocatorias de TP del curso actual 2020/2021.
- Apertura de expediente académico ante la *Inspección de Servicios de la Universidad*.

2. Instrucciones generales

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

1. Descárgate del Campus Virtual la plantilla del proyecto que contiene el método main. Debes desarrollar la práctica usando esa plantilla.
2. Pon los nombres de los componentes del grupo en el fichero "`NAMES.txt`", cada miembro en una línea separada.
3. Debes seguir estrictamente la estructura de paquetes y clases sugerida por el profesor.

4. Cuando entregues la práctica, sube un fichero **zip** del proyecto, incluyendo todos los subdirectorios excepto el subdirectorio **bin**. **Otros formatos (por ejemplo `7zip`, `rar`, etc.) no están permitidos**.

3. Introducción al simulador físico

En esta práctica vamos a implementar un simulador para algunas *leyes de la física* en un espacio bidimensional (2D). El simulador tendrá dos componentes principales:

- *Cuerpos*, que representan entidades físicas (por ejemplo planetas), que tienen una velocidad, aceleración, posición y masa. Estos cuerpos, cuando se les solicite, se pueden *mover*, modificando su posición de acuerdo a algunas leyes físicas.

- *Leyes de fuerza*, que aplican fuerzas a los cuerpos (por ejemplo, fuerzas gravitacionales).

Utilizaremos un diseño orientado a objetos para poder manejar distintas clases de cuerpos y de leyes de fuerzas. Además, utilizaremos genéricos para implementar factorías, tanto para los cuerpos como para las leyes de fuerza.

Un paso de simulación consiste en inicializar la fuerza aplicada a cada cuerpo, aplicar las leyes de fuerza para cambiar las fuerzas aplicadas a los cuerpos, y después solicitar a cada cuerpo que se *mueva*. En esta práctica:

- La entrada será: (a) un fichero que describe una lista de cuerpos en formato `JSON`; (b) las leyes de fuerza que se van a usar y; (c) el número de pasos que el simulador debe ejecutar.

- La salida será una estructura `JSON` que describe el estado de los cuerpos al inicio y después de cada paso de la simulación.

En el directorio resources puedes encontrar algunos ejemplos de ficheros de entrada, con los correspondientes ficheros de salida (ver la Sección 6.1). Debes asegurarte de que tu implementación genera una salida parecida sobre estos ejemplos. La comparación puedes hacerla usando los *comparadores de estados* que se explican en la Sección 5.4. En la Sección 7 describimos un visor para poder ver de forma gráfica la simulación.

4. Material necesario para la práctica

4.1. Movimiento y gravedad

Recomendamos leer el siguiente material relacionado con el movimiento y la gravedad, aunque puedes implementar la práctica sin leerlo.

- `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`

Operación	Descripción	En Java
Suma	$\vec{a}+\vec{b}$ se define como el nuevo vector (a_1+b_1, a_2+b_2)	a.plus(b)
Resta	$\vec{a}-\vec{b}$ se define como el nuevo vector (a_1-b_1, a_2-b_2)	a.minus(b)
Multiplicación escalar	$\vec{a} \cdot c$ (o $c \cdot \vec{a}$), donde c es un número real, se define como el nuevo vector $(c * a_1, c * a_2)$	a.scale(c)
Longitud	la longitud (o magnitud) de $\vec{a}$, denotado como $\|\vec{a}\|$, se define como $\sqrt{a_1^2 + a_2^2}$	a.magnitude()
Dirección	la dirección de $\vec{a}$ es un nuevo vector que va en la misma dirección que $\vec{a}$, pero su longitud es 1, i.e., se define como $\vec{a} \cdot \frac{1}{\|\vec{a}\|}$	a.direction()
Distancia	la distancia entre $\vec{a}$ y $\vec{b}$ se define como $\|\vec{a} - \vec{b}\|$	a.distanceTo(b)

Figura 1: Operaciones sobre vectores.

4.2. Vectores: Implementación

Un vector $\vec{a}$ es un punto (a_1, a_2) en un espacio 2D, donde a_1, a_2 son números reales (i.e., de tipo double). Podemos imaginar un vector como una línea que va desde el origen de coordenadas $(0, 0)$ al punto (a_1, a_2).

