# DBS\_VL10\_Zugriff auf Daten in RDBS aus Programmiersprachen

## Prinzipielle Vorgehensweisen

### Zugriff aus Anwendungen auf RDBMS

Probleme: Konzeptionelle Unterschiede zwischen Programmiersprache und RDBMS

* Probleme bei der Einbettung von SQL: **Impedance Mismatch**
* Unterschiede: SQL vs. imperative / objektorientierte Programmiersprache
* SQL: **deklarativ, mengenorientiert**; Ergebnis einer Anfrage ist eine (möglicherweise sehr große) Relation
* Programmiersprache: **imperativ und/oder objekt-orientiert; satz-orientiert**
* Unterschiede zwischen den **Datentypen**
* Lösungen zur Überwindung des Impedance Mismatch:
* **Iteratoren/Cursor** zur satzweisen Verarbeitung von Ergebnismengen
* **Mapping der Typen** DBMS ←→ Programmiersprache
* **Zugriffsfunktionen** zum Datenaustausch mit Typkonvertierung

### Mögliche Herangehensweisen

Prozedurale Schnittstelle (**Call-Level Interface**, CLI):

* **Bibliothek**, die Funktionen bereitstellt, um mit der DB zu kommunizieren
* **Dynamisches SQL**: SQL wird als **String** in der Wirtssprache zusammengebaut und an Datenbank gesendet
* Ergebnisverarbeitung in der Programmiersprache
* Diese Vorlesung: JDBC als Beispiel; Alternativen: ODBC, OCI, ...

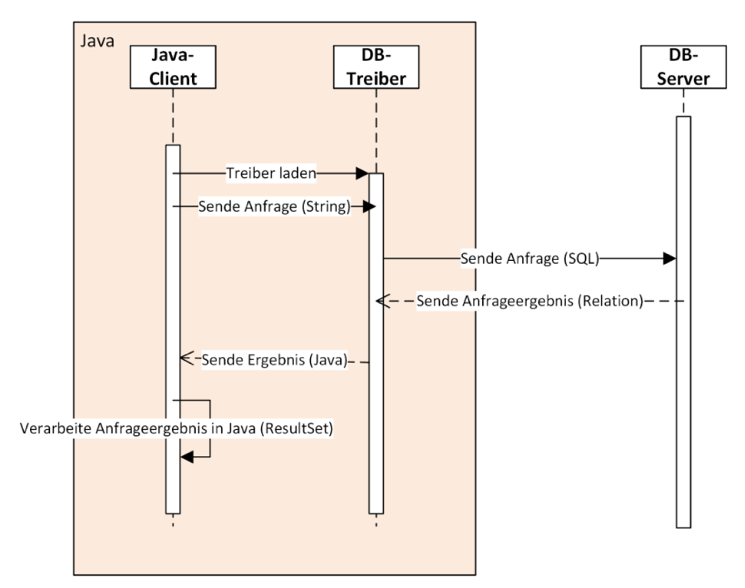
**Einbettung von SQL in eine Sprache**

* SQL wird in die Wirtssprache eingebettet
* **Statisches SQL**: Zur Übersetzungszeit geprüft
* Beispiel Pro\*C, SQLJ

**SQL um prozedurale Elemente erweitern**

* Beispiel PL/SQL (Oracle)
* Datenbanksysteme 2

**Funktionsbibliothek, mit der SQL-Statements zusammengestellt werden**

* Beispiel: JOOQ

## Zugriff aus Java: JDBC-Konzepte

Architektur für Java-basiertes CLI

* Java-Programm muss mit DB kommunizieren
* Erfordert Umsetzung der Java- Funktionsaufrufe in   
  DB-Aufrufe (Netzwerkprotokoll)
* Java-Programm soll unabhängig von DB sein
* DB-spezifische Umsetzung wird daher von einem   
  DB-Treiber (JDBC-Treiber) bereitgestellt
* Stammt vom Anbieter des DBMS, Java- Bibliothek

