Praktikum

DBAE / WI

JDBC

Pascal Reuss. M.Sc.

Raum A08b Spl.
Email: reusspa@uni-hildesheim.de

IIS Intelligente Informationssysteme

Stiftung Universität Hildesheim 2003

JDBC

- JDBC: Java Database Connectivity

- Stellt Methoden zur Verfügung, mit denen eine Verbindung zu einer Datenbank hergestellt werden kann, sowie Daten eingefügt, ausgelesen und geändert werden können.

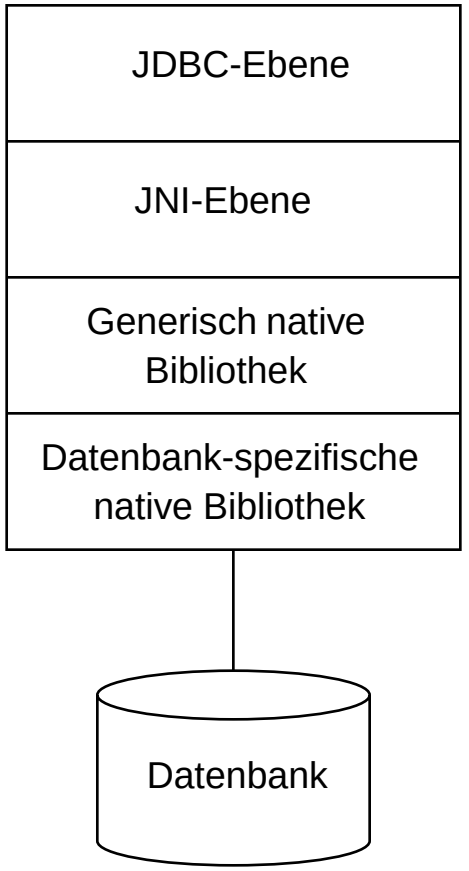
JDBC

- Klassen und Methoden, um auf relationale Datenbanken zuzugreifen.
  - Der Kern ist unter `java.sql.*` zu finden
  - Optionales Paket unter `javax.sql.*`

- Für die Kommunikation zwischen dem Java-Programm und der Datenbank wird ein Treiber benötigt:
  - Es gibt 4 Treiber-Typen.
  - Üblicherweise wird Typ 4 verwendet.

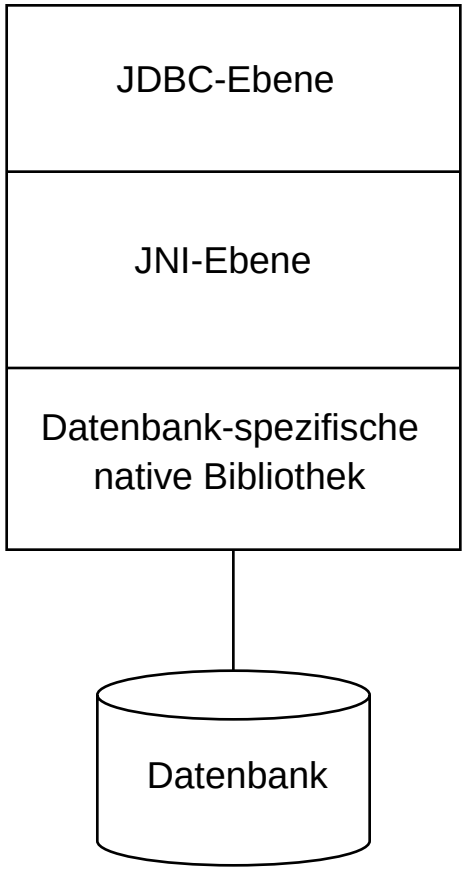
JDBC-Treiber (Typ 1)

- Hier wird eine generische native Bibliothek verwendet.
- Trotzdem wird oft eine Datenbank-spezifische native Bibliothek benötigt.
- Sehr langsam, da die Daten viele Ebenen durchlaufen müssen.
- Bsp: JDBC-ODBC-Brücke



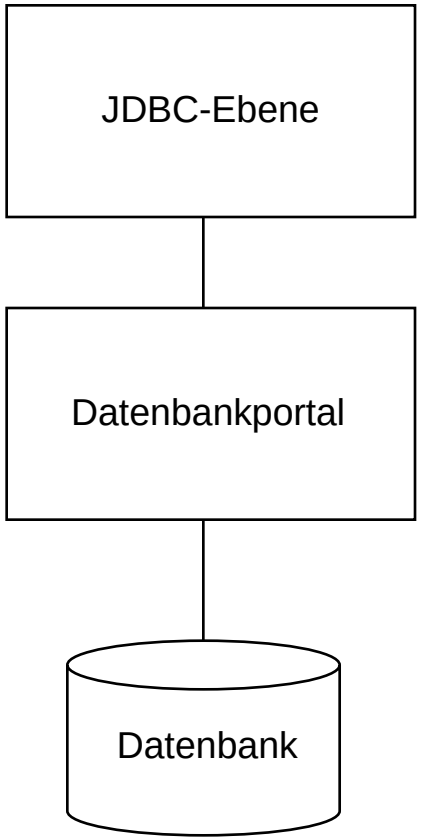
JDBC-Treiber (Typ 2)

- Keine generische native Bibliothek
- Schneller als Typ1.
- Aber plattformabhängig.



