Práctica 1: Simulador de Ecosistema

Objetivo: Diseño orientado a objetos, y uso de genéricos y colecciones.

Fecha de entrega: 04 de Marzo 2024, 08:30h

Control de Copias

Durante el curso se realizará control de copias de todas las prácticas, comparando las entregas de todos los grupos de TP2. 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 de la Facultad, que citará al alumno infractor y si considera que es necesario sancionar al alumno propondrá una de las tres medidas siguientes:

1. Calificación de cero en la convocatoria de TP2 a la que corresponde la práctica o examen.
2. Calificación de cero en todas las convocatorias de TP2 del curso actual.
3. Apertura de expediente académico ante la *Inspección de Servicios de la Universidad*.

Instrucciones Generales

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

1. **Lee el enunciado completo de la práctica antes de empezar**.
2. Crea un proyecto Java vacío en Eclipse (usar como mínimo JDK17).
3. Descarga `lib.zip` y descomprimelo en la raíz del proyecto (al mismo nivel de `src`), y añade las librerías al proyecto (Elegir `Project -> Properties -> BuildPath -> Libraries`, marcar `ClassPath`, pulsar el botón **"`Add Jars`"** y seleccionar la librería que quieres añadir).
4. Descarga `resources.zip` y descomprimelo en la raíz del proyecto (al mismo nivel de `src`).
5. Descarge `src.zip` y reemplaza el directorio src de proyecto por este `src` (o copia el contenido).
6. **Es muy importante que cada miembro del grupo haga los puntos 2-5, porque en el examen tendrás que hacerlo usando el `src.zip` de tu práctica (no vas a usar un proyecto ya montado).**
7. Es necesario usar exactamente la misma estructura de paquetes y los mismos nombres de clases que aparecen en el enunciado.
8. La generación de números aleatorios se debe hacer usando `Utils._rand` (ver el apartado "Generación de Números Aleatorios"). Está prohibido crear otra instancia de la clase **`Random`** o usar **`Math.random()`**.
9. Debes formatear todo el código usando la funcionalidad de Eclipse (`Source->Format`).
10. Todas las constructoras tienen que comprobar la validez de los parámetros y lanzar excepciones correspondientes con mensaje informativo (se puede usar `IllegalArgumentException`).
11. En la corrección tendremos en cuenta el estilo de código: uso de nombres adecuados para métodos/variables/clases, el formateo de código indicado en el punto anterior, etc.
12. Cuando entregues la práctica sube un fichero con el nombre **`src.zip`** que incluya solo la carpeta **`src`**. No está permitido llamarlo con otro nombre ni usar **`7zip`**, **`rar`**, etc.

Descripción General del Simulador

El simulador tiene como objetivo simular un ecosistema compuesto por animales. Los animales en la simulación pueden ser carnívoros o herbívoros. En esta práctica, tenemos dos tipos: lobos (carnívoros) y ovejas (herbívoros). Cada animal es un individuo y determina su propio comportamiento basándose en su entorno y su propio estado. Los estados posibles son: estado normal; buscando a otro animal para emparejarse; huyendo de otro animal que le resulta peligroso; está siguiendo a otro animal para cazarlo, etc. Cuando los animales se emparejan, pueden nacer otros animales que heredan propiedades de sus genitores. Los animales mueren cuando alcanzan un límite de edad o quedan sin energía.

Los animales actualizan sus estados mediante un método `update(double)`, donde el parámetro representa un intervalo de tiempo que corresponde a un paso en la simulación (hay que tenerlo en cuenta al actualizar la edad, la posición, etc). Usamos una clase `Vector2D` para representar un punto en un plano bidimensional, como la posición de un animal (ver el apartado "La Clase Vector2D").

La simulación incluye regiones donde se encuentran los animales, y el mundo está compuesto por una matriz de regiones. Al moverse, los animales pueden desplazarse de una región a otra. El objetivo de las regiones es proporcionar comida a los animales cuando lo necesitan. Vamos a tener varios tipos de regiones

que proporcionan comida según criterios distintos. La gestión de las regiones (añadir animales, quitar animales, pedir comida, etc.) se hace a través de un gestor de regiones.

La clase principal de la simulación incluye un gestor de regiones y una lista de animales, y permite añadir animales a la simulación, avanzar la simulación un paso, consultar el estado, etc. Un paso de la simulación incluye: quitar todos los animales muertos de la simulación; actualizar el estado de todos los animales vivos; y hacer nacer a los bebés que llevan los animales.

El bucle principal del simulador avanza la simulación varios pasos durante T segundos (por ejemplo, en cada paso avanza la simulación `0.003` segundos – el parámetro del método `update` que mencionamos arriba). El bucle muestra el estado actual de los animales usando el visor que proporcionamos con la práctica (ver el apartado "El Visor de Objetos"), y además, escribe el estado inicial y final de la simulación en un archivo usando el formato `JSON`. La configuración inicial del mundo se carga desde un archivo en formato `JSON` (ver el apartado "Análisis y Creación de Datos JSON en Java").

La Lógica del Simulador (el modelo)

Todas las clases/interfaces de este apartado tienen que ir en el paquete `simulator.model`.

El modelo incluye clases para representar animales, regiones, gestor de regiones y la clase principal del simulador. La funcionalidad de cada clase está dividida en varios interfaces para restringir lo que pueden hacer las distintas partes del simulador sobre las instancias de esas clases.

Todos los números que aparecen con fondo verde a continuación son parámetros que hemos elegido en nuestra implementación de la práctica para conseguir un comportamiento razonable. Es recomendable definirlos como constantes (como `final static`) para poder probar con varios valores. También puedes cambiar las fórmulas que usamos y/o los comportamientos de los animales en cada estado, siempre que obtengas un comportamiento razonable.

Clases/Interfaces Comunes

La Interfaz JSONable

Varios objetos de la simulación van a proporcionar su estado en formato `JSON`. Definimos la siguiente interfaz para implementar esta funcionalidad:

```
public interface JSONable {
  default public JSONObject as_JSON() {
        return new JSONObject();
   }
}
```

La Interfaz Entity

Varios objetos de la simulación necesitan actualizar sus estados en cada iteración (en principio los animales y las regiones). Definimos la siguiente interfaz para implementar esta funcionalidad:

```
public interface Entity {
  public void update(double dt);
}
```

Los Animales

No está permitido añadir un atributo para la región en las clases de animales. Toda la gestión de regiones se tiene que hacer a través del gestor de regiones.

Alimentación (Enumerado)

Los animales pueden ser herbívoros (`HERBIVORE`) o carnívoros (`CARNIVORE`). Definimos un tipo enumerado `Diet` con estos dos valores.

Estado de un Animal (Enumerado)

Los animales pueden estar en uno de los siguientes estados: normal (`NORMAL`), emparejamiento (`MATE`), hambre (`HUNGER`), peligro (`DANGER`), o muerto (`DEAD`). Definimos un tipo enumerado `State` con estos `5` valores.

