UNIVERSIDAD
COMPLUTENSE
MADRID

Tecnología de la Programación

Presentación de la Práctica 4

(Basado en la práctica de Samir Genaim)

Ana M. González de Miguel (ISIA, UCM)

Índice

1. Introducción

- ✓ Los objetivos cubiertos en esta práctica son: **diseño orientado a objetos**, **colecciones** y **genéricos**.
- ✓ Fecha de entrega: **11 de Marzo** a las 9:00.
- ✓ Las siguientes instrucciones son obligatorias:
  - Descárgate del campus la **plantilla del proyecto** que contiene el método *main* además de otras estructuras que deberás completar.
  - Pon los nombres de los integrantes del grupo en el fichero NAMES.txt. Escribe cada nombre en una línea.
  - Sigue la estructura de paquetes y clases sugerida por el profesor e incluida en el material disponible de la práctica.
  - Cuando entregues la práctica en el Campus Virtual, sube un fichero **zip** del proyecto, incluyendo todos los subdirectorios excepto **bin**. Otros formatos como **rar** no están permitidos.

- ✓ En esta práctica se desarrolla un **simulador de leyes físicas** de la gravedad y del movimiento.
- ✓ Los componentes principales son:
  - **Cuerpos.** Representan entidades físicas con velocidad, aceleración, posición y masa. Pueden modificar su posición de acuerdo a algunas leyes físicas.
  - **Leyes de gravedad**. Especifican como las fuerzas gravitacionales cambian las propiedades de un cuerpo (por ejemplo, su aceleración).
- ✓ Utilizaremos el diseño orientado a objetos para implementar estos componentes. Además usaremos genéricos de Java para desarrollar factorías de cuerpos y leyes de gravedad.
- ✓ Un **paso de simulación** consiste en aplicar las leyes de la gravedad para cambiar las propiedades de los cuerpos y solicitar a los cuerpos que se muevan.

- ✓ En esta práctica:
  - La **entrada** es a) un fichero con los cuerpos y sus propiedades en formato JSON; b) las leyes de la gravedad que se van a usar; c) el número de pasos que el simulador debe ejecutar.
  - La **salida** es una estructura JSON que describe el estado de los cuerpos al inicio y después de cada paso de simulación.
- ✓ En el directorio **resources/** puedes encontrar ficheros de entrada con los correspondientes ficheros de salida. Debes asegurarte que tu implementación **genera estas salidas**.
- ✓ Más adelante explicamos como **comparar** tu salida con la salida esperada. También describimos un **visor** para poder ver de forma gráfica la simulación.

2. Material de la Práctica

- ✓ Recomendamos leer el siguiente material:
  - https://en.wikipedia.org/wiki/Equations_of_motion
  - https://en.wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation
- ✓ Un **vector** $a^{->}$ es un punto $(a_1, \ldots, a_n)$ en un espacio Euclidiano n-dimensional, donde cada a_i es un número real (i.e., de tipo *double*); una línea que va desde el origen de coordenadas $(0, \ldots, 0)$ al punto $(a_1, \ldots, a_n)$.
- ✓ En el paquete **simulator.misc** hay una clase *Vector*, que implementa un vector y ofrece las operaciones necesarias para manipularlo. Estas operaciones están en la hoja 3 del enunciado.

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

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

  donde key_i es una secuencia de caracteres (que representa una clave) y $value_i$ puede ser un número, un *string*, otra estructura JSON, o un array $[o_1,...,o_k]$, donde o_i puede ser un número, un string, una estructura JSON, o un array de estructuras JSON.

- Por ejemplo:

```
{
    "type" : "basic",
    "data" : {
                "id" : "planet",
                "pos" : [0.0e00, 4.5e10],
                "vel" : [1.0e04, 0.0e00],
                "mass" : 1.5e30
              }
 }
```

- En el directorio **lib/** del material hemos incluido una **librería** que permite parsear 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**.

- ✓ Para comparar tu salida sobre los ejemplos suministrados, con la salida esperada, puedes usar la siguiente herramienta online: **http://www.jsondiff.com**. Además, el ejemplo en el paquete **extra.json** incluye otra forma de comparar estructuras JSON.
- ✓ Ten en cuenta que dos estructuras JSON se consideran semánticamente iguales si tienen el mismo conjunto de pares atributo-valor. No es necesario que sean sintácticamente iguales.

