J2EE-Praktikum

JDBC

Peter Thiemann

Universität Freiburg

J2EE-Praktikum, WS2005/2006

Inhalt

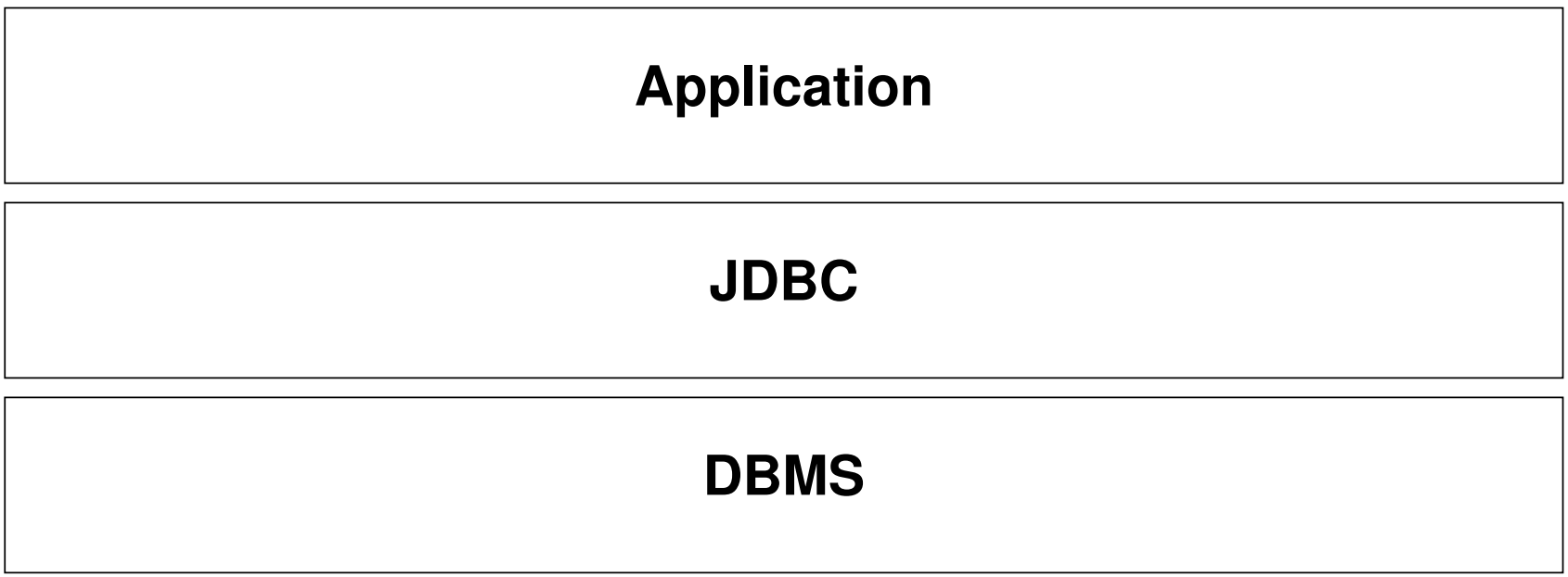
Was ist JDBC?

- JDBC = Java DataBase Connectivity
- Java API for access to SQL databases (SQL = Structured Query Language)
- Inspired by ODBC (Open DataBase Connectivity)
- Accesses database through JDBC drivers
  - JDBC methods transmit SQL commands as strings to a database
    ```
    "select name, lastname, salary from
    personnel where lastname = 'Smith'"
    ```
  - Database performs SQL commands and delivers the requested data and success or failure messages
  - JDBC methods read data delivered from the database

