What is a Transaction?

There are many situations where the business logic of your application requires not to execute a query unless another one completes.

For example, let's say you order a car online.
For that, an ordering application should perform the following steps: withdraw money from your bank account, put money into the seller account, and place a purchase order. Imagine that each step executes independently, and system error occurs after the first two steps, which means that money will be withdrawn from your account and sent to the seller, but you will never receive your car.

To avoid such situations, we can treat all three steps as a single unit that can succeed or fail completely. In that case, if a system error or any other failure occurs at any stage of the ordering, your money won't be transferred to the seller, and the seller doesn't place a purchase order. Such a unit of several SQL statements is called a *transaction*.

Commit and Auto-commit mode
To make all changes made by a transaction permanent in the database, we need to commit the transaction. However, when you create the JDBC connection, it is in *auto-commit mode*, which means that each statement is treated as a separate transaction. That is why database data is changed immediately after calling a statement execution method. As we discussed earlier, it is not always the behavior that we desire.
To disable auto-commit mode, we need to explicitly disable it for the current connection by calling the setAutoCommit(false) method of the Connection object. Once the auto-commit mode is disabled, calling execution methods of statement objects will not affect the database until you explicitly *commit* a transaction by calling the commit method of the Connection object. Let's look at the example and create an SQLite database called store.db that holds order and invoice tables:

```
CREATE TABLE "invoice"
(
   id              INTEGER PRIMARY KEY,
   shipping_address TEXT    NOT NULL,
   total_cost       INTEGER NOT NULL
);

CREATE TABLE "order"
(
   id          INTEGER PRIMARY KEY,
   invoice_id   INTEGER NOT NULL,
   product_name TEXT    NOT NULL,
   FOREIGN KEY (invoice_id) REFERENCES invoice (id)
```

);
Now we can create a Connection object called con and execute con.setAutoCommit(false) method to disable auto-committing so that we will create both an invoice and an order in the case of successful execution or none of them in the case of failure:

```
public class StoreDB {

    public static void main(String[] args) {
        String url = "jdbc:sqlite:path-to-store.db";

        SQLiteDataSource dataSource = new SQLiteDataSource();
        dataSource.setUrl(url);

        String insertInvoiceSQL = "INSERT INTO \"invoice\" " +
                "(id, shipping_address, total_cost) VALUES (?, ?, ?)";

        String insertOrderSQL = "INSERT INTO \"order\" " +
                "(id, invoice_id, product_name) VALUES (?, ?, ?)";

        try (Connection con = dataSource.getConnection()) {

            // Disable auto-commit mode
            con.setAutoCommit(false);

            try (PreparedStatement insertInvoice =
con.prepareStatement(insertInvoiceSQL);
                 PreparedStatement insertOrder =
con.prepareStatement(insertOrderSQL)) {

                // Insert an invoice
                int invoiceId = 1;
                insertInvoice.setInt(1, invoiceId);
                insertInvoice.setString(2, "Dearborn, Michigan");
                insertInvoice.setInt(3, 100500);
                insertInvoice.executeUpdate();

                // Insert an order
                int orderId = 1;
                insertOrder.setInt(1, orderId);
                insertOrder.setInt(2, invoiceId);
                insertOrder.setString(3, "Ford Model A");
                insertOrder.executeUpdate();

                con.commit();
            }
        } catch (SQLException e) {
            e.printStackTrace();
```

```
      }
   }
}
```
Whenever the commit is made, all transaction statements are executed and stay permanently in the database. It means that after calling the con.commit method, an invoice and the corresponding order are created.

If the setAutoCommit(true) method is called during a transaction, the transaction will be committed.

Rollback

Imagine that an error happens during statement execution, and SQLException is thrown. In the code above, we catch this exception and print an error stack trace. But we don't explicitly make sure that the transaction ends. As a result, we may commit invalid changes later if we continue using the same connection. To prevent this, we can rollback all transaction changes to undo all the changes made in the current transaction.
In JDBC, you can rollback a transaction by invoking the Connection object's rollback method. It is recommended to call the rollback method in the catch block if your code throws an SQLException during a transaction. Let's rewrite our catch statement to rollback the transaction:

```
catch (SQLException e) {
   if (con != null) {
      try {
        System.err.print("Transaction is being rolled back");
        con.rollback();
      } catch (SQLException excep) {
        excep.printStackTrace();
      }
   }
}
```

Not explicitly ending the transaction may also lead to the transaction living longer than it is necessary.

Savepoint

As we have mentioned earlier, there are two ways to end the transaction: using the commit method that saves all transaction changes or using the rollback method that discards them. In most cases, this is exactly what we need. However, sometimes we have to save some changes and discard others. For that, we have to create a *savepoint* in the code. All changes introduced up to the savepoint will be saved and can be committed later.
To create a savepoint in JDBC, we can call the setSavepoint method of the Connection object that returns a Savepoint object. We can roll back the transaction to the savepoint so that all changes made before the savepoint will be preserved, and all changes after the savepoint will be discarded. To rollback

the transaction to the specified savepoint, we can use the rollback(Savepoint savepoint) method that accepts a Savepoint object. To make it more concrete, let's consider the following example:

```
String insertInvoiceSQL = "INSERT INTO \"invoice\" " +
        "(id, shipping_address, total_cost) VALUES (?, ?, ?)";

String selectAddressSQL = "SELECT shipping_address " +
        "FROM \"invoice\" WHERE id = ?";

try (Connection con = dataSource.getConnection()) {

    // Disable auto-commit mode
    con.setAutoCommit(false);

    try (PreparedStatement insertInvoice =
con.prepareStatement(insertInvoiceSQL)) {

        // Create a savepoint
        Savepoint savepoint = con.setSavepoint();

        // Insert an invoice
        int invoiceId = 1;
        insertInvoice.setInt(1, invoiceId);
        insertInvoice.setString(2, "Dearborn, Michigan");
        insertInvoice.setInt(3, 100500);
        insertInvoice.executeUpdate();

        PreparedStatement selectAddress =
con.prepareStatement(selectAddressSQL);
        selectAddress.setInt(1, invoiceId);
        ResultSet resultSet = selectAddress.executeQuery();

        if (resultSet.getString(1).equals("Dearborn, Michigan")) {
            con.rollback(savepoint);
        }

        con.commit();
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

In the code above, we create a Savepoint object called savepoint and rollback the transaction changes to the savepoint if the invoice with id equals 1 has shipping address equals "Dearborn, Michigan".

It is possible to release a savepoint, or in other words, to remove it from the current transaction. One way to release a savepoint is by using the Connection.releaseSavepoint method that takes a Savepoint object as a

parameter. Any savepoint that has been created in a transaction is automatically released and becomes invalid when the transaction is committed or when the entire transaction is rolled back.

Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints created after the savepoint that was rolled back. If a savepoint has been released, attempting to reference it in a rollback operation throws an SQLException.

Conclusion

In this topic, we discussed what a transaction is and why we need it. Since JDBC enables auto-commit mode by default, we discussed why it could be detrimental from the business logic perspective. To disable the auto-commit mode, we can use the Connection.setAutoCommit method. If we disable the auto-commit mode, we have to explicitly end a transaction by committing all transaction changes using the Connection.commit method or by rolling back the transaction using the Connection.rollback method. We also examined transaction savepoints as an additional mechanism to control your application.