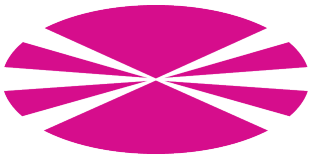

UNIVERSIDADE DA CORUÑA

Diseño Software

Boletín de Ejercicios 2 (2019-2020)

INSTRUCCIONES:

Fecha límite de entrega: 15 de noviembre de 2019 (hasta las 23:59).

- **Estructura de los ejercicios**
  - Se deberá crear un único proyecto para boletín cuyo nombre será el nombre del grupo de prácticas más el sufijo -B2 (por ejemplo `DS-11-01-B2`).
  - Se creará un paquete por cada ejercicio como en el boletín 1 (`e1`, ...) con las siguientes particularidades:
    - El ejercicio 2 y el 3 trabajan con las mismas clases por lo que es más cómodo resolverlos conjuntamente en el paquete `e2_e3`.
    - En el ejercicio 4 no hay que entregar código, sino un documento (PDF, PNG, ...). Deberéis crear un paquete `e4` y simplemente arrastrar vuestro documento dentro del paquete.
  - Es importante que sigáis detalladamente las instrucciones del ejercicio, ya que persigue el objetivo de probar un aspecto determinado de Java y la orientación a objetos.
- **Tests JUnit y cobertura**
  - Cada ejercicio deberá llevar asociado uno o varios tests JUnit que permitan comprobar que su funcionamiento es correcto.
  - Al contrario que en el primer boletín **es responsabilidad vuestra desarrollar los tests y asegurarse de su calidad** (índice de cobertura alto, un mínimo de 70-80 %, probando los aspectos fundamentales del código, etc.).
  - **IMPORTANTE: La prueba es parte del ejercicios. Si no se hace o se hace de forma manifiestamente incorrecta significará que el ejercicio está incorrecto.**
- **Evaluación**
  - Este boletín corresponde a 1/3 de la nota final de prácticas.
  - Aparte de los criterios fundamentales ya enunciados en el primer boletín habrá criterios de corrección específicos que detallaremos en cada ejercicio.
  - Un criterio general en todos los ejercicios de este boletín es que **es necesario usar genericidad** en todos aquellos interfaces y clases que sean genéricos en el API. Por lo tanto obtener el *warning* `unchecked` significará que no se está usando correctamente la genericidad y supondrá una penalización en la nota.
  - No seguir las normas aquí indicadas significará una penalización en la nota.

1. **Personas de un mercado**

   El objetivo del ejercicio es implementar las clases básicas para utilizar en un mercado. Necesitaremos por tanto almacenar información de Clientes, Dependientes y Reponedores. De todos ellos necesitamos saber su nombre, apellidos, DNI, dirección y teléfono. De estos campos, solo el nombre, los apellidos y el DNI son obligatorios. A parte de eso necesitamos la siguiente información específica de cada clase:

   - **Clientes**: 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).
   - **Dependientes**: 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 euros. Almacenaremos también su especialidad (carnicería, frutería, caja, etc.).
   - **Reponedores**: debemos saber su número de la seguridad social, su salario y el turno al que pertenecen (mañana o tarde, ya que no hay reponedores nocturnos. Intentar fijarle un turno nocturno provocará que se lance la excepción `IllegalArgumentException`). También tendremos que almacenar la información de la empresa de la que proceden ya que se suelen subcontratar.

   Encapsula los distintos atributos correctamente en la clase e incluye un método `toString` que permita representar como un `String` toda la información de cada clase. Emplea clases abstractas para generalizar características comunes entre distintas subclases.

   Crea una clase `Mercado` que tenga los métodos que se detallan a continuación:

   - Métodos para agregar clientes o trabajadores al mercado: `agregarCliente` y `agregarEmpleado` (para agregar un cliente o un empleado concreto) y los métodos `agregarClientes` y `agregarEmpleados` (para agregar una lista de clientes o empleados que se pasen por parámetro). Utiliza comodines donde consideres adecuado para flexibilizar el paso de parámetro siguiendo el principio *get y put*.
   - Un método `salariosMercado` que devuelve la suma total de los salarios de los empleados del mercado.
   - Un método `personasMercado` que devuelve una lista con todas las personas (clientes y trabajadores) involucrados en el mercado.

   Criterios:

   - Encapsulación
   - Creación de estructuras de herencia y abstracción.
   - Uso de polimorfismo y ligadura dinámica.
   - Uso de genericidad y comodines.

2. **Colección de monedas e interfaces `Comparable<T>` y `Comparator<T>`**

   Para comparar elementos y ordenar colecciones Java usa los interfaces `Comparable<T>` y `Comparator<T>` y métodos como `sort` de la clase `Collections`. A continuación incluimos una breve descripción de los mismos, la descripción completa está en la documentación del API (`https://docs.oracle.com/javase/8/docs/api/`). A la hora de analizar un interfaz prestad especial atención a los métodos abstractos ya que son los que, obligatoriamente, hay que implementar en el interfaz.

   - `Comparable<T>` incluye un método `compareTo(T o)` que compara un objeto con otro usando el *orden natural* especificado en el propio objeto. Devuelve un número entero negativo, cero o un número entero positivo, si el objeto actual es menor, igual o mayor que el objeto pasado por parámetro.
   - `Comparator<T>` es similar al anterior pero incluye un método para comparar dos objetos cualesquiera `compare(T o1, T o2)`. En este caso el resultado será un número entero negativo, cero o un número entero positivo, si el el primer argumento es menor, igual o mayor que el segundo.
   - `Collections.sort` tiene dos versiones, en la primera le pasas una lista de elementos que hayan implementado el interfaz `Comparable<T>` y el método te ordena la lista por su *orden natural*. En la segunda le pasas una lista de objetos cualesquiera y un `Comparator<T>` para esos objetos y te ordena la lista usando el comparador.