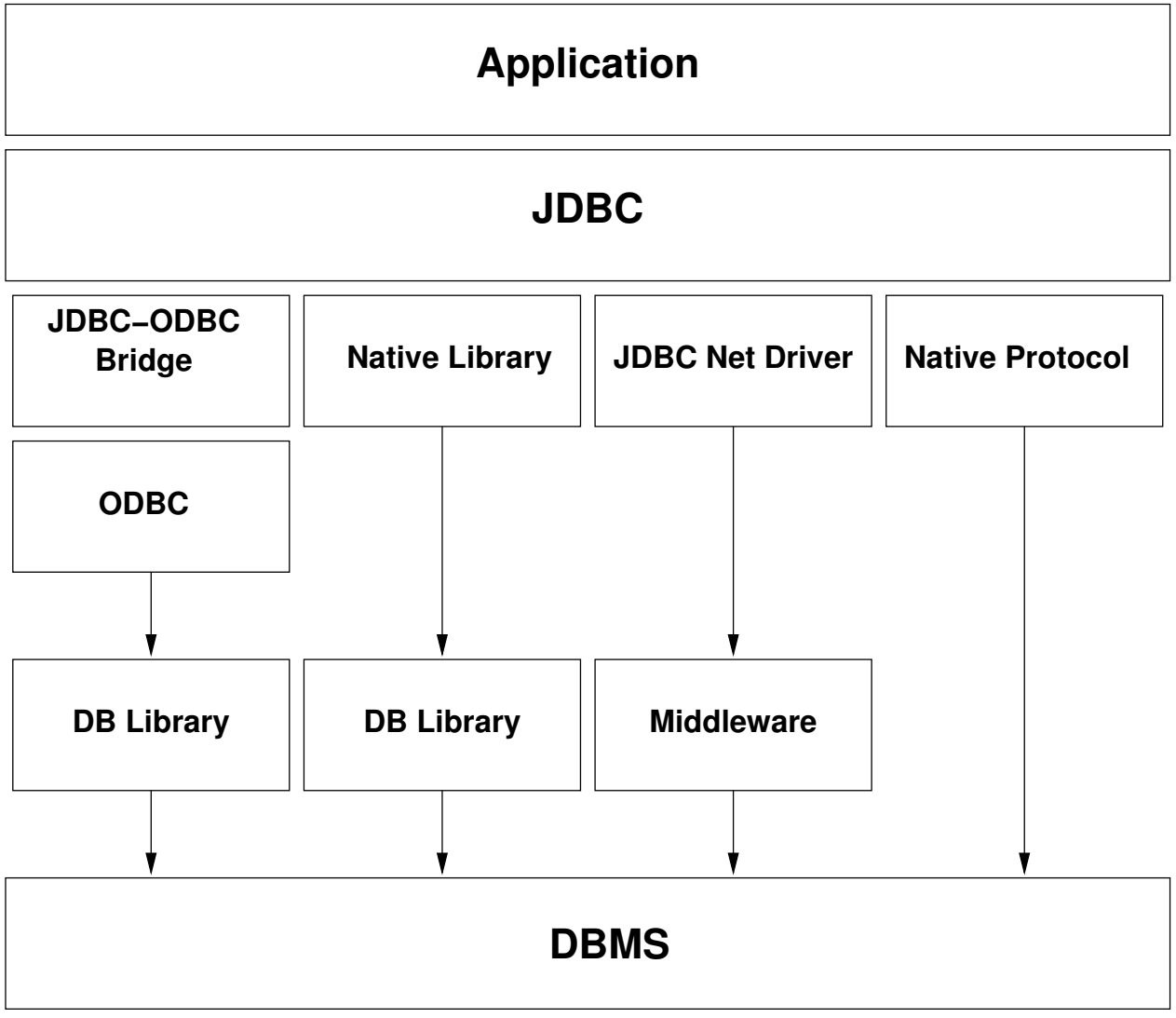
JDBC Features

- Query, insert, and update database content
- Support for
  - transactions
  - cursors
  - prepared statements and stored procedures
  - metadata (*e.g.*, column names)

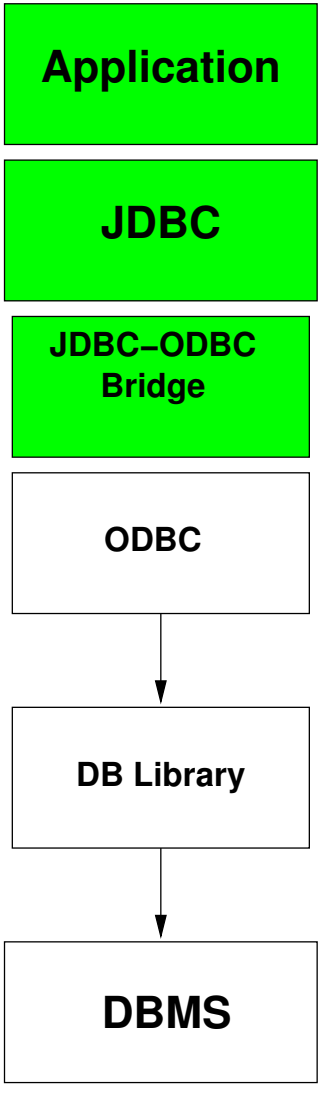
JDBC Application



Four Types of Drivers



Type 1 Drivers



Type 1 drivers implement the JDBC API as a mapping to another data access API, such as ODBC. Drivers of this type are generally dependent on a native library, which limits their portability. The JDBC-ODBC Bridge driver is an example of a Type 1 driver.

