3 Persistenz mit JDBC

In diesem Kapitel schauen wir uns mit JDBC eine Variante der Persistenz mit Java an. In Abschnitt 3.1 lernen wir das ***JDBC***-API (***Java Database Connectivity***) zur Verarbeitung von Daten mit relationalen Datenbanken (***RDBMS***) kennen. Die Programmierschnittstelle JDBC versteckt die technischen Details des Datenbankzugriffs vor dem Entwickler. Sowohl das Erzeugen von SQL-Kommandos als auch die Auswertung der von der Datenbank gelieferten Ergebnisse geschieht mithilfe von Klassen und Interfaces des JDBC-APIs.

Bei der Verarbeitung der Daten gibt es oftmals noch das Problem zu lösen, wie man Objekte mit ihren Beziehungen in den Tabellen einer relationalen Datenbank speichern kann. Dazu benötigt man eine Abbildung von Objekten in Datenbanken, das sogenannte ***Object-Relational Mapping*** (***ORM***). Wie man dieses selbst programmieren kann und welche Herausforderungen dabei zu meistern sind, beschreibt Abschnitt 3.2.

Alternativ zu JDBC kann man zur Persistierung das ***JPA*** (***Java Persistence API***) einsetzen. JPA erleichtert das ORM, weil eine noch stärkere Abstraktion als bei selbst erstellten ORM-Umsetzungen stattfindet: Zur Implementierung von Datenbankzugriffen werden in JPA keine SQL-Anweisungen mehr geschrieben, sondern automatisch durch JPA generiert. Dabei werden Besonderheiten verschiedener Datenbanksysteme (z. B. Unterschiede im unterstützten SQL-Befehlssatz) beachtet und durch JPA verborgen, sodass sie für uns als Entwickler keine Rolle mehr spielen. Darauf gehe ich in Kapitel 4 separat ein.

3.1 Datenbankzugriffe per JDBC

In diesem Abschnitt werden wir das ***JDBC*** zur Verarbeitung von Daten mit relationalen Datenbanken einsetzen.

Datenbankanbindung und Datenbanktreiber

Mit dem JDBC-API lassen sich unterschiedliche Datenbanken einheitlich ansprechen und die konkreten Details des Zugriffs vor dem Benutzer verstecken. Somit wird das Java-Programm von der Hard- und Software des RDBMS weitgehend unabhängig. Um aus Java auf ein RDBMS zuzugreifen, ist es nur notwendig, den Datenbanktreiber, bestehend aus bestimmten Klassen (häufig als jar-Dateien gebündelt), in den Classpath der eigenen Applikation aufzunehmen.

Die Anbindung eines Java-Programms an ein RDBMS erfolgt dann durch proprietäre Datenbanktreiber, die man vom jeweiligen Datenbankhersteller erhält. Diese Treiber realisieren die durch das JDBC-API vorgegebene Schnittstelle. Abbildung 3-1 zeigt das Zusammenspiel zwischen Java-Programm, JDBC-API, Datenbanktreiber und RDBMS, hier für Oracle und MySQL.

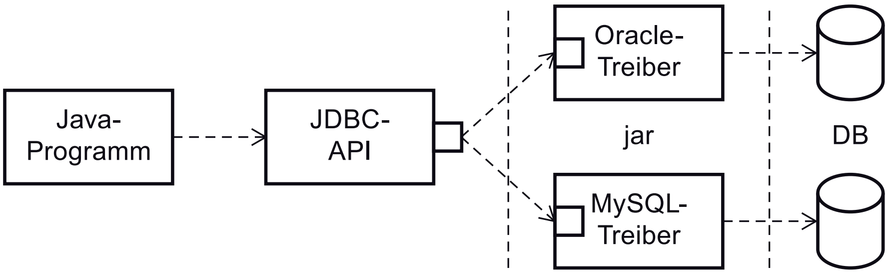

***Abbildung 3-1** Java-Programm, JDBC und DBMS im Zusammenspiel*

Bevor wir die einzelnen Schritte zum Datenbankzugriff per JDBC genauer betrachten, verdeutliche ich das grundsätzliche Vorgehen anhand eines Beispiels.

Für die nachfolgenden Beispiele gehen wir davon aus, dass die Tabelle `Personen` wie folgt aufgebaut und gefüllt ist:

```
ID     | Vorname | Name       | Geburtstag |
--------------------------------------------
1      | Peter   | Müller     | 1991-05-28 |
2      | Heinz   | Müller     | 1999-04-08 |
3140   | Werner  | Muster     | 1940-01-31 |
4711   | Marc    | Muster     | 1979-06-10 |
```

Einführendes Beispiel zur Datenbankabfrage

Zum Datenbankzugriff bietet das JDBC-API verschiedene Klassen und Interfaces, die in den Packages `java.sql` und `javax.sql` definiert sind. Insbesondere die Typen `Connection`, `Statement` und `ResultSet` spielen in den folgenden Abschnitten eine zentrale Rolle. Eine Datenbankverbindung wird durch ein `Connection`-Objekt repräsentiert. Ein solches dient zum Erzeugen von SQL-Anweisungen vom Typ `Statement`. Damit ausgeführte SQL-Abfragen liefern eine Ergebnismenge vom Typ `ResultSet`. Die dort enthaltenen Datensätze werden mithilfe der Methode `next()` durchlaufen. Zum Zugriff auf die Werte der Spalten nutzt man typsichere `get()`-Methoden. Beispielsweise liefert die Methode `getString(String)` den textuellen Wert derjenigen Spalte, deren Name als Parameter übergeben wurde.

