Kapitel 4

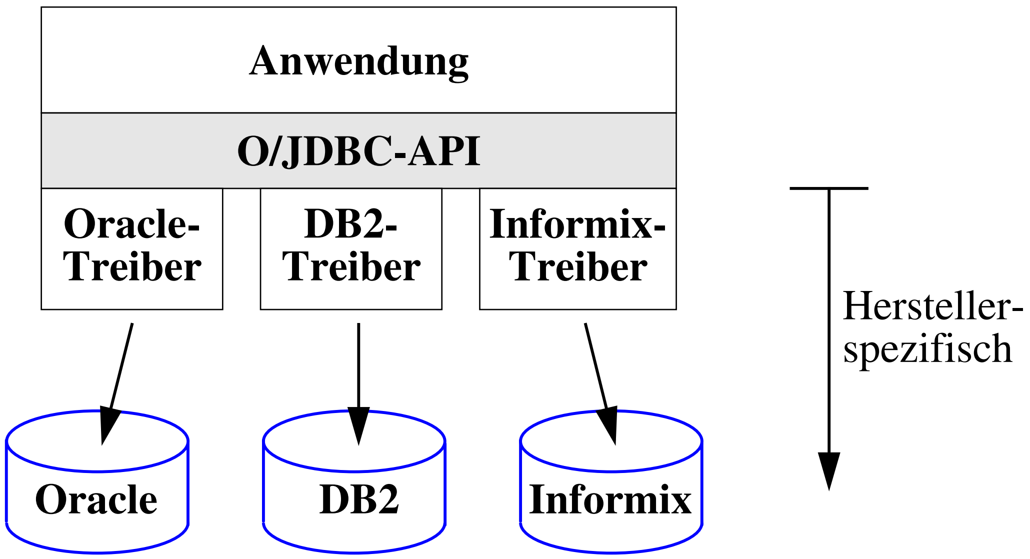
Datenbank-Gateways (insbes. JDBC)

Inhalt

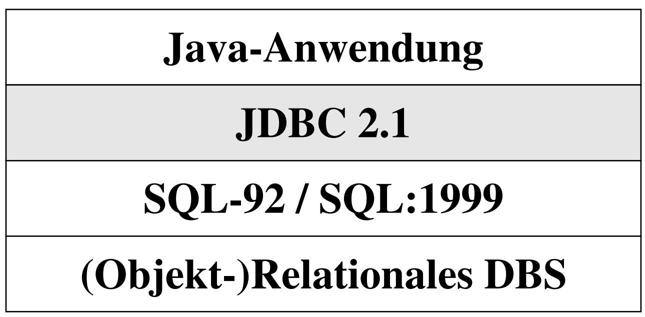
- ❑ Einführung (ODBC / JDBC)
- ❑ Architektur und Komponenten
- ❑ JDBC
  - ➪ Ausführung von SQL-Befehlen
  - ➪ Verarbeitung von (DB-)Daten in der Anwendung
- ❑ SQLJ
- ❑ Zusammenfassung

Einführung (1)

- ❑ Standard-APIs vom Typ CLI (Call Level Interface)
  - ➪ zum Zugriff auf (heterogene) Datenquellen;



  - ➪ ODBC: Open Database Connectivity
    - 1992 von Microsoft auf den Markt gebracht
    - ODBC-Treiber für nahezu alle DBVS verfügbar
  - ➪ JDBC: Java Database Connectivity
    - auf Grundlage von ODBC von SUN spezifiziert
    - Abstimmung auf Java, Vorteile von Java auch für API
    - Abstraktionsschicht zwischen Java-Programmen und SQL



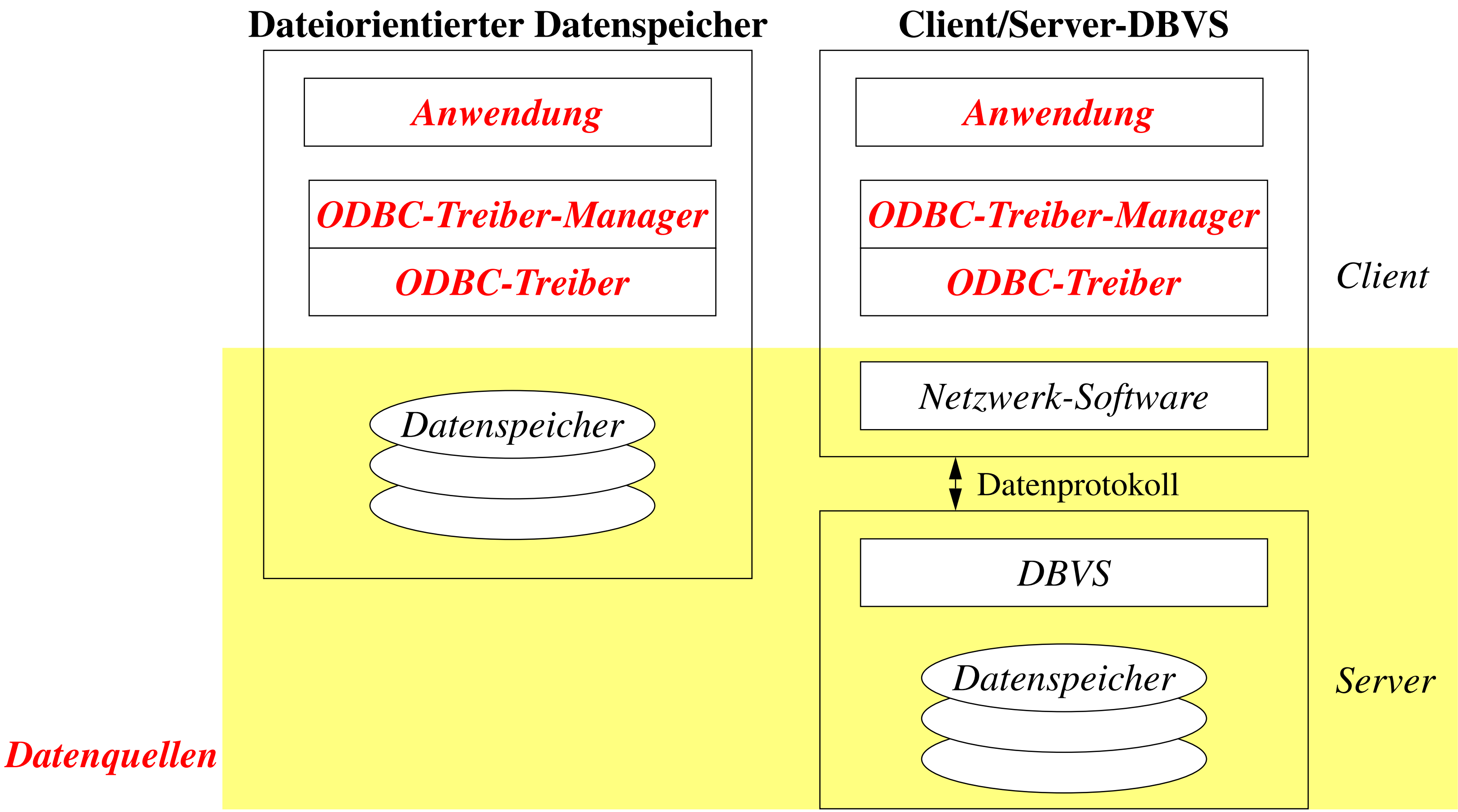
Einführung (2)

