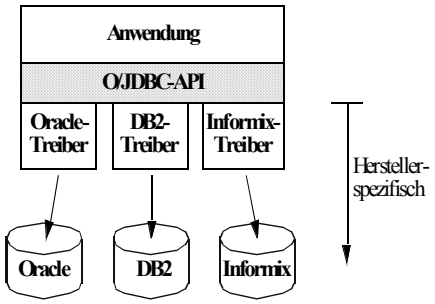
Kapitel 3
DB-Gateways

Inhalt:

- Überblick
- Architektur und Komponenten
- JDBC
- SQLJ
- Zusammenfassung

Standard-APIs vom Typ Call Level Interface

- Einheitlicher Zugriff auf Daten
  - Anfragesprache
  - Metadaten (z.T. Datenmodell)
  - Programmierschnittstelle
- Dynamisches, spätes Binden an DBS
  - Aufrufschnittstelle (CLI)
    - keine herstellerspez. Pre-Compiler
  - Dynamisches Binden von Laufzeitbibliotheken
  - Späte Anfrageübersetzung
- Gleichzeitiger Zugriff auf mehrere DBMS
  - Architektur unterstützt arbeiten mit (mehreren) herstellerspez. Treibern
  - Koordination durch Treibermanager
- Herstellerspezifische Erweiterungen möglich

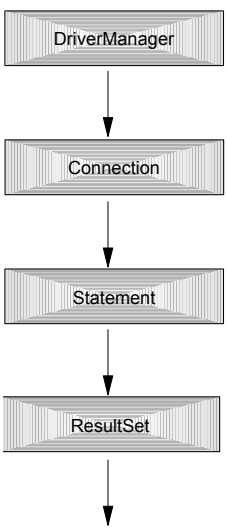


Entwicklung

- 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

Java-Anwendung
JDBC 3.0
SQL-92 / SQL:1999
(Objekt-)Relationales DBS

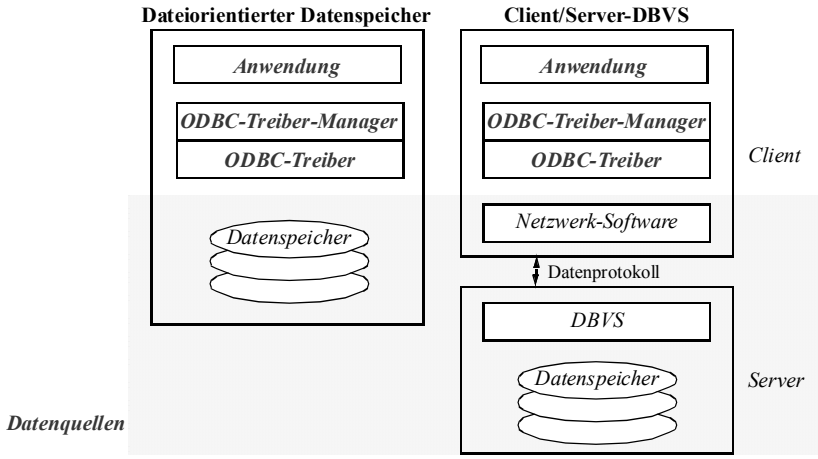
Beispiel: JDBC




```
String url = "jdbc:db2:mydatabase";
Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver");
Connection con = DriverManager.getConnection(url, "stefan",
     "sqljava");

String sqlstr = "SELECT * FROM Employees WHERE dept = 1234";
Statement stmt = con.createStatement( );

ResultSet rs = stmt.executeQuery(sqlstr);

while (rs.next() ) {
     String a = rs.getString(1);
     String str = rs.getString(2);
     System.out.print(" empno= " + a);
     System.out.print(" firstname= " + str);
     System.out.print("\n");
}
```

