Programación 3

Universidad de Alicante, 2019–2020

Práctica 3

Plazo de entrega: **Hasta el domingo 24 de noviembre de 2019 a las 23:59h**.

Peso relativo de está práctica en la nota de prácticas: 30%

IMPORTANTE: Todos tus ficheros de código fuente deben utilizar la codificación de caracteres UTF-8. No entregues código que use otra codificación (como Latin1 o iso-8859-1).

En caso de duda, consulta primero la sección 'Aclaraciones' al final de este enunciado. La iremos actualizando de vez en cuando.

Block World: Más bloques, nuevas criaturas y entrada/salida

En esta práctica vamos a trabajar los siguientes conceptos

- entrada/salida desde fichero o consola
- herencia simple de implementación
- polimorfismo (sobrescritura)
- herencia simple de interfaz

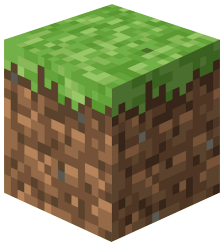
Para ello, vamos a introducir **entidades** de diferente tipo en el juego (monstruos y animales…), así como nuevas mecánicas de juego, como poder crear/destruir bloques y atacar a estas entidades.

También introduciremos un tipo de bloque *transitable*, a través del cual el jugador podrá pasar. Para ello introduciremos dos tipos nuevos de materiales líquidos: agua y lava. Tanto el jugador como las

criaturas del juego pueden estar dentro del agua sin perder salud (es decir, son anfibios y pueden respirar perfectamente en el agua). Sin embargo, los bloque de lava pueden causar daño al atravesarlos.

Las entidades y el jugador tienen algunas cosas en común: se pueden mover y algunas entidades pueden también atacar al jugador. Nos basaremos en estas cualidades comunes para crear jerarquías de herencia.

Bloques



Vamos a diferenciar dos tipos de bloques: sólidos y líquidos. Los bloques sólidos son el mismo tipo de bloque que teníamos en la práctica anterior. No se pueden atravesar, pero ahora pueden romperse si el jugador los golpea con algo. Cuando el bloque se rompe puede liberar items del mismo material del que está hecho el bloque. Estos items pasan a ocupar la posición del bloque destruido.

Los bloques líquidos no se rompen al golpearlos pero el jugador los puede atravesar, es decir, ocupar su misma posición en un instante dado. Existen sólo dos tipos de materiales líquidos: agua y lava. Los bloques de lava dañan al jugador al atravesarlos. Cada vez que el jugador se mueve a una posición ocupada por un bloque de lava, pierde un punto de salud. ¡Así que camina con cuidado!

Entidades

Vamos a generalizar el concepto de jugador como entidad viva que puede ocupar una posición en el mundo y tiene un nivel de salud determinado. Estas entidades estarán representadas por la clase *LivingEntity*, con un nivel de salud que puede ser de un máximo de 20 puntos, y que puede aumentar

o disminuir en función de la interacción de esta entidad con otras entidades. Por ejemplo, una entidad puede ser atacada por el jugador, causándole un cierto daño, y por tanto disminuyendo su salud. Las entidades mueren si su nivel de salud es igual o menor que cero, en cuyo caso desaparecen del mundo. Las entidades ocupan una única posición, es decir, tienen tamaño 1x1x1, como los bloques. El jugador no podrá moverse a posiciones ocupadas por entidades.

El jugador (*Player*) será por tanto un tipo de entidad. En el diagrama UML de esta práctica verás que algunos de los atributos y métodos que antes estaban en *Player* (los relativos a la salud y la posición del jugador), ahora están en *LivingEntity* y *Player* simplemente los heredará.

Otros tipos de entidades son las criaturas (*Creature*) que se dividen en dos tipos: monstruos (*Monster*) y animales (*Animals*). Para simplificar, consideraremos que las criaturas no se pueden mover. Son creadas en una posición del mundo y permanecen ahí sin moverse durante toda su existencia. Estas criaturas pueden ser atacadas por el jugador. Cada ataque del jugador decrementará su salud en una cierta cantidad de puntos.

Animales

Los animales, en caso de morir en un ataque liberan un item de tipo BEEF que pasa a ocupar la posición de la entidad. Los animales nunca atacan a otras entidades.

Monstruos

Los monstruos, en caso de sobrevivir a un ataque del jugador, le contraatacarán causandole una cierta cantidad de daño. Cuando un monstruo muere, desaparece del mundo sin dejar nada en su lugar.

Jugador

El jugador tiene las mismas propiedades y métodos que en la práctica anterior, sólo que ahora algunas de ellas las hereda de *LivingEntity*.

En esta práctica, además de recoger items, el jugador va a poder realizar ciertas interacciones con el mundo y otras entidades, como crear y destruir bloques y atacar a otras entidades que se encuentren en posiciones adyacentes. Para ello necesitamos modelar la orientación del jugador hacia una posición adyacente.

