UNIVERSIDADE DA CORUÑA

PROGRAMACIÓN ORIENTADA A OBJETOS. CURSO 2007-2008

Boletín de Ejercicios 2

1. Crea una clase denominada `LigaEstupenda` que se utilice para implementar un juego de la liga de Fútbol al estilo del que publican los diarios deportivos. La liga se basa en una clase `Jugador` que no es más que un enumerado que tiene la siguiente información sobre los jugadores: nombre, equipo en el que juegan, posición, valor aproximado en millones de ¬ y nacionalidad. A continuación se muestra una tabla de ejemplo:

Nombre	Equipo	Posición	Valor (en mill. €)	Nacionalidad
Agüita	Deportivo	Portero	9	Extracomunitario
Marchoso	Valencia	Medio	12	Español
Verdulero	Deportivo	Medio	9	Español
Feinho	Barcelona	Delantero	45	Comunitario
Raulinho	Real Madrid	Delantero	45	Español

   Crea en la clase `Jugador` suficientes entradas para poder jugar a la liga (al menos unas 30), no es necesario que pongas datos de jugadores reales. A continuación crea en la clase `LigaEstupenda` los siguientes métodos: (1) `introducir(Jugador)`, permite introducir un jugador en tu equipo de la liga (hasta un máximo de 11) (2) `eliminar(Jugador)` elimina dicho jugador de tu equipo, (3) `imprimirEquipo`, imprime los jugadores que componen el equipo y finalmente (4) `comprobarEquipo`, comprueba que el equipo cumple con las reglas de la liga, en caso de no cumplirlas mostrará por pantalla qué regla incumple:
   - Sólo puede haber un portero, cuatro defensas, cuatro medios y dos delanteros.
   - No puede haber más de 3 extracomunitarios en el equipo
   - El valor del equipo no puede superar los 200 millones de euros

   Tema: Enumerados

2. Adapta la clase `Circulo` definida en el ejercicio 8 del boletín anterior de forma que ahora, las coordenadas `X` e `Y` se representen a través de un objeto `Punto`. El objeto `Punto` no es más que una clase que almacena las coordenadas `X` e `Y` junto con sus métodos de lectura y escritura. Incluye en `Circulo` un método `getPunto` que devuelva un copia del punto interno de `Circulo` (ya que es un objeto mutable), para ello deberías incluir un constructor de copia en `Punto`. Crea también una clase `PuntoInmutable` que es igual a la clase `Punto` pero que no tiene métodos de escritura, por lo que el punto no puede modificarse una vez que se ha creado. Crea otra clase denominada `Circulo2` que hace uso de los puntos inmutables y, por lo tanto, no tiene necesidad de devolver copias de los puntos en el método `getPunto`. En la prueba del ejercicio muestra como el uso de copias en el método `getPunto` evita los efectos laterales que pueden ocurrir cuando se comparten instancias de objetos mutables.

   Tema: Composición de objetos, objetos inmutables.

3. El objetivo es implementar las clases básicas para utilizar en un supermercado. Necesitaremos por tanto almacenar información de `Clientes`, `Dependientes` y `Reponedores`. De todos ellos necesitamos saber su nombre, apellidos, DNI, dirección y teléfono. De los clientes también nos interesa su código de cliente y el número de compras realizadas, esta última característica es necesaria para obtener el descuento que se le debe aplicar al cliente (cada 100 compras se le aplica un 1% de descuento), debemos por lo tanto implementar un método `compra` que anote que realizó una compra. De los dependientes y reponedores debemos saber su número de la seguridad social, su salario y el turno al que pertenecen (mañana, tarde o noche). Para obtener el salario se tendrá en cuenta que si trabaja en turno de noche tiene un extra de 150€. De los dependientes almacenaremos su especialidad (carnicería, frutería, caja, etc.) y de los reponedores almacenaremos información de la compañía de la que proceden. Todos los atributos se definirán privados y deberán disponer de métodos de lectura y escritura. Se deberá incluir un método `toString` que sobrescriba el método `toString` de `Object` y que permita imprimir toda la información de cada clase. Emplea clases abstractas para generalizar características comunes entre distintas subclases.

