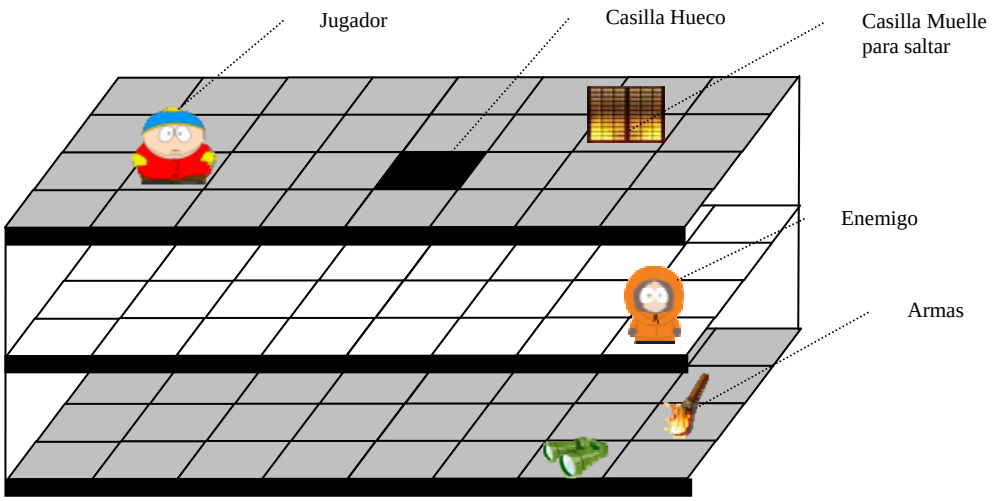
Prácticas de Programación Orientada a Objetos
Convocatoria de Febrero
Curso 2009/2010

Introducción

Este enunciado corresponde a la práctica de la **convocatoria de febrero** del curso 2009/2010. Si la práctica no se finaliza en esta convocatoria, habrá que realizar una nueva práctica para las siguientes convocatorias.

El objetivo de las prácticas de la asignatura es la aplicación de los conceptos del paradigma de programación orientado a objetos al desarrollo de un proyecto de programación.

El proyecto de programación consiste en la implementación de un sencillo **videojuego** en 2D. El juego está formado por un escenario en el que se sitúan elementos (personajes, armas, etc.). El escenario está organizado en plantas y éstas a su vez en casillas. Los elementos se sitúan en las casillas y pueden subir y bajar de planta a través de unas casillas especiales. El objetivo del juego es que el jugador escape del escenario partiendo de la primera planta. A continuación se muestra una representación esquemática del juego con un escenario de tres plantas.



El videojuego será programado en el lenguaje Java utilizando el entorno de desarrollo Eclipse. El aprendizaje del lenguaje Java se limitará a los conceptos fundamentales de la programación orientada a objetos y no al estudio de las bibliotecas del lenguaje. Sólo se estudiará la biblioteca de colecciones por su interés docente. Los aspectos de visualización gráfica y de procesamiento de los ficheros de configuración del juego serán proporcionados en sendas bibliotecas de código por los profesores de la asignatura.

La práctica será realizada en **grupos de dos alumnos** y será desarrollada en dos entregas. Tras cada entrega se realizará una entrevista de revisión y al finalizar todas las entregas se hará una entrevista para defender el trabajo de prácticas.

Entrega de prácticas

Los plazos de entrega de las prácticas son los siguientes:

- **Entrega 1**: 20 de noviembre de 2009.
- **Entrega 2**: 15 de enero de 2010.

Para realizar el **examen parcial de Java** hay que presentar la entrega 1.

Las **entrevistas** de prácticas se realizarán tras la finalización de los plazos de entrega de las prácticas.

La **entrega de prácticas** se realizará en **SUMA**. El **representante del grupo** debe subir el proyecto de programación (sólo el proyecto y no el espacio de trabajo completo de Eclipse) a su *zona de contenidos*. Para ello creará las carpetas "entrega1" y "entrega2" correspondientes a cada entrega. El nombre del proyecto en Eclipse debe ser el DNI del representante del grupo seguido de "`-practica-poo-e1`" o "`-practica-poo-e2`", para las entregas 1 y 2 respectivamente.

