Práctica 1: Simulador Físico

Fecha de entrega: 6 de marzo de 2023 a las 09:00am

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 2022/2023.
- 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).
- *Grupos de Cuerpos*, que representan conjuntos de cuerpos (por ejemplo, galaxias). Cada grupo tiene su ley de fuerza.

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 hacer lo siguiente para cada grupo: inicializar la fuerza de 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 los grupo, su leyes de fuerza, y sus cuerpos; y (b) 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. 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

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.

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.

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

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 estruc-

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

turas 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).

La clase base de cuerpos

La clase base de un cuerpo 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), un identificador de su grupo *gid* (String), un 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 $(0, 0)$. La constructora debe lanzar una excepción del tipo IllegalArgumentException en los siguientes casos: (1) cualquier parámetro es null, (2) *id* (o *gid*) no incluye al menos un carćter que no sea espacio en blanco – usa s.trim().length()>0, y (3) la masa no es positiva. Además esta clase debe contener los siguientes métodos:

- public String getId(): devuelve el identificador del cuerpo.
- public String getgId(): devuelve el identificador del grupo al que pertenece el 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 $(0, 0)$.
- abstract void advance(double dt): mueve el cuerpo durante dt segundos (las implementaciones están el las sub-clases).
- 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 en movimiento

Un **cuerpo en movimiento** se implementa a través de la clase MovingBody que extiende a Body, su método advance tiene el siguiete comportamiento:

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 $\vec{v} + \vec{a} \cdot t$.

Cuerpo estacionario

Representamos estos cuerpos a través de la clase StationaryBody que extiende a Body. Un cuerpo de este tipo no se mueve, es decir su método advance no hace nada.

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. La clase tiene una sola constructora que recibe un parámetro G (ver abajo). La constructora debe lanzar una excepción de tipo IllegalArgumentException el valor de G no es positivo.

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:

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

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 fuerza se implementa en la clase MovingTowardsFixedPoint, y simula un escenario en el cual se aplica una fuerza hacia un

punto fijo $\vec{c}$. Esta fuerza va a depender de un parámetro g y de la masa del cuerpo. Los valores de $\vec{c}$ (Vector2D) y g (double) se suministran a la clase a través de la constructora. La constructora debe lanzar una excepción de tipo IllegalArgumentException si $\vec{c}$ es null o g no es positivo.

Concretamente, para cada cuerpo b_i, el método apply añade la fuerza

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

al cuerpo b_i, donde $\vec{d_i}$ es la dirección del vector $\vec{c} - \vec{p_i}$. Esto provocará que el cuerpo b_i se mueva hacia $\vec{c}$ con una aceleración g.

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 aceleración nula.

Otras leyes de la gravedad

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

5.3. Body groups

Un grupo de cuerpos se implementa a través de la clase BodiesGroup, que representa un grupo de objetos de tipo Body. Un objeto de tipo BodiesGroup contiene (como atributos **private**) un identificador (String), leyes de fuerza (ForceLaws), y una lista de cuepos (List<Body>). La constructora debe recibir los valores iniciales del identificador y las leyes de fuerza. La constructora de lanzar una excepción del tipo IllegalArgumentException en los siguientes casos: (1) cualquier parámetro es null, (2) el identificador no incluye al menos un carćter que no sea espacio en blanco – usa s.trim().length()>0.

Además esta clase debe contener los siguientes métodos:

- public String getId(): devuelve el identificador del grupo.
- void setForceLaws(ForceLaws fl): cambia las leyes de fuerza a fl. Debe lanzar un excepción de tipo IllegalArgumentException si fl es null.
- void addBody(Body b): añade el cuerpo b a la lista de cuerpos. El método debe comprobar que no existe ningún otro cuerpo en el

grupo con el mismo identificador. Si existiera, el método debe lanzar una excepción del tipo IllegalArgumentException. Debe lanzar un excepción de tipo IllegalArgumentException si b es null.

- void advance(double dt): aplica un paso de simulación en el grupo, i.e.,

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

  Debe lanzar una excepción IllegalArgumentException en caso de que el valor de `dt` no sea positivo.

