Programmierung von Systemen – 16 – SQL in Java

Matthias Tichy & Stefan Götz | SoSe 2020

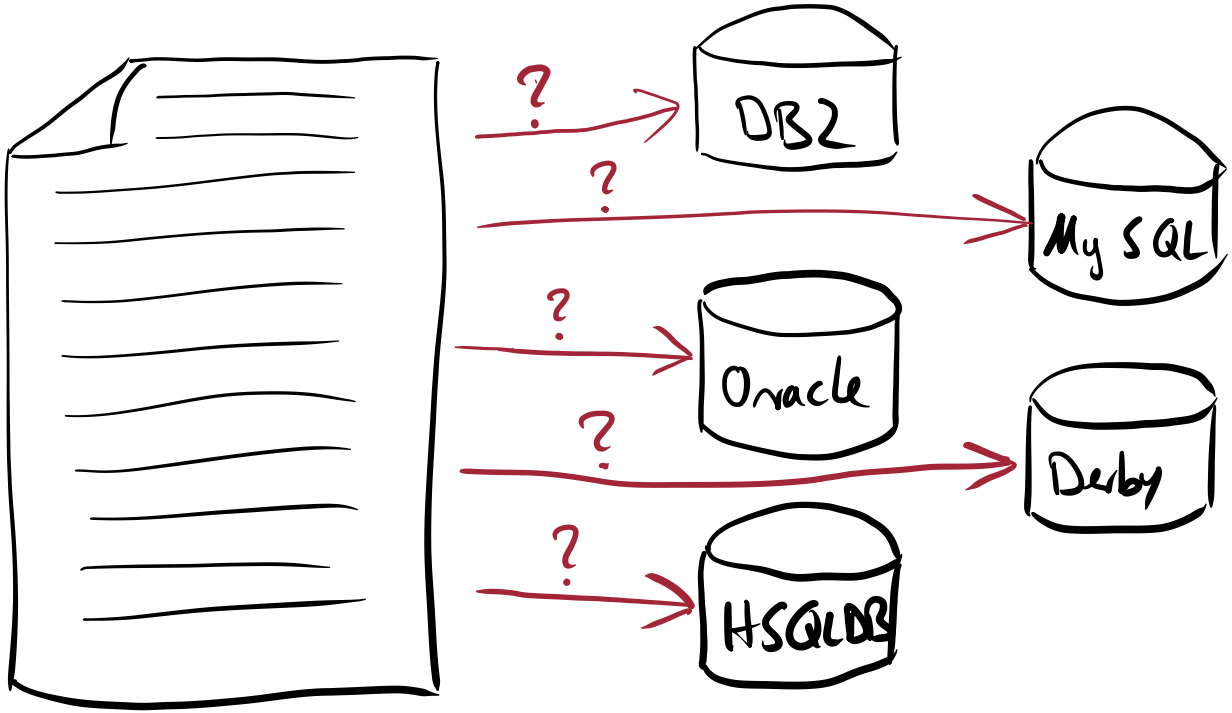
Ziele

- Daten aus Java Objekten in eine relationale Datenbank schreiben und wieder auslesen
- verschiedene Zugriffsmöglichkeiten auf DBMS von Java aus zumindest prinzipiell kennen
- Vor- und Nachteile der einzelnen Persistenz-Frameworks kennen und im Einsatz abwägen können
- Moderne Entwicklungen bzgl. Persistenz kennen
- JDBC anwenden können

Probleme?

- Mengen vs. Arrays
- Anzahl der Zeilen nicht bekannt
- Attribute in Spalten statt Objekte mit Feldern
- Anzahl der Spalten und evtl. deren Bezeichner nicht bekannt
  ➔ SELECT * FROM Mitarbeiter?
- Typen
- Wie kann ich das DBMS ansprechen?
- Wo ist überhaupt die Datenbank?
- Welche SQL-Befehle unterstützt das DBMS?

Ich nix verstehn



Ich nix verstehn

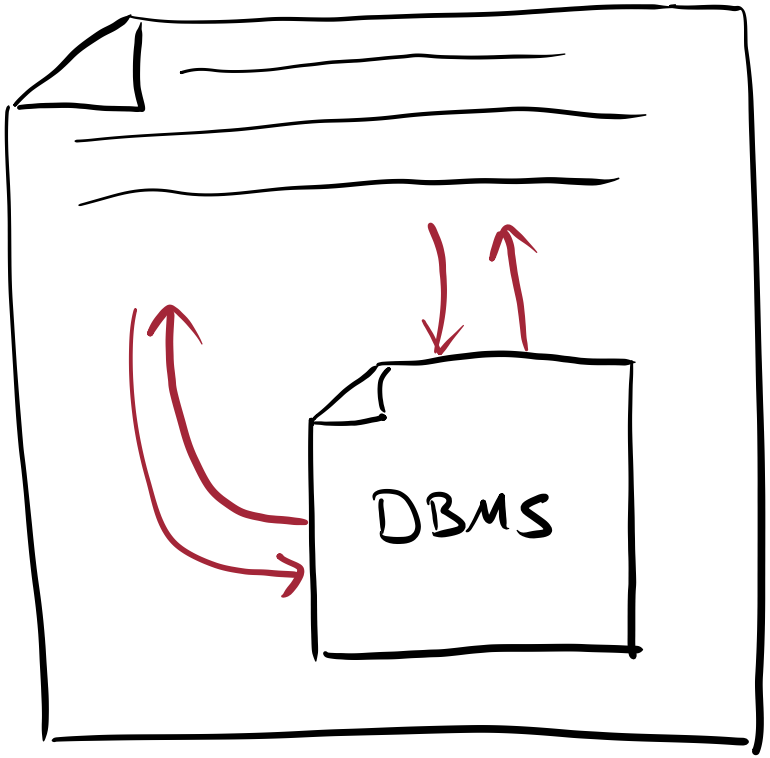
- SQL-Dialekt von verwendetem DBMS abhängig
- Wunsch: Anwendungsprogramm möglichst unabhängig von verschiedenen Dialekten
- Evtl. Abfragen der unterstützten Funktionen/Syntax und dynamische Anpassung der Anfragen

Hallo, DBMS!

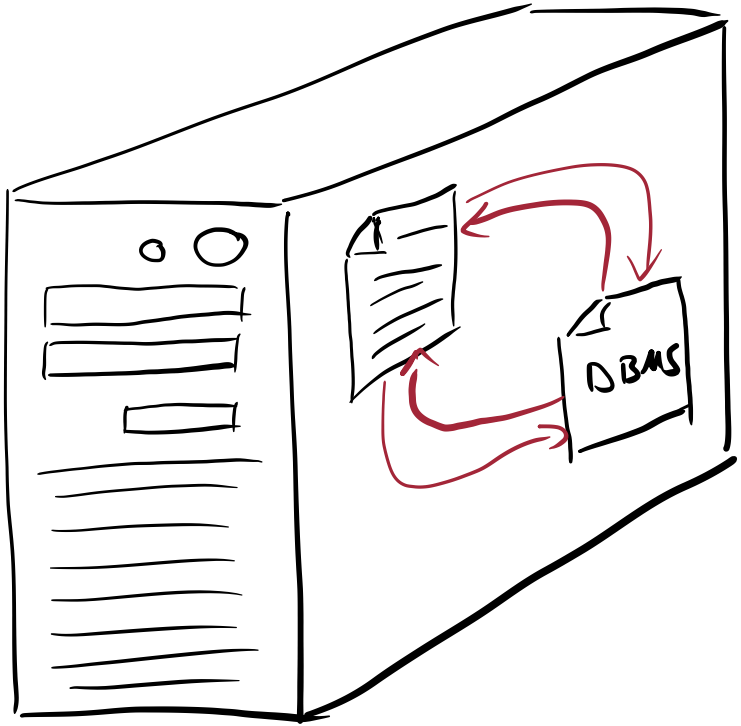
Kontakt abhängig von DBMS:

- als Bibliothek direkt im Programm
- als eigenständiges Programm auf dem gleichen Rechner
- als eigenständiges Programm auf einem anderen Rechner (Service)

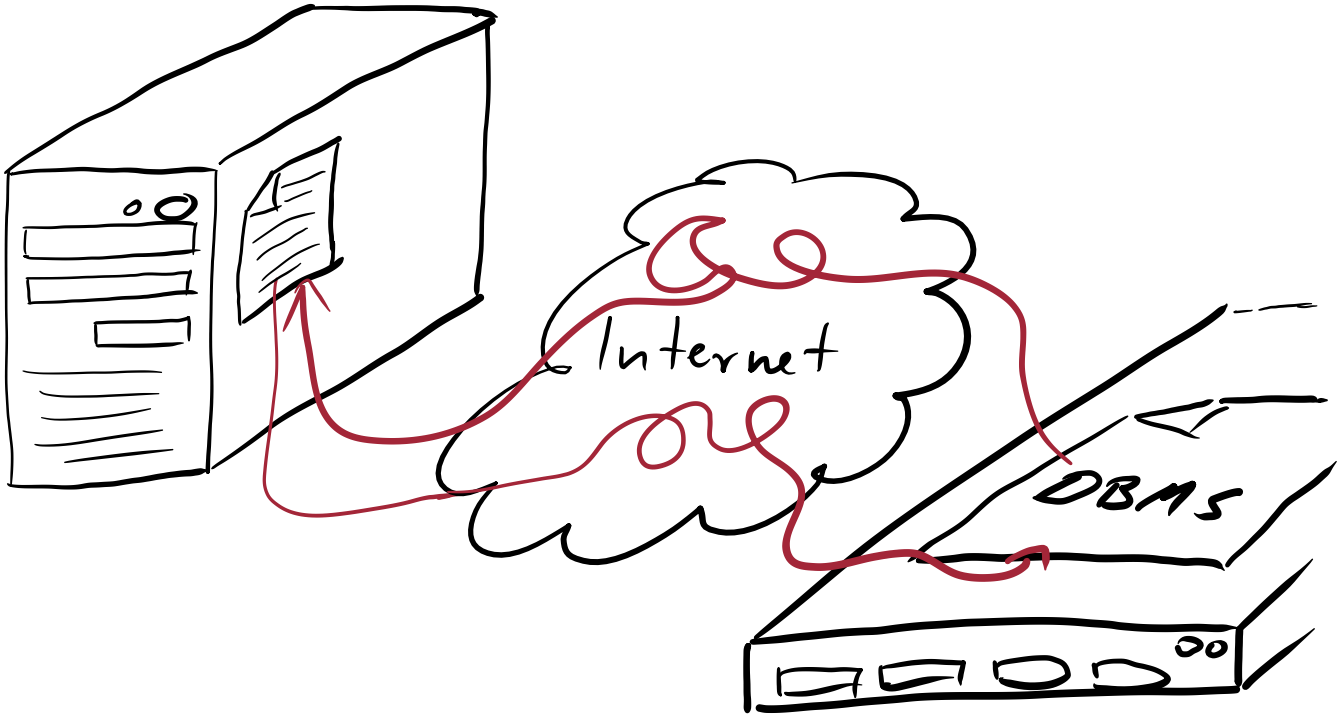
Hallo, DBMS!



Hallo, DBMS!



Hallo, DBMS!

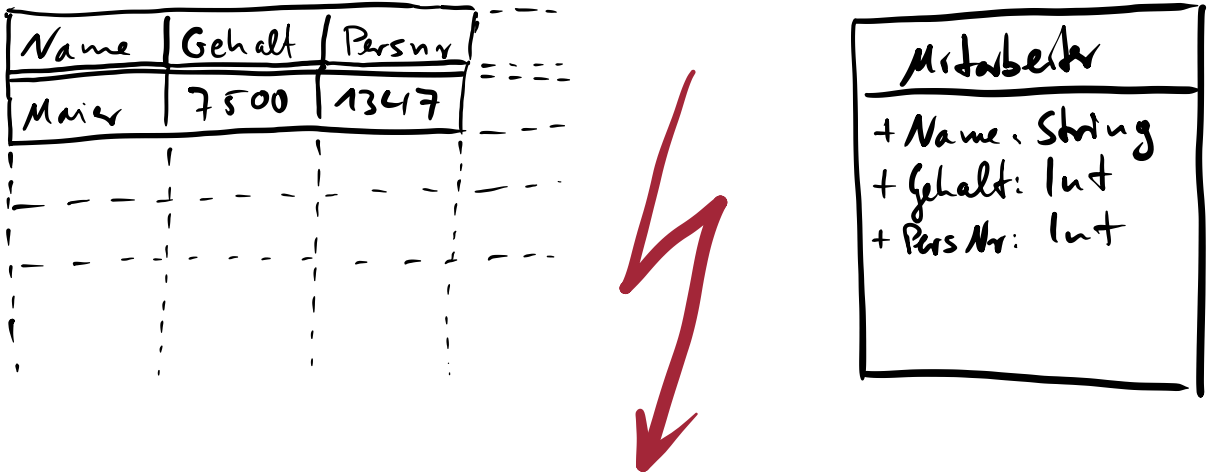


Was kriege ich?

- Abbildung der relationalen Mengen auf Arrays von Objekten nicht einfach
- "object-relational impedance mismatch"

- Kapselung der Ergebnismenge in einer Klasse (ResultSet)
  - flexible Zugriffsmöglichkeiten auf Zeilen und Spalten
  - "Metadaten" über Aufbau der Ergebnismenge

Was kriege ich?



Was sagt der SQL-Standard?

- SQL/Bindings
  - Embedded SQL: Statisches und dynamisches SQL
  - Benutzerdefinierte Datentypen und Routinen
- Call Level Interface (CLI)
  - Schnittstellendefinition zum Absetzen von SQL Statements über Methoden
- programmiersprachenneutrale Definition
- Bekannteste Implementierung von CLI: ODBC (Open Database Connectivity)

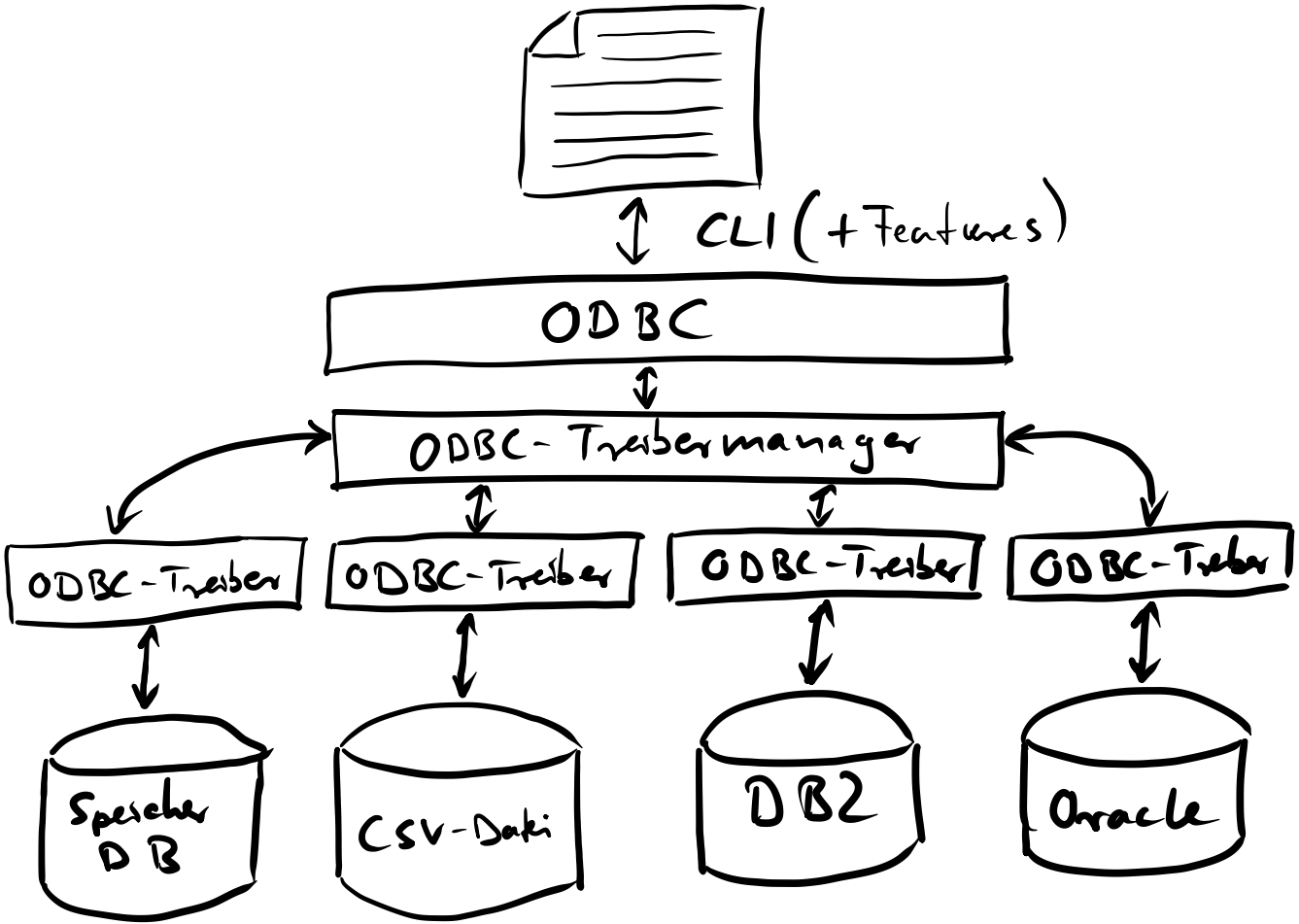
ODBC

- Version 1.0: 1992 von Microsoft
- aktuelle Version: 3.8 (seit 2009)

- API-Spezifikation und erste Implementierung des CLI

- Abstraktion von konkreter Datenhaltung/ Datenbankherstellern über ODBC-Treiber

- evtl. unterschiedlicher SQL-Funktionsumfang
  ➔ Metadaten abrufbar


CLI (+Features)
ODBC
ODBC-Treibermanager
ODBC-Treiber
ODBC-Treiber
ODBC-Treiber
ODBC-Treiber
Speicher DB
CSV-Daki
DB2
Oracle

JDBC

- Java Database Connectivity API
- einheitliche Datenbankschnittstelle
- Im JDK seit Version 1.1 (1997)

- analog zu ODBC: Datenbanktreiber von jedem DBMS-Anbieter

- Außerdem: JDBC-ODBC-Bridge
  ➔ jedes DBMS mit ODBC-Treiber nutzbar

Verwendung von JDBC

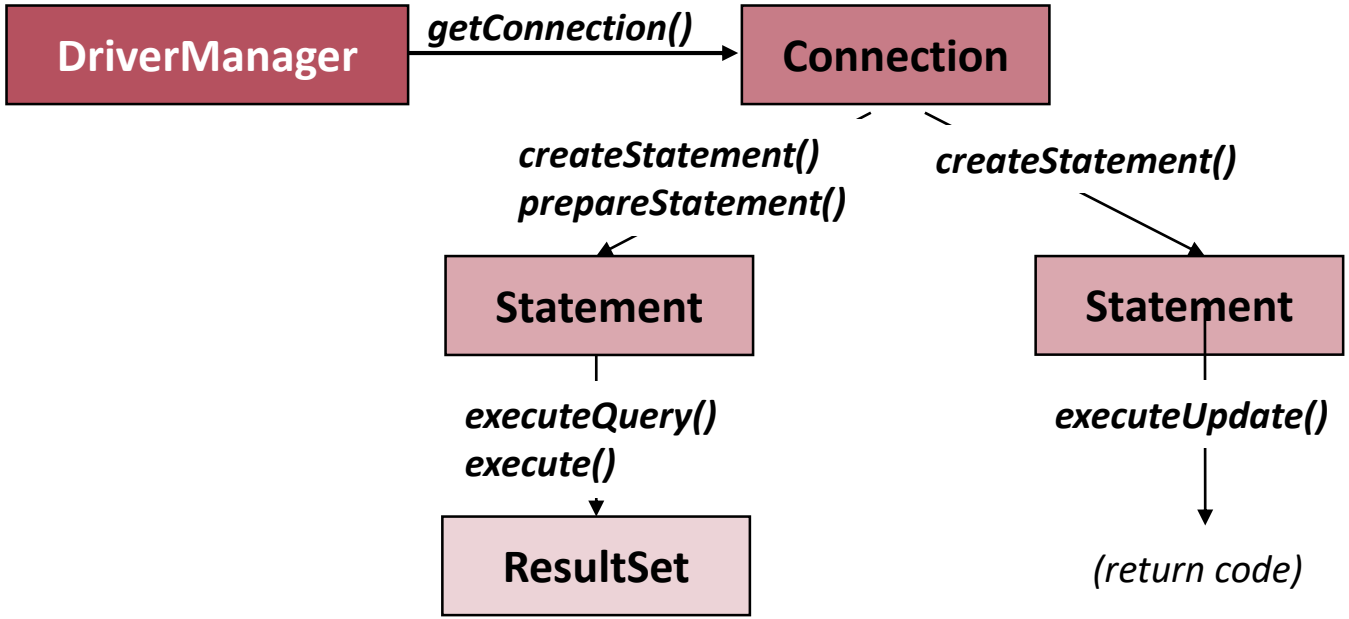
- **`java.sql.DriverManager`**
  Auswahl des passenden Treibers für die Datenbankanbindung

- **`java.sql.Connection`**
  Repräsentation der Verbindungen zur Datenbank

- **`java.sql.Statement`**
  Repräsentation einer SQL-Anweisung

- **`java.sql.ResultSet`**
  Repräsentation der Ergebnismenge

Verwendung von JDBC



Demo

Wichtige Aspekte:

- Check auf Treiberklasse
- Verbindungsaufbau mit Treiberauswahl über String

- Statement-Objekt von Connection-Objekt kreieren lassen

- **`executeQuery(String)`** für SFW-Abfragen mit ResultSet als Ergebnis
- **`executeUpdate(String)`** für UPDATE, INSERT, DELETE, CREATE, DROP-Befehle mit Anzahl der betroffenen Zeilen als Ergebnis

Demo

- Zugriff auf einzelne Zeilen über **`ResultSet.next()`**
- Zugriff auf einzelne Spalten über **`getString`**, **`getDouble, getInt, getDate`**… mit Spalten-nummer (beginnend bei 1) oder Attributname (langsamer, case insensitiv)

- Obwohl "ChefPersNr" ein Integer-Attribut ist, kann es als Stringwert ausgelesen werden.
  JDBC nimmt entsprechende Umwandlung vor.

- Prüfung auf Nullwert nach Lesevorgang des Attributs: **`wasNull()`**

Demo

- Sowohl das ResultSet als auch das Statement sollten mit **`close()`** geschlossen werden!
- Danach Objekte für weitere Anfragen bzw. Ergebnisse wiederverwendbar

- **`AutoCommit = true`**
  jedes SQL-Statement wird als einzelne Transaktion behandelt, d. h. nach jedem SQL-Statement wird ein Commit durchgeführt.
- **`AutoCommit = false`**
  ermöglicht es, mehrere Anweisungen zu „klammern“ und mit Commit oder Rollback abzuschließen.

SQL-Injection

- Einschleusen von eigenen SQL-Befehlen bzw. Verändern der existierenden SQL-Befehle
- Ursache: Fehlende Maskierung (Ausblendung) von Metazeichen
- Besonders bei Web-Anwendungen mit Parameterübergabe über URL

SQL-Injection – Beispiel

- ursprünglicher Befehl:
  SELECT * **FROM** Mitarbeiter
  WHERE name **LIKE** '%*nameToSearch*%'

- Mit *nameToSearch* =
  '; **UPDATE** Mitarbeiter **SET** Gehalt = 3500 **WHERE** PersNr = 2717; --

  wird daraus:

  SELECT * **FROM** mitarbeiter **WHERE** name **LIKE** '%';
  UPDATE Mitarbeiter **SET** Gehalt = 3500
  WHERE PersNr = 2717; --%'

SQL-Injection – Vermeidung

1. Parsen der Parameter und Entfernen aller unerwünschten Befehle
   ➔ bei über 300 Schlüsselwörtern sehr komplex

2. Statement komplett definieren und Parameter als echte Parameter übergeben
   ➔ PreparedStatement

PreparedStatement-Demo

- Definition über `prepareStatement(String)` der Connection-Klasse
- Parameter als alleinstehende Fragezeichen im String

- Parameter anschließend setzen mit (Index >= 1)
  - `PreparedStatement.setString(index, String)`
  - `PreparedStatement.setInt(index, int)`
  - …

- Ausführen mit `execute()` oder `executeQuery()`

- Performanzvorteile durch Wiederverwenden der Query und erneutes Setzen der Parameter

ResultSet - Besonderheiten

- Bisher:
  - jede Spalte in jeder Zeile nur genau einmal abfragen
  - kein "Rückwärtsgehen" im ResultSet möglich
  - Daten können nicht direkt geändert werden

- ResultSet erlaubt auch flexiblere Modi bzgl.
  - Cursorbewegung
  - Aktualisierung des ResultSets
  - Übernahme von Änderungen des ResultSets in DB

ResultSet - Besonderheiten

- Cursorbewegung und Aktualisierung:
  - **`TYPE_FORWARD_ONLY`** (default)
  - **`TYPE_SCROLL_INSENSITIVE`**:
    freier Zugriff, bei Änderung der Werte in der Datenbank, keine Aktualisierung des ResultSets
  - **`TYPE_SCROLL_SENSITIVE`**:
    freier Zugriff, ResultSet enthält stets aktuellste Werte

- Updatefunktionalität
  - **`CONCUR_READ_ONLY`** (default)
  - **`CONCUR_UPDATABLE`**:
    Änderungen im ResultSet werden in die DB geschrieben

ResultSet - Besonderheiten

- Angabe des Typs als Parameter bei `createStatement/prepareStatement`

- Befehle, um Cursor zu bewegen:
  - `next`
  - `previous`
  - `first`
  - `last`
  - `beforeFirst`
  - `afterLast`
  - `relative(int rows)`
  - `absolute(int row)`

ResultSet - Besonderheiten

- Befehle, um Werte zu verändern:
  - `updateInt(column, int)`
  - `updateString(column, String)`
  - …
- Immer abschließen mit **`updateRow()`**

ResultSet - Metadaten

- Metadaten (`ResultSet.getMetaData()`)
  - Anzahl der Spalten
  - Typen der Spalten
  - Namen der Spalten
  - maximale Breite der Spalten
  - …
- Damit auch dynamische Ausgabe möglich

Metadaten auch für DBMS

- Metadaten (`Connection.getMetaData()`)
  - DB-Tabellenbeschreibungen
  - Herstellername
  - Versionsnummer
  - Liste der unterstützten Funktionen
  - …

- Evtl. einige der ResultSet-Modi nicht von allen DBMS unterstützt

Vorteile/Nachteile JDBC

- einfach zu erlernen
- einfach anzuwenden
- weit verbreitet ((fast) alle Programmier-sprachen, alle DBMS)
- volle Mächtigkeit von SQL nutzbar

- keine Syntaxprüfung während Compilezeit
- hoher Testaufwand
- komplexe Behandlung der Ergebnisse
- Performanzprobleme durch Parsing der SQL-Befehle
- mehrere Schritte unbedingt einzuhalten (inkl. close())

jOOQ

- Externe Library → Wrapper für JDBC
- Parametrisierung über XML-Datei
- Codegenerierung auf Basis der XML-Datei

- Zusammenbau der SQL-Statements über verkettete Methodenaufrufe:

```
select().from(MITARBEITER).
    where(MITARBEITER.NAME.eq("Pauler")).
    fetch();
```

Vorteile/Nachteile jOOQ

- Typüberprüfung zur Compilezeit
- Abstraktion von SQL-Dialekt

- kein Zugriff auf volle SQL-Mächtigkeit
- eigene "Sprache" muss erlernt werden

Embedded SQL

- Warum nicht gleich SQL-Befehle in Quellcode? ➔ Precompiler
- Definiert in SQL/Bindings-Standard
- Für Java: SQLJ
- ABER:
  3. Treffer bei Google (nach 2x Wikipedia):
     "Why Did SQLJ Die?"

ORM – Hibernate

- Object-Relational-Mapping (ORM) versucht Objekte auf relationale DBMS abzubilden
- (ein) bekanntes Framework: Hibernate
- XML-Konfigurationsdatei für Code-/Daten-bankgenerierung oder Annotationen
- Zugriff nur auf Objekte, DB-Anbindung völlig transparent
  (➔ Nachteile bei komplexen Anfragen…)

JPA

- Java Persistence API (JPA) ist Schnittstellen-spezifikation von Java zur einheitlichen Verwendung von OR-Mappern

- Aktuelle Version 2.1 seit 2013

- Viele Ideen aus Hibernate flossen in JPA ein
- Hibernate wiederum implementiert JPA 2.1

- Typischerweise viel Konfigurationsaufwand für ORM

Ziele

- Daten aus Java Objekten in eine relationale Datenbank schreiben und wieder auslesen
- verschiedene Zugriffsmöglichkeiten auf DBMS von Java aus zumindest prinzipiell kennen
- Vor- und Nachteile der einzelnen Persistenz-Frameworks kennen und im Einsatz abwägen können
- Moderne Entwicklungen bzgl. Persistenz kennen
- JDBC anwenden können