Evaluación

La evaluación de la práctica será la media ponderada de las calificaciones de cada entrega. La nota de cada entrega se calcula restando a la puntuación máxima de 10 la penalización por los errores cometidos. Los errores no corregidos de la entrega 1 penalizarán en la entrega 2. Tras la revisión de la entrega 2 habrá que corregir los errores pendientes. Los errores no corregidos volverán a penalizar en la nota de la entrega 2.

Las prácticas que no implementen TODA la funcionalidad requerida no serán evaluadas.

La copia de prácticas, aun siendo parcial, será sancionada con el suspenso de las prácticas tanto para el grupo que copia como para el que deja copiar. Cada grupo es responsable de la custodia de sus prácticas.

Entrega 1

Objetivos formativos

- Conocer y aplicar correctamente el concepto de clase.
- Aplicar el principio de ocultación de la información en la definición de una clase. Definir correctamente la visibilidad de las declaraciones.
- Definir adecuadamente métodos de acceso y modificación para los atributos de una clase.
- Entender el concepto de propiedad calculada. Valorar cuando una propiedad conviene representarla con un atributo o realizar un cálculo para su obtención.
- Comprender la utilidad de los constructores como mecanismo de inicialización de los objetos. Asimismo, aplicar reutilización en la definición de constructores.
- Aplicar la sobrecarga en la definición de métodos como facilidad del lenguaje para definir signaturas coherentes de métodos y reutilizar la implementación entre métodos sobrecargados.
- Definición y uso de enumerados.
- Definición y uso de constantes.
- Conocer la diferencia entre atributos o propiedades de instancia y de clase, y aplicarlos correctamente.
- Comprender la semántica de las referencias en Java.
- Establecer relaciones entre clases mediante relaciones de clientela.
- Utilizar el concepto de herencia como mecanismo de reutilización de código donde se ha de cumplir la relación “es-un” entre las clases.
- Entender la necesidad de la redefinición de métodos en la aplicación de la herencia.
- Comprender el concepto de polimorfismo, ligadura dinámica y visibilidad para la herencia.
- Comprender el concepto de igualdad de objetos frente a identidad.
- Definir correctamente la copia de objetos en Java.
- Valorar la utilidad de la representación textual de un objeto y realizar su implementación.
- Documentar correctamente las clases en JavaDoc.
- Utilizar el concepto de paquete como mecanismo de organización del código.
- Comprender la importancia de la aplicación de una convención de nombrado en la escritura de código. Aplicar la convención de nombres de Java.
- Valorar la utilidad de las pruebas unitarias como mecanismo para asegurar la corrección del código.

Fecha de entrega: 20 de noviembre de 2009.

Valoración de la práctica: 40% de la nota final.

Pruebas unitarias `JUnit`

Junto a cada ejercicio se indicarán las pruebas `JUnit` que deberán implementarse para comprobar la correcta funcionalidad del código. Las pruebas `JUnit` se describen a través de los objetos de prueba que serán inicializados en el método `setUp` y los métodos que han de ser probados. En general, se prestará atención a los métodos que modifiquen el estado de los objetos. Las comprobaciones consisten en determinar si el estado de un objeto (valor de sus propiedades) corresponde con el esperado. Utiliza el método `assertEquals` para comprobar la igualdad de valores y el método `assertSame` para comprobar la identidad de objetos.

Las clases de prueba se situarán en la carpeta de código fuente "test". Las pruebas se nombrarán con el nombre de la clase que están probando acabado en "Test". Además, se situarán en el mismo paquete que las clases que están probando. En general, se seguirán las recomendaciones del Seminario 1.

Control de errores

En el código de esta entrega pueden aparecer casos de error, como por ejemplo, intentar obtener una casilla de una planta indicando una coordenada negativa. Estas situaciones de error serán tratadas en la entrega 2. Por tanto, en esta entrega asumimos que los parámetros de los métodos y constructores van a ser correctos.