   Tema: Encapsulación de atributos y clases abstractas.

4. Implementar el siguiente problema haciendo uso de las anotaciones. Se trata de crear una clase `Marcador` con dos atributos: `local` y `visitante`, que representan el tanteo obtenido por dos equipos en una competición deportiva. La clase marcador también incluirá atributos para representar los nombres de los equipos y un método `toString` para mostrar de forma legible el contenido del marcador (utiliza la anotación `@Override` para comprobar que sobrescribes convenientemente el método `toString` de la clase `Object`). Se creará un clase `Baloncesto` que se utiliza para mantener actualizado el marcador de un partido de baloncesto. De esta forma la clase `Baloncesto` utilizará una instancia de marcador como atributo interno. La clase `Baloncesto` incluirá tres métodos (`tiroLibre`, `canastaNormal` y `tiroTriple`) que aceptarán el nombre de un equipo y actualizarán su marcador convenientemente. Posteriormente decidimos incluir un nuevo método `canasta` que acepta como parámetros el nombre del equipo y el valor en puntos de la canasta. Los métodos anteriores (`tiroLibre`, `canastaNormal` y `tiroTriple`) se mantienen por cuestiones de compatibilidad hacia atrás pero se marcan como *depreciados* con la etiqueta `@Deprecated`. En la prueba de este ejercicio deben usarse estos métodos depreciados y comprobar como el compilador avisa de las existencia de los mismos. Como novedad el método `canasta` también incluirá en una variable de tipo `ArrayList` la evolución de la diferencia de puntos entre los dos equipos. Por ejemplo si después de meter una canasta el marcador es 20-15 se añadirá 5 a la lista, de esa forma podemos ver la evolución del marcador. Como no vamos a usar genericidad al usar el `ArrayList` deberemos marcar la clase con la anotación `@SuppressWarnings("unchecked")`. Comprobar como el warning aparece cuando no se usa la anotación. Incluir un método que imprima el array con la evolución del marcador.

   Tema: Anotaciones, composición

5. Crea una clase que represente un reloj digital que señale la hora, el minuto y el segundo (en un formato de 24 horas). La clase dispondrá de dos constructores, uno sin parámetros que pone el reloj a 00:00:00 y otro al que se le pasa la hora, los minutos y los segundos. Esta clase deberá contener los siguientes métodos: (1) `toString`(): imprime la hora correctamente formateada. (2) `ponHora(int hora, int minutos)`, que pone el reloj con la hora y minutos que le pasamos y con los segundos a cero (3) `ponHora(int hora, int minutos, int segundos)`: igual que el anterior pero que permite pasar los segundos por parámetro (4) `adelantarHora(int hora)`: debe adelantar el reloj el número de horas que se le pasa como parámetro. En los métodos y constructores que se utilizan para fijar una determinada hora en el reloj deberá comprobarse que la hora es válida y lanzar una excepción `IllegalArgumentException` en caso contrario.

   Tema: Sobrecarga, excepciones.

6. Crea una clase que contenga un `ArrayList` de objetos de tipo `Integer` (¡ojo! la clase `Integer`, no el tipo básico `int`). La clase tendrá un método `insertar(int i)` que insertará un valor `int` en el `ArrayList` de `Integer` haciendo uso del autoboxing (es decir, no hay que hacer una conversión explícita entre valores `int` e `Integer`). También tendrá un método `int extraer(int p)` que extraerá el elemento situado en la posición p, o devolverá una excepción si la posición no existe en el `ArrayList`, de nuevo haciendo uso del autounboxing sin hacer conversiones explícitas entre tipos. Incluye también un método `media` que recorre el `ArrayList` y calcula la media de los valores almacenados en el mismo. Para recorrer el `ArrayList` se utilizará el bucle `foreach`. Finalmente, incluye un método estático de utilidad denominado `diferencia` que, dado un valor entero, indique cuánto falta hasta llegar al máximo valor entero que puede representarse en Java usando un `int`. Para ello deberás importar la variable `MAX_VALUE` de `Integer` haciendo uso de la importación estática. Ten en cuenta que el ejercicio no será valido (aunque funcione correctamente) si no hace uso del autoboxing/autounboxing, el bucle foreach o la importación estática

