Examen parcial Prog. III

- Sólo se permite la conexión a internet durante cuarenta minutos. Aprovecha ese tiempo para hacer todas las búsquedas que necesites. **A partir de ese momento debes desconectar wifi/red**.
- En la web de la asignatura ("Entregables y Evaluación") tienes los ficheros con los que se va a trabajar en un fichero comprimido. Crea un proyecto en eclipse y descomprime ese fichero en el proyecto.
- Para entregar el examen, comprime todo el directorio src, y renombra el fichero (.zip o .rar) con tu dni, nombre y apellido. Asegúrate de que tiene los ficheros que has modificado.
- Al acabar el examen, vuelve a conectarte y entrega por ALUD en la tarea correspondiente.

1. Planteamiento

Para este examen, vamos a partir de un minijuego de plataformas parcialmente implementado. Ejecuta el fichero `PlataformasUD.java`. Observa que si pulsas "p" se quita o reactiva la pausa y verás el juego moverse. También puedes pulsar "v" para ver los vectores de velocidad, "c" para parar en los choques, y "+" o "-" para cambiar la velocidad de animación. Con los cursores mueves el personaje (el escudo de UDeusto).

La base del sistema son dos clases ya existentes:

- **`PlataformasUD`**. La clase principal que crea los elementos y los mueve con un bucle principal de juego que refresca el movimiento y la ventana varias veces por segundo.
- **`UDcito`**. Clase que permite crear la instancia del personaje controlado por el usuario. Hereda de una clase abstracta `ObjetoMovil`, de la que también hereda `Pelota` (los enemigos con los que se choca y pierde energía). Cada objeto móvil tiene una coordenada x,y y una velocidad x,y que es la que determina cómo los elementos se van moviendo, afectados por la gravedad. El movimiento y los choques ocurren de acuerdo a unos cálculos físicos que no tendrás que modificar, en la clase `Fisica.java`.

2. Tareas

Se pide que realices las siguientes tareas:

Tarea 1 [3,5]. **JUnit** (*PelotaTest.java – unas 20 líneas de código*). Vamos a comprobar un par de leyes físicas de movimiento con la pelota que tenemos definida. Para ello usaremos la clase *Pelota* y su método `mueveUnPoco` que sigue la ley física para definir cómo se va moviendo una pelota según el paso del tiempo. Para ello codifica los dos métodos de prueba propuestos, haciendo las siguientes cosas *(no hace falta que se vea la ventana gráfica)*:

- `testCaidaPelota`: Debería dar igual que una pelota se mueva 1 vez 100 milisegundos a que se mueva 100 veces un milisegundo. Crea dos pelotas iguales con la misma altura (p.ej. y=0), deja caer (*mueveUnPoco*) a una 100 veces un milisegundo y a la otra 1 vez 100 milisegundos y comprueba que la altura de ambas pelotas sigue siendo la misma (con cierta limitación de precisión). Prueba también que la pelota que se mueve 100 veces cada vez tiene la altura un poco más abajo (coordenada y mayor que la anterior).
- `testSubidaYCaidaPelota`: En un entorno ideal sin rozamiento un cuerpo que se lanza al aire con una velocidad sube y llega cayendo al mismo punto con la velocidad inversa. Prueba a crear una pelota, lanzarla hacia arriba con una velocidad-*y* negativa (p.ej. -20000), y muévela milisegundo a milisegundo hasta que su velocidad se invierta en la caída (>= +20000). Comprueba en el test que en ese momento la diferencia entre la velocidad inicial y esa velocidad es pequeña (p.ej. <=5). Comprueba también que en cada movimiento la velocidad va creciendo en valor, y que los milisegundos que se tarda en subir (los que pasan hasta que la velocidad se hace >= 0) son más o menos los mismos que los que se tarda en bajar. Y que la coordenada y inicial es aproximadamente la misma que la final (con un error de +/-10).

Tarea 2 [3]. Properties (*PlataformasUD.java – unas 15 líneas de código*). Utiliza la clase Properties para guardar la velocidad de juego (la que se puede regular con las teclas + y -) de modo que al empezar otra vez a jugar se utiliza la velocidad que estaba cuando acabamos la vez anterior. La variable interna que almacena esa velocidad es *MILIS_POR_MOVIMIENTO*.

Tarea 3 [4]. **Estructuras de datos y bases de datos** (*PlataformasUD.java y BD.java – unas 30 líneas de código, además del código clásico de base de datos*). Queremos guardar las estadísticas de interacción de los jugadores de nuestro juego. Para eso queremos guardar la información de cuándo se pulsan y sueltan las teclas de cursor (arriba, izquierda y derecha).