Folgendes Listing zeigt den beschriebenen Ablauf beim Datenbankzugriff. Hier wird aus `Personen`-Datensätzen und der korrespondierenden Spalte `Name` eine Menge von Nachnamen ermittelt:

```
// Achtung: Hier als Vereinfachung nur rudimentäre Fehlerbehandlung
public static void main(final String[] args) throws SQLException,
                                                     ClassNotFoundException
{
   // Datenbanktreiber laden (optional seit JDBC 4)
   Class.forName("org.hsqldb.jdbcDriver");

   // Datenbankverbindungseinstellungen definieren
   final String dbUrl = "jdbc:hsqldb:hsql://localhost/java-profi";
   final String dbUsername = "sa";
   final String dbPassword = "";

   Connection connection = null;
   Statement statement = null;
   ResultSet resultSet = null;

   try
   {
       // Datenbankverbindung aufbauen
       connection = DriverManager.getConnection (dbUrl,
                                                 dbUsername,
                                                 dbPassword);

       // SQL-Befehlsobjekt erstellen
       statement = connection.createStatement();

       // SQL-Abfrage ausführen
       resultSet = statement.executeQuery("SELECT * FROM Personen");

       // Ergebnismenge auswerten und speichern, JDK 7: Diamond Operator
       final Set<String> names = new TreeSet<>();
       while (resultSet.next())
       {
           names.add(resultSet.getString("Name"));
       }

       // Ausgabe der ermittelten Nachnamen mit JDK 8 forEach() und Lambda
       names.forEach(name -> System.out.println("Nachname='" + name + "'"));
   }
   finally
   {
       // Achtung: Nicht ganz korrekte Implementierung von Aufräumarbeiten
       if (resultSet!= null) resultSet.close();
       if (statement!= null) statement.close();
       if (connection!= null) connection.close();
   }
}
```

Listing 3.1 *Ausführbar als* '**FIRSTJDBCEXAMPLE**'

Das Programm FIRSTJDBCEXAMPLE erzeugt erwartungsgemäß folgende Ausgaben:

```
Nachname='Muster'
Nachname='Müller'
```

Nach diesem ersten Kennenlernen möchte ich aber noch darauf hinweisen, dass man alle verwendeten Datenbankressourcen explizit wieder schließen muss, um durch sie belegte Systemressourcen freizugeben. Dazu nutzt man die Methode `close()` für das jeweilige Objekt, wie hier sukzessive für die Typen `ResultSet`, `Statement` und `Connection`. Datenbankverbindungen vom Typ `Connection` belegen unter anderem Sockets zur Kommunikation und Speicher zur Pufferung von Daten (sowohl aufseiten des Programms als auch des DBMS!). Ähnliches gilt auch für die SQL-Anweisungsobjekte (mit dem Basistyp `Statement`), die Abfragedaten für Tabellen zwischenpuffern. Auch die Ergebnismengen vom Typ `ResultSet` belegen Ressourcen, nämlich zumindest einen Positionszeiger auf den aktuellen Datensatz und häufig auch einige weitere Verwaltungsinformationen.

Die gezeigte Implementierung ist jedoch problematisch, wenn während der Abarbeitung einer der `close()`-Methoden eine Exception auftritt: Würde etwa im Beispiel beim Ausführen der Anweisung `statement.close()` eine Exception ausgelöst, so würde die nachfolgende Anweisung `connection.close()` nicht mehr aufgerufen, sondern – sofern vorhanden – ein `catch`-Block angesprungen oder wie hier die Exception weiter propagiert. Um aber die Ausführung aller `close()`-Methoden zu garantieren, müssen die einzelnen Aufrufe jeweils in einem separaten `try`-`catch`-Block erfolgen, was aber recht lang und unleserlich wird. Häufig ist das nun beschriebene ARM (Automatic Resource Management) eine gute Alternative.

Aufräumarbeiten mit ARM vereinfachen ARM ermöglicht es, Ressourcen automatisch wieder zu schließen, sofern die Ressourcenanforderungen in einem speziellen `try`-Block eingeschlossen sind. Am Beispiel des `ResultSet`s zeige ich, dass dadurch der explizite Aufruf der Aufräumarbeiten entfällt und sich die Zugriffe eleganter und kürzer wie folgt schreiben lassen:

```
final String query = "SELECT * FROM Personen";
// Ergebnismenge im ARM-Block definieren und auswerten
try (final ResultSet resultSet = statement.executeQuery(query))
{
    final Set<String> names = new TreeSet<>();
    while (resultSet.next())
    {
        names.add(resultSet.getString("Name"));
    }
} // Ende des ARM-Blocks => automatische Ressourcenfreigabe
```

3.1.1 Schritte zur Abfrage von Datenbanken

Nachdem Sie den Zugriff auf Datenbanken per JDBC in Grundzügen kennengelernt haben, möchte ich nun den Ablauf und die notwendigen Schritte etwas konkretisieren:

1. JDBC-Datenbanktreiber laden
2. Datenbankverbindung aufbauen
3. SQL-Anweisungsobjekt erzeugen
4. SQL-Anweisung ausführen
5. Ergebnisse auswerten
6. SQL-Anweisungsobjekt schließen
7. Datenbankverbindung schließen

Der letzten beiden Punkte der Freigabe sind besonders wichtig, um Systemressourcen freizugeben, insbesondere da Datenbankverbindungen relativ schwergewichtig sind und einige Systemressourcen belegen. Aufgrund dessen sollte man zur Ausführung von SQL-Anweisungen nicht jedes Mal wieder eine eigene Datenbankverbindung aufbauen, sondern, wenn möglich, eine bereits bestehende nutzen. ***Die beiden Schritte 4 und 5 werden daher in der Regel mehrfach durchlaufen***. Zudem wird durch den Einsatz von ARM die explizite Freigabe überflüssig – aus Gründen der Vollständigkeit beschreibe ich dies aber nachfolgend noch kurz.

Schritt 1: JDBC-Datenbanktreiber laden

Bevor überhaupt Zugriffe auf die Datenbank erfolgen können, muss ein Datenbanktreiber geladen und initialisiert werden. Dies geschieht normalerweise einmalig zu Beginn des Programms. Für den Treiber der HSQLDB ist dafür lediglich folgende Zeile in den Sourcecode aufzunehmen:

```
Class.forName("org.hsqldb.jdbcDriver");
```

Erwähnenswert ist, dass durch den Aufruf `Class.forName(String)` durch den `ClassLoader` nur die namentlich übergebene Klasse geladen wird, aber keine Instanz davon erzeugt wird. Das Laden des Treibers geschieht automatisch beim Initialisieren der Klasse. Seit JDBC 4 ist nicht einmal mehr das explizite Laden der Datenbanktreiberklasse als expliziter Schritt erforderlich, sondern dies wird automatisch durch die Klasse `DriverManager` einmalig beim Aufbau einer Datenbankverbindung erledigt.

