Programming Advanced Java

WEEK 3 - JDBC

PXL DIGITAL

Goals

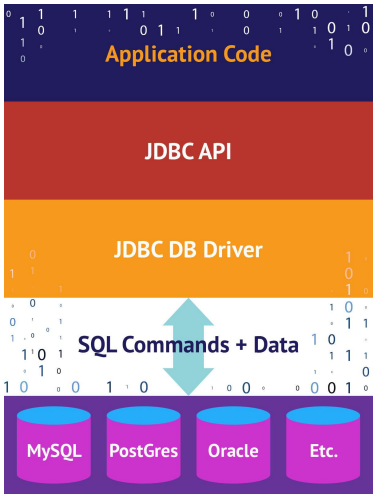
The **junior-colleague**

- can describe what JDBC is.
- can identify the base interfaces of JDBC.
- can use JDBC to connect to a database.
- can use JDBC to query a database.
- can use JDBC to create a table.
- can use JDBC to insert, update, and delete records in a table.
- can use transactions in a JDBC application.
- can describe what SQL injection is.
- can use Prepared Statements to prevent SQL injection.
- can explain the ACID-properties of a transaction.
- can explain different isolation levels and the problems that can possibly occur.
- can describe and implement the Data Access Object Pattern.

JDBC = **Java Database Connectivity**

Java's low-level API for making database connections and handling SQL queries and responses

JDBC is an *adapter layer* from Java to SQL: it gives Java developers a common interface for connecting to a database, issuing queries and commands, and managing responses.



HOGESCHOOL PXL

JDBC interfaces

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.ResultSet;
import java.sql.Statement;
```

Each of these imports provides access to a class that facilitates the standard Java database connection:

- `Connection` represents the connection to the database.
- `DriverManager` obtains the connection to the database. (Another option is `DataSource`, used for connection pooling.)
- `ResultSet` and `Statement` model the data result sets and SQL statements.

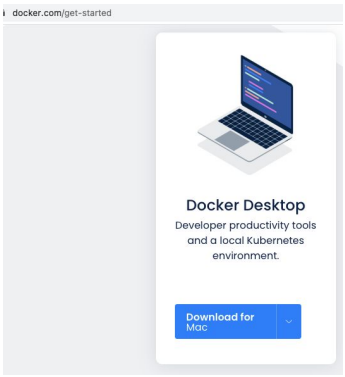
`SQLException` handles SQL errors between the Java application and the database.

Demo

Code: https://github.com/custersnele/JA2_introduction_jdbc

Prerequisite: having docker installed

https://www.docker.com/get-started

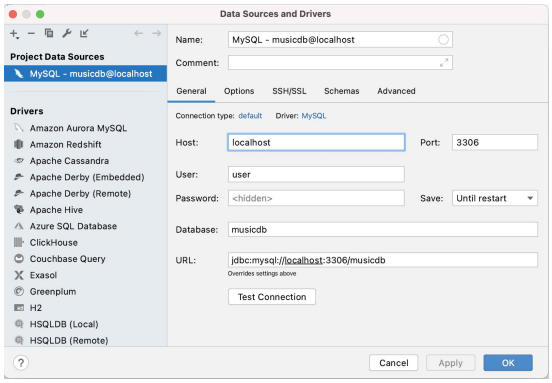


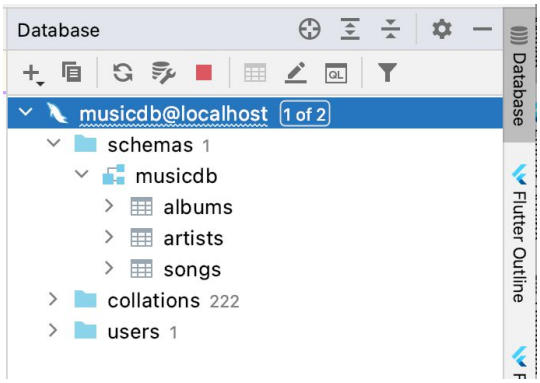
HOGESCHOOL PXL

Using docker

$ cd src/main/docker/

$ docker-compose up





HOGESCHOOL PXL

Obtain a database connection

