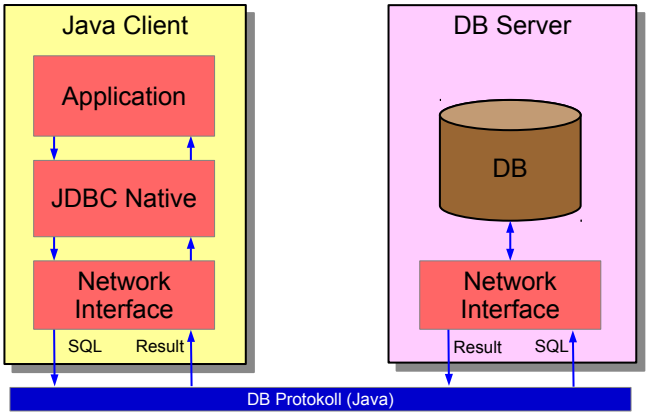
Name: _____
Klasse: _____
Datum: _____

notwendige Vorkenntnisse

- OOP-Grundlagen
- Datei-IO
- MariaDB ☞
- SQL ☞
- Maven ☞

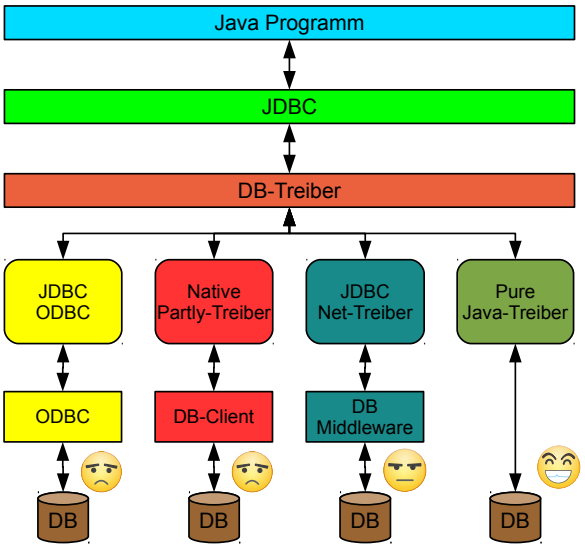
1 Java Database Connectivity



JDBC (Java Database Connectivity) ist eine Java Datenbankschnittstelle, die es ermöglicht, relationale Datenbanken verschiedener Hersteller über eine einheitliche Schnittstelle anzusprechen.

1.1 Schnittstelle

JDBC stellt eine Schnittstelle zur Anbindung an diverse Datenbanken zur Verfügung. Dabei wird die eigentliche Implementierung durch einen entsprechenden Treiber hergestellt.
☞JDBC-Tutorial ☞JDBC-FAQ



- **Typ 1: JDBC-ODBC bridge plus ODBC driver**:
  Die Verbindung zur Datenbank wird über ODBC hergestellt. Dabei stellt der JDBC-Treiber nur eine Kapselung der Aufrufe dar. Zusätzlich muss der ODBC-Treiber installiert werden.
  Wird nur noch in Ausnahmefällen benutzt.
- **Typ 2: Native-API partly-Java driver**:
  Die Verbindung wird über einen Datenbanktreiber hergestellt. Der JDBC-Treiber ruft die entsprechenden Datenbank-Client-Funktionen auf. Der DBMS-Client muss dabei installiert sein.
  Wird nur noch in Ausnahmefällen benutzt.
- **Typ 3: JDBC-Net pure Java driver**:
  Hier werden durch den JDBC-Treiber die Aufrufe in das entsprechende Protokoll übersetzt und dann von einem entsprechenden Server (middleware) in das DBMS-Protokoll übersetzt und an das DBMS geschickt.
  Wir bei sehr großen Systemen mit vielen Clients und meist einen oder mehreren „Zwischenservern“ benutzt.
- **Typ 4: Native-protocol pure Java driver**:
  Hier werden durch den JDBC-Treiber die Aufrufe direkt in das Netzwerkprotokoll des DBMS übersetzt. Hierzu muss aber das entsprechende proprietäre Protokoll des DBMS unterstützt werden.
  Der Standard!