❑ JDBC-Beispiel

```
import java.sql.*;
import java.io.*;
public class Beispiel (
    public static void main (String args[]) {
        try {
            // Schritt 1: Aufbau einer Datenbankverbindung
                try {
                    Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
                }
                catch (ClassNotFoundException cex) {
                    System.err.println(cex.getMessage());
                }
                Connection conn = DriverManager.getConnection(
                    “jdbc:odbc:pizzaservice”,””,””);
                // Schritt 2: Erzeugen einer Tabelle
                Statement stmt = conn.createStatement();
                stmt.executeUpdate(“CREATE TABLE PizzaTabelle (“ +
                    “id INTEGER, “ +
                    “name CHAR(2), “ +
                    “preis FLOAT)” );
                // Schritt 3: Füllen der Tabelle
                stmt.executeUpdate(“INSERT INTO PizzaTabelle “ +
                    “(id, name, pries) VALUES(12, “Margherita”, 7.20)”);
                ...
                // Schritt 4: Absetzen einer Anfrage
                ResultSet rs = stmt.executeQuery(“SELECT * FROM “ +
                    “PizzaTabelle WHERE preis = 7.20”);
                // Schritt 5: Ausgabe des Ergebnisses
                while(rs.next()) {
                    int id = rs.getInt(“id”);
                    String name = rs.getString(“name”);
                    float preis = rs.getFloat(“preis”);
                    System.out.println(“Treffer: “ + name + “ , “ + preis);
                    }
                // Schritt 6: Beenden der Verbindung
                rs.close();
                stmt.close();
                conn.close();
            }
                catch(SQLException ex) {
                    System.err.println(ex.getMessage());
                }
        }
    }
```

Architektur und Komponenten (1)

- ODBC-Architektur

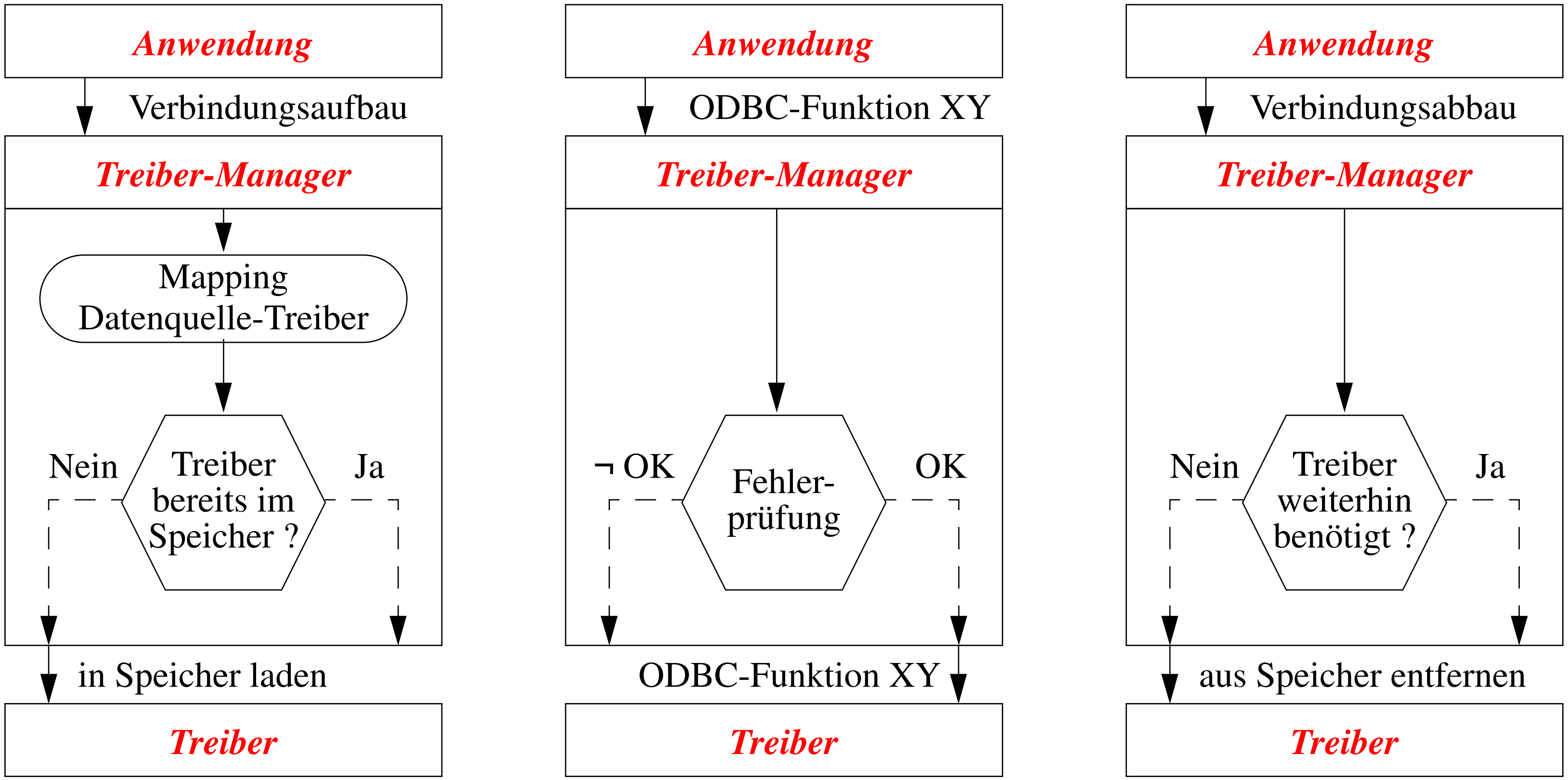


Architektur und Komponenten (2)