Einbinden von Datenbanktreibern Zum Zugriff auf HSQLDB muss lediglich die Datei `hsqldb.jar` mit in den `Classpath` aufgenommen werden, um den Datenbanktreiber in die Applikation einzubinden. In der Gradle-Build-Datei notieren wir folgende Abhängigkeit:

```
compile 'org.hsqldb:hsqldb:2.3.2'
```

Schritt 2: Datenbankverbindung aufbauen

Nachdem der Treiber erfolgreich geladen wurde, kann man mithilfe der Klasse `DriverManager` und der Methode `getConnection(String, String, String)` folgendermaßen Kontakt zur Datenbank aufnehmen:

```
final Connection connection = DriverManager.getConnection(dbUrl,
                                                          dbUsername,
                                                          dbPassword);
```

Das zurückgelieferte `Connection`-Objekt kapselt eine Verbindung zur Datenbank. Zur Identifizierung der Datenbank muss eine URL beim Verbindungsaufbau angegeben werden. Im Aufruf sehen wir zusätzlich die Angabe von Benutzername und Passwort.

Darüber steuert man Authentifizierung und Autorisierung, also unter anderem, welche Rechte der angemeldete Benutzer besitzt und welche Daten für ihn zugreifbar sind.

Der Aufbau der URL folgt nur zum Teil einem Standard und ist daher leider je nach verwendetem DBMS unterschiedlich, startet aber immer mit `jdbc:`. Tabelle 3-1 zeigt das Format für verschiedene gebräuchliche Datenbanken.

Tabelle 3-1 *Datenbank-URLs*

DBMS	URL
HSQLDB	`jdbc:hsqldb:hsql://serverName:port/databaseName`
MySQL	`jdbc:mysql://serverName:port/databaseName`
Oracle	`jdbc:oracle:thin:@serverName:port:databaseName`

Verbindungseinstellungen extern konfigurieren Bereits beim Betrachten des einleitenden Beispiels ahnt man, dass die gezeigte Art des Verbindungsaufbaus durch den Einsatz von Magic Strings wenig flexibel und daher aufwendig zu warten ist: Jede Änderung in den Verbindungseinstellungen erfordert auch Änderungen im Sourcecode. Das ist unpraktisch. Wünschenswert ist vielmehr eine externe Speicherung der Verbindungseinstellungen, wodurch eine einfache Neu- oder Umkonfiguration ohne Programmänderungen möglich wird.

Zur Konfigurationsverwaltung nutzen wir die Klasse `java.util.Properties`, die Einstellungen aus einer Datei einlesen kann. Basierend darauf erstellen wir eine Klasse `DbProperties`, um den Verbindungsaufbau zur Datenbank unabhängig von konkreten Angaben im Sourcecode zu machen:

```
final DbProperties dbProperties = new DbProperties();
connection = DriverManager.getConnection(dbProperties.getUrl(),
                                         dbProperties.getUserName(),
                                         dbProperties.getPassword());
```

Bei der Implementierung der Klasse `DbProperties` nutze ich bewusst Delegation anstelle von Vererbung von der Klasse `Properties`. Außerdem verwende ich das mit JDK 7 eingeführte Sprachfeature ARM zur automatischen Freigabe des zum Einlesen der Konfigurationsdatei genutzten `FileInputStream`s. Mit diesen Randbedingungen ergibt sich dann folgende Umsetzung:

```
public final class DbProperties
{
    private static final String   URL           = "db.url";
    private static final String   USERNAME      = "db.username";
    private static final String   PASSWORD      = "db.password";
    private static final String[] REQUIRED_KEYS = { URL, USERNAME, PASSWORD };
    private static final String   FILENAME      = "config/db.properties";

    private final String          filePath;
    private final Properties      properties    = new Properties();
```

```
    public DbProperties()
    {
        final File file = new File(FILENAME);
        filePath = file.getAbsolutePath();

        // Ressourcen werden automatisch durch ARM wieder freigegeben.
        try (final InputStream inputStream = new BufferedInputStream(
                                             new FileInputStream(file)))
        {
            properties.load(inputStream);

            ensureAllKeysAvailable();

            // Dieser Aufruf stellt sicher, dass der URL-Wert vorhanden ist.
            final String url = properties.getProperty(URL, "");
            if (url.isEmpty())
            {
                throw new IllegalStateException("db config file '" + filePath +
                      "' is incomplete! Missing value for key: '" + URL + "'");
            }
        }
        catch (final IOException ioException)
        {
            throw new IllegalStateException("problems while accessing " +
                    "db config file '" + filePath + "'", ioException);
        }
    }

    private void ensureAllKeysAvailable()
    {
        final SortedSet<String> missingKeys = new TreeSet<>();

        for (final String key : REQUIRED_KEYS)
        {
            if (!properties.containsKey(key))
            {
                missingKeys.add(key);
            }
        }

        if (!missingKeys.isEmpty())
        {
            throw new IllegalStateException("db config file '" + filePath +
                              "' is incomplete! Missing keys: " + missingKeys);
        }
    }

    public String getUrl()
    {
        return properties.getProperty(URL, "");
    }

    public String getUserName()
    {
        return properties.getProperty(USERNAME, "");
    }

    public String getPassword()
    {
        return properties.getProperty(PASSWORD, "");
    }
}
```

Die Klasse `DbProperties` liest die benötigten Informationen zur Datenbankverbindung aus einer Datei `db.properties` ein und stellt zudem die Konsistenz der Eingabewerte sicher. Bei Unstimmigkeiten wird eine `IllegalStateException` mit aussagekräftigem Fehlertext ausgelöst, um diesen Sachverhalt auszudrücken. Im Speziellen wird die Existenz aller benötigten Einträge geprüft. Außerdem wird dafür gesorgt, dass zumindest der in jedem Fall zum Verbindungsaufbau erforderliche URL-Eintrag einen Wert besitzt. Die Angabe der Werte für Benutzername und Passwort ist zwar optional – in der Datei müssen aber zumindest die Schlüssel hinterlegt sein. Demnach muss die Datei für unser Beispiel mindestens folgende Einträge enthalten:

```
db.url=jdbc:hsqldb:hsql://localhost/java-profi
db.username=sa
db.password=
```

Schritt 3: SQL-Anweisungsobjekt erzeugen

Um SQL-Anweisungen auszuführen, benötigt man ein SQL-Anweisungsobjekt. Ein solches instanziiert man durch Aufruf einer der folgenden, im Interface `Connection` definierten Erzeugungsmethoden:

```
Statement createStatement() throws SQLException;
PreparedStatement prepareStatement(String) throws SQLException;
```

Mit `createStatement()` erzeugt man eine allgemeine Repräsentation einer SQL-Anweisung, die eine SQL-Abfrage oder einen SQL-Befehl ausführen kann. Mit der Methode `prepareStatement(String)` konstruiert man ein `PreparedStatement` aus einer textuell vorliegenden SQL-Anweisung. Die so übergebene SQL-Anweisung wird von der Datenbank analysiert und deren Ausführung optimiert und vorbereitet. Das Interface `PreparedStatement` wird später in Abschnitt 3.1.5 besprochen. Zunächst konzentrieren wir uns nachfolgend auf die Ausführung von SQL-Anweisungen mithilfe von `Statement`-Objekten.