3. Implementación del Simulador Físico

- ✓ Para desarrollar el simulador físico debes desarrollar la estructura de paquetes y clases que se indica en el enunciado.
- ✓ Para llegar a esta estructura te damos tanto el diseño de alto nivel de la **arquitectura de la solución** como los diagramas UML del diseño detallado (en **resources/uml/**)

3.1. Cuerpos

- ✓ Debes implementar al menos los siguientes cuerpos:
  - Cuerpos básicos.
  - Cuerpos que pierden masa.
- ✓ Las clases de todos estos cuerpos deben colocarle dentro del paquete **simulator.model** (no es subpaquetes).
- ✓ Un **cuerpo básico** se implementa con la clase *Body* y representa una entidad física.
- ✓ Un objeto de tipo *Body* contiene un identificador *id* (String), una vector de velocidad $v^{->}$, un vector de aceleración $a^{->}$, un vector de posición $p^{->}$ y una masa *m* (double). Todos estos atributos deben ser *protected*.

✓ Además esta clase debe contener los siguientes métodos:

- `public String getId() // devuelve el identificador del cuerpo.`
- `public Vector getVelocity() // devuelve una copia del vector de velocidad.`
- `public Vector getAcceleration() // devuelve una copia del vector de aceleración.`
- `public Vector getPosition() // devuelve una copia del vector de posición.`
- `double getMass()// devuelve la masa del cuerpo.`
- `void setVelocity(Vector v) // hace una copia de v y se la asigna al vector de velocidad (devuelve new Vector(v)).`
- `void setAcceleration(Vector a) // hace una copia de a y se la asigna al vector de aceleración (devuelve new Vector(a)).`
- `void setPosition(Vector p) // hace una copia de p y se la asigna al vector de posición (devuelve new Vector(p)).`
- `void move(double t) // mueve el cuerpo durante t segundos utilizando los atributos del mismo. Concretamente cambia la posición a` $p^{->} + (v^{->} \cdot t + (\frac{1}{2}\, a^{->} \cdot t^2))$ `y la velocidad a` $v^{->} + (a^{->} \cdot t)$.
- `public String toString() // devuelve un string con la información del cuerpo en formato JSON:`

  `{ "id": id, "mass": m, "pos":` $p^{->}$`, "vel":` $v^{->}$`, "acc":` $a^{->}$`}`

- ✓ Un **cuerpo que pierde masa** se representa con la clase *MassLossingBody*. Esta clase 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 es *m *(1–lossFactor).*

- ✓ Para implementar este proceso debes hacer lo siguiente: usa un contador *c* (inicializado a 0,0 en la constructora) para acumular el tiempo (i.e., el parámetro *t* de *move*) y cuando *c* >= *lossFrequency* aplica la reducción y pon de nuevo *c* a 0,0.

3.2. Leyes de Gravedad

- ✓ Debes implementar al menos las siguientes leyes de gravedad:
  - Ley de Newton de la gravitación universal.
  - Cayendo hacia el centro.
  - Sin gravedad.
- ✓ Todas las clases e interfaces de las leyes de gravedad deben colocarse en el paquete **simulator.model** (no en un subpaquete).
- ✓ Para modelar las leyes de la gravedad utilizaremos una interfaz *GravityLaws*, que tiene únicamente el siguiente método:

```
public void apply(List<Body>bodies)
```

- ✓ Este método, en las clases que implementan esta interfaz, debe **aplicar las leyes de la gravedad** correspondientes para cambiar las propiedades (e.g. el vector de aceleración) de los distintos cuerpos de la **lista** que se pasa como parámetro.
- ✓ Implementaremos la **ley de gravitación universal de Newton** con la 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 $F_{i,j}^{->}$ es la fuerza aplicada por el cuerpo B_j sobre el cuerpo B_i. La fuerza total aplicada sobre B_i se define como la suma de todas las fuerzas aplicadas sobre B_i por otros cuerpos. Ahora, usando la segunda ley de Newton, i.e., $F^{->} = m \cdot a^{->}$, podemos concluir que la aplicación de $F_i^{->}$ sobre B_i cambia su aceleración a $F_i^{->} \cdot 1 / m_i$