- ❑ Anwendungen
  - ➪ Programme, die ODBC-Funktionalität nutzen
  - ➪ Nutzung
    - Verbindungen zu Datenquellen aufnehmen
    - SQL-Anfragen an Datenquellen absetzen
    - Ergbnisse entgegennehmen
- ❑ Treiber-Manager
  - ➪ Verwaltung der Interaktion zwischen Anwendung und Treiber
  - ➪ realisiert (n:m)-Beziehung zwischen Anwendung und Treiber
  - ➪ Aufgaben
    - Laden/Löschen des Treibers
    - Mapping zwischen Treibern und Datenquellen
    - Weiterleitung/Logging von Funktions-/Methodenaufrufen
- ❑ Treiber
  - ➪ Aufgaben
    - Verarbeitung von ODBC-Aufrufen
    - Weiterleitung von SQL-Anfragen an Datenquellen
    - Ggf. Ausführung von SQL-Anfragen
    - Verbergen der Heterogenität verschiedener Datenquellen
  - ➪ Arten: Ein-Stufen- (nur ODBC), Zwei-Stufen-, Drei-Stufen-Treiber (und höhere)

Architektur und Komponenten (3)

❑ Treiber-Manager (Forts.)



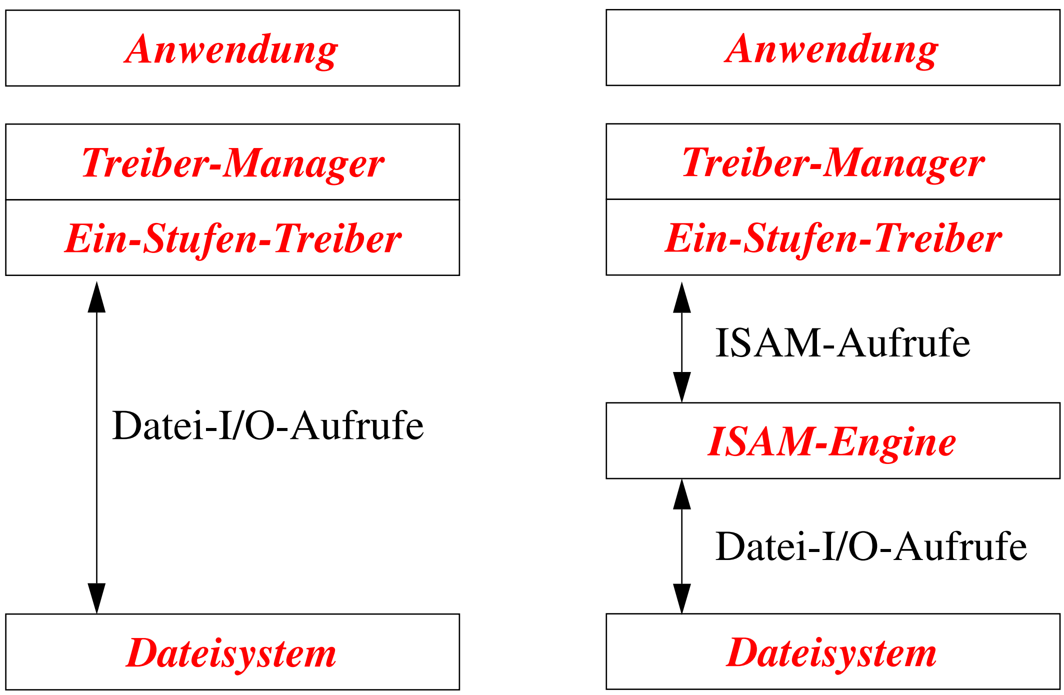
Architektur und Komponenten (4)

❑ **Treiber** (Forts.)

➩ **Ein-Stufen-Treiber**

- Zugriff auf Desktop-DBVS, ISAM- und flache Dateien
- Daten auf derselben Maschine wie Treiber

Zugriff auf flache Dateien ***Zugriff auf ISAM-Dateien***



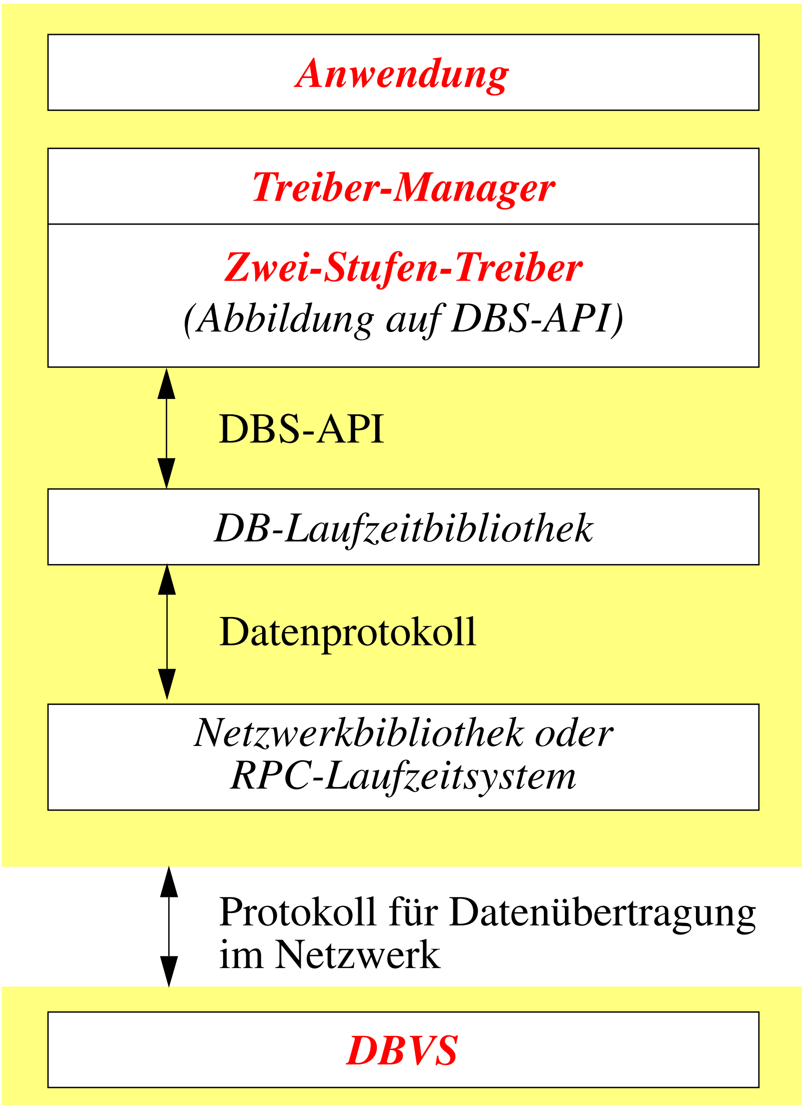
- Funktionalität:
  - ***komplette SQL-Verarbeitung***
    (Parsen, Optimierung, Bereitstellen des Ausführungsmoduls);
  - häufig keine Mehrbenutzer-/Transaktionsunterstützung