La Interfaz AnimalInfo

Es una interfaz que define qué información se puede ver sobre un animal, incluye métodos que consultan el estado del animal pero nunca lo modifican:

```
public interface Animalnfo extends JSONable { // Note that it extends JSONable
  public State get_state();
  public Vector2D get_position();
  public String get_genetic_code();
  public Diet get_diet();
  public double get_speed();
  public double get_sight_range();
  public double get_energy();
  public double get_age();
  public Vector2D get_destination();
  public boolean is_pregnant();
}
```

Cuando queramos pasar una instancia de la clase `Animal` a una parte del programa que no puede alterar el estado, la vamos a pasar como `Animalnfo`. Se pueden añadir más métodos si es necesario, siempre que no alteren el estado del animal.

Estrategias de Selección de Animales

En algunas circunstancias los animales tendrán que elegir un animal de una lista de animales (que están dentro de su campo visual). Por ejemplo, para emparejarse, para buscar un objetivo de caza, etc. Para que los animales puedan tener comportamientos de selección distintos (aunque los animales sean del mismo tipo), vamos a usar estrategias de selección. Usaremos la siguiente interfaz para representar una estrategia de selección:

```
public interface SelectionStrategy {
  Animal select(Animal a, List<Animal> as);
}
```

El método `select` selecciona para el animal “`a`” un animal de la lista “`as`” según los criterios de la estrategia concreta (se supone que el objeto que corresponde al animal “`a`” invocará a `select`). Si la lista está vacía, siempre devuelve **`null`**.

Implementa las siguientes estrategias:

- `SelectFirst`: devuelvo el primer animal de la lista “`as`”.
- `SelectClosest`: devuelve el animal más cercano al animal “`a`” de la lista “`as`”.
- `SelectYoungest`: devuelve el animal más joven de la lista “`as`”.

Puedes implementar más estrategias si quieres.

La Clase Animal

Representamos un animal con la *clase abstracta* `Animal` que implementa las interfaces `Entity` y `AnimalInfo`. Más abajo vamos a definir 2 tipos de animales que heredan de esta clase.

Atributos necesarios:

Cada animal tiene que llevar como mínimo los siguiente atributos (pueden ser `protected` para poder acceder directamente desde las subclases o `private` y definir getters correspondientes):

- `_genetic_code` (`String`): es una cadena de caracteres **no vacía** que representa el código genético, cada subclase va asignar un valor distinto a este campo. En principio se usa para saber si 2 animales pueden emparejarse o no (p.ej., si tienen el mismo código genético).
- `_diet` (`Diet`): indica si el animal es herbívoro o carnívoro.
- `_state` (`State`): el estado actual del animal.
- `_pos` (`Vector2D`): la posición del animal.
- `_dest` (`Vector2D`): el destino del animal (el animal siempre tiene un destino, y cuando lo alcanza elige otro, o lo cambia según si está siguiendo a otro animal o siendo perseguido por otro animal).
- `_energy` (`double`): la energía del animal. Cuando llega a `0.0` el animal muere.
- `_speed` (`double`): la velocidad del animal.
- `_age` (`double`): la edad del animal. Cuando llega a un máximo (dependiendo del tipo de animal) el animal muere.
- `_desire` (`double`): el deseo del animal, que cambia durante la simulación. Lo vamos a usar para decidir si un animal entra en (o sale de) un estado de emparejamiento.
- `_sight_range` (`double`): el radio del campo visual del animal (para decidir qué animales puede ver).
- `_mate_target` (`Animal`): una referencia a un animal con el que quiere emparejarse.
- `_baby` (`Animal`): una referencia que indica si el animal lleva un bebé que no ha nacido aún.
- `_region_mngr` (`AnimalMapView`): es el gestor de regiones para poder consultar información o hacer operaciones correspondientes (ver el apartado "El Gestor de Regiones"). Cuando creamos el objeto los inicializamos a `null`, hasta que el gestor de regiones inicialice el animal llamando a su método `init`.
- `_mate_strategy` (`SelectionStrategy`): es la estrategia de selección para buscar pareja.

Constructoras:

Es necesario tener 2 constructoras. La primera se usa para crear los objetos iniciales y la segunda para cuando nazca un animal.

La primera constructora es la siguiente (se pueden añadir más parámetros si es necesario)

```
protected Animal(String genetic_code, Diet diet, double sight_range,
                 double init_speed, SelectionStrategy mate_strategy, Vector2D pos)
```

Donde `genetic_code` tiene que ser una cadena de caracteres no vacía, `sight_range` y `init_speed` números positivos y `mate_strategy` no es `null`. Hay que lanzar la excepción correspondiente con un mensaje informativo si algún valor es incorrecto.

Los valores de `_genetic_code`, `_diet`, `_sight_range`, `_pos`, y `_mate_strategy` se inicializan a los valores recibidos. Inicializa `_speed` a `Utils.get_randomized_parameter(init_speed, 0.1)` — ver este método en la clase `Utils`. Nótese que el valor de `pos` puede ser `null` y en ese caso se inicializa a un valor aleatorio en el método `init` (no en la constructora, ver la descripción del método `init`).

Aparte de los valores recibidos, hay que inicializar los otros atributos de la siguiente manera: `_state` es `NORMAL`, `_energy` es `100.0`, `_desire` es `0.0`, y `_dest`, `_mate_target`, `_baby` y `_region_mngr` son `null`.

La segunda constructora se usa para cuando nazca un animal a partir de otros 2:

```
protected Animal(Animal p1, Animal p2)
```

Hay que inicializar los atributos de la siguiente manera: `_dest`, `_baby`, `_mate_target` y `_region_mngr` son `null`, `_state` es `NORMAL`, `_desire` es `0.0`, `_genetic_code` y `_diet` los hereda de `p1`, `_energy` es la media de las energías de `p1` y `p2`, `_pos` es una posición aleatoria cerca de `p1` usando por ejemplo:

```
p1.get_position().plus(Vector2D.get_random_vector(-1,1).scale(60.0*(Utils._rand
.nextGaussian()+1)))
```

`_sight_range` es una mutación de la media de los campos visuales de `p1` y `p2` usando por ejemplo:

```
Utils.get_randomized_parameter((p1.get_sight_range()+p2.get_sight_range())/2, 0.2)
```

`_speed` es una mutación de la media de la velocidades de `p1` y `p2` usando por ejemplo:

```
Utils.get_randomized_parameter((p1.get_speed()+p2.get_speed())/2, 0.2)
```

Métodos necesarios:

Además de los métodos de la interfaz que implementa, hay que implementar los siguientes métodos:

- `void init(AnimalMapView reg_mngr)`: el gestor de regiones invocará a este método al añadir el animal a la simulación:
  - Inicializar `_region_mngr` a `reg_mngr`.
  - Si `_pos` es `null` hay que elegir una posición aleatoria dentro del rango del mapa (**X** entre `0` y `_region_mngr.get_width()-1` y **Y** entre `0` y `_region_mngr.get_height()-1`). Si `_pos` no es `null` hay que ajustarlo para que esté dentro del mapa si es necesario (ver el apartado "Ajustar posiciones").
  - Elegir una posición aleatoria para `_dest` (dentro del rango del mapa).
- `Animal deliver_baby()`: devolver `_baby` y ponerlo a `null`. El simulador invocará a este método para que nazcan los animales.
- `protected void move(double speed)`: las subclases usan este método para actualizar la posición del animal (para que se mueva hacia `_dest` con velocidad `speed`). Esto se puede hace usando

  ```
  _pos = _pos.plus(_dest.minus(_pos).direction().scale(speed))
  ```
- `public JSONObject as_JSON()`: devuelve una estructura `JSON` como la siguiente:

  ```
  {
    "pos": [28.90696391797469,22.009772194487613],
    "gcode": "Sheep",
    "diet": "HERBIVORE",
    "state": "NORMAL"
  }
  ```

La Clase Sheep

Es una clase que representa una oveja. Es un animal herbívoro con código genético `"Sheep"`. Es un animal que no caza a otros animales, sólo come lo que proporciona la región en la que está, y puede emparejarse con otros animales con el mismo código genético.