```
try (Connection conn =
     DriverManager.getConnection("jdbc:mysql://localhost:3306/musicdb" , "user", "password"))
{
   LOGGER.info("Connnection established: "  + conn.getCatalog());
   LOGGER.info("Connnection established: "  + conn.getMetaData().getDriverName());
} catch (SQLException e) {
   LOGGER.fatal("Something went wrong." , e);
}
```

ConnectingToDatabase.java

HOGESCHOOL PXL

The interface java.sql.Connection extends AutoCloseable , so use the try-with-resources statement.

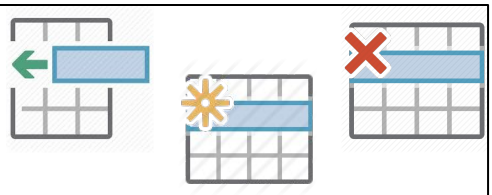
Create a Table

```
try (Connection conn =
    DriverManager.getConnection("jdbc:mysql://localhost:3306/musicdb", "user", "password");
    Statement statement = conn.createStatement()) {
    statement.execute("CREATE TABLE contacts (id INTEGER NOT NULL AUTO_INCREMENT, " +
      "name TEXT, " +
      "phone INTEGER, " +
      "email TEXT, " +
      "PRIMARY KEY (id))");
    LOGGER.info("Table 'contacts' created.");
} catch (SQLException e) {
    LOGGER.fatal("Something went wrong.", e);
}
```

CreateTable.java

HOGESCHOOL PXL

Insert, Update, and Delete



```
try (Connection conn =
        DriverManager.getConnection("jdbc:mysql://localhost:3306/musicdb" , "user", "password");
        Statement statement = conn.createStatement()) {

      statement.execute("INSERT INTO contacts (name, phone, email) "  +
      "VALUES('Joe', 45632, 'joe@anywhere.com')" );

      statement.executeUpdate("UPDATE contacts set phone='486666' WHERE name = 'Jane'" );

      statement.executeUpdate("DELETE FROM contacts WHERE email = 'dog@email.com'" );
   } catch (SQLException e) {
      LOGGER.fatal("Something went wrong." , e);
   }
```

DEMO InsertUpdateDelete.java

HOGESCHOOL PXL

Query

```
try (Connection conn =
        DriverManager.getConnection("jdbc:mysql://localhost:3306/musicdb" , "user", "password");
        Statement statement = conn.createStatement()) {
      statement.execute("SELECT * FROM contacts");
      ResultSet results = statement.getResultSet();
      int numberOfRows = 0;
      while (results.next()) {
         numberOfRows++;
         System.out.println(results.getString("name") + " " + results.getString("email"));
      }
      LOGGER.info("Number of rows in table: " + numberOfRows);
   } catch (SQLException e) {
      LOGGER.fatal("Something went wrong." , e);
   }
```

InsertUpdateDelete.java

HOGESCHOOL PXL

Auto-commit

When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed.

For a simple case:

```
conn.setAutoCommit(false);
statement.executeQuery(query);
statement.commit();
```

will be the same as:

```
conn.setAutoCommit(true);
statement.executeQuery(query);
```

InsertUpdateDelete.java

HOGESCHOOL PXL

https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html

Query

```
try (Statement statement = conn.createStatement();
     ResultSet results = statement.executeQuery("SELECT * FROM " + TABLE_ARTISTS)) {
     List<Artist> artists = new ArrayList<>();
     while (results.next()) {
        Artist artist = new Artist();
        artist.setId(results.getInt(COLUMN_ARTIST_ID));
        artist.setName(results.getString(COLUMN_ARTIST_NAME));
        artists.add(artist);
     }
     return artists;
  } catch (SQLException e) {
     LOGGER.fatal("Executing query failed. " , e);
     return null;
  }
}
```

QueryForArtists.java

HOGESCHOOL PXL

Query

```
10:36:32.697 [main] INFO  be.pxl.ja2.jdbc.model.MusicDatasource - MusicDatasource open...
10:36:32.701 [main] INFO  be.pxl.ja2.jdbc.model.MusicDatasource - SQL statement = SELECT a.name FROM albums a INNER JOIN artists ar ON a.artist = ar._id WHERE ar.name = 'Iron Maiden' ORDER BY a.name ASC
Powerslave
Seventh Son Of A Seventh Son
The Number of the Beast
10:36:32.727 [main] INFO  be.pxl.ja2.jdbc.model.MusicDatasource - MusicDatasource closed.
```

QueryAlbumsForArtist.java

Query: counting rows