- public JSONObject getState(): devuelve el siguiente objeto `JSON`, que representa un estado del grupo:

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

  donde *id* es el identificador del grupo y bb_i es el JSONObject devuelto por el método getState del i-ésimo cuerpo en la lista de cuerpos.

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

5.4. 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 advance 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 usamos por defecto para cada grupo. Si el valor es null, debe lanzar una excepción del tipo IllegalArgumentException.

La clase debe mantener además un mapa de grupos donde la clave es el identificado del grupo y el valor es un grupo (Map<String,BodiesGroup>, y el tiempo actual, que inicialmente será 0,0. Esta clase ofrece los siguientes métodos:

- public void advance(): aplica un paso de simulación:
  1. llama a advance(dt) de todos los grupos donde dt es el *tiempo real por paso*, y
  2. incrementa el tiempo actual en dt segundos.
- public void addGroup(String id): añade un nuevo grupo con identificador id al mapa de grupos. Al cread el objeto de tipo BodiesGroup hay que pasarle la leyes de fuerza que usamos por defecto. El método debe comprobar que no existe ningún grupo en el simulador con el mismo identificador. Si existiera, el método debe lanzar una excepción del tipo IllegalArgumentException.
- public void addBody(Body b): añade el cuerpo b al grupo con identificador b.getgId(). El método debe lanzar una excepción del tipo IllegalArgumentException si no existe un grupo con dicho identificador.
- public void setForceLaws(String id, ForceLaws fl: cambia las leyes de la fuerza del grupo con identificador *id* a *fl*. El método debe lanzar una excepción del tipo IllegalArgumentException si no existe un grupo con dicho identificador.
- public JSONObject getState(): devuelve el siguiente objeto `JSON`, que representa un estado del simulador:

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

  donde t es el tiempo actual y g_i es el JSONObject devuelto por el método getState del i-ésimo grupo. **El orden de los grupos tiene que ser el orden de creación, para esto hay que mantener una lista de identificadores (List<String>) de gru-**

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": {}
}
```

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

pos porque no hay orden garantizado para las claves del mapa.

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

5.5. Factorías

Como en la práctica tenemos varías factorias vamos a usar genéricos para evitar duplicar código. A continuación detallamos cómo implementarlas paso a paso. Todas las clases e interfaces deben colocarse en el paquete "simulator.factories" (no en subpaquetes).

Una factoría se modela con la interfaz genérica Factory<T>:

```
package simulator.factories;

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

El método createInstance recibe una estructura JSON que describe el objeto a crear (ver Figura 2), y devuelve una instancia de la clase correspondiente — una instancia de un subtipo de T. En caso de que info sea incorrecto, entonces lanza la correspondiente excepción. En nuestro caso, la estructura JSON que se pasa como parámetro al método createInstance incluye dos claves:

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

El método getInfo() devuelve una lista de objetos JSON que describe qué puede ser creado por la factoria (ver detalles a continuación).

Existen muchas formas de definir una factoría, que veremos durante el curso. Nosotros la vamos a diseñar utilizando lo que se conoce como *builder based factory*, que permite extender una factoría con más opciones sin necesidad de modificar su código. El elemento básico en una *builder based factory* es el *builder*, que es una clase capaz de crear una instancia de un tipo específico. Podemos modelarla como una clase genérica Builder<T>:

```
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);
}
```

El atributo _typeTag hace referencia al campo "type" de la estructura JSON correspondiente, descrita arriba, y el atributo _desc describe que tipo de objetos pueden ser creados por este "builder".

Las clases que extienden a Builder<T> son las responsables de asignar un valor a _type llamando a la constructora de la clase Builder, y también de definir el método createTheInstance para crear un objeto del tipo T (o de cualquier instancia que sea subclase de T) en caso de

que toda la información necesaria se encuente disponible en data. En otro caso genera una excepción de tipo IllegalArgumentException describiendo que información es incorrecta o no se encuentra disponible.

El método getInfo devuelve un objeto JSON con dos campos correspondientes a _typeTag y _desc, el cual será utilizado por el método getInfo() de la factoría.