Architektur und Komponenten (5)

- **Treiber** (Forts.)
  - **Zwei-Stufen-Treiber**
    - klassische Client/Server-Unterstützung
      - Treiber übernimmt Client-Rolle im Datenprotokoll mit DBVS (Server)
    - Realisierungsmöglichkeiten
      - direkte Teilnahme am Datenprotokoll

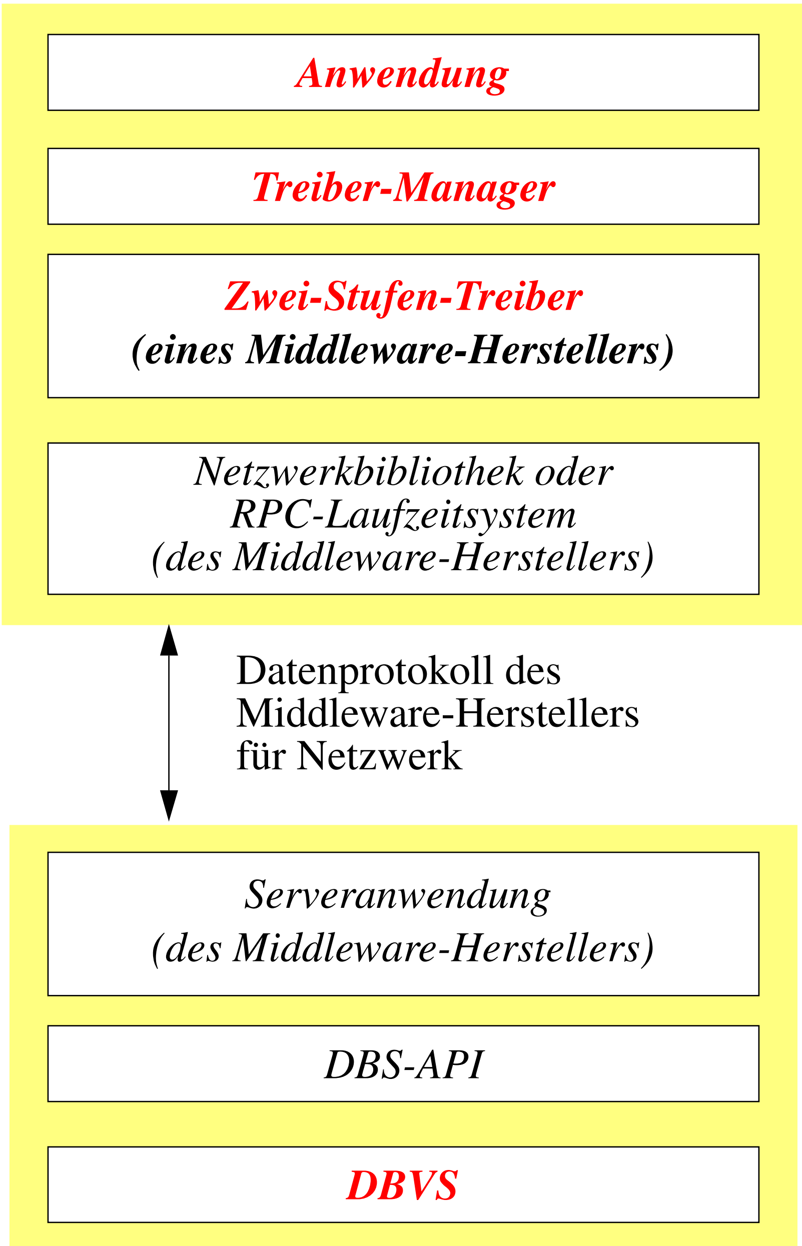
Architektur und Komponenten (6)

- **Treiber** (Forts.)
  - **Zwei-Stufen-Treiber** (Forts.)
    - Realisierungsmöglichkeiten (Forts.)
      - Abbildung von ODBC-Funktionen auf DBS-API



