

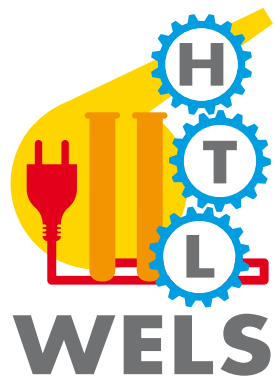
JDBC



JDBC

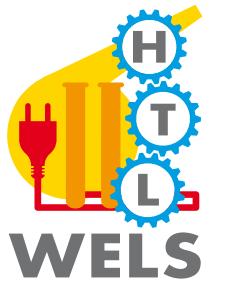
SEW 4

DI Thomas Helml





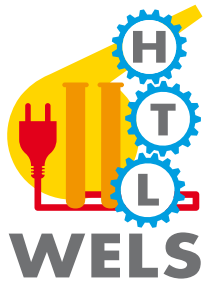
INHALTSVERZEICHNIS



- Begriffsdefinition
- Anwendungsarchitektur
- JDBC Treiber
- JDBC API Überblick
- JDBC Grundgerüst
- JDBC Datentypen
- Transaktionen
- PreparedStatement
- Blobs



BEGRIFFSDEFINITION

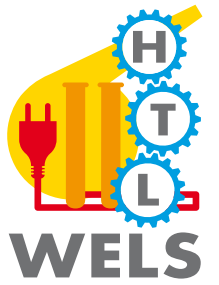




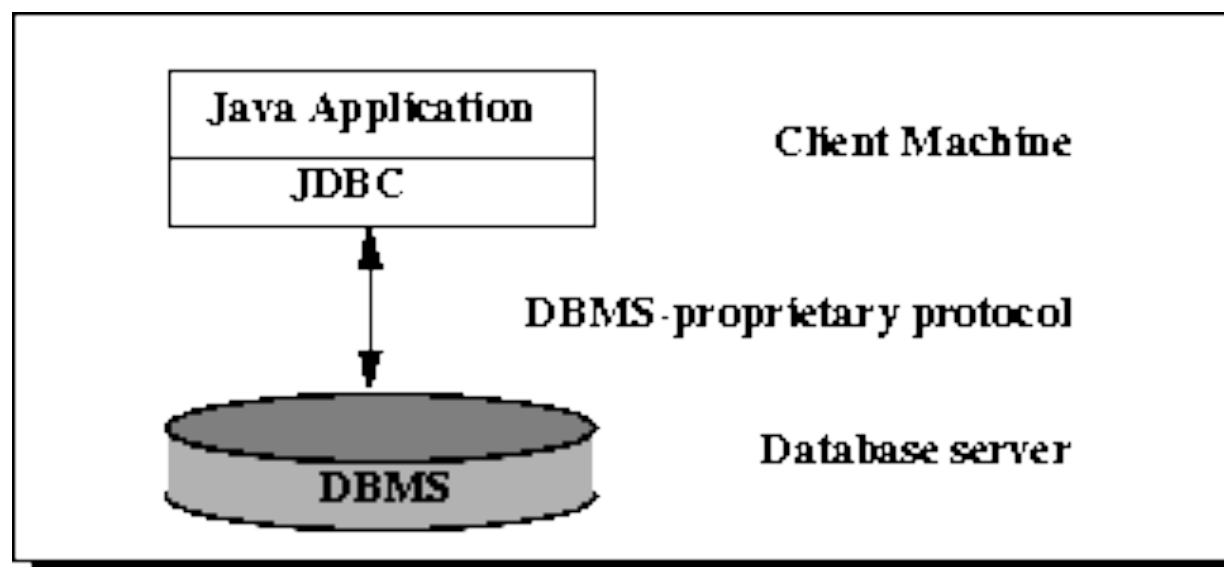
- Java DataBase Connectivity
 - Standard-Schnittstelle für Zugriff auf relationale DB mittels SQL und Java
 - Sammlung von Klassen und Interfaces
 - (Package: `java.sql`)
 - wird JDBC verwendet: kein DB-spezifischer Code im Programmen
 - Abstraktionsschicht zw. Java und SQL



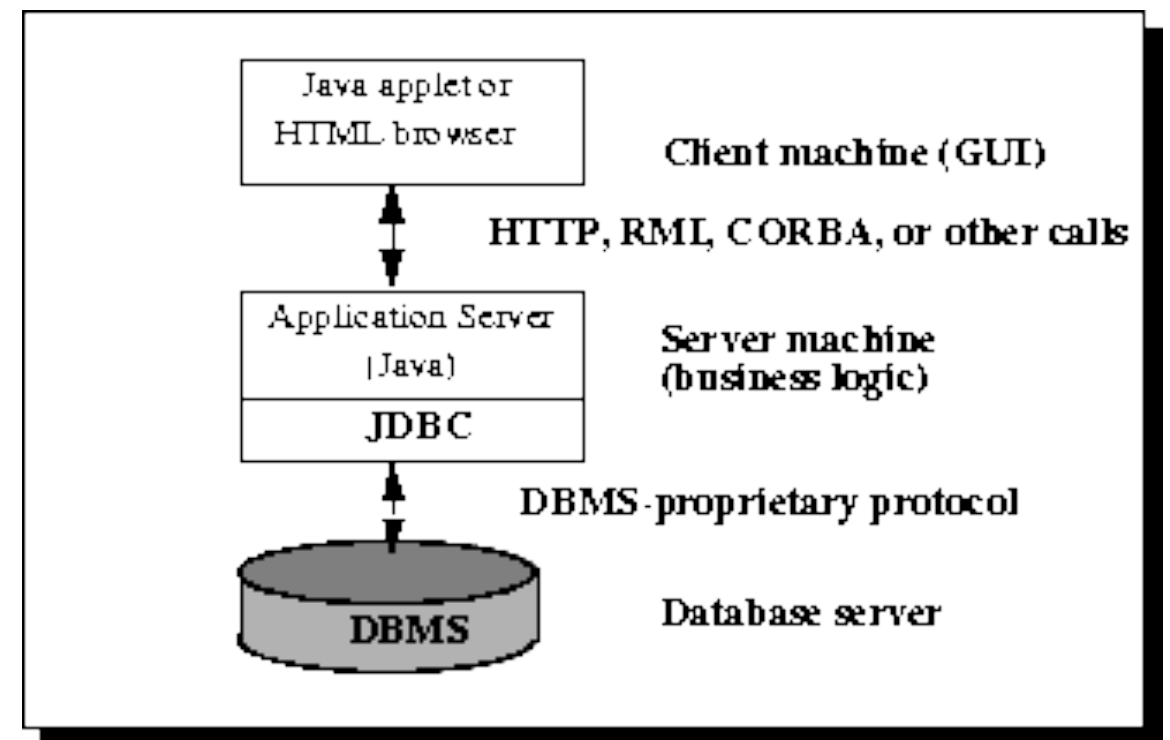
ANWENDUNGSARCHITEKTUR



- 2-stufige Architektur
 - Client Programm greift direkt auf DB zu (Netzwerk oder lokal)