Schritt 4: SQL-Anweisung ausführen

Ein Objekt vom Typ `Statement` ermöglicht es, SQL-Abfragen oder SQL-Befehle an die Datenbank abzusetzen. Dazu bietet das `Statement`-Objekt folgende Methoden:

```
ResultSet executeQuery(String)
int executeUpdate(String)
```

Die gewünschte SQL-Anweisung wird dabei textuell angegeben. Folgendes Listing zeigt jeweils ein Beispiel für beide Methoden:

```
// Abfrage erzeugen und ausführen
final String sqlQuery = "SELECT * FROM Personen WHERE Vorname LIKE 'M%'";
final ResultSet resultSet = statement.executeQuery(sqlQuery);

// Änderungsanweisung erzeugen und ausführen
final String sqlUpdate = "UPDATE Personen SET Vorname = 'Mike' " +
                                        "WHERE Vorname LIKE 'M%'";
final int updateCount = statement.executeUpdate(sqlUpdate);
```

Schritt 5: Ergebnisse auswerten

Abhängig vom Typ der ausgeführten SQL-Anweisung erhält man als Rückgabe eine Ergebnismenge vom Typ `ResultSet` oder einen Zahlenwert vom Typ `int`.

Ergebnisse von SQL-Abfragen auswerten Mit `executeQuery(String)` ausgeführte SQL-Abfragen liefern `ResultSet`-Objekte als Ergebnis. Zur Verarbeitung kann man das `ResultSet` durch Aufruf der Methode `next()` durchlaufen und sukzessive jeweils einen einzelnen Datensatz der Ergebnismenge ermitteln. Verschiedene `getXXX()`-Methoden erlauben es, die Werte der jeweiligen Spalte des Datensatzes typsicher zu ermitteln: Dabei steht `XXX` für einen bestimmten Datentyp, etwa `String`.[1] Zum Auslesen eines Werts existieren die Methoden `getString(String)` und `getString(int)`. Die erste Variante der Methode erhält als Parameter den Namen der gewünschten Spalte.[2] Die zweite nutzt die Nummer der Spalte, wobei die Nummerierung in JDBC bei 1 und nicht bei 0 beginnt. Falls das etwas verwirrend klingt, werfen Sie einen klärenden Blick auf Abbildung 3-2. Dort sind beide Varianten dargestellt.

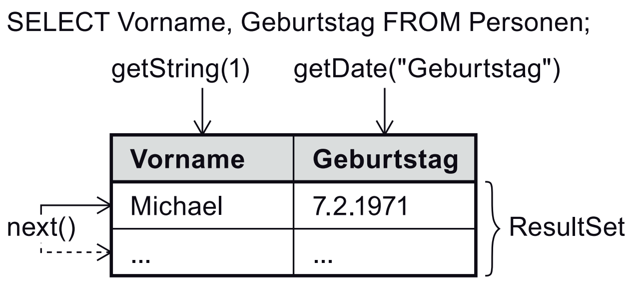

***Abbildung 3-2** DB-Zugriffe mit* `ResultSet`

[1] Allerdings ist es unabhängig vom tatsächlichen Typ der Spalte möglich, auch unsinnige Aufrufe auszuführen, etwa ein `getDate()` für die Spalte `Name`.

[2] Es kann auch ein Aliasname übergeben werden. Das funktioniert, sofern dieser im `SELECT`-Befehl vorkommt: Für den Befehl `SELECT Name AS Nachname FROM Personen` liefert der Aufruf von `getString("Nachname")` den gewünschten Wert. Generell wird zunächst versucht, eine Zuordnung zu einer Spalte über alle vergebenen Aliasnamen zu erzielen. Wird kein solcher gefunden, wird der angegebene Name als Spaltenname der Tabellen interpretiert.

Der Spaltenindex korrespondiert zu der Angabe (Position) der Spalte im zugehörigen `SELECT`-Befehl. Für den Befehl `SELECT Name, Vorname FROM Personen` liefert ein Aufruf von `getString(1)` demnach den Wert aus der Spalte `name`.

Führen wir dagegen ein `SELECT * FROM Personen` aus, so existiert kein direkter Bezugspunkt. Daher beziehen sich die Spaltenindizes in diesem Fall auf die Reihenfolge der Spalten gemäß der Definition im Datenbankmodell: Für die bereits vorgestellte `Personen`-Tabelle liefert `getString(1)` dann den Wert aus der Spalte `vorname`.

Im folgenden Beispiel wird zum Zugriff auf zwei Spalten sowohl die indizierte als auch die namensbasierte Variante genutzt. Mit `getString(1)` wird der Wert der Spalte `Vorname` und mit `getDate("Geburtstag")` die gleichnamige Spalte ausgelesen:

```java
final Map<String, Date> namesAndBirthdays = new TreeMap<>();

final String sqlQueryCommand = "SELECT Vorname, Geburtstag FROM Personen";
try (ResultSet resultSet = statement.executeQuery(sqlQueryCommand))
{
    while (resultSet.next())
    {
        namesAndBirthdays.put(resultSet.getString(1),
                              resultSet.getDate("Geburtstag"));
    }
}

// JDK 8: forEach() zur Ausgabe der ermittelten Paare Vorname => Geburtstag
namesAndBirthdays.forEach((k,v) -> System.out.println("Vorname='" + k + "' / " +
                                                       "'Geburtstag: " + v));
```

Listing 3.2 *Ausführbar als* '**JDBCRESULTSETEXAMPLE**'

Das Programm JDBCRESULTSETEXAMPLE produziert folgende Ausgaben:

```
Vorname='Heinz' / Geburtstag: 1999-04-08
Vorname='Marc' / Geburtstag: 1979-06-10
Vorname='Peter' / Geburtstag: 1991-05-28
Vorname='Werner' / Geburtstag: 1940-01-31
```

Ergebnisse von SQL-Befehlen auswerten Neben SQL-Abfragen, die mit `executeQuery(String)` ausgeführt werden, kann man auch SQL-Befehle mit `executeUpdate(String)` an die Datenbank absetzen, etwa einen `INSERT INTO`-, `UPDATE`- oder `DELETE FROM`-Befehl. Als Rückgabe erhält man in diesem Fall keine Ergebnismenge, sondern einen `int`-Wert, der die Anzahl der Datensätze zurückgibt, die durch die Ausführung des SQL-Befehls betroffen sind.

