

Programación 3

Universidad de Alicante, 2019–2020

-
- [Práctica 2](#)
 - [Block World : Bloques, items y un jugador](#)
 - [Introducción](#)
 - [Bloques](#)
 - [Items y materiales](#)
 - [Jugador](#)
 - [Diagrama de clases](#)
 - [Hoja de ruta](#)
 - [Clases](#)
 - [Location](#)
 - [ItemStack](#)
 - [Material](#)
 - [Block](#)
 - [Inventory](#)
 - [Player](#)
 - [World](#)
 - [BlockWorld](#)
 - [Programa principal, clase *World*](#)
 - [Librería `org.bukkit.util.noise`](#)
 - [Pruebas unitarias](#)
 - [Tips de Java](#)
 - [La clase Math](#)
 - [Excepciones](#)
 - [Constructores](#)

- [Documentación](#)
- [Paquetes y estructura de directorios](#)
- [Entrega](#)
- [Evaluación](#)
- [Aclaraciones](#)

Práctica 2

Plazo de entrega: Hasta el domingo 27 de octubre de 2019 a las 23:59h.

Peso relativo de esta práctica en la nota de prácticas: 35%

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 : Bloques, ítems y un jugador



En esta práctica, partiendo de las clases *Location* y *World* de la práctica anterior, construiremos una primera versión del juego con una funcionalidad que nos permitirá generar mundos con bloques, ítems y un jugador que interactuará con ellos.

Los principales conceptos de programación que trabajaremos son

- Relaciones entre objetos
- Excepciones

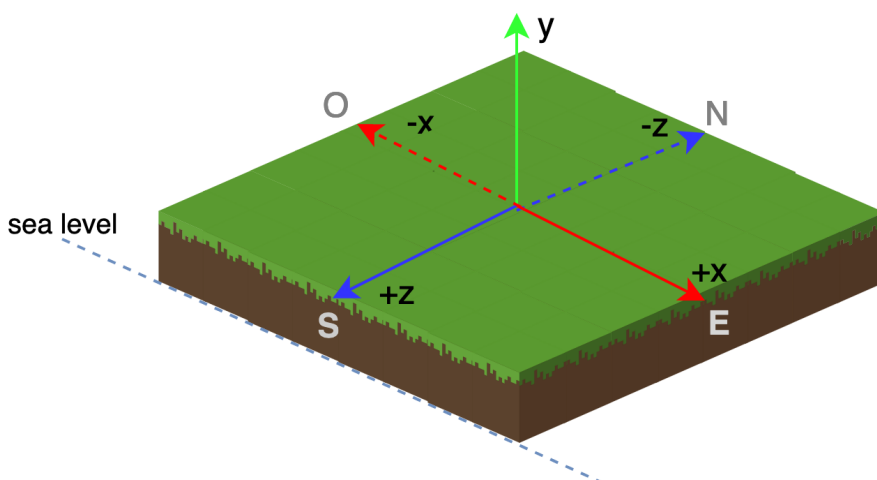
Introducción

Ya en el enunciado anterior comentamos que Block World es un juego formado por un mundo de bloques donde un jugador puede moverse libremente e interactuar con el mundo. Consulta el diagrama de clases que puedes encontrar en la sección [Diagrama de clases](#) mientras lees esta introducción.

Si no conoces Minecraft, el tipo de juego en el que la idea de estas prácticas está basada, te recomendamos que pruebes el juego libre [Minetest](#) para hacerte una idea de en qué consiste un mundo de bloques.

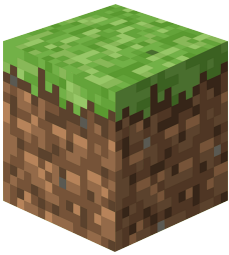
Sistema de coordenadas

Ya explicamos en la práctica anterior cómo funciona el sistema de coordenadas de Block World. Aquí tienes de nuevo la imagen que representa este sistema. Usala como referencia cuándo hablemos de posiciones y movimiento.



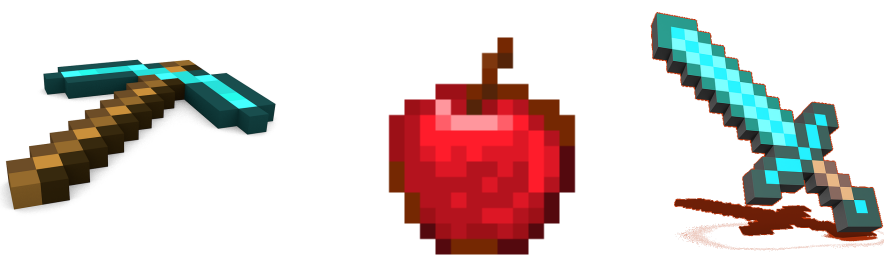
Los mundos de BlockWorld tienen un tamaño determinado, no son infinitos. Ese tamaño es un parámetro que define la longitud del mundo en el eje 'x' y 'z', de forma que podremos definir, por ejemplo, mundos de tamaño 100, que tendrán 100 bloques de este a oeste (eje 'x'), y 100 bloques de norte a sur (eje 'z'). En el eje 'y' todos los mundos tendrán, independientemente de su tamaño, 256 bloques.

Bloques



Cada bloque ocupa una localización en el mundo (representada por la clase *Location*) con coordenadas enteras, de manera que una localización en un determinado mundo sólo puede estar ocupada por un bloque. Los bloques tienen unas dimensiones de 1x1x1, con lo cual ocupan todo el volumen comprendido entre las coordenadas (x,y,z) e (x+1,y+1,z+1). Los bloques pueden ser de distintos materiales, representados por el tipo enumerado *Material*. Por el momento, consideraremos que todos los bloques son sólidos, es decir, no se pueden atravesar.

Items y materiales



Los items están representados por la clase *ItemStack*, que podemos traducir como *pila de items*, y que representa una cierta cantidad de items del mismo material: 3 bloques de granito, 2 espadas de madera, 1 manzana..., serían tres instancias distintas de la clase *ItemStack*.

Los bloques contienen un único item del mismo material del que está hecho el bloque. Por ejemplo, un bloque STONE contiene un *ItemStack* de tipo STONE con un sólo item.

Existen distintos tipos de items en el juego, que al igual que los bloques, están definidos por un material determinado. Además de poder ocupar una posición en el mundo, los items pueden ser recogidos por el jugador cuando éste se mueve a la posición donde están los items y entonces pasan a su inventario.

