DatabaseConcurrencyAssigment4

Implementation:

- A MasterThread = the main method of the project, which clears all the bookings, gets the notBookedCount which will be used by the concurrent threads and then starts 10 concurrent threads, displays the results statistics after all seats are booked;
- A UserThread, which will simulate real users by creating an instance of Reservation class and call the available methods for a normal user;
- A SharedCounter, for all the statistics to be counted from each thread and to be displayed in the MasterThread;
- A DataMapper which contains all the methods with the necessary queries for handling the transactions. reserve() and book() are particularly interesting and will be explained in detail futher on; clearAllBookings() is called in the main thread
- A DBFacade which opens and closes connections to the database and also acting like an interface to the DataMapper.

There will always be 10 concurrent threads, until the last seat is booked. This means that even when there are 1 to ~ 20 seats not booked (some of them with active reservations), there will still be 10 people trying to get them or it (the last one) in the same time. At this point we're going to see some extra errors which don't appear otherwise, because the threads are started, but they can't find a seat to reserve. After a thread finishes, it is removed from the array of 10 threads. Every second there will be started a number of threads that's needed to fill the array. We made it this way to not only have 10 concurrent threads, but also make them start exactly in the same millisecond if possible, so we would get concurrent SQL Queries - for the best purpose of this demonstration.

```
thread#41 succeeded to book A3: status 0
thread#37 reserved A21
thread#37 gave up after reserving A21
thread#35 failed to book A22: status -2
thread#42 succeeded to book A21: status 0
thread#48 started
thread#49 started
thread#50 started
thread#51 started
```

(Example of 4 threads being started exactly in the same time, after 4 others finished)

The UserThread starts by trying to reserve 1 seat and then it sleeps between 0 and 10 seconds. Only after this - if the seat was successfully reserved it moves further. We made it this way because for the last seats to be booked, there will be a lot of failures to reserve a seat, giving an absurd number of failures for reservation, not relevant for the purpose of demonstrating the transaction. With a chance of 75% the thread will continue with a booking, otherwise it will simply end, leaving the reservation in place, just like a normal user would close a browser's tab. If it continues with the booking, different errors might occur and the statistics about this are counted as well.

The DataMapper has 2 important methods for this whole process:

- **book(String plane_no, String seat_no, long id, Connection connection)**

As you can see below, the transaction to update a seat from reserved to booked can be done in one single request, so there's no need to put a lock on it:

```
String sqlString0 = "update SEAT "
        + "set BOOKED = ?, BOOKING_TIME = (select date_to_unix_ts(systimestamp) from DUAL) "
        + "where PLANE_NO = ? "
        + "and SEAT_NO = ? "
        + "and RESERVED = ? "
        + "and BOOKED is NULL "
        + "and (select date_to_unix_ts(systimestamp) from DUAL) - BOOKING_TIME <= 5";
```

Important notice: it will also consider as non-reserved the seats which were booked with 6 or more seconds before running the query; In the same time, as you can see in the image below, for reserving a seat are accepted all bookings with 0 to 5 seconds in the past, so 5 is accepted. For saving the number of seconds as an integer, we created a function in the database which returns the unix timestamp of the given date:

```
create or replace function date_to_unix_ts( PDate in date ) return number is
   l_unix_ts number;
begin
   l_unix_ts := ( PDate - date '1970-01-01' ) * 60 * 60 * 24;
   return round(l_unix_ts);
end;
```

- **reserve(String plane_no, long id, Connection connection)**

This method, which has to be called first, is a bit more tricky. The Update query alone would return an integer number - how many rows have been updated, meaning reserved in our case. But the requirement is to return the reserved seat number, so we need another query for this instead:

```
String sqlString0 = "select SEAT_NO from SEAT "
        + "where PLANE_NO = ? "
        + "and (RESERVED is NULL "
        + "  or (BOOKED is NULL and (select date_to_unix_ts(systimestamp) from DUAL) - BOOKING_TIME > 5)) "
        + "and ROWNUM <= 1 "
        + "for update ";
```

This will also put a lock on it, to ensure that no other user is able to update the same row between this moment of finding the empty seat, returning the seat number to the client and running the next query(in the image below) to actually reserve it.

```
String sqlString1 = "update SEAT "
        + "set RESERVED = ?, BOOKING_TIME = (select date_to_unix_ts(systimestamp) from DUAL) "
        + "where PLANE_NO = ? "
        + "and SEAT_NO = ?";
```

If the seat is successfully reserved, the connection commits; otherwise, it rollbacks. Thus it releases the lock.

Test results with lock, as described:

```
failed to reserve seats: 36
total reserved seats: 310
total successfully booked seats: 96
total reserved and not booked (~25% of total reserved): 63
total failed to book because of overbooking(because of the delay > 5s): 85
total reserved and timed out for booking: 64
total reserved and not booked - other errors: 2
BUILD SUCCESSFUL (total time: 5 minutes 25 seconds)
```

And without a lock (the line with “for update” was commented out):

```
failed to reserve seats: 17
total reserved seats: 354
total successfully booked seats: 96
total reserved and not booked (~25% of total reserved): 89
total failed to book because of overbooking(because of the delay > 5s): 132
total reserved and timed out for booking: 34
total reserved and not booked - other errors: 3
BUILD SUCCESSFUL (total time: 5 minutes 47 seconds)
```

“failed to reserve seats” = the threads that are running close to the end of the 4 minutes, when there are no more seats available for reserving so null is returned, but not all the seats have been booked.

“total reserved and not booked - other errors” = sql exceptions for trying to create a connection to the database. It was working perfect when we finished the java program but not now, at the time of writing the report.

“total reserved seats” = number of seats that have been reserved in order to proceed to next steps.

“total reserved and timed out for booking” = number of threads that successfully reserved, but attempted to book after 6 - 10 seconds, so the reservation actually expired.

“total failed to book because of overbooking*(because of the delay > 5s)” = much alike the previous one, except that there existed another thread that found the same reservation as expired, so it reserved it instead with another user id.
* mistake, “overreserving” instead of “overbooking”

The difference between having transactions and not having transactions lies in the last 2 statistics explained. In the 1st example the numbers are pretty normal, 64 timed out for booking / 85 overreserved, but in the second one you will see a shift: only 34 timed out for booking / 132 overreserved. In this number, 132, there are also included the ones that were overreserved as a result of not having a lock (a proper transaction), because of concurrent SQL queries. As a conclusion, users that might try to book just in time (0-5 seconds) after successfully reserving - would get an error, although it’s definitely not their fault that their reservation was taken by someone else.