Hinweis: Fehlerbehandlung und die Klasse `SQLException`

SQL-Anweisungen können aus verschiedenen Gründen fehlschlagen. Im einfachsten Fall wurde eine SQL-Anweisung falsch geschrieben. Möglicherweise ist auch die Verbindung zur Datenbank gestört. Beide Arten von Fehlern lösen `SQLException`s aus, wobei Syntaxfehler in der Regel nur durch Änderungen am Sourcecode zu korrigieren sind und Verbindungsprobleme operativ gelöst werden müssen.

Schritt 6: SQL-Anweisungsobjekt schließen

Für `ResultSet`-Objekte haben wir die Freigabe mithilfe der Methode `close()` bereits kennengelernt. Für `Statement`-Objekte geschieht dies analog wie folgt:

```
statement.close();
```

Schritt 7: Datenbankverbindung schließen

Analog zu `ResultSet`- und `Statement`-Objekten müssen insbesondere die Datenbankverbindungen wieder geschlossen werden:

```
connection.close();
```

Bedenken Sie, dass das Auf- und Abbauen von Datenbankverbindungen mit einem nicht unerheblichen Aufwand verbunden ist. Deshalb sollte man nicht leichtfertig für jedes Kommando eine neue Verbindung nutzen. Vielmehr bietet es sich an, Datenbankverbindungen zu »recyclen«. In folgendem Praxistipp gehe ich darauf ein.

Tipp: Connection Pooling

Je besser strukturiert eine Anwendung ist, desto leichter lässt sich vermeiden, dass für jede zu versendende SQL-Anweisung (versehentlich) immer wieder eine neue Verbindung eröffnet wird, statt eine bereits bestehende Datenbankverbindung wiederzuverwenden. Wenn die Applikation aber komplexer ist und aus mehreren Teilen besteht, gibt es vermutlich einige Stellen, die auf die Datenbank zugreifen. Für diesen Fall lässt sich Abhilfe durch einen sogenannten ***Connection-Pool*** schaffen.[a] Dieser öffnet eine gewisse Anzahl an Verbindungen zur Datenbank und verwaltet diese zentral. Benötigt die Applikation eine Verbindung zur Datenbank, so wird diese durch den Pool bereitgestellt und als besetzt markiert. Nach dem Abschluss der Arbeiten sollte eine Applikation die vor ihr genutzten Verbindungen wieder an den Pool »zurückgegeben«. Der Pool wird dies dann als wieder verfügbar markieren.

[a] Allen Interessierten empfehle ich einen Blick auf Apache Commons DBCP unter `http://commons.apache.org/dbcp/`.

Gerade in größeren Anwendungen finden oftmals diverse Zugriffe durch verschiedene Benutzer in kurzem zeitlichem Abstand statt. Dabei kann sich der Einsatz von ***Connection Pooling*** (sehr) positiv bemerkbar machen:

- Zum einen kann man Anfragen ressourcenschonend mit nur wenigen Datenbankverbindungen bearbeiten.
- Zum anderen verhindert man durch die Beschränkung auf eine maximale Anzahl an Verbindungen auch die Überlastung oder gar den Zusammenbruch eines Servers – dabei ist zudem zu bedenken, dass zum Teil auch andere Applikationen auf die Datenbank zugreifen.

Mögliche Probleme ohne Connection Pooling Ohne Connection Pooling bestünde die Gefahr, dass die Anzahl der Verbindungen proportional mit der Anzahl an Anfragen wächst und so über kurz oder lang ein Ressourcenproblem (Speicher, Prozessor, Netzwerk) auslöst. Man erlebt immer wieder Situationen, in denen extrem viele Verbindungen zeitgleich eröffnet werden sollen. Im positiven Fall geschieht dies aufgrund von großem Interesse. Im negativen Fall ist dies Teil einer Denial-of-Service-Attacke, die den Zusammenbruch des Systems zum Ziel hat. Bei einer solchen Attacke werden durch massive Zugriffe auf die eigentliche Applikation als Folge auch viele Verbindungen zur Datenbank geöffnet. Für beide beschriebenen Extremsituationen ermöglicht Connection Pooling, dass ein System arbeitsfähig bleibt, da maximal lediglich so viele Anfragen parallel verarbeitet werden, wie initial Verbindungen im Pool bereitgestellt wurden.

3.1.2 Besonderheiten von `ResultSet`

Ein `ResultSet` repräsentiert die Daten, die durch eine SQL-Abfrage von der Datenbank erhalten wurden. In dieser Ergebnismenge kann man jeweils nur einen einzigen Datensatz zur gleichen Zeit bearbeiten. Dieser aktuelle Datensatz wird durch einen Datensatzzeiger (auch ***Cursor*** genannt) referenziert, der sich beim Erzeugen eines `ResultSet`s logisch vor dem ersten Datensatz befindet und dann zur Verarbeitung der Daten sukzessive auf den gewünschten Datensatz bewegt werden kann. In der Konsequenz muss vor dem Beginn der Verarbeitung mindestens einmal die Methode `next()` aufgerufen werden, um den Datensatzzeiger auf den ersten Datensatz zu bewegen. Diese und weitere Navigationsmöglichkeiten werden wir im Verlauf dieses Abschnitts genauer betrachten.

Zuvor ist es noch wichtig, einen anderen Aspekt der Navigierbarkeit zu beleuchten: ***Die Daten der Ergebnismenge werden normalerweise nicht vollständig in einem `ResultSet` gespeichert und an das aufrufende Programm übertragen, sondern der Großteil davon verbleibt zunächst aufseiten der Datenbank***. Gegebenenfalls werden erst beim tatsächlichen Zugriff die Daten eines Ergebnisdatensatzes an das Programm übermittelt. Da dieses Vorgehen in seiner Reinform bezüglich der Performance der Zugriffe ungünstig ist (viele Zugriffe und kleine Datenübertragungen), werden normalerweise beim Lesen eines Datensatzes auch einige Datensätze hinter dem angeforderten Datensatz ermittelt und an das anfragende Programm versendet.[3] Vor allem dieser Mechanismus und auch die Navigation gemäß dem ITERATOR-Muster[4] ermöglichen es, selbst extrem große Ergebnismengen verarbeiten zu können, ohne an Hauptspeichergrenzen zu stoßen.