1.2 Verbindungsaufbau

Importieren Sie das Eclipse-Projekt `eclipse_db_jdbc_01.zip`.
Voraussetzung: OpenJDK 21 (☞Installationsanleitung) und ein aktuelles Eclipse EE (☞Installationsanleitung), z. B. 2023-12 oder neuer.
Überprüfen Sie die Einstellungen beim Projekt: Properties – Java Build Path – Libraries – JRE System Library

Als Datenbank wird der Getränkemarkt (DB gm3) verwenden (siehe Unterrichtseinheit Datenbank).

Einbinden des JDBC-Treibers für MariaDB mit Mavan

```
<dependency>
   <groupId>org.mariadb.jdbc</groupId>
   <artifactId>mariadb-java-client</artifactId>
   <version>3.3.3</version>
</dependency>
```

Der Code

```
12       try {
13          final Connection con = DriverManager.getConnection(
14                "jdbc:mariadb://localhost:3306/gm3?allowMultiQueries=true",
15                "test", "");
16          System.out.println("... connected");
17
18          final DatabaseMetaData meta = con.getMetaData();
19          System.out.format("Driver : %s %s.%s\n", meta.getDriverName(),
20                meta.getDriverMajorVersion(), meta.getDriverMinorVersion());
21          System.out.format("DB     : %s %s.%s (%s)\n",
22                meta.getDatabaseProductName(), meta.getDatabaseMajorVersion(),
23                meta.getDatabaseMinorVersion(),
24                meta.getDatabaseProductVersion());
25          con.close();
26
27       } catch (final SQLException e) {
28          System.out.println("Fehler: " + e.getMessage());
29       }
```

- Die Verbindung wird dann über die Methode `getConnection()` hergestellt.
  Tritt dabei ein Fehler auf (Server nicht erreichbar, falsche Kennung, ...), so wird der Fehler `SQLException` geworfen.
- Dabei wird die DB-URI, die Kennung und das Passwort angegeben.

  ```
  jdbc:mysql://[host][,failoverhost...][:port]/[database]
     [?propertyName1][=propertyValue1][&propertyName2][=propertyValue2]...
  ```

  Ausführliche Beschreibung der Parameter/Properties:
  Die wichtigsten Parameter sind:

  `allowMultiQueries` Legt fest, dass mehrere SQL-Befehle in einem Aufruf verarbeitet werden können.

  `autocommit` Legt fest, ob alle Änderungen soft wirksam werden oder erst nach einen `commit` (Standard=true). Wird der Wert auf `false` gesetzt bzw. `con.setAutoCommit(false)` verwendet, so muss nach Änderungen ein `con.commit()` bzw. ein `con.rollback()` erfolgen.
  ☞A Guide to Auto-Commit in JDBC.

Ausgabe

Die Ausgabe sieht dabei wie folgt aus:

```
... connected
Driver : MariaDB Connector/J 3.3
DB     : MariaDB 11.2 (11.2.3-MariaDB-1:11.2.3+maria~ubu2204)
```


Aufgabe **JDBC-01-1**

Probieren Sie den Code aus. Achten Sie darauf, dass die Datenbank auch gestartet ist!

Notiz:

1.3 ACID

ACID beschreibt die geforderten Eigenschaften bei Datenbank-Transaktionen.

Aufgabe **JDBC-01-2**

Beschreiben Sie diese Eigenschaften mit eigenen Worten (ACID bei Wikipedia):

A Atomicity – Atomarität (Abgeschlossenheit)

C Consistency –Konsistenzerhaltung

I Isolation – Isolation (Abgrenzung)

D Durability – Dauerhaftigkeit

1.4 Verbindungsaufbau optimieren

Da Verbindungen zur Datenbank sehr oft durchgeführt werden, ist es sinnvoll, gewisse Funktionalität in eine eigene Klasse auszulagern. Dabei ist ist die Klasse selbst eine factory und hält eine Instanz der Datenbankverbindung, welche dann als singelton bezeichnet wird.