From JDBC API Specification

- JDBC-ODBC Bridge

Properties of Type 1 Drivers

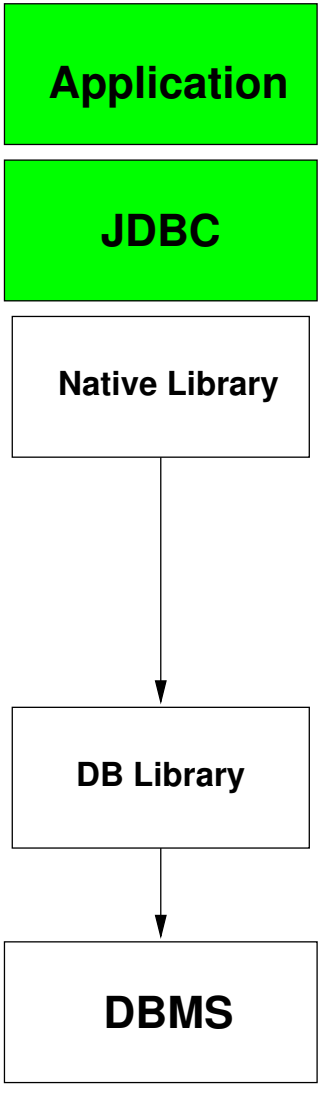
Advantages

- The same Type 1 JDBC driver can communicate with every database system for which an ODBC driver is available.
- Very simple driver.

Disadvantages

- Platform dependent (Microsoft)
  - ODBC must be installed
  - DB must support ODBC driver
- Use discouraged if pure Java driver available

Type 2 Drivers



Type 2 drivers are written partly in the Java programming language and partly in native code. These drivers use a native client library specific to the data source to which they connect. Again, because of the native code, their portability is limited.

From JDBC API Specification

- Native API driver

Properties of Type 2 Drivers

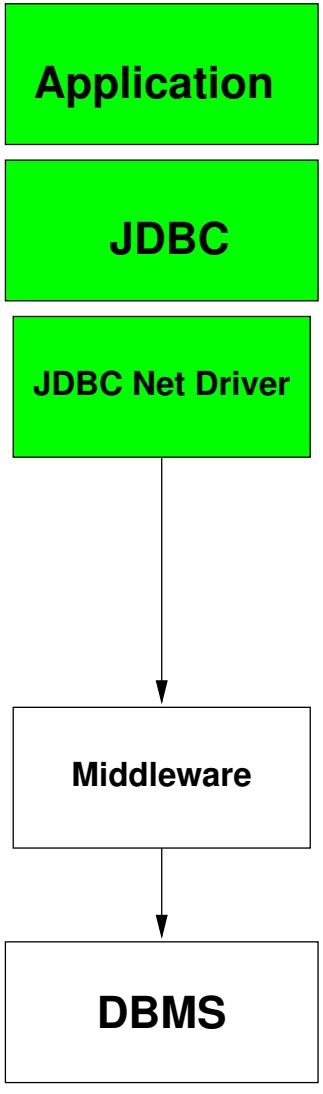
Advantages

- More functionality: tailored to the features of the database by vendor
- Better performance: avoids overhead of ODBC

Disadvantages

- Dependent on native code that makes the final database calls
- Tied to a specific operating system/architecture
- Reduced interoperability (compared to Type 4)

Type 3 Drivers



Type 3 drivers use a pure Java client and communicate with a middleware server using a database-independent protocol. The middleware server then communicates the client's requests to the data source.