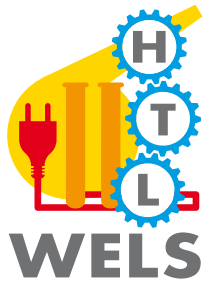


- 3-stufig:
- Trennung: Anwendungslogik und Benutzeroberfläche bzw. Datenverwaltung
- Client kommuniziert mit Applicationserver, der greift auf Datenbank zu



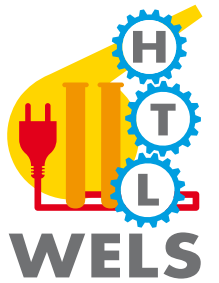


JDBC TREIBER

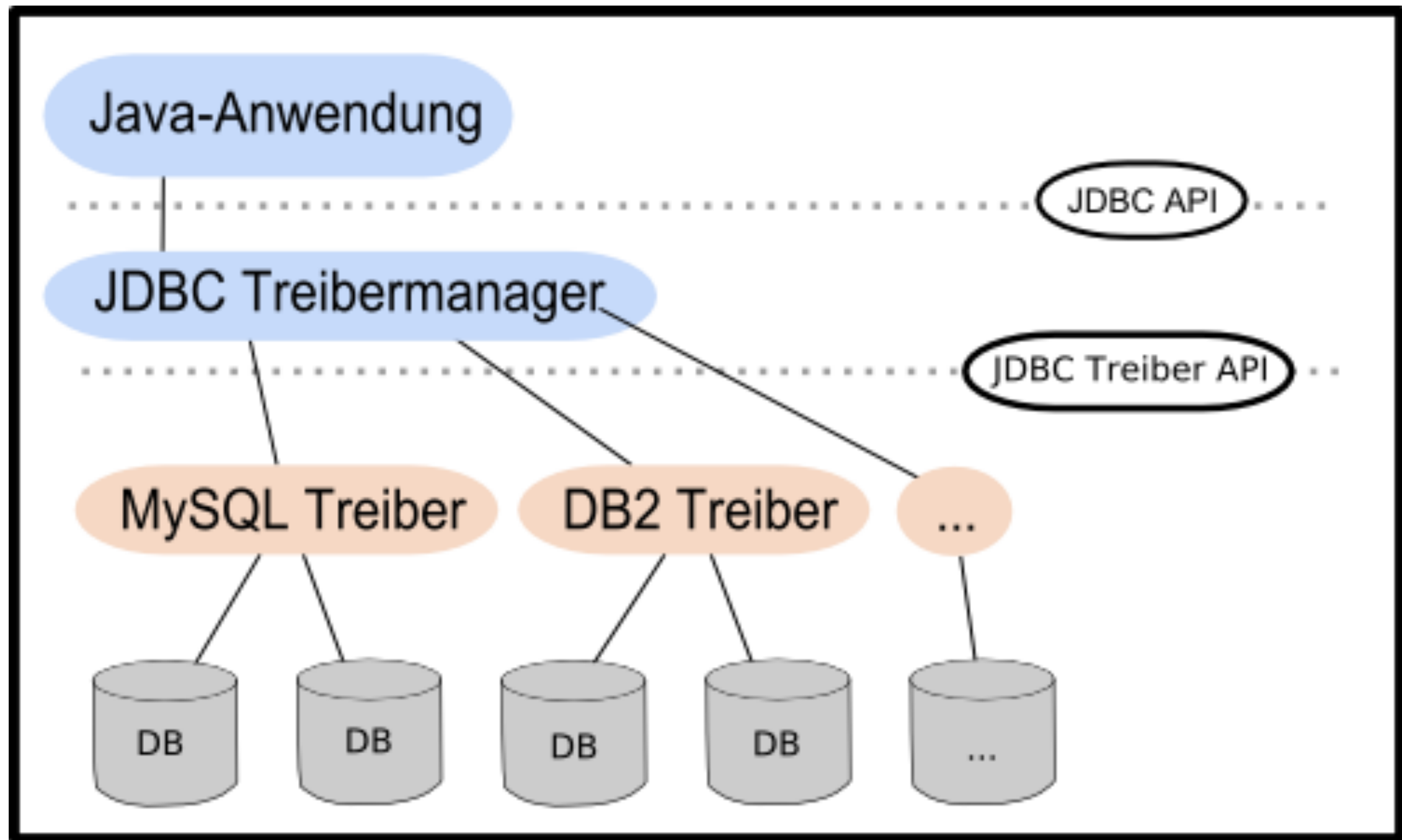




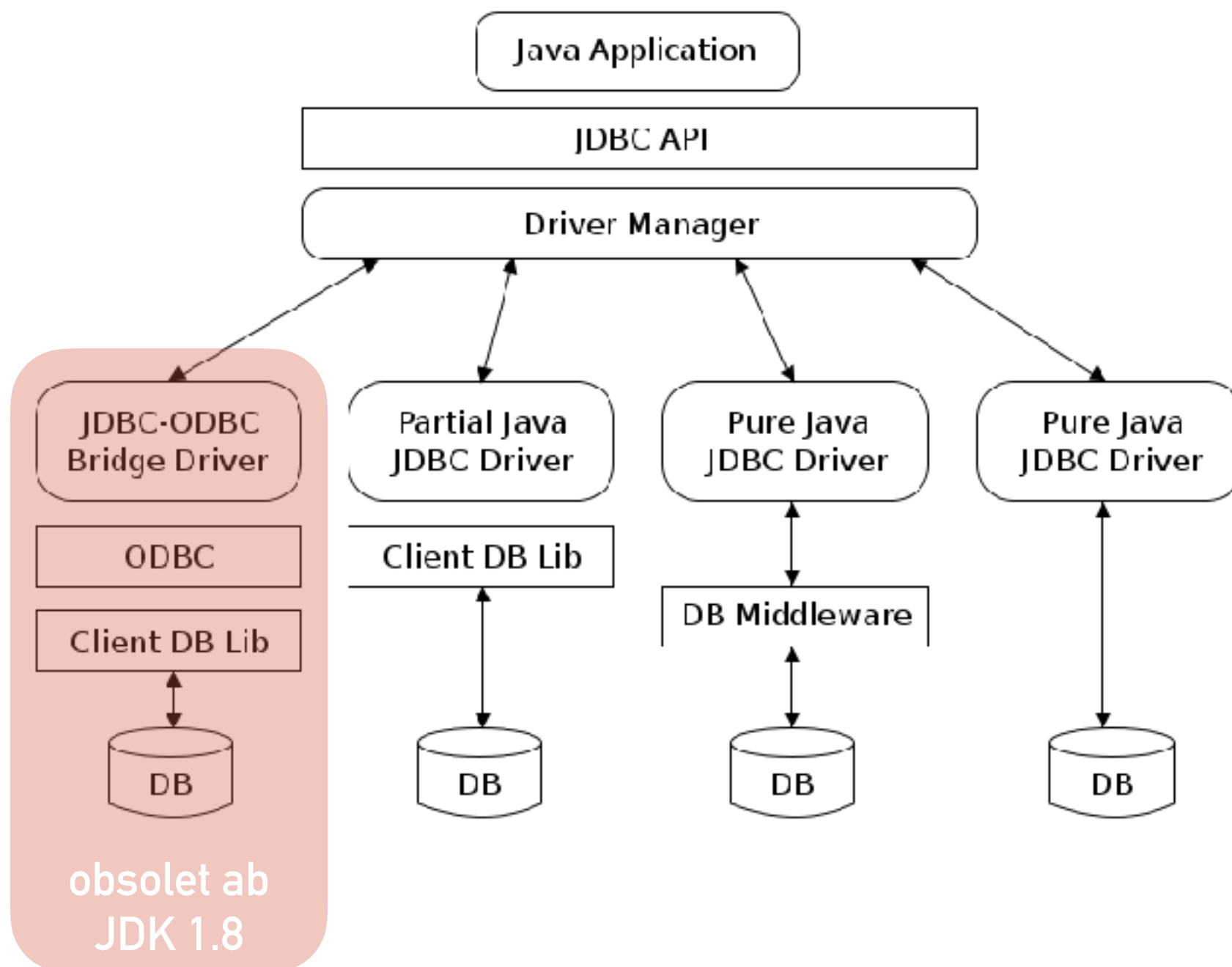
JDBC TREIBER



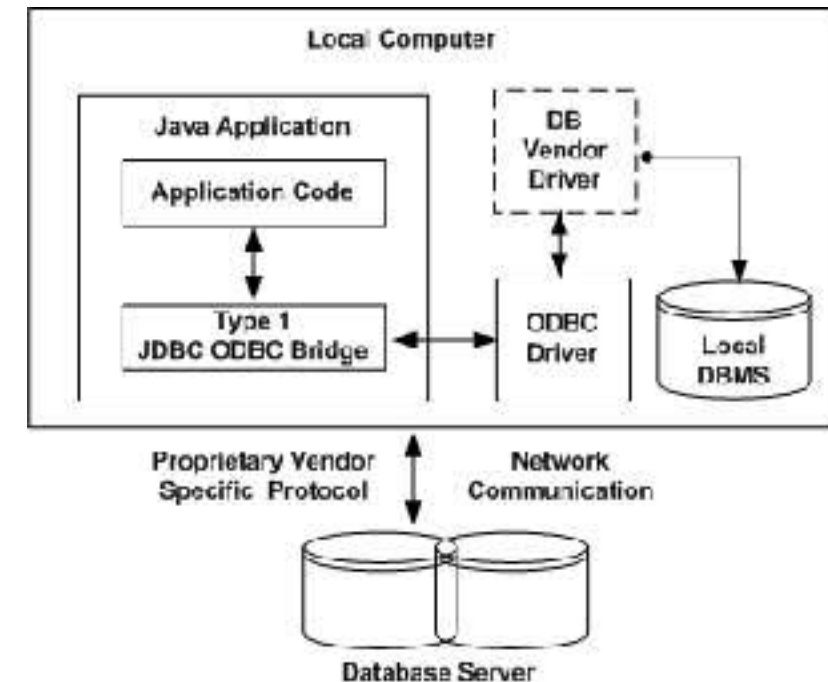
- jede DB hat herstellerabhängige Zugriffsschnittstelle
