HTL LEONDING

# JDBC
# Java Database Connectivity


Informatik


Medientechnik


Elektronik - Technische Informatik


Medizintechnik

# Relationale Datenbanksysteme (RDBMS)

- Ziele

  Speichern, Abfragen und Verändern von Daten in Tabellenform auf einem DB-Server

- Eigenschaften von RDBMS
  - Deklarative Abfragesprache (SQL)
  - (Paralleler) Mehrbenutzer-Zugriff
  - Rechteverwaltung für zugreifende Anwender
  - Strukturierung der Daten (über Datenbank-Schema)
  - Konsistenzgarantien für gespeicherte Daten (über Schema)
  - Konsistenzgarantien für Änderungsprozesse auf den Daten (Transaktionen)
  - Dauerhaftigkeit von Änderungen (Persistenz)
  - Wiederanlauf bei Systemabstürzen

# Tabellen/Relationen in RDBMS

- Speichern, Anfragen und Verändern von Daten in Tabellenform
- Tabelle/Relation hat Namen und feste Menge benannter Spalten
- Spalten assoziiert mit Wertemengen (Typen)
- Pro Zeile einer Tabelle ein Werte für jede Spalte
- Der Wert ist aus der Wertmenge der jeweiligen Spalte

# Bsp: Relation Product, DDL

| Spaltenname | Wertemenge (Typ) |
|---|---|
| productID | Int |
| productName | varchar(60) |
| categoryID | int |
| unitPrice | float |
| soldIn | int |

```
CREATE TABLE Products (
    productID int NOT NULL,
    productName varchar (60) NOT NULL,
    categoryID int,
    unitPrice float,
    soldIn int
);
```

# Schlüssel

- Identifikation der Zeilen einer Tabelle
- Entsprechende Spaltenwerte für alle Tupel der Tabelle unterschiedlich (TSA)
  - Spalten bei denen diese Bedingung erfüllt ist, nennt man Schlüssel
  - Häufig sind künstliche Schlüssel (Surrogate) sinnvoll
- Schlüssel können in der DDL deklariert werden
  - Das Datenbanksystem überwacht Schlüsselbedingung

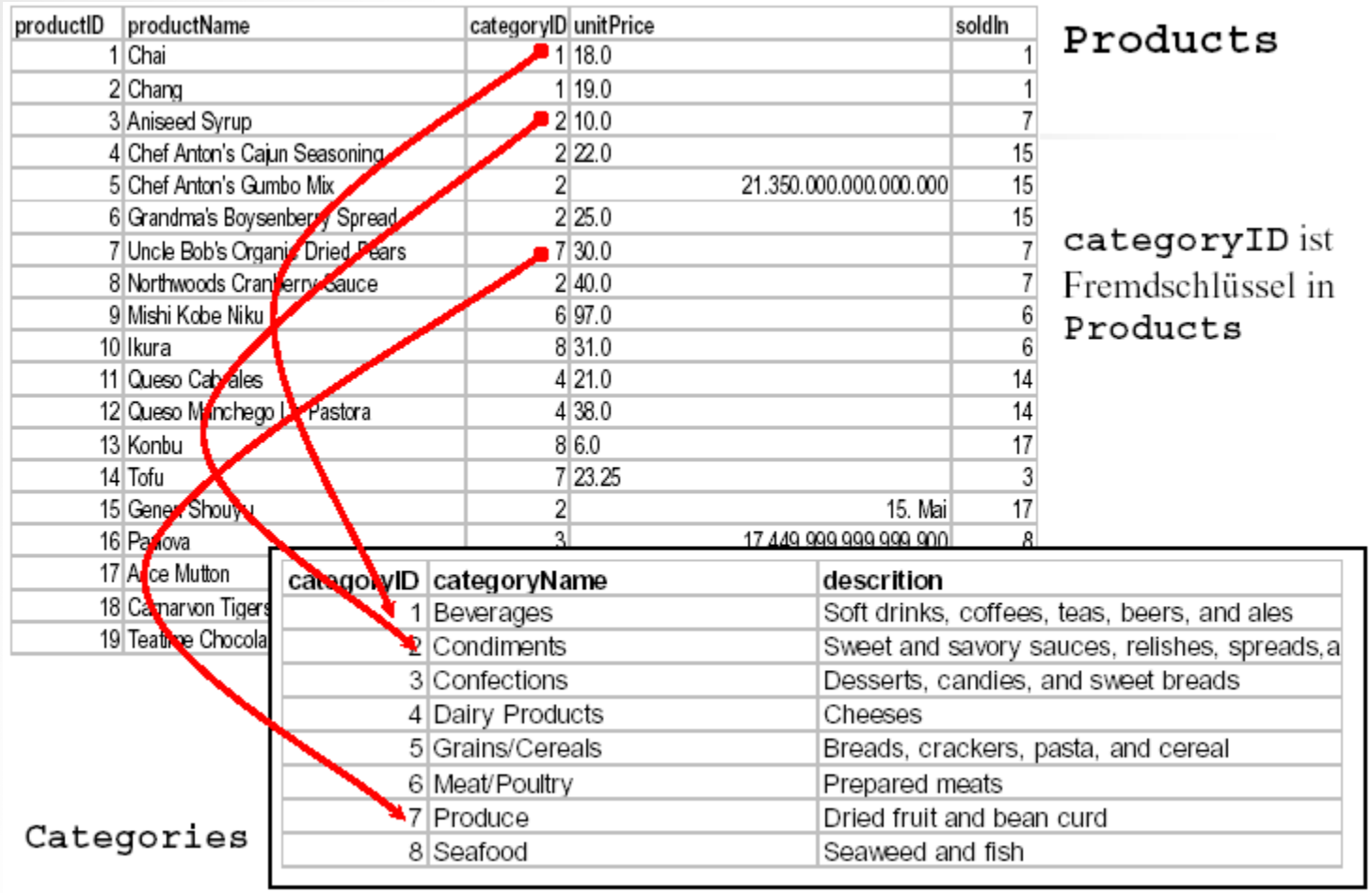
# DDL Programm für Tabelle Products (mit Schlüssel)

```
CREATE TABLE Products (
    PRIMARY KEY (productID),
    UNIQUE (productName),
    productID int NOT NULL,
    productName varchar (60) NOT NULL,
    categoryID int,
    unitPrice float,
    soldIn int
);
```

# Fremdschlüssel

- Verweis in Tabelle B auf einen Schlüssel in anderer Tabelle
- Tupel aus B eindeutig kann ein Tupel aus A referenzieren
- B besitzt dann einen sogenannten Fremdschlüssel
- Schlüsselwert muss in A existieren (Fremdschlüsselbedingung)
  - Essentiell um komplexe Strukturen in relationalen RDB aufzubauen
  - Kann mit der DDL ausgedrückt werden
  - Ähnlich dem Zeiger- oder Referenzkonzept aus OO

# Beispiel: Kategorie eines Produkts



| productID | productName | categoryID | unitPrice | soldIn |
|---|---|---|---|---|
| 1 | Chai | 1 | 18.0 | 1 |
| 2 | Chang | 1 | 19.0 | 1 |
| 3 | Aniseed Syrup | 2 | 10.0 | 7 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 22.0 | 15 |
| 5 | Chef Anton's Gumbo Mix | 2 | 21.350.000.000.000.000 | 15 |
| 6 | Grandma's Boysenberry Spread | 2 | 25.0 | 15 |
| 7 | Uncle Bob's Organic Dried Pears | 7 | 30.0 | 7 |
| 8 | Northwoods Cranberry Sauce | 2 | 40.0 | 7 |
| 9 | Mishi Kobe Niku | 6 | 97.0 | 6 |
| 10 | Ikura | 8 | 31.0 | 6 |
| 11 | Queso Cabrales | 4 | 21.0 | 14 |
| 12 | Queso Manchego La Pastora | 4 | 38.0 | 14 |
| 13 | Konbu | 8 | 6.0 | 17 |
| 14 | Tofu | 7 | 23.25 | 3 |
| 15 | Genen Shouyu | 2 | 15. Mai | 17 |
| 16 | Pavlova | 3 | 17.449.999.999.999.900 | 8 |
| 17 | Alice Mutton | | | |
| 18 | Carnarvon Tigers | | | |
| 19 | Teatime Chocolate | | | |

Products

categoryID ist Fremdschlüssel in Products

| categoryID | categoryName | descrition |
|---|---|---|
| 1 | Beverages | Soft drinks, coffees, teas, beers, and ales |
| 2 | Condiments | Sweet and savory sauces, relishes, spreads, a |
| 3 | Confections | Desserts, candies, and sweet breads |
| 4 | Dairy Products | Cheeses |
| 5 | Grains/Cereals | Breads, crackers, pasta, and cereal |
| 6 | Meat/Poultry | Prepared meats |
| 7 | Produce | Dried fruit and bean curd |
| 8 | Seafood | Seaweed and fish |

Categories

# DDL-Programm (Categories/Products mit Fremdschlüsseln)

```
CREATE TABLE categories (
    PRIMARY KEY (categoryID),
    categoryID int NOT NULL,
    categoryName varchar(40) NOT NULL,
    description varchar (80)
   );
CREATE TABLE products (
    PRIMARY KEY (productID),
    UNIQUE (productName),
    productID int NOT NULL,
    productName varchar (60) NOT NULL,
    categoryID int,
    unitPrice float,
    soldIn int,
    CONSTRAINT FK_products_category
    FOREIGN KEY (categoryID)
    REFERENCES categories(categoryID)
);
```

# SQL – Structured Query Language

- SQL: Anfrage- und Änderungssprache für relationale Datenbanken
- DDL: Data Definition Language
- DML: Data Manipulation Language zum Zugriff auf Daten
- SQL ist standardisiert (Konformität ?!)

# SQL – 4 Anweisungstypen

- SELECT für lesende Anfragen
- INSERT um Tupel einzufügen
- UPDATE um existierende Tupel zu verändern
- DELETE um Tupel zu löschen

# SELECT – Beispiele (1)

- SELECT * FROM Products;

| productID | productName | categoryID | unitPrice | soldIn |
|---|---|---|---|---|
| 1 | Chai | 1 | 18.0 | 1 |
| 2 | Chang | 1 | 19.0 | 1 |
| 3 | Aniseed Syrup | 2 | 10.0 | 7 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 22.0 | 15 |
| 5 | Chef Anton's Gumbo Mix | 2 | 21 | 15 |
| 6 | Grandma's Boysenberry Spread | 2 | 25.0 | 15 |
| 7 | Uncle Bob's Organic Dried Pears | 7 | 30.0 | 7 |
| 8 | Northwoods Cranberry Sauce | 2 | 40.0 | 7 |
| 9 | Mishi Kobe Niku | 6 | 97.0 | 6 |
| 10 | Ikura | 8 | 31.0 | 6 |
| 11 | Queso Cabrales | 4 | 21.0 | 14 |
| 12 | Queso Manchego La Pastora | 4 | 38.0 | 14 |
| 13 | Konbu | 8 | 6.0 | 17 |
| 14 | Tofu | 7 | 23.25 | 3 |
| 15 | Genen Shouyu | 2 | 15. Jan | 17 |
| 16 | Pavlova | 3 | 17.449.999.999.999.900 | 8 |
| 17 | Alice Mutton | 6 | 39.0 | 15 |
| 18 | Carnarvon Tigers | 8 | 62.5 | 9 |
| 19 | Teatime Chocolate Biscuits | 3 | 91.999.999.999.999.900 | 7 |

# SELECT – Beispiele (2): Projektion

- Projektion auf eine Spalte:
  - SELECT categoryName FROM Categories;

| categoryName |
| --- |
| Beverages |
| Condiments |
| Confections |
| Dairy Products |
| Grains/Cereals |
| Meat/Poultry |
| Produce |
| Seafood |

# SELECT – Beispiele (3): Selektion

- Alle Produkte teurer als 40...
  - SELECT * FROM Products WHERE unitPrice > 40;

| productID | productName | categoryID | unitPrice | soldIn |
|---|---|---|---|---|
| 9 | Mishi Kobe Niku | 6 | 97.0 | 6 |
| 18 | Carnarvon Tigers | 8 | 62.5 | 9 |
| 20 | Sir Rodney's Marmalade | 3 | 81.0 | 15 |
| 27 | Schoggi Schokolade | 3 | 44 | 3 |
| 28 | Rössle Sauerkraut | 7 | 45.6 | 3 |
| 29 | Thüringer Rostbratwurst | 6 | 124 | 3 |
| 38 | Côte de Blaye | 1 | 263.5 | 5 |
| 43 | Ipoh Coffee | 1 | 46.0 | 7 |
| 51 | Manjimup Dried Apples | 7 | 53.0 | 4 |
| 59 | Raclette Courdavault | 4 | 55.0 | 9 |
| 62 | Tarte au sucre | 3 | 49 | 9 |
| 63 | Vegie-spread | 2 | 44 | 15 |

# SELECT – Beispiele (4): Sortieren

- Name und Preis aller Produkte, die mehr als 40 kosten nach Preisen geordnet:

```
SELECT productName, unitPrice FROM Products
WHERE unitprice > 40 ORDER BY unitPrice;
```

| productName | unitPrice |
|---|---|
| Schoggi Schokolade | 44 |
| Vegie-spread | 44 |
| Rössle Sauerkraut | 46 |
| Ipoh Coffee | 46.0 |
| Tarte au sucre | 49 |
| Manjimup Dried Apples | 53.0 |
| Raclette Courdavault | 55.0 |
| Carnarvon Tigers | 62.5 |
| Sir Rodney's Marmalade | 81.0 |
| Mishi Kobe Niku | 97.0 |
| Thüringer Rostbratwurst | 124 |
| Côte de Blaye | 263.5 |

# SELECT – Beispiele (5): Aggregation

- Alle Namen und Preise des teuersten Produkts:

```
SELECT productName, unitPrice
    FROM Products
    WHERE unitPrice IN
    (SELECT MAX(unitPrice) FROM products)
```

| Côte de Blaye | 263.5 |
|---|---|

- Anzahl der Produkte:

```
SELECT COUNT(*) FROM Products
```

| 77 |
|---|

# SELECT – Beispiele (6): Duplikateliminierung

- Mehrfach-Einträge der selben Tupel Selektion
- Eliminierung von Duplikaten in Anfragen durch DISTINCT
  - Alle Preise (geordnet, ohne Duplikate):

```
SELECT DISTINCT unitPrice FROM Products
ORDER BY unitPrice;
```

# Joins auf Tabellen

- Voriges Beispiel hat zwei Tabellen mit einbezogen
  - („FROM Products, Categories“)
- Zuerst kartesisches Produkt der Tupel aus Products und Categories
  - (jedes Tupel aus P. mit jedem Tupel aus C.)
- Dann Selektion der Ergebnisse mit WHERE-Klausel („WHERE Products.categoryID = Categories.categoryID“)
- Hintergründe
  - Möglichkeit der verknüpften Anfrage nennt man Join oder Verbindung
  - Anfrage nutzt Fremdschlüsseleigenschaft für Products.categoryID
  - Dadurch sinnvolles Resultat: Die jeweilige Kategorie zu einem Produkt
  - Ähnlich der Derefenzierung (Verfolgen von Referenzen) in OO

# Join: Beispiel

- Alle Namen aller Produkte und der zugeordneten Kategorien für Produkt teurer als 30:

```
SELECT productName, categoryName
FROM Products, Categories
WHERE Products.categoryID = Categories.categoryID
AND Products.unitPrice > 30;
```

| productName | categoryName |
|---|---|
| Northwoods Cranberry Sauce | Condiments |
| Mishi Kobe Niku | Meat/Poultry |
| Ikura | Seafood |
| Queso Manchego La Pastora | Dairy Products |
| Alice Mutton | Meat/Poultry |
| Carnarvon Tigers | Seafood |
| Sir Rodney's Marmalade | Confections |
| Gumbär Gummibärchen | Confections |
| Schoggi Schokolade | Confections |
| Rössle Sauerkraut | Produce |

# INSERT – Beispiel

- Achtung: der erste Wert muss die Schlüsselbedingung erfüllen!

```
INSERT INTO Categories
(categoryID, categoryName, descrition)
VALUES (4711, 'Spirituosen','hochprozentiges...')
```

| categoryID | categoryName | descrition |
|---|---|---|
| 1 | Beverages | Soft drinks, coffees, teas, beers, and ales |
| 2 | Condiments | Sweet and savory sauces, relishes, spreads, |
| 3 | Confections | Desserts, candies, and sweet breads |
| 4 | Dairy Products | Cheeses |
| 5 | Grains/Cereals | Breads, crackers, pasta, and cereal |
| 6 | Meat/Poultry | Prepared meats |
| 7 | Produce | Dried fruit and bean curd |
| 8 | Seafood | Seaweed and fish |
| 4711 | Spirituosen | hochprozentiges... |

# DELETE

- Löscht Tabellen-Einträge
  - Ähnlich wie SELECT aber keine Joins und Projektionen...
- DELETE * from Categories
  - löscht alle Tupel in Categories
  - Fremdschlüsselbedingung  muss erfüllt sein
- DELETE * FROM Product WHERE unitPrice > 30
  - löscht alle Tupel in Products, die teurer als 30 sind...
  - hier natürlich keine Probleme mit Fremdschlüsseln

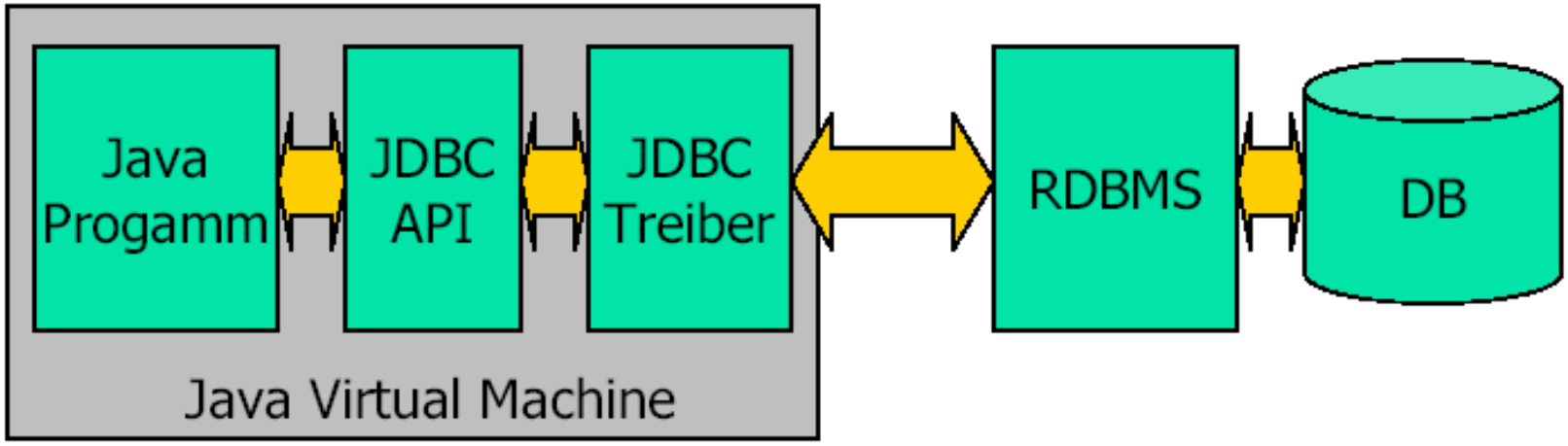
# UPDATE

Erhöhe den Preis bei Produkten teurer als 30 um 10%:

```
UPDATE Products
SET unitPrice = unitPrice * 1.1
WHERE unitPrice > 30
```

# Grundsätzliche Funktionsweise JDBC

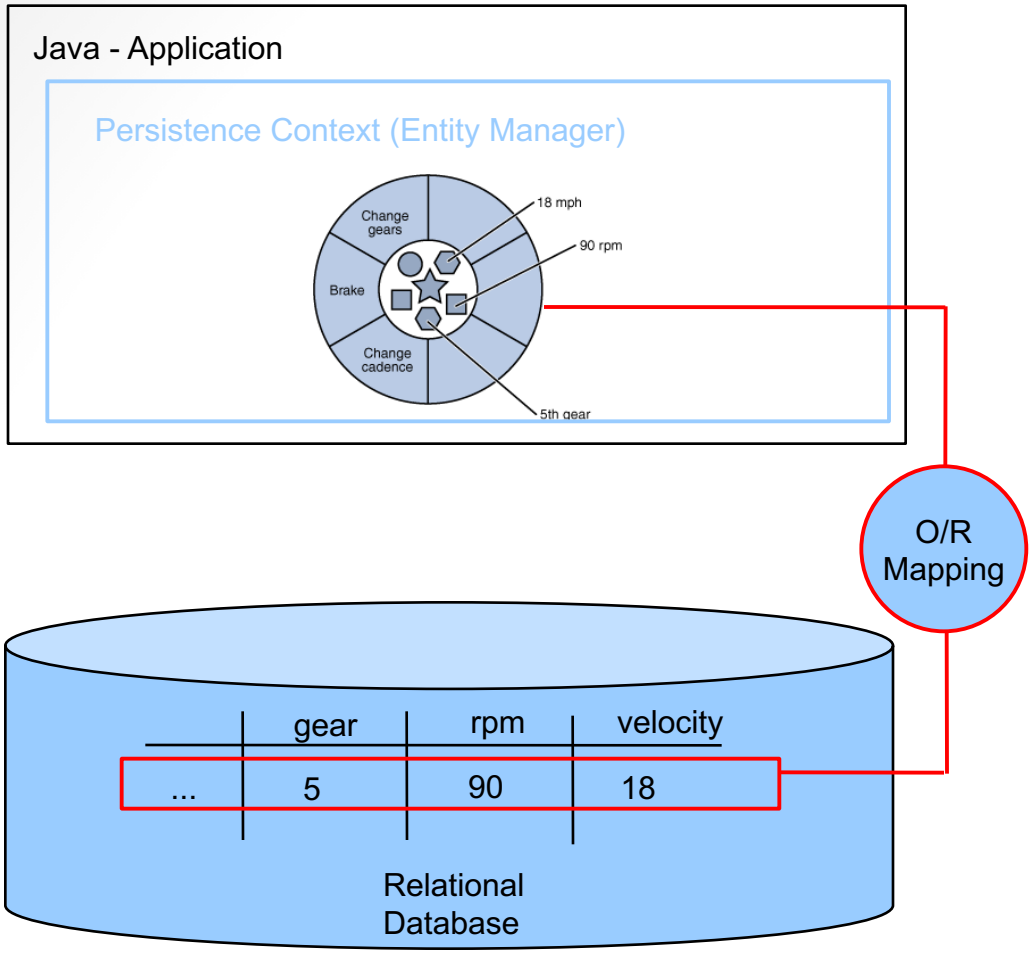
- Verwendung des JDBC API aus einem Java Programm
  - JDBC Treiber kommuniziert mit RDBMS
  - Treiber schickt SQL-Kommandos an DB
  - Gibt Resultate als Ergebnisse von Methodenaufrufen des JDBC API an das Java Programm zurück






HTL LEONDING

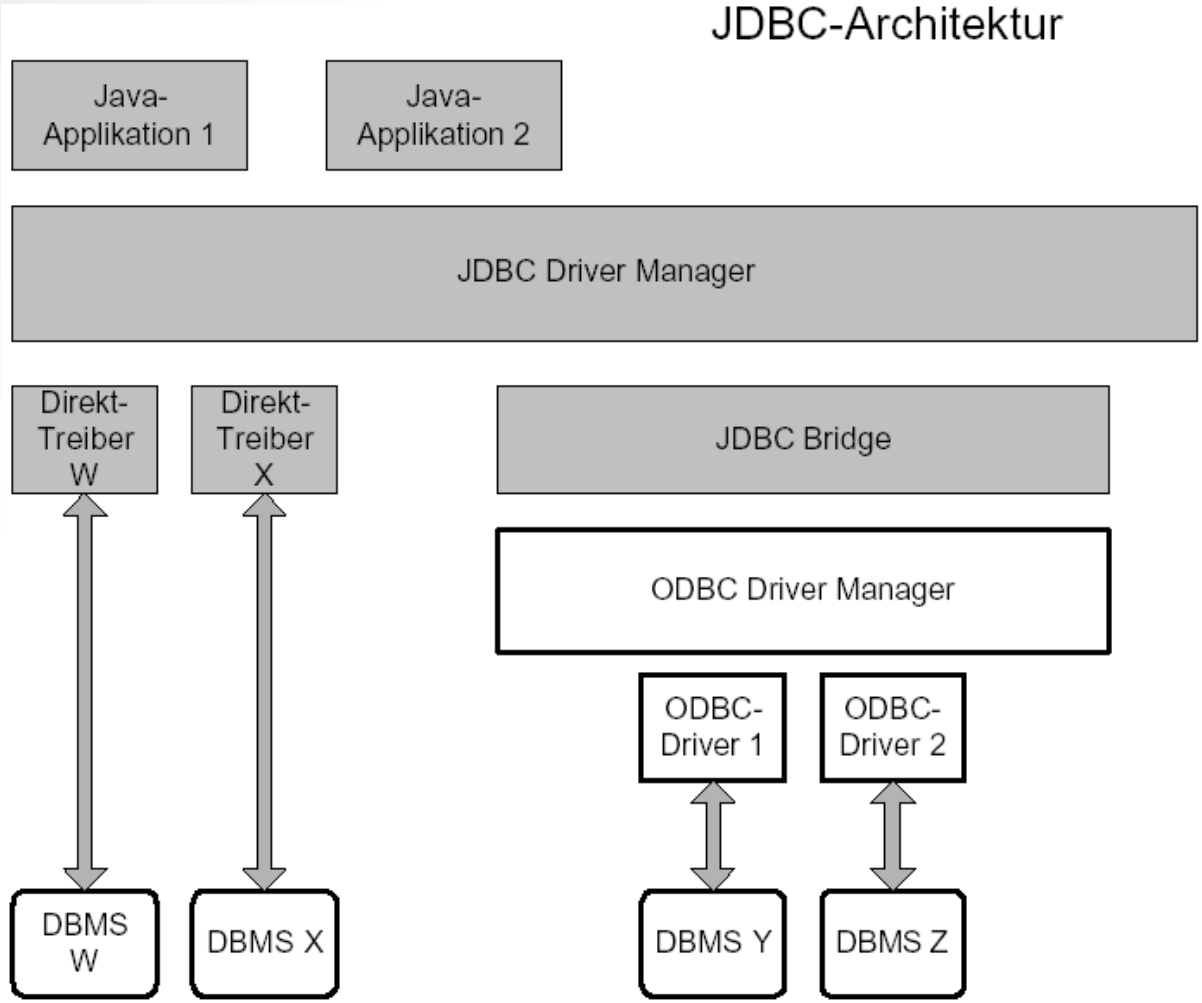
# „Modernere“ Zugriffstechnologie JPA






HTL LEONDING

# JDBC-Architektur



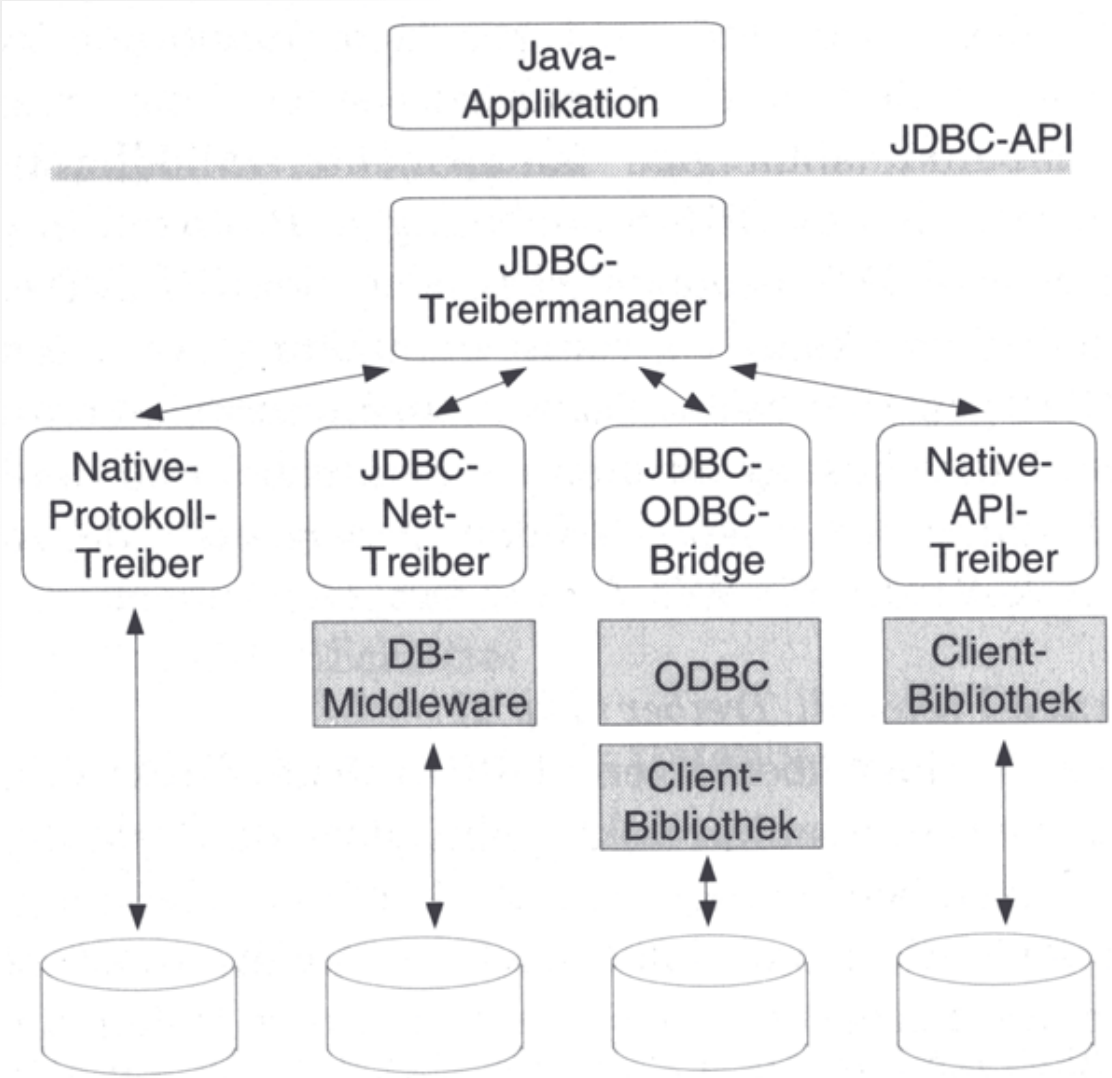



HTL LEONDING

# JDBC-Treiber

- Protokoll zur Kommunikation mit dem RDBMS
  - Hängt vom spezifischen RDBMS-Produkt ab
- Für jedes RDBMS-Produkt einen spezifischen JDBC Treiber
- JDBC-API abstrahiert von Treiber und RDBMS (Klassen und Schnittstellen immer gleich)
- (In Strings) eingebettetes SQL bleibt jedoch RDBMS spezifisch!
- Zu Beginn im Java Programm festlegen, welchen JDBC Treiber man braucht
- JDBC-Treiber werden meist als Java-Bibliotheken (.jar-Files) angeboten

# JDBC-Treibertypen



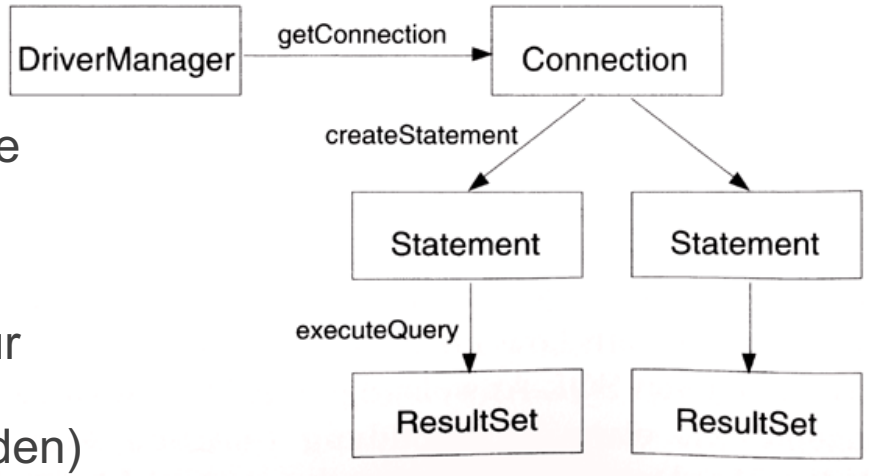



HTL LEONDING

# Ablauf eines JDBC Programms

1. JDBC Teiber laden
2. Verbindung zur Datenbank herstellen
3. SQL-Kommandos ausführen
4. Verbindung zur Datenbank schließen

# Wichtige JDBC-Klassen




- java.sql.Driver
  - Repräsentiert den Treiber um eine Datenbank anzusteuern
- java.sql.Connection
  - Repräsentiert eine Verbindung zur Datenbank (über sie können SQL-Kommandos abgesetzt werden)
- java.sql.Statement
  - Repräsentiert ein konkretes SQL-Kommando
  - Der eigentliche SQL-Ausdruck wird als Java-String übergeben
- java.sql.ResultSet
  - Das Resultat einer SQL-Anfrage (im Prinzip also die Ergebnistabelle), die man in einer Schleife zeilenweise lesen kann
- java.sql.SQLException
  - Bei fehlerhaften JDBC-Aufrufen

# Datenbankverbindung aufbauen

**Connection connection = java.sql.DriverManager.**
**getConnection("<JDBC URL>", user, password);**

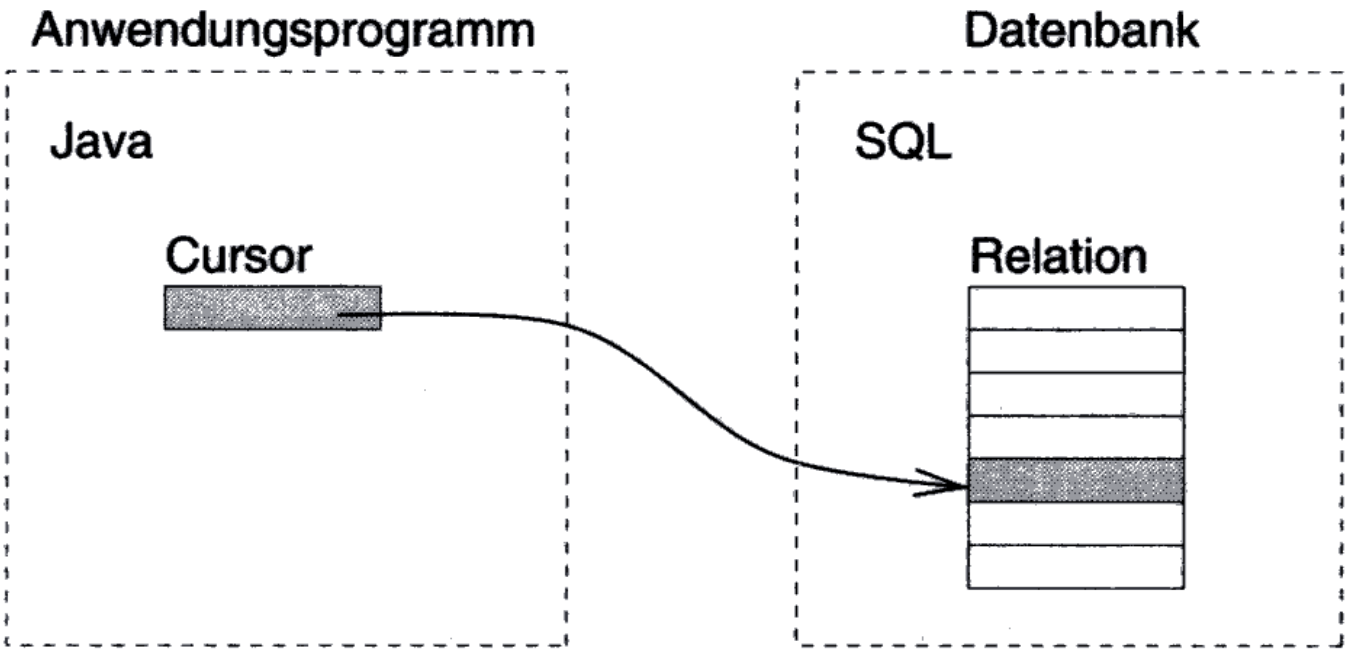
- **connection**-Objekt repräsentiert Verbindung
- Die **JDBC URL spezifiziert** eindeutig die **Verbindung** zur Datenbank
- **user** und **password** Strings falls die Datenbank **Authentifizierung** erfordert

# Statements erzeugen und verwenden

```
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery("SELECT
    productName FROM Products");
```

- Statements erhält man von **connection**-Objekten
  - Zum Ausführen von SQL Kommandos
- **executeQuery("SELECT... ")** liefert **Anfrage-Ergebnis-Objekt** (also für SELECT-Anfragen)
- **executeUpdate("UPDATE... ")** liefert **int** mit Anzahl der geänderten Tupel (also für UPDATE- und DELETE-Anfragen)
- **close() schließt Statement** und gibt entsprechende Ressourcen frei

# ResultSet: Konzept des Cursors



# Zeilenelemente aus ResultSet lesen

```
while (resultSet.next())
{
   String productName = resultSet.getString(1);
   String sameProductName = resultSet.getString("productName");
   System.out.println(productName);
}
```

- Mit **getString(...), getInt(...), getDouble(...)** … Werte einer Ergebniszeile auslesen
- **Automatische Konvertierung von SQL-Spaltenwerten in Java Datentypen**
- Zum Beispiel **varchar** (SQL) nach **String** (Java)
- **Wert** aus einer Spalte kann über
  - **Spaltenname (productName)**
  - **Stelligkeit der Ergebnisspalte (ab 1) addressiert werden**
- Parameter von **getXXX()**
- **next()** springt man in die nächste Ergebniszeile...

# Prepared Statements (1)

```
PreparedStatement statement =
   connection.prepareStatement(
   "SELECT productName FROM Products " +
   "WHERE unitPrice >= ?");
statement.setDouble(1, 30);
ResultSet resultSet = statement.execute();
```

- Prepared Statements sind parametrisierbare Statements
- Die Parameter sind in der SQL-Anfrage durch **?** markiert
- Sie müssen vor Anfrage-Ausführung durch konkreten Wert ersetzt werden

# Prepared Statements (2)

```
PreparedStatement statement =
   connection.prepareStatement(
   "SELECT productName FROM Products " +
   "WHERE unitPrice >= ?");
statement.setDouble(1, 30);
ResultSet resultSet = statement.execute();
```

- **setXXX(i, v)** setzt den Wert **v** für das **i**-te **?** ein (**i** >= 1)
- **XXX** ist Java-Typ
  - Implizite Konvertierung von Java-Typen nach SQL Typen
  - Im Beispiel **double** (Java) nach **float** (SQL)
- **execute()** führt die Anfrage aus (Ergebnis ist **ResultSet** – wie bei **executeQuery()** ...)
  - Zuvor müssen alle **?** aus der Anfrage mit **setXXX()** ersetzt sein!

# Wozu Prepared Statements?

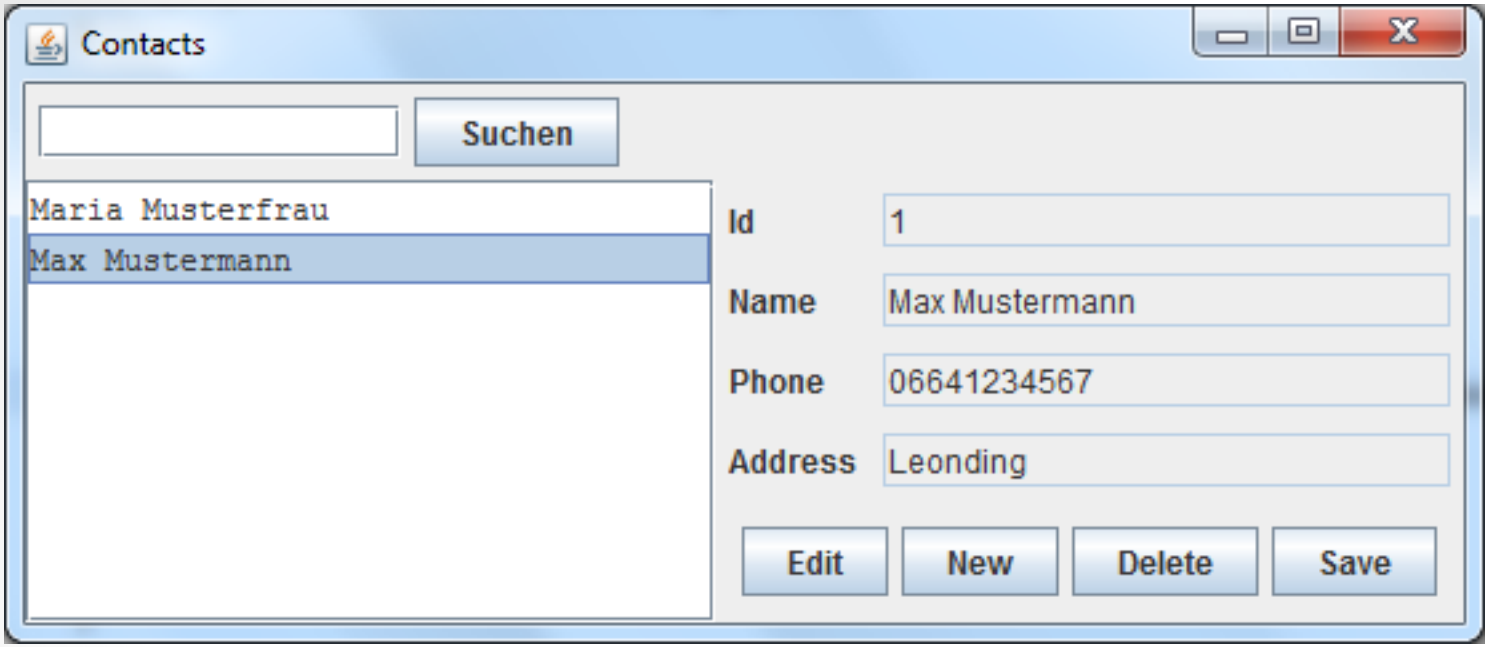
- Effizienz:
  - Prepared Statements können (evtl.) durch das RDBMS vorverarbeitet werden
  - Die Kosten pro Anfrage-Ausführung sinken
- Sauberer Code
  - Es muss nicht immer ein neuer String für jede SQL-Anfrage erzeugt werden
  - Erzeugung und Ausführung der Statements leicht trennbar
  - Einflussfaktoren auf Anfrage klar erkennbar durch **?**
    - (**unitPreis** im Beispiel)

# JDBC Exceptions

- **java.sql.SQLException** kann von so ziemlich jeder Methode unter **java.sql.*** geworfen werden!

- Die häufigsten Gründe:
  - Syntaktische und semantische Korrektheit der SQLStrings
    - kann zur (Java-)Übersetzungszeit nicht erkannt werden
    - Hier hilft nur Ausführen und Daumen drücken
  - Meldungen sind durch den JDBC-Treiber und das RDBMS bestimmt
    - Leider nicht immer aussagekräftig...

# Beispiel: Addressbuch mit DB-Anbindung

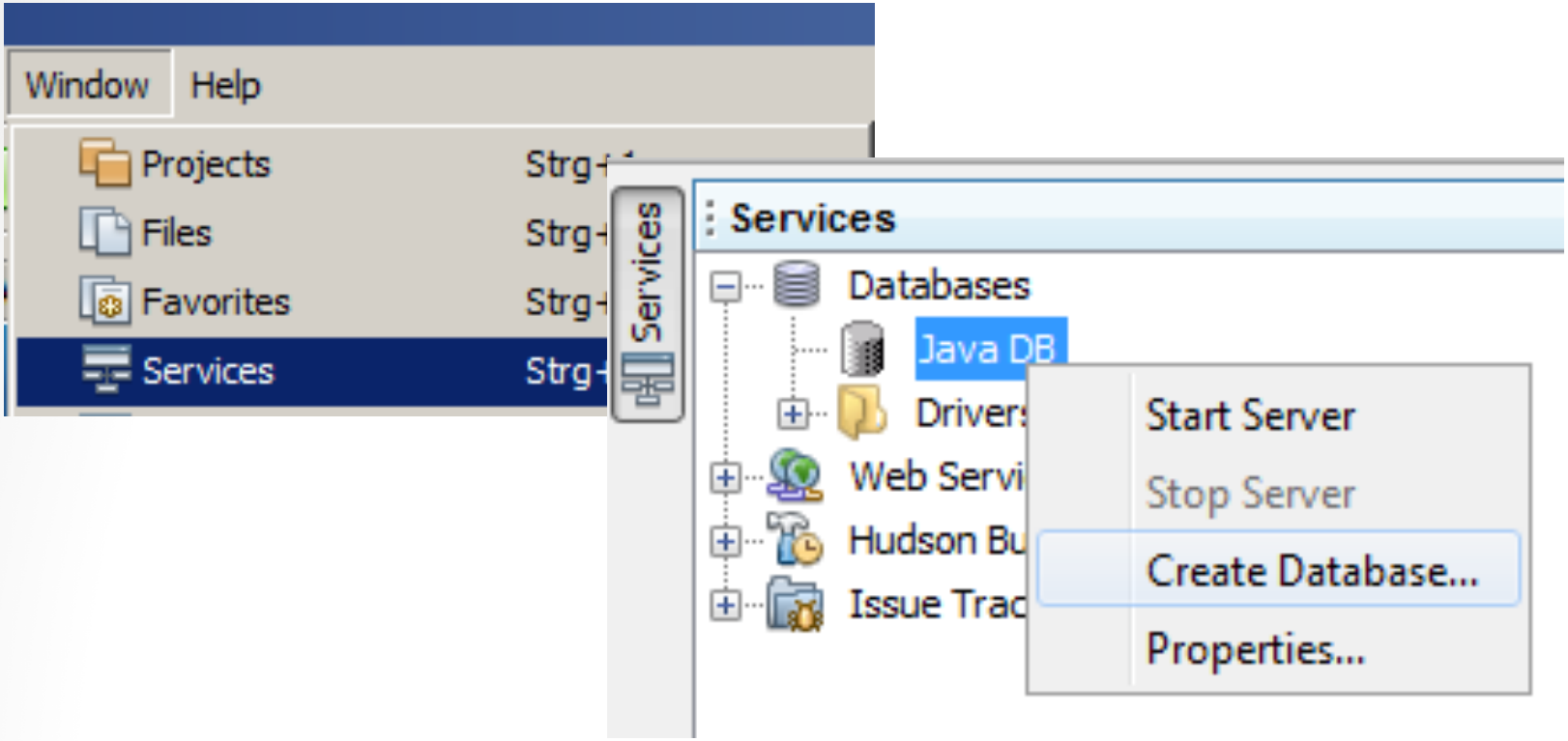





HTL LEONDING

# Datenbank anlegen 1/2

- Window-Services anzeigen lassen
- Wir verwenden Java-DB (Derby)
- Integrierte Managementoberfläche

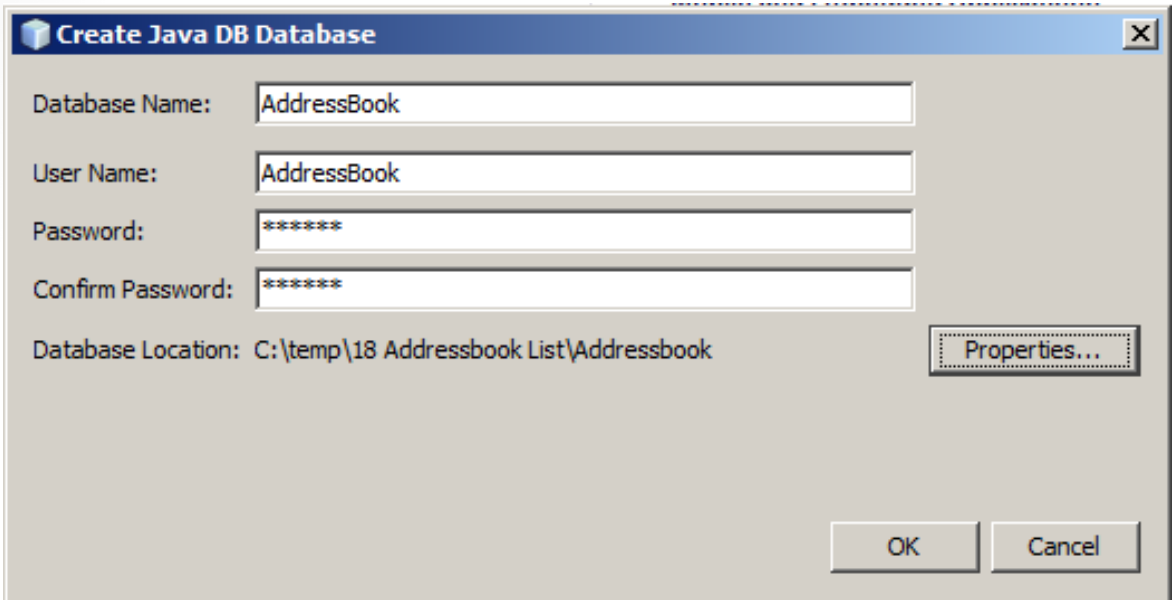





HTL LEONDING

# Datenbank anlegen 2/2

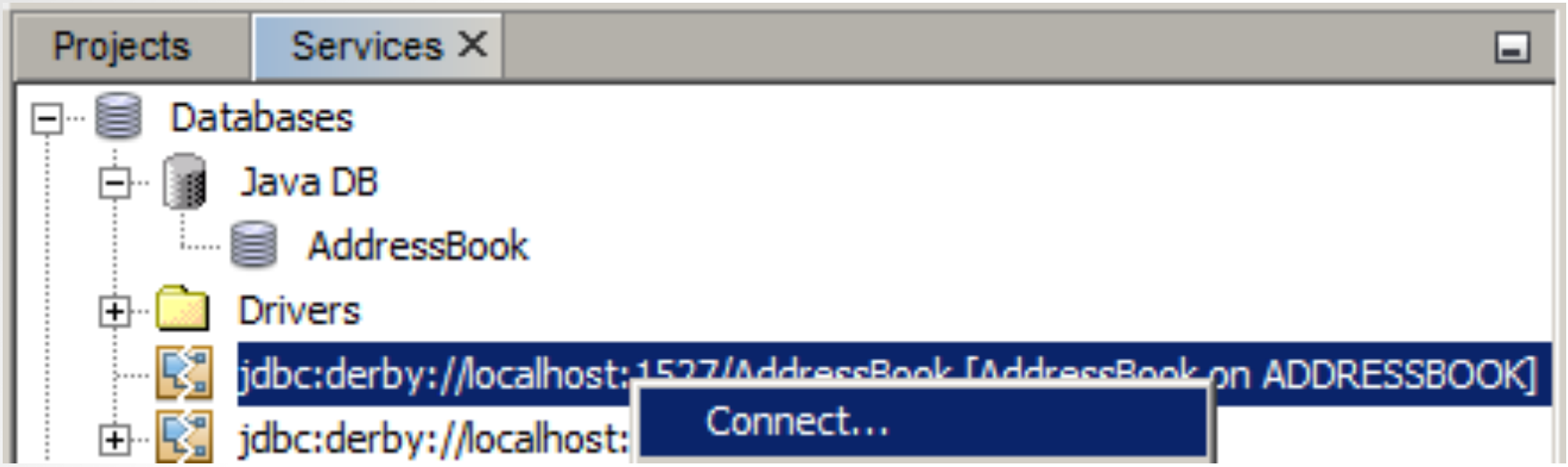
- DataBase Name: Anwendungsname
- User/Password für Testzwecke: [Anwendungsname]/passme
- Speicherort: Projektverzeichnis






HTL LEONDING

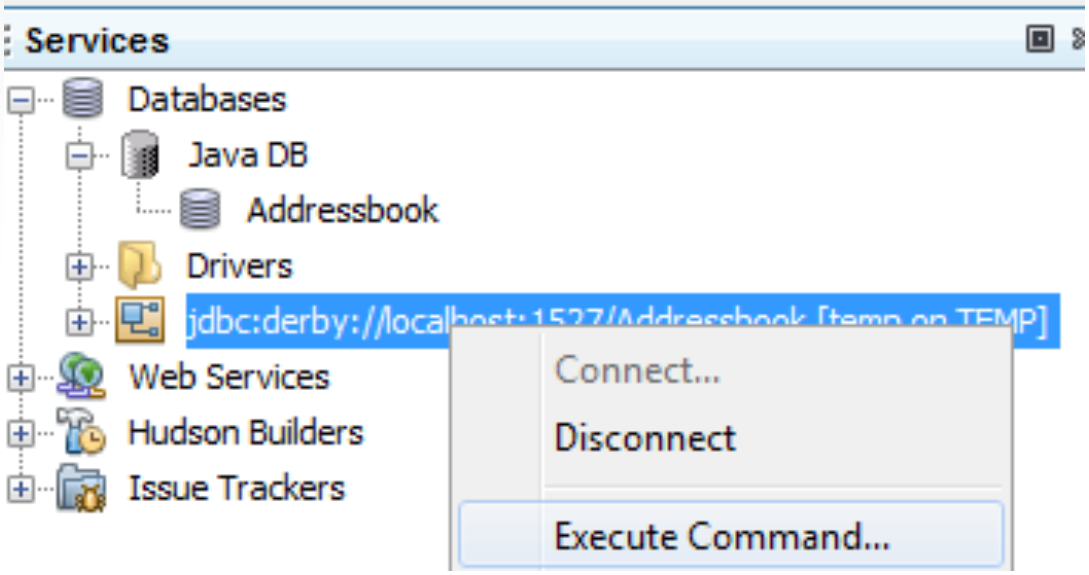
# Datenbankconnection ist angelegt






HTL LEONDING

# Tabellen per Script anlegen






HTL LEONDING

# Script zum Anlegen

- Eine Tabelle anlegen
- Id als Primärschlüssel mit Autoinkrement-Verhalten

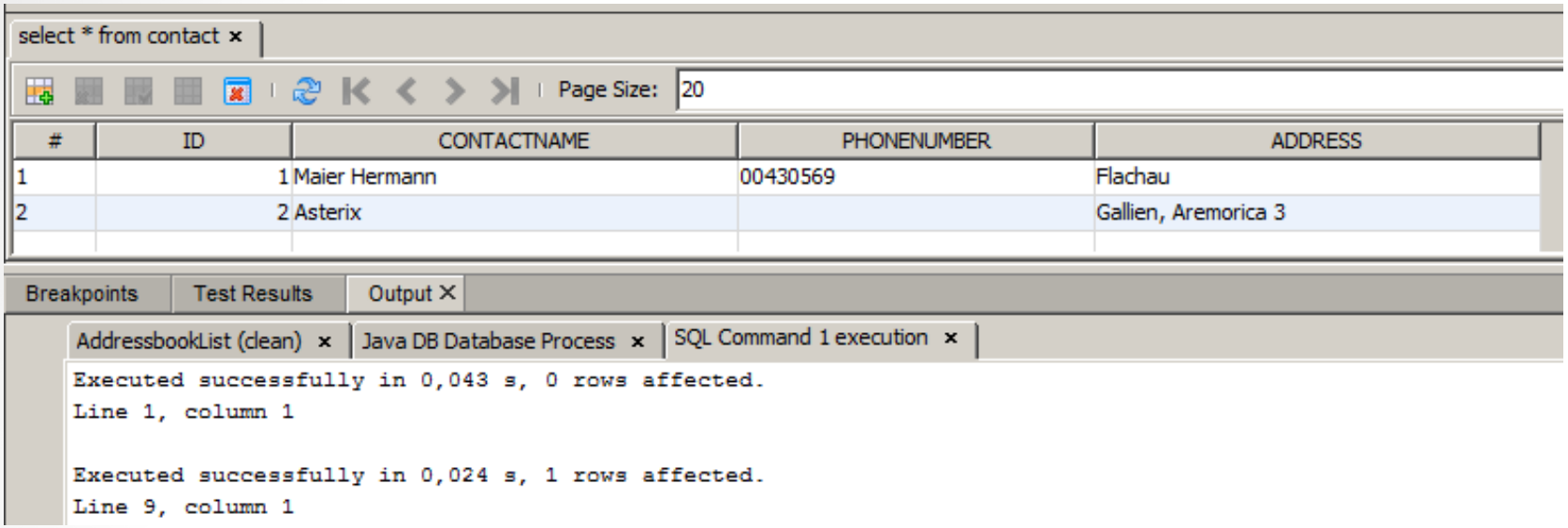
SQL Command 1 ×

Source History Connection: jdbc:derby://localhost:1527/AddressBook [AddressBook on ADDRESSBOOK]

```
1  create table contact (
2  id INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1),
3  contactname varchar(50),
4  phonenumber varchar(20),
5  address varchar(50),
6  PRIMARY KEY (id)
7  );
8
9  insert into contact (contactname, phonenumber, address) values('Maier Hermann', '00430569', 'Flachau');
10 insert into contact (contactname, phonenumber, address) values('Asterix', '', 'Gallien, Aremorica 3');
11 select * from contact;
```

# Positive Rückmeldung

- Tabellen sind angelegt
- Testdaten sind eingefügt

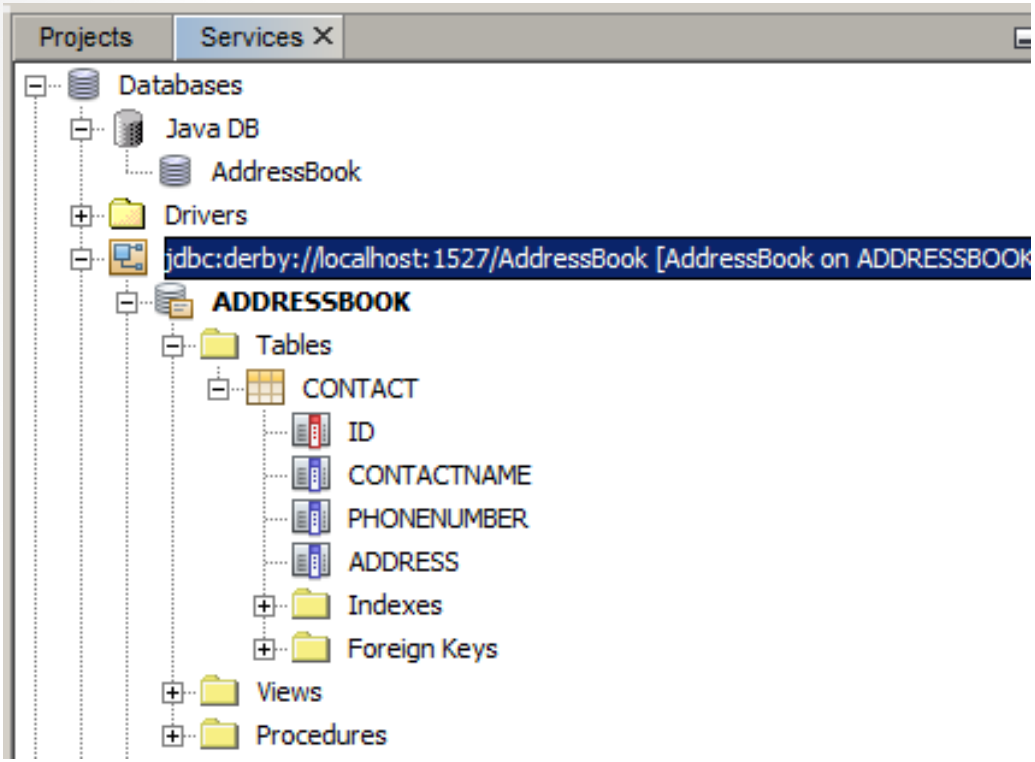





HTL LEONDING

# Ergebnis

- Schema für Benutzer (= Anwendungsname)
- Tabelle laut Script

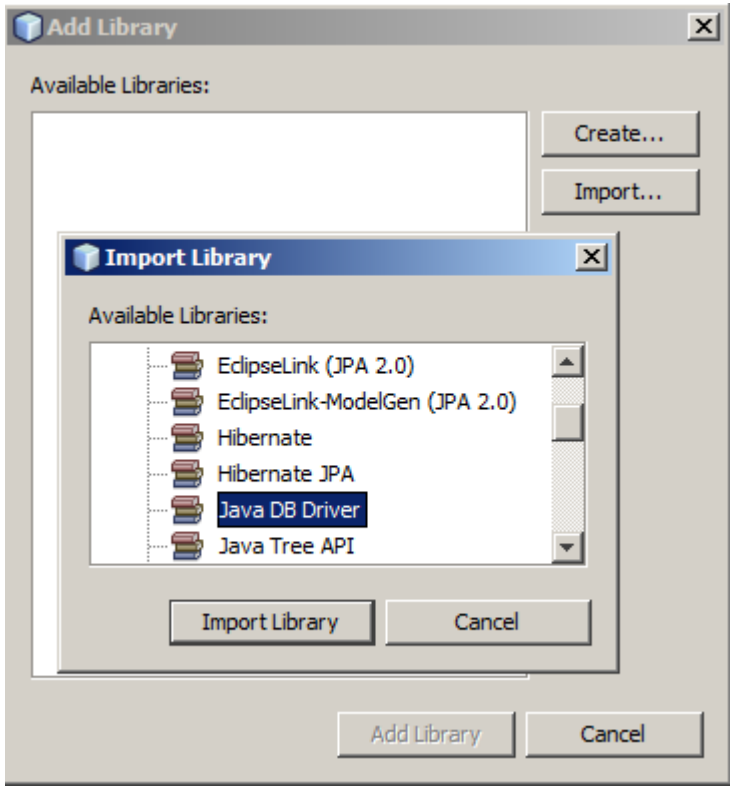





HTL LEONDING

# Treiber als Bibliothek einbinden

- Bibliothek einbinden
- Kontextmenü Folder Libraries
  - Add Library
  - Import
    - Java DB Driver
  - Add Library




HTL LEONDING

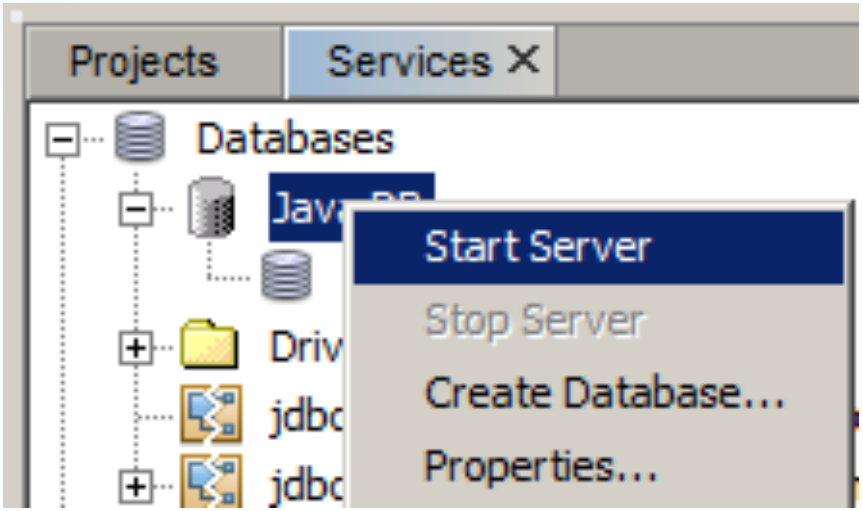
# Connection über Konstante definieren

- Wenn Verbindung nicht zustande kommt
  - Läuft der Server?
  - Stimmen die Zugangsdaten (Groß/Kleinschreibung)

```
public class Database {
    static final String DRIVER_STRING ="org.apache.derby.jdbc.ClientDriver";
    static final String CONNECTION_STRING = "jdbc:derby://localhost:1527/AddressBook";
    static final String USER = "AddressBook";
    static final String PASSWORD = "passme";
```

# Server starten/stoppen

- Services ➔ Databases






HTL LEONDING

# Datenbankzugriff als Singleton anlegen

```
public class Database {

    static final String DRIVER_STRING = "org.apache.derby.jdbc.ClientDriver";
    static final String CONNECTION_STRING = "jdbc:derby://localhost:1527/AddressBook";
    static final String USER = "AddressBook";
    static final String PASSWORD = "passme";
    static Database instance = null;
    Connection connection;

    /** * ...6 lines */
    public static synchronized Database getInstance() {
        if (instance == null) {
            instance = new Database();
        }
        return instance;
    }

    private Database() {
        connection = null;
        try {
            Class.forName(DRIVER_STRING);
            connection = DriverManager.getConnection(CONNECTION_STRING, USER, PASSWORD);
            connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
            connection.setAutoCommit(true);
        } catch (ClassNotFoundException ex) {
            System.out.println("Treiber laden nicht moeglich " + ex + "\n");
            System.exit(1);
        } catch (SQLException ex) {
            System.out.println("Verbindung zur Datenbank nicht moeglich " + ex + "\n");
            System.exit(1);
        }
```

# Connection

- Repräsentiert eine Zugriffssession auf die DB
- Im Kontext der Connection werden die SQL-Statements ausgeführt
- Connection ist teure Ressource ➔ möglichst mit einer auskommen (Singleton)
- Metadaten der DB sind über Connection abfragbar
- Natürlich sind auch Transaktionen möglich

# Kontakte gefiltert auslesen

- Mit PreparedStatement

```
public List<Contact> getContacts(String filter) throws SQLException {
    PreparedStatement preparedStatement = connection.prepareStatement(
            "select id, contactname, phonenumber, address from contact where contactname like ? order by
    preparedStatement.setString(1, filter);
    ResultSet rs = preparedStatement.executeQuery();
    List<Contact> contacts = new LinkedList<Contact>();
    while (rs.next()) {
        // Zugriff über columnlabel
        contacts.add(new Contact(rs.getInt("id"), rs.getString("contactname"),
          rs.getString("phonenumber"), rs.getString("address")));
        // Zugriff über columnindex BEGINNEND BEI 1!!!
        //contacts.add(new Contact(rs.getInt(1), rs.getString(2),rs.getString(3), rs.getString(4)));
    }
    rs.close();
    preparedStatement.close();
    return contacts;
}
```

# ResultSet-Methoden (Index/Name)

- String getString(int columnIndex)
- boolean getBoolean(int columnIndex)
- byte getByte(int columnIndex)
- short getShort(int columnIndex)
- int getInt(int columnIndex)
- long getLong(int columnIndex)
- float getFloat(int columnIndex)
- double getDouble(int columnIndex)
- Date getDate(int columnIndex)
- Time getTime(int columnIndex)
- Timestamp getTimestamp(int columnIndex)

# Mapping Java-Typen ⇔ SQL Typen

| SQL type | Java Type |
|---|---|
| CHAR, VARCHAR, LONGVARCHAR | String |
| NUMERIC, DECIMAL | java.math.BigDecimal |
| BIT | boolean |
| TINYINT | byte |
| SMALLINT | short |
| INTEGER | int |
| BIGINT | long |
| REAL | float |
| FLOAT, DOUBLE | double |
| BINARY, VARBINARY, LONGVARBINARY | byte[] |
| DATE | java.sql.Date |
| TIME | java.sql.Time |
| TIMESTAMP | java.sql.Timestamp |

# Update Contact

```
public void updateContact(Contact contact) throws SQLException {
    PreparedStatement updateContactPreparedStatement = connection.prepareStatement(
            "UPDATE contact SET contactname = ?, phonenumber = ?,  address = ? WHERE id = ?");
    updateContactPreparedStatement.setString(1, contact.getName());
    updateContactPreparedStatement.setString(2, contact.getPhoneNumber());
    updateContactPreparedStatement.setString(3, contact.getAddress());
    updateContactPreparedStatement.setInt(4, contact.getId());
    updateContactPreparedStatement.executeUpdate();
    updateContactPreparedStatement.close();
}
```

# Kontakt löschen

```
public void deleteContact(Contact contact) throws SQLException {
    PreparedStatement deleteContactPreparedStatement =
            connection.prepareStatement("delete from contact where id = ?");
    deleteContactPreparedStatement.setInt(1, contact.getId());
    deleteContactPreparedStatement.executeUpdate();
}
```