[3] Die Anzahl der zusätzlich ermittelten Datensätze lässt sich durch den Aufruf von `setFetchSize(int)` festlegen.

[4] Dieses beschreibe ich ausführlich in meinem Buch »Der Weg zum Java-Profi« [8].

Konfiguration eines `ResultSets`

Sofern nicht anders spezifiziert, erfolgt die Verarbeitung eines `ResultSet`s sukzessive in Vorwärtsrichtung. Des Weiteren können Datensätze standardmäßig ausschließlich gelesen, aber nicht modifiziert werden. Dieses Verhalten ist änderbar – allerdings nicht auf Ebene von `ResultSet`s. Stattdessen kann das `Statement`, das das `ResultSet` erzeugt, durch zwei Parameter die Eigenschaften zur Navigierbarkeit sowie zur Verarbeitung paralleler Zugriffe festlegen:

```
Statement createStatement(int resultSetType,
                          int resultSetConcurrency) throws SQLException;
```

Bevor ich auf mögliche Werte für die beiden Übergabeparameter eingehe, behandle ich zunächst noch einige Aspekte, die bei der Wahl der passenden Parameterwerte für einen Anwendungsfall eine wesentliche Rolle spielen.

Navigierbarkeit (`resultSetType`) Neben der Möglichkeit, sich in der Ergebnismenge vom Typ `ResultSet` per `next()` vorwärts zu bewegen, wird auch eine freie Navigation durch positionsbasierten wahlfreien Zugriff unterstützt. Dadurch kann man Datensätze direkt anspringen oder der Datensatzzeiger kann relativ zur Position des aktuellen Datensatzes (auch rückwärts) bewegt werden.

Read-only vs. Updatable (`resultSetConcurrency`) Neben den bereits bekannten `getXXX()`-Methoden zur Verarbeitung der Daten eines Ergebnisdatensatzes sind in `ResultSet` auch `updateXXX()`-Methoden definiert, die das Ändern einzelner Werte ermöglichen. Die `getXXX()`-Methoden können unabhängig von der Parametrierung des `ResultSet`s immer aufgerufen werden. Die `updateXXX()`-Methoden erfordern eine entsprechende Wertebelegung beim Aufruf von `createStatement(int, int)` für den Parameter `resultSetConcurrency`.

Sensitivität für Änderungen Während wir die Daten eines `ResultSet`s verarbeiten, können zeitgleich andere Benutzer oder Programme die Datenbank verwenden und dort Veränderungen vornehmen, die auch für die Datensätze gelten, die durch das `ResultSet` repräsentiert werden. Die Sensitivität für Änderungen bestimmt, ob in einem `ResultSet` derartige Änderungen widergespiegelt werden. Ist das `ResultSet` sensitiv für Änderungen durch andere, so liefern Aufrufe von `getXXX()`-Methoden zu einem Zeitpunkt möglicherweise andere Werte verglichen mit denen zum Konstruktionszeitpunkt des `ResultSet`s. Ist das `ResultSet` dagegen nicht sensitiv, so sieht man die Änderungen nicht und liest immer genau den Stand der Daten zum Konstruktionszeitpunkt (Snapshot). Mit Letzterem erhält man eventuell veraltete Werte. Allerdings kann diese Unabhängigkeit auch gewünscht sein. Die korrekte Einstellung variiert von Anwendungsfall zu Anwendungsfall und sollte damit sehr bewusst gewählt werden, worauf ich im Folgenden eingehe.

Wahl der Einstellungen Die beiden beschriebenen Eigenschaften eines `ResultSet`s werden mithilfe der Parameter `resultSetType` und `resultSetConcurrency` bestimmt. Für den Parameter `resultSetType` sind folgende Werte definiert:

- `ResultSet.TYPE_FORWARD_ONLY` – Dieser Wert erlaubt lediglich die Navigation in Vorwärtsrichtung und ***stellt den Standard dar***. Jeder Datensatz eines derartigen `ResultSet`s kann genau einmal verarbeitet werden.
- `ResultSet.TYPE_SCROLL_INSENSITIVE` – Die Wahl dieser Parametrierung erlaubt es, beliebige Datensätze anzuspringen (auch in Rückwärtsrichtung), sodass Datensätze bei Bedarf auch mehrfach verarbeitet werden können. Der Namenszusatz `INSENSITIVE` bezieht sich darauf, dass Änderungen anderer Transaktionen nicht wahrgenommen werden.
- `ResultSet.TYPE_SCROLL_SENSITIVE` – Diese Einstellung bietet einen dynamischen, wahlfreien Zugriff auf beliebige Datensätze. Zudem werden Änderungen, die im Rahmen anderer Transaktionen erfolgen, im `ResultSet` reflektiert.

Für den Parameter `resultSetConcurrency` sind folgende Werte zulässig:

- `ResultSet.CONCUR_READ_ONLY` – Dieser Wert legt fest, dass das `ResultSet` keine schreibenden Zugriffe auf die Datenbank ausführt. ***Das ist der Standard***.
- `ResultSet.CONCUR_UPDATABLE` – Sollen beim Verarbeiten eines `ResultSet`s Änderungen an dessen Datensätzen erfolgen, so ist diese Einstellung zu wählen. Allerdings funktioniert dies nur dann, wenn die Abfragen keine Joins oder Aggregats- bzw. Gruppierungsfunktionen nutzen. Der Grund besteht darin, dass bei diesen Formen von Abfragen keine Zuordnung zu einzelnen Datensätzen mehr möglich ist, da die Ergebnismenge nicht 1:1 auf Datensätze der Datenbank abbildbar ist.

Absolute und relative Positionierung

Wie oben dargestellt, kann es zur Verarbeitung einer Ergebnismenge praktisch sein, wenn diese neben der sequenziellen Verarbeitung einen wahlfreien Zugriff auf enthaltene Datensätze erlaubt. Allerdings muss diese Art der Navigation explizit gewählt werden. Dann können außer der Methode `next()` auch diverse weitere, im Folgenden aufgezählte Methoden zur Positionierung des Datensatzzeigers (Cursor) genutzt werden.[5]

- `boolean first()` – Springt zum ersten Datensatz der Ergebnismenge.
- `boolean last()` – Springt zum letzten Datensatz der Ergebnismenge.
- `void afterLast()` – Positioniert den Cursor hinter den letzten Datensatz.
- `void beforeFirst()` – Positioniert den Cursor vor den ersten Datensatz.
- `boolean absolute(int)` – Springt zum angegebenen Datensatz.