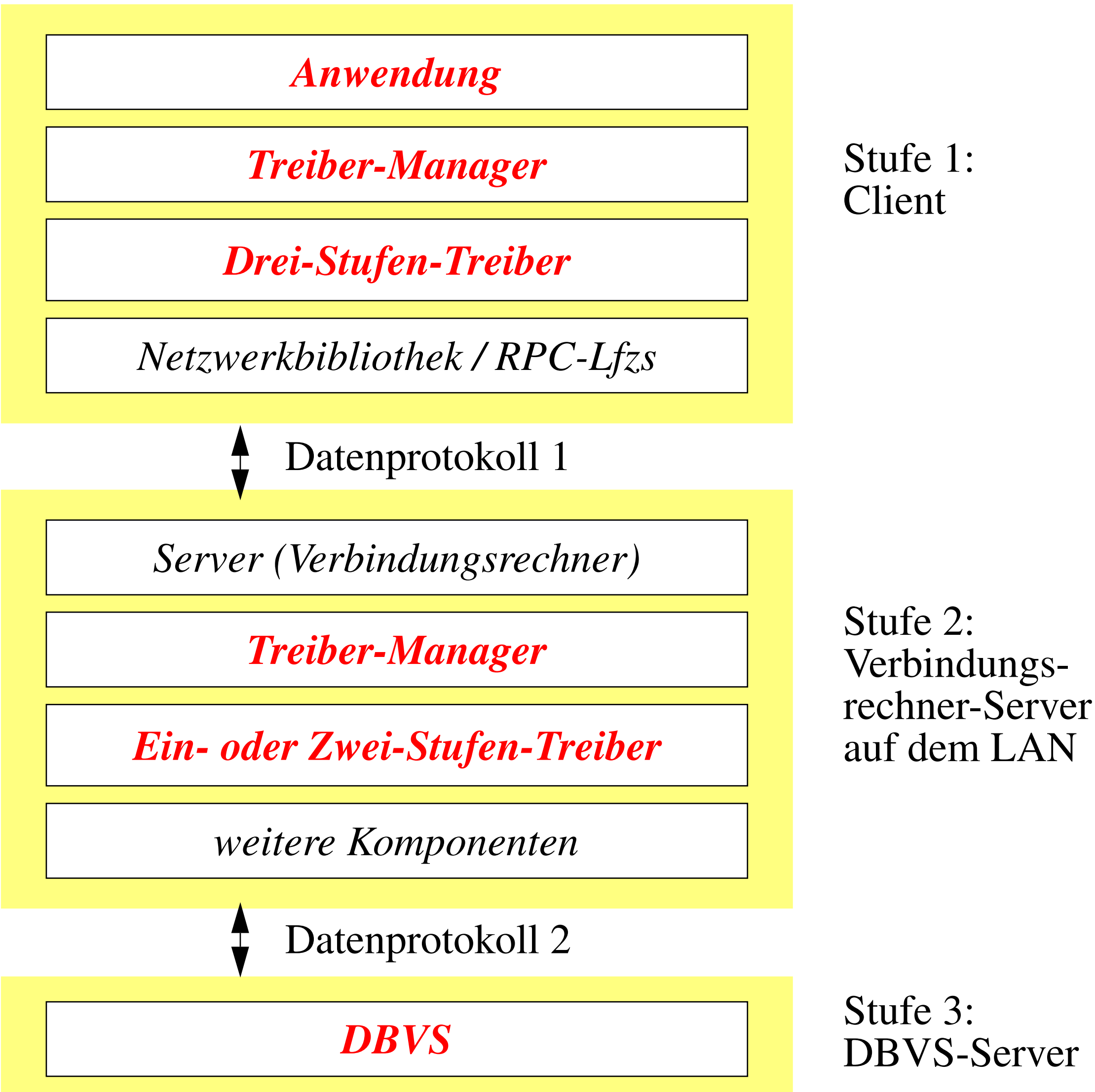
Architektur und Komponenten (7)

- **Treiber** (Forts.)
  - **Zwei-Stufen-Treiber** (Forts.)
    - Realisierungsmöglichkeiten (Forts.)
      - Middleware-Lösung



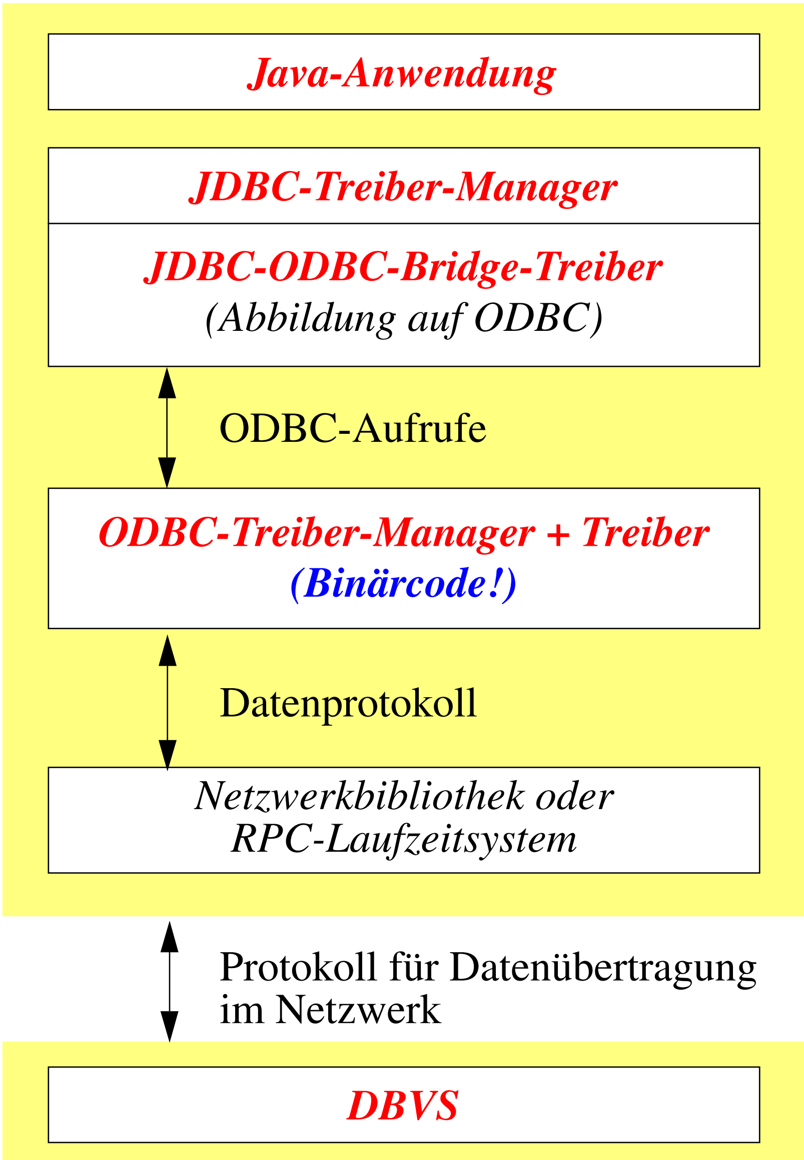
Architektur und Komponenten (8)

- **Treiber** (Forts.)
  - **Drei-Stufen-Treiber** (und höhere)
    - Verbindungsserver
      - Verbindung mit einem oder mehreren DBVS
      - Verlagerung der Komplexität von Client zu Verbindungs-server
      - theoretisch beliebig viele Verbindungsstufen



