Programación 3

Universidad de Alicante, 2019–2020

Práctica 4

Plazo de entrega: **Hasta el domingo 22 de diciembre de 2019 a las 23:59h**.

Peso relativo de está práctica en la nota de prácticas: 25%

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: Persistencia y sistema de puntuación

Módulo de persistencia

Este módulo nos permitirá guardar en disco la partida de BlockWorld en formato Minetest. Esto nos permitirá cargar el mundo BlockWorld en Minetest y visualizarlo. Ten en cuenta que hay algunas diferencias importantes entre los mundos BlockWorld y los de Minetest: éstos últimos tienen tamaño infinito y los tipos de bloques disponibles no son exactamente los mismos que tenemos en BlockWorld. De modo que lo que verás será una especie de mundo BlockWorld expandido, al que Minetest va añadiendo terreno conforme te vas moviendo.

La clase que implementa la persistencia de Minetest es *MinetestSerializer* y un tipo enumerado auxiliar *MinetestContentId*, que ya se dan implementados. Lamentablemente, la interfaz de los objetos que representan los mundos, el jugador y el inventario en Minetest (representados aquí por los interfaces *IWorld*, *IPlayer* e *IInventory*) no es exactamente igual a las clases respectivas de BlockWorld. Para no tener que modificar nuestras clases para introducir la posibilidad de guardar mundos de BlockWorld en formato Minetest, debemos crear una serie de *clases adaptadoras* cuya única función es hacer de puente entre la interfaz de nuestras clases y las de Minetest. Estas clases implementan las interfaces de Minetest delegando en las clases de BlockWorld para obtener la funcionalidad deseada. Además, de esta manera, tanto el sistema de persistencia (*MinetestSerializer*) como las clases adaptadoras están desacopladas y dependen de los interfaces y no al revés (sería un caso particular del *principio de inversión de dependencias*).

Módulo de puntuaciones

Este módulo proporciona un sistema de puntuaciones y ranking basado en algunas de las acciones que realiza el jugador: recoger items, minar bloques, moverse y conservar su nivel de salud y alimento. Hay dos clases principales en este sistema: *Ranking* y *Score*. Esta última representa, de forma abstracta, la puntuación de un jugador identificado por su nombre. Es una clase con un *orden natural* implementado mediante la interfaz *Comparable*, de manera que dados dos objetos de tipo *Score*, se pueda saber cual de ellos es el menor (o sin son iguales). Esto es necesario para que estos objetos se puedan ordenar en la clase *Ranking*, que automáticamente establecerá un ranking basado en puntuación. Los objetos de tipo *Score* puntúan al jugador al invocar al método *score()*.

Ambas clases son genéricas. La idea es que podamos definir una puntuación en base a alguna acción realizada por el jugador definiendo subclases de *Score*. Por ejemplo, la subclase

CollectedItemsScore puntúa al jugador según el item que haya recogido. Para ello, definimos esta clase como una subclase de *Score* que instancia su parámetro de tipo como un *ItemStack*:

```
class CollectedItemsScore extends Score<ItemStack> { ... }
```

Al hacerlo así, el método *score()* recibirá como parámetro un objeto de tipo ItemStack, en base al cual se le asignará una puntuación al jugador. De forma similar haremos con el resto de subclases de *Score*: *MiningScore* puntúa al jugador según el valor del bloque que acaba de minar, mientras que *PlayerMovementScore* le puntúa en base a la distancia recorrida desde su última posición conocida. *XPScore* es un caso especial de puntuación que combina otras puntuaciones (atributo *scores*) sacando la media de éstas y sumándole los valores actuales de salud y alimento del jugador.

Por último, la clase *Ranking* se limita a mantener en un conjunto ordenado las puntuaciones de varios jugadores. Estas puntuaciones serán todas del mismo tipo, p. ej., objetos de tipo *MiningScore*. Cada una de ellas es almacenada en el ranking tras acabar la partida, ya que una vez insertada una puntuación en el conjunto (*SortedSet*) no se debe modificar, ya que el conjunto no se reordena implícitamente en ese caso.