En el paquete "simulator.misc" hay una clase Vector2D, que implementa un vector y ofrece operaciones para manipularlo. En la Figura 1) aparece la descripción de la clase Vector2D que vamos a utilizar. **No se puede modificar nada en esta clase**. La clase Vector2D es inmutable, es decir no es posible cambiar el estado de una instancia después de crearla – las operaciones (como suma, resta, etc.) devuelven nuevas instancias.

4.3. Análisis y creación de datos JSON en Java

JavaScript Object Notation[1] (JSON) es un formato estándar de fichero que utiliza texto y que permite almacenar propiedades de los objetos utilizando pares de atributo-valor y arrays de tipos de datos. Utilizaremos JSON para la entrada y salida del simulador. Brevemente, una estructura JSON es un texto estructurado de la siguiente forma:

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

donde $\texttt{key}_i$ es una secuencia de caracteres (que representa una clave) y $\texttt{value}_i$ puede ser un número, un *string*, otra estructura JSON, o un array $\texttt{[o}_1\texttt{,...,o}_k\texttt{]}$, donde $\texttt{o}_i$ puede ser un número, un *string*, una estructura JSON, o un array de estructuras JSON. Por ejemplo:

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

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

En el directorio lib hemos incluido una librería que permite analizar un fichero JSON y convertirlo en objetos Java. Esta librería ya está importada en el proyecto y se puede usar también para crear estructuras JSON y convertirlas a *strings*. Un ejemplo de uso de esta librería está disponible en el paquete "extra/json".

5. Implementación del simulador físico

En esta sección describimos las diferentes clases (e interfaces) que debes implementar para desarrollar el simulador físico. Se recomienda fuertemente que sigas la estructura de clases y paquetes que aparece en el enunciado. Puedes encontrar diagramas UML en resources/uml.

5.1. Cuerpos

A continuación detallamos los diferentes tipos de cuerpos (*Body*) que debes implementar. Todas estas clases deben colocarse dentro del paquete "simulator.model" (no en subpaquetes).

Cuerpo básico

Un **cuerpo básico** se implementa a través de la clase Body, que representa una entidad física. Un objeto de tipo Body contiene (como atributos protected) un identificador *id* (String), una vector de velocidad $\vec{v}$, un vector de fuerza $\vec{f}$, un vector de posición $\vec{p}$ y una masa m (double). La constructora de un cuerpo debe pasar los valores iniciales al objeto que crea, excepto la fuerza que se establece inicialmente como el vector $(0, 0)$. Además esta clase debe contener los siguientes métodos:

- public String getId(): devuelve el identificador del cuerpo.
- public Vector2D getVelocity(): devuelve el vector de velocidad.
- public Vector2D getForce(): devuelve el vector de fuerza.
- public Vector2D getPosition(): devuelve el vector de posición.
- public double getMass(): devuelve la masa del cuerpo.
- void addForce(Vector2D f): añade la fuerza f al vector de fuerza del cuerpo (usando el método plus de la clase Vector2D).
- void resetForce(): pone el valor del vector de fuerza a $(0, 0)$.
- void move(double t): mueve el cuerpo durante t segundos utilizando los atributos del mismo. Concretamente:
  1. calcula la aceleración $\vec{a}$ usando la segunda ley de Newton, i.e., $\vec{a} = \frac{\vec{f}}{m}$. Sin embargo, si m es cero, entonces se pone $\vec{a}$ a $(0, 0)$.
  2. cambia la posición a $\vec{p} + \vec{v} \cdot t + \frac{1}{2} \cdot \vec{a} \cdot t^2$ y la velocidad a $\vec{v} + \vec{a} \cdot t$.
- public JSONObject getState(): devuelve la siguiente información del cuerpo en formato JSON (como JSONObject):

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

- public String toString(): devuelve getState().toString().

Observa que los métodos que cambian el estado del objeto son *package protected*. De esta forma se garantiza que ninguna clase fuera del modelo puede modificar el estado de los objetos correspondientes.

