UNIVERSIDADE DA CORUÑA

Diseño Software
Curso 2018-2019

Boletín de Ejercicios 2

NOTA: Lee las instrucciones de los ejercicios antes de empezar a realizarlos
Fecha límite de entrega: 16 de noviembre de 2018 (hasta las 23:59)

1. **Trabajadores de la universidad**

   En la *Universidade da Coruña* (UDC) existe una plantilla de personal que está dividida en dos grupos:

   - **Personal de Administración y Servicios (PAS)**: En el personal de administración se encuentra todo el personal que desempeña labores administrativas o de gestión de algún tipo de servicio. En este caso, por simplicidad, solo vamos a considerar que existe *personal administrativo* y *personal informático*.
   - **Personal Docente Investigador (PDI)**: Personal que se dedica a la docencia y a la investigación y en el que incluimos a *profesores* y a *investigadores contratados* en algún proyecto de investigación.

   Todo el personal se identifica con su nombre, edad y DNI y recibe un salario mensual que debemos de calcular, de forma simplificada, como el número de horas de trabajo a la semana multiplicado por el importe/hora de su categoría y por el número de semanas al mes (4). Los datos de cada tipo de personal son los siguientes (las cantidades no son reales):

   - **Administrativos**: 37 horas/semana y un importe de 7,5 euros/hora.
   - **Informáticos**: 40 horas/semanas y un importe de 6 euros/hora.
   - **Profesores**: 37 horas/semana con un importe de 8 euros/hora
   - **Investigadores**: 35 horas/semana con un importe de 7 euros/hora.

   Además los profesores y los PAS tienen los siguientes ingresos adicionales:

   - **Profesores**: A los profesores se les añade un complemento salarial en función de lo que se conoce como sexenios (periodos de 6 años evaluados positivamente),

y que se pueden conceder hasta un máximo de 6. En concreto, se les añade 100 euros al mes por sexenio reconocido. Una vez concedido el sexenio el aumento de sueldo se mantiene para siempre.

- **PAS**: En un mes determinado los PAS pueden acumular horas extras. Cada hora extra se paga a 6 euros/hora. A la hora de imprimir la nómina las horas extra de los PAS (después de computarse) se inicializan a cero de nuevo.

Finalmente, existirá una clase `Universidad` con un método `imprimirNominas` que dada una lista de empleados devolverá un `String` con un listado con el nombre, la categoría y la nómina de cada uno y, al final, el gasto total en personal de la universidad.

Por ejemplo, una salida por pantalla de `imprimirNominas` sería:

```
Marta(Informatico): 960 euros
Eduardo(Informatico): 960 euros
Carlos(Administrativo con 2 horas extra): 1122 euros
Elena(Profesor con 2 sexenios): 1384 euro
Silvia(Profesor): 1184 euro
Alberto(Investigador): 980 euros
Juan(Administrativo): 1110 euros
El gasto mensual de la UDC en personal es de 7700 euros
```

Se pide crear una estructura de clases que refleje la situación mencionada anteriormente. decidiendo si algunas clases deberían ser abstractas o no (y si deberían existir métodos abstractos).

Criterios:

- Creación de estructuras de herencia y abstracción.
- Uso de polimorfismo y ligadura dinámica para facilitar la futura extensión del código a nueva categorías.
- Uso de genericidad.

2. **Listas de canciones e interfaces `Comparable` y `Comparator`**

   Vamos a crear una clase `ListaCanciones` que se encarga de almacenar una lista de canciones en su interior (podéis utilizar la clase `Cancion` del boletín anterior pero copiándola en el nuevo proyecto). En `ListaCanciones` tendremos las siguientes funcionalidades:

   - Gestionar las canciones de la lista añadiendo canciones en una posición dada de la lista o al final de la misma, eliminando canciones de una posición o moviendo canciones de una posición a otra.
   - La lista de canciones tendrá un indicador que apunta a la canción actual que está sonando. Podrá saberse la posición del indicador y cambiarla a otra posición distinta para que empiece a tocar otra canción.
   - Un método para empezar a tocar la canción en la posición del indicador. Básicamente lo que hace es devolver la canción que está en la posición del indicador. Para simplificar el problema las canciones no acaban automáticamente y pasan a la siguiente, por lo que repetidas llamadas a este método devolverán la misma canción.
   - Pasar a reproducir la canción en la posición siguiente al indicador, si estamos al final de la lista se mueve a la canción en la primera posición.
   - Pasar a reproducir la canción en la posición anterior al indicador, si estamos al principio de la lista se mueve a la canción situada en la última posición.
   - Parar de reproducir una canción. El indicador estará en una posición dada pero no hay ninguna canción sonando. Podrá saberse si la lista de canciones está reproduciendo una canción o no.