Atributos necesarios:

Es necesario mantener una referencia (`_danger_source`) a otro animal que se considera como un peligro en un momento dado, y otra referencia (`_danger_strategy`) a una estrategia de selección para elegir un peligro de la lista de animales en el campo visual.

Constructoras:

Su *primera constructora*

```
public Sheep(SelectionStrategy mate_strategy, SelectionStrategy danger_strategy,
Vector2D pos)
```

recibe las estrategias y la posición y las almacena en los atributos correspondientes (llamando a la constructora de la superclase). El campo de vista inicial es `40.0` y la velocidad inicial es `35.0`.

Su *segunda constructora* se usa para cuando nazca un animal de tipo `Sheep`:

```
protected Sheep(Sheep p1, Animal p2)
```

Aparte de llamar a la constructora correspondiente de la superclase, tiene que heredar `_danger_strategy` de `p1` y poner su `_danger_source` a `null`.

Métodos Necesarios:

Es necesario implementar el método `update`. Este método tiene que hacer lo siguiente:

1. Si el estado es `DEAD` no hacer nada (volver inmediatamente).
2. Actualizar el objeto según el estado del animal (ver la descripción abajo).
3. Si la posición está fuera del mapa, ajustarla y cambiar su estado a `NORMAL`.
4. Si `_energy` es `0.0` o `_age` es mayor de `8.0`, cambia su estado a `DEAD`.
5. Si su estado no es `DEAD`, pide comida al gestor de regiones usando `get_food(this, dt)` y añadela a su `_energy` (manteniéndolo siempre entre `0.0` y `100.0`)

Para buscar un animal que se considere peligroso, hay que pedir al gestor de regiones la lista de animales **carnívoros** en el campo visual, usando el método `get_animals_in_range`, y después elegir uno usando la estrategia de selección correspondiente.

Para buscar un animal para emparejarse, hay que pedir al gestor de regiones la lista de animales **con el mismo código genético** en el campo visual, usando el método `get_animals_in_range`, y después elegir uno usando la estrategia de selección correspondiente.

Además de lo que explicamos a continuación recuerda que: cuando cambia a estado `NORMAL` tiene que poner `_danger_source` y `_mate_target` a `null`; cuando cambia a estado `MATE` tiene que poner `_danger_source` a `null`; y cuando cambia a `DANGER` tiene que poner `_mate_target` a `null`.

Cómo Actualizar el Objeto Según el Estado:

Si el estado actual es `NORMAL`:

1. Avanzar el animal según los siguiente pasos:
   1.1. Si la distancia del animal al destino (`_dest`) es menor que `8.0`, elegir otro destino de manera aleatoria (dentro de las dimensiones de mapa).
   1.2. Avanza (llamando a `move`) con velocidad `_speed*dt*Math.exp((_energy-100.0)*0.007)`.
   1.3. Añadir `dt` a la edad.
   1.4. Quitar `20.0*dt` a la energía (manteniéndola siempre entre `0.0` y `100.0`).
   1.5. Añadir `40.0*dt` al deseo (manteniéndolo siempre entre `0.0` y `100.0`).
2. Cambio de estado
   2.1. Si `_danger_source` es `null`, buscar un nuevo animal que se considere peligroso.
   2.2. Si `_danger_source` no es `null`, cambiar el estado a `DANGER`, y si es `null` y el deseo mayor de

65.0 cambiar el estado a MATE. En otro caso no hacer nada, seguimos con el estado actual.

Si el estado actual es DANGER:

1. Si _danger_source no es null y su estado es DEAD, poner _danger_source a null porque ya ha muerto por alguna razón y ya no es peligroso.
2. Si _danger_source es null, avanzar normalmente como el punto 1 del caso NORMAL arriba, y si _danger_source no es null:
   2.1. Queremos cambiar el destino para avanzar en la dirección contraria al peligro. Esto se puede hacer con _pos.plus(_pos.minus(_danger_source.get_position()).direction()) como destino.
   2.2. Avanza (llamando a move) con velocidad 2.0*_speed*dt*Math.*exp*((_energy-100.0)*0.007).
   2.3. Añadir dt a la edad.
   2.4. Quitar 20.0*1.2*dt a la energía (manteniéndola siempre entre 0.0 y 100.0).
   2.5. Añadir 40.0*dt al deseo (manteniéndolo siempre entre 0.0 y 100.0).
3. Cambio de estado
   3.1. Si _danger_source es null o _danger_source no está en el campo visual del animal
      3.1.1. buscar un nuevo animal que se considere como peligro.
      3.1.2. Si _danger_source es null:
         3.1.2.1. Si el deseo es menor que 65.0, cambia el estado a NORMAL, en otro caso cámbialo a MATE.

Si el estado actual es MATE:

1. Si _mate_target no es null y su estado es DEAD o está fuera del campo visual, poner _mate_target a null ya que no lo va a seguir para emparejarse.
2. Si _mate_target es null, buscar un animal para emparejarse y si no se encuentra uno avanza normalmente como el punto 1 del caso NORMAL arriba, en otro caso (donde _mate_target ya no es null):
   2.1. Queremos cambiar el destino para perseguir a _mate_target. Esto se puede hacer cambiándolo a _mate_target.get_position().
   2.2. Avanza (llamando a move) con velocidad 2.0*_speed*dt*Math.*exp*((_energy-100.0)*0.007).
   2.3. Añadir dt a la edad.
   2.4. Quitar 20.0*1.2*dt a la energía (manteniéndola siempre entre 0.0 y 100.0).
   2.5. Añadir 40.0*dt al deseo (manteniéndolo siempre entre 0.0 y 100.0).
   2.6. Si la distancia del animal a _mate_target es menor que 8.0, entonces van a emparejarse según los siguientes pasos:
      2.6.1. Resetear el deseo del animal y del _mate_target a 0.0.
      2.6.2. Si el animal no lleva un bebé ya, con probabilidad de 0.9 va a llevar a un nuevo bebé usando new Sheep(this, _mate_target).
      2.6.3. Poner _mate_target a null.
3. Si _danger_source es null buscar un nuevo animal que se considere como peligroso.
4. Si _danger_source no es null cambia de estado a DANGER, y si es null y el deseo es menor que 65.0 cambia de estado a NORMAL.

Si el estado actual es HUNGER:

Un objeto de tipo Sheep nunca puede estar en estado HUNGER.

La Clase Wolf

Es una clase que representa un lobo. Es un animal carnívoro con código genético "Wolf". Es un animal que caza a otros animales herbívoros y también puede comer lo que proporciona la región en la que está, y puede emparejarse con otros animales con el mismo código genético.

Atributos necesarios:

Es necesario mantener una referencia (`_hunt_target`) a otro animal al que quiere cazar en un momento dado, y otra referencia (`_hunting_strategy`) a una estrategia de selección para elegir un animal para cazar.