Cuerpo que pierde masa

Representamos estos cuerpos a través de la clase MassLossingBody. Estos cuerpos se caracterizan porque pierden, con cierta frecuencia, masa cuando se mueven. La clase MassLossingBody extiende a Body, y tiene los siguientes atributos:

- lossFactor: un número (double) entre 0 y 1 que representa el factor de pérdida de masa.
- lossFrequency: un número positivo (double) que indica el intervalo de tiempo (en segundos) después del cual el objeto pierde masa.

Los valores para estos atributos deben obtenerse a través de su constructora. El método move se comporta como el de Body, pero además *después de moverse*, comprueba si han pasado lossFrequency segundos desde la última vez que se redujo la masa del objeto. En tal caso, se reduce la masa de nuevo en lossFactor, i.e, la nueva masa será $m * (1 - \text{lossFactor})$. Para implementar este proceso debes hacer lo siguiente: usa un contador c (inicializado a 0,0) para acumular el tiempo (i.e., el parámetro t de move) y cuando $c \geq$ lossFrequency aplica la reducción y pon de nuevo c a 0,0.

Otras clases de cuerpos

Si quieres, puedes inventarte nuevas clases de cuerpos con diferentes comportamientos.

5.2. Leyes de fuerza

En esta sección describimos las diferentes clases de leyes de fuerza que debes implementar. Todas las clases e interfaces deben colocarse en el paquete "simulator.model" (no en un subpaquete). Para modelar las leyes de la gravedad utilizaremos una interfaz ForceLaws, que tiene únicamente el siguiente método:

- public void apply(List<Body> bodies)

Este método, en las clases que implementan esta interfaz, debe aplicar fuerzas a los distintos cuerpos de la lista que va como parámetro.

Ley de Newton de la gravitación universal

Implementaremos esta ley en una clase NewtonUniversalGravitation, que cambiará la aceleración de los cuerpos de la siguiente forma: dos cuerpos B_i y B_j aplican una fuerza gravitacional uno sobre otro, i.e., se atraen mutuamente. Supongamos que $\vec{F}_{i,j}$ es la fuerza aplicada por el cuerpo B_j sobre el cuerpo B_i (más tarde veremos como se calcula). La fuerza total aplicada sobre B_i se define como la suma de todas las fuerzas aplicadas sobre B_i por otros cuerpos, i.e., $\vec{F}_i = \sum_{i \neq j} \vec{F}_{i,j}$. Ahora, usando la segunda ley de Newton, i.e.,

$\vec{F} = m \cdot \vec{a}$, podemos concluir que la aplicación de $\vec{F_i}$ sobre B_i cambia su aceleración a $\vec{F_i} \cdot \frac{1}{m_i}$. Como un caso especial, si m_i es igual a 0,0, ponemos los vectores de aceleración y velocidad de B_i a $\vec{o} = (0, 0)$ (sin necesidad de calcular $\vec{F_i}$).

Vamos a explicar ahora como calcular $\vec{F_{i,j}}$. Según la ley de la gravitación universal de Newton, los cuerpos B_i y B_j generan una fuerza, uno sobre otro, que es igual a:

$$f_{i,j} = G * \frac{m_i * m_j}{|\vec{p_j} - \vec{p_i}|^2}$$

donde G es la constante gravitacional, que es aproximadamente $6{,}67 * 10^{-11}$ (6,67E−11 usando la sintaxis de Java). Observa que $|\vec{p_j} - \vec{p_i}|$ es la distancia entre los vectores $\vec{p_i}$ y $\vec{p_j}$, i.e., la distancia entre los centros de B_i y B_j. Ahora, para calcular la dirección de esta fuerza, convertimos $f_{i,j}$ en $\vec{F_{i,j}}$ como sigue: Sea $\vec{d_{i,j}}$ la dirección de $\vec{p_j} - \vec{p_i}$, entonces $\vec{F_{i,j}} = \vec{d_{i,j}} \cdot f_{i,j}$.

Avanzando hacia un punto fijo

Esta ley de gravedad se implementa en la clase MovingTowardsFixedPoint. La ley simula un escenario en el cual todos los cuerpos caen hacia el "centro del universo", i.e. tienen una aceleración fija de $g = 9{,}81$ en dirección al origen $\vec{o} = (0, 0)$. Técnicamente, para un cuerpo B_i, asumiendo que $\vec{d_i}$ es su dirección, su aceleración debería ponerse a: $-g \cdot \vec{d_i}$.