Orientación del jugador

Además de su posición, el jugador también tiene una orientación, que puede ser a cualquiera de las posiciones adyacentes a la suya. Inicialmente, el jugador está orientado hacia el sur. es decir, si su posición es (x,y,z), está orientado hacia la posición (x,y,z+1). La orientación determinará hacia que bloque o entidad adyacente el jugador ejecuta sus acciones. El jugador puede estar orientado hacia posiciones no válidas del mundo, es decir, cuando el jugador se encuentra en alguna de las posiciones límite del mundo, puede estar orientado hacia una posición más allá del límite. Si por ejemplo se encuentra en una posición lo más al norte posible de su mundo, puede orientarse hacia una posición que esté hacia el norte, noreste o noroeste, aunque esas posiciones no existan.

El item que está en la mano del jugador se puede usar en la posición adyacente hacia la cual se encuentra orientado el jugador. Dependiendo de lo que haya en esa posición (un bloque, una entidad o es una posición vacía), sucederá algo diferente al usar el item.

Orientar al jugador no consume puntos de alimento o salud.

Interacción entre el jugador y los bloques

El item que el jugador lleva en la mano puede usarse para golpear un bloque y destruirlo. Además, si el tipo de item es un bloque, se puede utilizar para crear nuevos bloques del mismo material que el item en una posición adyacente vacía.

Un jugador puede destruir un bloque que se encuentre en una posición adyacente a la suya. Cuando un bloque es destruido, libera los items que pueda guardar en su interior, que pasan a ocupar la posición del bloque. Cuando el jugador se desplace a esa posición, automáticamente recogerá los items y los guardará en su inventario.

Para destruir un bloque, el jugador debe golpearlo hasta que consiga romperlo. La cantidad de veces que debe hacerlo depende de la dureza del material y de con qué tipo de item lo está golpeando el jugador. Véase más abajo el método `useItem()` de la clase *BlockWorld*.

Interacción entre el jugador y las criaturas

Si un jugador accede a una posición adyacente a otra en la cual se encuentra una criatura, puede decidir atacarla. Las criaturas nunca atacarán a un jugador por iniciativa propia. Siempre atacará primero el jugador, causando un cierto daño a la entidad en función de que tipo de item lleve en la mano.

En caso de que se trate de un monstruo, si éste sobrevive, es decir, conserva un nivel de salud mayor que cero tras el ataque, atacará al jugador causándole una cierta cantidad de daño, restándole puntos de salud.

Entrada/Salida

Entrada

Mediante un fichero de texto que contiene diferentes órdenes, una por línea, crearemos el mundo, items, entidades y al jugador en localizaciones determinadas, desplazaremos al jugador y ejecutaremos acciones como atacar otras entidades o crear y destruir bloques. La clase encargada de leer el fichero de entrada y ejecutar las órdenes que en él aparecen es *BlockWorld*.

Formato del fichero de entrada

Este es el formato general del fichero de entrada:

```
<semilla> <tamaño> <nombre>
<orden>*
```

donde el asterisco indica que puede haber cero o más líneas que contienen órdenes. La primera línea esta formada por tres campos separados por espacios en blanco:

`<semilla>` es un número entero que se usa para inicializar el algoritmo de generación del mundo.

`<tamaño>` número entero positivo que indica el tamaño del mundo en bloques. Como se explicó en la práctica anterior, este tamaño indica la longitud del mundo en el eje *x* y en el eje *z*, con la localización x=0,z=0 en el centro del mundo.

`<nombre>` Cadena que representa el nombre del mundo. Puede contener espacios en blanco.

Ordenes del fichero de entrada

Cada órden ocupa una sola línea.

órden	descripción

órden	descripción
`move dx dy dz`	mueve al jugador, que está en la posición (x,y,z), a la posición (x+dx,y+dy,z+dz). dx, dy, dz pueden valer -1, 0 o 1.
`orientate dx dy dz`	orienta al jugador, que está en la posición (x,y,z), hacia la posición (x+dx,y+dy,z+dz). dx, dy, dz pueden valer -1, 0 o 1. Si los tres valores son cero, la orientación del jugador no cambia.
`useItem n`	acción de usar el item que está en la mano del jugador 'n' veces seguidas. El item se usa sobre la posición objetivo, determinada por la orientación del jugador.
`show`	Muestra información sobre el jugador y su entorno
`selectItem n`	Selecciona el n-ésimo item del inventario del jugador para llevarlo en la mano. `n` debe estar entre 1 y el tamaño máximo del inventario.

Ejemplo de fichero de entrada:

```
2019 101 Mundo de las tinieblas
move 0 0 1
move -1 0 1
orientate 1 0 0
selectItem 2
useItem 1
show
```

`useItem n` funciona de la siguiente manera cuando el item que el jugador lleva en la mano es un bloque, una herramienta o un arma:

- *sobre un bloque* : produce un daño equivalente al valor expresado abajo multiplicado por *n*. Si el daño es igual o mayor que el valor del material del bloque, el bloque será eliminado, pudiendo liberar los items que contiene antes de ser eliminado.
- *sobre una criatura* : produce un ataque a la entidad, la entidad recibe los puntos de daño indicados abajo multiplicados por 'n'. Si la criatura es un monstruo que no muere tras el ataque,

ésta atacará a su vez al jugador con ‘0.5*n’ puntos de daño.