En primer lugar tendrás que utilizar un par de estructuras de datos para calcular la información. Observa el método *gestionTeclas* marcado con la tarea 3. Ese método se está llamando en cada iteración de bucle de juego (muchas veces por segundo), recibiendo como parámetro cada una de las tres teclas "UP", "LEFT" y "RIGHT", con la información de si cada una está pulsada (*true*) o no (*false*).
Define un mapa que asocie a cada clave de tecla (los tres strings UP, LEFT, RIGHT) una información booleana que vaya cambiando según las teclas se pulsen o suelten, y actualiza esa información desde el método.
Define además otro mapa que asocie a cada clave de tecla un linkedlist de tiempos (long), de manera que cada vez que se pulsa una tecla que no estaba pulsada se añada el tiempo de pulsación (*System.currentTimeMillis()*), y cada vez que se suelta esa tecla si no estaba soltada se añada el tiempo de suelta (es decir, cada vez que se pulse y suelte una tecla de cursor, deben añadirse a la lista solo dos tiempos, el de inicio de pulsación y el de soltado).
Visualiza por consola la situación de los mapas con el botón "test" para comprobar que se está haciendo bien. Por ejemplo podrías ver algo así en consola:

```
Mapa teclas pulsadas: {LEFT=true, RIGHT=false, UP=true}
Mapa tiempos de pulsación: {LEFT=[1511096427752, 1511096428768, 1511096435553, 1511096439372,
    1511096441740], RIGHT=[1511096426863, 1511096427680, 1511096431863, 1511096435444, 1511096439270,
    1511096441740], UP=[1511096441407, 1511096444792, 1511096445300]}
```

En segundo lugar, guarda esta información en base de datos. Tienes ya la definición de dos tablas en la clase *BD.java*, preparada para el driver sqlite (método *usarCrearTablasBD*). La primera tabla tiene una fila para cada tecla con un contador y ahí debes ir actualizando el número total de veces que se ha pulsado cada tecla. En la segunda tabla cada fila tiene la tecla, información de pulsación o suelta, y tiempo (long). Debes añadir todas las pulsaciones y sueltas, cada una con el valor correspondiente. Lanza la actualización de base de datos desde el botón "Guardar" de la ventana.

Si detectas que te falta algún método en la clase BD.java, añádelo.

. ----------

Tarea 4 [3,5]. **Recursividad** (*PlataformasUD.java – 10-15 líneas de código*). Observa el test 2 (pulsando el botón de reinicio). Verás que al chocar el escudo contra la pelota magenta "se meten" uno dentro de la otra, con lo cual el cálculo de rebote será incorrecto (La superficie de choque de nuestro udcito está representada con el círculo verde que le rodea). Se propone que encuentres el punto del movimiento más cercano en el cual ocurrió el choque.
Para ello tienes planteado un método *calcularChoqueExacto* que deberás programar **recursivamente** para que, por aproximaciones sucesivas, calcule el punto aproximado en el que los círculos chocaron por primera vez.
El planteamiento es aprovechar el método que permite deshacer el último movimiento (*deshazUltimoMovimiento*) y cada vez volver a hacer otro movimiento con menos milisegundos (*mueveUnPoco*). Hazlo recursivamente de modo que se encuentre en 5 llamadas recursivas un punto más preciso de choque. Puedes descomentar el código que está preparado para esta llamada en el método procesaChoque, y te quedará por codificar el método recursivo.

Quien prefiera esta tarea de otra forma más puramente geométrica la puede enfilar de una forma alternativa: Hay dos círculos que se mueven el uno hacia el otro y chocan. ¿En qué punto exacto han chocado? En ese caso el método se podría plantear como

```
private void calcularChoqueExactoAlternativo( VentanaGrafica ventana,
                double c1x1, double c1y1, double c1x2, double c1y2, double radio1,
                double c2x1, double c2y1, double c2x2, double c2y2, double radio2 )
```

Las coordenadas x,y finales (...x1,...y1) las tenemos en cada objeto, y las iniciales (...x2,...y2) las tenemos restando las finales de cada objeto menos las avanzadas en el último movimiento (`obj.getX() - obj.getAvanceX()` y lo mismo con y). Puedes hacer el final de la recursividad cuando los puntos empiezan a estar muy próximos (centésimas o milésimas de píxel).

Puedes consultar los dos vídeos que ilustran el concepto visual de la primera y la segunda por si te ayudan.