Sin fuerza

Esta ley de gravedad se implementa en la clase NoForce. Simplemente no hace nada, i.e., su método apply está vacío. Esto significa que los cuerpos se mueven con una aceleración fija.

Otras leyes de la gravedad

Si quieres, puedes inventarte e implementar otras leyes de la gravedad.

5.3. La clase simulador

Para implementar el simulador vamos a utilizar la clase PhysicsSimulator, dentro del paquete "simulator.model" (no en subpaquetes). Su constructora tiene los siguientes parámetros, para inicializar los campos correspondientes:

- *Tiempo real por paso*: un número de tipo double que representa el tiempo (en segundos) que corresponde a un paso de simulación — se pasará al método move de los cuerpos. Debe lanzar una excepción IllegalArgumentException en caso de que el valor no sea válido.
- *Leyes de fuerza:* un objeto del tipo ForceLaws, que representa las leyes de fuerza que el simulador aplicará a los cuerpos. Si el valor es null, debe lanzar una excepción del tipo IllegalArgumentException.

La clase debe mantener además una lista de cuerpos de tipo List<Body> y el tiempo actual, que inicialmente será 0,0. Esta clase ofrece los siguientes métodos:

- public void advance(): aplica un paso de simulación, i.e.,

1. llama al método resetForce de todos los cuerpos,
2. llama al método apply de las leyes de fuerza,
3. llama a move(dt) para cada cuerpo, donde dt es el *tiempo real por paso*, y
4. finalmente incrementa el tiempo actual en dt segundos.

- public void addBody(Body b): añade el cuerpo b al simulador. El método debe comprobar que no existe ningún otro cuerpo en el simulador con el mismo identificador. Si existiera, el método debe lanzar una excepción del tipo IllegalArgumentException.
- public JSONObject getState(): devuelve el siguiente objeto `JSON`, que representa un estado del simulador:

  $$\texttt{\{ "time": } T\texttt{, "bodies": [}json_1\texttt{, }json_2\texttt{, ...] \}}$$

  donde T es el tiempo actual y $json_i$ es el JSONObject devuelto por el método getState del i-ésimo cuerpo en la lista de cuerpos.
- public String toString(): devuelve getState().toString().

5.4. Comparadores de estados

En esta sección describimos una manera de comprobar, durante la simulación, si dos estados (devueltos por el método getState() de la clase PhysicsSimulator) son iguales. Todas las clases/interfaces que aparecen deben estar colocadas en el paquete simulator.control. Representamos un comparador de estados usando la interfaz:

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

cuyo método devuelve true o false dependiendo de si los estados s1 y s2 (calculados a partir del método getState() de la clase PhysicsSimulator) son iguales o distintos. Implementaremos dos clases de comparadores, que denominaremos como *igualdad de estados*, de dos formas diferentes.

Igualdad de masas

Este comparador tienes que implementarlo en la clase MassEqualStates. Dos estados s1 y s2 son iguales si:

- El valor de sus claves "`time`" son iguales.
- Para todo i, el i-ésimo cuerpo de la lista de cuerpos de s1 y el de s2 tienen el mismo valor en las claves "`id`" y "`mass`".

Igualdad módulo epsilon

Este comparador tienes que implementarlo en la clase EpsilonEqualStates. Tiene una única constructora que recibe como argumento un valor de tipo double, al que denominamos eps, y almacena dicho valor en el atributo correspondiente.

Dos números a y b son iguales módulo eps si "Math.abs(a-b) <= eps", y dos vectores v1 y v2 son iguales módulo eps si "v1.distanceTo(v2) <= eps". Entonces decimos que dos estados s1 y s2 son iguales si:

- El valor de sus claves "`time`" son iguales.
- Para todo i, el i-ésimo cuerpo en la lista de cuerpos de s1 y s2 tienen el mismo valor en sus claves "`id`", y las claves"`m`", "`p`", "`v`" y "`f`" son iguales módulo epsilon.

