# JDBC

Dr. Heinz Schiffermüller


HTBLA-Kaindorf
Abteilung EDVO


Erstellung: März 2013
Letzte Änderungen: Dezember 2016

## Inhaltsverzeichnis

# 1 Einführung

## 1.1 Aufbau der Datenbankverbindung mit JDBC

Folgende Schritte sind notwendig um eine Java-Datenbankapplikation zu erstellen:

(1) Einbinden des Datenbank-Library in das NetBeans-Projekt:

Im Projektverzeichnis auf den Ordner "Libraries" klicken, das Kontextmenü öffnen und "Add library …" auswählen. Anschließend die gewünschte Library auswählen, z.B.:

- ☐ MySQL JDBC Driver
- ☐ PostgreSQL JDBC Driver

(2) Laden des Datenbank-Drivers in der Java-Applikation:

Erst durch das Laden des Datenbank-Drivers ist es möglich eine Verbindung zur Datenbank herzustellen:

```
try
{
  Class.forName("org.postgresql.Driver");
}
catch (ClassNotFoundException e)
{
  e.printStacktrace();
}
```

Seit Java 1.6 ist dieser Schritt optional. Das Laden erfolgt automatisch, unter der Voraussetzung dass die Library, wie in Schritt (1), eingebunden wurde.

(3) Aufbau einer Verbindung (Connection) zur Datenbank:

Der Aufbau einer Verbindung erfolgt mit der statischen Methode **`getConnection()`** der Klasse **`DriverManager`**:

```
try
{
  Connection con = DriverManager.getConnection(
       "jdbc:postgresql://localhost/mydb", "dbuser", "secret");
}
catch (SQLException e)
{
  e.printStacktrace();
}
```

Die drei Übergabeparameter haben dabei folgende Bedeutung:

- ☐ **`jdbc:postgresql://localhost/mydb`** besteht aus drei Teilen, getrennt durch einen ': '
  - **`jdbc`** – fixer Wert, der immer gleich bleibt
  - **`postgresql`** – Name des verwendete DBMS, z.B.: **`postgresql`**, **`mysql`**, **`oracle`**

- **//localhost/mydb** – definiert die URL zur Datenbank: Hostname (auch als IP-Adresse) gefolgt vom Namen der Datenbank zu der verbunden wird. Fehlt der Datenbankname so wird zur Standarddatenbank (z.B.: **postgres** oder **mysql**) verbunden.

- **dbuser**: ein Benutzername, der auf dem Datenbankserver eingerichtet ist
- **secret**: Passwort des Benutzers

Beenden einer Verbindung:

```
try
{
  con.close();
}
catch (SQLException e)
{
  e.printStacktrace();
}
```

# 2 Verwendung von SQL-Anweisungen

Für SQL-Abfragen und Manipulationen stehen in Java drei Interfaces zur Verfügung:

- Das Interface **`Statement`**: für normale SQL-Anweisungen
- Das Interface **`PreparedStatement`**: für SQL-Anweisungen die vorbereitet und wiederverwendet werden.
- Das Interface **`CallableStatement`**: für den Aufruf von Funktionen und Prozeduren, die in der Datenbank gespeichert sind (stored procedures).

Implementierende Instanzen dieser Interfaces werden über die Methoden **`createStatement()`**, **`prepareStatement()`** und **`prepareCall()`** des Connection-Objekts erzeugt. Die Klassen selbst sind in der eingebundenen Datenbank-Library definiert.

## 2.1 Verwendung von `Statements`

Objekte der Klasse **`Statement`** werden über das **`Connection`**-Objekt mit Hilfe der Methode **`createStatement()`** erzeugt. Das Statement-Objekt bietet drei verschiede Methoden zum Aufruf von SQL-Anweisungen:

- **`executeUpdate()`**
- **`executeQuery()`**
- **`execute()`**

### 2.1.1 Die Methode executeUpdate()

```
int executeUpdate(String sql) throws SQLException
```

Mit der Methode **`executeUpdate()`**, werden Aktionen wie **`INSERT`**, **`UPDATE`** und **`DELETE`**, aber auch **`CREATE TABLE`** und **`DROP TABLE`** ausgeführt.

```
public void useStatement() throws SQLException
{
  Statement stat = con.createStatement();
  String sqlString = "UPDATE language SET name = 'Chinese' "
           + "WHERE name = 'Mandarin';";
  int anz = stat.executeUpdate(sqlString);
  System.out.println(anz + " datasets changed");
  stat.close();
}
```

Die Methode **`executeUpdate()`** gibt die Anzahl der Datensätze zurück, die durch den SQL Befehl verändert wurden, bzw. -1 für ein Statement ohne Aktualisierungszähler. An die Methode kann auch ein String übergeben werden, der aus mehreren SQL-Befehlen besteht, die durch einen Strichpunkt getrennt sind.

