UNIVERSIDADE DA CORUÑA

PROGRAMACIÓN ORIENTADA A OBJETOS. CURSO 2008-2009

Boletín de Ejercicios 2

NOTA: La fecha límite para la entrega del boletín es el **19 de Diciembre de 2008**.

1. Escribe una clase Hora que permita representar la hora de un reloj representada a través de horas, minutos, segundos, y el modo (24 horas o 12 horas). Los valores válidos de los minutos y los segundos están en el intervalo [0, 59]. Los valores válidos de la hora están en el intervalo [0, 23] si el reloj está en el modo 24 horas o en el intervalo [1, 12] si el reloj está en el modo 12 horas.

   Los atributos deben ser privados y se deben facilitar métodos de lectura y escritura de los mismos. Se deberá garantizar siempre que los atributos tienen valores válidos (si se intenta cambiar un atributo a un valor no válido el cambio no se llevará a cabo). Si se cambia el modo de 24 a 12 horas o viceversa habrá que cambiar el valor de la hora si es necesario.

   Deben existir tres constructores, uno por defecto que inicialice los valores de la hora a cero, uno al que se le pasan los valores de la hora, los minutos y los segundos por parámetro (si algún valor no es correcto el reloj quedará inicializado a cero) y un tercero que inicializa la hora a partir de otro objeto Hora (constructor de copia).

   Como métodos incluye un método equals: "`public boolean equals (Object o)`" que nos indica si dos horas son la misma. Ten en cuenta que dos horas pueden ser la misma hora pero estar representadas en modos diferentes (24 horas o 12 horas). Incluye también un método toString: "`public String toString()`" que nos devuelve una versión en modo texto de la hora, por ejemplo: 05:00:00 pm (modo 12 horas) ó 17:00:00 (modo 24 horas).

   Tema: Igualdad de objetos, Encapsulación de atributos

2. Crea una clase que represente el funcionamiento de un despertador utilizando la clase Hora del ejercicio anterior. Un despertador consta de dos horas internas: una que representa la hora actual y otra que representa la alarma del despertador. El despertador debe poder realizar las siguientes funciones: (1) devolver la hora actual y la hora de la alarma del despertador (como la clase Hora es mutable el despertador debería devolver copias de las mismas para evitar su modificación externa), (2) devolver un objeto hora que señale el tiempo que falta hasta que suene el despertador, (3) incluir un método `equals` que diga si dos despertadores están en el mismo estado (sus horas de reloj y de despertador son las mismas, acuérdate de que deberás usar el método `equals` de Hora para comparar horas) (4) un constructor de copia que permita crear un nuevo despertador a partir de otro ya creado, deberá ser una copia profunda y deberá hacer uso de los constructores de copia de la clase Hora; (5) una propiedad que permita activar o desactivar la alarma y (6) constructores variados que incluyan las opciones de creación que creáis más razonables. El despertador no tendrá que actualizar la hora del reloj a medida que pasa el tiempo.

   Tema: Composición de objetos, Copia e igualdad profunda de objetos

3. Define una clase `Cuenta` que representa a una cuenta bancaria. Esta cuenta debe almacenar la siguiente información: (1) Propietario de la cuenta, (2) Saldo (podemos suponer que en cantidades enteras, sin decimales, de euros) y (3) Movimientos. El propietario de la cuenta es otro objeto en el que se almacena el nombre del propietario, su DNI y su dirección. Los movimientos de la cuenta serán un array de objetos de tipo `Movimiento`. El movimiento indicará la fecha, la cantidad involucrada en el mismo, el tipo de movimiento (cargo o abono) y el saldo final después de su ejecución. Para simplificar el problema podemos definir inicialmente el array con un tamaño elevado (por ejemplo, 100 posiciones) y suponer que nunca se va a llegar al límite. Los movimientos en el array se almacenan según el orden en el que se producen (por lo tanto las fechas de los mismos deberán ser ascendentes).