Este comparador es útil ya que cuando se ejecutan las operaciones sobre datos de tipo double, se puede perder algo de precisión, lo que provoca que se obtengan resultados ligeramente diferentes dependiendo del order en que las operaciones se han realizado. Cuando compares tu salida con la salida esperada, se admiten valores ligeramente diferentes cambiando el valor de eps.

5.5. Factorías

Una vez que hemos definido las diferentes clases de cuerpos y leyes de la gravedad, utilizaremos factorías para separar su creación desde el simulador. Necesitamos dos factorías: una para los cuerpos y otra para las leyes de la gravedad. Las factorías las implementaremos usando genéricos, ya que tienen una parte en común. A continuación detallamos cómo implementar estas factorías paso a paso. Todas las clases e interfaces deben colocarse en el paquete "simulator.factories" (no en subpaquetes).

La interfaz "Factory"

Modelamos esta factoría a través de la interfaz genérica Factory<T>, con los siguientes métodos:

- public T createInstance(JSONObject info): recibe una structura `JSON` que describe el objeto a crear (ver la sintaxis más abajo), y devuelve una instancia de la clase correspondiente – una instancia de un subtipo de T. En caso de que info sea incorrecto, entonces lanza una excepción del tipo IllegalArgumentException.
- public List<JSONObject>getInfo(): devuelve una lista de objetos `JSON`, que son "plantillas" para structuras `JSON` válidas. Los objetos de esta lista se pueden pasar como parámetro al método createInstance. Esto es muy útil para saber cuáles son los valores válidos para una factoría concreta, sin saber mucho sobre la factoría en si misma. Por ejemplo, utilizaremos este método cuando mostremos al usuario los posibles valores de las leyes de la gravedad.

La estructura `JSON` que se pasa como parámetro al método createInstance incluye dos claves: *type*, que describe el tipo de objeto que se va a crear y; *data*, que es una estructura `JSON` que incluye toda la información necesaria para crear el objeto. En la Figura 2 se incluye una tabla con las estructuras `JSON` que utilizaremos para crear instancias de cuerpos y leyes de la gravedad.

Los elementos de la lista que devuelve getInfo son estructuras `JSON` de las listadas en la Figura 2, con algunos valores por defecto para las claves de la sección *data* (o en lugar de valores, podemos usar *strings* que describan las clases correspondientes). Además cada elemento de la lista incluye una clave *desc*, que contiene un *string* para describir la plantilla. Por ejemplo, para la ley de Newton de la gravitación universal, podemos usar:

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

Cuerpo básico

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

Cuerpo que pierde masa

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

Ley de Newton

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

Movimiento hacia un punto fijo

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

Sin fuerza

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

Igualdad de masas

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

Igualdad módulo epsilon

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

Figura 2: Formato JSON para cuerpos, leyes de la gravedad, y comparadores de estados

Factorías basadas en constructores

Las factorías concretas que vamos a desarrollar están basadas en el uso de constructores. Un constructor es un objeto que es capaz de crear una instancia de un tipo específico, i.e., puede manejar una structura JSON con un valor concreto para la clave *type*. Un constructor se modela usando la clase genérica Builder<T>, con los siguientes métodos:

- public T createInstance(JSONObject info): si la información suministrada por info es correcta, entonces crea un objeto de tipo T (i.e., una instancia de una subclase de T). En otro caso devuelve null para indicar que este constructor es incapaz de reconocer ese formato. En caso de que reconozca el campo *type* pero haya un error en alguno de los valores suministrados por la sección *data*, el método lanza una excepcion IllegalArgumentException.
- public JSONObject getBuilderInfo(): devuelve un objeto JSON que sirve de plantilla para el correspondiente constructor, i.e., un valor válido para el parámetro de createInstance (ver getInfo() de Factory<T>).

Usa esta clase para definir los siguientes constructores:

- BasicBodyBuilder que extiende a Builder<Body>, para crear objetos de la clase Body.
- MassLosingBodyBuilder que extiende a Builder<Body>, para crear objetos de la clase MassLosingBody.