### 2.1.2 Die Methode `executeQuery()`

```
ResultSet executeQuery(String sql) throws SQLException
```

Die Methode **`executeQuery()`** dient für Datenbankabfragen (SQL-Queries) und liefert ein Objekt der Klasse **`ResultSet`** zurück, das die Ergebnismenge der Datenbankabfrage in Form von Datensätzen enthält. Die Datensätze können sequentiell durchlaufen werden:

```
Statement stat = con.createStatement();
String sqlString = "SELECT * FROM film;";
ResultSet rs = stat.executeQuery(sqlString);
while (rs.next())
{
  String name = rs.getString("filmName");
  System.out.println(name);
}
stat.close();
```

Auf die Werte der einzelnen Spalten in einem Datensatz des **`ResultSets`** kann mit typenspezifischen getter-Methoden zugegriffen werden. Der Zugriff erfolgt entweder über den Spaltennamen, oder über den, bei 1 beginnendem Spaltenindex:

```
String name = rs.getString("filmName");
name = rs.getString(1);
int length = rs.getInt("length");
length = rs.getInt(2);
 ...
```

SQL-Datentypen und Java-Datentypen sind teilweise unterschiedlich. Die folgende Tabelle gibt die zu SQL äquivalenten Datentypen in Java an:

| *SQL-Datentyp* | *Java-Datentyp* | *Getter-Methode* |
|---|---|---|
| `INTEGER, INT` | `int` | `getInt()` |
| `SMALLINT` | `short` | `getShort()` |
| `NUMERIC, DECIMAL, DEC` | `java.math.BigDecimal` | `getBigDecimal()` |
| `FLOAT, DOUBLE` | `double` | `getdouble()` |
| `REAL` | `float` | `getFloat()` |
| `CHARACTER,CHAR,VARCHAR` | `String` | `getString()` |
| `BOOLEAN` | `boolean` | `getBooleaan()` |
| `DATE` | `java.sql.Date` | `getDate()` |
| `TIME` | `java.sql.Time` | `getTime()` |
| `TIMESTAMP` | `java.sql.Timestamp` | `getTimeStamp()` |
| `BLOB` | `java.sql.Blob` | `getBlob()` |
| `ARRAY` | `java.sql.Array` | `getArray()` |

Für alle Zeit- und Datums-Werte werden Objekte von Klassen aus dem package **`java.sql`** zurückgegeben, die sich von Objekten der Klassen aus dem package **`java.util`** unterscheiden. Hier müssen eventuell Konvertierungen durchgeführt werden.

Für die Konvertierung zwischen den Klassen **`java.sql.Date, java.sql.Time, java.sql.Timestamp`** auf der SQL-Seite und den Klassen **`java.time.LocalDate, java .time.LocalTime, java.time.LocalDateTime`** auf der Java-Seite, werden folgende Methoden verwendet:

```
LocalDate ld = LocalDate.now();
java.sql.Date date = java.sql.Date.valueOf(ld);
ld = date.toLocalDate();

LocalTime lt = LocalTime.now();
java.sql.Time time = java.sql.Time.valueOf(lt);
lt = time.toLocalTime();

LocalDateTime ldt = LocalDateTime.now();
java.sql.Timestamp timestamp = java.sql.Timestamp.valueOf(ldt);
ldt = timestamp.toLocalDateTime();
```