Ejercicios

1. Implementa el **enumerado `Direccion`** con los valores ARRIBA, ABAJO, DERECHA y IZQUIERDA. Sitúa el enumerado en el paquete `muelles.escenario`. El enumerado debe proporcionar la siguiente funcionalidad:

   - Método *girarSentidoReloj* que al ser aplicado sobre una dirección retorna la siguiente dirección en el sentido de las agujas del reloj. Por ejemplo, si el objeto enumerado representa ARRIBA, la siguiente dirección según las agujas del reloj será DERECHA.
   - Método *fromString* que retorne una dirección a partir de una cadena en minúsculas. Por ejemplo, si se pasa como parámetro del método la cadena "arriba" devolverá ARRIBA. Si la cadena no corresponde con ninguna de las direcciones, el valor de retorno será nulo.

   Pruebas JUnit.

   Objetos de prueba: un objeto *direccion1* que representa ARRIBA.

   Pruebas:

   - *testGirarSentidoReloj*: definimos una variable local en la que almacenamos el resultado de los giros. Realizamos cuatro giros y comprobamos que cada uno de ellos da el resultado esperado.
   - *testFromString*: aplica el método con las cadenas "arriba", "abajo", "izquierda" y "derecha" y comprueba que el valor es el esperado. Además, prueba con una cadena incorrecta y comprueba que el valor es nulo.

2. Define la **clase `Casilla`** en el paquete `muelles.escenario`. Las casillas representan los componentes básicos de las plantas que forman el escenario del juego. Están situadas en unas coordenadas *x* e *y* sobre una planta (la relación con la clase **`Planta`** se verá en el siguiente ejercicio). Las coordenadas de una casilla no varían una vez que han sido establecidas y sólo pueden tomar valores enteros mayores o iguales que cero. Una casilla puede contener un elemento del juego (jugador, arma, etc.). Asimismo, la casilla puede estar oculta o visible.

   A continuación se indica la **funcionalidad** que debe proporcionar la clase:

   - Un constructor que cree un objeto a partir de sus coordenadas y de su visibilidad inicial.
   - Un segundo constructor que tiene como argumentos sólo las coordenadas y establece por defecto que la casilla no es visible.
   - *getX* y *getY:* métodos de consulta de las coordenadas de la casilla.
   - *isVisible*: permite consultar si la casilla es visible.
   - *mostrar*: hace que la casilla sea visible.
   - *ocultar*: establece la casilla como no visible.
   - *setElemento*: sitúa un elemento dentro de la casilla. Si la casilla contenía un elemento, éste saldrá de la casilla. Implementa una versión sobrecargada de este método que añada un segundo argumento que sea el estado de visibilidad de la casilla (visible/oculta) tras insertar el elemento.
   - *getElemento*: permite obtener el elemento que contiene la casilla.
   - *removeElemento*: saca el elemento, si lo hubiera, de la casilla.
   - *isLlena*: nos informa si la casilla contiene algún elemento.

   Nota: en este ejercicio crearemos la clase `Elemento` vacía y la situaremos en el paquete `muelles.elementos`. De este modo el tipo `Elemento` podrá ser usado para la implementación de la clase `Casilla`.

   Pruebas JUnit.

   Objetos de prueba: un objeto casilla (*casilla1*) construido con el primer constructor y *casilla2* utilizando el segundo, y dos elementos (*elemento1* y *elemento2*).

   Pruebas:

   - *testCasilla*: se comprobará que todas las propiedades de los objetos *casilla1* y *casilla2* tengan el valor esperado (x, y, visibilidad, etc.).
   - *testVisible*: se aplicarán los métodos *mostrar* y *ocultar* sobre el objeto *casilla1*, y se comprobará que el estado de visibilidad cambia adecuadamente tras la aplicación de cada uno de ellos.
   - *testElemento*: se añadirá *elemento1* al objeto *casilla1* y se comprobará que está llena y que el elemento que se almacena es el correcto. A continuación se añadirá *elemento2* y se comprobará que está llena y que el elemento que almacena es *elemento2*. Por último, se aplicará el método *removeElemento* y comprobará que la casilla no tiene elemento y está vacía.