Dazu wird die Klasse `Util` mit statischen Methoden erstellt. Diese enthält folgende Methoden und ein Feld:

- `con`
  Hier wird die `Connection` gespeichert.

```
14 public class Util {
15
16    // singelton
17    private static Connection con = null;
```

- `getConnection(String db)`
  Hier wird die Verbindung zur Datenbank aufgebaut, wenn die Verbindung noch nicht hergestellt worden ist. Als Parameter wird der Name der Datenbank angegeben. Dieser wird verwendet, um die gleichnamige Properties-Datei (diese muss sich im root-Bereich des Projektes befinden) mit den Zugangsdaten zu laden.

```
30    // factory methode
31    public static Connection getConnection(final String db) {
32
33       if (con == null) {
34          try {
35             final Properties prop = new Properties();
36             prop.load(new FileReader(db + ".properties"));
37             final String dburl = prop.getProperty("DBURL");
38             final String dbuser = prop.getProperty("DBUSER");
39             final String dbpw = prop.getProperty("DBPW");
40
41             con = DriverManager.getConnection(dburl, dbuser, dbpw);
42          } catch (SQLException | IOException e) {
43             throw new RuntimeException(e);
44          }
45       }
46       return con;
47    }
```

- `close(AutoCloseable obj)`
  Hier wird bei allen Objekten, die `AutoCloseable` beinhalten, die Methode `close()` aufgerufen. Eventuelle Fehler werden dabei ignoriert.

```
19    // close
20    public static void close(final AutoCloseable obj) {
21       if (obj != null) {
22          try {
23             obj.close();
24          } catch (final Exception e) {
25             // ignore
26          }
27       }
28    }
```

- `Util()`
  Der Konstruktur wird auf `private` gesetzt. Somit kann keine Instanz der Klasse über `new` erzeugt werden, sondern nur über die factory-Methode `getConnection()`.

```
74    private Util() {
75    }
```


Der Inhalt der Properties-Datei (Name `gm3.properties`) sieht wie folgt aus:

```
DBURL=jdbc:mariadb://localhost:3306/gm3?allowMultiQueries=true
DBUSER=test
DBPW=
```

Der Aufruf in der java-Klasse sieht dabei wie folgt aus:

```
11      try {
12         final Connection con = Util.getConnection("gm3");
13         System.out.println("... connected");
14
15         final DatabaseMetaData meta = con.getMetaData();
16         System.out.format("Driver : %s %s.%s\n", meta.getDriverName(),
17               meta.getDriverMajorVersion(), meta.getDriverMinorVersion());
18         System.out.format("DB     : %s %s.%s (%s)\n",
19               meta.getDatabaseProductName(), meta.getDatabaseMajorVersion(),
20               meta.getDatabaseMinorVersion(),
21               meta.getDatabaseProductVersion());
22         Util.close(con);
23
24      } catch (final SQLException e) {
25         System.out.println("Fehler: " + e.getMessage());
26      }
```

Aufgabe **JDBC-01-3**

1. Beschreiben Sie in eigenen Worten, was eine `factory` mit einem `singelton` ist.

2. In der Methode `getConnection()` wird ein `try catch` Block verwendet. Welchen Vorteil hat es, dass die beiden Fehler `SQLException` und `IOException` in eine `RuntimeException` „umgewandelt“ werden?

3. Warum ist es wichtig, die DB-Verbindung mit `close()` wieder zu schließen?

1.5 Abfrage erstellen

```
Statement st       = con.createStatement();
ResultSet rs       = st.executeQuery(<SQL>);
int i              = st.executeUpdate(<SQL>);

boolean hasResults = st.execute(<SQL>);
ResultSet rs       = st.getResultSet();

st.addBatch(<SQL>);
int[] ii = st.executeBatch();
```

Die Klasse `Statement` dient dazu, einen SQL-Befehl aufzunehmen und eine entsprechende Abfrage an das DBMS zu schicken. Dabei wird zwischen folgenden Abfragetypen unterschieden:

Query Mit `executeQuery()` wird ein `SELECT`-Statement (oder ein ähnliches Statement wie beispielsweise `SHOW TABLES`) an die DB gesendet. Als Rückgabewert erhält man ein `ResultSet` mit den entsprechenden Daten.

