Trabajo JDBC 2ª Convocatoria

Estructura de Tablas Base de Datos

1. Tabla de recorridos:
   recorridos (idRecorrido(PK), estacionOrigen, estacionDestino, kms, horaSalida, horaLlegada, precio)
   que representa un recorrido del tren que podría repetirse en varias fechas, caracterizado por una estación origen, otra de destino, una hora de salida y otra de llegada.
   Como en Oracle no hay campos TIME las horas de salida y llegada se han representado mediante campos TIMESTAMP en los que despreciaremos la fecha.
2. Tabla de viajes:
   viajes (idViaje(PK), idTren, idRecorrido ----> FK a recorridos, fecha, nPlazasLibres, realizado, idConductor)
   que representa el viaje que hace un tren en un determinado recorrido y en una determinada fecha.
   nPlazasLibres representa el número de plazas libres que quedan en el tren.
   realizado es un booleano que indica si el viaje ya se ha realizado o no, y el idConductor identifica al conductor (no intervienen en este problema).
3. Tabla de tickets
   tickets (idTicket(PK), idViaje (FK ------> Viajes, fechaCompra, cantida, precio)
   Que representa un ticket o un conjunto de varios tickets (por eso hay un campo **cantidad** que indica cuantos) correspondientes a un viaje.
   El campo **precio** contiene la multiplicación del campo **cantidad** por el campo **precio de la tabla recorridos**, correspondiente al recorrido de ese viaje.

En la carpeta sql se haya un script SQL de carga que borra las tablas y las crea de nuevo, y el cual es llamado de manera automática al ejecutar el método *main* de la clase *Main*. rellena con varias filas de ejemplo.

A su vez, la practica necesita de la librería *user_library* para Eclipse, la cual se encuentra incluida en el código repartido junto al tema de JDBC.

También se recomienda experimentar con la configuración de registros de la aplicación (logs), la cual se encuentra en el archivo log4j.properties. En ese archivo se pueden configurar los límites para la impresión de registros tanto en fichero como en la consola.

Descripción General de la práctica:

Se debe finalizar la implementación de la clase ServicioImp, el cual contiene los siguientes métodos a completar por el alumno:

```
public void comprarBillete(Time hora, Date fecha, String origen, String destino, int nroPlazas) throws SQLException
```

Este método, dado una hora, una fecha, una ciudad origen y una ciudad destino, los cuales sirven para caracterizar un viaje, inserta una fila en la tabla de tickets por la compra de **nroPlazas**, decrementando el número de plazas libres de ese viaje.

```
public void anularBillete(Time hora, java.util.Date fecha, String origen, String destino, int nroPlazas, int ticket) throws SQLException
```

A la inversa que el método de **comprarBilletes**, este método aumentara el número de plazas libres para el viaje indicado.

```
public void modificarBillete(int billeteId, int nuevoNroPlazas) throws SQLException;
```

Este método, dado el identificador del billete y un número nuevo de plazas, sirve para actualizar los datos del billete, incluyendo el número de plazas reservadas previamente y el precio del billete. El precio se ajusta dependiendo de si el número de plazas nuevo es mayor o menor que el anterior.

Estos métodos lanzan sólo excepciones del tipo **CompraBilleteTrenException** que hay que añadir a las que ya existen. Esta clase necesita implementarla también el alumno, e incluye dos posibles códigos de error:

1. La excepción registrará el problema de que no queden plazas suficientes para ese viaje. El código será el "1" y el mensaje de error "No hay plazas suficientes."
2. La excepción registrará el problema de que no exista un viaje que corresponda con ese origen, destino, fecha y hora de salida. Su código será el "2" y el mensaje de error "No existe viaje para tal fecha, hora, origen y destino.".

El resto de las excepciones serán tratadas dentro del propio método.

Toda excepción sea del tipo que sea retrocederá la transacción y se propagará al método que hace la llamada a la transacción.

Las excepciones que no sean del tipo **CompraBilleteTrenException** dejará el mensaje del error con logger.error(...) en el logger.

Recuerda hacer rollback en cualquiera de las excepciones.

Adicionalmente, se deben implementar a su vez pruebas automáticas que demuestren el correcto funcionamiento del metodo **anularBillete**.

Para ello, se debe implementar el método **ejecutarTestsAnularBilletes** de la clase **Tests.** En esta misma clase **Tests** se encuentra el método **ejecutarTestsCompraBilletes**, el cual ejecuta las pruebas para el método **comprarBillete,** y que se puede usar de ejemplo a seguir.

Adicionalmente, la clase **Tests** incluye el metodo **ejecutarTestsModificarBillete**, el cual ejecuta las pruebas para el método **modificarBillete,** y que se puede usar también de ejemplo a seguir.

Condiciones de entrega:

La práctica se debe entregar en una única carpeta comprimida (ZIP) en el buzón de entrega correspondiente en el aula virtual y contener una sola carpeta que se pueda descomprimir y que usando la función de Eclipse "Open Projects from File System" importe y se pueda ejecutar.

No se corregirá ni puntuará una práctica que no se importe y ejecute correctamente.

No se admitirán entregas por correo electrónico, así que no apuréis la fecha de entrega para evitar problemas técnicos a la hora de entregar.

Notas finales:

1. Utiliza la clase PoolDeConexiones.java que se te provee.
2. Utiliza sentencias preparadas, aprovechando que esa clase te implementa una caché de sentencias.
3. Libera todos los recursos utilizados con un bloque *finally*.
4. Las pruebas automáticas se invocan desde el *main*, a través de la clase *Tests*.