- ✓ Como un caso especial, si m_i es igual a 0,0, ponemos los vectores de aceleración y velocidad de B_i a $o^{->} = (0, \ldots, 0)$ (sin necesidad de calcular $F_i^{->}$).
- ✓ Vamos a explicar ahora como calcular $F_{i,j}^{->}$. 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:

$$f_{i,j} = G * m_i * m_j / |p_j^{->} - p_i^{->}|^2$$

donde G es la constante gravitacional, que es aproximadamente 6,67 10−11 (6,67E−11 usando la sintaxis de Java). Observa que $|p_j^{->} - p_i^{->}|$ es la distancia entre los vectores $p_i^{->}$ y $p_j^{->}$ 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 $F_{i,j}^{->}$ como sigue: Sea $d_{i,j}^{->}$ la dirección de $p_j^{->} - p_i^{->}$ entonces $F_{i,j}^{->} = d_{i,j}^{->} \cdot f_{i,j}$.

- ✓ La ley de gravedad (**cayendo hacia el centro**) se implementa en la clase *FallingToCenterGravity*.
- ✓ Esta 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 $o^{->} = (0, \ldots, 0)$.
- ✓ Técnicamente, para un cuerpo B_i, asumiendo que $d_i^{->}$ es su dirección, su aceleración debería ponerse a: $-g \cdot d_i^{->}$
- ✓ La ley **sin gravedad** se implementa en la clase *NoGravity*. Simplemente no hace nada, i.e., su método *apply* está vacío. Esto significa que los cuerpos se mueven con una aceleración fija.

3.3. Factorías

- ✓ Una vez que hemos definido las diferentes clases de cuerpos y leyes de la gravedad, utilizaremos **factorías** para separar la lógica de creación de objetos de la lógica del 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.
- ✓ Todas las clases e interfaces de las factorías deben colocarse en el paquete **simulator.factories** (no en subpaquetes).

- ✓ Modelamos las factorías a través de una **interfaz genérica** *Factory<T>*, con los siguientes métodos:
  - `public T createInstance(JSONObject info) // recibe un objeto JSON que describe el componente a crear, y devuelve una instancia de la clase correspondiente – una instancia de un subtipo de T. En caso de que info sea incorrecto, entonces lanza una excepción del tipo IllegalArgumentException.`
  - `public List<JSONObject> getInfo() // devuelve una lista de objetos JSON, que son "plantillas" para estructuras 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 (ver estructuras en la página 7 del enunciado).

- Los elementos de la lista que devuelve *getInfo* son estructuras JSON como las que se muestran en el enunciado. En estas estructuras se muestran valores por defecto. En lugar de valores podemos usar *strings* que describan las clases correspondientes.
- 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"
```

- ✓ 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 estructura JSON con un valor concreto para la clave *type*.
- ✓ Un constructor se modela usando la **clase genérica y abstracta** *Builder<T>*, con dos atributos (*typeTag* y *desc*) y 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 excepción 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>).`

- `protected abstract T createTheInstance(JSONObject jsonObject) // se invoca en createInstance cuando typeTag coincide con el valor de la clave type del parámetro info. En esta invocación se usa el JSONObject de la clave data de info.`

✓ Usa esta clase para definir los siguientes constructores:

- *BasicBodyBuilder* que extiende a *Builder<Body>*, para crear objetos de la clase *Body* – pone la aceleración a (0, . . . , 0).
- *MassLosingBodyBuilder* que extiende a *Builder<Body>*, para crear objetos de la clase *MassLosingBody* – pone la aceleración a (0, . . . , 0).
- *NewtonUniversalGravitationBuilder* que extiende a *Builder<GravityLaws>*, para crear objetos de la clase *NewtonUniversalGravitation*
- *FallingToCenterGravityBuilder* que extiende a *Builder<GravityLaws>*, para crear objetos de la clase *FallingToCenterGravity*
- *NoGravityBuilder* que extiende a *Builder<GravityLaws>*, para crear objetos de la clase *NoGravity*

- 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);
```

3.4. La Clase Simulador

- ✓ El simulador lo implementamos en 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 la gravedad // un objeto del tipo GravityLaws, que representa las leyes de la gravedad 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.