### 2.1.3 Die Methode `execute()`

```
boolean execute(String sql) throws SQLException
```

Die Methode **`execute()`** kann prinzipiell für mehrere SQL-Befehle verwendet werden, findet aber speziell dann Verwendung wenn mehrere **`ResultSet`**-Objekte von der Datenbank zurückgegeben werden. Die Befehle werden in einem String zusammengefasst und durch Strichpunkte getrennt.
Die Methode liefert **`true`** zurück wenn das erste Ergebnis ein **`ResultSet`** ist, **`false`** wenn ein Object-Count existiert oder keine Ergebnisse zurückkommen. Werden mehrere **`ResultSets`** zurückgegeben, können die Methoden **`getResultSet()`** und **`getMoreResults()`** dazu verwendet um auf das erste oder weitere **`ResultSets`** zuzugreifen. Die Methode **`getUpdateCount()`** liefert die Anzahl der geänderten Datensätze zurück, bzw. -1 wenn ein **`ResultSet`** zurückkommt oder kein Update auf der Datenbank ausgeführt wurde.

## 2.2 Verwendung von `PreparedStatements`

**`PreparedStatement`**–Objekte werden, ebenso wie **`Statements`**, über die Connection aber mit Hilfe der Methode **`prepareStatement()`** erzeugt. Ein **`PreparedStatement`** enthält ein SQL-Statement in vorkompilierter Form und wird nur einmal erzeugt, wobei ein oder mehrere '?' als Platzhalter im SQL-Statement eingesetzt werden können. An diesen Positionen werden anschließend konkrete Werte über setter-Methoden in das Statement eingesetzt. **`PreparedStatements`** sind besonders dann effizient, wenn eine SQL-Anweisung mehrfach, mit verschiedenen Werten verwendet wird.
Die Verwendung von **`PreparedStatements`** ist meist sehr einfach, da im Vergleich zu **`Statements`** umständliche String-Manipulationen wegfallen, wie z.B. bei INSERT-Anweisungen.

Das folgende Beispiel zeigt die Verwendung eines **`PreparedStatements`**:

```
private PreparedStatement actorInsertStmt;
private static final String actorInsert =
     "INSERT INTO actor (first_name, last_name) VALUES ( ? , ? ); ";

public List<String> insertActor(String firstName, String lastName)
                                                        throws SQLException
{
  if (actorInsertStmt == null)
  {
    actorInsertStmt = con.prepareStatement(actorInsert);
  }
  actorInsertStmt.setString(1, firstName);
  actorInsertStmt.setString(2, lastName);
  actorInsertStmt.executeUpdate();
    . . .
}
```

Die einfachen Hochkommata werden, wie das z.B. beim Einsetzen von Strings notwendig ist, automatisch eingefügt.
Für **`PreparedStatements`** werden ebenfalls die Methoden **`execute()`**, **`executeQuery()`** und **`executeUpdate()`** verwendet.

## 2.3 Verwendung von `CallableStatements`

**`CallableStatements`** werden über die Connection mit Hilfe der Methode **`prepareCall()`** erzeugt. Ein **`CallableStatement`** ruft eine Stored Procedure oder eine Stored Function in der Datenbank auf.
Im folgenden Beispiel muss die Funktion **`film_in_stock()`** in der Datenbank definiert sein. Das '?' dient wieder als Platzhalter, diesmal für die Übergabeparameter an die Funktion:

```
private CallableStatement filmInStockStmt;
private static final String filmInStock = "{call film_in_stock( ? , ? )}";

public List<String> getFilmsInStock(int film_id, int store_id)
                                        throws SQLException
{
  . . .
  if (filmInStockStmt == null){
    filmInStockStmt = con.prepareCall(filmInStock);
  }
  filmInStockStmt.setInt(1, film_id);
  filmInStockStmt.setInt(2, store_id);

  ResultSet rs = filmInStockStmt.executeQuery();
    . . .
}
```

In diesem Beispiel werden zwei IN-Parameter, zur Übergabe von Werten an die Stored-Procedure verwendet. Es besteht auch die Möglichkeit OUT-Parameter zu definieren um von der Stored-Procedure Werte zurückzubekommen wie das folgende Beispiel zeigt:

```
private CallableStatement someStmt;
private static final String someCall = "{call func( ? )}";

public void callSomeStmt()throws SQLException
{
  . . .
  if (someStmt == null){
    someStmt = con.prepareCall(someCall);
  }
  someStmt.registerOutParameter(1, java.sql.Types.INTEGER);
  someStmt.execute();
  int anz = someStmt.getInt(1);
  . . .
}
```

Bei Verwendung von OUT-Parameter, muss mit der Methode **`registerOutParameter()`** eine Deklaration mit dem richtigen SQL-Typ erfolgen. Der zurückgelieferte Wert kann mit getter-Methoden abgefragt werden.
Liefert des **`CallableStatement`** ein **`ResultSet`** zurück so darf der OUT-Parameter erst nach Durchlaufen des **`ResultSet`** abgefragt werden.