3. Define la clase **`Planta`** en el paquete `muelles.escenario`. Una planta está formada por un array bidimensional de casillas. Se caracteriza por su tamaño en los ejes X e Y, que no varía. Por ejemplo, una planta con un tamaño 4 en X y 4 en Y tendrá 16 casillas, y las coordenadas de las casillas comienzan a numerarse por 0 en los dos ejes. Una planta puede estar conectada a una planta superior y a una planta inferior. La conexión entre plantas permite construir el escenario del juego (entrega 2).

   Entre las clases **`Planta`** y **`Casilla`** se establece una relación de clientela bidireccional. Una planta está formada por varias casillas y una casilla puede pertenecer a una planta o a ninguna. Por tanto, en la clase **`Casilla`** deberemos definir un atributo para almacenar la planta y varios métodos para gestionar la relación de clientela: *getPlanta*, *setPlanta* y *removePlanta*. Además, añade un nuevo constructor en el que se establezca la planta asociada a una casilla así como la visibilidad y su posición *x* e *y*. Téngase en cuenta que la casilla debe introducirse dentro de la planta a la que pertenece.

   La funcionalidad que ofrece la clase **`Planta`** es la siguiente:

   - Un constructor que crea una planta a partir de sus dimensiones en el eje X e Y. Se encargará de construir todas las casillas que conforman la planta. Inicialmente todas las casillas de la planta están ocultas. Por ejemplo, si las dimensiones son 3x3, creará 9 casillas que quedarán dispuestas de forma conveniente en el array bidimensional que define su estructura de datos.
   - *getTamX* y *getTamY*: devuelven el tamaño de la planta en los ejes X e Y, respectivamente.
   - *setCasilla*: establece una casilla en la planta. Este método tendrá como argumento una casilla. La casilla será situada en la planta en la posición que tenga establecida (propiedades x e y del objeto casilla). Nótese que la casilla que estuviera situada en esas coordenadas deja de pertenecer a la planta, es decir, la antigua casilla no debe pertenecer a ninguna planta. Por último, téngase en cuenta que existe una relación bidireccional entre la casilla y la planta. Es necesario que se establezcan correctamente estas relaciones.
   - *getCasilla*: permite consultar la casilla situada en una posición (x, y) de la planta.
   - *getCasillas*: devuelve una lista con todas las casillas de la planta. Las casillas estarán ordenadas en primer lugar por su coordenada X y luego por la coordenada Y. Es decir, primero la (0, 0), luego (0, 1), …,(1, 0), etc.
   - *getCasillasAdyacentes*: retorna una lista con las casillas adyacentes a una posición. Por ejemplo, si la planta tiene las dimensiones 3x3 y consultamos la posición (2, 1), retornará las casillas situadas en (1, 1), (2, 0) y (2, 2). Nótese que este método retornará como máximo 4 casillas y como mínimo dos (casillas situadas en las esquinas).
   - *getCasillaAleatoria*: devuelve una casilla situada en una posición aleatoria de la planta. Consulta la clase `java.util.Random` de la biblioteca estándar de Java para la generación de números aleatorios.
   - *setPlantaSuperior*: establece la planta superior. Téngase en cuenta que el objeto receptor de la llamada debe ser la planta inferior de la planta que se está estableciendo como superior. Es decir, que las relaciones de clientela *planta inferior* y *planta superior* son inversas la una de la otra.
   - *getPlantaSuperior*: permite consultar la planta superior.
   - *setPlantaInferior*: establece la planta inferior. Hay que tener en cuenta las mismas consideraciones que en *setPlantaSuperior*.
   - *getPlantaInferior*: devuelve la planta inferior.

- *subirElemento*: sube a la planta superior el elemento que contiene la casilla situada en una posición de la planta. Este método tiene como argumentos las coordenadas x e y de la casilla que almacena el elemento. Se encarga de eliminar el elemento de la casilla y situarlo en la casilla de la misma posición de la planta superior. Téngase en cuenta que es posible que la casilla no tenga ningún elemento o que no haya planta superior. En el caso en que no haya planta superior el elemento sale de la casilla.
- *bajarElemento*: método análogo al anterior que se encarga de bajar a la planta inferior el elemento que contiene la casilla situada en una posición de la planta.
- *getElementos*: devuelve una lista con todos los elementos que contiene la planta.