   Dada la colección de monedas creada en el boletín anterior (cuyas clases deberéis copiar a este boletín), usaremos los interfaces y métodos antes citados para conseguir el siguiente comportamiento en nuestra colección de monedas:

   - El *orden natural* de las monedas de Euro consiste en ordenarlas primero por valor (las de más valor primero), luego por países (por orden alfabético incremental de su nombre), y finalmente por diseño (por orden alfabético incremental de su descripción).
   - La colección de monedas permitirá ordenar las monedas por su *orden natural*.
   - La lista permitirá ordenar las monedas usando cualquier comparador de monedas que se le pase por parámetro. En este ejercicio realizaremos el siguiente comparador:
     - Un comparador que organiza las monedas por países (por orden alfabético incremental de su nombre), luego por valor (monedas más grandes primero) y finalmente por año (años anteriores primero).

   En la clase `String` tenemos el método `int compareTo(String str)` y el método `int compareToIgnoreCase(String str)` que nos facilitarán realizar comparaciones alfabéticas de *strings*.

   Criterios:

   - Ordenación de una colección.
   - Uso de `Comparable<T>` y `Comparator<T>`.
   - Uso de colecciones de objetos y genericidad.

3. **Colección de monedas e interfaces `Iterable<T>` e `Iterator<T>`**

   Para iterar sobre colecciones de objetos Java define los interfaces `Iterable<T>` e `Iterator<T>`. `Iterable<T>` lo implementa una colección para indicar que es posible obtener un iterador sobre la misma y tiene los siguientes métodos que implementar:

   - `Iterator<T>iterator()` devuelve un iterador sobre elementos de tipo `T`.

   El interfaz `Iterator<T>` tiene los siguientes métodos:

   - `boolean hasNext()` devuelve `true` si la iteración tiene elementos que recorrer.
   - `E next()` devuelve el siguiente elemento (`E`) a recorrer en la iteración.
   - `void remove()` elimina de la colección el último elemento devuelto usando `next()`. Solo puede llamarse una vez por cada ejecución de `next()`. En caso de que nunca se haya llamado a `next()` o de que se intente llamar dos veces a `remove()` sin haber llamado a `next()` el método lanzará la excepción `IllegalStateException`.

   En este ejercicios vamos hacer que la colección de monedas sea una colección iterable pero con una peculiaridad. A la colección de monedas le indicaremos cuál es el país que vamos a usar para iterar, y de esa forma la colección nos permitirá obtener un iterador que devuelva solo las monedas que pertenezcan a dicho país. Así, si en nuestra colección tenemos las siguientes monedas: {`EURO2-ES, EURO1-ES, CENT20-FR, EURO1-ES, EURO1-IT, CENT50-IT`}, la iteración sobre las monedas españolas devolverá: {`EURO2-ES, EURO1-ES, EURO1-ES`}

   Los **aspectos a tener en cuenta** al desarrollar la iteración serán:

   - Deberéis hacer uso de las clases `Iterable<T>` e `Iterator<T>`.
   - Si el país que se le indica para hacer la iteración es `null` el iterador recorrerá todas las monedas de la lista.
   - La iteración será *fail-fast*, cualquier modificación realizada sobre los elementos de la lista durante la iteración (y desde fuera del iterador) hará que el iterador lance la excepción `ConcurrentModificationException`.
   - La iteración permitirá, sin embargo, realizar la operación `remove()` que eliminará la última moneda devuelta por `next()` de la colección.
   - Es posible apoyarse en otros iteradores definidos en el API de Java, siempre y cuando la implementación de la estrategia *fail-fast* y el lanzamiento de excepciones (como son `NoSuchElementException`, `IllegalStateException` o `ConcurrentModificationException`) se implementen en vuestro iterador.
   - La iteración tiene que hacerse sobre los elementos de la colección sin modificarlos. No es adecuado reordenar la colección para hacer la iteracción, ni copiarla sobre otro tipo de colección para poder recorrerla.

   Criterios:

   - Iteración de colecciones de forma independiente a su implementación.
   - Uso de los interfaces `Iterable<T>` e `Iterator<T>` cumpliendo las especificaciones.
   - Uso de genericidad.

4. **Diseño UML**

   El objetivo de este ejercicio es desarrollar el modelo estático y el modelo dinámico en UML del **primer ejercicio**. En concreto habrá que desarrollar:

   - **Diagrama de clases UML** detallado en donde se muestren todas las clases con sus atributos, sus métodos y las relaciones presentes entre ellas. Prestad especial atención a poner correctamente los adornos de la relación de asociación (multiplicidades, navegabilidad, nombres de rol, etc.)
   - **Diagrama dinámico UML**. En concreto un diagrama de secuencia que muestre el funcionamiento del método `salariosMercado`.

   Para entregar este ejercicio deberás crear un paquete `e4` en el proyecto NetBeans del segundo boletín y situar ahí (simplemente arrastrándolos) los diagramas correspondientes en un formato fácilmente legible (PDF, PNG, JPG, ...) con nombres fácilmente identificables.

   Os recomendamos para UML usar la herramienta **MagicDraw** de la cual disponemos de una licencia de educación (en Moodle explicamos cómo conseguir la licencia).

   Criterios:

   - Los diagramas son completos: con todos los adornos adecuados.
   - Los diagramas son correctos: se corresponden con el código desarrollado pero no están a un nivel demasiado bajo (especialmente los diagramas de secuencia).
   - Los diagramas son legibles: tienen una buena organización, no están borrosos, no hay que hacer un zoom exagerado para poder leerlos, etc.