Constructoras:

Su *primera constructora*

```
public Wolf(SelectionStrategy mate_strategy, SelectionStrategy hunting_strategy,
Vector2D pos)
```

recibe las estrategias y la posición y simplemente le almacena en los atributos correspondientes (llamando a la constructora de la superclase). El campo de vista inicial es `50.0` y la velocidad inicial es `60.0`.

Su *segunda constructora* se usa para cuando nazca un animal de tipo `Wolf`:

```
protected Sheep(Wolf p1, Animal p2)
```

Aparte de llamar a la constructora correspondiente de la superclase, tiene que heredar `_hunting_strategy` de `p1` y poner su `_hunt_target` a `null`.

Métodos Necesarios:

Es necesario implementar el método `update`. Este método tiene que hacer lo siguiente:

1. Si el estado es `DEAD` no hacer nada (volver inmediatamente).
2. Actualizar el objeto según el estado del animal (ver la descripción abajo)
3. Si la posición está fuera del mapa, la ajusta y cambia su estado a `NORMAL`.
4. Si `_energy` es `0.0` o `_age` es mayor de `14.0`, cambia su estado a `DEAD`.
5. Si su estado no es `DEAD`, pide comida al gestor de regiones usando `get_food(this, dt)` y la añade a su `_energy` (manteniéndolo siempre entre `0.0` y `100.0`)

Para buscar un animal para cazar, hay que pedir al gestor de regiones la lista de animales **herbívoros** en el campo visual, usando el método `get_animals_in_range`, y después elegir uno usando la estrategia de selección correspondiente.

Para buscar un animal para emparejarse, hay que pedir al gestor de regiones la lista de animales **con el mismo código genético** en el campo visual, usando el método `get_animals_in_range`, y después elegir uno usando la estrategia de selección correspondiente.

Además de lo que explicamos a continuación recuerda que: cuando cambia a estado `NORMAL` tiene que poner `_hunt_target` y `_mate_target` a `null`; cuando cambia a estado `MATE` tiene que poner `_hunt_target` a `null`; cuando cambia a estado `HUNGER` tiene que poner `_mate_target` a `null`.

Cómo Actualizar el Objeto Según el Estado:

Si el estado actual es `NORMAL`:

1. Avanzar el animal según los siguiente pasos:
   1.1. Si la distancia del animal al destino (`_dest`) es menor que `8.0`, elegir otro destino de manera aleatoria (dentro de las dimensiones de mapa).
   1.2. Avanza (llamando a `move`) con velocidad `_speed*dt*Math.exp((_energy-100.0)*0.007)`.
   1.3. Añadir `dt` a la edad.
   1.4. Quitar `18.0*dt` a la energía (manteniéndola siempre entre `0.0` y `100.0`).
   1.5. Añadir `30.0*dt` al deseo (manteniéndolo siempre entre `0.0` y `100.0`).
2. Cambio de estado
   2.1. Si su energía es menor que `50.0` cambia de estado a `HUNGER`, y si no lo es y su deseo es mayor que `65.0` cambia de estado a `MATE`. En otro caso no hace nada.

Si el estado actual es `HUNGER`:

1. Si `_hunt_target` es `null`, o no es `null` pero su estado es `DEAD` o está fuera del campo visual, buscar otro animal para cazarlo.
2. Si `_hunt_target` es `null`, avanzar normalmente como el punto 1 del caso `NORMAL` arriba, y si `_hunt_target` no es `null`:
   2.1. Queremos cambiar el destino para avanzar hacia el animal que quiere cazar. Esto se puede hacer con `_hunt_target.get_position()` como destino.
   2.2. Avanza (llamando a `move`) con velocidad `3.0*_speed*dt*Math.exp((_energy-100.0)*0.007)`.
   2.3. Añadir `dt` a la edad.
   2.4. Quitar `18.0*1.2*dt` a la energía (manteniéndola siempre entre `0.0` y `100.0`).
   2.5. Añadir `30.0*dt` al deseo (manteniéndola siempre entre `0.0` y `100.0`).
   2.6. Si la distancia del animal a `_hunt_target` es menor que `8.0`, entonces va a cazar según los siguientes pasos:
      2.6.1. Poner el estado `_hunt_target` a `DEAD`.
      2.6.2. Poner `_hunt_target` a `null`.
      2.6.3. Sumar `50.0` a la energía (manteniéndola siempre entre `0.0` y `100.0`).
3. Cambiar de estado
   3.1. Si su energía es mayor que `50.0`
      3.1.1. Si el deseo es menor que `65.0` cambia el estado a `NORMAL`.
      3.1.2. En otro caso cámbialo a `MATE`.
   3.2. Si su energía es menor, no hacer nada.

Si el estado actual es `MATE`:

1. Si `_mate_target` no es `null` y su estado es `DEAD` o está fuera del campo visual, poner `_mate_target` a `null` ya que no lo va a seguir para emparejarse.
2. Si `_mate_target` es `null`, buscar un animal para emparejarse y si no se encuentra uno avanza normalmente como el punto 1 del caso `NORMAL` arriba, en otro caso (`_mate_target` ya no era `null`):
   2.1. Queremos cambiar el destino para perseguir a `_mate_target`, esto se puede hacer con `_mate_target.get_position()` como destino.
   2.2. Avanza (llamando a `move`) con velocidad `3.0*_speed*dt*Math.exp((_energy-100.0)*0.007)`.
   2.3. Añadir `dt` a la edad.
   2.4. Quitar `18.0*1.2*dt` a la energía (manteniéndola siempre entre `0.0` y `100.0`).
   2.5. Añadir `30.0*dt` al deseo (manteniéndolaosiempre entre `0.0` y `100.0`).
   2.6. Si la distancia del animal a `_mate_target` es menor que `8.0`, entonces van a emparejarse según los siguientes pasos:
      2.6.1. Resetear el deseo del animal y del `_mate_target` a `0.0`.
      2.6.2. Si el animal no lleva un bebe ya, con probabilidad de `0.9` va a llevar a un nuevo bebe usando new Wolf(this, _mate_target).
      2.6.3. Quitar `10.0 de` la energía (manteniéndola siempre entre `0.0` y `100.0`).
      2.6.4. Poner _mate_target a `null`.
3. Si su energía es menor que `50.0` cambia de estado a `HUNGER`, y si no lo es y el deseo es menor que `65.0` cambia de estado a `NORMAL`.

Si el estado actual es `DANGER`:

Un objeto de tipo `Wolf` nunca puede estar en estado `DANGER`.

Regiones

Las regiones son zonas donde se encuentran los animales. Las regiones proporcionan comida según distintos criterios.

La Interfaz FoodSupplier

Usamos la siguiente interfaz para pedir comida para el animal `a` durante `dt` segundos:

```
public interface FoodSupplier {
  double get_food(Animal a, double dt);
}
```

La Interfaz RegionInfo

Es una interfaz que define qué información se puede ver sobre una región e incluye métodos que consultan el estado de la región pero nunca lo modifican. De momento la interfaz solo extiende `JSONable`, más adelante añadiremos más métodos.

```
public interface RegionInfo extends JSONable {
  // for now it is empty, later we will make it implements the interface
  // Iterable<AnimalInfo>
}
```

La Clase Region

Representamos una región con la clase abstracta `Region` que implementa las interfaces `Entity`, `FoodSupplier` y `RegionInfo`.