- *sobre una posición con items* : no sucede nada.
- *sobre una posición vacía* : si tenemos un bloque en la mano, se crea un bloque de ese mismo material en esa posición.

Daño producido por el tipo de item que el jugador lleva en la mano:

- bloque : daño = 0.1
- herramienta o arma : daño = valor del item en el tipo enumerado *Material*

Cada acción `useItem` se considera una acción independiente, que consume 0.1 puntos de alimento o salud, según corresponda (esto ya sucedía en la práctica anterior). Por tanto, una orden `useItem n` consume `0.1*n` puntos.

Cuando el item es comida, la acción no consume puntos y la posición objetivo es irrelevante. El jugador ingiere la cantidad de comida especificada por el parámetro ‘n’, tal y como ocurría en la práctica anterior.

Si el jugador no lleva ningún item en la mano, no ocurre nada ni el jugador pierde puntos.

El juego termina cuando no quedan órdenes por procesar o el jugador a muerto, en cuyo caso no se ejecutan las órdenes restantes.

En la entrada pueden aparecer órdenes desconocidas, en cuyo caso se mostrará un error por la salida de error y se continuará leyendo la siguiente órden. (Véase el método *BlockWorld.play(Scanner)*).

Asume que los ficheros de entrada no contendrán errores de formato, por lo cual puedes asumir que cada línea que contiene una orden conocida estará bien formada.

