Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Folien basierend auf Thorsten Papenbrock

- Übung Datenbanksysteme I

Integration von SQL und Programmiersprache

Leon Bornemann
HS 1
Hasso Plattner Institut

Übersicht

Integration von SQL und Programmiersprache

1) Motivation/Impedance Mismatch
2) Stored Procedures
3) DBMS Zugriff aus Programmiersprachen
   - Embedded SQL
   - JDBC
   - Object Relational Mapping (Hibernate)

Übersicht

Integration von SQL und Programmiersprache

1) Motivation/Impedance Mismatch
2) Stored Procedures
3) DBMS Zugriff aus Programmiersprachen
   - Embedded SQL
   - JDBC
   - Object Relational Mapping (Hibernate)

Grenzen von SQL

1. **Bedingte Anweisungen**
   - Erhöhe das Gehalt eines *Arbeitsnehmers* um 2%, falls er eine Belobigung erhalten hat
2. **Darstellung von Daten (z.B. Web-Anwendungen)**
   - Erstelle eine übersichtliche und schöne Repräsentation für eine Menge von *Produkt*-Tupeln
3. **Komplizierte Fragestellungen (z.B. Duplikaterkennung)**
   - Finde alle *Kunden*-Tupel, die sich sehr ähnlich sind
4. **String-Operationen (z.B. String-Splitting)**
   - Teile das Attribut *Name* auf in die Attribute *Vor-*, *Mittel-* und *Nachname* und weise den neuen Attributen anschließend passende Typen zu

Impedance Mismatch

- **Bisher: Generische SQL Schnittstelle**
  - Absetzen einzelner, unverknüpfter SQL-Statements
  - Verwendung über Kommandozeile oder DBMS-spezifischer GUIs
  - Selten genutzt (Datenbank-Administratoren)
  - Allerdings: SQL ist prinzipiell Turing-vollständig!

- **Jetzt: SQL in Programmiersprachen**
  - Einbettung in (große) Softwarekomponenten
  - Verwendung aus dem Quellcode heraus
  - Ausgiebig genutzt in allen möglichen Software-Projekten

    ➢ Impedance Mismatch

Impedance Mismatch

Begriff aus der Elektotechnik

- **Impedance Mismatch**
  = Verwendung unterschiedlicher Datenmodelle
  - *Relationales Model (DBMS)*
    - Primitive: Relationen und Attribute
    - Kontrolle: Nebenbedingungen
    - Modell: Deklarativ
  - *Generisches Modell (Programmiersprachen)*
    - Primitive: Pointer, verschachtelte Strukturen und Objekte
    - Kontrolle: Schleifen und Verzweigungen
    - Model: Imperativ (meistens)

➢ Datentransfer zwischen den Modellen notwendig!

Impedance Mismatch

- **Idee 1: Nutze SQL allein, allerdings …**
  - nicht alle Anweisungen ausdrückbar (z.B. nicht “n!“)
  - Ausgabe beschränkt auf Relationen (z.B. keine Grafiken)
- **Idee 2: Nutze Programmiersprache allein, allerdings …**
  - Anweisungen sind meist viel komplexer als SQL-Anfragen
  - Abstraktion von Speicherstrukturen nicht möglich
    - Verlust der physischen Datenunabhängigkeit!
  - DBMS sind extrem effizient
- **Idee 3: Kombiniere SQL und Programmiersprachen**
  - Programmiersprache (“Host Language“) für komplexe Operationen
  - Embedded SQL für effizienten Datenzugriff
  - Explizites Mapping der gelesenen und geschriebenen Daten

Kombination von SQL und Programmiersprachen

Zwei grundverschiedene Anwendungsfälle

Stored Procedures

- Erweiterung der DBMS Funktionalität
  - z.B. neue Aggregationsfunktionen, Operationen auf Strings, etc…
  - → User Defined Functions (UDF) können dann in SQL-Anfragen verwendet werden
- Zugriff auf DBMS innerhalb von Programmiersprachen
  - Laden von Daten aus der Datenbank
  - Speichern von Objekten (oder Teilen von Ihnen) in der Datenbank