### JDBC (Java Database Connectivity)

* JDBC 1.0 im Rahmen des JDK 1.1 (1996)
* Aktuelle Versionen JDBC 4.3 im Rahmen von Java SE 9
* Ziel: **Herstellerunabhängige** API zum Zugriff auf beliebige relationale DBMS   
  (**A**pplication **P**rogramming **I**nterface, Programmierschnittstelle für Anwendungen)
* Kernfunktionen: Senden von SQL-Befehlen an das DBMS, Verarbeiten der Ergebnisse
* Dynamisches SQL: SQL wird als String in Java “zusammengebaut”

### JDBC-Treiber

* Kapseln hersteller-abhängige Anteile und bieten gemeinsame Programmierschnittstelle
* Software für Connect mit DB und die Ausführung von SQL-Befehlen (inkl. Holen der Ergebnismenge); kann auch DBS-spezifische Funktionen verarbeiten
* Setzt auf Netzwerk-Protokoll auf: TCP/IP, ODBC, net8,...

### Laden des JDBC-Treibers in Java

**Neue Variante**:

Connection conn = **DriverManager.getConnection**(   
"jdbc:oracle:thin:@dboracleserv.inform.hs-hannover.de:1521:db01", "user", "password");

**Alte Variante**:

* Zu nutzen für alte DB-Treiber

**Class.forName**("oracle.jdbc.driver.OracleDriver");

* Class.forName lädt die Klasse des Treibers
* Der Treiber registriert sich dabei selbst bei DriverManager

Damit weiß der DriverManager, dass jdbc:oracle:thin von der Klasse oracle.jdbc.driver.OracleDriver gehandhabt wird

## Konkrete SQL-Anweisungen in JDBC

### Anfragen: Grundlagen

#### Ausführen einer Anfrage

Hauptprobleme (Impedance Mismatch):

* **Statische Anfragen selten hilfreich**
  + Anfragen benötigen Parameter, die dynamisch gefüllt werden können
  + Verwende speziellen Code im Anfrage-String (?), der dann ersetzt wird; Identifikation über Position oder Bezeichner
* **DB arbeitet mit Relationen**
  + Relationen haben keine 1:1-Entsprechung in Java-Typen
  + Definition eines ResultSet in der API, das zeilenweise (Cursor-Prinzip) in Java verarbeitet werden kann
* **Übersetzung zwischen Datentypen** der DB und in Java durch spezifische Methoden in JDBC-API beim Setzen/Lesen von Werten

#### Ausführen einer statischen Anfrage

**SQL-Anfrage erzeugen:**

String query = "SELECT employee\_id, last\_name FROM hr.employees";

**Statement** stmt = conn.**createStatement**();

**SQL-Anfrage ausführen:**

**ResultSet** rs = stmt.**executeQuery**(query);

**Ergebnis verarbeiten:**

while (rs.next()) {  
 long empid = rs.**getLong**("employee\_id");  
 String lastname = rs.**getString**("last\_name");

System.out.println("Employee: ID="+empid+" LastName="+lastname);

}

#### Ausführen einer Anfrage

*Ressourcenfreigabe*

rs.**close**();

stmt.**close**();

Besser mit Auto-Closable:

String query = "SELECT employee\_id, last\_name FROM hr.employees";

try (**Statement** stmt = conn.**createStatement**()) {

try (**ResultSet** rs = stmt.**executeQuery**(query)) {

while (rs.next()) {

long empid = rs.**getLong**("employee\_id");

String lastname = rs.**getString**("last\_name");

System.out.println("Employee: ID="+empid+" LastName="+lastname);

}

}

} // Demo 1

#### Ausführen einer Anfrage

Nachbearbeitung/Einbettung

**Fehlerbehandlung**

* Fehler grundsätzlich als Ausnahmen der Klasse SQLException mit try/catch- Block abfangen
* Details zu einem aufgetretenen Fehler über getMessage:

try {

... Anweisungen, die eine SQLException erzeugen können

} catch (**SQLException** e) {  
 System.out.println("SQL-Fehler beim Ausführen der Anfrage: "+query);