Existen diferentes tipos de materiales, definidos en el tipo enumerado *Material*, que a su vez corresponden a una de estas categorías: material de bloque, comida, herramienta o arma. Por simplicidad hemos agrupado todas estas categorías bajo la misma clase *Material*. Cada material tiene asociado un valor que, dependiendo de la categoría, tiene un significado distinto. También tienen asociado un símbolo representado por un carácter, que nos servirá a la hora de imprimir por consola cada elemento.

Categoría	Valor
material de bloque	dureza del material
comida	puntos de comida/salud que ganas al comerla
herramienta	dureza de la herramienta
arma	cantidad de daño que produce al atacar con ella

Material	Valor	Símbolo
Bloques		
BEDROCK	-1 (irrompible)	*
CHEST	0.1	C
SAND	0.5	a
DIRT	0.5	d
GRASS	0.6	g

Material	Valor	Símbolo
STONE	1.5	s
GRANITE	1.5	r
OBSIDIAN	5	o
Comida		
WATER_BUCKET	1	W
APPLE	4	A
BREAD	5	B
BEEF	8	F
Herramienta		
IRON_SHOVEL	0.2	>
IRON_PICKAXE	0.5	^
Arma		
WOOD_SWORD	1	i (minúscula)
IRON_SWORD	2	I (mayúscula)

Una posición del mundo puede estar ocupada, por tanto, por un bloque o por un número indeterminado de items (un *ItemStack*).

Jugador



El jugador, representado por la clase *Player*, es capaz de moverse libremente por el mundo, recoger y seleccionar items e ingerir comida. El jugador ocupa una posición en el mundo, pero puede moverse a posiciones adyacentes en cualquier sentido, es decir, desde su posición (x,y,z) a cualquier posición $(x+dx, y+dy, z+dz)$, donde dx, dy, dz son $-1, 0$ o 1 , siempre que esa posición no esté ocupada por un bloque

El jugador es creado (*spawn*) en la posición $x=0, z=0$. La coordenada y dependerá de a qué altura esté el terreno en ese punto.

Salud y alimento

El jugador, además de un nivel de salud inicial máximo de 20 puntos, dispone de un nivel de alimento máximo inicial de 20 puntos. Cada movimiento del jugador resta 0.05 puntos de alimento. Cada acción de usar el item que lleva en la mano, excepto la de ingerir comida, resta 0.1, hasta que el nivel llega a cero. Cuando el nivel de alimento llega a cero, se empieza a perder nivel de salud en la misma proporción.

Cuando el jugador ingiere un item de comida, se incrementa su nivel de alimento en la cantidad correspondiente al tipo de comida. Si el nivel ya está al máximo (20 puntos), entonces se incrementa el nivel de salud en la cantidad de alimento que quede por ingerir. Por ejemplo, si comemos una manzana (4 puntos) y nuestro nivel de alimento es 17, se incrementará éste en tres puntos y nuestro nivel de salud en otro punto (si no está ya al máximo). Ambos niveles pueden tener un valor máximo de 20 puntos. Por tanto, el jugador sólo puede recuperar puntos de salud si su nivel de alimento está al máximo.

Las mecánicas de pérdida y regeneración del nivel de alimento y salud son mucho más complejas en Minecraft. Consulta [alguno de los wikis sobre el juego](#) si quieres saber más sobre ello ([también en español](#), aunque no tan actualizado).

El jugador muere si su nivel de salud es igual o menor que cero. A partir de ese momento ya no puede realizar ninguna acción (moverse, seleccionar items o ingerir comida).

Inventario del jugador

El jugador posee un inventario de capacidad infinita, donde en cada posición del inventario puede guardar una pila de items (*ItemStack*). Una posición especial (llamada 'inHand') corresponde con el item que el jugador lleva en la mano. El jugador se crea con un item en la mano que es una espada de madera (material *WOOD_SWORD*).

La única forma de añadir items al inventario es recogerlos del suelo. Cada vez que el jugador se desplaza a una posición del mundo donde hay items, los recoge automáticamente y se guardan en su inventario. Al guardar estos items (objetos de tipo *ItemStack*) en el inventario, estos ocupan una nueva posición del inventario, independientemente de si anteriormente ya había items del mismo tipo en el inventario o no, lo cual significa que podemos tener manzanas en varias posiciones del inventario, por ejemplo.