Architektur

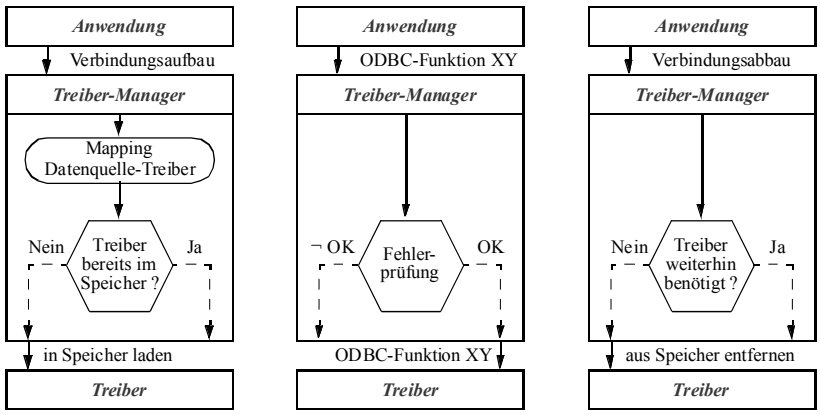
- Beispiel: ODBC-Architektur



Komponenten

- Anwendungen
  - Programme, die DB CLI-Funktionalität nutzen
  - Nutzung
    - Verbindungen zu Datenquellen aufnehmen
    - SQL-Anfragen an Datenquellen absetzen
    - Ergebnisse entgegen nehmen (und verarbeiten)
- 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
    - Einfache Fehlererkennung
- Treiber
  - Verarbeitung von CLI-Aufrufen
  - Weiterleitung von SQL-Anfragen an Datenquellen
  - Ggf. Ausführung von SQL-Anfragen
  - Verbergen der Heterogenität verschiedener Datenquellen

Treiber-Manager

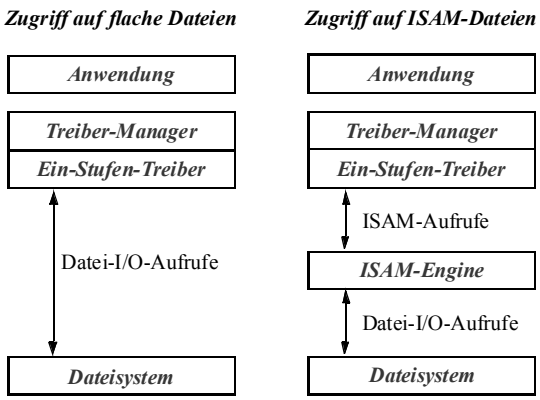


Realisierungsalternativen

- Treiberarten
  - Ein-Stufen-Treiber (nur ODBC)
  - Zwei-Stufen-Treiber
  - Drei-Stufen-Treiber (und höhere)
- JDBC Treibertypen
  - Typ 1: JDBC-ODBC bridge
  - Typ 2: Part Java, Part Native
  - Typ 3: Intermediate DB Access Server
  - Typ 4: Pure Java
- Anwendungsprogramm ist unabhängig von Realisierungsalternativen

Ein-Stufen-Treiber

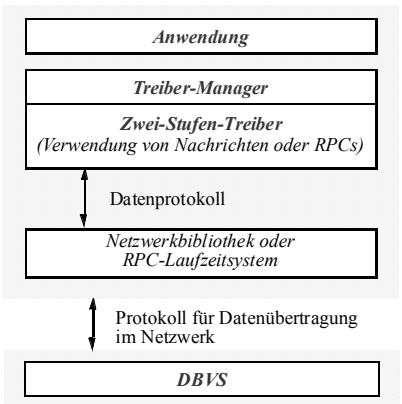
- Zugriff auf Desktop-DBVS, ISAM- und flache Dateien
- Daten auf derselben Maschine wie Treiber
- Funktionalität:
  - komplette SQL-Verarbeitung (Parsen, Optimierung, Bereitstellen des Ausführungsmoduls)
  - häufig keine Mehrbenutzer- bzw. Transaktionsunterstützung