Utiliza la clase Builder<T> para definir los siguientes "builders" concretos:

- MovingBodyBuilder que extiende a Builder<Body>, crea instancias de la clase MovingBody.
- StationaryBodyBuilder que extiende a Builder<Body>, crea instancias de la clase StationaryBodyBuilder.
- NewtonUniversalGravitationBuilder que extiende a Builder<ForceLaws>, crea instancias de la clase NewtonUniversalGravitation. La clave "G" es opcional, con valor por defecto 6.67E-11.
- MovingTowardsFixedPointBuilder que extiende a Builder<ForceLaws>, crea instancias de la clase MovingTowardsFixedPoint. Las claves "c" y "g" son opcionales, con valor por defecto $(0, 0)$ y 9,81 respectivamente.
- NoForceBuilder que extiende a Builder<ForceLaws>, crea instancias de la clase NoForce.

El JSON correspondiente se muestra en la Figura 2. Todos los "builders" deben lanzan excepciones cuando los datos de entrada no sean válidos. Por ejemplo si los vectores no son 2D o si falta alguna clave en el JSON.

Una vez que los "builders" están preparados, implementamos una *builder based factory* genérica. Es una clase que tiene una lista the "builders", de tal forma que cuando queramos crear un objeto a partir de una estructura JSON, recorre todos los "builders" hasta que encuentra uno con el que poder generar la instancia correspondiente:

```
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 JSONObject()

    // 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);
  }

}
```

Utilizaremos esta clase para crear dos factorías, una para los cuerpos y otra para las leyes de la fuerza. Este ejemplo muestra como podemos crear una factoía de cuerpos con la clase que hemos implementado:

```
ArrayList<Builder<Body>> bodyBuilders = new ArrayList<>();
bodyBuilders.add(new MovingBodyBuilder());
bodyBuilders.add(new StationaryBodyBuilder());
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 datos desde un InputStream dado y añadirlos al simulador; (2) ejecutar el simulador un número determinado de pasos y mos-

trar 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, una factoría de cuerpos (Factory<Body>), y una factoría de leyes de fuerza Factory<ForceLaws>.

La clase Controller ofrece los siguientes métodos:

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

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

  donde

  - g_i es un identificador de grupo,
  - l_i es una estructura JSON de la forma

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

    donde *id* es un identificador de grupo, y *laws* es un JSON que define leyes de fuerza de acuerdo a la sintaxis de la Figura 2.
  - bb_i es una estructura JSON que define un cuerpo de acuerdo a la sintaxis de la Figura 2.

  La clave "law" es opcional. Este método primero transforma la entrada JSON en un objeto JSONObject, utilizando:

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

  y luego (1) llama a addGroup del simulador para cada g_i, (2) llama a setForceLaws del simulador para cada l_i (después de convertir l_i a un objeto de tipo ForceLaws usando la factoría), y (3) llama a addBody para cada bb_i (después de convertir bb_i a un objeto Body usando la factoría).

- public void run(int n, OutputStream out): 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.

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 [-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.
```

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

```
  -i resources/examples/input/ex1.json -o resources/tmp/myout.json
                                       -s 1000 -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/input/ex1.json, y el fichero de salida es resources/tmp/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.

Para el caso de la ley de fuerza mtfp observa que hay dos parámetros opcionales: el *centro* y la *aceleración*, cuyos valores por defecto son $(0, 0)$ y $9{,}81$ respectivamente. Estos parámetros, se pueden introducir a través de la línea de comandos, utilizando la sintaxis -fl mtfp:{g:39,c:[3,4]}. Ten en cuenta que se pueden introducir los dos parámetros (el orden es irrelevante), ninguno (se usan los valores por defecto), o uno de ellos (para el otro se usa el valor por defecto).

En el fichero "resources/examples/expected_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/examples/expected_output", para los ejemplos de "resources/examples/inpu

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 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 y _forceLawsFactory) – 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 y -o. 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 controlador (instancia de la clase Controller), pasándole el simulador, la factoría de cuerpos y la factoría de leyes de fuerza.

- carga los datos en el similador llamando al método loadData 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/input" incluye algunos ficheros de entrada, y el directorio "resources/examples/expected_output" 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/examples/expected_output/README.md" para ver que comandos se han usado.

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 2 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.