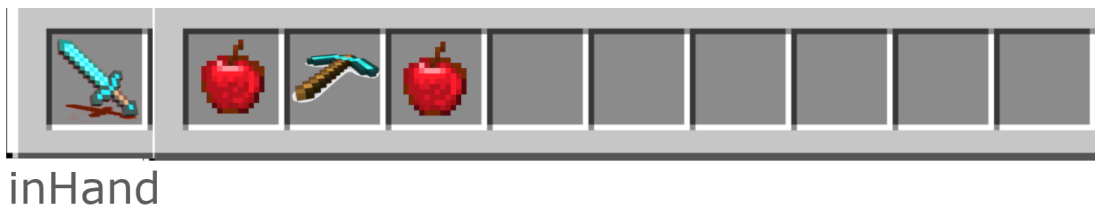
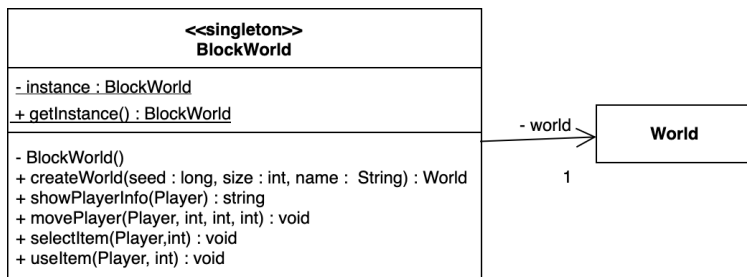
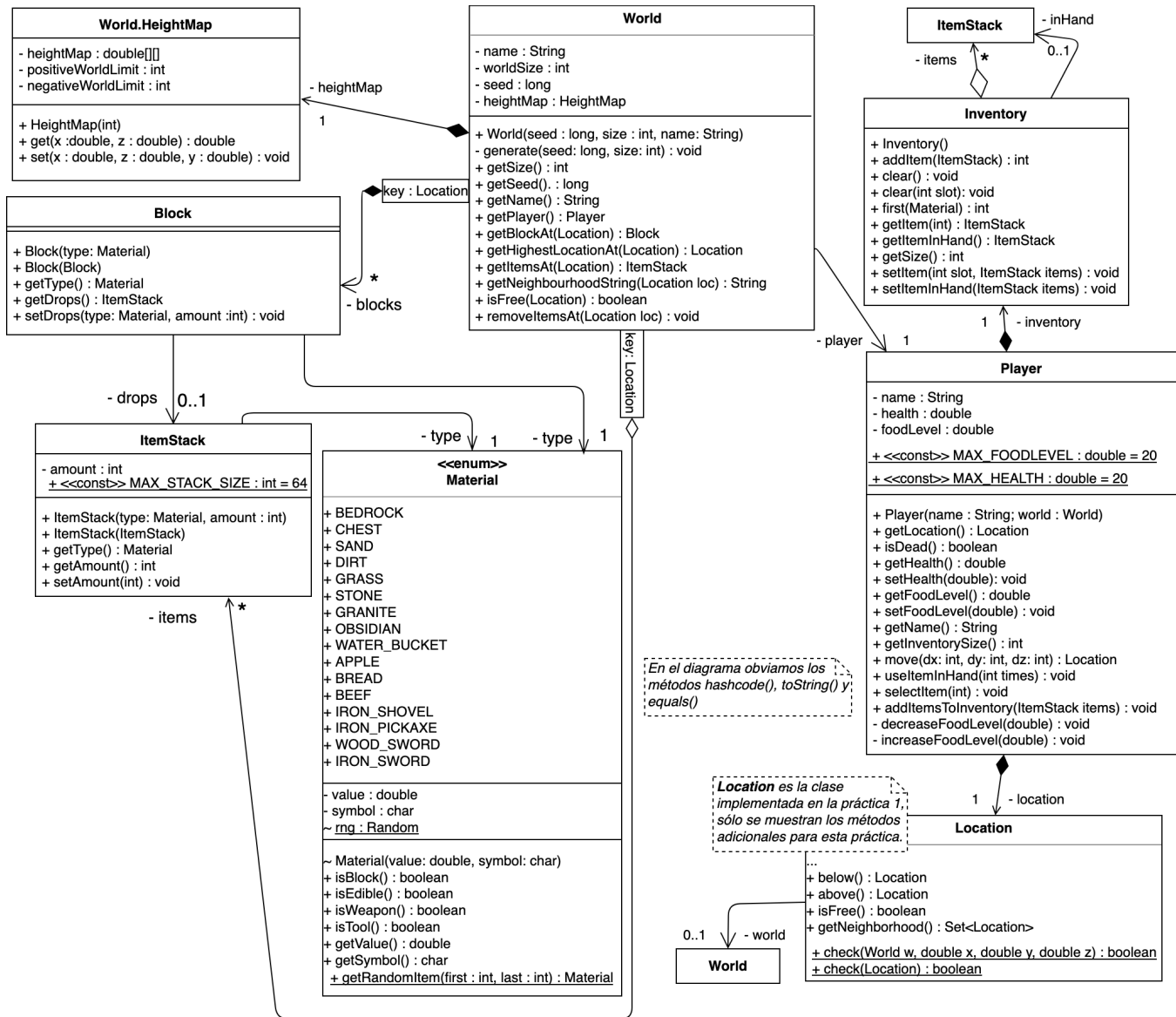


Diagrama de clases

Los diagramas de clases UML a continuación representan las clases de nuestro modelo:



Las nuevas clases pertenecen al paquete *model*, que ya ha sido creado en la práctica anterior.

Las siguientes secciones describen todos los métodos que hay que implementar. Los atributos y las relaciones no se incluyen, porque ya se muestran en el diagrama UML.

Allí donde se indique, tu código debe comprobar que los valores de los argumentos que se pasan a un método son correctos. En el caso concreto de argumentos que son referencias, no es necesario comprobar si la referencia es 'null' a menos que se indique.

Los métodos *equals()* y *hashCode()* deben implementarse para todas las clases del modelo (excepto para tipos enumerados, como *Material*), a menos que se indique lo contrario.

Hoja de ruta

Te aconsejamos que sigas esta pequeña guía de implementación para ir construyendo tu práctica.

Primera parte

Ignora el tratamiento de errores por el momento.

Comienza implementando las clases de más bajo nivel, como *Location*, *Material*, *ItemStack* y *Block*. Realiza las pruebas necesarias para asegurarte de que estas clases funcionan correctamente.

Sigue con las clases *Inventory* y *Player* (no olvides realizar pruebas unitarias de todo lo que se te ocurra).

A continuación, completa la clase *World*.

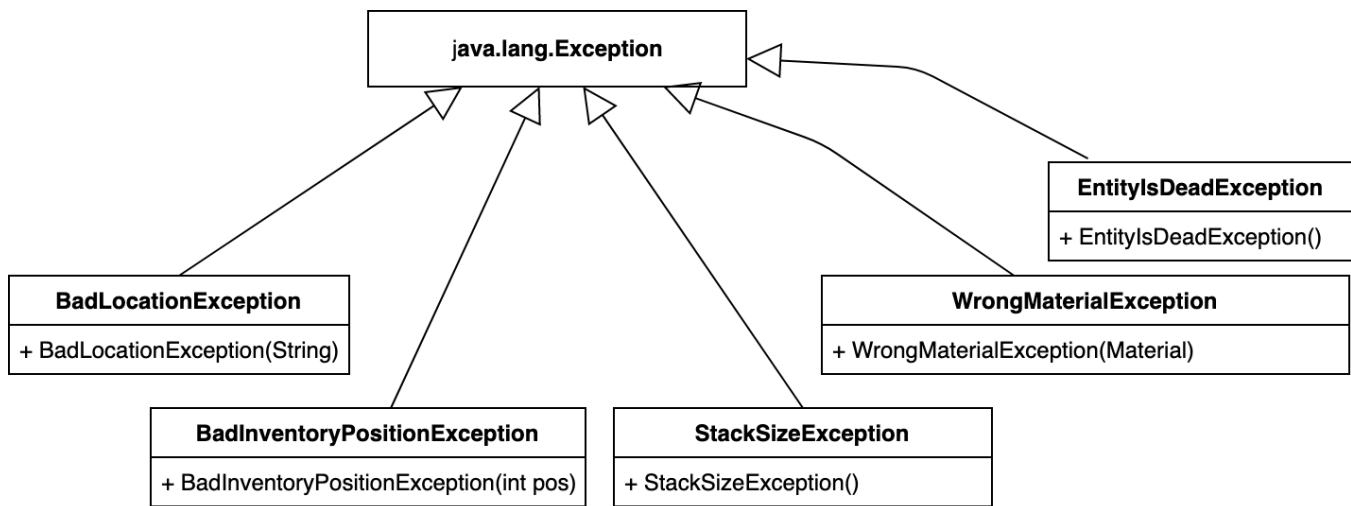
Finalmente, implementa *BlockWorld*

Las pruebas unitarias que puedes diseñar para probar tu código serán de casos que no producen errores.