Zwei-Stufen-Treiber

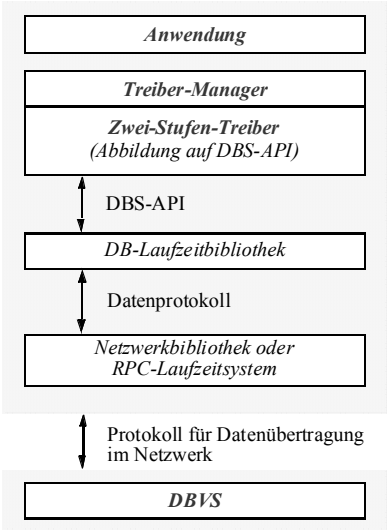
- klassische Client/Server-Unterstützung
  - Treiber übernimmt Client-Rolle im Datenprotokoll mit DBVS (Server)
- Realisierungsmöglichkeiten
  - direkte Teilnahme am Datenprotokoll
  - Abbildung von ODBC-Funktionen auf DBS-API
  - Middleware-Lösung

direkte Teilnahme am Datenprotokoll



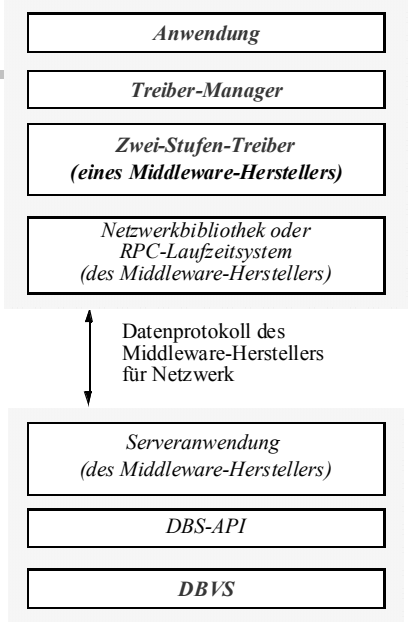
Zwei-Stufen-Treiber (Forts.)

Abbildung von ODBC-Funktionen auf DBS-API



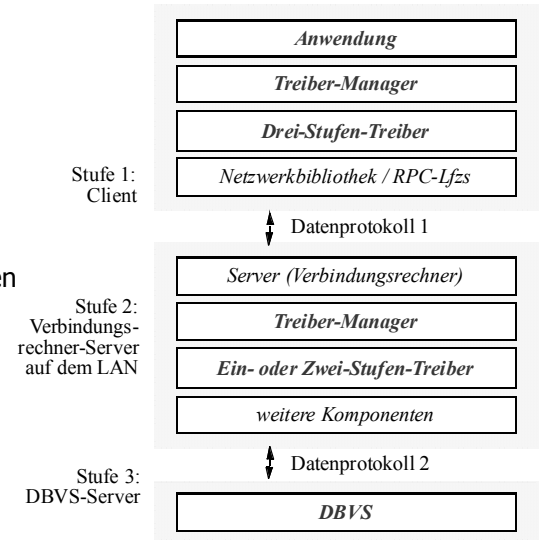
Zwei-Stufen-Treiber (Forts.)

Middleware-Lösung



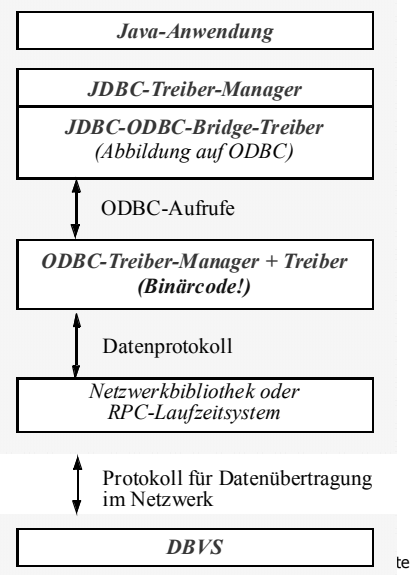
Drei-Stufen-Treiber (und höhere)

- Verbindungsserver
- Verbindung mit einem oder mehreren DBVS
- Verlagerung der Komplexität von Client zu Verbindungsserver
- theoretisch beliebig viele Verbindungsstufen