- NewtonUniversalGravitationBuilder que extiende a Builder<ForceLaws>, para crear objetos de la clase NewtonUniversalGravitation. La clave "G" es opcional, con valor por defecto 6.67E-11.
- MovingTowardsFixedPointBuilder que extiende a Builder<ForceLaws>, para crear objetos de la clase MovingTowardsFixedPoint. Las claves "c" y "g" son opcionales, con valores por defecto $(0, 0)$ y 9,81 respectivamente.
- NoForceBuilder que extiende a Builder<ForceLaws>, para crear objetos de la clase NoForce.
- MassEqualStatesBuilder que extiende a Builder<StateComparator>, para crear objetos de la clase MassEqualStates.
- EpsilonEqualStatesBuilder que extiende a Builder<StateComparator>, para crear objetos de la clase EpsilonEqualStates. La clave "eps" es opcional, con valor por defecto 0.0.

Todos los "builders" deben lanzan excepciones cuando los datos de entrada no sean válidos. Por ejemplo si los vectores no son 2D.

Una vez que los constructores están preparados, implementamos una factoría genérica BuilderBasedFactory<T>, que implementa a Factory<T>. La constructora de esta clase recibe como parámetro una lista de constructores:

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

El método createInstance de la factoría ejecuta los constructores uno a uno hasta que encuentre el constructor capaz de crear el objeto correspondiente — debe lanzar una excepcion IllegalArgumentException en caso de fallo. El método getInfo() devuelve en una lista las estructuras JSON devueltas por getBuilderInfo().

El siguiente ejemplo muestra como se puede crear una factoría de cuerpos usando las clases que hemos desarrollado:

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

5.6. El controlador

El controlador se implementa en la clase Controller, dentro del paquete "simulator.control" (no en un subpaquete). Es el encargado de (1) leer los cuerpos desde un InputStream dado y añadirlos al simulador; (2) ejecutar el simulador un número determinado de pasos y mostrar los diferentes estados de cada paso en un OutputStream dado. La clase recibe en su constructora un objeto del tipo PhysicsSimulator, que se usará para ejecutar las diferentes operaciones y un objeto del tipo Factory<Body>, para construir los cuerpos que se leen del fichero. La clase Controller ofrece los siguientes métodos:

- public void loadBodies(InputStream in): asumimos que in contiene una estructura JSON de la forma:

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

donde bb_i es una estructura JSON que define un cuerpo de acuerdo a la sintaxis de la Figura 2. Este método primero transforma la entrada JSON en un objeto JSONObject, utilizando:

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

y luego extrae cada bb_i de jsonInupt, crea el correspondiente cuerpo b usando la *factoría de cuerpos*, y lo añade al simulador llamando al método addBody.

- public void run(int n, OutputStream out, InputStream expOut, StateComparator cmp): ejecuta el simulador n pasos, y muestra los diferentes estados obtenidos en out, utilizando el siguiente formato JSON:

  { "states": [s_0,s_1,...,s_n] }

  donde s_0 es el estado inicial del simulador, y cada s_i, $i \geq 1$, es el estado de la simulación obtenido después de ejecutar el paso i-ésimo. Observa que el estado s_i se obtiene llamando al método getState() del simulador. Por otro lado, cuando se llama a este método con $n < 1$, la salida debería incluir s_0. Lee detenidamente la Sección 6.2 para imprimir de forma correcta en un OutputStream.

  El tercer parámetro expOut es un InputStream que contiene la salida esperada o null. Si expOut no es null, primero debes convertirlo en una estructura JSON, y después, en cada paso de la simulación, debes comparar la estructura actual del simulador con la esperada usando el comparador de estados recibido en el cuarto argumento, y si el resultado no es igual, lanzar una excepción con un mensaje que incluya los estados que son diferentes y el paso de la ejecución en el se ha producido la diferencia (crea tu propia excepción para encapsular esta información).

5.7. La clase Main

En el paquete "simulator.launcher" puedes encontrar una versión incompleta de la clase Main. Esta clase procesa los argumentos de la línea de comandos e inicia la simulación. La clase también analiza algunos argumentos de la línea de comandos utilizando la librería common-cli (incluida en el directorio lib y ya importada en el proyecto). Tendrás que extender la clase Main para analizar todos los posibles argumentos.