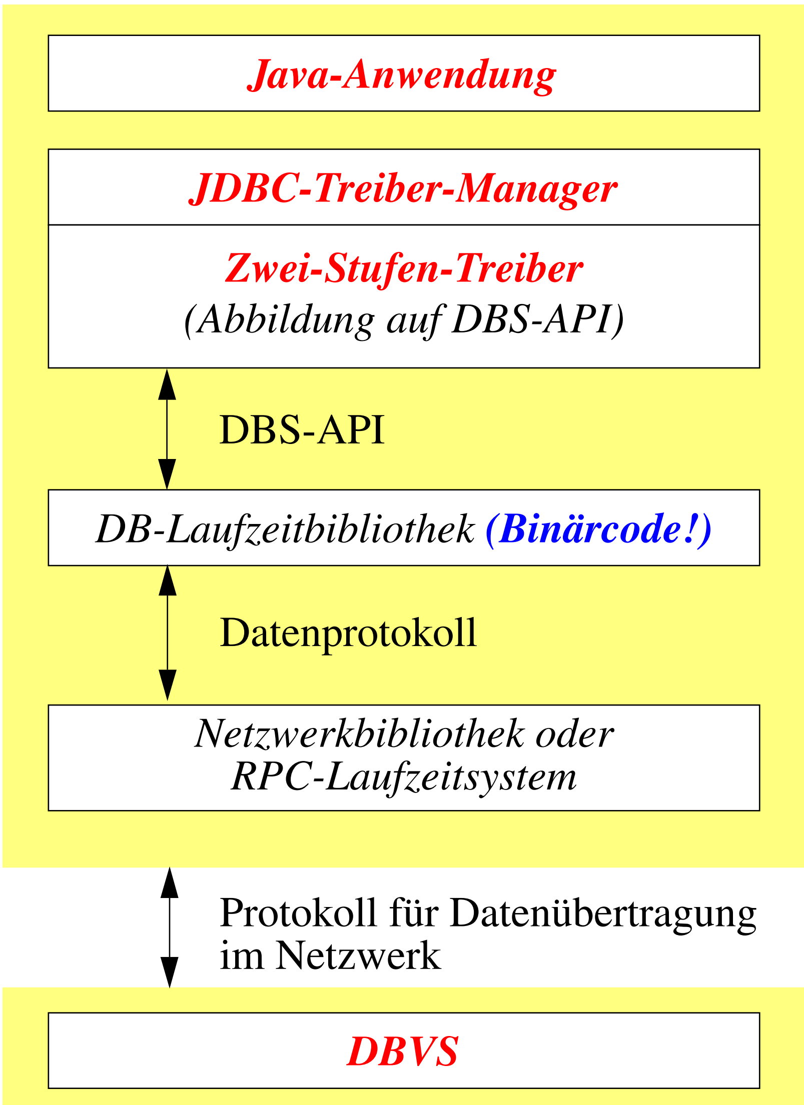
Architektur und Komponenten (9)

- **Treiber** (Forts.)
  - **JDBC-Treiber** (Forts.)
    - Typ-1-Treiber
      - Binärcode beim Client erforderlich
      - Verwendung der JDBC-ODBC-Bridge



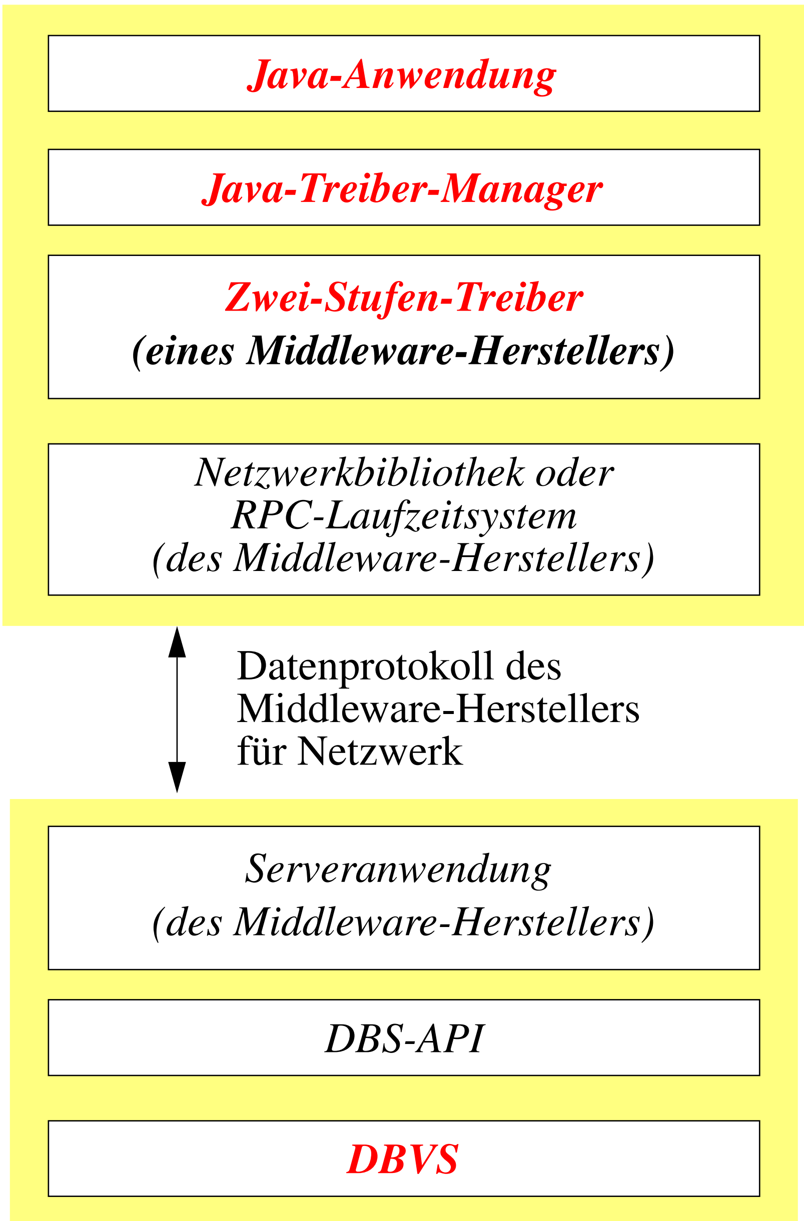
Architektur und Komponenten (10)

- **Treiber** (Forts.)
  - **JDBC-Treiber** (Forts.)
    - Typ-2-Treiber
      - Native-API-Partial-Java-Treiber
      - Binärcode beim Client erforderlich
      - Abbildung des JDBC-Interface auf DBS-API