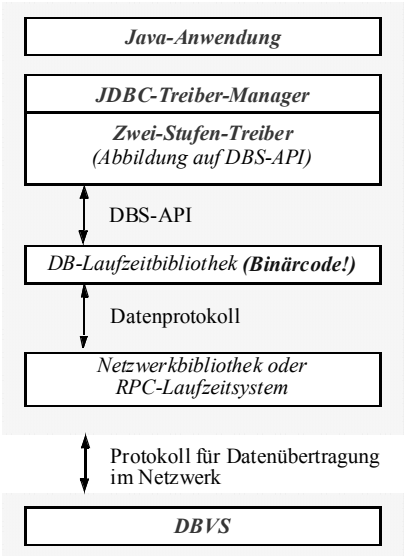
JDBC-Treiber Typ 1

- Verwendung der JDBC-ODBC-Bridge
  - benutzen des Java Native Interface (JNI)
- Binärcode beim Client erforderlich
- Nicht für Applet-basierten DB-Zugriff geeignet
  - security ...



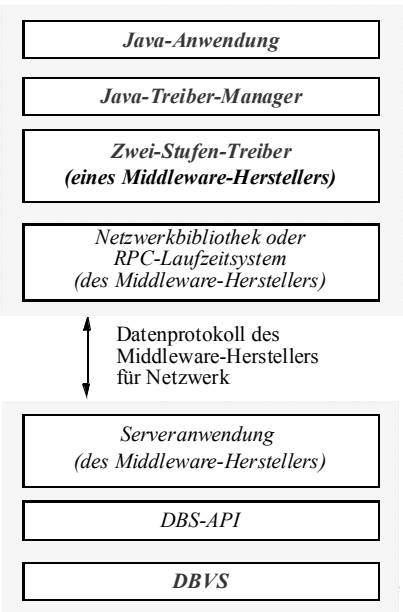
JDBC-Treiber Typ 2

- Native-API-Partial-Java-Treiber
  - Abbildung des JDBC-Interface auf DBS-API
- Binärcode beim Client erforderlich
- Nicht für Applet-basierten DB-Zugriff geeignet



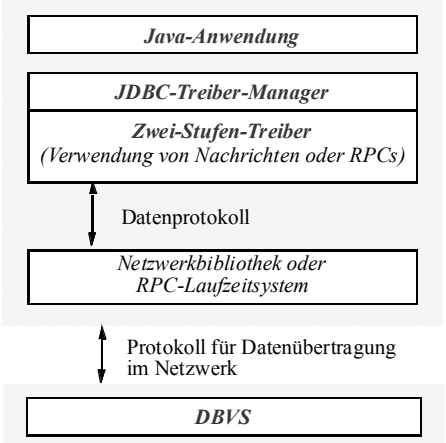
JDBC-Treiber Typ 3

- Net-Protocol-All-Java-Treiber
  - Treiber kommuniziert mit JDBC server/gateway über DBMS-herstellerunabhängiges Netzwerkprotokoll
  - JDBC server/gateway arbeitet mit DBMS API
- Middleware-Lösung
- kein Binärcode beim Client erforderlich
- Applet-basierter DB-Zugriff möglich



JDBC-Treiber Typ 4

- Native-Protocol-All-Java-Treiber
- direkte Teilnahme am Datenprotokoll
- kein Binärcode beim Client erforderlich



Treiber - Aufgaben im Einzelnen

- Verwalten von Verbindungen
- Fehlerbehandlung
  - Standard-Fehlerfunktionen/-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)