Al ejecutar Main con el argumento -h (o --help) debe mostrar por consola lo siguiente:

```
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.
```

Como ejemplo de uso del simulador utilizando la línea de comandos, mostramos:

```
-i resources/examples/ex1.2body.json -o resources/output/myout.json
                                     -s 10000 -dt 3000 -fl nlug
```

que ejecuta el simulador 1000 pasos con un valor para `dt` de 3000 segundos, usando las leyes de la gravedad nlug (leyes de Newton), donde el fichero de entrada es resources/examples/ex1.2body.json, y el fichero de salida es resources/output/myout.json.

Veamos ahora otras opciones de gran utilidad:

- Si reemplazamos el parámetro "-fl nlug" por "-fl nlug:{G:6.67E-10}", entonces se usará la constante gravitacional 6.67E-10 en lugar de la constante por defecto – ver método parseForceLawsOption para entender como "-fl nlug:{G:6.67E-10}" se convierte a un JSONOnject, que más tarde se pasará a la factoría correspondiente.

- Si añadimos la opción "-eo /path/to/file", entonces la salida se compara con la contenida en el fichero "/path/to/file", usando el comparador de estados por defecto (igualdad módulo epsilon con eps=0.0). Por ejemplo, puedes cambiar el valor de "esp" a 0.5, usando la opción "-cmp epseq:eps=0.5".

En el fichero "resources/output/README.md" puedes encontrar información sobre los argumentos de la línea de comandos que hemos usado para generar todos los ficheros de salida de "resources/out", para los ejemplos de "resources/examples".

La clase Main que os facilitamos no está completa. Tienes que completarla haciendo lo siguiente:

- Añade el código necesario para poder utilizar las opciones -o, -eo y -s. Para poder hacerlo, tienes previamente que entender cómo se usa la librería commons-cli — empieza por el método parseArgs.

- Completa el método init() para crear e inicializar las factorías (atributos _bodyFactory, _forceLawsFactory, y _stateComparatorFactory) – revisa la información que aparece al final de la Sección 5.5, donde hay un ejemplo de código.

- Completa el método startBatchMode() de forma que:

  - cree el simulador (una instancia de PhyicsSimulator), pasando como argumentos las leyes de la fuerza y el *delta time* (las opciones -fl y -dt respectivamente).

- cree los ficheros de entrada y salida tal y como vengan especificados por las opciones -i, -o, y -eo. Recuerda que si la opción -o no aparece en la línea de comandos, entonces se utiliza la salida por consola, i.e., System.out para mostrar la salida.
- cree un comparador de estados de acuerdo con la información que aparece en la opción -cmp.
- cree un controlador (instancia de la clase Controller), pasándole el simulador y la factoría de cuerpos.
- añada los cuerpos al simulador llamando al método loadBodies del controlador.
- inicie la simulación llamando al método run del controlador y pasándole los argumentos correspondientes.

6. Extra

6.1. Ejemplos de entrada y salida

El directorio "resources/examples" incluye algunos ficheros de entrada, y el directorio "resources/out" contiene las salidas esperadas cuando se ejecuta el simulador sobre los ficheros de entrada, usando diferentes opciones en la línea de comandos – mira el fichero "resources/out/README.md" para ver que comandos se han usado. Tu implementación debe generar una salida similar, i.e., una salida que sea válida si usamos el comparador módulo epsilon con un valor relativamente pequeño para eps.

En "resources/out/README.md" también tienes los comandos que hemos usado para comparar la salida usando el comparador módulo epsilon. Puedes cambiar el valor de eps si lo crees necesario, pero no uses valores grandes.

6.2. Cómo escribir en un OutputStream

Supongamos que out es una variable del tipo OutputStream. Para escribir en ella es conveniente usar un PrinterStream de la siguiente forma:

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

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

// run the sumulation n steps, etc.

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

7. Visualización de la salida

Como habrás observado, la salida de la práctica es una estructura JSON que describe los diferentes estados de la simulación, y que no es fácil de leer. En la Práctica 5 desarrollaremos una interfaz gráfica que permitirá visualizar los estados con animación. Pero hasta entonces, puedes usar "resources/viewer/viewer.html" para visualizar la salida de tu

programa. Es un fichero HTML que utiliza JavaScript para ejecutar la visualización. Ábrelo con un navegador, como por ejemplo Firefox, Safari, Internet Explorer, Chrome, o el navegador de Eclipse.