Segunda parte (excepciones)

Introduce el tratamiento de errores mediante excepciones, siguiendo el mismo orden de clases que en la primera parte. Ten en cuenta que al introducir excepciones en un método, tendrás que modificar el código cliente de ese método (es decir, el código que invoca a ese método) para que realice las acciones adecuadas al recibir la excepción.

Las clases que representan excepciones, en el siguiente diagrama, pertenecen al paquete *model.exceptions*.



Consulta la sección [Tips de Java](#) más abajo para información sobre la implementación de excepciones.

Ahora puedes añadir pruebas unitarias que comprueben las posibles condiciones de error para cada método. Aquí tienes un ejemplo de una prueba unitaria que comprueba si un método lanza una excepción:

```
// En la anotación @Test indicamos la excepción que esperamos que se produzca
// El test fallará si la excepción no se produce.
@Test(expected=BadLocationException.class)
public final void testGetBlockAtException() {
    World e = new World(1, 100, "Earth");
    World m = new World(2, 100, "Mars");

    // esta llamada debe lanzar la excepción BadLocationException porque la posición pasada
    // como argumento pertenece a un mundo distinto.
    e.getBlockAt(new Location(m,0,0,0));
}
```

Clases

A continuación se describen los métodos de cada una de las clases del modelo (excepto las excepciones).

Location

Se trata de la misma clase de la práctica anterior. En ella no comprobábamos si las coordenadas de la posición eran válidas para el mundo dado (excepto la coordenada 'y'). Como ahora los mundos se definen con un tamaño determinado (atributo *worldSize* de *World*), vamos a añadir, entre otros, un método de clase *check()* que nos servirá para comprobar si las coordenadas están dentro de los límites del mundo. Por tanto, debes eliminar las comprobaciones que se hacían en el método *Location.setY()*, que ahora se harán en este nuevo método. Los métodos de instancia de la clase *Location*, por tanto, no realizará ninguna comprobación de rango de las coordenadas, excepto donde se indique en este enunciado.

El código cliente tiene libertad para crear objetos *Location* que representen posiciones fuera de los límites de un mundo (p. ej., la posición (10, -5, 10), que tiene altura negativa). En caso de que necesite comprobar que la posición está dentro de los límites, usará el método *Location.check()*.

Una posición puede no estar asociada a un mundo (su atributo *world* es *null*).

check(World w, double x, double y, double z), check(Location)

Método de clase que comprueba que los valores 'x', 'y', 'z' están dentro de los límites del mundo dado. Por ejemplo, un mundo de tamaño 51 tiene su extremo noroeste a nivel del mar está en la posición (-25,63,-25) y su extremo sureste, también al nivel del mar, está en la posición (25,63,25). Para un mundo de tamaño 50, estos extremos serán (-24,63,-24) y (25,63,25), respectivamente. Por tanto en este segundo caso, el rango para las coordenadas 'x', 'z' está limitado a [-24..25] y el rango para la coordenada 'y' está limitado a [0..255].

isFree()

Comprueba si una posición no está ocupada por un bloque o el jugador. Una posición sin mundo asociado nunca está libre.

Devuelve 'true' si la posición está libre, 'false' en caso contrario.

below()

Devuelve la posición justo debajo de ésta (x, y-1, z).

Lanza la excepción *BadLocationException* si la posición actual pertenece a un mundo y su altura es cero (no existen alturas negativas en los mundos de BlockWorld).

above()

Devuelve la posición justo encima de ésta (x, y+1, z)

Lanza *BadLocationException* si la posición actual pertenece a un mundo y su altura es *Location.UPPER_Y_VALUE*.

getNeighborhood()

Obtiene las posiciones adyacentes a ésta. Si esta posición pertenece a un mundo, devuelve sólo aquellas posiciones adyacentes válidas para ese mundo (es decir, las que estén dentro de sus límites), si no, devuelve todas las posiciones adyacentes. Se consideran posiciones adyacentes todas aquellas que rodean a la posición actual, es decir, dada la posición (x,y,z), cualquier posición cuyas coordenadas tengan un valor +1, 0 o -1 respecto a la posición actual (esta última no se incluye en el vecindario).

ItemStack

Esta clase representa una cierto número de items (*amount*) del mismo tipo (*type*), denominado como *pila de items*. El tamaño máximo de una pila de items es 64.

ItemStack(Material type, int amount)

Constructor. Si el material es herramienta o arma, sólo se puede crear una unidad. Tampoco se pueden crear pilas de items con cero o un número negativo de unidades.

Lanza *StackSizeException* si la cantidad no está entre 1 y MAX_STACK_SIZE. o si el material es de tipo herramienta o arma y 'amount' es distinto de 1.

ItemStack(ItemStack items)

Constructor de copia.

getAmount(), getType()

Getters triviales. Se pueden generar automáticamente con Eclipse.

setAmount(int amount)

Asigna una cantidad de unidades a la pila de items.

Lanza *StackSizeException* si la cantidad no está entre 1 y MAX_STACK_SIZE. o si el material es de tipo herramienta o arma y 'amount' es distinto de 1.

toString()

Devuelve una cadena con el formato `(type,amount)` . Por ejemplo una pila de 3 manzanas se representa como `(APPLE,3)` .

hashCode() y equals(Object)

Genéralos mediante Eclipse.

Material

Tipo enumerado de Java que enumera los distintos materiales que pueden formar parte de los mundos de BlockWorld. Hay cuatro categorías de materiales: bloques, comida, armas y herramientas (ver la sección *Items y materiales* más arriba).

Debes definir cada una de las constantes (BEDROCK, CHEST, etc.) en el mismo orden del diagrama de clases, para que el ordinal asociado a cada constante sea el correcto.

Material(double value, char symbol)

Constructor que asigna un símbolo y un valor a cada material.

isBlock()

Indica si el material es de bloque.

isEdible()

Indica si el material es comida.

isTool()

Indica si el material es una herramienta.

isWeapon()

Indica si el material es un arma.

getValue(), getSymbol()

Getters triviales.

static getRandomItem(int first, int last)

Obtiene un material al azar entre las posiciones 'first' y 'last' del tipo enumerado, ambas incluidas.

Block

Representa un bloque, la unidad de construcción en BlockWorld. Un bloque está formado por un tipo de material para bloques y puede contener en su interior una pila de items.

Block(Material type)

Constructor. Crea un bloque del material especificado.

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

Block(Block)

Copy constructor.

getType() y getDrops()

Getters triviales.

setDrops(Material type, int amount)

Reemplaza los items que contiene un bloque (creando un nuevo `ItemStack`). Cualquier contenido previo se pierde. Los bloques sólo pueden contener un `ItemStack` con un item, excepto si son de tipo CHEST, en cuyo caso se permite que contengan `ItemStacks` con varios items.