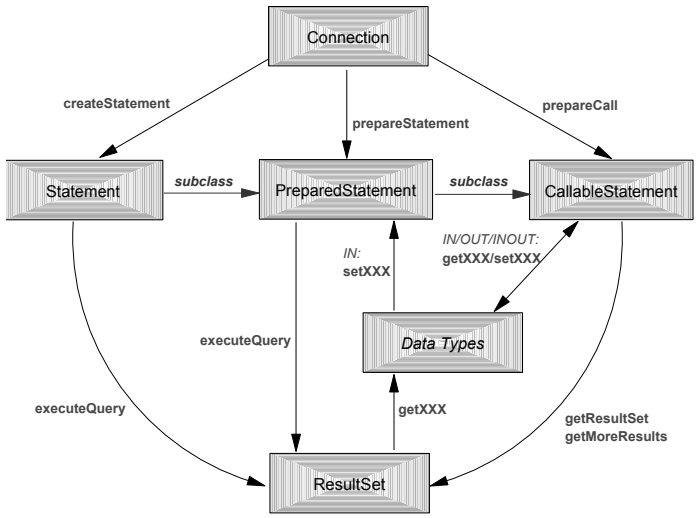
JDBC - Datenquellen

- "Traditionelles" Verbindungsmodell
  - 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 – Traditionelles Verbindungsmodell

- 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;

JDBC – Wichtigste Schnittstellen



JDBC – Beispiel (1)

```
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)" );
```

JDBC - Beispiel (2)

```
// Schritt 3: Füllen der Tabelle
stmt.executeUpdate("INSERT INTO PizzaTabelle(id, name, preis) 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());
} } }
```


JDBC – Verarbeiten von Anfrageergebnissen

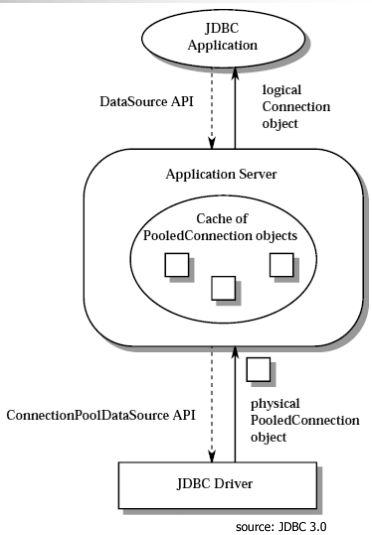
- Verarbeiten von DB-Daten mit JDBC
  - ResultSet
    - **getXXX**-Methoden
    - Scrollable-ResultSets
    - Updatable-ResultSets
  - Datentypen
    - Konvertierungsfunktionen
    - Streams zur Übertragung großer Datenmengen
    - Unterstützung von SQL:1999-Datentypen ab JDBC 2.0
      - LOBS (BLOBS, CLOBS)
      - Arrays
      - UDTs
      - Referenzen

JDBC DataSource

- 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
- Schritte
  - DataSource-Objekt wird erzeugt, konfiguriert, und mit JNDI registriert
    - Administrationsfunktion eines Anwendungsservers
    - geschieht ausserhalb der Anwendungskomponente
  - Anwendung benutzt logischen Namen um DataSource-Objekt zu erlangen
    - JNDI lookup
    - keine Treiberspezifischen Details sichtbar
  - Anwendung erzeugt mit Hilfe der DataSource ein Connection-Objekt
    - DataSource.getConnection( )
  - Rekonfiguration der DataSource möglich, ohne dass die Anwendung betroffen ist

Connection Pooling

- Dient Verbesserung der Performanz, Skalierbarkeit
  - Verbindungsaufbau ist teuer, resourcenintensiv
    - Bereitstellen von Kommunikations- und Speicherresourcem
    - Authentifikation, Erzeugen von Sicherheitskontext
- Server-seitige Anwendungskomponenten
  - DB-Zugriff oft über wenige (gemeinsame) Benutzerkennungen
  - DB-Verbindung wird meist nur für die Dauer eines (kurzen) Verarbeitungsschritts gehalten
- Ermöglicht Wiederverwendung von physischen Verbindungen zu Datenbanken
  - open -> "get connection from pool"
  - close -> "return connection to pool"
- Connection pooling kann (genau wie verteilte Transaktionsverarbeitung) durch DataSource, Connection interfaces "versteckt" werden
  - JDBC definiert zusätzliche implementierungsorientierte Interfaces
  - für die Anwendung nicht relevant