Atributos necesarios:

Un atributo con la lista de animales que se encuentran en la región. Mantenerlo `protected` para poder acceder directamente desde las subclases.

Constructoras:

Tiene solo una constructora por defecto que inicializa la lista de animales.

Métodos necesarios:

Además de los métodos de la interfaz que implementa, hay que implementar los siguientes métodos:

- `final void add_animal(Animal a)`: añade el animal a la lista de animales.
- `final void remove_animal(Animal a)`: quita el animal de la lista de animales.
- `final List<Animal> getAnimals():` devuelve una versión **inmodificable** de la lista de animales.
- `public JSONObject as_JSON()`: devuelve una estructura `JSON` como la siguiente donde a_i es lo que devuelve `as_JSON()` del animal correspondiente:

  {
  "animals":[a_1,a_2,...],
  }

La Clase DefaultRegion

La clase `DefaultRegion` representa una región que da comida sólo a animales herbívoros. No tiene constructoras (solo la constructora por defecto, que está definida automáticamente),

Su método `get_food(a,dt)` devuelve `0.0` si el animal que pide comida es carnívoro, y lo siguiente si es

herbívoro donde `n` es el número de animales herbívoros en la región:

```
60.0*Math.exp(-Math.max(0,n-5.0)*2.0)*dt
```

Su método `update` no hace nada.

La Clase DynamicSupplyRegion

La clase `DynamicSupplyRegion` representa una región que da comida sólo a animales herbívoros, pero la cantidad de comida puede decrecer/crecer. Su constructora recibe la cantidad inicial de la comida (número positivo de tipo `double`) y un factor de crecimiento (número no negativo de tipo `double`).

Su método `get_food(a,dt)` devuelve `0.0` si el animal que pide comida es carnívoro, y lo siguiente si es herbívoro donde `n` es el número de animales herbívoros en la región y `_food` es la cantidad actual de comida:

```
Math.min(_food,60.0*Math.exp(-Math.max(0,n-5.0)*2.0)*dt)
```

Además quita el valor devuelto a la cantidad de comida `_food` que tiene la región actualmente. Su método `update` incrementa, con probabilidad `0.5`, la cantidad de comida por `dt*_factor` donde `_factor` es el factor de crecimiento.

El Gestor de Regiones

El gestor de regiones es una clase que facilita el trabajo con regiones. Representa un mapa con altura y anchura fijas y mantiene una matriz de regiones (con números de columnas y filas fijas). Se puede tanto registrar un animal (y lo coloca en su región), como eliminar un animal (y lo quita de su región). Un animal puede pedir comida al gestor y el gestor delega la petición a la región correspondiente, etc.

La Interfaz MapInfo

Es una interfaz que representa el mapa y nos permite consultar información pero nunca modificar el mapa.

```
public interface MapInfo extends JSONable {
  public int get_cols();
  public int get_rows();
  public int get_width();
  public int get_height();
  public int get_region_width();
  public int get_region_height();
}
```

La Interfaz AnimalMapView

Es una interfaz que representa lo que un animal puede ver del gestor de regiones. En principio puede ver el mapa, pedir comida, y además puede pedir la lista de animales en su campo visual que además satisfacen alguna condición.

```
public interface AnimalMapView extends MapInfo, FoodSupplier {
  public List<Animal> get_animals_in_range(Animal e, Predicate<Animal> filter);
}
```

La Clase RegionManager

Representamos el gestor de regiones con la clase `RegionManager` que implementa la interfaz `AnimalMapView`.

Atributos necesarios:

Tiene que mantener información básica sobre el mapa (anchura/altura de mapa, columnas, filas, anchura/altura de una región). Además, tiene que mantener una matriz de regiones con número de filas y columnas correspondientes (`_regions`), y un mapa (`_animal_region`) de tipo `Map<Animal, Region>` que asigna a cada animal su región actual.

Constructoras:

Hay solo una constructora que recibe el número de columnas, el número de filas, la anchura, y la altura.

```
public RegionManager(int cols, int rows, int width, int height)
```

Tiene que almacenar los parámetros en los atributos correspondientes, y calcular la anchura y altura de una celda (dividir anchura/altura total por el número de columnas/filas) y almacenarlo en los atributos correspondientes. Además, debe inicializar la matriz `_regions` con regiones de tipo `DefaultRegion` (usando la constructora por defecto) e inicializar `_animal_region` con una estructura de datos adecuada.

Métodos necesarios:

Además de los métodos de la interfaz que implementa, hay que implementar los siguientes métodos:

- `void set_region(int row, int col, Región r)`: modifica la región localizada en la columna `row` y file `col` a `r`. Además añade todos los animales que estaban en la región anterior a `r` y actualiza sus entradas en `_animal_region`.
- `void register_animal(Animal a)`: encuentra la región a la que tiene que pertenecer el animal (a partir de su posición) y lo añade a esa región y actualiza `_animal_region`.
- `void unregister_animal(Animal a)`: quita el animal de la región a la que pertenece y actualiza `_animal_region`.
- `void update_animal_region(Animal a)`: encuentra la región a la que tiene que pertenecer el animal (a partir de su posición actual), y si es distinta de su región actual lo añade a la nueva región, lo quita de la anterior, y actualiza `_animal_region`.
- `public double get_food(Animal a, double dt)`: llama a `get_food` de la región a la que pertenece el animal y devuelve el valor correspondiente.
- `void update_all_regions(double dt)`: llama a `update` de todas la regiones en la matriz de regiones.
- `public List<Animal> get_animals_in_range(Animal a, Predicate<Animal> filter)`: devuelve un lista de todos los animales que están en el campo visual del animal `a` y cumplen la condición `filter`. Debe consultar sólo las regiones en el campo visual.
- `public JSONObject as_JSON()`: devuelve una estructura `JSON` de la siguiente forma

  ```
  {
   "regiones":[o_1,o_2,...],
  }
  ```

  donde o_i es una estructura `JSON` que corresponde a una región y tiene la siguiente forma

  ```
  {
    "row": i,
    "col": j,
  ```

```
  "data": r
}
```

donde `r` es lo que devuelve `as_JSON()` de la región en la fila `i` y columna `j`.

La Clase Simulator

Esta es la clase principal del modelo, a través de la cual podemos añadir animales, modificar regiones, y avanzar la simulación. La representamos con la clase `Simulator` que implementa la interfaz `JSONable`.

Atributos necesarios:

Esta clase tiene que recibir como atributos una factoría de animales y otra de regiones (ver el apartado "Las Factorías"). Además tiene que llevar un gestor de regiones, una lista con todos los animales que están participando en la simulación, y el tiempo actual (`double`).

Constructoras:

Tiene solo una constructora, que recibe las dimensiones y las factorías:

```
public Simulator(int cols, int rows, int width, int height,
                 Factory<Animal> animals_factory, Factory<Region> regions_factory)
```

La constructora almacena los parámetros en atributos correspondientes, crea el gestor de regiones y la lista de animales, e inicializa el tiempo a `0.0`.