Lanza *`StackSizeException`* si la cantidad de items no es correcta: sólo un item para cualquier tipo de bloque excepto los tipo CHEST, o la cantidad de items está fuera de rango para una pila de items.

toString()

Devuelve una cadena con el formato `[type]` , donde 'type' es el tipo de material del bloque.

hashCode() y equals(Object)

Genéralos con Eclipse, pero usando sólo el atributo *type*.

Inventory

Inventory()

Crea un inventario inicialmente vacío.

addItem(ItemStack items)

Añade una pila de items al inventario en un nuevo espacio. De esta forma, pueden existir items del mismo material en distintas posiciones del inventario. Devuelve el número de items añadidos.

clear()

Vacía el inventario, incluyendo el item que se lleva en la mano.

clear(int slot)

Elimina los items del espacio indicado. Esto hace que el tamaño del inventario disminuya en una unidad.

Lanza *`BadInventoryPositionException`* si el espacio indicado no existe.

first(Material type)

Obtiene el índice del primer espacio del inventario que contenga items de ese tipo (cero indica el primer espacio) o -1 si no hay items de este tipo.

getItem(int slot)

Obtiene los items en una posición dada del inventario. Devuelve null si la posición no existe.

getItemInHand()

Obtiene los items que el jugador lleva en la mano (puede devolver 'null' si no lleva nada en la mano).

getSize()

Obtiene el tamaño del inventario, sin incluir el item 'inHand'.

setItem(int pos, ItemStack items)

Guarda los items en la posición dada del inventario. Si en esa posición había otros items, serán reemplazados por éstos.

Lanza *BadInventoryPositionException* si la posición no existe (contando desde cero).

setItemInHand(ItemStack items)

Asigna los items a llevar en la mano. Si el argumento 'items' es null indica que no llevamos nada. Si teníamos algo en la mano previamente, será reemplazado por los nuevos items.

toString()

Obtiene una cadena que representa el contenido del inventario. Por ejemplo, si en la mano el jugador lleva una espada de madera y en el resto del inventario lleva 3 manzanas y un pico de hierro (por ese orden), la cadena resultante será

```
(inHand=(WOOD_SWORD,1),[(APPLE,3), (IRON_PICKAXE,1)])
```

hashCode() y equals(Object)

Genéralos automáticamente usando los atributos 'inHand' e 'items'.

Player

Representa al jugador de BlockWorld. El jugador posee un inventario de items y conoce su propia posición en el mundo. Posee un nivel de salud y de alimento que irá variando durante el desarrollo del juego.

Las constantes MAX_HEALTH y MAX_FOODLEVEL representan los niveles máximos de salud y alimento respectivamente que un jugador puede tener. El nivel de salud puede llegar a ser negativo, pero el nivel de alimento no (su valor mínimo es cero).

Player(String name, World world)

Crea un jugador en la posición (0,*,0) en la superficie del mundo dado (* representa el nivel de la superficie), con una espada de madera en su mano y el resto de su inventario vacío. Inicialmente tiene los niveles de salud y alimento al máximo.

Este constructor no lanza excepciones verificadas.

getName(), getLocation(), getHealth(), getFoodLevel()

Getters triviales.

setHealth(double health)

Establece el nivel de salud, que satura en MAX_HEALTH.

setFoodLevel(double)

Establece el nivel de alimento actual del jugador, que satura en MAX_FOODLEVEL.

isDead()

Comprueba si el jugador tiene un nivel de salud igual o inferior a cero.

move(int dx, int dy, int dz)

Dada (x,y,z) , la posición actual del jugador, lo mueve a la posición $(x+dx,y+dy,z+dz)$. La posición de destino sólo puede ser una posición adyacente a la actual, no debe estar ocupada por un bloque o el propio jugador.

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

Lanza *BadLocationException* si la posición de destino no es adyacente a la posición actual, está ocupada o no es válida.

El jugador no se mueve si el método lanza alguna de estas excepciones. Si consigue moverse, su nivel de alimento/salud se decrementa en 0.05 puntos.

useItemInHand(int times)

Si lo que tenemos en la mano es comida, incrementa el nivel de alimento/salud del jugador (ver la sección *Jugador* más arriba). El jugador ingiere la cantidad de comida especificada por el parámetro 'times' si tiene suficientes items de comida en la mano, si no, ingerirá la cantidad que tenga disponible en la mano. Cada unidad de comida restaura los puntos correspondientes de nivel de alimento o salud, según corresponda (ver la sección Jugador). Por ejemplo, si lleva 3 manzanas en la mano y se come una (times=1), le quedarán dos manzanas en la mano. Si lleva 3 manzanas y quiere comerse cuatro (times=4) sólo podrá comerse las tres que lleva.

Si el jugador no tiene ningún item en la mano, no hace nada. Pero si el item que lleva en la mano no es comida, decrementa el nivel de alimento/salud del jugador $0.1 * \text{times}$ puntos.

Lanza *IllegalArgumentException* si el argumento 'times' es negativo o cero, sin llegar a usar el item.

Lanza *EntityIsDeadException* si el jugador está muerto antes de usar el item.

selectItem(int pos)

Intercambia el item en la mano por el que está en la posición 'pos'. Si no llevamos item en la mano, selecciona el que está en la posición 'pos' del inventario para llevarlo en la mano, eliminándolo de la lista del inventario.

Lanza *BadInventoryPositionException* si la posición seleccionada no existe.

addItemToInventory(ItemStack items)

Añade los items en un nuevo espacio del inventario del jugador.

getInventorySize()

Obtiene el tamaño del inventario, sin contar el item que el jugador lleva en la mano.

decreaseFoodLevel(double d)

Decrementa el nivel de alimento/salud en 'd' unidades de acuerdo a las reglas descritas en la sección *Jugador*.

increaseFoodLevel(double d)

Incrementa el nivel de alimento/salud en 'd' unidades de acuerdo a las reglas descritas en la sección *Jugador*.

toString()

Devuelve una cadena con información del jugador. El formato es el siguiente:

```
Name=<name>
<location>
Health=<health>
Food level=<foodLevel>
Inventory=<inventory>
```

donde los campos entre < y > son la representación en forma de cadena de los correspondientes atributos, con un salto de línea al final. Por ejemplo, la representación del inventario será el resultado de invocar al método *toString()* de la clase *Inventory*.

Aquí tienes un ejemplo de como se vería la cadena obtenida al imprimirla por la salida estándar:

```
Name=Steve
Location{world=Block World,x=4,y=4,z=4}
Health=9.2
```

```
Food level=18.5
```

```
Inventory=(inHand=(WOOD_SWORD,1),[(APPLE,3), (IRON_PICKAXE,1)])
```

hashCode() y equals(Object)

Genéralos automáticamente usando todos los atributos.