[5] Für ein Standard-`ResultSet` lösen diese Methoden beim Aufruf jedoch eine `java.sql.SQLFeatureNotSupportedException` aus.

- `boolean relative(int)` – Bewegt den Cursor relativ um die angegebene Anzahl an Datensätzen. Ein negativer Wert bewegt den Cursor in Richtung Anfang des `ResultSet`s. Positive Werte bewegen den Cursor vorwärts.
- `boolean previous()` – Bewegt den Cursor auf den vorherigen Datensatz.

Neben diesen Methoden zur Navigation sind für uns noch zwei weitere Methoden von Interesse, da diese in den folgenden Abschnitten zum Einsatz kommen:

- `int getRow()`[6] – Liefert die Position des Datensatzzeigers.
- `Object getObject(int)` und `Object getObject(String)` – Ermittelt den Wert der als Index oder Name angegebenen Spalte als Java-Objekt vom Typ `Object`. Der Rückgabetyp ist ein Java-Typ und entspricht dem korrespondierenden SQL-Typ der Spalte. Man erhält also z. B. Objekte vom Typ `String`, `Date` usw. oder Wrapper-Objekte, die man in primitive Werte umwandeln kann, sofern in der Spalte kein `NULL`-Wert gespeichert ist. Darauf komme ich später zurück.

Änderungen an `ResultSets`

Im Folgenden gehe ich davon aus, dass ein `ResultSet` erzeugt wurde, das die Eigenschaft `ResultSet.CONCUR_UPDATABLE` besitzt. Wie erwähnt, kann man dadurch nicht nur frei über die Ergebnismenge navigieren, sondern auch Änderungen vornehmen. Im Interface `ResultSet` sind für diesen Zweck zu den Lesemethoden `getXXX()` korrespondierende `updateXXX()`-Methoden definiert:

```
resultSet.updateString(columnName, newValue);
resultSet.updateRow();
```

Im folgenden Listing sehen wir den Einsatz der Methode `updateString(String columnName, String newValue)` zur Änderung textueller Werte. Um auch andere Typen verarbeiten zu können, bietet das Interface `ResultSet` verschiedene überladene Varianten dieser Methode. Änderungen sind zunächst nur im `ResultSet` selbst wirksam. Für eine dauerhafte Speicherung ist es notwendig, die Änderungen durch einen Aufruf von `updateRow()` in der Datenbank zu sichern.

Mit diesem Basiswissen realisieren wir nun eine Methode `updatePersons()`, mit der der Wert einer beliebigen Spalte vom Typ `String` geändert werden kann. Das nutzen wir hier, um die Vornamen einiger `Personen`-Datensätze zu ändern. Dazu ermittelt die Methode `updatePersons()` alle Personen und liest den Wert der als Parameter `columnName` übergebenen Spalte aus. Alle Werte, die dem `originalValue` entsprechen, werden auf den Wert `newValue` gesetzt.

---

[6] Die Namensgebung von JDBC orientiert sich fast überall sehr eng an ODBC (Open Database Connectivity) und ist meiner Meinung nach etwas unschön. Hier wäre `getRowNumber()` oder `getRowIndex()` ein geeigneterer Name gewesen, da ja keine Zeile, sondern eine Zeilennummer zurückgeliefert wird.

```
private static void updatePersons(final Connection connection,
                                  final String columnName,
                                  final String originalValue,
                                  final String newValue)
                    throws SQLException
{
    final String sqlQuery = "SELECT * FROM Personen";

    try (final Statement statement = connection.createStatement(
                                     ResultSet.TYPE_SCROLL_INSENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
         final ResultSet resultSet = statement.executeQuery(sqlQuery))
    {
        while (resultSet.next())
        {
            final String value = resultSet.getString(columnName);
            if (value.equals(originalValue))
            {
                // Änderungen am ResultSet vornehmen
                resultSet.updateString(columnName, newValue);
                // Änderungen in DB schreiben
                resultSet.updateRow();
            }
        }
    }
}
```

Die ausführende `main()`-Methode ändert Vornamen durch einen Aufruf wie folgt:

```
public static void main(final String[] args) throws SQLException
{
    final DbProperties dbProps = new DbProperties();

    try (final Connection connection = DriverManager.getConnection(
                                            dbProperties.getUrl(),
                                            dbProperties.getUserName(),
                                            dbProperties.getPassword()))
    {
        // Alle Personen mit Vornamen Mike in Michael umbenennen
        updatePersons(connection, "Vorname", "Mike", "Michael");
    }
}
```

Listing 3.3 *Ausführbar als* '**JDBCRESULTSETUPDATEEXAMPLE**'

Verarbeitung von Änderungen an den Datensätzen Im obigen Programm haben wir die Änderungen durch einen Aufruf von `updateRow()` in der Datenbank persistiert. In der Praxis ist die Verarbeitung von Datensätzen aber meistens deutlich komplexer. Mitunter sollen Änderungen doch nicht in der Datenbank gesichert werden. Für diesen Fall stellt das `ResultSet` zwei Methoden bereit: Zum einen kann man durch den Aufruf von `cancelRowUpdates()` die zuvor gemachten Änderungen verwerfen. Dadurch werden die durch das `ResultSet` ursprünglich ermittelten Originalwerte wiederhergestellt. Zum anderen ist es durch einen Aufruf von `refreshRow()` möglich, die Daten des Datensatzes erneut aus der Datenbank zu lesen. Dadurch kann man möglicherweise in der Zwischenzeit stattgefundene Änderungen anderer Benutzer oder Programme berücksichtigen, d. h. in die eigenen Daten aufnehmen. Weil das Verhalten vom gewählten Typ des `ResultSet`s sowie von den gewählten Isolationsgraden anderer Transaktionen abhängig ist (siehe dazu auch Abschnitt 3.1.6), sollte `refreshRow()` mit Bedacht und auf den jeweiligen Anwendungsfall bezogen genutzt werden.

Änderungen an den Datensätzen eines `ResultSets` Neben der Änderung einzelner Werte innerhalb eines Datensatzes eines `ResultSet`s ist es auch möglich, neue Datensätze zu einem `ResultSet` hinzuzufügen bzw. bestehende Datensätze daraus zu löschen. Dazu dienen die Methoden `insertRow()` und `deleteRow()`.