Update Mit `executeUpdate()` wird ein `CREATE`, `DROP`, `INSERT`, `UPDATE` etc. an die DB gesendet. Als Rückgabewert erhält man die Anzahl an geänderten Datensätzen bzw. Tabellen.

beliebig Mit `execute()` kann jedes beliebige Statement an die DB gesendet werden. Als Rückgabewert erhält man die Information, ob ein `ResultSet` abgefragt werden kann, welches man mit der Methode `getResultSet()` erhält.

Batch Sollen viele SQL-Statements auf einmal ausgeführt werden, so können diese mit der Methode `addBatch()` einer Liste hinzugefügt und dann die Liste mit `executeBatch()` auf einmal ausgeführt werden.

Der Aufruf in der java-Klasse `Bsp_Select1` sieht dabei wie folgt aus:

```
12      try {
13         final Connection con = Util.getConnection("gm3");
14         System.out.println("... connected");
15
16         final Statement statement = con.createStatement();
17
18         final String sql1 = "desc lieferant";
19         final ResultSet rs = statement.executeQuery(sql1);
20
21         final ResultSetMetaData md = rs.getMetaData();
22         final int colcount = md.getColumnCount();
23         while (rs.next()) {
24            for (int i = 1; i <= colcount; i++) {
25               final Object obj = rs.getObject(i);
26               System.out.print(obj == null ? "null" : obj.toString());
27               System.out.print("\t");
28            }
29            System.out.println();
30         }
31
32         Util.close(statement);
33         Util.close(con);
34
35      } catch (final Exception e) {
36         System.out.println("Fehler: " + e.getMessage());
37      }
38   }
```


Information über das ResultSet

```
ResultSetMetaData meta   = rs.getMetaData();
int               colmax = meta.getColumnCount();
```

Über die Methode `getMetaData()` lassen sich zusätzliche Informationen über das Abfrageergebnis ermitteln. Zum Beispiel mit der Methode `getColumnCount()` die Anzahl an Spalten.

Ausgabe des ResultSets

```
while (rs.next()) {
   for (int i = 1; i <= colmax; ++i) {
      Object o = rs.getObject(i);
      System.out.print(o == null ? "null" : o);
      System.out.print("\t");
   }
   System.out.println();
}
```

Mit der Methode `next()` wird jeweils zum nächsten Datensatz gesprungen. Am Anfang steht dabei der Zeiger vor dem ersten Datensatz. Ist das Ende der Abfrage erreicht, liefert die Methode `false` zurück.

Über die Methode `getObject()` (bzw. `getString()`, `getInt()`, ...) erhält man den Inhalt der entsprechenden Spalte.

Hat ein Spaltenwert den Wert `NULL`, so liefert die Methode den `null`-Wert zurück. Dies muss bei der Weiterverarbeitung berücksichtigt werden.

Ausgabe

```
... connected
id int(11)  NO PRI   null  auto_increment
name  varchar(50) YES      null
strasse  varchar(50) YES      null
telefon  varchar(50) YES      null
email varchar(50) YES      null
oid   int(11)  YES   MUL   null
```

Die Ausgabe sieht jetzt nicht wirklich prikelnd aus.