Nota: la clase `LinkedList` puede ser utilizada para manejar listas. Se encuentra definida en el paquete `java.util` de la biblioteca estándar. Consulta la documentación `JavaDoc` para su uso.

Pruebas JUnit.

Modifica las pruebas de la clase **`Casilla`** para probar la relación de clientela con la clase **`Planta`**.

Objetos de prueba: añade dos objetos planta (*planta1* y *planta2*) y un nuevo objeto casilla (*casilla3*) que se crea utilizando el nuevo constructor que lo sitúa en el objeto *planta1*.

Pruebas:
- *testCasilla*: añadimos las comprobaciones para el objeto *casilla3* y la comprobación de la planta para todas las casillas.
- *testPlanta*: sobre el objeto *casilla3*, quitamos la planta y comprobamos que no está asociado a ninguna planta. Seguidamente lo asociamos con el objeto *planta2* y comprobamos que la asociación se ha realizado correctamente en ambos sentidos, es decir, *planta2* se ha asociado a *casilla3* y que *casilla3* se encuentra en *planta2* en la posición en la que se ha creado.

Pruebas de la clase **`Planta`**.

Objetos de prueba: tres plantas con dimensiones 3x3 (*planta0*, *planta1* y *planta2*). Establecemos *planta0* como la planta inferior de *planta1* y *planta2* como la planta superior de *planta1*. Por último creamos tres objetos elemento (*elemento1*, *elemento2* y *elemento3*).

Pruebas:

- *testPlanta*: comprobamos que las tres plantas tengan las dimensiones correctas y que están asociadas correctamente a sus plantas superior e inferior. Para cada planta comprobaremos que tiene casillas para todas las posiciones, que las coordenadas y la planta de las casillas son correctas, y que la lista de casillas que devuelve la planta es correcta.
- *testCasilla*: obtenemos una casilla de *planta1* y la almacenamos en una variable local. Creamos una nueva casilla sin planta y la establecemos en la posición que hemos consultado anteriormente (las mismas coordenadas y planta). Comprobamos que la antigua casilla no tiene planta asociada, que la nueva casilla está asociada a *planta1* y que la casilla se encuentra en la posición esperada.
- *testSubirElemento*: establecemos *elemento1* en una casilla de *planta0*. Aplicamos el método para subir un elemento y comprobamos que la casilla de *planta0* ya no contiene el elemento y que el elemento está situado en la casilla de la misma posición de *planta1*. Repetimos la acción de subir *elemento1* en *planta1* y finalmente sobre *planta2*, realizando las comprobaciones tras aplicar cada subida. Por último, comprobamos que si en la casilla no hubiera ningún elemento, la casilla permanece vacía tras aplicar el método.

- *testBajarElemento*: realiza unas pruebas similares a las del método anterior, pero esta vez partiendo de *planta2* y aplicando el método *bajarElemento*.
- *testElementos*: sitúa los tres elementos en distintas posiciones de *planta1*. Aplica el método de consulta de elementos y comprueba que devuelve los elementos esperados.
- *testCasillasAdyacentes*: obtiene las casillas adyacentes de una posición de *planta1* y comprueba que son las esperadas.

4. Define la clase **`CasillaAleatoria`** en el paquete `muelles.escenario`. Una casilla aleatoria es una casilla que se caracteriza por estar siempre visible. Si intentamos situar un elemento en esta casilla, la casilla lo situará en otra casilla elegida aleatoriamente de la planta a la que pertenezca. Si la casilla no tiene planta, entonces hará lo mismo que una casilla normal.

   Pruebas JUnit.

   Objetos de prueba: un objeto planta (*planta1*), un objeto casilla aleatoria situada sobre la *planta1* (*casilla1*), otra casilla aleatoria que no esté en ninguna planta (*casilla2*) y dos elementos (*elemento1* y *elemento2*).