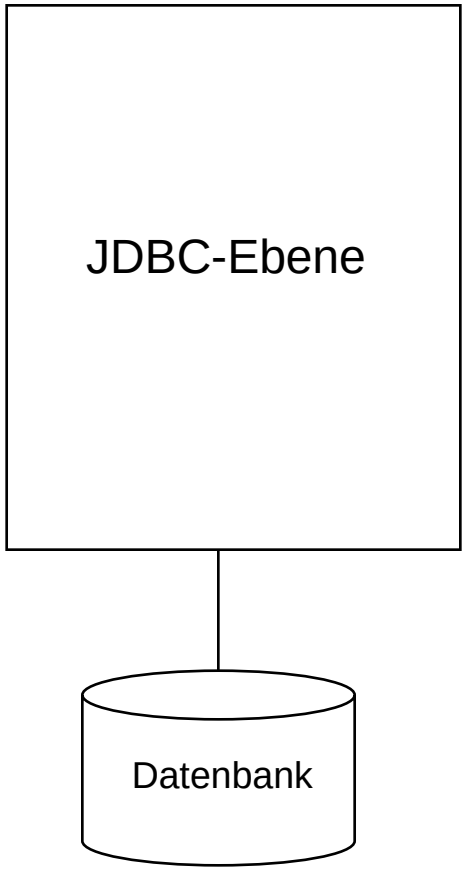
JDBC-Treiber (Typ 3)

- Reines Java.
- Kommuniziert über ein unabhängiges Protokoll mit einem Datenbankportal (Middleware).
- Wird normalerweise verwendet in Verbindung mit Applets.
- Kann unter Umständen sehr langsam werden.

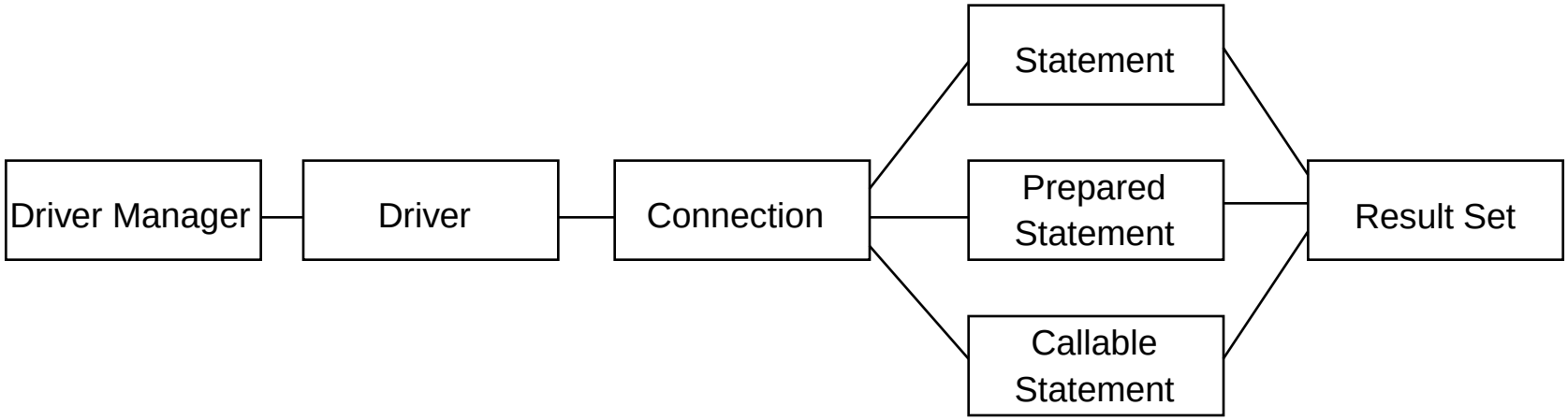


JDBC-Treiber (Typ 4)

- Reines Java
- Schneller als Typ 2
  - wegen des Just-In-Time(JIT)-Compilers
- Datenbank-spezifisch.
- Typ 4 ist heutzutage der Standard.