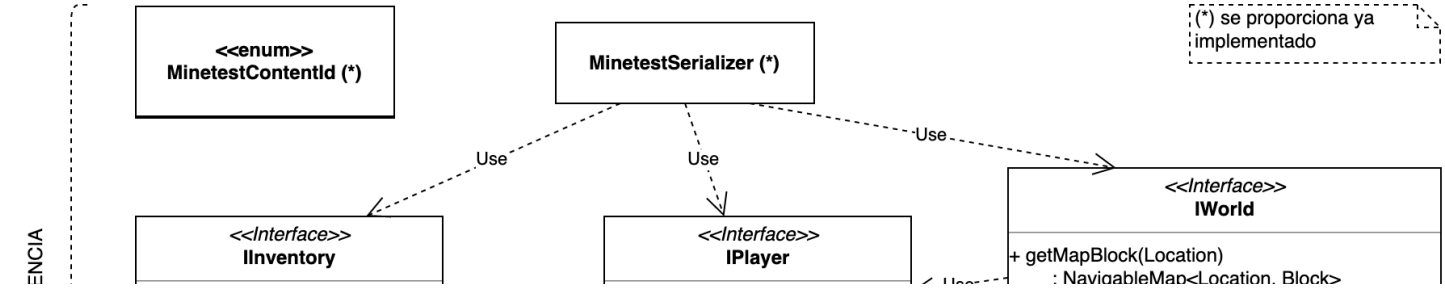
Entrada/Salida

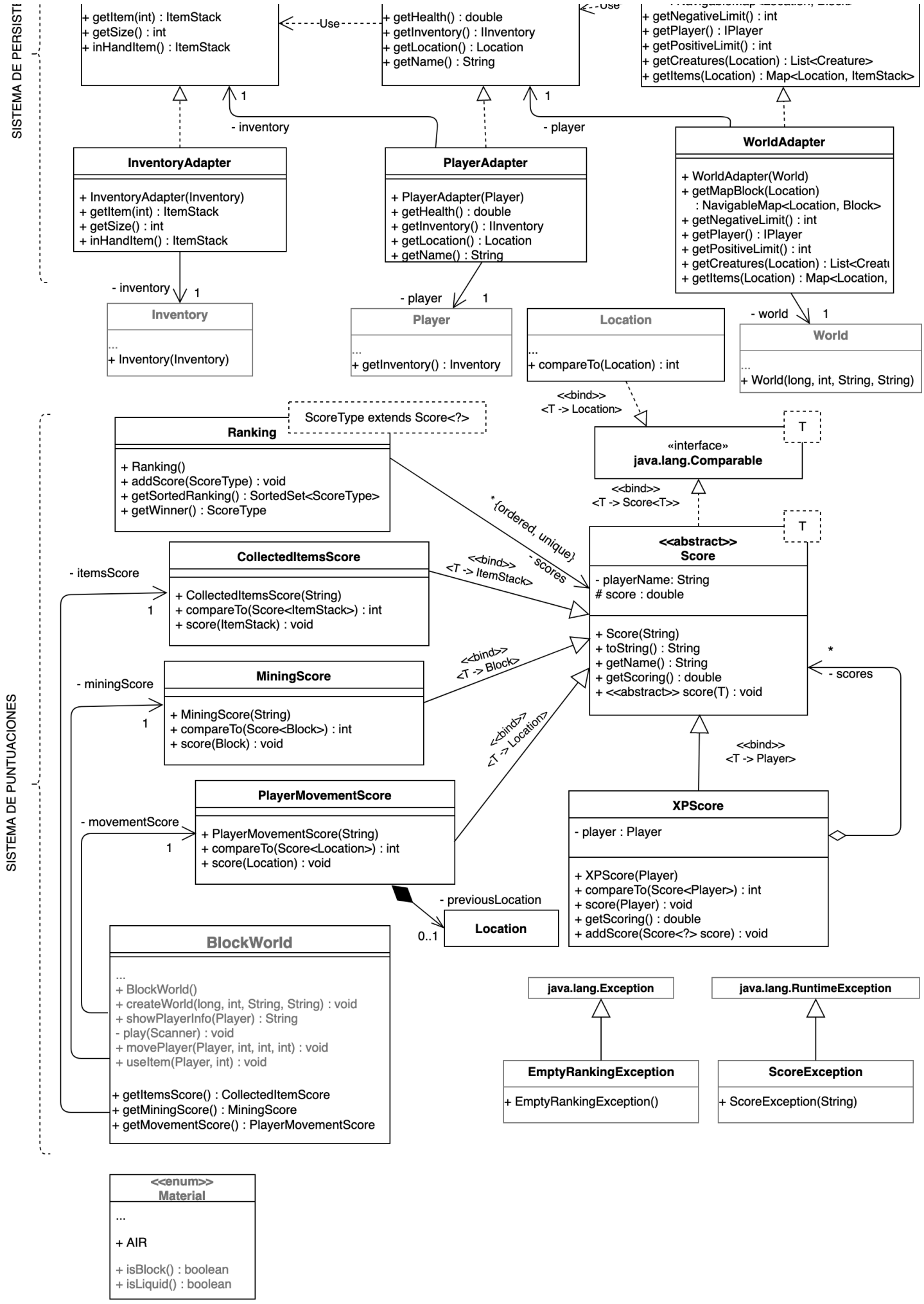
El único cambio a realizar es el formato de la orden de creación de mundos, para incluir el nombre del jugador.

```
<semilla> <tamaño> <nombre del jugador> <nombre del mundo>
```