- JDBC-Treiber dient als Übersetzer auf diese Schnittstelle
- für jedes DBMS wird eigener Treiber benötigt



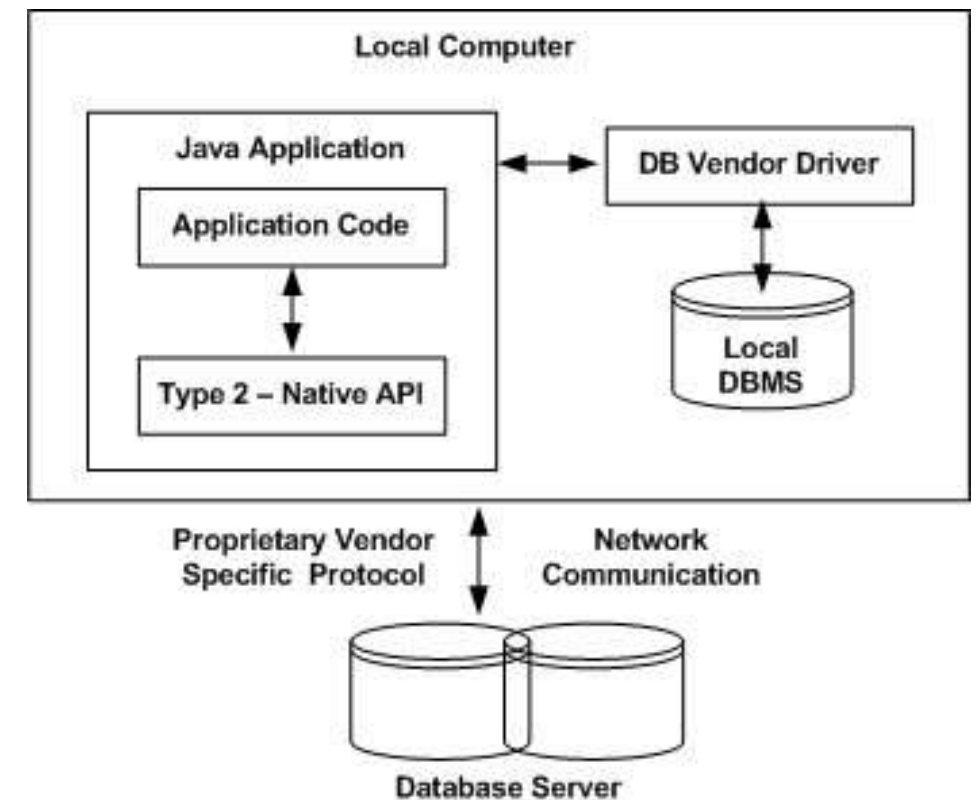
➤ 4(3) Typen von Treiber:



- Typ1: JDBC-ODBC-Bridge Treiber
 - Treiber benutzen das ODBC (Open Database Connectivity) von Microsoft
 - Konvertierung von JDBC auf ODBC
 - ODBC Treiber müssen extra installiert werden
 - Paket: `oracle.jdbc.odbc`
 - ab Java 8 entfernt