- La clase *PhysicsSimulator* debe contener los siguientes métodos:
  - `public void advance() // aplica un paso de simulación, i.e., primero llama al método apply de las leyes de la gravedad, después llama a move(dt) para cada cuerpo, donde dt es el tiempo real por paso, y 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 String toString() // devuelve un string que representa un estado del simulador, utilizando el siguiente formato JSON:`

    `{ "time": T, "bodies": [json`$_1$`, json`$_2$`, . . .] }`

    donde T es el tiempo actual y json$_i$ es el string devuelto por el método toString del i-ésimo cuerpo en la lista de cuerpos.

3.5. 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.
- Ambos parámetros son necesarios como atributos de la clase *Controller*.

✓ La clase *Controller* ofrece los siguientes métodos:

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

  `{ "bodies": [bb1,...,bbn] }`

  donde bb_i es una estructura JSON que define un cuerpo de acuerdo a la sintaxis dada en el enunciado. 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 jsonInput, crea el correspondiente cuerpo b usando la factoría de cuerpos (con el método createInstance), y lo añade al simulador llamando al método addBody.

- `public void run(int n, OutputStream out) // ejecuta el simulador n pasos y muestra los diferentes estados en out, utilizando el siguiente formato JSON:`

  `{ "states": [s0,s1,...,sn] }`

  donde s_0 es el estado del simulador antes de ejecutar ningún paso, y cada s_i con i >= 1, es el estado del simulador inmediatamente después de ejecutar el i-ésimo paso de simulación. Observa que el estado s_i se obtiene llamando al método toString() del simulador.

3.6. 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 parsea 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 parsear todos los posibles argumentos.

- ✓ Ejecutar *Main* con el argumento *-h* (or *--help*) debe mostrar por consola lo siguiente:

```
usage: simulator.launcher.Main [-dt <arg>] [-gl <arg>] [-h] [-i
<arg>] [-m
<arg>] [-o <arg>] [-s <arg>]
-dt,--delta-time <arg> A double representing actual time, in seconds,
per simulation step. Default value: 2500.0.
-gl,--gravity-laws <arg> Gravity laws to be used in the simulator.
Possible values: 'nlug' (Newton's law of universal gravitation),
'ftcg' (Falling to center gravity), 'ng' (No gravity). 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.
```

- Este texto de ayuda describe como ejecutar el simulador. Por ejemplo:

```
-i resources/examples/ex4.4body.txt
-o resources/examples/ex4.4body.out -s 100 -gl nlug
```

  ejecutaría el simulador 100 pasos, usando las leyes de la gravedad *nlug* (ley de Newton de la gravitación universal). La entrada la lee del fichero *resources/examples/ex4.4body.txt* y la salida la escribe en *resources/examples/ex4.4body.out*.

- En la cabecera de la clase *Main* puedes encontrar mas ejemplos de uso de la línea de comandos.

- ✓ Modifica la clase Main para que tenga la siguiente funcionalidad:
  - Añade la opción -o (o --output) en la línea de comandos para que el usuario pueda escribir el nombre del fichero en el cual escribir la salida. En caso de que no se especifique el fichero de salida, se utilizará la salida por consola *System.out*.
  - Añade la opción -s (o --steps) en la línea de comandos para que el usuario pueda especificar el númeo de pasos de simulación. Sino se especifica nada, el valor por defecto será de 150.
  - Completa la implementación del método *init()* para crear e inicializar las factorías (campos *_bodyFactory* and *_gravityLawsFactory*) – ver el final de la Sección 5.3 en el enunciado.

- Completa la implementación del método *startBatchMode()* de forma que cree una instancia del simulador y del controlador; establezca las leyes de gravedad del simulador de acuerdo con lo especificado a través de la opcion -gl; cree los ficheros correspondientes de entrada y salida teniendo en cuenta las opciones -i y -o, añada los cuerpos al simulador (invocando al método *loadBodies* del controlador); e inicie la simulación llamando al método *run* del controlador.

4. 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.
- ✓ 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.