Ausgabe mit dem Text-Table-Formatter

Mit der Bibliothek `text-table-formatter` lassen sich die Ausgaben deutlich schöner darstellen.

```
<dependency>
   <groupId>org.nocrala.tools.texttablefmt</groupId>
   <artifactId>text-table-formatter</artifactId>
   <version>1.2.4</version>
</dependency>
```

Dazu wird im Beispiel die Ausgabe mittels `printRs` aus der Klasse `Util` erledigt. Somit kann von überall zukünftig die Ausgabe ganz einfach relalisert werden.

```
11      try {
12         final Connection con = Util.getConnection("gm3");
13         System.out.println("... connected");
14
15         final Statement statement = con.createStatement();
16
17         final String sql1 = "desc lieferant";
18         final ResultSet rs = statement.executeQuery(sql1);
19
20         Util.printRs(rs);
21
22         Util.close(statement);
```


```
23          Util.close(con);
24
25       } catch (final Exception e) {
26          System.out.println("Fehler: " + e.getMessage());
27       }
```

Die Code der Methode sieht dabei wie folgt aus:

```
50    public static void printRs(final ResultSet rs) {
51
52       try {
53          final ResultSetMetaData rsmeta = rs.getMetaData();
54          final int cols = rsmeta.getColumnCount();
55          final Table t = new Table(cols);
56
57          for (int i = 1; i <= cols; i++) {
58             final String label = rsmeta.getColumnLabel(i);
59             t.addCell(label);
60          }
61
62          while (rs.next()) {
63             for (int i = 1; i <= cols; i++) {
64                final Object obj = rs.getObject(i);
65                t.addCell(obj == null ? "" : obj.toString());
66             }
67          }
68          System.out.println(t.render());
69       } catch (final SQLException e) {
70          throw new RuntimeException(e);
71       }
72    }
```

Ausgabe

Die Ausgabe sieht dabei deutlich schicker aus. Die Bibliothek bietet noch weitere Ausgabemöglichkeiten an, mehr dazu in der Dokumentation.

```
... connected
+-------+-----------+----+---+-------+--------------+
|Field  |Type       |Null|Key|Default|Extra         |
+-------+-----------+----+---+-------+--------------+
|id     |int(11)    |NO  |PRI|       |auto_increment|
|name   |varchar(50)|YES |   |       |              |
|strasse|varchar(50)|YES |   |       |              |
|telefon|varchar(50)|YES |   |       |              |
|email  |varchar(50)|YES |   |       |              |
|oid    |int(11)    |YES |MUL|       |              |
+-------+-----------+----+---+-------+--------------+
```

Aufgabe JDBC-01-4

1. Mit der Methoden `createStatement()` wird ein Objekt der Klasse `Statement` erzeugt. Welche Arten von SQL-Befehlen können Sie hier an die Datenbank senden?

2. Welche Vorteile ergeben sich, wenn man mehrere SQL-Befehle mit `addBatch()` in einem Stück an die Datenbank sendet?

ResultSets genauer betrachtet

Bisher wurde das `Statement` ohne weitere Angaben mit `con.createStatement()` verwendet.
Folgende Einstellungen sind hier möglich, die das `ResultSet` entsprechend beeinflussen.

`TYPE_FORWARD_ONLY`	Das `ResultSet` kann nicht gescrollt werden, der Cursor bewegt sich nur vorwärts.
`TYPE_SCROLL_INSENSITIVE`	Das `ResultSet` kann gescrollt, der Cursor kann sowohl nach vorne und hinten relativ zu der aktuellen Position bewegt werden. Es kann auf eine absolute Position gesprungen werden.
`TYPE_SCROLL_SENSITIVE`	Wie `TYPE_SCROLL_INSENSITIVE`, jedoch werden Änderungen am Datenbestand berücksichtigt.
`CONCUR_READ_ONLY`	Das `ResultSet` ist read only.
`CONCUR_UPDATABLE`	Das `ResultSet` nimmt Änderungen entgegen und leitet diese an die DB weiter.

Zu beachten ist, dass nicht alle JDBC-Treiber und DBMS obiges Verhalten unterstützen. Zur Sicherheit sollte hier beim Treiber nachgefragt werden.