El formato de los campos `<semilla>`, `<tamaño>` y `<nombre del mundo>` es el mismo que en la práctica anterior. El `<nombre del jugador>` debe ser una única palabra (sin espacios ni tabuladores).

Diagrama de clases




SISTEMA DE PERSISTI
+ getItem(int) : ItemStack
+ getSize() : int
+ inHandItem() : ItemStack
Use
+ getHealth() : double
+ getInventory() : IInventory
+ getLocation() : Location
+ getName() : String
+ getNegativeLimit() : int
+ getPlayer() : IPlayer
+ getPositiveLimit() : int
+ getCreatures(Location) : List<Creature>
+ getItems(Location) : Map<Location, ItemStack>
1
- inventory
1
- player
InventoryAdapter
+ InventoryAdapter(Inventory)
+ getItem(int) : ItemStack
+ getSize() : int
+ inHandItem() : ItemStack
PlayerAdapter
+ PlayerAdapter(Player)
+ getHealth() : double
+ getInventory() : IInventory
+ getLocation() : Location
+ getName() : String
WorldAdapter
+ WorldAdapter(World)
+ getMapBlock(Location)
: NavigableMap<Location, Block>
+ getNegativeLimit() : int
+ getPlayer() : IPlayer
+ getPositiveLimit() : int
+ getCreatures(Location) : List<Creatu
+ getItems(Location) : Map<Location,
- inventory
1
Inventory
...
+ Inventory(Inventory)
- player
1
Player
...
+ getInventory() : Inventory
Location
...
+ compareTo(Location) : int
- world
1
World
...
+ World(long, int, String, String)
<<bind>>
<T -> Location>
ScoreType extends Score<?>
Ranking
+ Ranking()
+ addScore(ScoreType) : void
+ getSortedRanking() : SortedSet<ScoreType>
+ getWinner() : ScoreType
T
«interface»
java.lang.Comparable
<<bind>>
<T -> Score<T>>
T
<<abstract>>
Score
- playerName: String
score : double
+ Score(String)
+ toString() : String
+ getName() : String
+ getScoring() : double
+ <<abstract>> score(T) : void
*{ordered, unique}
- scores
*
- scores
- itemsScore
1
CollectedItemsScore
+ CollectedItemsScore(String)
+ compareTo(Score<ItemStack>) : int
+ score(ItemStack) : void
<<bind>>
<T -> ItemStack>
- miningScore
1
MiningScore
+ MiningScore(String)
+ compareTo(Score<Block>) : int
+ score(Block) : void
<<bind>>
<T -> Block>
<<bind>>
<T -> Location>
<<bind>>
<T -> Player>
- movementScore
1
PlayerMovementScore
+ PlayerMovementScore(String)
+ compareTo(Score<Location>) : int
+ score(Location) : void
XPScore
- player : Player
+ XPScore(Player)
+ compareTo(Score<Player>) : int
+ score(Player) : void
+ getScoring() : double
+ addScore(Score<?> score) : void
- previousLocation
0..1
Location
SISTEMA DE PUNTUACIONES
BlockWorld
...
+ BlockWorld()
+ createWorld(long, int, String, String) : void
+ showPlayerInfo(Player) : String
- play(Scanner) : void
+ movePlayer(Player, int, int, int) : void
+ useItem(Player, int) : void
+ getItemsScore() : CollectedItemScore
+ getMiningScore() : MiningScore
+ getMovementScore() : PlayerMovementScore
java.lang.Exception
java.lang.RuntimeException
EmptyRankingException
+ EmptyRankingException()
ScoreException
+ ScoreException(String)
<<enum>>
Material
...
+ AIR
+ isBlock() : boolean
+ isLiquid() : boolean

En este diagrama de clases UML las clases en gris son de 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.

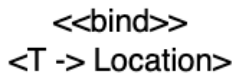
En este diagrama aparecen clases o interfaces genéricas, como *Score*, clases que heredan de éstas, como *MiningScore*, y relaciones con atributos especiales, como la asociación entre *Ranking* y *Score*, cuya representación en UML explicamos a continuación.