```
String sql = "SELECT COUNT(*) AS count FROM " + table;
try (Statement statement = conn.createStatement();
    ResultSet results = statement.executeQuery(sql)) {
  if (results.next()) {
     return results.getInt("count");
  }
  return -1;
} catch (SQLException e) {
  LOGGER.fatal("Executing query failed. ", e);
  return -1;
}
```

CountingRows.java

HOGESCHOOL PXL

SQL injection

```
Enter a title: Go Your Own Way
10:44:40.674 [main] INFO  be.pxl.ja2.jdbc.model.MusicDatasource - Query: SELECT
ar.name, al.name, s.track, s.title FROM albums al
  INNER JOIN artists ar ON ar._id = al.artist
  INNER JOIN songs s ON al._id = s.album
WHERE s.title = 'Go Your Own Way'

Enter a title: Go Your Own Way' or 1=1 or '
10:45:13.498 [main] INFO  be.pxl.ja2.jdbc.model.MusicDatasource - Query: SELECT
ar.name, al.name, s.track, s.title FROM albums al
  INNER JOIN artists ar ON ar._id = al.artist
  INNER JOIN songs s ON al._id = s.album
WHERE s.title = 'Go Your Own Way' or 1=1 or ''
```

QueryForSongInfo.java

Prepared Statements

```
public static final  String QUERY_SONG_INFO_PREP_STATEMENT =
     "SELECT ar.name, al.name, s.track, s.title FROM albums  al\n" +
     "  INNER JOIN artists  ar ON ar._id = al.artist \n" +
     "  INNER JOIN songs s ON al._id = s.album \n" +
     "WHERE s.title = ?";

PreparedStatement  querySongInfo  = conn.prepareStatement( QUERY_SONG_INFO_PREP_STATEMENT);