From JDBC API Specification

- Network protocol driver

Properties of Type 3 Drivers

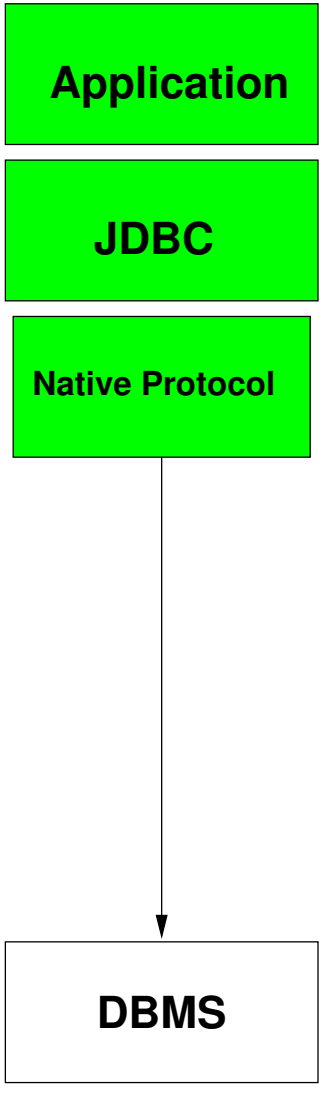
Advantages

- Platform independence through use of Java
- Database-independent communication protocol $\Rightarrow$ can be used with many different databases
- Middleware (application server) offers additional security and flexibility

Disadvantages

- Dependent on particular middleware implementation
- Restricted to databases supported by middleware

Type 4 Drivers



Type 4 drivers are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source.

From JDBC API Specification

- Native Protocol Driver

Properties of Type 4 Drivers

Advantages

- No system dependent drivers or libraries
- Better performance than Type 1 and Type 2 drivers by avoiding call conversion
- Better performance than Type 3 by avoiding protocol conversion

Disadvantages

- Driver specific to particular database
- Migration to different database requires new driver
- Not generally offered by DB vendors

Avoid Type 1 and Type 3 if possible

Reasons for using a Type 1 driver

- No Type 2 or Type 4 driver exists for the DBMS
- Application has a specific target platform

Reasons for using a Type 3 driver

- No Type 4 driver exists for the DBMS

Choose between Type 2 and Type 4

- Choose Type 4 if portability is critical
- For server applications, Type 2
- Choose Type 4 if it is known to be more efficient

Verwendung von JDBC

- Laden des JDBC-Treibers
- Aufbau von Verbindungen zur Datenbank
- Absetzen von SQL-Statements und Verarbeitung der Ergebnisse
- Schließen von Verbindungen zur Datenbank

Programmmuster

```
// load JDBC-driver
Class.forName("oracle.jdbc.driver.OracleDriver");

// create connection
String url = "jdbc:oracle:thin:@131.159.30.26:1521:dbprak";
Connection conn = DriverManager.getConnection (url, "name", "passwd");
conn.setAutoCommit(true);

// execute statement
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM personnel");
// Ergebnisse verarbeiten
rs.close();

// commit
conn.commit();

// close connection
conn.close();
```

Methoden von java.sql.Connection

Isolationslevel

```
setTransactionIsolation(int level);
```

Savepoint-Unterstützung

- `rollback(Savepoint savePoint);`
- `Savepoint setSavepoint();`
- `releaseSavepoint(Savepoint savePoint);`

Commits/Rollback

- `setAutoCommit(boolean autoCommit);`
- `commit();`
- `rollback();`

Transaktionsunterstützung

TRANSACTION_NONE

no transaction support

TRANSACTION_READ_UNCOMMITTED

allows a row changed by one transaction to be read by another transaction before any changes in that row have been committed (a "dirty read"). If any of the changes are rolled back, the second transaction will have retrieved an invalid row.

TRANSACTION_READ_COMMITTED

prohibits a transaction from reading a row with uncommitted changes in it.

Transaktionsunterstützung/2

TRANSACTION_REPEATABLE_READ

prohibits a transaction from reading a row with uncommitted changes in it, and it also prohibits the situation where one transaction reads a row, a second transaction alters the row, and the first transaction rereads the row, getting different values the second time (a "nonrepeatable read").

TRANSACTION_SERIALIZABLE

enforces TRANSACTION_REPEATABLE_READ and further prohibits the situation where one transaction reads all rows that satisfy a WHERE condition, a second transaction inserts a row that satisfies that WHERE condition, and the first transaction rereads for the same condition, retrieving the additional "phantom" row in the second read.

Dirty Read

```
; transaction #1

UPDATE product
  SET unit_price =
      unit_price + 95
  WHERE
    NAME = 'T Shirt';


ROLLBACK;
UPDATE product
  SET unit_price =
      unit_price + 0.95
  WHERE
    NAME = 'T Shirt';
```

```
transaction #2


SELECT SUM( quantity * unit_price )
  FROM product;
```

Non-repeatable Read

```
; transaction #1

SELECT unit_price
  FROM product
  WHERE
    NAME = 'T Shirt';
```

```
transaction #2

UPDATE product
  SET unit_price = 17.50
  WHERE NAME = 'T Shirt';
COMMIT;
```

```
SELECT unit_price
  FROM product
  WHERE
    NAME = 'T Shirt';
```