## 2.4 Verwendung von Views

Wenn Views in der DB definiert sind erfolgt der Zugriff von Java gleich wie auf Tabellen, wobei an Stelle des Tabellennamens der Name der View verwendet wird.

## 2.5 Metadaten

Um weitere Informationen über eine Tabelle und deren Struktur oder über die Datenbank zu bekommen stehen zwei Klassen zur Verfügung:

```
ResultSetMetaData rsmd = rs.getMetaData();   // Metadaten zu einer Query
                                             // Aufgerufen über ein ResultSet
DatabaseMetaData dbmd = con.getMetaData();   // Metadaten zu einer DB
                                             // Aufgerufen über eine Connection
```

### 2.5.1 Die Klasse `ResultSetMetaData`

Einige Methoden der Klasse **`ResultSetMetaData`** sind:

```
ResultSetMetaData rsmd = rs.getMetaData();
int colCnt = rsmd.getColumnCount();       // Anzahl der Spalten
String colName = rsmd.getColumnName(1);   // Name der 1.Spalte
int colSize = rsmd.getColumnDisplaySize(1);
                            // max.Anzahl an Zeichen der 1.Spalte
int colType = rsmd.getColumnType(1);
                            //Spaltentyp als java.sql.Types Konstamte
boolean b = rsmd.isAutoIncrement(1);      // Ob 1.Spalte autoinkrement ist
```

Um die Anzahl der Zeilen eines **`ResultSets`** zu ermitteln gibt es keine eigene Methode, es kann aber folgender Workarround verwendet werden:

```
rs.last();
int rows = rs.getRow();
rs.beforeFirst();
```

Zu diesem Zweck muss der Datensatz-Cursor scrollbar sein, d.h. das **`ResultSet`** muss in beide Richtungen laufen können. Das muss beim Erzeugen des **`Statement`**- oder **`PreparedStatement`**-Objekts eingestellt werden:

```
Statement stat = con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
                                      ResultSet.CONCUR_READ_ONLY);
```

### 2.5.2 Die Klasse `DataBaseMetaData`

Zum Abruf von Datenbank-Metadaten gibt es mehrere Methoden:

```
DatabaseMetaData dbmd = con.getMetaData();
System.out.println("Driver: " + dbmd.getDriverName()
                + "-" + dbmd.getDriverVersion());
ResultSet rs = dbmd.getPrimaryKeys(null, null, "film_actor");
while (rs.next())
{
  System.out.println(rs.getString("column_name"));
}
```

### 2.5.3 Metadaten über die Datenbank abfragen

Das Information-Schema bietet in postgres die Möglichkeit eine Reihe von Metadaten über die Datenbank, zu der die **`Connection`** besteht, abzufragen.
Query um alle Tabellennamen und Views sowie den dazugehörige Tabellentyp (BASE_TABLE oder VIEW) auszugeben:

```
String sql = "SELECT * FROM information_schema.tables "
          + "WHERE table_schema = 'public';";
ResultSet rs = stat.executeQuery(sql);
while (rs.next())
{
  String name = rs.getString("table_name");
  String type = rs.getString("table_type");
  System.out.println(name+ " " +type);
}
```

Query um alle Spaltennamen, Spaltenbreiten und Datentypen einer Tabelle auszugeben:

```
String sql = "SELECT column_name, character_maximum_length, data_type "
           + "FROM information_schema.columns "
           + "WHERE table_name = 'film'";
ResultSet rs = stat.executeQuery(sql);
while (rs.next())
{
  String colName = rs.getString("column_name");
  int maxLen = rs.getInt("character_maximum_length");
  String type = rs.getString("data_type");
  System.out.println(colName + " - " + maxLen+ " - " + type);
}
```