System.out.println(e.**getMessage**());

}

#### Fehlerbehandlung

SQLException

Bei Ausführung eines SQL-Befehls wird im Fehlerfall eine Exception vom Typ SQLException geworfen

Für eine SQLException e gibt es drei Standard-Methoden

* e.getMessage(): String, der den Fehler beschreibt
* e.getSQLState(): identifiziert Fehler nach ANSI-92-SQL-Zustandscode
* e.getErrorCode(): identifiziert Fehler gemäß Datenbank-Hersteller

Beispiel: Select auf ungültige Spalte (SELECT photo FROM hr.employees)

• Message: ORA-00904: "PHOTO": ungültiger Bezeichner

• SQL-State: 42000

• ErrorCode: 904

#### Optimierung des DB-Zugriffs

Prepared Statements

**Prepared Statements (vorbereitete Anweisungen)**

* SQL-Befehl wird vor Ausführung zum RDBMS geschickt
  + vorkompilieren und (optimalen) Ausführungsplan erstellen
  + vorübersetzter Befehl mit unterschiedlichen Parametern ausführbar
  + Laufzeit-Vorteile (nur einmal geparst und auf Korrektheit geprüft)
* Verwendung von Parametern
  + Fragezeichen ? ist Platzhalter für **Werte**, nicht für **Tabellen**- oder **Spaltennamen** im SQL-Befehl
  + Werden durch eigenen Befehl gesetzt: setX( ) für jeden Standard-Java-Typ X
  + Einmal gesetzter Parameter behält Wert, bis er neu gesetzt wird

#### Ausführen einer parametrisierten Anfrage

SQL-Anfrage erzeugen:

String query = "SELECT employee\_id, last\_name FROM hr.employees

WHERE **department\_id = ?** AND **salary > ?**";

**PreparedStatement** stmt = conn.**prepareStatement**(query);

stmt.**setInt**(1, 80);  
 stmt.**setFloat**(2, 10000);  
 ResultSet rs = stmt.**executeQuery**();

// Demo 2

SQL-Anfrageergebnis verarbeiten wie zuvor

#### Übersetzung von Standard-Datentypen

**Mapping von Java-Typen auf SQL-Datentypen**

* Bsp.: String und VARCHAR bzw. int und NUMBER

**Handhabung von NULL-Werten**

* Java-Objekttypen können NULL als null darstellen
  + VARCHAR NULL wird als String null zurückgegeben
* Bei primitiven Datentypen (int etc.) geht das nicht, daher:

Beim Abfragen:

int deptid = rs.**getInt**("department\_id"); if (**rs.wasNull**()) { ... }

Beim Setzen von Werten:

stmt.**setNull**(1, Types.NUMERIC); // Demo 3 und 4

#### JDBC: Fallstricke

* Statement endet nicht mit " ; " als Terminator (wie im Oracle SQL Developer)
* SQL-Befehle sind nicht case-sensitiv Ausnahme (!): String-Literale, die in '...' stehen (z.B. in WHERE-Klausel)
* Zeilentrenner in Java ist nicht Zeilentrenner in SQL(!):

String query =

"SELECT employee\_id, last\_name" +

"FROM hr.employees";

### DDL und DML

* JDBC-Nutzung nicht nur für Anfragen (Data Query Language) möglich
* Es können auch Daten verändert (Data Manipulation Language) werden
* Es kann auch das DB-Schema bearbeitet werden (Data Definition Language)
* Konzepte der Anfragen werden im Wesentlichen übertragen

#### DDL-Befehle in Java

##### Erzeugen einer Tabelle

1. **Schritt: SQL-Befehl wird in einen String geschrieben**

String createOrderItems =

"CREATE TABLE order\_items( " +

"   order\_id NUMBER(8), " +

"   line\_item\_id NUMBER(5), " +

"   product\_name VARCHAR2(50), " +

"   unit\_price NUMBER(8,2), " +

"   quantity NUMBER(5), " +

"   PRIMARY KEY (order\_id, line\_item\_id) " +

")";

1. **Schritt: Statement-Objekt für Änderungen und Anfragen erzeugen**

Statement stmt = conn.**createStatement**();

* + Bezieht sich auf bestimmte Connection conn

1. **Schritt: Ausführen des SQL-Statements**

stmt.**executeUpdate**(createOrderItems); // Demo 5

**Bemerkungen**

* CREATE TABLE ist ein DDL-Befehl (wie ALTER TABLE, DROP TABLE, ....)
* Führt eine Änderung am Datenbank-Schema durch
* Deshalb wird executeUpdate(<SQL-String>) aufgerufen!
* Analog für andere DDL-Befehle und DML-Befehle (INSERT, UPDATE, DELETE)

#### DDL-Befehle in Java

*Vollständiges Beispiel (direkt in main nur zu Demo-Zwecken!)*

public class CreateTableDemo {

public static void main(String[] args) {

try (Connection conn = DriverManager.**getConnection**(...) ){

String createOrderItems =

"CREATE TABLE order\_items(" +

" order\_id NUMBER(8), " +

" line\_item\_id NUMBER(5)," +

" product\_name VARCHAR2(50)," +

" unit\_price NUMBER(8,2)," +

" quantity NUMBER(5)," +

" PRIMARY KEY (order\_id, line\_item\_id) " +

")";

try (Statement stmt = conn.**createStatement**()) {

stmt.**executeUpdate**(createOrderItems);

} catch (SQLException e) {

System.out.println("Fehler beim Ausführen einer Datenbank-Anfrage: ");

System.out.println("Message: " + e.getMessage());

System.out.println("Errorcode: " + e.getErrorCode());

}

}

}

} // Demo 5

### DML-Befehle in Java

#### Beispiel INSERT

Schritte genau wie bei CREATE TABLE

* String mit SQL-INSERT-Befehl füllen
* Statement-Objekt für ein Connection-Objekt erzeugen
* executeUpdate() für Statement-Objekt ausführen   
   (mit dem String als Parameter)
* liefert Anzahl der betroffenen Datensätze, bei INSERT also meist 1
* String-Literale werden in einfache Hochkommata ' ' eingeschlossen
* Kombination mit Parametern möglich wie zuvor

int num = stmt.**executeUpdate**("INSERT INTO orderItems VALUES "

+ "(123,12,'SampleItem1',48.32,12)");

#### Vollständiges Beispiel (direkt in main nur zu Demo-Zwecken!)

public class InsertDemo {

public static void main(String[] args) {

Connection conn = ...;

String insertItem1 = "INSERT INTO order\_items VALUES (123,12,'SampleItem1',48.32,12)";

String insertItem2 = "INSERT INTO order\_items VALUES (124,12,'SampleItem2',12.1,1)";

String insertItem3 = "INSERT INTO order\_items VALUES (125,14,'SampleItem3',0.89,200)";

try (Statement stmt = conn.createStatement()) {

int num = stmt.executeUpdate(insertItem1);

num += stmt.executeUpdate(insertItem2);  
 num += stmt.executeUpdate(insertItem3);  
 System.out.println("Tabelle orderItems "+num+" Zeilen eingefügt!");

}

}

} // Demo 6

#### Beispiel UPDATE

Änderungsoperation analog zu INSERT-Operation

Connection conn = ...;

String updateItem = "**UPDATE** orderItems **SET** unit\_price = 0.69, "

+ "quantity = 250 **WHERE** order\_id = ?";

try (PreparedStatement stmt = conn.**prepareStatement**(updateItem)) {

stmt.setInt(1, 123);

int num = stmt.**executeUpdate**(updateItem1);

System.out.println(

...

*Wichtig: Rückgabewert von executeUpdate(): Anzahl der geänderten Datensätze*

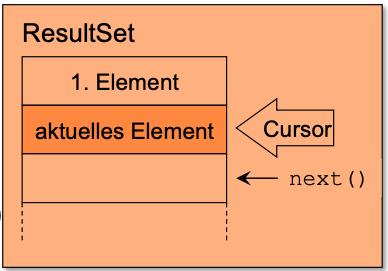
### DQL im Detail

#### SELECT im Detail

**Schritte analog**

* String mit SQL-SELECT-Befehl zusammenbauen
* Für Connection-Objekt ein Statement-Objekt erzeugen
* executeQuery()-Methode für Statement ausführen
* erzeugt Objekt vom Typ ResultSet und liefert es zurück

**ResultSet-Objekt**

* enthält Treffermenge des SELECT, d.h. Menge von Datensätzen mit den   
  selektierten Spalten (Ergebnisrelation)
* Methoden zum Iterieren der Datensätze verfügbar
* Methoden zum Zugriff auf einzelne Spalten verfügbar

**Iteration durch ResultSet**

* Jedes ResultSet-Objekt hat einen internen Cursor (Lesezeiger)
* Cursor zeigt auf aktuelle Zeile in Ergebnistabelle
* Methode boolean next()
* positioniert den Cursor auf nächsten Datensatz zum Zugriff
* erstes next() positioniert Cursor auf 1. Datensatz
* next() liefert false, wenn es keinen weiteren Datensatz im ResultSet gibt
* Weitere Methoden (Auszüge)
* previous eine Zeile rückwärts, liefert false vor der ersten Zeile
* first
* last

#### Vollständiges Beispiel→Zugriff auf Spalte per Name bzw. Position

try (Statement stmt = conn.createStatement()) {

String query = "**SELECT** first\_name, last\_name, salary "

+ " **FROM** hr.employees **WHERE** salary > 5000";

try (ResultSet rs = stmt.**executeQuery**(query)) {

while (rs.next()){

String last\_name = rs.**getString("last\_name");**

double sal = **rs.getDouble(3**)**;**

System.out.println(last\_name + "\t" + sal);

}

}

} catch(SQLException e) {

e.printStackTrace();

}

#### SELECT auch komplexer möglich

SELECT-Befehle mit Verbund

* Gesucht: Mitarbeitername mit zugehörigem Abteilungsnamen

String query = "**SELECT** first\_name, last\_name, department\_name FROM"

+ " hr.employees **LEFT OUTER JOIN** hr.departments"  
 + " **ON** (employees.department\_id = departments.department\_id)"

+ " **WHERE** salary > ?";

PreparedStatement stmt = conn.**prepareStatement**(query);

stmt.setDouble(1, 7000.0);  
 **ResultSet** rs = stmt.**executeQuery**();  
 String emp;

String dept;

while (rs.**next**()){

emp = rs.**getString**(1)+" "+rs.**getString**(2);

dept = rs.**getString**(3);

System.out.println(emp+" works in "+dept);

}

### Wichtige Methoden in Klassen

**Statement/PreparedStatement**

|  |  |  |
| --- | --- | --- |
| Methode | Parameter | Beschreibung |
| executeQuery | String | als Parameter übergebene SQL-Anfrage wird in der DB ausgeführt. Rückgabewert ist ein Objekt vom Typ  ResultSet  Parameterübergabe bei prepare bzw. execute |
| executeUpdate | String | als Parameter übergebener SQL-DML-Befehl (INSERT, UPDATE, DELETE) wird in der DB ausgeführt. Rückgabewert ist die Anzahl der eingefügten, geänderten bzw. gelöschten Datensätze  als Parameter übergebener SQL-DDL-Befehl wird in der DB ausgeführt und liefert Rückgabewert 0  Parameterübergabe bei prepare bzw. execute |
| close | keine | bewirkt Schließen des Statement-Objekts und eventuell vorhandenen ResultSet-Objekts: alle DB und JDBC- Ressourcen, die dadurch belegt waren, werden unmittelbar frei gegeben (bspw. Cursor) |

### Datentypen

#### Datenbank-spezifische Anpassungen

#### Konvertierung von Datentypen

Problem 1: Datenbanken haben unterschiedliche Datentypen und Namen dafür;

Typnamen werden bspw. in CREATE TABLE genutzt

Lösung

* Schreiben von DB-unabhängigem Programm-Code
* Datentypen benutzen, die allgemein verbreitet sind (INTEGER, NUMERIC, VARCHAR,...)
* Meta-Daten über angeschlossene Datenbank erfragen (welche Datentypen gibt es in angeschlossener DB?)

Problem 2: Java-Datentypen und RDBMS-Datentypen sind verschieden Lösung

* RDBMS 🡪 Java
* Konvertierung erfolgt über Zugriffsmethoden des ResultSet (getInt(..), getString(..), .... )
* Java 🡪 SQL
* String-Literale vermeiden (nicht portabel, fehlerträchtig!)
* PreparedStatements und Setter-Methoden verwenden (setInt(...), ...)

##### Beispiel DATE

**Version 1: Oracle-Spezifisch (nicht portabel):**

String insert1 = "**INSERT INTO** member **VALUES** (1, 'Fritz Meier', "+

**"to\_date**(**'1965-10-12','YYYY-MM-DD'**))";

stmt = conn.**createStatement**();

stmt.**executeUpdate**(insert1);

**Version 2: Prepared Statement (portabel):**

String insert2 = "**INSERT INTO** member **VALUES** (2, 'Franz Schulze', **?**)";

pstmt = conn.**prepareStatement**(insert2);  
 Date gebTag = java.sql.Date.valueOf("1972-10-17"); // java.sql.Date !

pstmt.**setDate**(1, gebTag);

pstmt.**executeUpdate**();

**Version 3: JDBC-Literal (portabel):**

String insert3 = "**INSERT INTO** member **VALUES** (3, 'Lena Müller', **{d'1985-02-07'}**)";

stmt = conn.**createStatement**();  
 stmt.**executeUpdate**(insert3); // Demo 7

#### Spezifische SQL-Syntax

* JDBC-Treiber geben die SQL-Anweisungen unverändert an die Datenbank weiter
* Ausnahme: Spezielle Ersetzungen, mit {} markiert (siehe Datumsliteral)
* Dies bedeutet: DBMS-spezifisches SQL ist immer möglich
* Dies ist allerdings nicht portabel!
* Beispiel:

SELECT sysdate, ... FROM member

* Funktioniert auf Oracle, aber nicht auf PostgreSQL
* Lösung: Wann immer möglich, Standard-SQL verwenden;  
  in diesem Fall: current\_date verwenden

Ein JDBC-Programm ist nur dann portabel, wenn die SQL-Statements auch portabel sind!

Verbleibendes Problem: SQL-Dialekte sind nicht vollständig kompatibel   
*Problematisch insb. bei speziellen Funktionen, z.B. Datumsumrechnungen:*

String query = "**SELECT** add\_months(dob, 1) **FROM** member";

stmt.**executeQuery**(query);

Obiges Beispiel funktioniert mit Oracle, aber nicht mit PostgreSQL

## Metadaten aus der DB in JDBC

* **Metadaten** sind Daten, die Datenbankstrukturen und deren Eigenschaften beschreiben
* Metadaten ermöglichen es allgemeinen Zugriffsschichten oder Werkzeugen, mit beliebigen **Datenbankstrukturen zu arbeiten**
* Bspw. zur Laufzeit **Informationen über** **ResultSet** erhalten
* Anwendungsbeispiel: SQL-Command-Line-Interpreter
  + erhält zur Laufzeit einen SQL-Befehl als String, soll das Ergebnis anzeigen
* Wie viele Spalten hat ein ResultSet?
* Was ist der Name und was der Datentyp einer Spalte?
* Von welcher Tabelle stammt eine Spalte?
* ....
* Bspw. zur Laufzeit **Informationen über Datenbank** erhalten
* Anwendungsbeispiel: Datenbank-Browser
* Welche Tabellen gibt es?
* Was ist der Primärschlüssel für eine Tabelle?