Representación de una clase o interfaz genérica en UML

Este diagrama representa una interfaz genérica con un **parámetro de tipo** *T*, que se declararía como

```
interface Comparable<T> {...}
```

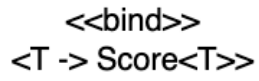
(no debes definirla tú, pertenece a la librería estándar de Java *java.lang*).



Especificación del argumento de tipo al heredar de una clase genérica

Este texto que aparece en el UML representa el **argumento de tipo** que debemos usar al indicar que *Location* implementa el interfaz *Comparable*. En este caso estamos definiendo una clase no genérica que implementa una interfaz genérica:

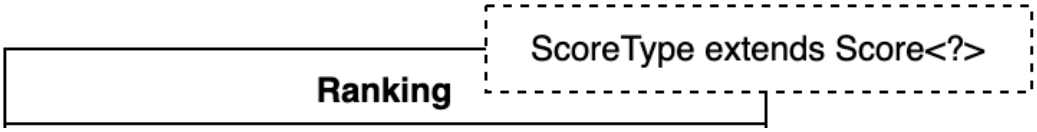
```
class Location implements Comparable<Location> {...}
```



Especificación del argumento de tipo al heredar de una clase genérica

En este caso definimos la clase abstracta genérica *Score* indicando que implementa el interface *Comparable*:

```
abstract class Score<T> implements Comparable<Score<T>> {...}
```



Definición de una clase con genericidad restringida

En esta imagen podemos ver como la clase *Ranking* tiene un parámetro de tipo restringido llamado *ScoreType*, es decir, que debe ser de tipo *Score*:

```
class Ranking<ScoreType extends Score<?>> {...}
```

Fíjate que sería un error indicar `Score<T>`, ya que el identificador `T` no ha sido definido como parámetro de tipo (el parámetro de tipo de *Ranking* es `ScoreType`). Pero como no sabemos de que tipo exactamente será el argumento de tipo de *Score*, indicamos el comodín `?`.

Por último, fíjate en la asociación entre *Ranking* y *Score*. Se trata de una relación *con atributos*, donde la expresión `* {ordered, unique}` nos indica que es una relación de uno a muchos (por el asterisco) y que los objetos de tipo *Score* que guardemos en *Ranking* deben estar ordenados (de ahí que *Score* implemente el interfaz *Comparable*) y deben ser únicos. Es decir, debemos almacenarlos en un conjunto ordenado, representado por la interface *java.util.SortedSet* y su implementación *java.util.TreeSet*.

Clases a modificar

Material

Añade al final de la lista de materiales el material *AIR*, valor=0, símbolo=' ' (espacio en blanco). No lo usaremos explícitamente en BlockWorld, pero sí lo usará el sistema de persistencia. Se considera un material líquido y de bloque.

Location

Esta clase debe implementar el interfaz *Comparable* para establecer un orden natural entre los objetos de tipo *Location*.

compareTo(Location other)

Método de la interfaz *Comparable* que hay que implementar. Dados dos objetos `loc1` y `loc2` de tipo *Location*,

```
loc1 < loc2 : (loc1.x < loc2.x) v (loc1.x = loc2.x ^ loc1.y < loc2.y) v (loc1.x = loc2.x ^ loc1.y
loc1 = loc2 : (loc1.x = loc2.x ^ loc1.y = loc2.y ^ loc1.z = loc2.z)
loc1 > loc2 :  en caso contrario
```

Por tanto, dada la instrucción `loc1.compareTo(loc2)`, el método *compareTo* devolverá un número negativo si `loc1 < loc2`, cero si son iguales o un número positivo si `loc1 > loc2`. Véase la documentación de la interfaz *Comparable* para más información.

Inventory

Implementa el constructor de copia de esta clase.

Player

getInventory()

Devuelve una copia del inventario del jugador.

World

World(long seed, int size, String name, String playerName)

Modifica el constructor sobrecargado de World para añadirle un cuarto parámetro que sea el nombre del jugador. Como el método *generate()* crea un jugador de nombre por defecto “Steve”, vuélvelo a crear tras invocar a *generate()*.

BlockWorld

Esta clase se encargará de actualizar las puntuaciones del jugador por recoger items (*CollectedItmesScore*), minar bloques (*MiningScore*) y moverse por el mundo (*PlayerMovementScore*).