Métodos necesarios:

Hay que implementar los siguientes métodos:

- `private set_region(int row, int col, JSONObject r)`: añade la región `r` al gestor de regiones en la posición `(row,cols)`.
- `void set_region(int row, int col, JSONObject r_json)`: crea una región `R` a partir de `r_json` y llama a `set_region(row,col,R)`.
- `private void add_animal(Animal a)`: añade el animal a la lista de animales y lo registra en el gestor de regiones.
- `public void add_animal(JSONObject a_json)`: crea un animal `A` a partir de `a_json` y llama a `add_animal(A)`.
- `public MapInfo get_map_info()`: devuelve el gestor de regiones.
- `public List<? extends Animalnfo> get_animals()`: devuelve una versión **inmodificable** de la lista de animales.
- `public double get_time()`: devuelve el tiempo actual.
- `public void advance(double dt)`: avanza la simulación un paso (hay que tener cuidado de no modificar la lista de animales mientras la estamos recorrando):
  - Incrementar el tiempo por `dt`.
  - Quitar todos los animales con estado `DEAD` de la lista de animales y eliminarlos del gestor de regiones.
  - Para cada animal: llama a su `update(dt)` y pide al gestor de regiones que actualice su región.
  - Pedir al gestor de regiones actualizar todas las regiones.
  - Para cada animal: si `is_pregnant()` devuelve `true`, obtenemos el bebé usando su método

`deliver_baby()` y lo añadimos a la simulación usando `add_animal`.

- `public JSONObject as_JSON()`: devuelve una estructura `JSON` como de la siguiente donde `t` es el tiempo actual y `s` es lo que devuelve `as_JSON()` del gestor de regiones:

```
{
 "time": t,
 "state": s,
}
```

Como puedes observar, hay dos versiones de los métodos `add_animal` y `set_region`, unas reciben la entrada como `JSON` mientras la otras reciben los objetos correspondientes después de crearlos. Las que reciben los objetos son `private`. El objetivo de tener las 2 versiones es facilitar el desarrollo y la depuración de la práctica: en tu primera implementación, antes de implementar las factorías, cambia esos métodos de `private` a `public` y úsalos directamente para añadir animales y regiones desde fuera. Solo cuando implementes las factorías cambialas a `private` de nuevo. De esta manera puedes depurar el programa sin haber implementado las factorías.

El Controlador

Todas las clases/interfaces de este apartado tienen que ir en el paquete **`simulator.control`**.

El controlador se implementa en la clase `Controller` que se encarga de `(1)` sacar las especificaciones de los animales/regiones desde un `JSONObject` y añadirlos al simulador; `(2)` ejecutar el simulador para un tiempo determinado y escribir los diferentes estados inicial y final en un `OutputStream` dado.

Atributos necesarios:

Tiene que tener un atributo (`_sim`) para la instancia de `Simulator`.

Constructoras:

La única constructora recibe como parámetro un objeto del tipo `Simulator` y lo almacena en el atributo correspondiente.

```
public Controler(Simulator sim)
```

Métodos necesarios:

- `public void load_data(JSONObject data)`: asumimos que `data` tiene las dos claves "`animals`" y "`regions`", siendo este último opcional. Los valores de estas claves son de tipo `JSONArray` (lista) y cada elemento de la lista es un `JSONObject` que corresponde a una especificación de animales o regiones. Para cada elemento hay que hacer lo siguiente (es muy importante añadir las regiones antes de añadir los animales):
  - Cada `JSONObject` en la lista de regiones (si la hay porque es opcional) tiene la forma

    `{"row": [rf,rt], "col": [cf,ct], "spec": O}`

    donde `rf, rt, cf, y ct son enteros y O` es un `JSONObject` que describe una región (ver el apartado "Las Factorías"). Hay que llamar a `_sim.set_region(R,C,O)` para cada `rf≤R≤rt` y `cf≤C≤ct` (es decir usando un bucle anidado para modificar varias regiones).
  - Cada `JSONObject` en la lista de animales tiene la forma

    `{"amount: N, "spec": O}`

    donde `N` es un número entero positivo y `O` es un `JSONObject` que describe un animal (ver el apartado "Las Factorías"). Hay que llamar a `_sim.add_animal(O)` en bucle `N` veces para añadir `N` animales de este tipo.

- `public void run(double t, double dt, boolean sv, OutputStream out)`: es un método para ejecutar el simulador (en bucle) llamando a `_sim.advance(dt)` hasta que pasen `t` segundos (es decir hasta que `_sim.get_time()>t`). Además, tiene que escribir en `out` una estructura JSON de la siguiente forma:

  ```
  {
   "in": init_state,
   "out": final_state
  }
  ```

  Donde `init_state` es el resultado que devuelve `_sim.as_JSON()` antes de entrar en el bucle, y `final_state` es el resultado que devuelve `_sim.as_JSON()` al salir del bucle.

  Además si el valor de `sv` es `true`, hay que mostrar la simulación usando el visor de objetos (ver el apartado "El Visor de Objetos").

Las Factorías

Todas las clases/interfaces de este apartado tienen que ir en el paquete `simulator.factories`.

Como en la práctica tenemos varías factorías vamos a usar genéricos para evitar duplicar código. A continuación detallamos cómo implementarlas paso a paso.

La Interfaz Factory<T>

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

```
public interface Factory<T> {
  public T create_instance(JSONObject info);
  public List<JSONObject> get_info();
}
```

El método `createInstance` recibe una estructura `JSON` que describe el objeto 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 la excepción correspondiente. 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, etc.

El método `get_info` devuelve una lista de objetos `JSON` que describen qué puede ser creado por la factoria, ver detalles a continuación.

Existen muchas formas de definir una factoría, que veremos durante el curso (o en la asignatura Ingeniería de Software). 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. Es una combinación de los patrones de diseño `Command` y `Factory`.

La Clase Builder<T>

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>`:

```
public abstract class Builder<T> {
  private String _type_tag;
```

```
    private String _desc;

    public Builder(String type_tag, String desc) {
        if (type_tag == null || desc == null || type_tag.isBlank() || desc.isBlank())
            throw new IllegalArgumentException("Invalid type/desc");
        _type_tag = type_tag;
        _desc = desc;
    }

    public String get_type_tag() {
        return _type_tag;
    }

    public JSONObject get_info() {
        JSONObject info = new JSONObject();
        info.put("type", _type_tag);
        info.put("desc", _desc);
        JSONObject data = new JSONObject();
        fill_in_data(data);
        info.put("data", data);
        return info;
    }

    protected void fill_in_data(JSONObject o) {
    }

    @Override
    public String toString() {
        return _desc;
    }

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

El atributo `_type_tag` coincide con el campo "`type`" de la estructura `JSON` correspondiente, y el atributo `_desc` describe que tipo de objetos pueden ser creados por este builder (para mostrar al usuario). Las subclases tienen que sobreescribir `fill_in_data` para rellenar los parámetros en `data` si es necesario.

Las clases que extienden a `Builder<T>` son las responsables de asignar un valor a `_type_tag` llamando a la constructora de la clase `Builder`, y también de definir el método `createInstance` 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 `get_info` devuelve un objeto `JSON` con dos campos correspondientes a `_type_tag` y `_desc`, el cual será utilizado por el método `get_info()` de la factoría. Si queremos añadir más información tenemos que sobreescribir `fill_in_data` para rellenarla.

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

- `SelectFirstBuilder`
- `SelectClosestBuilder`
- `SelectYoungestBuilder`
- `SheepBuilder`

- `WolfBuilder`
- `DefaultRegionBuilder`
- `DynamicSupplyRegionBuilder`

A continuación puedes encontrar el `JSON` correspondiente que admite cada builder. Todos los *builders* deben lanzar excepciones cuando los datos de entrada no son válidos (usar `IllegalArgumentException` con un mensaje informativo).

El `JSON` que admiten los Builders

`SelectFirstBuilder`