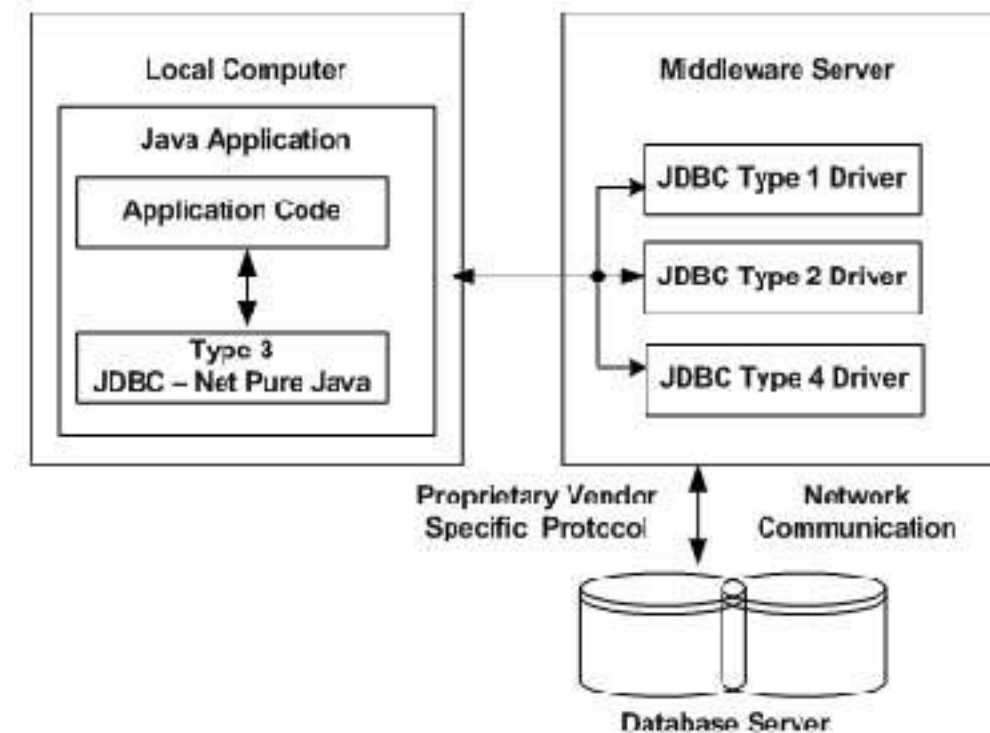


- Typ2: Native API Treiber
 - Implementieren herstellerabhängige DB-Protokoll
 - 2 Bestandteile:
 - Java
 - plattformabhängiger Code (z.B. DLLs)
 - Anwendungen sind nur bedingt portierbar

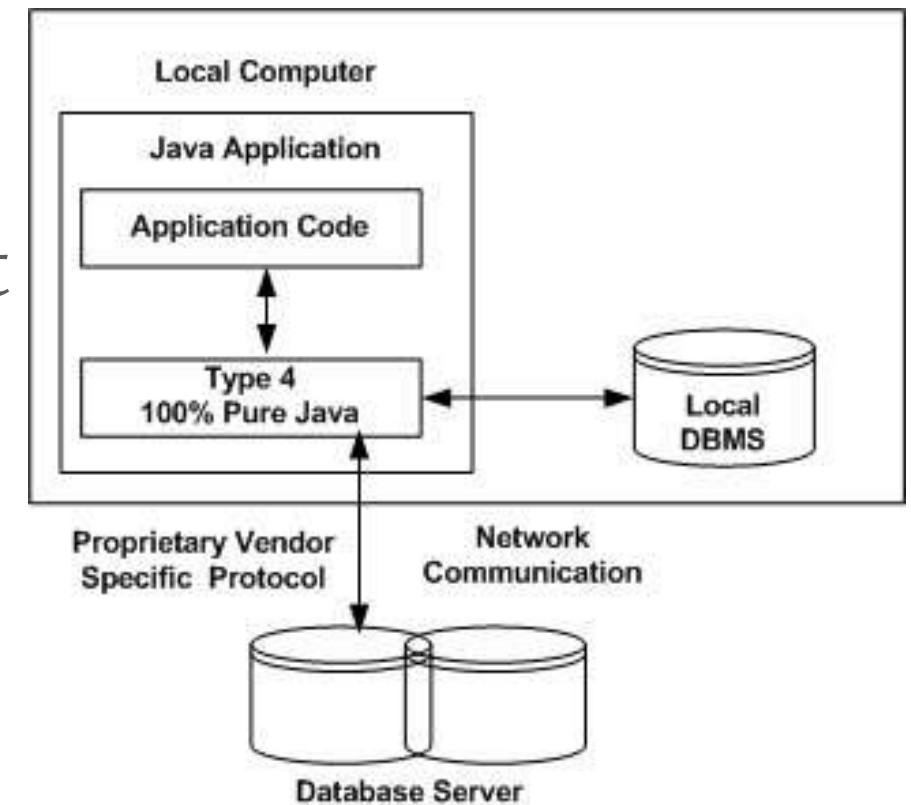


➤ Typ3: JDBC-Net Treiber

- komplett in Java realisiert
- Zwischenschicht (Middleware) übernimmt Übersetzung von JDBC auf herstellerabhängiges DB-Protokoll
- flexible Lösung: Wechsel des DBMS wird von Anwendung nicht bemerkt

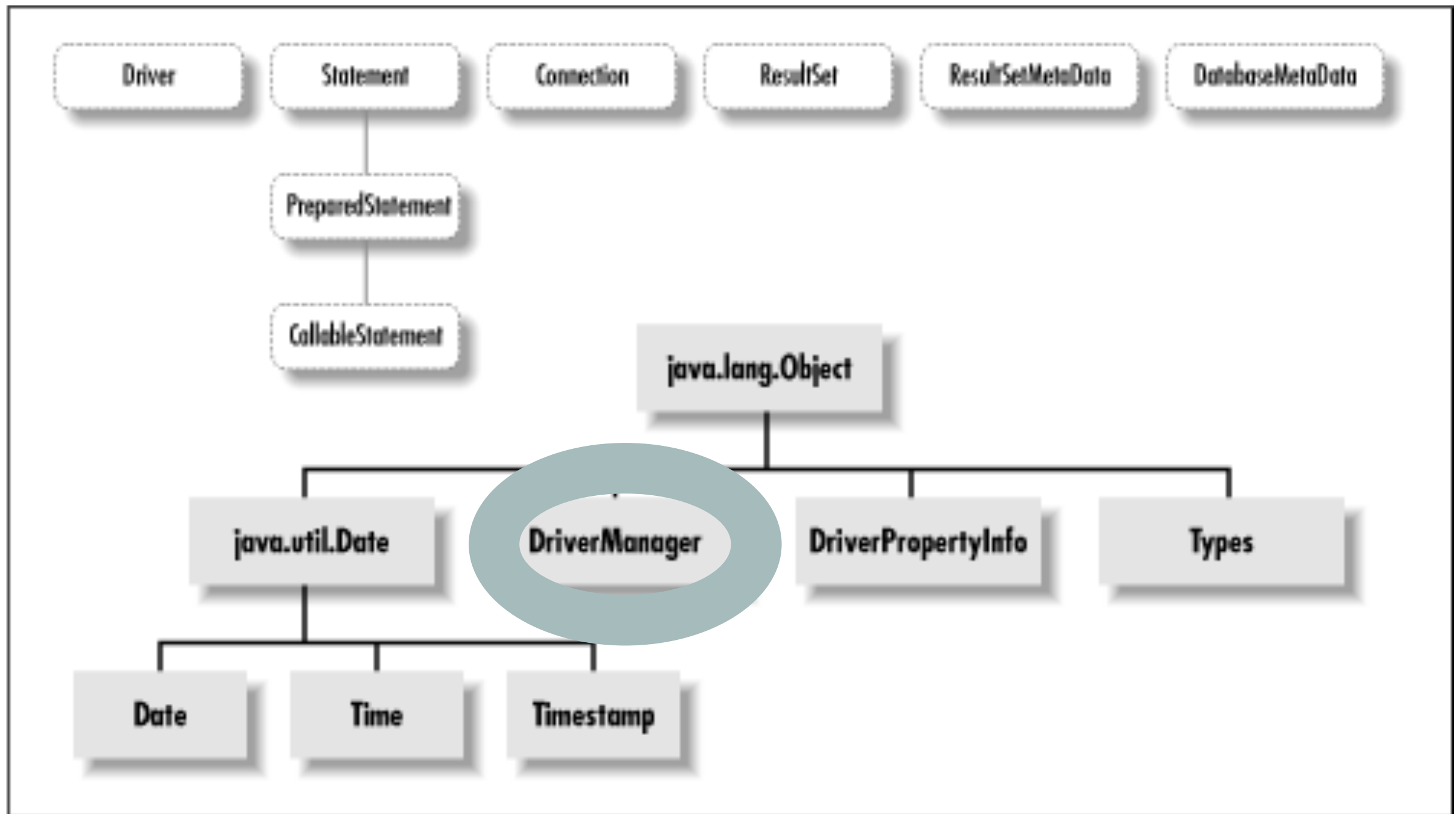
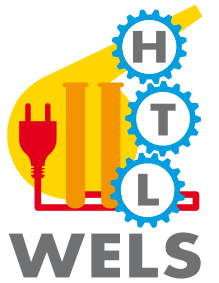