Query um den PrimaryKey einer Tabelle auszugeben:

```
String sql = "SELECT column_name FROM "
           + "FROM information_schema.key_column_usage "
           + "WHERE table_name = 'film'";
ResultSet rs = stat.executeQuery(sql);
while (rs.next())
{
  String colName = rs.getString("column_name");
  System.out.println("PrimaryKey: " + colName);
}
```

## 2.6 Scrollbare und aktualisierbare `ResultSets`

Seit JDBC 2 werden scrollbare und aktualisierbare **`ResultSets`** unterstützt.
Scrollbare **`ResultSets`** ermöglichen es, das **`ResultSet`** vorwärts und rückwärts zu durchlaufen, oder an eine beliebige Position im **`ResultSet`** zu springen.
Aktualisierbare **`ResultSets`** ermöglichen es, Werte im **`ResultSet`** zu ändern, wobei die Änderungen automatisch in der Datenbank aktualisiert werden.

Die Objekte der Klassen **`Statements`** oder **`PreparedStatements`** müssen dazu wie folgt erzeugt werden:

```
Statement stat = con.createStatement(type, concurrency);

PreparedStatement stat = con.prepareStatement(command, type, concurrency);
```

Der Parameter ***`command`*** im **`PreparedStatement`** enthält den SQL-Befehl.

Der Parameter ***`type`*** ermöglicht das Scrollen und kann auf folgende Konstanten der Klasse **`ResultSet`** gesetzt werden:

| ***Wert:*** | ***Bedeutung*** |
|---|---|
| **`TYPE_FORWARD_ONLY`** | Das **`ResultSet`** ist nicht scrollbar |
| **`TYPE_SCROLL_INSENSITVE`** | Das **`ResultSet`** ist scrollbar und nicht sensitiv für Datenbankänderungen |
| **`TYPE_SCROLL_SENSITVE`** | Das **`ResultSet`** ist scrollbar und sensitiv für Datenbankänderungen |

Sensitiv für Datenbankänderungen, bedeutet, dass eine Änderung in der Datenbank, z.B. durch andere User, sofort im **`ResultSet`** sichtbar ist.

Für den Parameter ***`concurrency`***, der die Aktualisierung steuert, können folgende Konstanten verwendet werden:

| ***Wert:*** | ***Bedeutung*** |
|---|---|
| **`CONCUR_READ_ONLY`** | Das **`ResultSet`** kann nicht für die Aktualisierung der DB verwendet werden |
| **`CONCUR_UPDATABLE`** | Das **`ResultSet`** kann für die Aktualisierung der DB verwendet werden |

Allerdings ist nicht jede Datenbank scrollbar oder aktualisierbar. Die Methoden **`supportsResultSetType()`** und **`supportsResultSetConcurrency()`** der Klasse **`DatabaseMetaData`** geben darüber Aufschluss.

### 2.6.1 Scrollbare `ResultSets`

Zum Scrollen des **`ResultSets`** können folgende Methoden verwendet werden:

**`boolean previous()`**
Bewegt den Cursor zur vorherigen Zeile dieses **`ResultSets`**.

Liefert **false** wenn der Cursor vor dem ersten Datensatz steht, sonst **true**.

**boolean absolute(int row)**
Bewegt den Cursor zur Zeile **row** dieses **ResultSets**.
Rückgabewert **true** wenn der Cursor an dieser Position ist, **false** wenn er vor dem ersten oder hinter dem letzten Datensatz ist.

**boolean relative(int rows)**
Bewegt den Cursor um **rows** Zeilen relativ zu aktuellen Position, entweder vorwärts (**rows** positiv) oder rückwärts (**rows** negativ).
Rückgabewert **true** wenn der Cursor an einer gültigen Position ist, sonst **false**.