   Pruebas:

   - *testCasillaAleatoria*: comprobamos que las casillas están visibles.
   - *testOcultar*: tras intentar ocultar *casilla1*, comprobamos que se mantiene visible.
   - *testElemento*: comprobamos que *planta1* y *casilla1* no tienen elementos. Establecemos *elemento1* en *casilla1* y comprobamos que la casilla sigue sin tener elementos y que la planta contiene el nuevo elemento. Por último, situamos *elemento2* en *casilla2* y comprobamos que está en la casilla.

5. Define la clase **`Hueco`** en el paquete `muelles.escenario`. Un hueco es una casilla que se caracteriza por bajar de planta los elementos que se sitúan sobre ella. Por tanto, nunca almacenará elementos, aunque el hueco no tenga planta.

   Pruebas JUnit.

   Objetos de prueba: creamos dos objetos planta (*planta0* y *planta1*) y establecemos *planta1* como la planta superior de *planta0*. Creamos dos casillas hueco (*huecoPlanta0* y *huecoPlanta1*) y las situamos en dos posiciones diferentes de *planta0* y *planta1*, respectivamente. Por último, creamos dos elementos (*elementoPlanta0* y *elementoPlanta1*).

   Métodos de prueba:

   - *testSetElemento*: en primer lugar comprobamos que ninguna de las plantas ni de los huecos tiene elementos. Añadimos *elementoPlanta1* sobre *huecoPlanta1*. Comprobamos que ni el hueco ni la planta contienen al elemento y que el elemento está en la *planta0*. Situamos *elementoPlanta0* sobre el *huecoPlanta0*. Comprobamos que el elemento no está en el hueco ni en la planta.

6. Define la clase **`Muelle`** en el paquete `muelles.escenario`. Un muelle es un tipo de casilla que se comporta al revés que un hueco, esto es, sube de planta los elementos que se sitúan sobre él. Sin embargo, añade una característica distinta. Sólo puede impulsar a un número limitado de elementos. En el caso de que se haya sobrepasado el límite, se comportará igual que una casilla normal. Además, debe ser posible consultar si un muelle está operativo, es decir, si puede seguir impulsado elementos. El límite de impulsos de un muelle está limitado a 10 (constante). Finalmente, si la casilla no tiene planta, entonces hará lo mismo que una casilla normal.

   Pruebas JUnit.

   Implementa unas pruebas similares a la clase **`Hueco`**, pero en este caso adaptadas a las características de la clase **`Muelle`**. Añade la prueba *testImpulsos* que sitúe elementos sobre el muelle hasta que deje de impulsarlos.

Métodos `equals`, `clone` y `toString`

Programa los métodos de la clase **`Object`** que se proponen a continuación siguiendo las recomendaciones del Seminario 2.

1. Método `toString` en todas las clases. No es necesario implementarlo en el enumerado.
2. Método `clone` en las clases que representan a las casillas. La copia de una casilla tendrá las mismas coordenadas, el mismo estado de visibilidad, pero no tendrá planta ni elemento. En el caso del muelle, la copia tendrá el mismo número de impulsos.
3. Método `equals` en las clases que representan a las casillas. Dos casillas son iguales si son del mismo tipo, están en la misma posición y tienen el mismo estado de visibilidad. En el caso del muelle, se deberá tener en cuenta los impulsos. En la igualdad se ignora la planta y el elemento.
4. Método `clone` de la clase `Planta`. La copia de la planta tendrá las mismas dimensiones y las casillas serán copias de las originales y estarán situadas en la nueva planta. Además, no estará conectada a las plantas superior e inferior.
5. Método `equals` en la clase `Planta`. Dos plantas son iguales si tienen las mismas dimensiones y las casillas situadas en cada posición son iguales (`equals`). En la igualdad se ignora las plantas superior e inferior.

Pruebas JUnit.

Añade los métodos *testEquals* y *testClone* en las clases de pruebas de las casillas y la clase **`Planta`**. En estos métodos comprueba que la implementación de los métodos `equals` y `clone` se ha realizado correctamente. Por último, es importante comprobar que un objeto es igual (`equals`) a una copia obtenida con el método `clone`.