World

La mejor estrategia para implementar las relaciones con Block e ItemStack es utilizar mapas, usando como clave la posición en el mundo (Location) y como valor el objeto correspondiente. De esta manera, será muy sencillo averiguar qué bloque o items hay en una posición del mundo. Por ejemplo, implementaremos el atributo *blocks* así:

```
private Map<Location,Block> blocks;
```

y en cualquier momento, dada una posición `loc`, podremos preguntar al mapa que nos devuelva el bloque que está en esa posición:

```
Block bloque = blocks.get(loc);
```

En esta [Guía fácil de Java Collection Framework \(JCF\)](https://www.dlsi.ua.es/asignaturas/prog3/Block_World/p2/Practica_2.html) puedes aprender como se usan los mapas y otras estructuras de datos útiles que usaremos en las prácticas (listas y conjuntos). Luego puedes consultar la [documentación de la API del JCF](#) (paquete *java.util*) para conocer bien como funcionan estas estructuras de datos.

El método *generate()*, que ya se da implementado, es el encargado de generar el terreno, las entidades y el jugador del mundo. Para ello utiliza un mapa de alturas, representado por el atributo *heightMap*, que indica la altura del terreno para cada posición del plano x-z. Este atributo es un objeto de la clase interna *World.HeightMap*, que también se da ya implementada. Así, si (x,z) es un punto de ese plano, donde 'x' y 'z' son dos enteros, `heightMap.get(x,z)` representa la coordenada 'y' de la superficie del mundo en ese punto.

Descarga la [clase World](#), que ya incluye los atributos, la clase interna *HeightMap* y el método *generate()*.

World(long seed, int size, String name)

Crea un mundo de tamaño $size * size$ en el plano (x,z). *seed* es la semilla para el generador de terreno y *name* el nombre del mundo. Invoca al generador de terreno (método *generate()*).

size es un número entero positivo que indica el tamaño del mundo en bloques. Los mundos de *BlockWorld* tiene forma cuadrada, por lo que este tamaño indica la longitud del mundo en el eje x y en el eje z, de manera que el mundo se genera con la localización $x=0, z=0$ en el centro del mundo. El eje 'y' siempre tendrá el máximo tamaño, 256 bloques. Por ejemplo, un tamaño 51 generará un mundo cuyo extremo noroeste a nivel del mar está en la posición (-25,63,-25) y cuyo extremo sureste, también al nivel del mar, está en la posición (25,63,25). Para un tamaño 50, estos extremos serán (-24,63,-24) y (25,63,25), respectivamente.

Lanza la excepción *IllegalArgumentException* si el tamaño indicado no es mayor que cero.

generate(int seed, int size)

(Este método se da ya implementado)

Genera el terreno, los items que encontraremos sobre él y al jugador. Echale un vistazo si quieres ver cómo está implementado. Está basado en el [algoritmo clásico de generación de terreno de Minecraft](#).

getSize(), getSeed(), getName(), getPlayer()

getters triviales.

getBlockAt(Location loc)

Obtiene el bloque que está en la posición indicada, o *null* si no hay ningún bloque ahí.

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

getHighestLocationAt(Location ground)

Obtiene la posición que representa el nivel del suelo en la posición (x,*,z) indicada por el argumento. Por ejemplo, si loc=(10,100,20) y el bloque más alto en esta posición se encuentra a nivel del mar, devolvería la posición (10,63,20).

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

getItemsAt(Location loc)

Obtiene los items que están en la posición dada, o *null* si no había items en esa posición.

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

getNeighbourhoodString(Location loc)

Devuelve una cadena que contiene una representación de las posiciones adyacentes a una dada.

Cada posición se representa mediante un carácter asociado al objeto que ocupe dicha posición, o un punto '.' si no está ocupada. Para bloques, herramientas, comida o armas, se usará el símbolo correspondiente. Para el jugador, se usa la letra 'P'. Para posiciones que están fuera de los límites del mundo, usa la letra 'X' (mayúscula).

Por ejemplo, si queremos representar el vecindario de la posición (x,y,z) en la que se encuentra el jugador, podríamos obtener

```
"aaa aaa aaa\n... .Pa aaa\n... A.B aaa\n"
```

que al imprimirlo por consola quedará como

```
aaa aaa aaa
... .Pa aaa
... A.B aaa
```

donde el primer grupo de 3x3 caracteres de la izquierda representa el nivel y+1 de bloques adyacentes a (x,y,z); el grupo central representa el nivel 'y', y el grupo de la derecha el nivel y-1.

Cada grupo de 3x3 que representa un nivel del eje 'y', tiene el norte arriba y el este a la derecha, por tanto, las posiciones representadas son

```
(x-1,y+1,z-1) (x,y+1,z-1) (x+1,y+1,z-1) <espacio> (x-1,y,z-1) (x,y,z-1) (x+1,y,z-1) <espacio> (x-1
(x-1,y+1,z) (x,y+1,z) (x+1,y+1,z) <espacio> (x-1,y,z) (x,y,z) (x+1,y,z) <espacio> (x-1
(x-1,y+1,z+1) (x,y+1,z+1) (x+1,y+1,z+1) <espacio> (x-1,y,z+1) (x,y,z+1) (x+1,y,z+1) <espacio> (x-1
```



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

isFree(Location)

Comprueba si la posición dada está libre, es decir, no está ocupada por un bloque o por el jugador.

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

removeItemsAt(Location loc)

Elimina los items de la posición dada.

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

toString()

Devuelve una cadena con el nombre del mundo.

Métodos hashCode() y equals()

Estos métodos se pueden generar mediante Eclipse, pero usando únicamente los atributos 'name', 'worldSize' y 'seed'. Es decir, dos mundos se consideran iguales y tienen el mismo código *hash* si se llaman igual, tienen el mismo tamaño y se han generado a partir de la misma semilla.

BlockWorld

La clase *BlockWorld* representa a todo el juego y su funcionalidad básica, y por tanto sólo debe existir una instancia de ella en el programa. Este tipo de clases se conocen como clases *singleton*

(clases con una única instancia).

Para asegurar que sólo existe una única instancia de la clase, el constructor es privado y el método encargado de crear esa instancia y guardar una referencia a ella en el atributo 'instance' la primera vez que es invocado es *getInstance()*.

El resto de métodos de la clase se encargan de implementar la funcionalidad del juego (apoyándose obviamente en el resto de clases del modelo), como crear el mundo, desplazar al jugador, y ejecutar acciones como ingerir comida.

static getInstance()

Obtiene una referencia a la única instancia de esta clase.