- Typ4: Native-Protocol Treiber
 - Treiber komplett in Java
 - Implementieren DB Protokoll direkt
 - DBMS Hersteller liefern diese Treiber



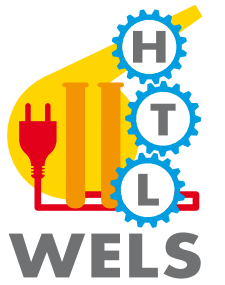


JDBC API



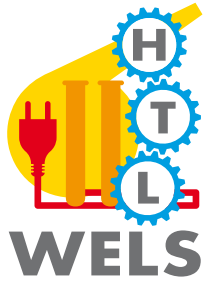


JDBC API





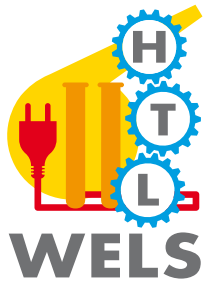
JDBC API



- Klassen:
 - DriverManager:
 - Laden des JDBC-Treibers
 - Aufbau der Datenbankverbindung
 - SQLException:
 - Behandlung im Fehlerfall



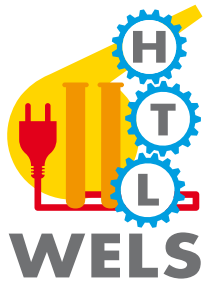
JDBC API



- Interfaces:
 - **Connection**: repräsentiert eine DB-Verbindung
 - **Statement**: führt SQL Anweisungen über die DB-Verbindung aus
 - **ResultSet**: Methoden, um auf das Ergebnis der SQL-Abfrage zuzugreifen

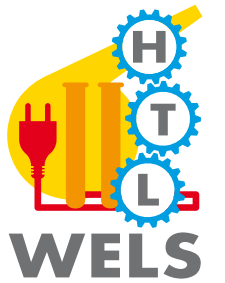


JDBC GRUNDGERÜST





JDBC GRUNDGERÜST



➤ Schritt 1: Treiber laden (ab JDK 1.6. überflüssig!)

```
try {  
    Class.forName( "org.hsqldb.jdbcDriver" );  
}  
  
catch ( ClassNotFoundException e ) {  
    System.err.println( "Keine Treiber-Klasse!" );  
    return;  
}
```

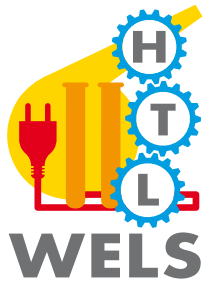
ab 1.6:

Datei: META-INF/services/java.sql.Driver

Inhalt: Name der Treiberklasse



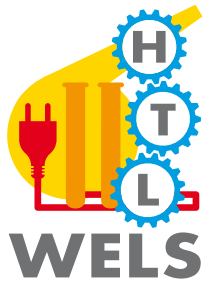
JDBC GRUNDGERÜST



- Schritt 2: Verbindungsaufbau
 - externes Property File für Verbindungsdaten
 - „dbconnect.properties“
 - Vorteil: DBMS kann gewechselt werden kann, ohne Programmänderung



JDBC GRUNDGERÜST



dbconnect.properties:

`driver=org.hsqldb.jdbcDriver`

`url=jdbc:hsqldb:file:tutego`

`username=sa`

`password=`

`#Verbindungsdaten für MySQL`

`#driver=com.mysql.jdbc.Driver`

`#url=jdbc:mysql://localhost:3306/tutego`

`#username=pc`

`#password=pc`

Datenbanken werden über URL exakt identifiziert

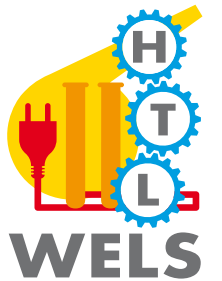
`jdbc:<subprotokoll>:<subname>`

Subprotokoll: Art des verwendeten Treibers (z.B. odbc, mysql, ...)

Subname: ist die eigentliche Datenbank



JDBC GRUNDGERÜST

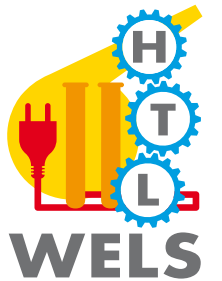


➤ Code zum Laden der Eigenschaften aus Property-File:

```
try (FileInputStream in = new FileInputStream("dbconnect.properties");) {  
    Properties prop = new Properties();  
    // Properties laden  
    prop.load(in);  
  
    String driver = prop.getProperty("driver");  
    String url = prop.getProperty("url");  
    String user = prop.getProperty("user");  
    String pwd = prop.getProperty("pwd");  
}  
catch ...
```




JDBC GRUNDGERÜST



➤ Datenbankverbindung aufbauen:

```
Connection con = DriverManager.getConnection(url, user, pwd);
```

url am Beispiel HSQL:

```
jdbc:hsqldb:file:<Dateipfad>;user=SA;password=;  
ifexists=true;shutdown=true
```

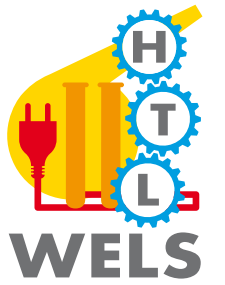
ifexists: nur, wenn bestehende DB verbinden