   Tema: autoboxing/autounboxing, importación estática, Bucle foreach, genericidad.

7. Crea una clase clase `Coche` que representa a un vehículo que tiene las siguientes características: matrícula, modelo, precio, peso en kilos, longitud en metros, altura en metros, cilindros (en línea o en V) y frenos (de tambor o de disco). Las dos últimas propiedades habrá que implementarlas como enumerados. Crea una clase `Concesionario` que se encarga de almacenar una lista de coches que hay almacenados en su interior para la venta. La clase `Concesionario` deberá incluir métodos para incluir coches en el garaje y para extraerlos (usando su matrícula para identificarlos). Finalmente incluiremos un método `listar` que nos permitirá listar por pantalla los coches que hay a la venta en el concesionario. El orden natural de los coches se define por su número de matrícula, por lo que los coches deberían implementar el interfaz `Comparable` de forma que antes de listarlos se ordenen por número de matrícula. La clase concesionario también implementará un método `listar` al que se le podrá pasar un objeto que implemente el interfaz `Comparator` por parámetro. Este método ordenará los coches según el criterio incluido en la clase `Comparator` y mostrará el resultado de esta ordenación. Crea clases que implementen el interfaz `Comparator` y que permitan ordenar los coches por precio, peso, longitud y altura. El código de prueba deberá mostrar como funcionan las distintas ordenaciones.

   Tema: Genericidad, API de Colecciones, interfaces `Comparable` y `Comparator`.
   De utilidad: En la clase `Collections` existen varios métodos para ordenar colecciones.

8. Implementar un interfaz `Operable` que use genericidad y que declare las siguientes operaciones: `suma`, `resta`, `división` y `multiplicación`. Cada una de estas operaciones se aplicará sobre el objeto actual pasando por parámetro un elemento de tipo `T` y obteniendo como resultado otro elemento de tipo `T`. Por ejemplo la suma se definirá como `T suma(T obj);` indicando que la suma del objeto actual con el objeto que se pasa por parámetro será otro objeto del mismo tipo. Se debe crear una clase `Racional` y otra `Complejo` que implementen dicho interfaz. Los números racionales son números compuestos por un numerador y un denominador de tipo entero, las operaciones sobre ellos se definen de la siguiente forma:

   - Racional (hay que tener en cuenta que el denominador no puede ser cero):

     Suma: (a/b) + (c/d) = (ad + bc) / bd

     Resta: (a/b) - (c/d) = (ad - bc) / bd

     Multiplicación: (a/b) * (c/d) = ac / bd

     División: (a/b) / (c/d) = ad / bc

   Los números Complejos a su vez están formados por una parte real y una parte imaginaria de tipo entero. Las operaciones sobre estos números se definen como:

   - Complejo:

     Suma: (a + bi) + (c + di) = (a + c) + (b + d)i

     Resta: (a + bi) - (c + di) = (a - c) + (b - d)i

     Multiplicación: (a + bi) * (c + di) = (ac – bd) + (ad +bc)i

     División: (a + bi) / (c + di) = ((ac + bd) / (c^2 + d^2)) / ((bc – ad) / (c^2 + d^2))

   A mayores deben tener métodos de lectura de las propiedades de cada uno de estos números y será necesario sobrescribir el método `toString` para que muestre a los números de la siguiente forma: Racional como `a/b` y Complejo como `a + bi`

   Tema: Definición y utilización de interfaces con genericidad