BlockWorld()

Inicializa las puntuaciones a null.

createWorld(long seed, int size, String name, String playerName)

Recibe un nuevo parámetro *playerName* que es el nombre del jugador. Tras crear el mundo crea los objetos que representan las puntuaciones.

showPlayerInfo(Player)

Añade una última línea que muestre las puntuaciones del jugador, como en el siguiente ejemplo:

```
Location{world=test,x=22.0,y=64.0,z=23.0}
Orientation=Location{world=test,x=0.0,y=-1.0,z=1.0}
Health=9.699999999999934
Food level=0.0
Inventory=(inHand=(GRASS,1),[(DIRT,1), (IRON_PICKAXE,1), (WOOD_SWORD,1), (APPLE,2)])
... ... ...
... .P. ...
... ... .g.
Scores: [items: 233.6, blocks: 6.0, movements: 14143.492]
```

play(Scanner)

Modifica este método para que lea el nombre del jugador en la instrucción de creación del mundo (véase la sección sobre Entrada/Salida).

movePlayer(Player, int, int, int)

Modifica este método para que puntúe al jugador por el movimiento realizado y por el item que recoja, si procede.

useItem(Player, int)

Modifica este método para que puntúe al jugador si ha minado un bloque.

getItemsScore(), getMiningScore(), getMovementScore()

Getters. Devuelven el objeto de tipo *Score* correspondiente.

Sistema de persistencia (paquete *model.persistence*)

IInventory

Esta interfaz representa un inventario de Minetest.

getItem(int pos)

Obtiene el item que se encuentra en la posición ‘pos’ del inventario, donde pos=0 representa la primera posición. Si no hay item en esa posición o esta no existe, devuelve ‘null’.

getSize()

Obtiene el tamaño del inventario.

inHandItem()

Obtiene el item del inventario que el jugador lleva en la mano (o ‘null’ si no lleva nada).

IPlayer

Esta interfaz representa un jugador de Minetest.

getHealth()

Obtiene el nivel de salud del jugador.

getInventory()

Obtiene una copia del inventario del jugador.

getLocation()

Obtiene la posición del jugador.

getName()

Obtiene el nombre del jugador.

IWorld

Esta interfaz representa un mundo de Minetest.

getMapBlock(Location loc)

Este método obtiene un mapa ordenado de bloques (*NavigableMap*) indexado por objetos de tipo *Location* que representa un area de 16x16x16 bloques (lo que en la terminología de Minetest se conoce como un *MapBlock*). La posición 'loc' corresponde al bloque situado en la esquina noroeste del nivel inferior de este área.

Todas las posiciones usadas como clave en el mapa de bloques son relativas a la posición 'loc' (ver la sección *Aclaraciones* para más información).

getNegativeLimit()

Obtiene el valor límite negativo de las posiciones de este mundo. Si el mundo tiene tamaño 51, por ejemplo, este método devolverá -25. Si el tamaño es 50, devolverá -24.

getPlayer()

Obtiene el jugador de este mundo.

getPositiveLimit()

Obtiene el valor límite positivo de las posiciones de este mundo. Si el mundo tiene tamaño 50 o 51, por ejemplo, este método devolverá 25.

getCreatures(Location loc)

Obtiene una lista con todas las criaturas que habitan en el mundo y cuya posición queda dentro del área de 16x16x16 bloques cuya esquina noroeste del nivel más inferior corresponde a la posición 'loc'.

getItems(Location loc)

Obtiene un mapa de objetos *ItemStack* indexado por su posición en el mundo, con todos los items cuya posición en el mundo queda dentro del área de 16x16x16 bloques cuya esquina noroeste del nivel más inferior corresponde a la posición 'loc'.

InventoryAdapter

Esta clase implementa los métodos del interfaz *IInventory*, apoyándose en los métodos de la clase *Inventory*.

InventoryAdapter(Inventory)

Constructor sobrecargado a partir de un objeto *Inventory* que queremos adaptar a la interfaz *IInventory*.

PlayerAdapter

Esta clase implementa la interfaz *IPlayer* apoyándose en los métodos de *Player*.

PlayerAdapter(Player p)

Constructor sobrecargado a partir de un objeto *Player* que queremos adaptar a la interfaz *IPlayer*. Construye un *InventoryAdapter* a partir del inventario del jugador 'p'.

WorldAdapter

Esta clase implementa la interfaz *IWorld* apoyándose en los métodos de *World*.

WorldAdapter(World w)

Constructor sobrecargado a partir de un objeto *World* que queremos adaptar a la interfaz *IWorld*. Construye un *PlayerAdapter* a partir del jugador del mundo 'w'.