**int getRow()**
Liefert die aktuelle Cursorposiiton (=Zeilennummer) zurück. Die Zeilennummerierung beginnt bei 1.

**void first()**
Bewegt den Cursor zu ersten Zeile.

**void last()**
Bewegt den Cursor zu letzten Zeile.

**void beforeFirst()**
Bewegt den Cursor vor die erste Zeile.

**void afterLast()**
Bewegt den Cursor hinter die letzte Zeile

**booelan isFirst()**
**true** wenn der Cursor in der ersten Zeile ist.

**booelan isLast()**
**true** wenn der Cursor in der letzten Zeile ist.

**booelan isBeforeFirst()**
**true** wenn der Cursor vor der ersten Zeile ist.

**booelan isAfterLast()**
**true** wenn der Cursor hinter der letzten Zeile ist.

### 2.6.2 Aktualisierbare `ResultSets`

Nicht jedes **ResultSet** ist aktualisierbar, daher kann die Methode **getConcurrency()** der Klasse **ResultSet** verwendet werden. Diese Methode gibt einen int-Wert zurück, der über die Konstanten **ResultSet.CONCUR_READ_ONLY** oder **ResultSet.CONCUR_UPDATABLE** abgefragt werden kann.
Bezieht sich eine SQL-Abfrage nur auf eine einzelne Tabelle oder auf mehrere Tabellen, die über die Primärschlüssel verknüpft sind, so ist das **ResultSet** meist aktualisierbar.

Das Aktualisieren ist immer nur für den gerade aktuellen Datensatz, mit Hilfe der **updateXxx()**-Methoden möglich. In diesen Methoden kann entweder mit dem Spaltenindex oder den Spaltennamen auf das jeweilige Element zugegriffen werden. Die Aktualisierung der Daten in der Datenbank erfolgt erst nach Aufruf der Methode **updateRow()**.

```
String sqlQuery = "SELECT * FROM film";
Statement stat = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
          ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stat.executeQuery(sqlQuery);
 ...
if (rs.absolute(5))
{
  rs.updateInt("release_year", 2007);
  rs.updateRow();
}
 ...
```

Aktualisierungen können mit der Methode **cancelRowUpdates()** abgebrochen werden.

Um einen vollständigen neuen Datensatz in das **ResultSet** einzufügen, wird zuerst die Methode **moveToInsertRow()** verwendet um zur Einfügezeile zu gelangen. Anschießend werden mit den **updateXxx()**-Methoden Werte in diesen Datensatz eingefügt. Durch Aufruf der Methode **insertRow()** wird der neue Datensatz in die Datenbank eingefügt. Mit **moveToCurrentRow()** rückt der Cursor wieder zurück zur ursprünglichen Position, vor dem Aufruf von **moveToInsertRow()**.

```
rs.moveToInsertRow();                       // zur Einfügezeile gehen
rs.updateString("title", "AGENT TRUMAN");   // Werte einfügen
 ...
rs.insertRow();                             // Zeile in DB einfügen
rs.moveToCurrentRow();                      // zurück zur Cursorposition
```

Mit der Methode **deleteRow()** wird die Zeile an der aktuellen Cursorposition gelöscht.

***Hinweis:***
Alle gerade gezeigt Methoden können durch die SQL-Anweisungen **UPDATE**, **INSERT** und **DELETE** ersetzt werden. Es sollte, besonders in Hinblick auf die Performance, überlegt werden ob aktualisierbare **ResultSets** oder SQL-Anweisungen verwendet werden.

## 2.7 SQL-Exceptions

Neben den bekannten Exception Methoden wie **`toString()`**, **`getMessage()`** und **`printStacktrace()`** existieren für SQL-Exceptions noch weitere Methoden:

- **`String getSQLState()`**
  Gibt einen 5-stelligen SQL-Fehlercode zurück
- **`int getErrorCode()`**
  Gibt den Herstellerspezifischen ErrorCode zurück
- **`SQLException getNextException()`**
  Liefert die nächste Exception in der Exception-Kette

# 3 Struktur einer Desktop-DB-Applikation

Die Klasse **DataBase** wird als Singleton implementiert:

```
private DataBase() throws FileNotFoundException,
                          IOException, ClassNotFoundException, SQLException
{
  loadProperties();             // Laden der DB-Properties
  Class.forName(DB_driver);     // Laden des DB-Drivers - optional!
  connect();                    // Connecten zur DB
}
```

In der Methode **connect()** wird eine Verbindung (=Connection) zur Datenbank aufgebaut. URL, Username und Passwort sind entsprechend zu definieren.

```
public void connect() throws SQLException
{
  con = DriverManager.getConnection(DB_url, DB_username, DB_password);
  cc = new DBCachedConnection(con);
}
```