```
{
 "type": "first"
 "data": {}
}
```

`SelectClosestBuilder`

```
{
 "type": "closest"
 "data": {}
}
```

`SelectYoungestBuilder`

```
{
 "type": "youngest"
 "data": {}
}
```

`SheepBuilder`

```
{
 "type": "sheep"
 "data": {
      "mate_strategy" : { … }
      "danger_strategy" : { … }
      "pos" : {
          "x_range" : [ 100.0, 200.0 ],
          "y_range" : [ 100.0, 200.0 ]
      }
 }
}
```

El valor de la clave `"mate_strategy"` es un `JSON` de una estrategia que hay que construir usando la factoría de estrategias. Es opcional, y si no existe usamos la estrategia `SelectFirst.`

El valor de la clave `"danger_strategy"` es un `JSON` de una estrategia que hay que construir usando la factoría de estrategias. Es opcional, y si no existe usamos la estrategia `SelectFirst.`

La clave `"pos"` es opcional, si no existe usamos `null`. Si existe hay que elegir una posición inicial donde la coordenada `X` esté dentro del rango `"x_range"` y la coordenada `Y` esté dentro del rango `"y_range".`

Nótese que este builder requiere acceso a una factoría de estrategias (hay que pasarla como parámetro a la constructora).

`WolfBuilder`

```
{
```

```
 "type": "wolf"
 "data": {
      "mate_strategy" : { ... }
      "hunt_strategy" : { ... }
      "pos" : {
          "x_range" : [ 100.0, 200.0 ],
          "y_range" : [ 100.0, 200.0 ]
      }
 }
}
```

El valor de la clave `"mate_strategy"` es un `JSON` de una estrategia que hay que construir usando la factoría de estrategias. Es opcional, y si no existe usamos la estrategia `SelectFirst`.

El valor de la clave `"hunt_strategy"` un `JSON` de una estrategia que hay que construir usando la factoría de estrategias. Es opcional, y si no existe usamos la estrategia `SelectFirst`.

La clave `"pos"` es opcional, si no existe usamos `null`. Si existe hay que elegir una posición inicial donde la coordenada `X` esté dentro del rango `"x_range"` y la coordenada `Y` esté dentro del rango `"y_range"`.

Nótese que este builder requiere acceso a una factoría de estrategias (hay que pasarla como parámetro a la constructora).

`DefaultRegionBuilder`

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

`DynamicSupplyRegionBuilder`

```
{
 "type" : "default",
 "data" : {
        "factor" : 2.5,
        "food" : 1250.0
 }
}
```

La clave `"factor"` es opcional con valor por defecto `2.0`. La clave `"food"` es opcional con valor por defecto `1000.0`.

La Clase `BuilderBasedFactory<T>`

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

```
public class BuilderBasedFactory<T> implements Factory<T> {
  private Map<String, Builder<T>> _builders;
  private List<JSONObject> _builders_info;

  public BuilderBasedFactory() {
        // Create a HashMap for _builders, and a LinkedList _builders_info
        // ...
```

```
    }

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

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

    public void add_builder(Builder<T> b) {
          // add an entry "b.getTag() |-> b" to _builders.
          // ...
          // add b.get_info() to _buildersInfo
          // ...
    }

    @Override
    public T create_instance(JSONObject info) {
          if (info == null) {
                throw new IllegalArgumentException("'info' cannot be null");
          }

          // Look for a builder with a tag equals to info.getString("type"), in the
          //  map _builder, and call its create_instance method and return the result
          // if it is not null. The value you pass to create_instance is the following
          // because 'data' is optional:
          //
          //   info.has("data") ? info.getJSONObject("data") : new getJSONObject()
          // ...

          // If no builder is found or the result is null ...
          throw new IllegalArgumentException("Unrecognized 'info':" + info.toString());
    }

    @Override
    public List<JSONObject> get_info() {
          return Collections.unmodifiableList(_builders_info);
    }
}
```

Como Crear e Inicializar Las Factorías

Utilizaremos la clase `BuilderBasedFactory` para crear 3 factorías (para las estrategias, para los animales, y para las regiones). Este ejemplo muestra como podemos crear una factoría de estrategias:

```
// initialize the strategies factory
List<Builder<SelectionStrategy>> selection_strategy_builders = new ArrayList<>();
selection_strategy_builders.add(new SelectFirstBuilder());
selection_strategy_builders.add(new SelectClosestBuilder());
Factory<SelectionStrategy> selection_strategy_factory =
        new BuilderBasedFactory<SelectionStrategy>(selection_strategy_builders);
```

Recuerda que `SheepBuilder` y `WolfBuilder` requieren acceso a la factoría de estrategias (hay que pasarla

como parámetro a sus constructoras).

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`). Tendrás que completar la clase `Main` para analizar todos los posibles argumentos que aparecen abajo.

Al ejecutar Main con el argumento `-h` (o `--help`) debe mostrar por consola lo siguiente (ahora no muestra todos los parámetros):

```
usage: simulator.launcher.Main [-dt <arg>] [-h] [-i <arg>] [-m <arg>] [-o
       <arg>] [-sr <arg>] [-sv] [-t <arg>]
 -dt,--delta-time <arg>      A double representing actual time, in
                             seconds, per simulation step. Default value: 0.03.
 -h,--help                   Print this message.
 -i,--input <arg>            Initial configuration file.
 -o,--output <arg>           Output file, where output is written.
 -sv,--simple-viewer         Show the viewer window in console mode.
 -t,--time <arg>             An real number representing the total
                             simulation time in seconds. Default value: 10.0.
```

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

```
-i resources/examples/ex1.json -o resources/tmp/myout.jso -t 60.0 -dt 0.03 -sv
```

que ejecuta el simulador usando como fichero de entrada `resources/examples/ex1.json` y creando el fichero de salida `resources/tmp/myout.json`. El parámetro `-t` indica la duración de la simulación (`60.0` segundos), el parámetro `-dt` indica el valor del tiempo-por-paso en segundos (`0.03`), y `-sv` indica que queremos mostrar el visor.

El código proporcionado implementa sólo las opciones `-i` y `-t`. Hay que extenderlo para implementar el resto de opciones como se indica en la salida de ayuda arriba, y almacenar sus valores en atributos para poder usarlos desde otros métodos.

Además hay que completar la parte que ejecuta el simulador:

- Completar el método `init_factories` para inicializar las factorías y almacenarlas en los atributos correspondientes.
- Completar el método `start_batch_mode` para que haga lo siguiente `(1)` cargar el archivo de entrada en un `JSONObject`; `(2)` crear el archivo de salida; `(3)` crear una instancia de `Simulator` pasando a su constructora la información que necesita; `(4)` crear una instancia de `Controller` pasandole el simulador; `(5)` llamar a `load_data` pasandole el `JSONObject` de la entrada; y `(6)` llamar al método `run` con los parámetros correspondents; y `(7)` cerrar el archivo de salida.

El archivo de entrada continente un `JSON` de la siguiente forma:

```
{
 "width": w,
 "height": h,
 "rows": c,
 "cols": e,
 "animals": [a_0,a_1,...,a_k]
 "regiones": [r_0,r_1,...,e_l]
}
```

Donde `w`, `h`, `c` y `r` son enteros (úsalos para crear la instancia de `Simulator`), y a_0 y r_0 son objetos `JSON` que va a usar `load_data` del controlador (ver el apartado "El Controlador").

Apéndice

Análisis y Creación de Datos JSON en Java

JavaScript Object Notation (`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. Una estructura `JSON` es un texto estructurado de la siguiente forma:

{ "key_1": $value_1$, ..., "$value_n$": valuen }

donde key_i es una secuencia de caracteres (que representa una clave) y $value_1$ puede ser un número, un string, otra estructura `JSON`, o un array $[o_1,...,o_k]$, donde o_i puede ser como $value_i$.

En el directorio `lib` hay una librería que permite analizar un fichero `JSON` y convertirlo en objetos `Java`. Esta librería 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`.

El Visor de Objetos

El visor es una clase que proporcionamos para visualizar el estado del simulador (en la práctica 2 vais a implementar una interfaz gráfica más avanzada). La clase del visor es `SimpleObjectViewer` que se encuentra en `viewer.jar` (está en la carpeta `lib`).

La clase permite dibujar una lista de objetos, donde la descripción de cada objeto es una instancia del registro `SimpleObjectViewer.ObjInfo`

```
public record ObjInfo(String tag, int x, int y, int size) {
}
```

donde `tag` es la categoría del objeto, `(x,y)` es la coordenada donde queremos dibujar el objeto, y `size` es el tamaño (la longitud de un lado de un cuadrado en píxeles). Las coordenadas tienen que ser no negativas (si son negativas no aparecen en la ventana). La coordenada `x` es la horizontal, la coordenada `y` es la vertical, y la esquina superior-izquierda es el `(0,0)`. Hay otra constructora que no recibe `size` y usa `10` como un valor por defecto.

Para dibujar los animales es necesario convertir una lista de tipo `List<? extends AnimalInfo>` a `List<ObjInfo>` usando el siguiente método:

```
private List<ObjInfo> to_animals_info(List<? extends AnimalInfo> animals) {
 List<ObjInfo> ol = new ArrayList<>(animals.size());
 for (AnimalInfo a : animals)
       ol.add(new ObjInfo(a.get_genetic_code(),
                          (int) a.get_position().getX(),
                           (int) a.get_position().getY(),8));
 return ol;
}
```

Puedes reemplazar el tamaño `8` por algo que depende de `a.get_age()` para que los objetos tengan tamaños según su edad, por ejemplo `(int)Math.round(a.get_age())+2`. Al principio del método `run` en la clase `Controller`, usar el siguiente código para crear la instancia de la clase `SimpleObjectViewer` y

dibujar el estado inicial:

```
SimpleObjectViewer view = null;
if (sv) {
   MapInfo m = _sim.get_map_info();
   view = new SimpleObjectViewer("[ECOSYSTEM]",
                                 m.get_width(), m.get_height(),
                                 m.get_cols(), m.get_rows());
   view.update(to_animals_info(_sim.get_animals()), _sim.get_time(), dt);
}
```

Para dibujar el estado del simulador en cada iteración se puede usar la siguiente llamada después de la llamada a `_sim.advance(dt)`:

```
if (sv) view.update(to_animals_info(_sim.get_animals()), _sim.get_time(), dt);
```

Nótese que el método `view.update` detiene la ejecución para que el tiempo real pasado desde la última llamada a `view.update` sea igual (o mayor) de `dt`, y de esta manera el tiempo real y el tiempo del simulador serán similares (si no pasas `dt` no se detiene el tiempo).

Al final del método `run`, se puede usar el siguiente código para cerrar la ventana del visor (también se puede cerrar con un click sobre el icono **x** como cualquier otra ventana):

```
if (sv) view.close();
```

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(...);
```

Ejemplos de Entrada

El directorio `resources/examples/input` incluye algunos ejemplos de ficheros de entrada y el directorio `resources/expected_output` contiene las salidas correspondientes cuando se ejecuta el simulador como se indica en `resources/examples/expected_output/README.md`.

Recuerda que proporcionamos los ficheros de salida sólo para poder ver la estructura general de la salida, y que la salida de tu implementación no tiene que ser idéntica.

Generación de Números Aleatorios

En muchas partes del simulador usamos números aleatorios. En `Java` esto normalmente se hace usando la clase `Random`. Esta clase usa una semilla a partir de la cual genera números aleatorios, y por defecto usa el tiempo actual como semilla y así en cada ejecución generamos números aleatorios distintos. Sin embargo, existen situaciones en las que *queremos generar los mismos números aleatorios en cada ejecución*, por ejemplo cuando depuramos el programa, algo que se puede conseguir si siempre usamos la misma instancia de `Random` con la misma semilla.

La clase `simulator.misc.Utils` que proporcionamos incluye una instancia de la clase `Random` como un atributo estático `_rand`. Siempre úsala para garantizar que generamos números aleatorios usando la misma instance de `Random`, y si quieres usar la misma semilla en varias ejecuciones puedes añadir la siguiente línea al principio del método `Main.main` (puedes cambiar la semilla `21474836471` por cualquier número):

```
Utils._rand.setSeed(21474836471);
```

La Clase Vector2D

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

En el paquete `simulator.misc` hay una clase `Vector2D`, que implementa un vector y ofrece operaciones correspondientes:

Operación	Descrepción	En Vector2D
Suma	$\vec{a} + \vec{b}$ se define como el nuevo vector $\left(a_1 + b_1, a_2 + b_2\right)$	**`a.plus(b)`**
Resta	$\vec{a} - \vec{b}$ se define como el nuevo vector$\left(a_1 - b_1, a_2 - b_2\right)$	**`a.minus(b)`**
Multiplicación escalar	$a \cdot \vec{c}$ (o $c \cdot \vec{a}$), donde c es un número real, se define como el nuevo vector $\left(c * a_1, c * a_2\right)$	**`a.scale(c)`**
Longitud	La longitud (o magnitud) de $\vec{a}$ denotado como $\left\|\vec{a}\right\|$ 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 $\vec{a} \cdot \frac{1}{\left\|\vec{a}\right\|}$	**`a.direction()`**
Distancia	la distancia entre $\vec{a}$ y $\vec{b}$ se define como $\left\|\vec{a} - \vec{b}\right\|$	**`a.distanceTo(b)`**

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 `plus`, `minus`, etc.) devuelven nuevas instancias.

Ajustar posiciones

Se puede usar el siguiente código para ajustar la posición `(x,y)` para que esté dentro del mapa, suponiendo que la anchura del mapa es `width` y la altura es `height`:

```
while (x >= width) x = (x - width);
while (x < 0) x = (x + width);
while (y >= height) y = (y - height);
while (y < 0) y = (y + height);
```

Esto hace que cuando el animal sale de un lado, aparezca en el otro.