- **Zugriff auf DB** erfolgt dann über **Connection-Objekt con!!**
- Infos über DBMS abfragen:

```
DatabaseMetaData getMetaData() throws SQLException
```



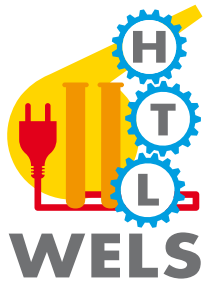
ÜBUNG



➤ 401_DB_Properties

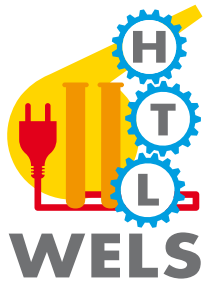


HYPERSQL DATABASE



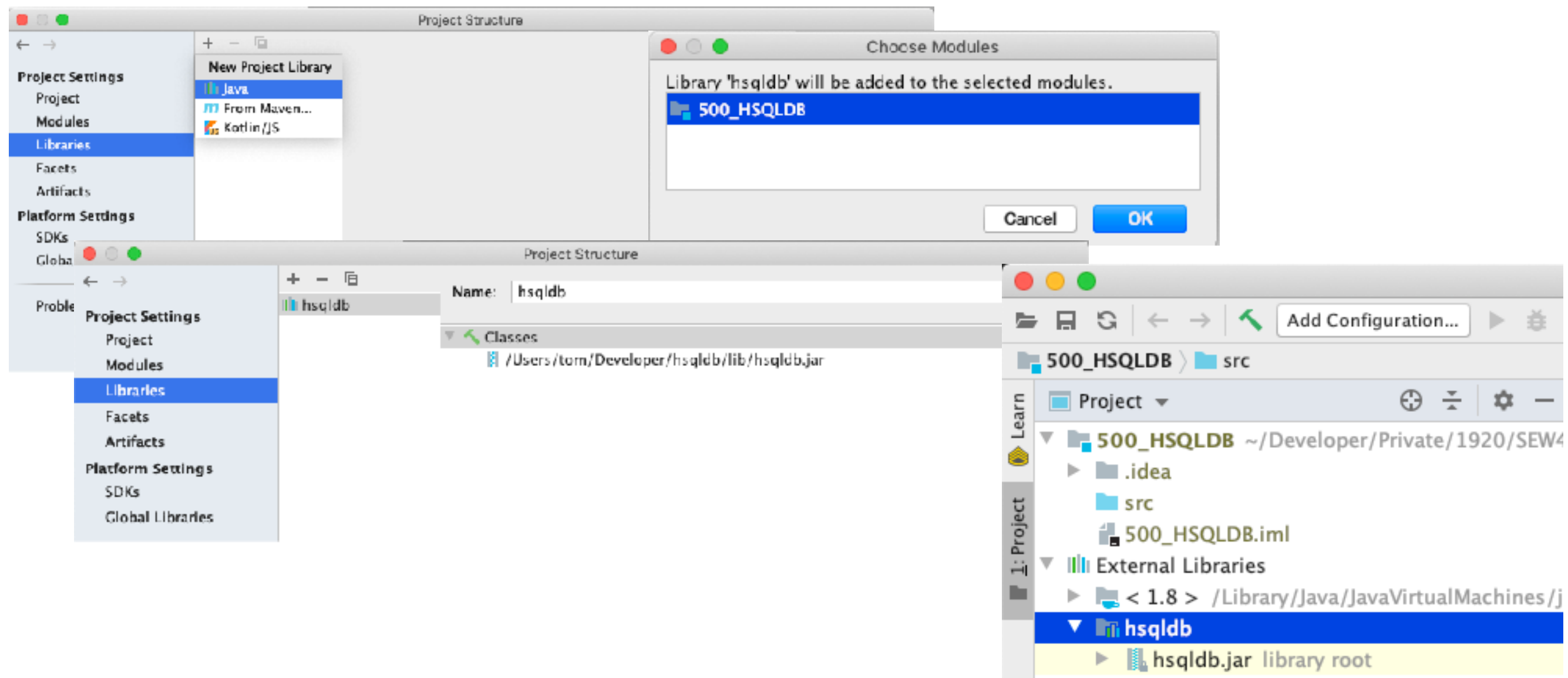


HYPERSQL DATABASE



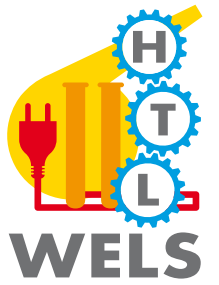
- Lade hsqldb herunter:
 - `http://hsqldb.org/`
- HSQL DB Manager starten:
 - `bin/runManagerSwing.bat`
- Erstelle Testdaten in der Datenbank:
 - *Options - Insert Test Data*

- Hsqldb in IntelliJ einbinden:
 - Projekt anlegen - Project Structure - Settings - Libraries +
 - hsqldb/lib/hsqldb.jar



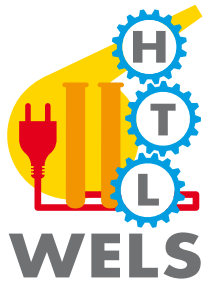


JDBC DATENTYPEN





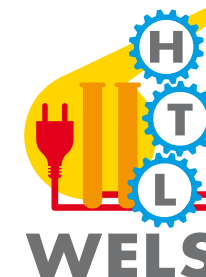
JDBC DATENTYPEN



- SQL arbeitet mit anderen Datentypen als Java
 - darum wurden eigene JDBC-Datentypen zur Typkonvertierung definiert
 - diese findet man unter: `java.sql.Types`



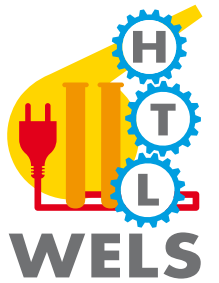
JDBC DATENTYPEN



JDBC Typ	Java Typ	Java Object-Typ
TINYINT	byte	java.lang.Byte
SMALLINT	short	java.lang.Short
INTEGER	int	java.lang.Integer
BIGINT	long	java.lang.Long
REAL	double	java.lang.Double
FLOAT	double	java.lang.Double
DOUBLE	double	java.lang.Double



JDBC DATENTYPEN



JDBC Typ	Java Typ	Java Object-Typ
DECIMAL	<code>java.math.BigDecimal</code>	<code>java.math.BigDecimal</code>
NUMERIC	<code>java.math.BigDecimal</code>	<code>java.math.BigDecimal</code>
DATE	<code>java.sql.Date</code>	<code>java.sql.Date</code>
TIME	<code>java.sql.Time</code>	<code>java.sql.Time</code>
TIMESTAMP	<code>java.sql.Timestamp</code>	<code>java.sql.Timestamp</code>