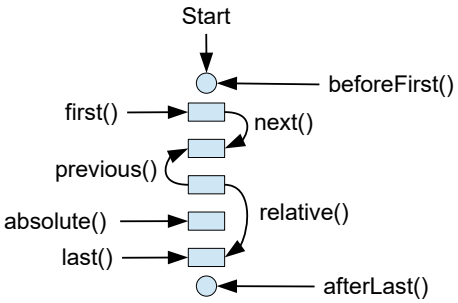
Der verwendete MariaDB-Treiber hat dabei folgende Möglichkeiten (Klasse `Bsp_Statement`).

```
11      try {
12         final Connection con = Util.getConnection("gm3");
13         System.out.println("... connected");
14
15         final DatabaseMetaData meta = con.getMetaData();
16         System.out.format("%s %s.%s (%s)\n", meta.getDatabaseProductName(),
17               meta.getDatabaseMajorVersion(), meta.getDatabaseMinorVersion(),
18               meta.getDatabaseProductVersion());
19         System.out.println("TYPE_FORWARD_ONLY       : "
20               + meta.supportsResultSetType(ResultSet.TYPE_FORWARD_ONLY));
21         System.out.println("TYPE_SCROLL_INSENSITIVE : "
22               + meta.supportsResultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE));
23         System.out.println("TYPE_SCROLL_SENSITIVE   : "
24               + meta.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE));
25         System.out.println("TYPE_FORWARD_ONLY       + CONCUR_READ_ONLY   : "
26               + meta.supportsResultSetConcurrency(ResultSet.TYPE_FORWARD_ONLY,
27                     ResultSet.CONCUR_READ_ONLY));
28         System.out.println("TYPE_FORWARD_ONLY       + CONCUR_UPDATABLE   : "
29               + meta.supportsResultSetConcurrency(ResultSet.TYPE_FORWARD_ONLY,
30                     ResultSet.CONCUR_UPDATABLE));
31         System.out.println("TYPE_SCROLL_INSENSITIVE + CONCUR_READ_ONLY   : "
32               + meta.supportsResultSetConcurrency(ResultSet.TYPE_SCROLL_INSENSITIVE,
33                     ResultSet.CONCUR_READ_ONLY));
34         System.out.println("TYPE_SCROLL_INSENSITIVE + CONCUR_UPDATABLE   : "
35               + meta.supportsResultSetConcurrency(ResultSet.TYPE_SCROLL_INSENSITIVE,
36                     ResultSet.CONCUR_UPDATABLE));
37         System.out.println("TYPE_SCROLL_SENSITIVE   + CONCUR_READ_ONLY   : "
38               + meta.supportsResultSetConcurrency(ResultSet.TYPE_SCROLL_SENSITIVE,
39                     ResultSet.CONCUR_READ_ONLY));
40         System.out.println("TYPE_SCROLL_SENSITIVE   + CONCUR_UPDATABLE   : "
41               + meta.supportsResultSetConcurrency(ResultSet.TYPE_SCROLL_SENSITIVE,
42                     ResultSet.CONCUR_UPDATABLE));
43         Util.close(con);
44      } catch (final Exception e) {
45         System.out.println("Fehler: " + e.getMessage());
46      }
47   }
```

Ausgabe

```
... connected
MariaDB 11.2 (11.2.3-MariaDB-1:11.2.3+maria~ubu2204)
TYPE_FORWARD_ONLY       : true
TYPE_SCROLL_INSENSITIVE : true
TYPE_SCROLL_SENSITIVE   : false
TYPE_FORWARD_ONLY       + CONCUR_READ_ONLY   : true
TYPE_FORWARD_ONLY       + CONCUR_UPDATABLE   : true
TYPE_SCROLL_INSENSITIVE + CONCUR_READ_ONLY   : true
TYPE_SCROLL_INSENSITIVE + CONCUR_UPDATABLE   : true
TYPE_SCROLL_SENSITIVE   + CONCUR_READ_ONLY   : false
TYPE_SCROLL_SENSITIVE   + CONCUR_UPDATABLE   : false
```


Cursor bewegen



Bei den ersten Beispielen wurde gezeigt, dass der Cursor mit `next()` um eine Position weiter gesetzt wird. Folgende Cursorbewegungen sind möglich:

`next()`	Der Cursor wird auf den nächsten Datensatz gesetzt. Liefert `false`, wenn der Cursor am Ende angekommen ist.
`previous()`	Der Cursor wird auf den vorherigen Datensatz gesetzt. Liefert `false`, wenn der Cursor am Anfang angekommen ist.
`first()`	Der Cursor wird auf den ersten Datensatz gesetzt. Liefert `false`, wenn keine Datensätze vorhanden sind.
`last()`	Der Cursor wird auf den letzten Datensatz gesetzt. Liefert `false`, wenn keine Datensätze vorhanden sind.
`beforeFirst()`	Der Cursor wird vor den ersten Datensatz gesetzt.
`afterLast()`	Der Cursor wird nach den letzten Datensatz gesetzt.
`relative()`	Der Cursor wird relativ zum aktuellen Datensatz bewegt. Liefert `false`, wenn der Cursor am Ende/am Anfang angekommen ist.
`absolute()`	Der Cursor wird auf den absoluten Datensatz gesetzt. Liefert `false`, wenn der Cursor am Ende/am Anfang angekommen ist.

ResultSet ändern

Wird ein `ResultSet` mit `TYPE_SCROLL_SENSITIVE` und `CONCUR_UPDATABLE` erzeugt, so lässt es sich über die Methoden `updateXXX` entsprechend ändern. Ein `updateRow()` sorgt dafür, dass die Änderung selbständig durchgeführt wird, ohne dass dabei ein `UPDATE`-Befehl erzeugt werden muss.

Als Beispiel wird hier die Tabelle `LIEFERANT2` verwendet.

```
CREATE OR REPLACE TABLE LIEFERANT2 (
   id INTEGER UNSIGNED PRIMARY KEY AUTO_INCREMENT,
   name VARCHAR(255) NOT NULL,
   updated TIMESTAMP DEFAULT {ts '2022-01-01 01:00:00'},
   created TIMESTAMP NULL,
   umsatz DECIMAL(7,2)
);
INSERT INTO LIEFERANT2 SET name='Niedermair';
INSERT INTO LIEFERANT2 SET name='Kobold';
INSERT INTO LIEFERANT2 SET name='Meier';
INSERT INTO LIEFERANT2 SET name='Zeus';
```

Aufgabe JDBC-01-5

Erstellen die die Tabelle `LIEFERANT2` in Java - siehe `Bsp_createLIEFERANT2`.

Datensätze ändern

```
   st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                            ResultSet.CONCUR_UPDATABLE);

   ResultSet rs = st.executeQuery("SELECT id, name, umsatz FROM LIEFERANT2");

   while (rs.next()) {
       double umsatz = rs.getDouble(3);
       rs.updateDouble(3, umsatz + 100);
       rs.updateRow();
   }
```

Aufgabe JDBC-01-6

Ändern Sie die Daten entsprechend in Java - siehe `Bsp_changeLIEFERANT2`.

Datensätze einfügen

```
   st = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                   ResultSet.CONCUR_UPDATABLE);
   ResultSet rs = st.executeQuery(
                  "SELECT id, name, umsatz FROM LIEFERANT2");

   rs.moveToInsertRow();
   rs.updateString("name", "Pumukel");
   rs.updateDouble("umsatz", 99.99);
   rs.insertRow();
```

Aufgabe JDBC-01-7

Fügen Sie die Daten entsprechend in Java ein - siehe `Bsp_insertLIEFERANT2`.

Datensätze löschen

```
st = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                ResultSet.CONCUR_UPDATABLE);
ResultSet rs = st.executeQuery(
               "SELECT id, name, umsatz FROM LIEFERANT2");

// letzten Datensatz loeschen
rs.last();
rs.deleteRow();
```

Aufgabe JDBC-01-8

Löschen Sie die Daten entsprechend in Java - siehe `Bsp_deleteLIEFERANT2`.

1.6 Abfragen sicherer machen – Prepared Statement

Wird ein SQL-Befehl an die Datenbank gesendet, so wird dieser zuerst auf korrekte Syntax geprüft, dann vom SQL-Parser „zerlegt“ und in einzelne Aufrufe aufgeteilt. Besser ist es, ein Grundgerüst zu verwenden, welches schon vorher übersetzt werden kann und dann nur noch die Werte entsprechend anzupassen.