try {
  querySongInfo .setString(1, title);
  LOGGER.info("Query: " + querySongInfo .toString());
  ResultSet results  = querySongInfo .executeQuery();
  ...
} catch (SQLException e) {
  LOGGER.fatal("Executing query failed. " , e);
  return null;
}
```

QueryForSongInfo.java

HOGESCHOOL PXL

A **PreparedStatement** is a pre-compiled SQL statement. It is a subinterface of **Statement**. PreparedStatement objects have some useful additional features than Statement objects. Instead of hard coding queries, PreparedStatement object provides a feature to execute a parameterized query.

Advantages of PreparedStatement

- When PreparedStatement is created, the SQL query is passed as a parameter. This PreparedStatement contains a pre-compiled SQL query, so when the PreparedStatement is executed, DBMS can just run the query instead of first compiling it. So PreparedStatement are Faster for successive calls. How it works :-

1. Precompilation is done by the database. Some simpler databases don't precompile statements at all. Others might precompile it on the prepareStatement call, and yet others might do it when execute is first called on the statement, taking values of the parameters into account

1. when compiling the statement.
2. Databases that do precompile statements usually cache them, so in all probability the prepared statement won't be compiled again. Some JDBC drivers (eg. Oracle's) even cache prepared statements, so they haven't actually closed it when ps.close() was called.

- We can use the same PreparedStatement and with different parameters at the time of execution.
- An important advantage of PreparedStatements is that they prevent SQL injection attacks.

Bron:

https://www.geeksforgeeks.org/how-to-use-preparedstatement-in-java

Transactions

What is a Transaction?

Unit of work executed to retrieve, insert, remove and/or update data.

In a RDBMS, all transaction must be...

Atomic
Consitent
Isolated
Durable

https://www.youtube.com/watch?v=VRm2UMsFVz0

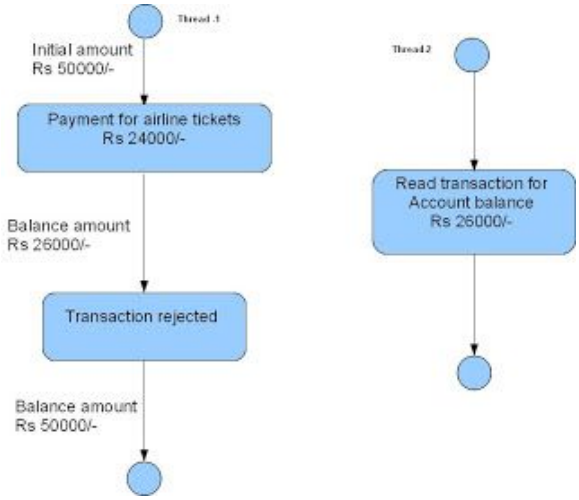
Isolation level

Isolated

The statements are executed in a seemingly sequential way.
(..depends on isolation level)

Transactions: Dirty read

Dirty read occurs wherein one transaction is changing the tuple/record, and a second transaction can read this tuple/record before the original change has been committed or rolled back. This is known as a dirty read scenario because there is always the possibility that the first transaction may rollback the change, resulting in the second transaction having read an invalid value.

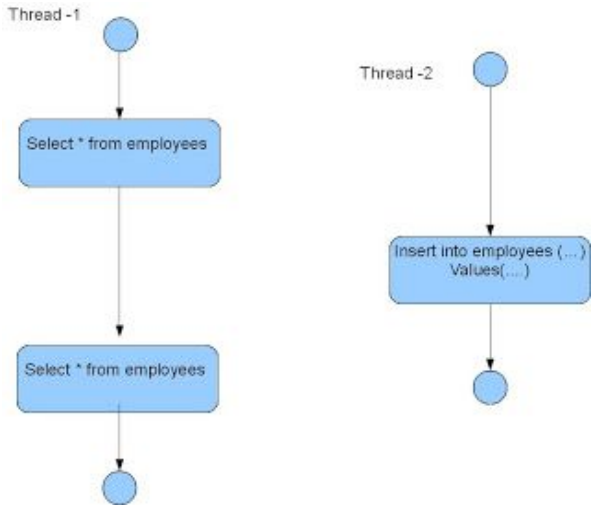


HOGESCHOOL PXL

http://javadata.blogspot.com/2011/05/dirty-readphantom-read-and-non.html

Transactions: Phantom read

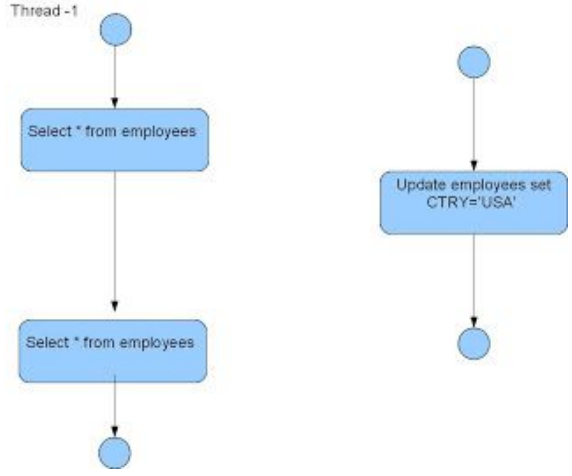
Phantom read occurs where in a transaction same query executes twice, and the second result set includes rows that weren't visible in the first result set. This situation is caused by another transaction inserting new rows between the execution of the two queries.



HOGESCHOOL PXL

Transactions: Non Repeatable Read

Non Repeatable Reads happen when in a same transaction same query yields different results. This happens when another transaction updates the data returned by other transaction.



HOGESCHOOL PXL

Transactions: Isolation level

Transaction Isolation Level

		Isolation Level			
		Read Uncommitted	Read Committed	Repeatable Read	Serializable
Problem Type	Dirty Read	Possible	Not Possible	Not Possible	Not Possible
	Nonrepeatable Read	Possible	Possible	Not Possible	Not Possible
	Phantom Read	Possible	Possible	Possible	Not Possible

Transactions

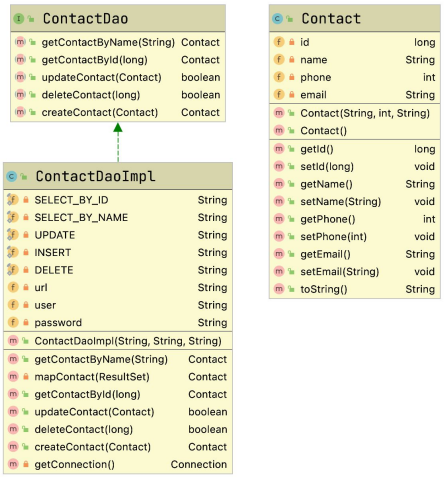
InsertSong
ReadUpdateThread

HOGESCHOOL PXL

DAO (Data Access Objects)

Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern.

- Data Access Object Interface - This interface defines the standard operations to be performed on a model object(s).
- Data Access Object concrete class - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
- Model Object or Value Object - This object is simple POJO containing get/set methods to store data retrieved using DAO class.



DEMO

UsingContactDao
ContactDaoImplTest

HOGESCHOOL PXL

https://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm