Laboratorio 3 – BankersQueue

Este laboratorio tiene tres objetivos fundamentales:

1. Implementar una estructura de datos secuencial: la cola.
2. Implementar un iterador para la cola del punto 1.
3. Implementar una clase que implemente la estructura de la cola (punto 1) para resolver un problema concreto: **una cola de clientes en un banco**.

Tarea 1: Creación de BankersQueue<E>

El primer paso será crear una cola. Para ello, será necesario definir una interfaz, que denominaremos **Queue**<*E*>:

```
1 public interface Queue<E> {
2     void add(E e);
3     void remove();
4     E element();
5     boolean isEmpty();
6     int size();
7 }
```

Id con cuidado ya que si por un descuido (o por aceptar erróneamente una sugerencia de IntelliJ) importáis java.util.Queue, estaríais usando la interfaz **Queue**<*E*> predefinida, que incluye más operaciones.

Las operaciones que debe contener la interfaz **Queue**<*E*> son:

- **add**: añade un elemento al final de la cola.
- **remove**: elimina el primer elemento de una cola no vacía, o lanza *NoSuchElementException* si esta está vacía.
- **element**: devuelve el primer elemento de la cola, sin modificarla, o lanza *NoSuchElementException* si esta está vacía.
- **isEmpty**: devuelve cierto si la cola está vacía, o falso en caso contrario.
- **size**: devuelve el número de elementos en la cola.

Como se puede comprobar *remove* y *element* lanzan *NoSuchElementException*, clase predefinida que representa una excepción en tiempo de ejecución y, por tanto, no comprobada.

La implementación que se os pide es la llamada "**Banker's Queue"**, en la que una cola se implementa usando dos pilas. Aunque la biblioteca de clases de java

incluye una clase `Stack`<*E*>, usaremos dos `ArrayList`<*E*>, lo que simplificará la solución de la Tarea 2.

En una cola, las diferentes operaciones necesitan acceder eficientemente tanto a la primera posición como a la última:

- **remove** y **element** a la primera
- **add** a la última

Como en una pila solamente disponemos de acceso a su última posición, necesitaremos dos pilas diferentes, **front** y **back**, en las que sus últimos elementos coincidan con los primeros y últimos de la cola.

Obviamente en algún momento se deberán transferir elementos de una a otra, pero el coste amortizado de dicha operación será O(1) por las mismas razones de que lo es en el caso de tener que aumentar el tamaño de un `ArrayList`.

En un momento dado, la cola 1 -> 2 -> 3 -> 4-> 5 puede estar representada por las pilas:

- front: 2 <- 1
- back: 3 <- 4<- 5

Fijaos en que para obtener el primer elemento hemos de consultar la cima de la pila front (1), y para añadir por ejemplo un 6 hemos de hacer push sobre la pila back.

¿Cuándo se pasan elementos de una pila a la otra? Cuando se necesita acceder a la parte inicial, pero front está vacío. ¿Qué se hace? Transferir elementos de la pila back hacia front. Por ejemplo, después de dos removes front quedaría vacío por lo que, si preguntamos por element, hemos de trasladar elementos de una pila a la otra, es decir:

- front: 5 <-4<-3
- back:

Y ahora podemos ya responder que el element es 3.

En este apartado se pide implementar la clase `BankersQueue`<*E*> usando la estrategia descrita y una clase de pruebas que compruebe el buen funcionamiento de todas las operaciones.

Tarea 2: Implementar un Iterador para BankersQueue<E>

La Tarea 2 se centra en hacer que la clase implementada en la Tarea 1 (***BankersQueue*** *<E>)* implemente las operaciones no optativas de la interfaz *Iterable<E>*.

```
1 public class BankersQueue<E>
2         implements Queue<E>, Iterable<E> {
3     ...
4     Iterator<E> iterator();
5 }
```

Para hacer su implementación, seguiremos la misma estrategia que se ha mostrado en los apuntes, es decir, usaremos una clase interna privada que implemente la interfaz Iterator<E> y un contador para detectar si se ha modificado la cola mientras se usaba el iterador:

- **hasNext()**: Indica si existe un elemento "siguiente" en la cola.
- **next()**: Devuelve el siguiente elemento en la cola y si esta está vacía lanzará la excepción *NoSuchElementException*. Si la lista se ha modificado durante el recorrido lanzará la excepción **no comprobada** *ConcurrentModificationException*.
- **remove():** Lanza *UnsupportedOperationException*.

Para implementarla, pensad en cómo usar sendos iteradores sobre las listas que forman la representación de la cola.

Como siempre, realizad pruebas usando JUnit 5 comprobando que todas las operaciones de los iteradores funcionan correctamente.

Tarea 3: Implementar un simulador de colas en un banco usando la clase BankersQueue<E>

Una vez disponemos de la cola, esta tarea se centra en simular la cola de clientes de un banco. Para tal efecto, partiremos de una lista de *BankClient* (clase que debéis implementar), que contendrá los siguientes campos, con sus métodos de lectura/escritura:

- **int arrivalTime (tiempo de llegada)**: lo simplificaremos a los segundos que han pasado desde la apertura de la oficina para su atención al público y la llegada del cliente.
- **int exitTime (tiempo de salida)**: indica cuantos segundos han pasado desde la apertura de la oficina hasta la salida del cliente, una vez atendido.

Lógicamente será el resultado de *tiempo de llegada* + *tiempo de espera* + *tiempo de servicio*.

Para simplificar, el *tiempo de servicio* (tiempo que se tarda en atender a un cliente) será siempre 120 segundos (definid una constante). Por otro lado, *el tiempo de espera* será la diferencia entre el *tiempo de llegada* y el momento en que un cajero lo atiende.

Considerando esto, implementad un simulador (clase *BankSimulator*), tal que:

- Simule la llegada de 100 clientes, que van llegando de uno en uno cada 15 segundos.
- Plantee 10 escenarios distintos: con 1 cajero atendiendo, con 2, con 3, … y hasta con 10 cajeros.
- Calcule el tiempo medio necesario que cada cliente ha pasado en el banco para ser atendido (tiempo de salida menos el tiempo de llegada), considerando cada uno de los 10 escenarios.

Otras consideraciones

No olvidéis compilar usando la opción –Xlint:unchecked para que el compilador os avise de errores en el uso de genéricos.

Entrega

El resultado obtenido debe ser entregado por medio de un proyecto IntelliJ por cada grupo, comprimido y con el nombre "Lab3_NombreIntegrantes". En el proyecto deberán estar presentes las pruebas realizadas con JUnit 5.

Además, se debe entregar un documento de texto, máximo de cuatro páginas, en el cual se explique la solución a cada una de las tareas **y en el que se espera que proporcionéis diagramas que muestren el diseño de las operaciones de las colas y sus iteradores, así como los resultados de las simulaciones.**

Consideraciones de Evaluación

A la hora de evaluar el laboratorio se tendrán en cuenta los siguientes aspectos:

- El código de cada proyecto ofrece una solución a cada una de las tareas planteadas.

- Realización de pruebas con JUnit 5.0.

- Explicación adecuada del trabajo realizado, es decir, esfuerzo aplicado al documento de texto solicitado.

- Calidad y limpieza del código.