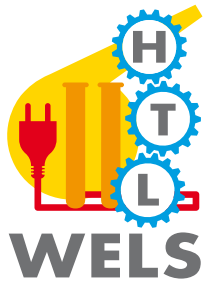
JDBC DATENTYPEN



JDBC Typ	Java Typ	Java Object-Typ
CHAR	<code>java.lang.String</code>	<code>java.lang.String</code>
VARCHAR	<code>java.lang.String</code>	<code>java.lang.String</code>
LONGVARCHAR	<code>java.lang.String</code>	<code>java.lang.String</code>
BIT	<code>boolean</code>	<code>java.lang.Boolean</code>
BINARY	<code>byte[]</code>	<code>byte[]</code>
VARBINARY	<code>byte[]</code>	<code>byte[]</code>
LONGVARBINARY	<code>byte[]</code>	<code>byte[]</code>



LOCALDATETIME



➤ Konvertierung Java <—> SQL

```
LocalDateTime ldt =
```

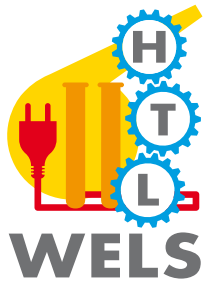
```
    LocalDateTime.parse("2019-10-31T17:42:16",  
        DateTimeFormatter.ISO_LOCAL_DATE_TIME);
```

```
Timestamp ts = Timestamp.valueOf(ldt);
```

```
ldt = ts.toLocalDateTime();
```



LOCALDATE



➤ Konvertierung Java <—> SQL

```
LocalDate ld =
```

```
    LocalDate.parse("2019-10-31T17:42:16",  
        DateTimeFormatter.ISO_LOCAL_DATE);
```

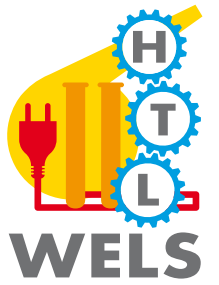
```
// Achtung: java.sql.Date!!
```

```
Date d = Date.valueOf(ld);
```

```
ld = d.toLocalDate();
```



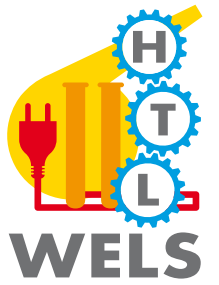
ÜBUNG



➤ 402_SQL_Types

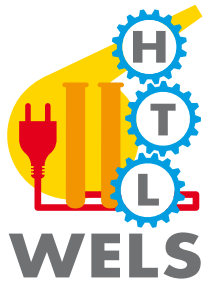


JDBC GRUNDGERÜST





JDBC GRUNDGERÜST



- Basisschnittstelle für alle SQL-Anweisungen:

```
java.sql.Statement
```

- Statement Object erzeugen:

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

- SQL-Anweisungen ausführen

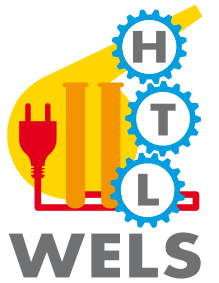
```
ResultSet rs = stmt.executeQuery("SELECT *  
FROM CUSTOMER");
```

- SQL-Anweisung enthält kein abschließendes Semikolon
(Treiber ergänzt es bei Bedarf)

- unterschiedlich von DBMS zu DBMS



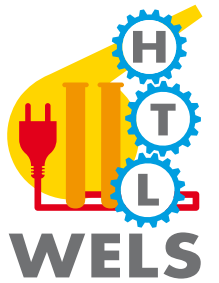
JDBC GRUNDGERÜST



- Ergebnis der SQL Abfrage (Tabelle): `ResultSet`
- Ergebniscursor (=Position in Ergebnismenge)
 - Zeilenweise vorwärts mittels `.next()`
 - Anfang: `.first()`
 - Ende: `.last()`
- Spaltenzugriff über `getXXX()` Methoden
 - Spaltenindex bzw. Spaltenname als Parameter



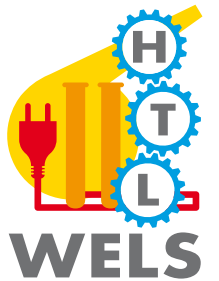
JDBC GRUNDGERÜST



- Entsprechend den Java-Typen gibt es passende getter-Methoden in der Klasse `ResultSet`:
 - `byte getByte(...)`
 - `short getShort(...)`
 - `int getInt(...)`
 - ...



JDBC GRUNDGERÜST

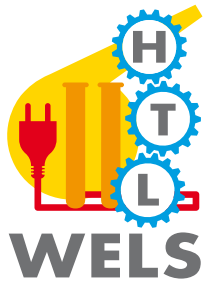


➤ Beispiel 1:

```
ResultSet rs = stmt.executeQuery( "SELECT * FROM  
    CUSTOMER" );  
while ( rs.next() )  
    System.out.printf( "%s, %s %s, %s %s \n",  
        rs.getString(1),  
        rs.getString(2),  
        rs.getString(3),  
        rs.getString(4),  
        rs.getString(5)  
    );
```



JDBC GRUNDGERÜST

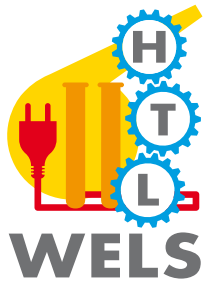


➤ Beispiel 2:

```
rs = stmt.executeQuery( "SELECT LASTNAME,  
                        CITY FROM Customer" );  
  
while ( rs.next() )  
    System.out.printf( "%s, %s  \n",  
        rs.getString(1),  
        rs.getString(2)  
    );
```



JDBC GRUNDGERÜST

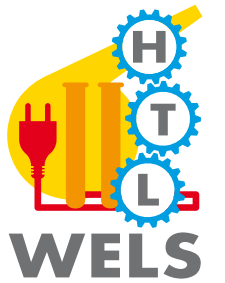


➤ Beispiel 3:

```
rs = stmt.executeQuery( "SELECT LASTNAME,  
                        CITY FROM Customer" );  
  
while ( rs.next() )  
    System.out.printf( "%s \n",  
                      rs.getString( "LASTNAME" ) );
```



JDBC GRUNDGERÜST



- Freigabe der DB Ressourcen
 - RecordSet
 - Statement
 - Connection
- mit `.close()` schließen (bzw. `try-with-resources`)!!