In der Methode **loadProperties()** werden die wichtigsten Datenbank-Eigenschaften aus einer Property-Datei geladen:

```
private void loadProperties() throws FileNotFoundException, IOException
{
  Properties props = new Properties();
  FileInputStream fis = new FileInputStream(property_filename);
  props.load(fis);
  this.DB_url = props.getProperty("DB_url");
  this.DB_username = props.getProperty("DB_user");
  this.DB_password = props.getProperty("DB_password");
  this.DB_driver = props.getProperty("DB_driver");
}
```

Weitere Methoden:

```
getStatement()              // Statement aus CachedConnection holen

releaseStatement()          // Statement in CachedConnection zurückgeben

getConnection()             // getter-Methode für die Connection zur DB
```

Klasse **CachedConnection**

Enthält eine **Queue<Statement>** von Statement Objekten, sowie die Methoden:

```
getStatement()              // Statement aus CachedConnection holen

releaseStatement()          // Statement in CachedConnection zurückgeben
```

Klasse **DBAccess**

Enthält Methoden um den eigentlichen Datenbankzugriff für die Applikation zu implementieren.

Zugriff auf die DB entweder über Statements aus der **CachedConnection** oder über die **Connection** mit Hilfe von **Prepared**- und **CallableStatements**.

Datei **database.properties**

Properties-Datei mit allen DB-spezifischen Eigenschaften:

```
DB_url = jdbc:postgresql://localhost/sakila
DB_user = postgres
DB_password = postgres
DB_driver = org.postgresql.Driver
```

# 4 Struktur einer Web-DB-Applikation

Bei Desktop-Applikationen ist es ausreichend eine Comnection zur Datenbank aufzubauen. Bei Web-Applikationen hingegen ist es notwendig jeden Benutzer eine eigene Connection zur Verfügung zu stellen um einen ausreichend schnellen Zugriff zu gewährleisten. Die Struktur der Datenbankanbindung muss daher auf Connection-Pooling umgestellt werden, wobei folgende Klassen zum Einsatz kommen:

- **DB_Config** als Interface mit den notwendigen Verbindungs-Parametern zur DB
- **DB_ConnectionPool** als Klasse zur Verwaltung des Connection-Pools
- **DB_PStatPool** als Klasse zur Verwaltung aller PreparedSatatements
- **DB_Stmt_Type** Enum zur Verwaltung der Statement-Typen und SQL-Strings
- **DB_Access** Klasse mit der Schnittstellendefinition zur Datenbank

Das Interface **DB_Config**:

```
public interface DB_Config {
  public static final String DB_NAME = "booksdb";
  public static final String DB_USER = "postgres";
  public static final String DB_PASSWD = "postgres";
  public static final String DB_URL = "jdbc:postgresql://localhost/";
  public static final String DB_DRIVER = "org.postgresql.Driver";
}
```

Das Klasse **DB_ConnectionPool**, implementiert als Singleton, verwaltet alle Connections in einem Pool:

```
public class DB_ConnectionPool implements DB_Config {

  private LinkedList<Connection> connections = new LinkedList<>();
  private final int POOL_SIZE = 150;
  private int numCon = 0;

  private static DB_ConnectionPool theInstance = null;

  public static DB_ConnectionPool getInstance() {
    if (theInstance == null) {
      theInstance = new DB_ConnectionPool();
    }
    return theInstance;
  }

  private DB_ConnectionPool() {
    try {
      Class.forName(DB_DRIVER);
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
      throw new RuntimeException(e.toString());
    }
  }
```

```
  public synchronized Connection getConnection() throws SQLException  {
    if (connections.isEmpty()) {
      if (numCon == POOL_SIZE) {
        throw new RuntimeException"connection-limit reached - try later");
      }
      Connection con = DriverManager.getConnection(DB_URL + DB_NAME,
              DB_USER,
              DB_PASSWD);
      numCon++;
      return con;
    } else {
      return connections.poll();
    }
  }

  public synchronized void releaseConnection(Connection conn) {
    connections.offer(conn);
  }
}
```