JDBC-Komponenten



PostgreSQL JDBC

Download unter

http://jdbc.postgresql.org/download.html

Bibliothek muss in das Projekt eingebunden werden

Anschließend können die Befehle verwendet werden

DriverManager

- DriverManager
  - stellt die Datenbankverbindung her
    - `public static Connection getConnection(String url)`
    - `public static Connection getConnection(String url, String username, String password)`
    - Die url gibt an, welche Datenbank verwendet wird und hat die Form
      - `jdbc:drivertype:[driversubtype]://params`
  - ist auch für das Laden des Treiber zuständig
    - z.B. `Class.forName("org.postgresql.Driver“);`
- Driver
  - Für die Herstellung der Datenbankverbindung zuständig

Connection (1)

- Connection
  - Datenbankinformationen holen
    - Metadaten über Datenbanken holen (mit der Methode `getMetaData`)
  - Datenbanktransanktionen verwalten
    - Mit dem auto-commit-Flag kann man angeben, ob die Transaktionen automatisch durchgeführt werden sollen oder nicht
      - `public void set AutoCommit(boolean autoCommitFlag)`
    - Transaktionen abschicken
      - `public void commit()`
    - Änderungen in Transaktionen rückgängig machen
      - `public void rollback()`
    - Verbindung schließen
      - `public void close()`

Connection (2)

- Connection
  - Datenbankanweisungen erstellen
    - Datenbankbefehle ausführen (für die 3 Arten von Statements)

    - Statements werden mit den folgenden Methoden erstellt
      - `public Statement createStatement()`
      - `public Statement createStatement(int resultSetType, int resultSetConcurrency)`

      - `public PreparedStatement prepareStatement(String sql)`
      - `public PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency)`

      - `public CallableStatement prepareCall(String sql)`
      - `public CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency)`

Statements (1)

- SQL-Anweisungen ausführen
  - Mit Ergebnisausgabe (SELECT-Anweisungen)
    - `ResultSet executeQuery(String sqlQuery)`
  - Ohne Ergebnisausgabe (INSERT, UPDATE, DELETE)
    - `int executeUpdate(String sqlUpdate)`
- Anweisungen schließen
  - `public void close()`

Statements (2)

- Prepared Statements
  - SQL-Anweisungen mit Parametern (mittels ‘?‘)
    - Beispiel

```
PreparedStatement pstmt =
con.prepareStatement(“select * from Person where
name = ?“);
```

  - Die Werte der Parameter werden mit folgenden methoden eingesetzt
    - `pstmt.setString(1, “Newo“);`
    - `pstmt.setNull(1, Types.INTEGER);`

Statements (3)

- Callable Statements
  - In der Datenbank gespeicherte Prozeduren aufrufen
  - Vorteil: meistens schnell
  - Nachteil: Syntax für Prozeduren kann sich je nach DBMS ändern
  - Beispiel

```
CallableStatement cstmt = con.prepareCall("{
call findPopularName ?}");
```

ResultSet (1)

- Ergebnismenge der SELECT-Querys
  - Mit der Methode `next()` kann nacheinander die Zeilen der Menge bearbeiten
  - Die `getString()` und `getObject()` Methoden sind die meistbenutzten Methoden um die Spalteninhalte herauszulesen
    - getString(int | String)
    - getBoolean(int | String)
    - getDate(int | String)
    - usw.

```
while(rs.next())        {
        String event = rs.getString("name");
        Object count = (Integer) rs.getObject("age");
}
```

ResultSet (2)

- Typen der Ergebnismenge
  - TYPE_FORWARD_ONLY (nur in eine Richtung durchlaufen)
  - TYPE_SCROLL_SENSITIVE (dynamische Sicht)
  - TYPE_SCROLL_INSENSITIVE (statische Sicht)

- Nebenläufigkeit der Ergebnismenge
  - CONCUR_READ_ONLY
    - Mit der Ergebnismenge kann aus der DB nur gelesen werden
  - CONCUR_UPDATABLE
    - Mit der Ergebnismenge kann man Zeilen aktualisieren, hinzufügen oder löschen

ResultSet (3)

- Methoden:
  - next() — Ruft die nächste Zeile auf
  - absolute(int position) — Bestimmte Zeile anspringen
  - afterLast() — Ans Ende springen
  - beforeFirst() — An den Anfang springen

  - setFetchDirection(int modus) *FetchForward, FetchReverse* — Ändert die Richtung, in der das ResultSet durchlaufen wird
  - setFetchSize(integer size) — Legt die Anzahl der Zeilen fest, die mit jedem Aufruf von next() abgerufen werden

ResultSet (4)

- Datensätze per ResultSet einfügen
  - Datensätze können direkt im ResultSet geändert werden

```
//leeren Datensatz holen
ResultSet rs = stmt.executeQuery()

//ResultSet zum Einfügen vorbereiten
rs.moveToInsertRow();
rs.updateString(„email“, „test@gmx.de“);
rs.updateInt(„age“, 28);

//Tatsächlich einfügen
rs.insertRow();
```

ResultSet (5)