Nota sobre getMapBlock(Location loc)

En Minetest los mundos son infinitos, es decir, no tienen límites, por lo que las áreas de bloques que este método devuelve siempre corresponden a posiciones que están dentro de los (inexistentes) límites del mundo.

No es el caso de BlockWorld. Como los mundos de BlockWorld están limitados en sus tres dimensiones, algunas de las áreas que este método obtiene contendrán posiciones que estén fuera de los límites del mundo, ya que el área siempre debe tener tamaño 16x16x16. Estas posiciones no tendrán bloque asociado en el mapa (su valor será 'null'). Las posiciones que estén dentro de los límites del mundo BlockWorld pero que no contengan bloques (técnicamente, aquellas que contienen

aire), tendrán como valor asociado un bloque de aire (Material 'AIR'), ya que se asume que en Minetest las posiciones 'vacías' contienen aire. El resto de posiciones válidas tendrán asociado el bloque correspondiente en el mundo BlockWorld.

Nota sobre getCreatures() y getItems()

Ni la lista de criaturas ni el mapa de items que se obtienen invocando a estos métodos deben seguir un orden específico.

Sistema de puntuación (paquete *model.score*)

Score

Es una clase abstracta genérica que implementa el interfaz *Comparable*. El tipo 'T' determinará el tipo de puntuación que se quiere definir.

Score(String)

Constructor sobrecargado a partir del nombre de un jugador. Inicializa la puntuación a cero.

toString()

Devuelve una cadena con el formato

```
<nombre del jugador>:<puntuación>
```

Por ejemplo,

```
Isabel:37.5
```

getName()

Obtiene el nombre del jugador.

getScoring()

Obtiene la puntuación del jugador.

score(T)

Método abstracto que incrementa (o decrementa) la puntuación del jugador en función del objeto que se pasa como argumento. Obviamente, el comportamiento de este método dependerá del tipo de puntuación de que se trate y se implementará en las subclases de *Score*.

Ranking<ScoreType extends Score<?>>

Esta clase genérica mantiene automáticamente un ranking de puntuaciones del tipo indicado por el parámetro de tipo `ScoreType` .

Ranking()

Constructor por defecto. Crea un objeto de tipo *SortedSet* para guardar las puntuaciones ordenadas.

addScore(ScoreType)

Añade una puntuación al ranking.

getSortedRanking()

Obtiene un conjunto ordenado de las puntuaciones (un *SortedSet*).

getWinner()

Obtiene la puntuación ganadora del ranking.

Lanza *EmptyRankingException* si no se han añadido todavía puntuaciones al ranking.

CollectedItemsScore

Representa una puntuación basada en el valor de los items recogidos por un jugador en su deambular por un mundo BlockWorld.

CollectedItemsScore(String)

Constructor sobrecargado a partir del nombre de un jugador.

compareTo(Score)

Las puntuaciones se comparan por su valor. Como queremos implementar un orden descendente de puntuaciones para el ranking, dados dos objetos *CollectedItemsScore*, será menor aquel que represente una puntuación MAYOR y serán iguales si ambos objetos tienen la misma puntuación. El nombre de los jugadores es irrelevante. Esto nos dará un orden de mayor a menor puntuación.

score(ItemStack)

Incrementa la puntuación del jugador en una cantidad igual al valor del material de los items multiplicado por la cantidad de items que contiene el *ItemStack*.

MiningScore

Representa una puntuación basada en el valor del material de los bloques destruídos por un jugador en su deambular por un mundo BlockWorld.

MiningScore(String)

Constructor sobrecargado a partir del nombre de un jugador.

compareTo(Score)

Las puntuaciones se comparan por su valor. Como queremos implementar un orden descendente de puntuaciones para el ranking, dados dos objetos *MiningScore*, será menor aquel que represente una puntuación MAYOR y serán iguales si ambos objetos tienen la misma puntuación. El nombre de los jugadores es irrelevante. Esto nos dará un orden de mayor a menor puntuación.

score(Block)

Incrementa la puntuación del jugador en una cantidad igual al valor del material del bloque minado.

PlayerMovementScore

Representa una puntuación basada en la distancia recorrida por un jugador en su deambular por un mundo BlockWorld. Para poder calcular esta distancia es necesario almacenar como atributo la posición previa del jugador.