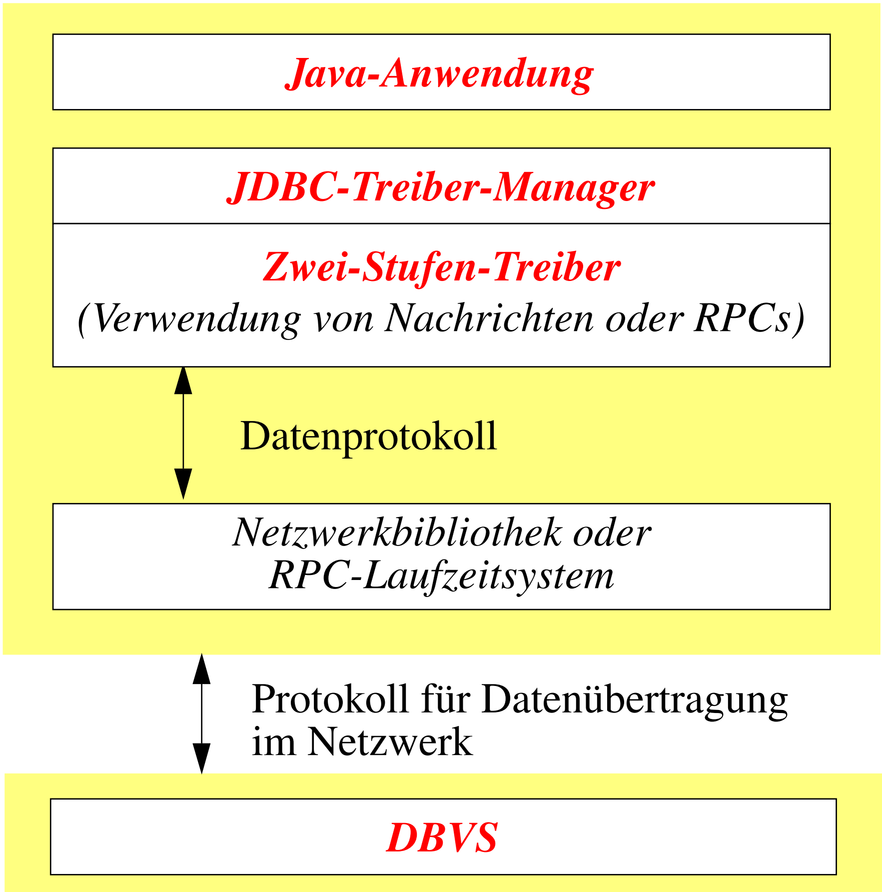
Architektur und Komponenten (11)

- **Treiber** (Forts.)
  - **JDBC-Treiber** (Forts.)
    - Typ-3-Treiber
      - Net-Protocol-All-Java-Treiber
      - kein Binärcode beim Client erforderlich
      - Middleware-Lösung



Architektur und Komponenten (12)

- **Treiber** (Forts.)
  - **JDBC-Treiber** (Forts.)
    - Typ-4-Treiber
      - Native-Protocol-All-Java-Treiber
      - kein Binärcode beim Client erforderlich
      - direkte Teilnahme am Datenprotokoll



Architektur und Komponenten (13)

- **Treiber** (Forts.)
  - Aufgaben im einzelnen
    - Verwalten von Verbindungen
    - Fehlerbehandlung
      - Standard-Fehlerfunktionen
      - Standard-Fehlercodes
      - Fehlermeldungen
      - ...
    - Umwandlung von SQL-Anfragen
      - bei Abweichungen des DBVS vom Standard
    - Datentypumwandlungen
    - Katalogfunktionen
      - Umwandlung von Zugriffen auf Metadaten
    - Informationsfunktionen
      - geben Informationen über Treiber (selbst), zugehörige Datenquellen und von der Datenquelle unterstützte Datentypen
    - Optionsfunktionen
      - Parameter für Verbindungen und Anweisungen (z. B. Wartezeiten für Abarbeitung von Anweisungen)

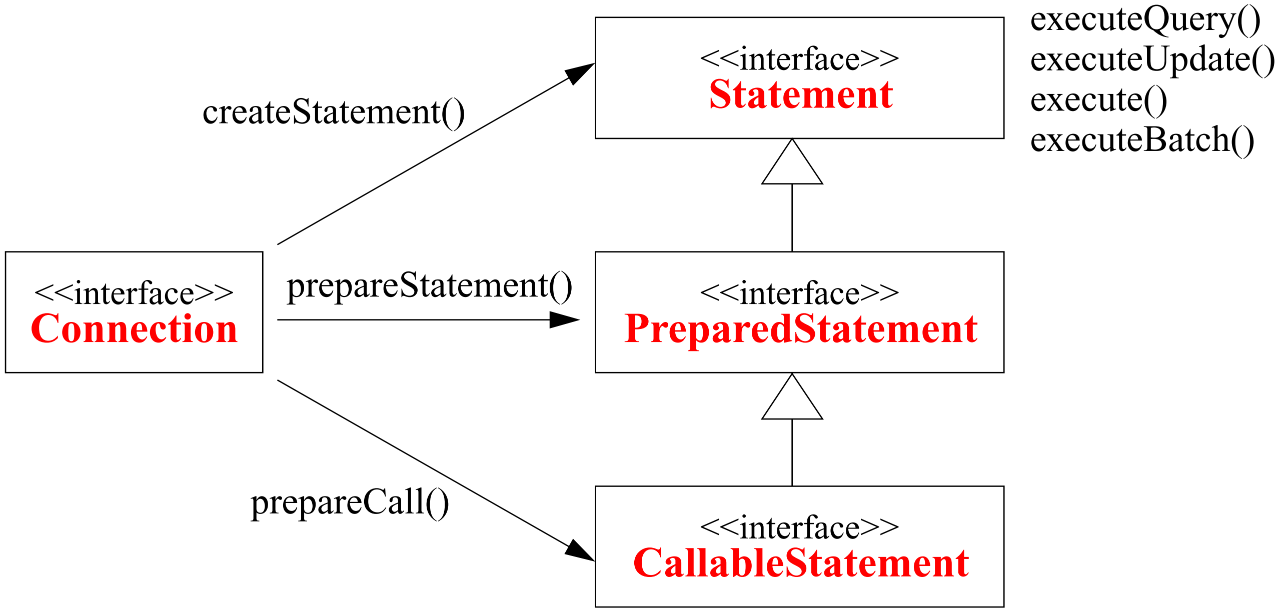
Architektur und Komponenten (14)