BlockWorld()

Constructor por defecto privado. Únicamente inicializa el atributo 'world' a null.

createWorld(long seed, int size, String name)

Invoca al constructor de *World* para crear un nuevo mundo.

showPlayerInfo(Player player)

Obtiene una cadena con información sobre el jugador y lo que hay en las posiciones adyacentes a él, con el siguiente formato de ejemplo:

```
Name=Steve
Location{world=Block World,x=4,y=4,z=4}
Health=9.2
Food level=18.5
Inventory=(inHand=(WOOD_SWORD,1),[(APPLE,3), (IRON_PICKAXE,1)])
aaa aaa aaa
... .PC aaa
... E.B aaa
```

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

Mueve al jugador a la posición adyacente $(x+dx, y+dy, z+dz)$, haciendo que recoja los ítems de esa posición, si los hubiere.

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

selectItem(Player player, int pos)

Invoca al método *Player.selectItem()* con el parámetro 'pos'.

Lanza indirectamente *BadInventoryPositionException*.

useItem(Player p, int times)

Hace que el jugador use el ítem que lleva en la mano 'times' veces invocando al método *Player.useItemInHand()*.

Lanza indirectamente *EntityIsDeadException* e *IllegalArgumentExcepcion*.

Programa principal, clase *World*

Descarga este archivo zip con [clases que contienen un programa principal](#) y descomprímelo en la carpeta 'src/mains' de tu proyecto.

Puedes usar *Main2_partida1.java* como un ejemplo inicial muy sencillo de partida parcial que usa las clases implementadas en esta práctica. Ten presente, en cualquier caso, que este código explora un subconjunto muy reducido de todas las posibles situaciones que se pueden dar en el juego. El archivo [partida1-SALIDA.txt](#) contiene una salida de ejemplo de este programa.

También puedes usar *Main2_RandomWalk.java*, que ejecuta un bucle que hace que el jugador se pasee de forma aleatoria por el mundo, recogiendo los ítems que va encontrando e ingiriendo comida de cuando en cuando. Puedes experimentar con diferentes semillas y tamaños para crear diferentes mundos y dar un paseo por ellos.

Descarga la clase [World](#) con sus atributos y los métodos necesarios para generar mundos aleatorios ya implementados.

Librería [org.bukkit.util.noise](#)

El algoritmo de generación de mundos utiliza una librería de funciones que generan diversas variantes de [ruido Perlin](#). Descarga la librería [org.bukkit.util.noise.jar](#), que debes incluir en tu proyecto como librería externa.

Pruebas unitarias

Aquí tienes, para empezar, los [test de la práctica 1 adaptados a la práctica 2](#), que tienen en cuenta los cambios introducidos en la clase *Location* en lo relativo a la comprobación de coordenadas. Copia la carpeta `model` que se crea al descomprimirlo en la carpeta `test` de tu proyecto. Estos test utilizan el constructor de *World* tal y como estaba definido en la práctica 1, por lo que puedes incluirlo en tu código de esta práctica para comprobar que estos test funcionan bien. La ventaja es que ese constructor no llama al método *World.generate()*, por lo que los test se ejecutarán más rápido. Para que los test funcionen correctamente, es suficiente con que los métodos *Location.check()* comprueben la coordenada 'y' y quitar el código que comprueba esta coordenada del resto de métodos de la clase *Location*.

Tests previos

Aquí puedes descargar los test previos para [Location](#), [Material](#) e [ItemStack](#) y para el [resto de clases](#). Cópialos a una carpeta `model` en el directorio `test` de tu proyecto.

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!

Tips de Java

La clase Math

Esta clase contiene métodos para realizar ciertas operaciones numéricas como raíz cuadrada, logaritmos, etc. Visita la [documentación de java.lang.Math](#) para ver que funciones implementa. Necesitarás algunas de ellas.

Excepciones

AVISO IMPORTANTE: Los métodos sólo deben lanzar las excepciones que se indican para cada uno en el enunciado, de lo contrario es posible que la aplicación no compile con las pruebas unitarias con las que se evaluará. En particular, es importante que un método que no lanza excepciones capture las que lanzan los métodos a los que invoca, de modo que él mismo no declare lanzar ninguna excepción. Por ejemplo, *showPlayerInfo(Player player)* no lanza excepciones, pero evidentemente invoca (o debería invocar) a *World.getNeighbourhoodString(Location)* que sí las lanza. Esto significa que *showPlayer()* debe capturar esas excepciones y tratarlas.

Todas las clases que representan excepciones se implementan de forma similar: deben heredar de la clase `Exception` como indica el diagrama de excepciones:

```
class UnaExcepcion extends Exception {  
    ...  
}
```

y deben implementar un constructor. Los constructores que reciben un objeto `String` como argumento, simplemente invocan al constructor de la clase `Exception` pasándole ese mismo objeto:

```
public UnaExcepcion(String mensaje) {  
    super(mensaje);  
}
```

Los que no reciben ningún argumento le pasarán al constructor de `Exception` una cadena que describa el error que se ha producido. Por ejemplo, “La entidad esta muerta”.

En el caso de *WrongMaterialException*, cuyo constructor recibe un objeto de tipo *Material* como argumento, usará este objeto para construir un mensaje (el `String` que se le pasará al constructor de

Exception) que incluya el nombre del material que ha producido la excepción. En el caso de *BadInventoryPositionException*, genera un mensaje que incluye la posición de inventario no válida.

Situación en la que una función o bien retorna algo o bien lanza una excepción:

```
public int f() {
    int res=0;
    try {
        res = g(); // g() podría lanzar una excepción de tipo Exception
    } catch(Exception ex) {
        System.out.println(ex.getMessage());
    }
    return res;
}
```

Esta función no se puede escribir así:

```
public int f() {
    try {
        return g(); // g() podría lanzar una excepción de tipo Exception
    } catch(Exception ex) {
        System.out.println(ex.getMessage());
    }
}
```

porque el compilador de Java se quejará de que *f()* debe retornar algo (si se produce la excepción, *f()* no retornará nada).

Código cliente que llama a métodos que lanzan excepciones

En ocasiones sabemos que la excepción no se producirá. Sin embargo, como el método declara que la lanza, debemos capturarla y hacer algo con ella. Aquí hay dos estrategias. O bien la ignoramos sin hacer nada en la cláusula 'catch' que la captura, o bien la relanzamos como una *RuntimeException* (excepción no verificada), que indicará un error de programación: `throw new RuntimeException(cause)`.

```
void f() {
    World w = new World(1,100,"Earth");
    Location loc = new Location(w,0,0,0);
    try {
```

```
// este método declara que puede lanzar BadLocationException si 'loc'
// no pertenece al mundo 'w', pero sabemos que esto no sucederá,
// a menos que hayamos cometido un error de programación.
w.getBlockAt(loc);
} catch (BadLocationException ex) {
    // la relanzamos como una RuntimeException
    throw new RuntimeException(ex);
}
}
```

Lo interesante de esta opción es que

- *f()* no tiene que declarar lanzar excepciones (*RuntimeException* es una excepción no verificada)
- Nos permitirá ‘cazar’ posibles errores de programación en tiempo de ejecución, ya que la excepción hará que nuestro programa termine y será capturada por la máquina virtual, que nos avisará de ello.