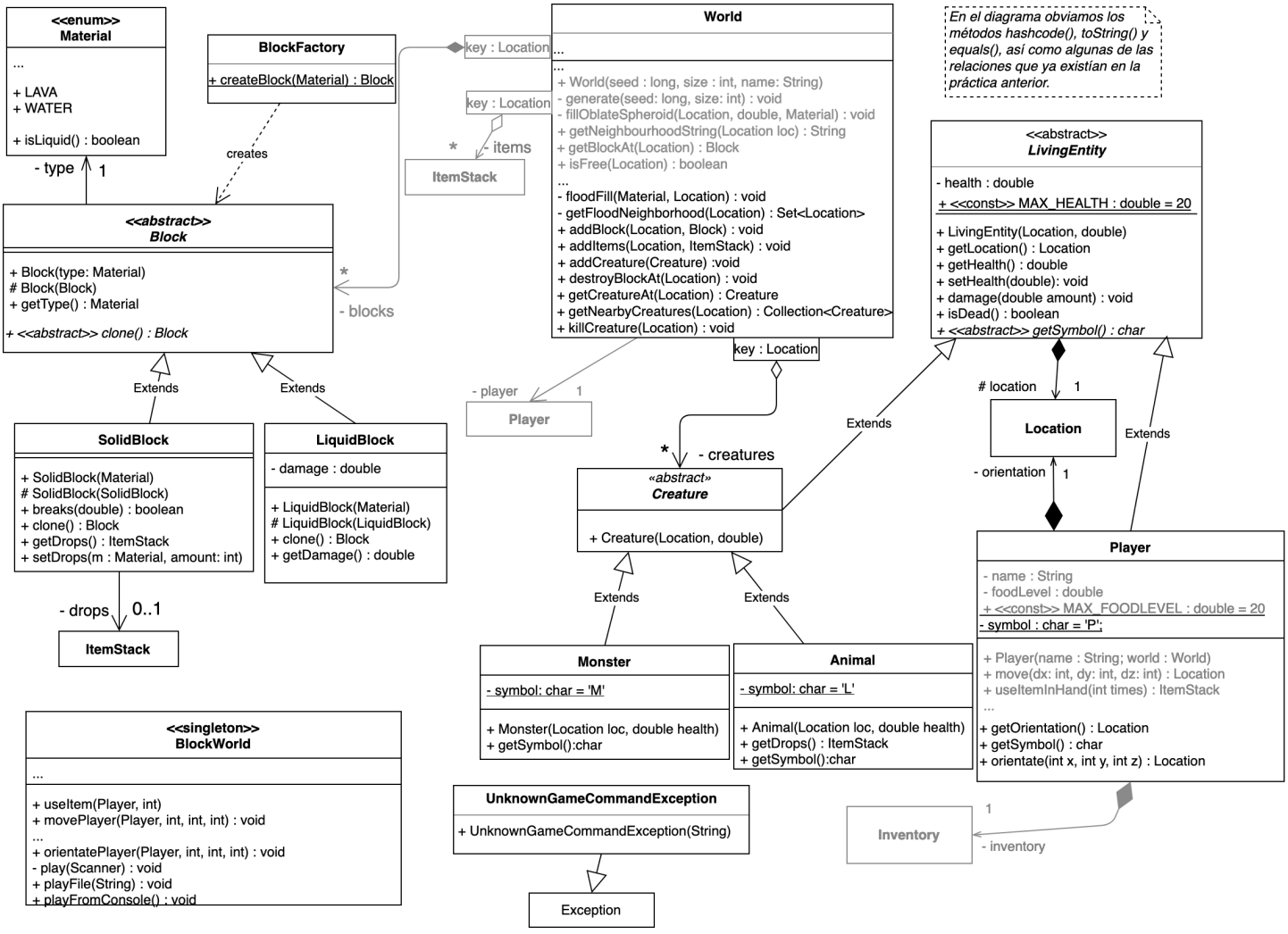
Salida

La salida será por consola. Mediante la orden *show* podemos imprimir el inventario del jugador, nivel de salud y comida, su posición, y lo que hay en las posiciones adyacentes al jugador (método *BlockWorld.showPlayerInfo()*).

Ejemplo de salida al invocar la orden *show*:

```
Name=Steve
Location{world=Block World,x=4.0,y=4.0,z=4.0}
Orientation=Location{world=Block World,x=1.0,y=0.0,z=0.0}
Health=9.2
Food level=18.5
Inventory=(inHand=(WOOD_SWORD,1),[(APPLE,3), (IRON_PICKAXE,1)])
nnn nnn nnn
... .PL nnn
... MNB nnn
```

Diagrama de clases



En este diagrama de clases UML puedes ver que algunas cosas están en gris, se trata atributos, métodos y relaciones que ya existían en la práctica anterior. Los métodos en gris sufrirán algún cambio en su implementación en esta práctica. Los métodos que permanecen igual que en la práctica anterior no aparecen en el diagrama. El resto son componentes que añadiremos en esta práctica.

Clases

Material

Incluye las nuevas constantes LAVA y WATER, en ese orden, según la tabla de abajo. Cambia el carácter asociado al material SAND por ‘n’ (ene minúscula).

Material	Valor	Símbolo
LAVA	1.0	#
WATER	0.0	@

Estos son materiales para bloques líquidos. El resto de material para bloques se usará para crear bloques sólidos.

isBlock()

Incluye los nuevos materiales LAVA y WATER.

isLiquid()

Devuelve cierto si el material es de tipo bloque líquido (LAVA o WATER).

Jerarquía de *Block*

Para modelar los distintos tipos de bloques hemos generalizado el concepto de bloque. Lo que en la práctica anterior era la clase *Block*, ahora es la clase *SolidBlock*, que representa bloques que no se pueden atravesar y que pueden contener items en su interior. El otro tipo de bloques que vamos a tener son los bloques hechos de material líquido, representados por la clase *LiquidBlock*, los cuales se pueden atravesar pero no contienen items. Como superclase de estas dos, definimos la clase abstracta *Block*, que como puedes observar, mantiene su relación con *Material*, pero la relación que tenía con *ItemStack* y los métodos que gestionan esta relación pasan a la clase *SolidBlock*.

Crea primero las subclases *SolidBlock* y *LiquidBlock* como clases vacías y mueve el código relevante de *Block* a *SolidBlock*. Luego haz la clase *Block* abstracta, añádele el método *clone()* e impleméntalo

en las subclases.

Block

Block(Material)

Constructor sobrecargado.

Lanza *WrongMaterialException* si el material no es material para bloques.

Block(Block)

Constructor de copia. Tiene visibilidad protegida.

getType()

Mismo método que en la práctica anterior.

clone()

Método abstracto. Su objetivo es devolver una copia del objeto 'this' al ser invocado (se implementa en las subclases).

Te permitirá obtener una copia de cualquier tipo de bloque (SolidBlock o LiquidBlock) cuando lo necesites.

hashCode() y equals(Object)

Regenéralos, ya que los atributos de *Block* han cambiado.

toString()

Mismo método que en la práctica anterior.

SolidBlock

SolidBlock(Material)

Constructor sobrecargado. Crea un bloque del material dado que no contiene items.

Lanza *WrongMaterialException* si el material no es material para bloques sólidos.

SolidBlock(SolidBlock)

Constructor de copia.

breaks(double damage)

Devuelve cierto si la cantidad de daño (*damage*) es igual o mayor que la dureza del material del bloque.

clone()

Implementación de *Block.clone()*

getDrops() y setDrops(Material, int)

Son los mismos métodos que antes estaban en *Block* y hemos 'bajado' a *SolidBlock*.

hashCode() y equals(Object)

Genéralos automáticamente con Eclipse.

toString()

No se implementa.

LiquidBlock

LiquidBlock(Material)

Constructor sobrecargado. Asigna la cantidad de daño correspondiente al valor del material.

Lanza *WrongMaterialException* si el material no es material para bloques líquidos.

LiquidBlock(LiquidBlock)

Constructor de copia.

clone()

Implementación de *Block.clone()*

getDamage()

Devuelve el daño que este bloque produce al atravesarlo.

hashCode() y equals(Object)

Genéralos automáticamente con Eclipse.

toString()

No se implementa.