Die Vorteile dabei sind:

- Der SQL-Befehl muss nur einmal geprüft und übersetzt werden.
- Ein Großteil der SQL-Injection[1] Angriffe laufen damit ins Leere.

Ein Prepared Statement wird mit der Methode `prepareStatement()` erzeugt, die ein Objekt der Klasse `PreparedStatement` zurückliefert.

```
PreparedStatement pst = connection.prepareStatement(
         "SELECT * from SPIELER WHERE Geschlecht=?");
```

Dynamische Parameter werden dabei mit dem ’?’ versehen. Dabei erhält das erste Fragezeichen die Nummer 1, das Zweite die Nummer 2, ...

Die eigentlichen Werte werden dabei über „setter“ festgelegt (Methode `setXXX()`), die dabei typspezifisch sind.

```
pst.setString(1, "weiblich");
ResultSet rs = pst.executeQuery();
```

Die wichtigsten Methoden sind (Auszug):

`clearParameters()`	Clears the current parameter values immediately.
`execute()`	Executes the SQL statement in this PreparedStatement object, which may be any kind of SQL statement.
`executeQuery()`	Executes the SQL query in this PreparedStatement object and returns the ResultSet object generated by the query.
`executeUpdate()`	Executes the SQL statement in this PreparedStatement object, which must be an SQL Data Manipulation Language (DML) statement, such as INSERT, UPDATE or DELETE; or an SQL statement that returns nothing, such as a DDL statement.
`getMetaData()`	Retrieves a ResultSetMetaData object that contains information about the columns of the ResultSet object that will be returned when this PreparedStatement object is executed.
`getParameterMetaData()`	Retrieves the number, types and properties of this PreparedStatement object’s parameters.

1 siehe https://de.wikipedia.org/wiki/SQL-Injection

`setArray()`	Sets the designated parameter to the given java.sql.Array object.
`setAsciiStream()`	Sets the designated parameter to the given input stream.
`setBigDecimal()`	Sets the designated parameter to the given java.math.BigDecimal value.
`setBinaryStream()`	Sets the designated parameter to the given input stream.
`setBlob()`	Sets the designated parameter to the given java.sql.Blob object.
`setBoolean()`	Sets the designated parameter to the given Java boolean value.
`setByte()`	Sets the designated parameter to the given Java byte value.
`setCharacterStream()`	Sets the designated parameter to the given Reader object.
`setClob()`	Sets the designated parameter to the given java.sql.Clob object.
`setDate()`	Sets the designated parameter to the given java.sql.Date value using the default time zone of the virtual machine that is running the application.
`setDouble()`	Sets the designated parameter to the given Java double value.
`setFloat()`	Sets the designated parameter to the given Java float value.
`setInt()`	Sets the designated parameter to the given Java int value.
`setLong()`	Sets the designated parameter to the given Java long value.
`setNCharacterStream()`	Sets the designated parameter to a Reader object.
`setNClob()`	Sets the designated parameter to a java.sql.NClob object.
`setNString()`	Sets the designated parameter to the given String object.
`setNull()`	Sets the designated parameter to SQL NULL.
`setObject()`	Sets the value of the designated parameter using the given object.
`setRef()`	Sets the designated parameter to the given REF(<structured-type>) value.
`setRowId()`	Sets the designated parameter to the given java.sql.RowId object.
`setShort()`	Sets the designated parameter to the given Java short value.
`setSQLXML()`	Sets the designated parameter to the given java.sql.SQLXML object.
`setString()`	Sets the designated parameter to the given Java String value.
`setTime()`	Sets the designated parameter to the given java.sql.Time value.
`setTimestamp()`	Sets the designated parameter to the given java.sql.Timestamp value.
`setURL()`	Sets the designated parameter to the given java.net.URL value.

Aufgabe **JDBC-01-9**

Geben Sie Daten entsprechend in Java mit einem Prepared Statement aus.
siehe `Bsp_PreparedStatementLIEFERANT2`