Constructores

Recuerda que deben inicializar las referencias y estructuras de datos asociadas a las relaciones.

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.

Paquetes y estructura de directorios

Son los mismos que en la práctica anterior. Todas las nuevas clases pertenecen al paquete *model*. Las clases de excepciones pertenecen al paquete *model.exceptions*.

Entrega

La práctica se entrega en el [servidor de prácticas del DLSI](https://www.dlsi.ua.es/asignaturas/prog3/Block_World/p2/Practica_2.html).

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-blockworld-p2.tgz model
```

Sube este fichero `prog3-blockworld-p2.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

Lee ahora la sección de aclaraciones de la práctica anterior y la que describe los requisitos mínimos para que tu práctica sea corregida.

Implementación de *Location*

Algunos de los nuevos métodos de *Location* necesitan ciertas cosas que implementamos en *World*.

- `Location.check()` necesita conocer el tamaño del mundo, por lo que tendrás que implementar el método que lo proporciona en *World* primero.
- `Location.isFree()` puedes dejarlo pendiente hasta que tengas la implementación de `World.isFree()` terminada (pon un comentario `//TODO ...'` en ese método para recordarte que su implementación esta pendiente.

Location.check()

Si la posición no está asociada a ningún mundo, estos métodos siempre devuelven cierto.

Implementación de *Player*

Para implementar el constructor necesitarás implementar primero *World.getHighestLocationAt()*.

Player.useItemInHand()

Siempre que el jugador consuma todos los items que lleva en la mano, ésta quedará vacía (`inHand=null`).

BlockWorld

Aunque los métodos de *BlockWorld* que reciben un *Player* podrían recibir un jugador que no está asociado al mismo mundo que *BlockWorld*, en esta práctica no es necesario comprobar que ambos mundos sean el mismo.

BlockWorld.selectItem() y Player.selectItem()

El argumento 'pos' de ambos métodos hace referencia a una posición del inventario sin contar la posición 'inHand'. La primera es la posición 0 (cero). Por ejemplo, en este inventario:

```
(inHand=(WOOD_SWORD,1),[(IRON_SWORD,1), (BREAD,1), (BEEF,3)])
```

la posición 1 es la que corresponde a `(BREAD,1)` .

Block

En el diagrama UML faltaba el constructor de copia de la clase *Block*. Se ha actualizado.

World

La nueva notación en el diagrama UML de las relaciones de *World* con *Block* e *ItemStack* en el diagrama UML reflejan ahora el hecho de que se implementan mediante mapas cuya clave son objetos de tipo *Location*.

Tests de la práctica 1 adaptados a la práctica 2

Para que compilen y funcionen, tendrás que mantener el constructor de *World* de la práctica 1 (*World(String)*). Déjalo tal y como lo tenías en la práctica 1, no hace falta que lo modifiques (no se usará para nada de ahora en adelante). Puedes anotarlo como un método *obsoleto* (anotación *@Deprecated*):

```
/**
 *
 * ...
 * @deprecated no usar a partir de la práctica 2
 */
@Deprecated
public World(String name) {
    ...
}
```

Definición de método *obsoleto* en la documentación oficial the *javadoc*:

... Un método obsoleto (*@Deprecated*) ya no es importante. De hecho, es tan prescindible que ya no deberías usarlo más, pues ha sido reemplazado por otro y puede dejar de existir en el futuro.

Gestión de las sentencias 'import' en nuestro código

Debido a la ayuda que nos ofrecen los IDE como Eclipse a la hora de escribir código (autocompleción, generación de código,...) a veces podemos inadvertidamente incluir alguna sentencia 'import' no deseada. Antes de entregar nuestro código, debemos asegurarnos de que no

contiene 'imports' que no necesitamos (básicamente, cualquiera que no pertenezca al paquete 'model' o a los paquetes java.lang o java.util).

Por suerte, el propio Eclipse nos permite 'limpiar' nuestro código de 'imports' que no necesitamos. Para cada fichero de código Java, ejecuta la opción de menú 'Source' -> 'Organize imports'. Esto limpiará tu código de sentencias 'imports' que no necesitas.

Generador de números aleatorios en *Material*

El método *Material.getRandomItem()* necesita utilizar un generador de números aleatorios (RNG, por sus siglas en inglés) para obtener el ordinal del tipo enumerado a devolver. Como el método *World.generate()* utiliza *getRandomItem()* para generar los items que podemos encontrar por el mundo, es necesario que todos declaremos el RNG de la misma forma para que los mundos que se generen sean iguales. Esto es especialmente importante a la hora de la corrección de la práctica.

Crea e inicializa el siguiente atributo estático en *Material*:

```
static Random rng = new Random(1L);
```

y utiliza esta implementación de *getRandomItem()*:

```
public static Material getRandomItem(int first, int last) {  
    int i = rng.nextInt(last-first+1)+first;  
    return values()[i];  
}
```

IMPORTANTE: Hazlo exactamente así y no uses otros RNG como *Math.random()* ya que éstos utilizan semillas aleatorias que no podemos controlar.

Errata en el tipo enumerado *Material*

El símbolo de SAND es 'a' (a minúscula) y no 'n', como aparecía antes en el enunciado en castellano. Verás que en los ejemplos de salida y en los test se usa 'a'.

Erratas en los test previos.

En el siguiente test de 'World_PreP2Test.java':

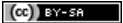
```
@Test
public final void testGetBlockAt() {
    /*
        ... i.. gg.
        ... .P. gg.
        ... F.. g..
    */
}
```

La 'F' debe ser 'A'

Y en *testRemoveItemsAtWorld5x5()* del mismo fichero:

```
...
/* Location{world=World 5x5,x=2.0,y=59.0,z=0.0}
    (BEEF,3)
    */
    fail("Impementa esta prueba de forma similar a la de arriba");
...
```

en vez de BEEF debe indicar WATER_BUCKET.

 © Pedro José Ponce de León, Juan Antonio Pérez; Sánchez Martínez. Universitat d'Alacant. Diseño: HTML5 UP.
[Wikimedia Commons](#).