BlockFactory

Esta es una clase *factoría*, cuyo único método tiene como cometido crear bloques según el tipo de material utilizado.

createBlock(Material)

Devuelve un nuevo bloque, sólido o líquido, creado a partir del material dado. Si el material es líquido, creará un *LiquidBlock*, sino creará un *SolidBlock*.

Lanza indirectamente *WrongMaterialException* si intentamos construir bloques con materiales no adecuados.

LivingEntity

Esta clase representa una entidad que vive en un mundo. Tienen un cierto nivel de salud y ocupan una posición en el mundo. Parte de su implementación la traeremos de la clase *Player* de la práctica

anterior. En concreto, los atributos ‘location’ y ‘health’ de *Player* ahora pasan a esta clase. Los métodos

- getHealth()
- setHealth(double)
- getLocation()
- isDead()

tendrán la misma implementación que tenían en *Player* en la práctica anterior.

Esta clase y todas sus subclases se deben definir dentro del paquete `model.entities` .

LivingEntity(Location loc, double health)

Crea una entidad en la posición dada y con el nivel de salud indicado, que satura en MAX_HEALTH.

damage(double amount)

Inflige daño a la entidad restándole ‘amount’ unidades de salud.

getSymbol()

Método abstracto que devuelve un carácter que representa a la entidad.

Métodos toString(), hashcode() y equals(Object)

Genéralos automáticamente con Eclipse usándolos los atributos ‘location’ y ‘health’.

Player

Esta clase heredará la implementación de los métodos que hemos ‘subido’ a la superclase *LivingEntity* (ya no se implementan aquí). El resto de métodos que teníamos en la práctica anterior mantienen su implementación, excepto el constructor y *move()* que ahora deben tener en cuenta la orientación del jugador, si procede.

En el constructor, invoca al constructor de LivingEntity con una posición (0,0,0) y luego modifica el atributo *location* (tiene visibilidad *protected*) para colocar al jugador en la superficie del mundo.

Un jugador se crea con su orientación inicial hacia la posición que está al sur de la del jugador. La orientación puede guardarse internamente como una posición absoluta o relativa a la posición del jugador. Si decides implementarlo como una posición absoluta, *move()* tendrá que actualizarla cada vez que el jugador se mueve.

useItemInHand(int times)

La única modificación en este método es que debe devolver el item que lleva el jugador en la mano (el método devuelve ahora un ItemStack). Este item puede ser 'null' si, por ejemplo, el item era comida y la hemos ingerido toda.

getOrientation()

Devuelve la orientación del jugador como una posición absoluta.

getSymbol()

Devuelve el carácter 'P', que representa al jugador.

orientate(int dx, int dy, int dz)

Cambia la orientación del jugador. Si el jugador está en la posición (x,y,z), lo orienta hacia la posición (x+dx,y+dy,z+dz). Devuelve un objeto Location que representa dicha posición en el mundo del jugador.

Lanza *EntityIsDeadException* si se intenta orientar a un jugador que ha muerto. Esta condición se comprueba en primer lugar.

Lanza *BadLocationException* Si dx==dy==dz==0 (un jugador no puede estar orientado hacia sí mismo) o la orientación no es a una posición adyacente.

hashcode() y equals()

Regenera estos métodos usando todos los atributos de instancia de *Player*, después de haber implementado *LivingEntity* y el resto de *Player*.

Fíjate como tanto *hashcode()* como *equals()* invocan a los métodos homónimos de la superclase.

toString()

Debes modificarlo para incorporar información sobre la orientación del jugador. Abajo tienes un ejemplo de la cadena que debe producir este método (seis líneas en total). La orientación se indica como una posición relativa a la posición del jugador.

```
Name=Steve
Location{world=test,x=22.0,y=62.0,z=18.0}
Orientation=Location{world=test,x=1.0,y=-1.0,z=-1.0}
Health=8.9
Food level=16.7
Inventory=(inHand=(IRON_SWORD,1),[(WOOD_SWORD,1), (GRASS,1), (IRON_SWORD,1), (IRON_SWORD,1)])
```

Creature

Creature es una clase abstracta que representa a criaturas que habitan los mundos de BlockWorld, a excepción de los jugadores. Pueden ser atacadas por los jugadores.

No es necesario implementar los métodos *hashcode()*, *equals()* y *toString()* para esta clase.

Creature(Location loc, double health)

Constructor sobrecargado.

Métodos toString(), hashcode() y equals(Object)

No se implementan.

Monster

Representa a criaturas que pueden atacar a los jugadores.

Monster(Location loc, double health)

Constructor sobrecargado.

getSymbol()

Devuelve el carácter ‘M’, que representa a un monstruo.

Métodos hashcode() y equals(Object)

No se implementan.

toString()

Impleméntalo para obtener la misma cadena que *LivingEntity.toString()*, pero cambiando 'LivingEntity' por 'Monster'.