- JDBC-Datenquellen
  - Verbindung zu Datenquelle über Verbindungs-URL: ‘**jdbc: <*subprotokoll*>:<*subname*>**’
    - ***subprotokoll***: Name des Treibers oder des Datenbankverbindungsprotokolls
    - ***subname***: dient der Identifikation des DBS; abhängig vom Subprotokoll
  - Beispiele:
    - **jdbc:odbc:kunden;UID=John;PWD=Maja; CacheSize=20**
    - **jdbc:openlink://kundenhost.firma.com:2000/ SVT=Oracle7/Database=kunden**

- JDBC-Verbindungsmodelle
  - JDBC-Methode: ***DriverManager.getConnection***
    - Parameter: JDBC-URL, User-ID, Paßwort
  - JDBC-Methode: ***Driver DriverManager.getDriver(String url)***
    - Treiber-Manager fragt alle bei ihm registrierten Treiber, ob sie ***url*** auswerten können;
    - falls einer der Treiber TRUE zurückliefert, kann mit ***Driver.getPropertyInfo(String url, Properties info)*** ermittelt werden, welche Parameter noch fehlen; ***DriverManager.getConnection(String url, Properties info)*** stellt dann die Verbindung her;

Ausführen von SQL-Befehlen mit JDBC

- JDBC-Interfaces Connection und Statement



Verarbeitung von Daten mit JDBC (1)

- ResultSet
  - **getXXX**-Methoden
  - Scrollable-ResultSets
  - Updatabale-ResultSets
- Datentypen
  - Konvertierungsfunktionen
  - Streams zur Übertragung großer Datenmengen
  - Unterstützung von SQL:1999-Datentypen ab JDBC 2.0
    - LOBS (BLOBS, CLOBS)
    - Arrays
    - UDTs
    - Referenzen
- Metadaten
  - Funktionen für Metadaten-Lookup
  - wichtig für generische Anwendungen
- Transaktionen
  - Auto-Commit, manuelles Commit
  - ReadOnly, ReadWrite
  - Isolationsebenen

Verarbeitung von Daten mit JDBC (2)

- Exception Handling
- JDBC 2.1 Optional Package
  - DataSource (JNDI)
  - Connection Pooling
  - Verteilte Transaktionen (X/OPEN DTP, JTA)
  - RowSets
- DataSource Interface
  - Motivation: mehr Portabilität durch Abstraktion von Driver-spezifischen Verbindungsdetails
  - Anwendung benutzt logischen Namen zum Verbindungsaufbau über Java Naming and Directory Service
  - ermöglicht Erzeugen, Registrieren, Rekonfigurieren, Neuzuordnung zu einer anderen physischen Datenbank ohne Mitwirken der Anwendung (z.B. über administrative Schnittstellen eines Anwendungsservers).
- Connection Pool
  - ermöglicht Wiederverwendung von physischen Verbindungen zu Datenbanken
  - dient Verbesserung der Performanz, Skalierbarkeit
  - kann (genau wie verteilte Transaktionsverarbeitung) durch DataSource, Connection interfaces "versteckt" werden

SQLJ (1)

- ❑ (Statisches) SQL eingebettet in Java
  - ➪ kombiniert Vorteile von eingebettetem SQL mit Anwendungsportabilität (“binary portability”)
  - ➪ nutzt JDBC als “Infrastruktur”, kann mit JDBC Aufrufen in der gleichen Anwendung kombiniert werden
  - ➪ ANSI/ISO-Standard: SQL/OLB (Object Language Bindings)
- ❑ Vorteile von SQLJ gegenüber JDBC (Entwicklung)
  - ➪ bessere sprachliche Einbettung
  - ➪ vereinfachte Programmierung, kürzere Programme
  - ➪ Prüfung der Typkorrektheit von Hostvariablen, syntaktischen Korrektheit u. Schemakorrektheit von SQL Befehlen zur Entwicklungszeit
  - ➪ ermöglicht Authorisierung zur Programmausführung (statt Tabellenzugriff)
- ❑ Mögliche Performanzvorteile
  - ➪ SQL Übersetzung, Zugriffskontrolle und Optimierung zum Kompilierungszeitpunkt möglich
  - ➪ herstellerspezifische Optimierungen werden unterstützt (SQLJ Customizer)
- ❑ Geringe Flexibilität
  - ➪ deshalb Interoperabilität mit JDBC notwendig

SQLJ (2)

- Beispiel: single row select
  - **SQLJ**:
    ```
    #sql { SELECT ADDRESS INTO :addr FROM EMP
     WHERE NAME=:name };
    ```
  - **JDBC**:
    ```
    java.sql.PreparedStatement ps =
      con.prepareStatement("SELECT ADDRESS
                           FROM EMP WHERE NAME=?");
    ps.setString(1, name);
    java.sql.ResultSet names = ps.executeQuery();
    names.next();
    name = names.getString(1);
    names.close();
    ```

- Iteratoren zur Verarbeitung von Mengen von Resultatstupeln
  - vergleichbar mit SQL Cursor, JDBC ResultSet
  - jede Deklaration eines SQLJ Iterators in der Anwendung führt zur Generierung einer Iteratorklasse
  - generische Methoden zur Iteration
  - “Postitioned Iterator” ermöglicht Zugriff auf Resultate über FETCH ... INTO ...
  - “Named Iterator” erhält Zugriffsmethode für jede Spalte der Resultatstabelle

SQLJ (3)

- Binäre Portabilität
  - Java als plattformunabhängige Wirtssprache
  - generischer SQLJ-Vorübersetzer (statt herstellerspezifischer Technologie)
  - generierter Code nutzt "Standard" JDBC per default
  - kompilierte SQLJ Anwendung (Java bytecode) ist portabel (herstellerunabhängig)
  - herstellerspezifische Anpassung/Optimierung ist nach der Übersetzung möglich (Customizer)

Zusammenfassung

- Gateways
  - ODBC / JDBC
  - erlauben den Zugriff auf heterogene Datenquellen mittels Standardsprache
  - keine Integration
  - kapseln die herstellerabhängigen Anteile
  - hohe Akzeptanz; fast alle Hersteller von Software zur Verwaltung von Datenquellen liefern zugehörige Treiber
- JDBC
  - ’für Java’, ’in Java’
  - besondere Bedeutung, da auch im Rahmen von weiteren Middleware-Technologien zu Zwecken des DB-Zugriffs genutzt (z. B. EJB, J2EE)
- SQLJ
  - verbindet Vorteile der Einbettung von SQL in Wirtssprache (hier Java) mit Herstellerunabhängigkeit, Portabilität