Umgang mit `NULL`-Werten

Bis hierher haben wir einen Sachverhalt kaum betrachtet: `NULL`-Werte in Tabellen. Für die nachfolgenden Ausführungen sollte man wissen, dass in Datenbanken für aus Java-Sicht primitive Typen auch der Wert `NULL` in einer Spalte gespeichert werden kann.

Bei der Verarbeitung von `ResultSet`s greift man auf die Werte von Spalten mit `getXXX()`-Methoden zu, etwa mit `getInt(String)`. Dann erhält man einen Wert des primitiven Typs `int`. In Java kann dieser niemals den Wert `null` besitzen, bei Datenbankabfragen sind aber `NULL`-Werte für Spalten aller Typen erlaubt. Dadurch entspricht der über die korrespondierende `getXXX()`-Methode gelesene Wert nicht unbedingt dem Datenbankinhalt, falls in der Datenbank ein `NULL`-Wert gespeichert ist. Es muss eine Abbildung auf einen primitiven Wert erfolgen. Während es im Java-Sourcecode beim Auto-Unboxing für Wrapper-Klassen bei der Konvertierung eines `null`-Werts zu einer `NullPointerException` kommt, ***liefern dagegen die `getXXX()`-Methoden den Defaultwert für den primitiven Datentyp, also für Zahlentypen den Wert 0 bzw. für boolesche Variablen den Wert `false`***. Bei der Rückgabe kann man demnach nicht unterscheiden, ob der gelieferte Wert tatsächlich so in der Datenbank gespeichert ist oder ob dort ein `NULL`-Wert vorliegt. Was kann man also tun?

Zum einen ist es möglich, die Methode `getObject()` zu nutzen, die für `NULL`-Werte auch immer den Java-Wert `null` zurückgibt. Für andere Werte muss dann allerdings immer noch eine Typumwandlung auf den gewünschten primitiven Typ vorgenommen werden. Zum anderen kann man über die Methode `wasNull()` ermitteln, ob in einer Spalte ein `NULL`-Wert enthalten ist.

Nehmen wir an, die Tabelle `Personen` würde um eine Spalte `Lieblingszahl` erweitert, deren Angabe optional wäre. Eine Methode, die alle Lieblingszahlen aufsammelt und diese sortiert zurückliefert, soll natürlich nur Lieblingszahlen, aber nicht den Defaultwert 0 für `NULL`-Spalten enthalten, sofern dieser Wert nicht explizit als Lieblingszahl in den Daten vorhanden ist. Um dies zu erreichen, nutzen wir den Aufruf der Methode `wasNull()` wie folgt:

```java
final int lieblingszahl = resultSet.getInt("Lieblingszahl");
if (!resultSet.wasNull())
{
    lieblingszahlen.add(lieblingszahl);
}
```

3.1.3 Abfrage von Metadaten

Bei SQL-Abfragen sind wir bislang immer von einer bestimmten Reihenfolge der Spalten ausgegangen. Diese Annahme ist aber für eine Vielzahl benutzerdefinierter SQL-Abfragen nicht gültig. Um die Ergebnisse beliebiger SQL-Abfragen auswerten zu können, ist es wünschenswert, Zugriff auf Metadaten zu Datenbanken und Ergebnismengen zu besitzen – ähnlich wie es Reflection für Informationen zu Klassen und deren Merkmalen erlaubt. Nachfolgend wollen wir in den nächsten Abschnitten schrittweise ein praktisches SQL-Abfrage-Tool entwickeln. Dieses Tool soll die Eingabe beliebiger SQL-Abfragen erlauben und diese an die Datenbank absetzen. Die darüber ermittelten Ergebnisse sollen in einer Tabelle dargestellt werden. Abbildung 3-3 zeigt eine mögliche SQL-Abfrage und die resultierende Darstellung der Ergebnismenge.

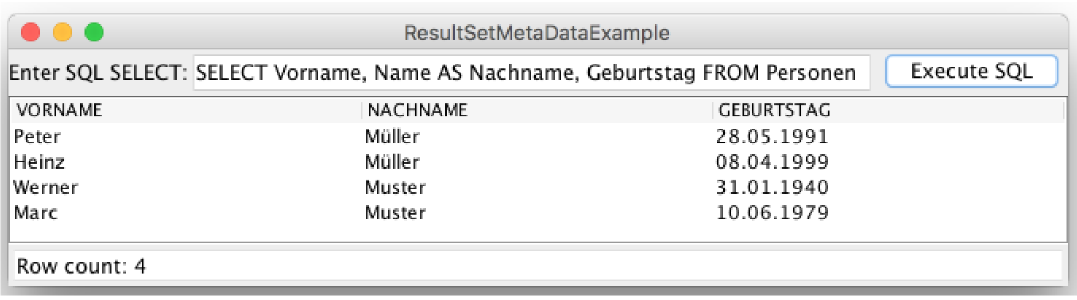

Abbildung 3-3 *Einfaches Tool zum Datenbankzugriff*

Ermittlung von Metadaten

Metadaten zu einer Datenbank kann man über das Interface `Connection` und dessen Methode `getMetaData()` ermitteln. Die Rückgabe ist ein Objekt vom Typ `DatabaseMetaData`. Die dort angebotene Funktionalität ist recht umfangreich und eignet sich daher nicht so gut zu einer kurzen Besprechung und Vorstellung der Verarbeitung von Metadaten. Weil Metadaten aber nützlich sein können, um Applikationen elegant zu erstellen, möchte ich deren Verarbeitung anhand eines Beispiels für den weniger umfangreichen und dadurch verständlicheren Typ `ResultSetMetaData` besprechen. Eine Instanz davon erhält man durch den Aufruf der Methode `getMetaData()` für ein Objekt vom Typ `ResultSet`.

Bevor wir mit der Implementierung der gezeigten Applikation beginnen, lernen wir zunächst einige Methoden des Typs `ResultSetMetaData` kennen:

- `getColumnCount()` – Ermittelt die Anzahl der Spalten.
- `getColumnName()` bzw. `getColumnLabel()` – Die Methode `getColumnName()` liefert den Namen der Spalte, wie er im Datenbankmodell hinterlegt ist. Als Erweiterung dazu berücksichtigt `getColumnLabel()` auch einen Aliasnamen, sofern dieser im `SELECT`-Befehl angegeben wurde.