Animal

Representa a criaturas que no atacan al jugador, pero que pueden proporcionarle carne si el jugador las mata.

Animal(Location loc, double health)

Constructor sobrecargado.

getDrops()

Proporciona una unidad de Material.BEEF.

getSymbol()

Devuelve el carácter 'L', que representa a un animal.

Métodos hashcode() y equals(Object)

No se implementan.

toString()

Impleméntalo para obtener la misma cadena que *LivingEntity.toString()*, pero cambiando 'LivingEntity' por 'Animal'.

World

Al igual que hicimos con las relaciones de *World* con *Block* e *ItemStack*, implementaremos la relación de *World* con *Creature* como un mapa indexado por objetos de tipo *Location*.

Como el jugador puede crear y destruir bloques, cada vez que esto suceda, se debe actualizar, si procede, el valor de *heightMap* en ese punto con `heightMap.set(x,z,y)`, de modo que siempre tengamos almacenada ahí la coordenada 'y' del bloque más alto. Así evitaremos tener que escribir código que la busque cada vez que la necesitemos (que será a menudo).

Los métodos *floodFill()* y *getFloodNeighborhood()* se proporcionan ya implementados en este archivo World_incomplete.java, junto a nuevas versiones de los métodos que crean el mundo, *generate()* y *fillOblateSpheroid()*. Ahora los mundos se generan con agua, lava y criaturas (monstruos y animales).

Los métodos que no aparecen en el diagrama UML y que ya existían en la práctica anterior no sufren ninguna modificación.

World(long, int, String)

Recuerda inicializar el mapa de criaturas.

Vuelve a descargar los métodos que generan el mundo y sustituye los métodos *generate()*, *fillOblateSpheroid()* que tenías en la práctica 2 por estas nuevas versiones. El fichero también contiene algunos métodos nuevos (privados) necesarios para generar agua y lava (*floodFill()* y *getFloodNeighborhood()*).

getBlockAt(Location)

Devuelve una copia del bloque que está en la posición dada, o 'null' si no hay ningún bloque.

Lanza *BadLocationException* si la posición 'loc' no pertenece a este mundo (pertenece a otro o no tiene un mundo asociado).

getNeighbourhoodString(Location)

En esta práctica debe mostrar también a las criaturas que se encuentran en el mundo. Además, dado que hay bloques líquidos cuya posición puede estar ocupada también por el jugador, una criatura o un item, en ese caso se representará a la entidad o al item y no al bloque.

Cada bloque, como en la práctica anterior, se representará por el símbolo del material que lo forma (son todo letras minúsculas). En el caso de objetos ItemStack que contienen material de bloque, se representarán con la misma letra, pero mayúscula.

Aquí tienes algunos ejemplos:

```
@@@ @@@ @@@
@@@ @P@ @@>
@@@ @@@ A@@
```

en este primer ejemplo, el jugador está bajo el agua (suponemos que es anfibio y puede respirar perfectamente bajo el agua). Por debajo de él, también bajo el agua, hay una manzana y una pala de hierro.

```
@@@ @@@ ggg
@@@ @PG ggd
@@@ ggg ddd
```

en este segundo ejemplo, el jugador está 'en la orilla' y hacia el este tiene un item de hierba ('G'), resultado de haber roto previamente un bloque de hierba que había en ese lugar.

Lanza *BadLocationException* si la posición 'loc' no pertenece a este mundo (pertenece a otro o no tiene un mundo asociado).

isFree(Location)

Comprueba si la posición dada no está ocupada por un bloque SOLIDO, el jugador o una criatura.

Lanza *BadLocationException* si la posición no es de este mundo.

addBlock(Location, Block)

Añade un bloque a este mundo en la posición dada. Si la posición estaba anteriormente ocupada por otro bloque, criatura o item, éstos serán eliminados del mundo. Este método actualiza el mapa de superficie del mundo, si procede.

Lanza *BadLocationException* si la posición no es de este mundo, está fuera de sus límites o está ocupada por el jugador.

addCreature(Creature)

Añade una criatura a este mundo. La criatura sólo puede colocarse en una posición libre. Si en la posición había items, estos serán eliminados del mundo.

Lanza *BadLocationException* si la posición de la criatura no es de este mundo, está fuera de sus límites o está ocupada.

addItems(Location, ItemStack)

Añade una pila de items a este mundo, en la posición dada, que debe estar libre. Si había otros items en esa posición, son reemplazados.

Lanza *BadLocationException* si la posición no es de este mundo, está fuera de sus límites o está ocupada.

destroyBlockAt(Location)

Destruye el bloque en la posición dada, eliminándolo del mundo. Coloca los items que el bloque pudiera contener en la misma posición. Estos items no 'caen' si no hay nada bajo ellos.

Los bloques que están a altura cero (la coordenada `y` de su posición en el mundo es cero) no se pueden eliminar del mundo.

Este método actualiza el mapa de superficie del mundo, si procede.