- Datensätze per ResultSet ändern
  - Datensätze können direkt im ResultSet geändert werden

```
//Datensätze holen
ResultSet rs = stmt.executeQuery()

//ResultSet durchlaufen
while (rs.ext() {
        rs.updateString(„email“, „test@gmx.de“);
        rs.updateInt(„age“, 28);

        //Datensatz ändern
        rs.updateRow();
}
```

ResultSet (6)

- Datensätze per ResultSet löschen
  - Datensätze können direkt im ResultSet geändert werden

```
//Datensätze holen
ResultSet rs = stmt.executeQuery()

while (rs.ext() {
        //Datensatz löschen
        rs.deleteRow();
}
```

SQL Exceptions

```
try {
    //Load Driver
    //Get connection
    //Create statement object
    //Execute SQL query, get a ResultSet

} catch(ClassNotFoundException e) {
    out.println("Couldn't load database driver: " + e.getMessage());

} catch(SQLException e) {
    out.println("SQLException caught: " + e.getMessage());

} finally {
    //Always close the database connection
    try {
        if (con != null) con.close();
    } catch (SQLException ignored) {}
}
```

Transaktionen (1)

- SQL Statements zu einer Transaktion zusammenfassen
  - Entweder alle Statements einer Transaktion ausführen oder keins
  - Rollback bei Fehler

```
Connection con = getConnection();              //Connection holen
try {
      con.setAutoCommit(false)                 //Transaktionsmodus einschalten
      pstmt.prepareStatement(„INSERT INTO….)        //SQL Statements
      pstmt1.prepareStatement(„INSERT INTO….)

      pstmt.executeQuery();                    //Ausführungen zusammenfassen
      pstmt1.executeQuery();
      con.commit();                            //tatsächlich ausführen
}
```

Transaktionen (2)

```
try {
        siehe vorherige Folie
}  catch (SQLException se) {
        con.rollback();                    //Wenn Fehler ganze Transaktion rückgängig
}
```

Methode setSavePoint() zum definieren von Zwischenzielen. Dieser Savepoint kann an die rollback() Methode übergeben werden, um nur bis zu dem Zwischenziel rückgängig zu machen.

Beispiel

```
Connection con;
String driver = "org.postgresql.Driver";                    // DB Treiber
String DB_SERVER = "localhost:5432";                        // DB Server Adresse
String DB_NAME = "Musik";                                   // DB Name
String password = "regis";                                  // DB Passwort
String user = "regis";                                      // DB User
String url = "jdbc:postgresql://" + DB_SERVER + "/" + DB_NAME;

try {
// Laden der JDBC Bridge
Class.forName( driver );

// Aufbau der Verbindung
con = DriverManager.getConnection( url, user, password );

Statement stmt = con.createStatement();
    ResultSet rs=stmt.executeQuery("select name from person");

    while(rs.next()){
       System.out.println(rs.getString(1));
    }
}
```

Metadaten (1)

- Informationen über eine Tabelle
  - Metadaten eines `ResultSets`
  - Mit Hilfe der Klasse `ResultSetMetaData`
  - Man kann z.B. folgende Informationen anfordern
    - Anzahl der Spalten
      - `int getColumnCount();`
    - Name einer Spalte
      - `String getColumnLabel(int spaltenNr);`
    - SQL-Typ einer Spalte
      - `String getColumnTypeName(int spaltenNr);`
    - Name der Tabelle, aus der eine Spalte stammt
      - `String getTableName(int SpaltenNr);`
    - usw.

Metadaten (2)

- Informationen über eine Datenbank
  - Metadaten über zu der Verbindung gehörende Datenbank
  - Mit Hilfe der Klasse DatabaseMetaData
  - Man kann z.B. folgende Informationen anfordern
    - Kann die Datenbank nur gelesen oder kann auch in die Datenbank geschrieben werden?
      - `boolean isReadOnly();`
    - Lassen sich outer joins durchführen?
      - `boolean supportsOuterJoins();`
    - Sind gespeicherte Prozeduren erlaubt?
      - `boolean supportsStoredProcedures();`
    - usw.

SQL-Injektion

- Einschleusen von SQL-Code bei einer Datenbankanwendung
  - Daten löschen
  - Daten verändern
  - Daten ausspähen
  - (Administrator-) Rechte erlangen

- Schutz gegen SQL-Injektion
  - Verwendung von `PreparedStatement` (statt `Statement`)
    - Wertzuweisung nach dem Erzeugen der `PreparedStatement`-Instanz mit Hilfe von typsicheren Settern
    - PreparedStatements sollten auch wegen des Geschwindigkeitsvorteils vorgezogen werden

Literaturhinweise

- Mark Wutka:
  - J2EE Developer‘s Guide

- Java ist auch eine Insel
  http://www.galileocomputing.de/openbook/javainsel8