   Incluye en la clase Cuenta los siguientes métodos: (1) `getSaldo`, para obtener el saldo de la misma; (2) `void reintegro (Date fecha, int k)` e `ingreso (Date fecha, int k)`, para retirar o ingresar dinero en la cuenta, modificará la propiedad `Saldo` y añadirá un nuevo movimiento al array de movimientos. Si la fecha es anterior que la del último movimiento de la cuenta se deberá lanzar una excepción `IllegalArgumentException` e impedir la modificación de la cuenta. (3) `ultimosMovimientos`, imprime como máximo los 10 últimos movimientos de la cuenta con un formato similar al siguiente.

```
Fecha        Saldo      Movimiento    Nuevo Saldo
10/02/2008   1.000 €     -500 €         500 €
11/02/2008     500 €      200 €         700 €
13/02/2008     700 €    1.000 €       1.700 €
15/02/2008   1.700 €     -700 €       1.000 €
```

   Tema: Composición de objetos, uso de excepciones, formateo de strings

4. Vamos a definir el sistema de vigilancia de una casa con las siguientes estancias: entrada, cocina, salón, despacho, escaleras de bajada, sótano, escaleras de subida, habitación principal, habitación secundaria, habitación de invitados, escaleras al ático y ático. Como las estancias de una casa no es algo que varíe habitualmente vamos a definirlas como un tipo enumerado, en el que ademas de definir los elementos del enumerado incluyamos los siguientes métodos: `toString` (que permite mostrar el nombre de la estancia de una forma mas amigable que si mostráramos el nombre del enumerado), `metrosCuadrados` (devuelve el número de metros cuadrados de cada estancia). Incluir el el enumerado los atributos que creáis oportunos para dar esta funcionalidad.

   El siguiente paso será definir el sistema de vigilancia, para ello haremos uso de la clase `java.util.EnumSet`. Un `EnumSet` podría verse como un array en el que el índice son los enumerados y el contenido es una variable booleana. Crearemos entonces una clase Vigilancia que almacena un atributo `estanciasActivas` de tipo `EnumSet` y que almacenará el conjunto de estancias para las cuales se mantiene una vigilancia activa. Incluiremos también métodos para activar y desactivar estancias (es decir, introducirlas o extraerlas del conjunto de instancias activas). Por último incluiremos los siguientes métodos: `informeEstanciasActivas` (devuelve un `String` que muestra todas las instancias para las que hay activada una vigilancia activa) y `metrosCuadradosVigilados` (devuelve un entero que representa la suma de metros cuadrados de las estancias que están siendo vigiladas). Deberéis aseguraros de que no se lanza el warning "unchecked" al usar genericidad, porque sino el ejercicio no sería considerado correcto.

   Tema: Enumerados, clase `EnumSet`, manejo de Strings
   De Utilidad: Cuando en un método se nos pide pasar la clase de elemento (por ejemplo, el método `allOf` de `EnumSet`) se utilizará la sintaxis `MiClase.class`.

5. Crea una clase que representa el funcionamiento básico de una matriz de enteros. La matriz podrá crearse indicando el número de filas y de columnas (sus valores se inicializarán a cero) o a partir de un array bidimensional que se pase por parámetro. Una vez creado el objeto matriz se podrán llevar a cabo las siguientes operaciones: (1) acceder y modificar los valores concretos de una posición de una matriz, (2) sumar dos matrices (3) multiplicar dos matrices, (4) devolver la matriz como un array bidimensional (5) devolver una fila o columna de la matriz como un array unidimensional.

   Tema: Clases, Encapsulación de atributos, Arrays

6. En este ejercicio partiremos de la matriz realizada en el punto anterior (realmente no es necesario que hagáis todo el ejercicio anterior para resolver este, simplemente hay que tener realizado el funcionamiento básico de la matriz y el apartado 1 de métodos de acceso y modificación). El ejercicio consiste en realizar dos iteradores, el primero recorre la matriz por filas y luego por columnas y el segundo recorre por columnas y luego por filas. Las clases que implementen los iteradores deben implementar el interfaz `Iterator` (no es necesario implementar el método `remove` y se debe lanzar una excepción si se intenta usar) y es necesario crear dos métodos (`iteradorFilasColumnas` e `iteradorColumnasFilas`) en la matriz que devuelvan los iteradores que hemos creado para que sean accesibles a otras clases. Ten en cuenta que la forma más cómoda de implementar un iterador es como una clase interna de la propia clase Matriz.