Lanza *BadLocationException* si la posición no es de este mundo, no hay un bloque en esa posición, o se trata de un bloque a altura cero.

getCreatureAt(Location)

Devuelve la criatura que se encuentra en la posición dada, o 'null' si no hay ninguna en esa posición o la posición no existe en este mundo.

Lanza *BadLocationException* si la posición no es de este mundo.

getNearbyCreatures(Location)

Devuelve todas las criaturas vivas que estén ocupando posiciones adyacentes a la dada.

Lanza *BadLocationException* si la posición no es de este mundo.

killCreature(Location)

Elimina del mundo la criatura que está en la posición dada.

Lanza *BadLocationException* si la posición no es de este mundo o no hay una criatura en esa posición.

equals(), hashCode() y toString()

Se mantienen igual que en la práctica anterior.

BlockWorld

movePlayer(Player, int, int, int) : void

Debes añadir código para tener en cuenta que podemos atravesar bloques líquidos y que estos pueden dañarnos al atravesarlos.

Lanza las mismas excepciones que en la práctica anterior.

useItem(Player p, int times)

Hace que el jugador ‘p’ use el item que lleva en la mano ‘times’ veces, invocando al método *Player.useItemInHand()*. Si el jugador no llevaba ningún item en la mano, era comida, o está orientado hacia una posición fuera de los límites del mundo, no hace nada más.

Si el item es un bloque, una herramienta o un arma, calcula el daño que produce (ver orden *useItem* en la sección de Entrada/Salida.

Dependiendo entonces de lo que haya en la posición hacia la que está orientado el jugador, hará lo siguiente:

- Si hay un bloque sólido y el daño es igual o mayor que el valor del material del bloque, el bloque será eliminado, liberando los items que contenga en su interior, que pasarán a ocupar la posición que ocupaba el bloque.
- Si hay una criatura la ataca infligiéndole el daño calculado. Si la criatura muere es eliminada del mundo; en caso de ser un animal, dejará en la posición que ocupaba un item de carne (BEEF). Si la criatura no muere y es un monstruo, contraatacará al jugador infligiéndole un daño igual a '0.5*times'.
- Si en la posición objetivo no hay nada y el item que el jugador lleva en la mano es de material de bloque, creará un bloque sólido de ese mismo material en esa posición (siempre será un bloque sólido por que el jugador no tiene forma de conseguir items de agua o lava). Este nuevo bloque no contendrá items. Siempre se creará un único bloque, independientemente del valor de 'times'.

Lanza indirectamente *EntityIsDeadException* e *IllegalArgumentException*.

orientatePlayer(Player p, int dx, int dy, int dz)

Invoca al método *orientate()* del jugador.

Lanza indirectamente *BadLocationException* y *EntityIsDeadException* (ver método *Player.orientate()*).

playFile(String path)

Abre el fichero de entrada indicado y ejecuta cada una de sus órdenes (invocando a *play(Scanner)*).

Lanza *FileNotFoundException* si no se encuentra el fichero.

playFromConsole()

Similar a *playFile()*, pero abre la entrada estándar en lugar de un fichero de texto para leer de ella las órdenes del juego. No lanza excepciones.

play(Scanner sc)

Ejecuta las órdenes que va leyendo, línea a línea, del objeto Scanner pasado como argumento. Captura las excepciones que se puedan producir durante la ejecución de dichas órdenes, mostrando el mensaje de la excepción por la salida de error. Si en la entrada aparece una órden desconocida, muestra un mensaje de error adecuado por la salida de error y continúa leyendo la siguiente órden. Deja de leer del fichero si no queda nada más que leer o el jugador ha muerto.

equals(), hashCode() y toString()

No se implementan para esta clase.

Programa principal

Aquí tienes una par de programas principales:

- Main3_interactivo.java puede recibir como argumento el nombre de un fichero, en cuyo caso lo abre y lee las órdenes de él. Si no recibe ningún argumento, lee las órdenes de la entrada estándar.

- Main3_RandomWalk.java Crea un mundo y hace que el jugador lo recorra de forma aleatoria. En cada paso, un tercio de las veces intenta usar el item de la mano sobre una posición adyacente, preferiblemente una sobre la que haya una criatura. Cada 20 movimientos selecciona un item de su inventario. La ejecución termina cuando el jugador muere o ha hecho un número máximo de movimientos.

Pruebas unitarias

Descarga los test de la práctica 2 adaptados a ésta práctica, que tienen en cuenta los cambios introducidos en este enunciado. Cópialos al paquete `model` dentro de la carpeta de código fuente `test` de tu proyecto Eclipse.

Tests previos

Aquí puedes descargar los test previos para esta práctica. El archivo comprimido contiene dos carpetas: `files` y `model`, que debes añadir a tu proyecto en una carpeta de código fuente (New -> 'Source folder') llamada `pretest`. Es necesario que se llame así porque la ruta que se usa en las pruebas para buscar los archivos de entrada y salida asume que los test se encuentran en una carpeta con ese nombre.