Die Bereitstellung aller **PreparedStatement**-Objekte erfolgt in der Klasse **DB_PStatPool** , auch als Singleton implementiert. Da jedes **PreparedStatement** über eine **Connection** erzeugt wird, kann es auch nur für diese **Connection** wieder-verwendet werden. Die Datenstruktur die hier zum Einsatz kommt ist eine **Map**, die als Key die jeweilige **Connection** verwendet. Der Value dieser **Map** ist wieder eine **Map** die für den Zugriff auf die **PreparedStaement**–Objekte als Key die Enum **DB_StatementType** benutzt:

```
public class DB_PStatPool  {

  private static DB_PStatPool theInstance;
  private final DB_ConnectionPool connectionPool =
                        DB_ConnectionPool.getInstance();

  public static DB_PStatPool  getInstance() {
    if (theInstance == null) {
      theInstance = new DB_PStatPool ();
    }
    return theInstance;
  }

  private Map<Connection, Map<DB_StatementType, PreparedStatement>> pStatMap
          = new HashMap<>();

  private DB_PStatPool () {
  }

  public synchronized PreparedStatement
  getPreparedStatement(DB_StatementType pStatType, Connection connection)
        throws SQLException {
    Map<DB_StatementType,PreparedStatement> connMap = pStatMap.get(connection);
    if (connMap == null) {
      connMap = new HashMap<>();
      pStatMap.put(connection, connMap);
    }
    PreparedStatement pStat = connMap.get(pStatType);
    if (pStat == null) {
      pStat = connection.prepareStatement(pStatType.getSqlString());
```

```
      connMap.put(pStatType, pStat);
    }
    return pStat;
  }
}
```

Die einzelnen SQL-Strings für die **PreparedStatements** werden in der Enumeration **DB_StatementType** gespeichert:

```
public enum DB_StatementType {
  GetBooksByTitle("SELECT * FROM books WHERE UPPER(title) LIKE UPPER( ? )"),
  GetBooksByAuthor("SELECT ...");

  private String sqlString;

  private DB_StatementType(String sqlString) {
    this.sqlString = sqlString;
  }

  public String getSqlString() {
    return sqlString;
  }
}
```

Das Klasse **DB_Access** dient als Schnittstelle zwischen von der Web-Anwendung zur Datenbank und wird ebenfalls als Singleton implementiert. **Statement**-Objekte werden über den ConnectionPool realisiert, auf **PreparedStatement**-Objekte wird über den ConnectionPool und den StatementPool zugegriffen. Alle Connection-Objekte müssen mit der Methode **releaseConnection()** an den Pool zurückgegeben werden.

```
public class DB_Access {
  private final DB_ConnectionPool connectionPool =
                            DB_ConnectionPool.getInstance();
  private final DB_PStatPool  statementPool =
                             DB_PStatPool .getInstance();
  private static DB_Access theInstance = null;

  public static DB_Access getInstance() throws RuntimeException {
    if (theInstance == null) {
      theInstance = new DB_Access();
    }
    return theInstance;
  }

  private DB_Access()

  /**
   * Usage of a Prepared Statement to find all books that match a given
   * part of a title
   *
   * @param title part of a title
   * @return List of books found in database
   * @throws SQLException
   */
  public List<Book> selectBooksByTitle(String title) throws SQLException {
    List<Book> books = new ArrayList<>();
    Connection connection = connectionPool.getConnection();
```

```
    PreparedStatement pStat =
      statementPool.getPreparedStatement(DB_StatementType.GetBooksByTitle,
            connection);
    pStat.setString(1, "%" + title + "%");
    ResultSet rs = pStat.executeQuery();
    while (rs.next()) {
      books.add(new Book(rs.getString("title"),
              rs.getString("isbn"),
              rs.getInt("publisher_id"),
              rs.getString("url"),
              rs.getDouble("price")));
    }
    connectionPool.releaseConnection(connection);
    return books;
  }

  /**
   * Usage of a Statement to find all books in the database
   *
   * @returnList of all books found in database
   * @throws SQLException
   */
  public List<Book> getAllBooks() throws SQLException {
    List<Book> books = new ArrayList<>();
    Connection connection = connectionPool.getConnection();
    String sqlString = "SELECT * FROM books";
    Statement statement = connection.createStatement();
    ResultSet rs = statement.executeQuery(sqlString);
    while (rs.next()) {
      books.add(new Book(rs.getString("title"),
              rs.getString("isbn"),
              rs.getInt("publisher_id"),
              rs.getString("url"),
              rs.getDouble("price")));
    }
    connectionPool.releaseConnection(connection);
    return books;
  }
}
```