PlayerMovementScore(String)

Constructor sobrecargado a partir del nombre de un jugador. Inicializa a ‘null’ la posición previa.

compareTo(Score)

Las puntuaciones se comparan por su valor. En este caso queremos implementar un orden ascendente de puntuaciones, de manera que aquellos jugadores que hayan realizado un menor número de movimientos aparezcan primero en el ranking. Por tanto, dados dos objetos *PlayerMovementScore*, será menor aquel cuya puntuación represente una distancia recorrida menor y serán iguales si ambos objetos tienen la misma puntuación. El nombre de los jugadores es irrelevante.

score(Location loc)

Incrementa la puntuación del jugador en una cantidad igual a la distancia entre la posición previa y la posición 'loc'. Actualiza la posición previa a 'loc'. Si es la primera vez que se invoca al método no habrá posición previa y la puntuación será cero.

XPScore

Esta clase representa una agregación de puntuaciones para un jugador. Su puntuación XPScore será igual a la media de estas puntuaciones más su nivel de salud y su nivel de alimento. Al tratarse de una agregación de puntuaciones, habrá que recalcular la puntuación cada vez que se añada una nueva puntuación o cuando se le pida al objeto que nos dé su puntuación (método *getScoring()*)

XPScore(Player)

Constructor sobrecargado a partir de un jugador. Reserva memoria para una lista de puntuaciones.

compareTo(Score)

Las puntuaciones se comparan por su valor. Como queremos implementar un orden descendente de puntuaciones para el ranking, dados dos objetos *XPScore*, será menor aquel que represente una puntuación MAYOR y serán iguales si ambos objetos tienen la misma puntuación. El nombre de los jugadores es irrelevante. Esto nos dará un orden de mayor a menor puntuación.

score(Player p)

Calcula y almacena la media de las puntuaciones más el nivel de salud y el nivel de alimento del jugador 'p'.

Lanza *ScoreException* (excepción no declarada) si el jugador 'p' no es el mismo con el que se creó esta puntuación.

getScoring()

Recalcula la puntuación y la devuelve.

addScore(Score<?>)

Añade una nueva puntuación y recalcula la puntuación agregada.

EmptyRankingException y ScoreException

Se definen en el paquete `model.exceptions.score`, de forma similar a las excepciones definidas en la práctica 2.

Programa principal

Aquí tienes una par de programas principales:

- Main4_ranking.java : Sirve para probar el sistema de puntuación. Te permite crear partidas aleatorias que el programa jugará automáticamente, mostrando al final los rankings y ganadores.
- Main4_persistencia.java : Permite guardar una partida como un mundo de Minetest.

Código y librerías auxiliares para el sistema de persistencia

Aquí tienes algunas cosas que necesitarás para completar y poder ejecutar tu práctica:

- Minetest serializer 0.1.1: contiene las clases *MinetestSerializer* y *MinetestContentId*. Debes copiarlo al directorio raíz de tu proyecto Eclipse y añadirlo al proyecto (Botón derecho -> Build Path -> Add External Archives…).
- Driver JDBC Sqlite 3: driver JDBC de la base de datos Sqlite 3 que se utiliza en Minetest para guardar los datos de un mundo. Procede de la misma forma que con el archivo anterior.
- Carpeta 'world_data': contiene ficheros de configuración necesarios para generar mundos Minetest. Descomprime el archivo y copia la carpeta `world_data` que contiene al directorio raíz de tu proyecto.

Recuerda refrescar (tecla F5) tu proyecto Eclipse para que 'vea' estos nuevos archivos una vez instalados.

Puedes obtener más información acerca de cómo Minetest almacena mundos en disco en el documento world_format.txt en la página de Github del proyecto.

Para poder cargar los mundos generados con tu práctica en Minetest, deberás modificar el fichero de configuración del juego, llamado `minetest.conf`. Consulta el fichero README.md de Minetest, sección 'Paths' para saber donde encontrarlo según tu sistema operativo. Localiza en el fichero una línea que contenga la palabra `ignore_world_load_errors` o añádela:

```
ignore_world_load_errors = true
```

Esto hará que Minetest no salga de la partida al encontrar errores en el formato del mundo que se está jugando. El código del serializador está aún en fase beta y probablemente lo iremos actualizando.

Abrir un mundo BlockWorld en Minetest

Localiza el directorio donde Minetest almacena los mundos. Normalmente es un subdirectorio llamado `worlds` que puedes encontrar en la misma ruta donde encontraste 'minetest.conf'. Copia la carpeta del mundo creado con tu práctica en esa carpeta.