Phantom Row

```
; transaction #1

SELECT * FROM department
  ORDER BY dept_id;



SELECT * FROM department
  ORDER BY dept_id;
```

```
transaction #2

INSERT INTO department
  (dept_id, dept_name, dept_head_id)
   VALUES(600, 'Foreign Sales', 129);
COMMIT;
```

Guarantees of Transaction Levels

TRANSACTION_...	dirty read	non-repeatable read	phantom read
NONE	yes	yes	yes
READ_UNCOMMITTED	yes	yes	yes
READ_COMMITTED	no	yes	yes
REPEATABLE_READ	no	no	yes
SERIALIZABLE	no	no	no

Simple Statements

```
conn = ds.getConnection();
stmt = conn.createStatement();

// for select statements
if (stmt.execute("SELECT * FROM product WHERE NAME = 'T Shirt'"))
  ResultSet rs = stmt.getResultSet();

ResultSet rs =
  executeQuery("SELECT * FROM product WHERE NAME = '" +name+ "'");

// for insert and update
int nRows = stmt.executeUpdate(
  "UPDATE product SET unit_price = " +
  "  unit_price + " + newPrice +
  " WHERE NAME = 'T Shirt'");
```

java.sql.ResultSet

- `ResultSet` contains result of query as a table
- processing using a cursor
- `rs.next();` // advance to next row
- `rs.previous();` // previous row (don't use)
- `rs.absolute(5);` // absolute row position
- `rs.relative(5);` // relative positioning
- `rs.first();`
- `rs.last();`
- changes through result set (if so configured)
  - `rs.deleteRow();`
  - `rs.updateString("name", "Turing");`
  - `rs.updateRow();` // flush update

java.sql.ResultSet

- read from result set
  - `rs.getString(5)` // get column 5
  - `rs.getString("unit_price")` // get column by name (don't use)
  - `rs.getLong(5)` // retrieve a long
  - etc ...
- insert in result set (if so configured)
  - `rs.moveToInsertRow();` // special row, remembers current row
  - `rs.updateString(1, "Turing");`
  - `rs.updateInt(2, 35);`
  - `rs.updateBoolean(3, true);`
  - `rs.insertRow();` // insert in DB
  - `rs.moveToCurrentRow();` // pop back to current row

Code Example

```
Statement stmt = dbcon.createStatement();
ResultSet result = stmt.executeQuery("SELECT * FROM books");
while (result.next()) {
  int id = result.getInt(1);                  // getInt("ID")
  String title = result.getString("Title"); // getString(2)
  String subtitle = result.getString(3);    // getString("subtitle")
  String author = result.getString(4);      // getString("author")
  int year = result.getInt("Year");           // getInt(5)
}
```

java.sql.PreparedStatement

- defines statement template
- using ? as placeholder
- sent to the DBMS for precompilation
- can be instantiated and executed with different values

```
PreparedStatement ps =
  conn.prepareStatement(
    "UPDATE employees SET salary = ? WHERE id = ?");
ps.setBigDecimal(1, 153833.00);
ps.setInt(2, 110592);
ps.execute();
```

Metadata

- `DatabaseMetaData dbmd = conn.getMetaData();`
  - DB version
  - specific features
  - infos about schemas, tables, keys, etc
- `ResultSetMetaData metaData = rs.getMetaData();`
  - # columns, column names, column types
  - potentially inefficient: entire result set may have to be constructed before meta data is available

Portability of Applications

- Modern application programs, especially Java programs, should be portable.
- Sun marketing slogan: WORA (Write once, run anywhere)
- Portable JDBC programs only with of Type 3 and Type 4 drivers.
- They do not contain computer or operating system-specific code.
- JDBC supplies a unified interface.
- SQL is a standardized query language for databases.

But:

- Not every database system supports all JDBC functions.
- Database manufactures maintain own SQL dialect with further functions and commands that cannot be used any more on other database systems, in case of migration.

Portability of JDBC Applications

- Theoretically only exchange of JDBC drivers is required (1 File).
- In practice: manual checking if the new DBMS understands the SQL dialect used.
- The parts of the program, in which non-standard SQL commands are sent, must be adapted to the new DBMS.
- Concentrate the SQL instructions and SQL generation in a few, central classes, if possible.
- Avoid SQL language constructs that can only be processed by a specific database system.
- Avoid database-specific SQL data types in table definition.
- Use SQL Data types for which equivalent types exist in other database systems.
- Perform customization via properties.