Diverse Ansätze

Übersicht

Integration von SQL und Programmiersprache

1) Motivation/Impedance Mismatch
2) Stored Procedures
3) DBMS Zugriff aus Programmiersprachen
   - Embedded SQL
   - DBMS Funktionsbibliotheken (JDBC)
   - Object Relational Mapping (Hibernate)

Konzept und Umsetzung

- **Persistant Stored Modules (PSM)**
  - „Gespeicherte Prozeduren“, engl. *Stored Procedures*
  - Speichern Prozeduren als Datenbankelemente
  - Mischen SQL und Programmiersprache
  - Können in regulären SQL Ausdrücken verwendet werden

Bringt Programmiersprache zu SQL

```
CREATE PROCEDURE <Name>(<Parameter in/out>)
    <Lokale Variablendeklarationen>
    <Body der Prozedur>;
```

```
CREATE FUNCTION <Name>(<Parameter in>) RETURNS <Typ>
    <Lokale Variablendeklarationen>
    <Body der Prozedur>;
```

```
CREATE PROCEDURE MeanVar(   IN studioName CHAR[15],
                        OUT mittelwert REAL,
                        OUT varianz REAL)

    DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
    DECLARE FilmCursor CURSOR FOR
        SELECT Laenge FROM Filme WHERE StudioName = studioName;
    DECLARE neueLaenge INTEGER;
    DECLARE filmAnzahl INTEGER;

    BEGIN
        SET mittelwert = 0.0;
        SET varianz = 0.0;
        SET filmAnzahl = 0;
        OPEN FilmCursor;
        FilmLoop: LOOP
            FETCH FilmCursor INTO neueLaenge;
            IF Not_Found THEN LEAVE FilmLoop END IF;
            SET filmAnzahl = filmAnzahl + 1;
            SET mittelwert = mittelwert + neueLaenge ;
            SET varianz = varianz + neueLaenge * neueLaenge;
        END LOOP;
        CLOSE FilmCursor;
        SET mittelwert = mittelwert / filmAnzahl;
        SET varianz = varianz / filmAnzahl - mittelwert * mittelwert;
    END;
```

Parameter in/out

Lokale Variablen-deklarationen

Body der Prozedur

Erstelle eine Funktion, die den Durchschnitt und die Varianz der Filmlänge von Filmen eines Studios ermittelt

Unterstützte Programmiersprachen sind abhängig vom DBMS

Externe Definition

- Stored Procedures können externen Code verwenden
- Code muss dazu in bestimmtem Verzeichnis liegen
- Beispiel:
  - Datenbank: DB2
  - Externe Sprache: Java

```
CREATE PROCEDURE PARTS_ON_HAND(
    IN PARTNUM INTEGER,
        OUT COST DECIMAL(7,2),
        OUT QUANTITY INTEGER)
EXTERNAL NAME 'parts.onhand'
LANGUAGE JAVA
PARAMETER STYLE JAVA;
```

Übersicht

Integration von SQL und Programmiersprache

1) Motivation/Impedance Mismatch
2) Stored Procedures
3) DBMS Zugriff aus Programmiersprachen
   - Embedded SQL
   - DBMS Funktionsbibliotheken (JDBC)
   - Object Relational Mapping (Hibernate)

SQL und andere Programmiersprachen

1. **Embedded SQL**
   - Integriert SQL in andere Programmiersprachen:
     - ADA, C, C++, Cobol, Fortran, M, Pascal, PL/I, …
   - Bettet SQL beim Preprocessing in den Programmfluss ein
2. **DBMS-Funktionsbibliotheken**
   - Verbindet DBMS mit Programmiersprachen
   - Benötigt kein Preprocessing
   - Call-Level-Interface (CLI) für C
   - Java Database Connectivity (JDBC) für Java
3. **Object-Relational Mappings**
   - Speichert Repräsentationen der Objekte in Datenbank
   - Explizites Mapping der Objektfelder auf Attribute

Mehr oder weniger veraltet

Übersicht

Integration von SQL und Programmiersprache

1) Motivation/Impedance Mismatch
2) Stored Procedures
3) DBMS Zugriff aus Programmiersprachen
   - Embedded SQL
   - DBMS Funktionsbibliotheken (JDBC)
   - Object Relational Mapping (Hibernate)

Übersetzung

Programmiersprache & Embedded SQL

↓

Präprozessor

↓

Programmiersprache & Funktionsaufrufe

↓

Compiler

↓

Ausführbares Programm

Bringt SQL in die Programmiersprache

DBMS-spezifisch, d.h. Kompiliervorgang muss für jedes DBMS (und möglicherweise auch für verschiedene Versionen desselben DBMS) erneut ausgeführt werden!

Anfragen ohne Ergebnis

Füge ein Tupel in die Datenbank ein, welches mit Variablen aus dem Programm befüllt wird.

1. Host-Variablen deklarieren

```
EXEC SQL BEGIN DECLARE SECTION;
    char studioName[50], studioAdr[256];
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

2. Anfrage ohne Ergebnis

```
EXEC SQL INSERT INTO Studio(Name, Adresse)
         VALUES (:studioName, :studioAdr);
```

Hostvariablen mit Doppelpunkt

- Verfahren für jeden SQL-Ausdruck, der kein Ergebnis liefert:
  INSERT, DELETE, UPDATE, CREATE, DROP, …

Anfragen mit einem Ergebnis-Tupel

Lese die Werte eines Tupels in Variablen des Programms ein

1. Host-Variablen deklarieren

```
EXEC SQL BEGIN DECLARE SECTION;
     char studioName[50], studioAdr[256];
     char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

2. Anfrage mit einem Ergebnis-Tupel

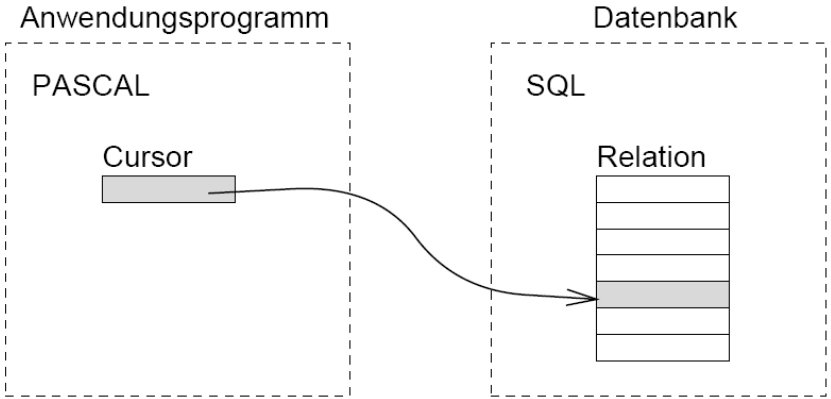
```
EXEC SQL SELECT Name, Adresse
         INTO :studioName, :studioAdr
         FROM Studio
         WHERE id = 310;
```

Anfragen mit vielen Ergebnis-Tupeln

1. Host-Variablen deklarieren

```
EXEC SQL BEGIN DECLARE SECTION;
     char studioName[50], studioAdr[256];
     char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

2. Anfrage mit vielen Ergebnis-Tupeln



Anfragen mit vielen Ergebnis-Tupeln

```
void printGehaltsBereiche() {

        Stellen = 0: Anzahl Manager = 2
        Stellen = 1: Anzahl Manager = 0
        Stellen = 2: Anzahl Manager = 2
        Stellen = 3: Anzahl Manager = 12
        Stellen = 4: Anzahl Manager = 39
        Stellen = 5: Anzahl Manager = 43
        Stellen = 6: Anzahl Manager = 31
        Stellen = 7: Anzahl Manager = 25
        Stellen = 8: Anzahl Manager = 6
        Stellen = 9: Anzahl Manager = 2
        Stellen = 10: Anzahl Manager = 0
        Stellen = 11: Anzahl Manager = 0
        Stellen = 12: Anzahl Manager = 2
        Stellen = 13: Anzahl Manager = 1
        Stellen = 14: Anzahl Manager = 1


}
```

Gib für jede Dezimalstellenanzahl von 0 bis 14 die Anzahl an Managern mit einem Gehalt dieser Stellen aus

Anfragen mit vielen Ergebnis-Tupeln

```
void printGehaltsBereiche() {
   int i, stellen, counts[15];
   EXEC SQL BEGIN DECLARE SECTION;
        int gehalt; char SQLSTATE[6];
   EXEC SQL END DECLARE SECTION;

   EXEC SQL DECLARE managerCursor CURSOR FOR
        SELECT Gehalt FROM Manager;
   EXEC SQL OPEN managerCursor;

   for (i = 0; i < 15; i++) counts[i] = 0;

   while (1) {
        EXEC SQL FETCH FROM managerCursor INTO :gehalt;
        if (strcmp(SQLSTATE,"02000")) break;
        stellen = 1;
        while ((gehalt /= 10) > 0) stellen++;
        if (stellen < 15) counts[stellen]++;
   }

   EXEC SQL CLOSE managerCursor;

   for (i = 0; i < 15; i++)
        printf("Stellen = %d: Anzahl Manager = %d\n", i, counts[i]);
}
```

Wird als Nebeneffekt befüllt

Cursor deklarieren und öffnen

Tupel abrufen und Cursor weitersetzen

≡ NO DATA

Cursor schließen

Übersicht

Integration von SQL und Programmiersprache

1) Motivation/Impedance Mismatch
2) Stored Procedures
3) DBMS Zugriff aus Programmiersprachen
   - Embedded SQL
   - DBMS Funktionsbibliotheken (JDBC)
   - Object Relational Mapping (Hibernate)

Übersetzung

Programmiersprache &
Embedded SQL

Präprozessor

Programmiersprache &
Funktionsaufrufe

Plus SQL, zumeist
Als String-Parameter

Compiler

SQL-Bibliothek
(Treiber) vom
DBMS-Hersteller

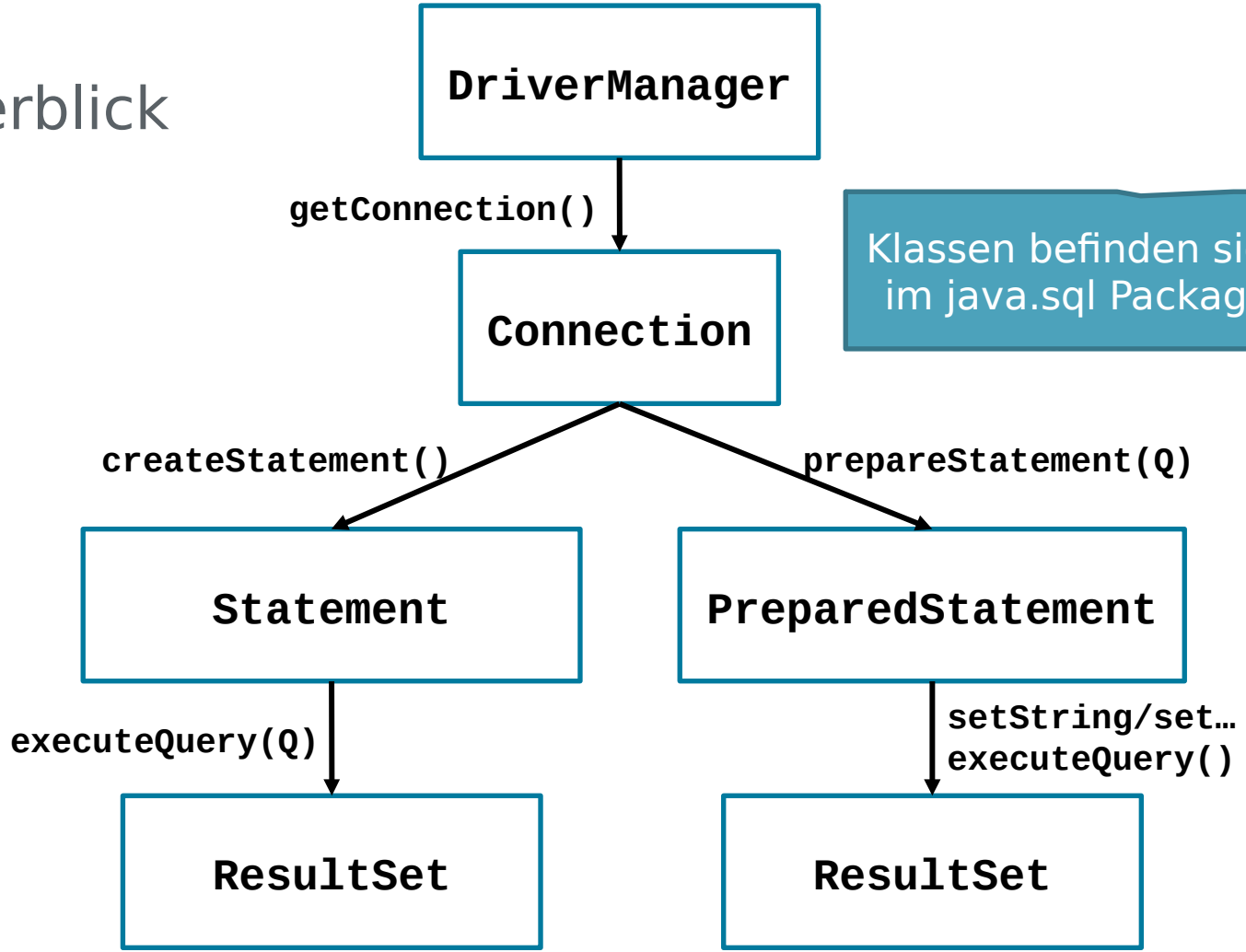
Ausführbares Programm

Konzept

- **DBMS-Spezifische Funktionsbibliotheken**
  - Ermöglichen …
    - Programmieren in einer Programmiersprache (Wirtssprache)
    - Einbettung von Datenbankanfragen in SQL
  - Bieten spezielle Funktionen für den Datenbankzugriff
  - Umgehen den Präprozessor
    - Kompiliertes Ergebnis ist aber gleich!

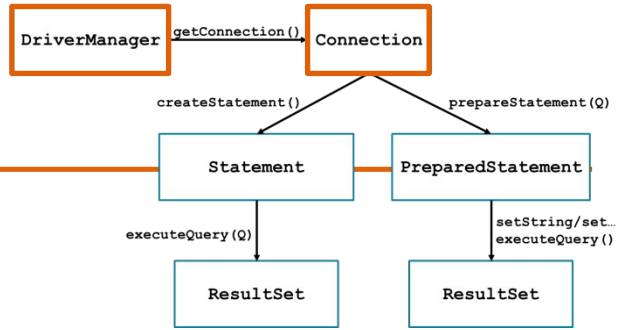
- **Call-Level-Interface (CLI)**
  - Verbindet C mit DBMS
  - Adaptiert von ODBC (Open Database Connectivity)
- **Java Database Connectivity (JDBC)**
  - Verbindet Java mit DBMS
  - Nutzt Objektorientierung im Gegensatz zu CLI

JDBC
Überblick



Klassen befinden sich im java.sql Package

Erste Schritte



1. **DBMS-spezifischen Treiber einbinden**
   - JDBC-Bibliothek in den Classpath einbinden
2. **Driver explizit laden**
   - Class.forName(„org.postgresql.Driver“);

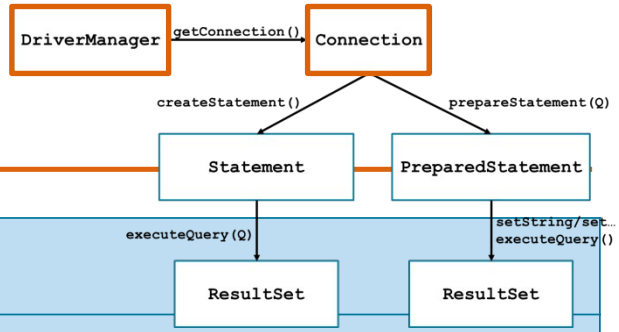
For Pre JDBC 4.0 Drivers

3. **Verbindung zur Datenbank aufbauen**

```
String URL = "jdbc:postgresql://<server>:<port>/<db_name>";
String name = "<username>";
String pw = "<password>";
Connection con = DriverManager.getConnection(URL, name, pw);
```

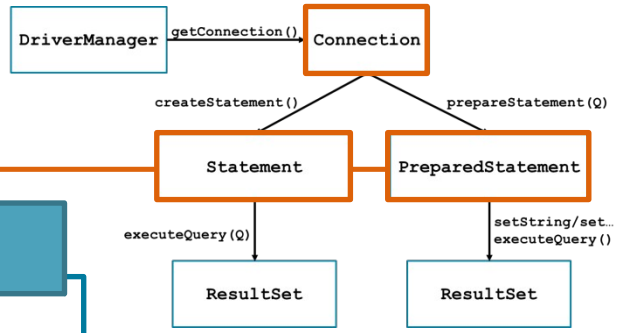
- URL ist DBMS- und Datenbank-spezifisch
- URL-Pattern: "jdbc:subprotocol:datasource"

JDBC
DBMS-URLs



Vendor	DBMS-URL
DB2	jdbc:db2://{host}[:{port}]/{dbname}
Derby	jdbc:derby://server[:port]/databaseName[;URLAttributes=value[;...]]
HSQLDB	jdbc:hsqldb[:{dbname}][:hsql://{host}[/{port}]]
MS SQL-Server	jdbc:microsoft:sqlserver://{host}[:{port}][;DatabaseName={dbname}]
MySQL	jdbc:mysql://{host}[:{port}]/[{dbname}]
Oracle	jdbc:oracle:thin:@{host}:{port}:{dbname}
PostgreSQL	jdbc:postgresql://[{host}[:{port}]]/{dbname}

Statements



Statement

Noch ohne SQL-Anfrage

```
Statement stmt = con.createStatement();
```

Prepared Statement

Für häufige SQL-Anfragen

```
PreparedStatement pstmt = con.prepareStatement(<Anfrage>);
```

SQL Ausdrücke ausführen

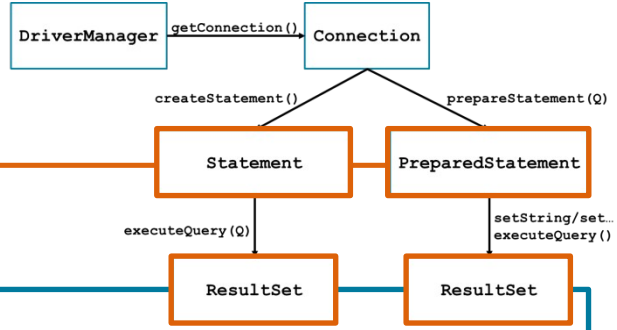
1. `ResultSet rs = stmt.executeQuery(<Anfrage>);`
2. `ResultSet rs = pstmt.executeQuery();`

ResultSet als Rückgabewert

3. `stmt.executeUpdate(UpdateAnfrage)`
4. `pstmt.executeUpdate()`

Ohne Rückgabewert

JDBC
Beispiele



Liste alle Manager-Gehälter:

```
1.  Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT Gehalt FROM Manager");
```

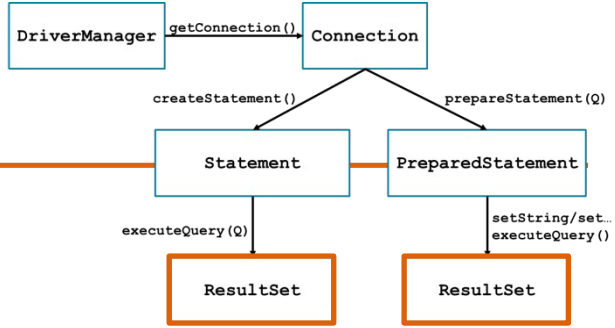
```
2.  PreparedStatement pstmt = con.prepareStatement(
                              "SELECT Gehalt FROM Manager");
    ResultSet rs = pstmt.executeQuery();
```

Füge neues Schauspieler-Tupel ein:

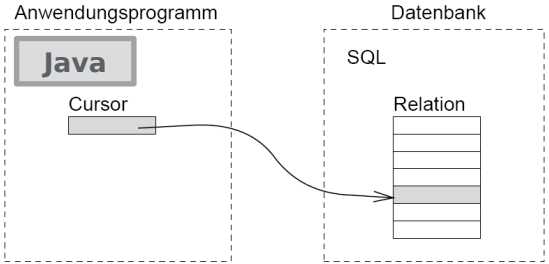
```
3.  Statement stmt = con.createStatement();
    stmt.executeUpdate("INSERT INTO spielt_in VALUES(" +
                       "'Star Wars', 1979, 'Harrison Ford'");
```

```
4.  PreparedStatement pstmt = con.prepareStatement(
                       "INSERT INTO spielt_in VALUES(" +
                       "'Star Wars', 1979, 'Harrison Ford'");
    pstmt.executeUpdate();
```

ResultSet: Cursor



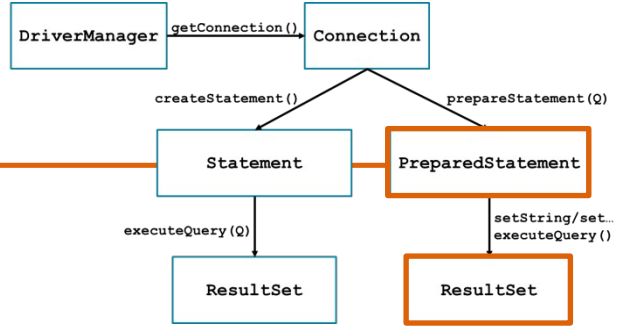
- **Methoden des ResultSet**
  - **`next()`**
    - liefert nächstes Tupel
    - liefert **`FALSE`**, falls kein weiteres Tupel vorhanden
  - **`getString(i)`**
    - Liefert Wert des i-ten Attributs des aktuellen Tupels
    - **`getInt(i)`**, **`getFloat(i)`** usw.
    - Alternativ: **`getString("<AttributName>")`**



- Anwendungsbeispiel:

```
while(gehaelter.next()){
        gehalt = gehaelter.getInt(1);
        // Operationen auf gehalt
}
```

JDBC
Paramater

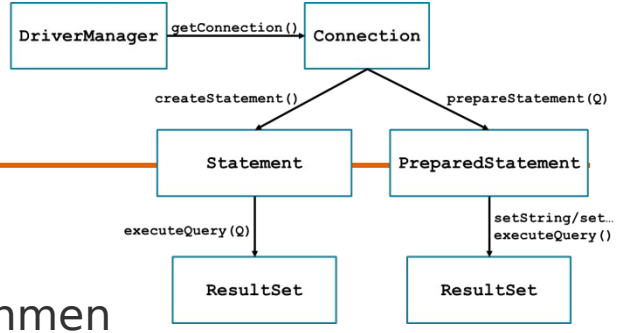


- Definition mittels **`PreparedStatement`**
- Fragezeichen als Platzhalter für Parameter
  - Bindung mittels **`setString(i, v)`**, **`setInt(i, v)`** usw.

- Beispiel: Einfügen eines neuen Studios

```
String studioName = "Pixar Animation Studios";
String studioAdr = "Emeryville, Vereinigte Staaten";
PreparedStatement studioStmt = con.prepareStatement(
    "INSERT INTO Studio(Name, Adresse) VALUES(?, ?)");