Minetest, en su configuración básica, tiene dos modos de juego: el modo normal, llamado 'Minetest Game' y el modo 'minimal' o de desarrollo. Puedes cambiar entre ambos en la pantalla del menú de Minetest, usando los iconos de la parte de abajo. Los mundos que tu práctica generará están configurados para ser abiertos en el modo normal. Esto es configurable. Edita el archivo `world.mt` de tu mundo y localiza la línea `gameid = minetest`. Si la cambias por `gameid = minimal`, tu mundo aparecerá en el modo 'minimal' en la pantalla principal de Minetest. Puedes probarlo si tienes problemas para abrir tu mundo en el modo normal.

Los mundos generados a partir de la práctica se pueden abrir con éxito en la versión Minetest 5.2.0-dev-10.11.6 en MacOS y Minetest 5.1.0 en Ubuntu GNU/Linux 18.04 "Bionic Beaver".

Pruebas unitarias

Practica desarrollando tus propias pruebas unitarias. Próximamente publicaremos algunos test previos a la entrega.

Descarga los test de la práctica 3 adaptados a ésta práctica, que tienen en cuenta los cambios introducidos en este enunciado. Copia las carpetas `model` y `files` incluídas en el archivo a 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 instrucciones *fail()* en el código). 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.persistence` : Módulo de persistencia: MinetestContentId, MinetestSerializer, IInventory, IPlayer, IWorld, InventoryAdapter, PlayerAdapter, WorldAdapter
- `model.score` : Módulo de puntuación : Ranking, Score, CollectedItemsScore, MiningScore, PlayerMovementScore, XPScore
- `model.exceptions` : Todas las excepciones
- `model.exceptions.score` : Las excepciones del sistema de puntuación (EmptyRankingScore, ScoreException)
- `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-p4.tgz model
```

Sube este fichero `prog3-p4.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.

XPScore

Presta atención al hecho de que la puntuación XPScore depende de las puntuaciones agregadas, las cuales podrían variar entre el momento en que las agregamos a un XPScore (con 'addScore()') y el momento en que llamamos a 'getScoring()', por lo que la puntuación se debe recalcular siempre que llamamos a este último método.

XPScore.score(Player p)

Ten en cuenta que puede darse el caso de que no existan puntuaciones asociadas, en cuyo caso la media de las puntuaciones será cero y la puntuación será igual a la suma de la salud y nivel de alimento del jugador 'p'.

Score

Había una errata en el diagrama UML: el atributo que almacena el nombre del jugador debe ser privado. (Aparecía como público).

IWorld / WorldAdapter

getMapBlock(Location loc)

Las posiciones mediante las cuales se indexan los bloques en el objeto *NavigableMap* que se devuelve deben ser **relativas** a la posición 'loc'.

Por ejemplo, si realizamos la llamada `getMapBlock(new Location(aWorld, -10, 60, 20))` , obtendremos todos los bloques comprendidos entre la posición (-10,60,20) y la posición (5,75,35), ambas inclusive, es decir, cualquier bloque cuyas coordenadas en el mundo 'aWorld' cumplan

- -10 <= x < 6
- 60 <= y < 76
- 20 <= z < 36

que al usarlas como clave en el *NavigableMap* serán relativas a (-10, 60, 20). Es decir, la posición (-10,60,20) será la (0,0,0) y el resto cumplirán que

- 0 <= x < 16
- 0 <= y < 16
- 0 <= z < 16

getItems(Location loc), getCreatures(Location loc)

Estos métodos, a diferencia de *getMapBlock()*, devuelven posiciones absolutas.

Ejemplo: si realizamos la llamada `getItems(new Location(aWorld, -2, 50, 4))` y en la posición (1,60,8) hay un item, la clave correspondiente a ese item en el mapa que devuelve el método será esa posición (1,60,8).

En el caso de *getCreatures()*, cada criatura tiene ya asociada su posición absoluta.

MinetestSerializer

- Actualizado MinetestSerialzer a la versión 0.1.1. Corrige un error al guardar inventarios vacíos. Descárgalo desde la sección de Librerías.

Test previos

Se han actualizado los test previos. En el método 'setup()' de *PlayerAdapter_PreP4Test.java* el 'player2' debe estar en 'venus', no en 'mercury'.

Hemos detectado dos 2 erratas en dos de los enunciados de los siguientes test:

- *WorldAdapter_PreP4Test.testGetCreatures1()*

Donde dice: `-que hay 3 Animals y 11 Monsters` debe decir,

```
- que hay 3 Animals y 12 Monsters
```

- *WorldAdapter_PreP4Test.testGetCreatures2()*

Donde indica `World75` debe poner `World10`

Puedes descargarlos con las erratas corregidas, pero quizás sea más simple que lo cambies en tu propia versión de los test, ¡sobre todo si has hecho modificaciones en ellos!