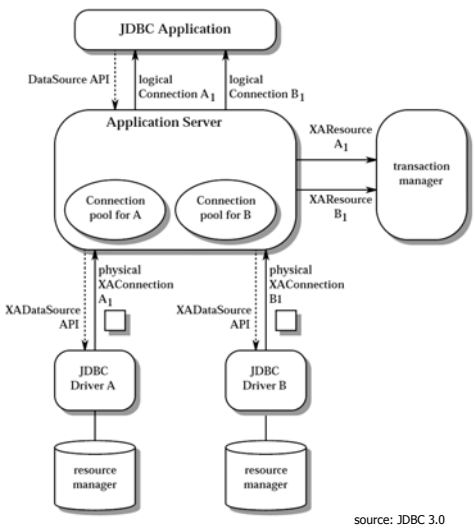


source: JDBC 3.0

Transaktionen in JDBC

- Connection interface – transaktionsorientierte Methoden für lokale TAs
  - commit()
  - rollback()
  - get/setTransactionIsolation()
    - NONE, READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE
  - get/setAutoCommit()
- Verteilte Transaktionsverarbeitung
  - erfordert Interaktion mit Transaktionsmanager
    - X/Open DTP, Java Transaction Service (JTS)
  - Demarkation der Transaktionsgrenzen
    - Java Transaction API (JTA)
      - UserTransaction Objekt
    - NICHT über Connection-Interface
  - JDBC definiert zusätzlich Interfaces zur Implementierung der Treiberfunktionalität
    - XADataSource, XAConnection, ...

Verteilte Transaktionen mit JDBC



source: JDBC 3.0

JDBC – Weitere Funktionen

- Metadaten
  - Funktionen für Metadaten-Lookup
  - wichtig für generische Anwendungen
- Exception Handling
- RowSets
- Batch Updates
- Savepoints
- ...

SQLJ

- (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 und Schemakorrektheit von SQL-Befehlen zur Entwicklungszeit
  - ermöglicht Authorisierung zur Programmausführung (statt Tabellenzugriff)
- Mögliche Performanzvorteile
  - Übersetzung und Optimierung von SQL-Anweisungen sowie Zugriffskontrolle zum Kompilierungszeitpunkt möglich
  - herstellerspezifische Optimierungen werden unterstützt (SQLJ Customizer)
- Geringe Flexibilität
  - deshalb Interoperabilität mit JDBC notwendig

SQLJ - Beispiel

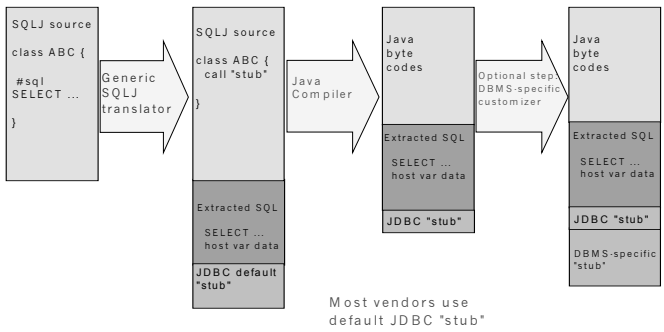
- Single row select
  - SQLJ:
    ```
    #sql { SELECT ADDRESS INTO :addr FROM EMP
      WHERE NAME=:name };
    ```
  - JDBC:
    ```
    java.sql.PreparedStatement ps =
      con.prepareStatement("SELECT ADDRESS FROM EMP WHERE NAME=?");
    ps.setString(1, name);
    java.sql.ResultSet names = ps.executeQuery();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


SQLJ - Iteratoren

- Iteratoren zur Verarbeitung von Mengen von Ergebnistupeln
  - vergleichbar mit SQL-Cursor, JDBC-ResultSet
  - jede Deklaration eines SQLJ-Iterators in der Anwendung führt zur Generierung einer Iteratorklasse
  - generische Methoden zur Iteration
  - "Positioned Iterator" ermöglicht Zugriff auf Ergebnisse über FETCH ... INTO ...
  - "Named Iterator" erhält Zugriffsmethode für jede Spalte der Ergebnistabelle

SQLJ - 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 (insbes. J2EE)
- SQLJ
  - verbindet Vorteile der Einbettung von SQL in Wirtssprache (hier Java) mit Herstellerunabhängigkeit, Portabilität