studioStmt.setString(1, studioName);
studioStmt.setString(2, studioAdr);
studioStmt.executeUpdate();
```

Zusammenfassung



1. JDBC-Bibliothek einbinden
   - DBMS-JAR in den Classpath aufnehmen
2. Verbindung zur Datenbank aufbauen
   - Ggf. Driver-Klasse explizit laden
   - **`Connection`** über **`DriverManager`** herstellen
3. SQL-Anfragen ausführen
   - **`Statements`** oder **`PreparedStatements`** nutzen
4. Ergebnis der SQL-Anfragen abfragen
   - **`ResultSet`** auswerten

Übersicht

Integration von SQL und Programmiersprache

1) Motivation/Impedance Mismatch
2) Stored Procedures
3) DBMS Zugriff aus Programmiersprachen
   - Embedded SQL
   - DBMS Funktionsbibliotheken (JDBC)
   - Object Relational Mapping (Hibernate)

Object Relational Mapping

- **Bisher: Zugriff auf Daten in Datenbank**
  - Spezielle, generische Klassen (ResultSet, Row, etc…)
  - Nachteile: nicht Objektorientiert, keine statischen Typ-Checks

- **Idee: Speichere Objekte direkt in der Datenbank**
  - 1 Objekt = 1 Tupel, 1 Klasse = 1 Tabelle
  - Mapping von Basis-Datentypen
  - Pointer als Fremdschlüssel
  - Löst Impedance Mismatch??

Beispiel für Java: Hibernate

```
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;
   private List certificates;
```

```
public class Certificate{
   private int id;
   private String name;
```

Normale Klassendefinitionen in java

Mapping zur Datenbank: XML-Datei

```
<hibernate-mapping>
   <class name = "Employee" table = "EMPLOYEE">

      <meta attribute = "class-description">
         This class contains the employee detail.
      </meta>

      <id name = "id" type = "int" column = "id">
         <generator class="native"/>
      </id>

      <list name = "certificates" cascade="all">
         <key column = "employee_id"/>
         <list-index column = "idx"/>
         <one-to-many class="Certificate"/>
      </list>

      <property name = "firstName" column = "first_name" type = "string"/>
      <property name = "lastName" column = "last_name" type = "string"/>
      <property name = "salary" column = "salary" type = "int"/>

   </class>
```

Mapping zu Datenbanktabelle

Referenz zu einer anderen Entität

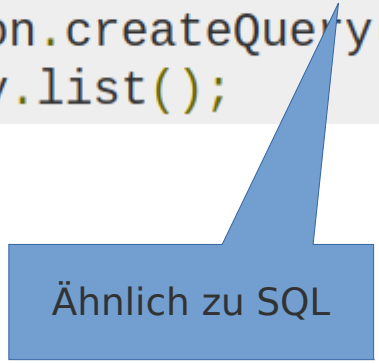
Attributmappings

2. Möglichkeit: Mapping durch Annotationen

```java
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
   @Id @GeneratedValue
   @Column(name = "id")
   private int id;

   @Column(name = "first_name")
   private String firstName;

   @Column(name = "last_name")
   private String lastName;

   @Column(name = "salary")
   private int salary;
```

Hibernate Query Language (HQL)

```
String hql = "SELECT E.firstName FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```



Object Relational Mapping

Datenbank wir aus Sicht des Programmieres konstruiert!

Im klassischen Use-case werden Datenbanken häufig von Nicht-Informatikern verwendet

- **Vorteile**
  - (Scheinbar!) einfache Kombination der beiden Datenmodelle/Paradigmen
  - Macht es einfach Objekte eines Programms zu persistieren

- **Nachteile**
  - Programmieren wird komplizierter (viele Dinge werden ineffizient)
  - Manche Datenstrukturen (z.B. Linked List) eignen sich schlecht für Object-Relational Mappings
  - Engere Kopplung der Datenbanktabellen an das Programm

Object Relational Mapping

- **Fazit**
  - Prinzipiell angenehmes Mapping zwischen Objekten/Klassen und Daten in der Datenbank
  - Konstruiert Datenbank aus der Sicht des Programmierers (Code first)
  - Versteckt viel Komplexität
  - Kann bei Benutzung von Anfängern sehr ineffizient sein (Klassisch: N+1 Problem)
  - Eignet sich nicht für alle Datenstrukturen → große Mengen an Pointern zu sich selbst sind problematisch
  - → Impedance Mismatch nur teilweise gelöst

Das ist nicht immer positiv!

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Übung Datenbanksysteme I

Embedded SQL, Stored Procedures, JDBC

Leon Bornemann
HS I
Hasso Plattner Institut