JDBC verfügt über 2 Klassen für Metadaten

* ResultSetMetaData liefert Informationen zu ResultSet
* DatabaseMetaData liefert Informationen zu Datenbanksystem und DB-Schema (Methode in Connection)

Informationen zum ResultSet: Klasse ResultSetMetaData

* Liefert Methoden, um ResultSet zu analysieren
* Notwendig bei Anfragen, die erst zur Laufzeit erzeugt werden
* Erzeugung eines Objektes vom Typ ResultSetMetaData über statische Methode getMetaData() der Klasse ResultSet

stmt = conn.createStatement();

stm ResultSet rs = stmt.executeQuery(query);

**ResultSetMetaData** rsmd = **rs.getMetaData**();

### Metadaten des Result-Set

Methoden von ResultSetMetaData

* Anzahl der Spalten im ResultSet: int getColumnCount()
* rs.getString(i) liefert den Inhalt einer Spalte beliebigen Typs als Java-String zurück
* Beispiel: unbekanntes ResultSet ausgeben:

stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery(query);

**ResultSetMetaData** rsmd = rs.**getMetaData**();

int numberOfColumns = rsmd.**getColumnCount**(); // Ausdruck eines beliebigen ResultSets

while (rs.next()) {

for (int i = 1; i <= numberOfColumns; i++)

System.out.print(rs.getString(i) + "\t");

System.out.println();

} //Demo8

Typinformationen über Spalten im ResultSet

* Name einer Spalte: String getColumnName(int column)
* Bsp.: Spaltennamen in einer Titelzeile ausdrucken

for (int i = 1; i <= numberOfColumns; i++) {

System.out.print(rsmd.**getColumnName**(i) + "\t");

}

* Datentyp einer Spalte: int getColumnType(int column) liefert int-Zahl entsprechend java.sql.Types (JDBC-Datentyp)
* Von der DB benutzter Name des Datentyps einer Spalte:  
  String getColumnTypeName(int column)

for (int i = 1; i <= numberOfColumns; i++) {

System.out.print(rsmd.**getColumnTypeName**(i) + "\t");

}

* Anzahl der Ziffern eines numerischen Datentyps:

int getPrecision(int column)

* Anzahl der Nachkommastellen eines numerischen Datentyps:

int getScale(int column)

* müssen in eine Spalte Werte eingetragen werden:

boolean isNullable(int column)

* ... (siehe JDBC-API)

### Metadaten zum DB-Schema

Beispielmethoden der Klasse DatabaseMetaData (Details und Weiteres siehe JDBC-API):

|  |  |
| --- | --- |
| Signatur | Beschreibung |
| ResultSet getCatalogs() | Gibt verfügbare Katalognamen zurück |
| ResultSet getSchemas() | Gibt verfügbare Schemanamen zurück |
| ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types) | Gibt eine Beschreibung der passenden Tabellen zurück (Ergebnis hat 10 Spalten mit Details zu den Tabellen) |
| boolean supportsX()  Beispiel: supportsFullOuterJoins() | Gibt zurück, ob die DB eine spezifische Funktionalität X besitzt |
| ResultSet getPrimaryKeys(String catalog, String schema, String table) | Gibt eine Beschreibung der Primärschlüssel passender Tabellen zurück |

**Zusammenfassung**

* Unterschiedliche Methoden zum Zugriff auf Daten in einer DB aus Programmen möglich
* Häufigste Variante: Client-Treiber, der Zugriffe in nativer DB-Sprache in Programmiersprache kapselt  
  → hier: Java und JDBC
* JDBC Ausführung klassischer Datenbankoperationen
* Auslesen von Metadaten zur Datenbank in einer Java-Anwendung