Como en la práctica anterior, hay algunos test que faltan por completar en cada fichero (busca el comentario `//TODO`). Estos test se usarán en la corrección final, por lo que te aconsejamos que los

completes para asegurarte una buena nota. En el examen de prácticas es muy posible que se os pida implementar algún test unitario, ¡por lo cual os conviene practicar!

Estructura de paquetes y directorios

La práctica debe ir organizada en los siguientes paquetes o directorios:

- `model` : Todas las clases menos las que están en los siguientes paquetes.
- `model.entities` : Todas las clases de la jerarquía de *LivingEntity*
- `model.exceptions` : Todas las excepciones
- `mains` : programas principales

Los ficheros fuente deben tener documentación (comentarios) en el formato de Javadoc, pero no se debe entregar los ficheros HTML que genera esta herramienta.

Documentación

Tu código fuente ha de incluir todos los comentarios de Javadoc que ya se indicaban en el enunciado de la primera práctica. No incluyas el HTML generado por Javadoc en el fichero comprimido que entregues.

Entrega

La práctica se entrega en el servidor de prácticas del DLSI.

Debes subir allí un archivo comprimido con tu código fuente (sólo archivos .java). En un terminal, sitúate en el directorio 'src' de tu proyecto Eclipse e introduce la orden

```
tar czvf prog3-p3.tgz model
```

Sube este fichero `prog3-p3.tgz` al servidor de prácticas. Sigue las instrucciones de la página para entrar como usuario y subir tu trabajo.

Evaluación

La corrección de la práctica es automática. Esto significa que se deben respetar estrictamente los formatos de entrada y salida especificados en los enunciados, así como la interfaz pública de las clases, tanto en la signatura de los métodos (nombre del método, número, tipo y orden de los argumentos de entrada y el tipo devuelto) como en el funcionamiento de éstos. Así, por ejemplo, el método `modelo.Coordenada(int, int)` debe tener dos enteros como argumento y guardarlos en los atributos correspondientes.

Tienes más información sobre el sistema de evaluación de prácticas en la ficha de la asignatura.

Además de la corrección automática, se va a utilizar una aplicación detectora de plagios. Se indica a continuación la normativa aplicable de la Escuela Politécnica Superior de la Universidad de Alicante en caso de plagio:

> “Los trabajos teórico/prácticos realizados han de ser originales. La detección de copia o plagio supondrá la calificación de”0" en la prueba correspondiente. Se informará la dirección de Departamento y de la EPS sobre esta incidencia. La reiteración en la conducta en esta u otra asignatura conllevará la notificación al vicerrectorado correspondiente de las faltas cometidas para que estudien el caso y sancionen según la legislación vigente".

Aclaraciones

- Aunque no se recomienda, se pueden añadir los atributos y métodos privados que se considere oportuno a las clases. No obstante, eso no exime de implementar TODOS los métodos presentes en el enunciado, ni de asegurarse de que funcionan tal y como se espera, incluso si no se utilizan nunca en la implementación de la práctica.
- Cualquier aclaración adicional aparecerá en esta sección.

Material

Las nuevas constantes LAVA y WATER deben añadirse, en ese orden, al final de la lista de materiales, tras IRON_SWORD.

World.addBlock(), World.addCreature(), World.addItems()

Se ha modificado la descripción de estos métodos para resaltar que deben lanzar BadLocationException cuando la posición no es del mundo actual, lo cual incluye el caso en que ésta queda fuera de los límites del mundo. Por ejemplo, para *World.addBlock()*, donde antes indicaba

> *Lanza BadLocationException si la posición no es de este mundo o está ocupada por el jugador.*

ahora indica

> *Lanza BadLocationException si la posición no es de este mundo, está fuera de sus límites o está ocupada por el jugador.*

De forma similar se ha modificado la descripción de *World.addCreature()* y *World.addItems()*.

BlockWorld.useItem()

Tras invocar a *Player.useItemInHand()*, no debe hacer nada más cuando la orientación del jugador sea hacia una posición fuera de los límites del mundo.

Se ha indicado esta circunstancia también en la descripción del método.

Sobre la orientación del jugador

El jugador puede orientarse hacia cualquier posición adyacente a la suya, incluso si esa posición queda fuera de los límites del mundo. En este sentido, conviene aclarar que una *posición adyacente* es aquella que, relativa a la posición del jugador, tiene como valor absoluto de sus coordenadas 0 o 1, exceptuando la propia posición del jugador. Estas posiciones (en coordenadas absolutas) pueden ser *posiciones adyacentes válidas* si están dentro de los límites del mundo, o *posiciones adyacentes no válidas* si están fuera de los límites del mundo.

Así pues,

- el método *Location.getNeighborhood()* devuelve sólo *posiciones adyacentes válidas*.

- el método *Player.orientate()* puede orientar al jugador hacia cualquier *posición adyacente*. Por tanto la posición absoluta que este método devuelve, en el caso de que el jugador se encuentre en una posición límite, podría estar fuera de los límites del mundo.