   Crear también el interfaz `Matrizador` que incluya un método `int get(int fila, int columna)` que devuelve el elemento de la posición indicada y hacer que la Matriz cumpla ese interfaz. De esta manera vemos como los interfaces pueden utilizarse para hacer que clases antiguas (legacy) implementen nuevos interfaces y se incluyan más fácilmente dentro del código moderno (un buen ejemplo es la clase `java.util.Vector` que implementa los interfaces del nuevo API de colecciones).

   Tema: Iteradores, interfaces, clases internas.

EJERCICIO EXTRA: Se contará como un ejercicio extra el hecho de que consigáis que vuestros iteradores sean "fail-fast", esto es, que si en medio de la iteración se altera el estado de la matriz entonces se deberá lanzar la excepción `IllegalStateException`. Para que este ejercicio extra pueda contabilizarse es necesario que la prueba JUnit incluya la prueba del iterador fail-fast comprobando como se lanza la excepción en el momento adecuado e incluyendo los `println` que comenten y resalten suficientemente lo que está ocurriendo.

7. Necesitamos implementar las clases que representen figuras geométricas, tanto 2D como 3D. Deberá haber una clase `Figura` que actúe como superclase de todas las figuras. Las figuras deberán tener un punto (x, y) que represente su punto interno y que permite situar a las figuras en un espacio euclídeo, también tendrán otras propiedades como radio, ancho, alto, etc. (evidentemente no todas las figuras tendrán todas las propiedades). Además las figuras tendrán otros métodos como área, perímetro, volumen, etc. (también no todas las figuras tendrán todos los métodos). Crea una estructura de clases abstractas o interfaces que intente capturar las características comunes de las figuras en distintas clases/interfaces. Es decisión vuestra elegir el mejor esquema de estructuración de clases y deberéis crear un número significativo de figuras: cuadrado, circulo, esfera, cubo, cilindro, etc.

   Crea una clase `UtilidadesFigura` que incluya varios métodos estáticos para operar con las figuras, por ejemplo habrá un método que dado un array de Figuras calcule el área media de las

figuras ahí almacenadas (y lo mismo para el volumen). El array tiene que ser de la clase base `Figura`. No todas las figuras tiene un área o un volumen, por lo cual las figuras para las que no son aplicables esos métodos pero estén en el array no se tendrán en cuenta. Hay que elegir la mejor estrategia para resolver el problema: maximizar el interfaz de `Figura` para que todas las figuras tengan volumen (por ejemplo) o usar `instanceof` y conversiones explícitas de tipos.

Para que el ejercicio sea considerado correcto es necesario que el array genérico de figuras sea recorrido usando un bucle foreach y también es necesario usar la importación estática para acceder al número PI de la clase `Math`.

Tema: Clases abstractas, Interfaces, jerarquías de herencia

8. El objetivo de este ejercicio es crear la estructura básica de un juego de cartas en base clases abstractas e interfaces. De esta forma tendremos un esqueleto o *framework* sobre el que luego crear de forma sencilla juegos de cartas que tengan características similares. Es responsabilidad vuestra definir la estructura de las clases pero deberán cumplir una serie de condiciones: clases como `Baraja`, `Carta` o `Juego` deberán ser clases abstractas o interfaces que después podrán implementarse con clases concretas como `BarajaEspañola`, `CartaEspañola` o `JuegoCartaMasAlta`. El objetivo final es crear un juego sencillo que demuestre que la estructura de clases funciona adecuadamente. Así crearemos el juego Carta Más Alta que enfrenta a dos jugadores, se reparten todas las cartas, cada jugador saca una carta aleatoria de su mazo y gana el jugador que ha sacado la carta más alta. El jugador que haya ganado más manos gana la partida cuando se acaban las cartas.

   Tema: Clases abstractas, Interfaces, jerarquías de herencia

   De Utilidad: Como probar que el juego de cartas funciona correctamente se acerca más a un test funcional que a un test de unidad, los tests de este ejercicio deberán probar que los elementos individuales funcionan correctamente (Mazo, Carta, etc.). El tests de la clase del juego simplemente tendrá que mostrar por consola una serie de mensajes que permitan comprobar que el juego discurre correctamente, pero no hará falta incluir sentencias de JUnit. De todas formas los mensajes deberán mostrarse dentro de un método @Test de JUnit para que aparezcan cuando hacemos el resto de pruebas.