   Finalmente hay que incluir métodos para organizar las canciones dentro de la lista, para ello vamos a hacer uso de los interfaces `Comparable` y `Comparator` de Java y de 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` 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` 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` y el método te ordena la lista por su orden "natural". En la segunda le pasas una lista de objetos cualesquiera y un `Comparator` para esos objetos y te ordena la lista usando el comparador.

Usaremos los interfaces y métodos antes citados para conseguir el siguiente comportamiento en nuestra lista de canciones:

- El orden natural de las canciones es el orden alfabético a partir del título de la canción. En caso de que el título sea igual se ordenará también por orden alfabético del álbum.
- La lista de canciones permitirá ordenar las canciones por su orden "natural". Al reordenar de cualquier forma la lista se paran las canciones que están tocando y el indicador pasa a la primera posición (pero sin tocar la canción).
- La lista permitirá ordenar las canciones usando cualquier comparador de canciones que se le pase por parámetro. En este ejercicio realizaremos dos comparadores:
  - Un comparador que organiza las canciones por orden alfabético primero de autor, luego de álbum y finalmente de título de la canción.
  - Un comparador que organiza las canciones por orden alfabético de estilo y luego por orden alfabético del título.

En la clase `String` tenemos un método `int compareToIgnoreCase(String str)` que nos facilitará realizar comparaciones alfabéticas de *strings*.

Criterios:

- Abstracción y encapsulación de una lista de canciones.
- Uso de `Comparable` y `Comparator`.
- Uso de colecciones de objetos y genericidad.

3. **Recorridos de listas e iteradores**

En un concurso de telerrealidad solo queda una plaza disponible para que entre el último concursante pero quedan muchos candidatos posibles (n) que han sacado la misma calificación en las pruebas clasificatorias.

Para desempatar los productores han decidido alinear a todos los concursantes, asignarles un número consecutivo del 1 a n y luego ir recorriendo la lista eliminando concursantes siguiendo el siguiente proceso:

- El recorrido por la lista será con "rebote". Esto quiere decir que al llegar al final de la lista (contando hacia adelante, se volverá hacia el principio contando hacia atrás). Así el recorrido de una lista con cinco elementos sería `1-2-3-4-5-4-3-2-1-2-3`, etc.
- Se sacará un número (k) que representa el momento en el que paramos de hacer un recorrido por la lista, seleccionamos un candidato y lo eliminamos de la lista.
- Después de eliminar un candidato de la lista se continuará el proceso con el siguiente elemento, recorriendo la lista con rebote y contando k posiciones y eliminado al candidato de dicha posición.
- El último candidato existente en la lista será el ganador que ocupe la última plaza disponible del concurso.

Por ejemplo, un recorrido `n=5` y `k=3` sería:

```
1-2-3-4-5 => Es la lista de candidatos y empezamos a contar en 1.
1-2-4-5   => Contamos tres posiciones y eliminamos el candidato 3.
1-2-5     => Empezamos en 4, contamos 3 y rebotamos a 4 eliminándolo.
1-5       => Seguimos en 2, al rebotar volvemos a 2 y lo eliminamos.
1         => Seguimos en 5, después de dos rebotes eliminamos el 5.
El candidato en la posición 1 es el ganador.
```

Lo que se pide es escribir una función que, dados los números n y k me devuelva la posición en la lista que será la ganadora. Este programa deberá crear una lista con las posiciones y recorrerla usando un objeto de tipo `Iterator` que recorra la lista con rebote.

El interfaz `Iterator` tiene los siguientes métodos que tendremos que implementar:

- `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()` devuelve 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`.

Para comprobar que la función que obtiene la posición ganadora está usando los iteradores de forma correcta vamos a definir un nuevo iterador sobre la lista que sea

un iterador circular, es decir, cuando llega la final de la lista no rebota, sino que empieza de nuevo por el principio.

Por ejemplo, un recorrido `n=5` y `k=3` en una lista circular sería:

```
1-2-3-4-5 => Es la lista de candidatos y empezamos a contar en 1.
1-2-4-5   => Contamos tres posiciones y eliminamos el candidato 3.
2-4-5     => Empezamos en 4, contamos 3 y volvemos a 1 eliminándolo.
2-4       => Seguimos en 2, llegamos a 5 y lo eliminamos.
4         => Seguimos en 2, después 4 y volvemos a 2 y lo eliminamos.
El candidato en la posición 4 es el ganador.
```

Actualiza por lo tanto la función que obtiene la posición ganadora para que, además de los parámetros n y k, acepte un tercer parámetro que indique el tipo de recorrido que se quiere hacer en la lista (con rebote o circular). El código que realiza la iteración tiene que ser el mismo, lo que cambia es el iterador usado.

Criterios:

- Iteración de colecciones de forma independiente a su implementación.
- Uso del interfaz `Iterator`.
- 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 `imprimirNominas`.

   Para entregar este ejercicio deberás crear un directorio `doc` en el proyecto NetBeans del segundo boletín y situar ahí 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.