➤ Datenbankänderungen

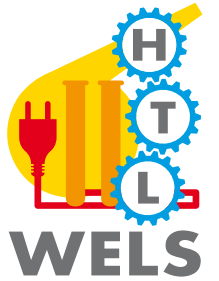
- Änderungen der DB-Struktur (z.B. CREATE TABLE)
- Änderungsbefehle (INSERT, UPDATE, DELETE)

```
int executeUpdate(String sql) throws  
SQLException
```

- Rückgabewert bei INSERT, UPDATE, DELETE
 - Anzahl der geänderten Datensätze



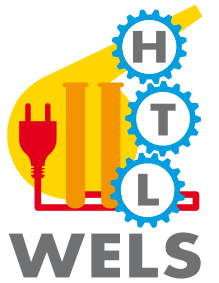
ÜBUNG



➤ 403_JDBC_Grundgeruest

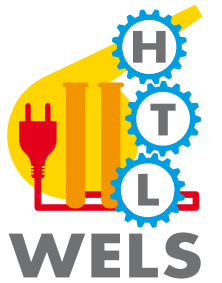


TRANSAKTIONEN





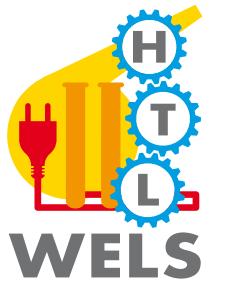
TRANSAKTIONEN



- Transaktion
 - Reihe von SQL Anweisungen
 - überführt DB von konsistenten in neuen konsistenten Zustand
- Alle Anweisungen werden entweder
 - Durchgeführt = commit
 - Abgebrochen und zurückgenommen = roll back



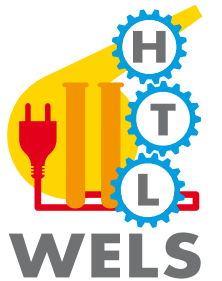
TRANSAKTIONEN



- Auto Commit Mode
 - jedes SQL Statement = eine einzelne Transaktion
 - => Default Einstellung
- Manueller Modus:
 - Transaktion wird gestartet, sobald vorige geschlossen wird (commit oder rollback)



TRANSAKTIONEN



Interface Connection

```
setAutoCommit (boolean autoCommit) throws SQLException
```

```
// autoCommit=true => AutoCommit ein
```

```
// autoCommit=false => AutoCommit aus
```

```
boolean getAutoCommit() throws SQLException
```

```
// Prüfung, ob Auto-Commit eingeschaltet ist
```

```
void commit() throws SQLException
```

```
// falls Auto-Commit ausgeschaltet ist
```

```
// Transaktion soll abgeschlossen, Änderungen in DB übernehmen
```

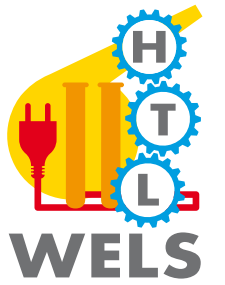
```
void rollback() throws SQLException
```

```
// falls Auto-Commit ausgeschaltet:
```

```
// Transaktion rückgängig machen
```



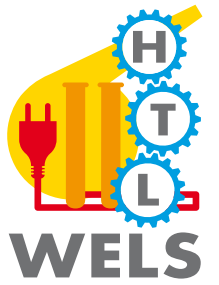
ÜBUNG



➤ 404_Transaktionen



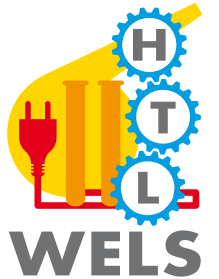
PREPAREDSTATEMENTS



- Man stelle sich eine Maske für Suchanfrage vor:
 - immer gleiche SQL-Anweisung im Hintergrund
 - lediglich Parameter ändert sich
- Beschleunigung durch **PreparedStatement**
 - DB parst SQL Statement
 - bereitet Statement für Durchführung vor (vorkompiliert)
 - nur mehr Parameter wird ausgetauscht



PREPAREDSTATEMENTS



`PreparedStatement prepareStatement (String sql)`
throws `SQLException`

- `sql` beinhaltet die SQL-Abfrage
- Parameter werden als Platzhalter „?“ im SQL String angegeben
- vor Ausführung, Parameter setzen:

```
void setByte (int idx, byte x)
```

```
void setShort (int idx, short x)
```

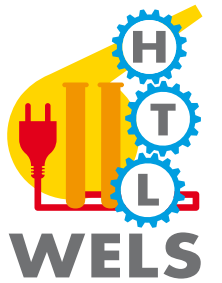
```
void setInt (...)
```

```
...
```

```
// idx von links mit 1 beginnend
```



PREPAREDSTATEMENTS



- ACHTUNG – **null** Werte

```
setNull(int idx, int sqlType)
```

- passender Typ aus `java.sql.Types` notwendig!

- Parameter löschen:

```
void clearParameters() throws SQLException
```

- Ausführung:

```
ResultSet executeQuery() throws SQLException
```

```
int executeUpdate() throws SQLException
```



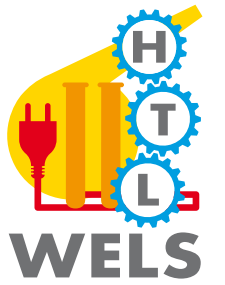
```

// Database related Code/Method using prepared SQL statement:
boolean isAutoCommit = conn.getAutoCommit();
try (PreparedStatement prepStmt = conn.prepareStatement(
    "INSERT INTO OBJ VALUES (?, ?, ?)")) {
    // clear recent statements from reused statements internal batch
    //if (canBatch) {
    //    prepStmt.clearBatch();
    //}
    // avoid auto commit following each execute*
    conn.setAutoCommit(false);
    // run through collection
    for (Object elem: coll) {
        // complete SQL statement parameters and batch/execute
        prepStmt.setInt(1, ...);
        ...
        if (canBatch) {
            prepStmt.addBatch();
        } else {
            prepStmt.executeUpdate();
        }
    }
    if (canBatch) {
        int[] res = prepStmt.executeBatch();
    }
    // commit changes only if all statements above succeeded
    conn.commit();
    //conn.setAutoCommit(true);
} catch (Exception ex) {
    // undo changes in the database
    conn.rollback();
    // give caller chance to react to failure
    throw ex;
} finally {
    conn.setAutoCommit(isAutoCommit);
}

```



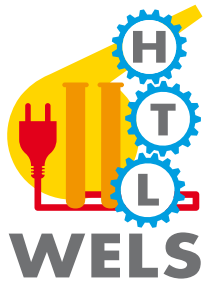
ÜBUNG



➤ 405_PreparedStatements

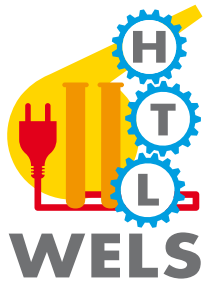


BLOBS





BLOBS

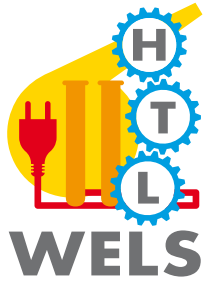


- BLOBs = Binary Large Objects
 - Verwende Streams zum Lesen/Schreiben von Daten
- BLOB in DB speichern:

```
void setBinaryStream (int parameterIndex,  
                      InputStream x,  
                      int length) throws SQLException  
  
// parameterIndex ... Index des Platzhalters im SQL-String  
// x ... InputStream (Lesen vom Datenträger: FileInputStream)  
// length ... Größe der Datei in Byte
```



BLOBS



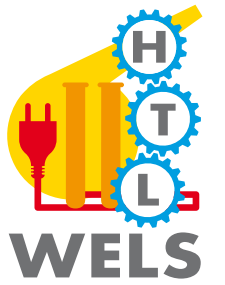
- BLOB aus DB lesen:

```
InputStream getBinaryStream  
    (int columnIndex) throws SQLException
```

```
InputStream getBinaryStream  
    (String columnName) throws SQLException
```



ÜBUNG



➤ 406_Blobs