Praktikum Softwareentwicklung 2

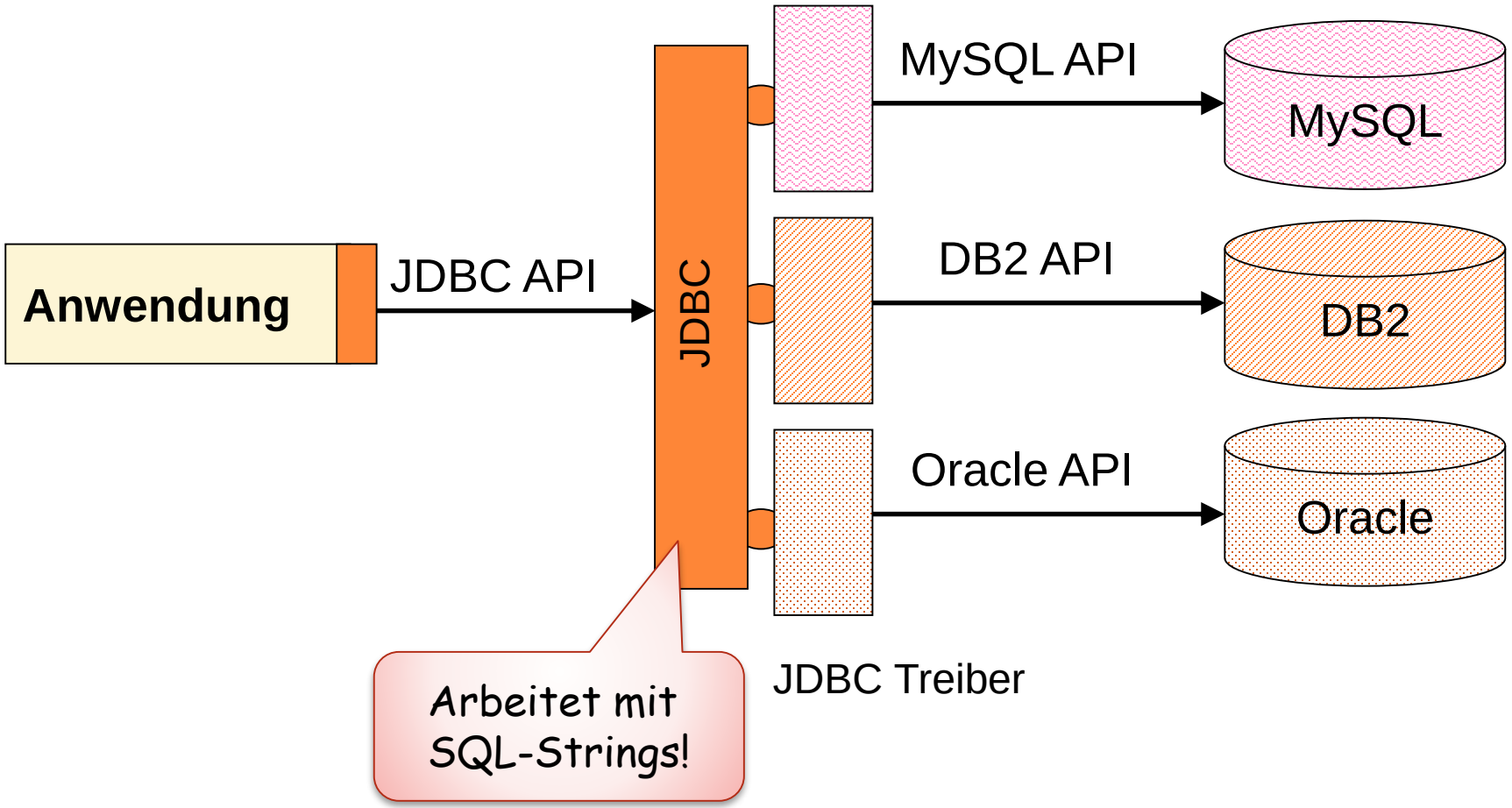
JDBC

JDBC

- Einführung
- Verbindungsaufbau
- Datenbankanweisungen
- Arbeiten mit ResultSet
- Typen
- Metadaten
- Transaktionen
- Zusammenfassung
- Datenbanksystem Derby

Motivation

Lösung: Zwischenschicht



Treiberinstallation

Download

- http://developers.sun.com/product/jdbc/drivers
- Datenbankhersteller

Installation

- Eintragen in den Klassenpfad

Achtung: Ab Version Java 6 nicht mehr nötig!

Registrieren bei Anwendung

- Bei dem Programmstart durch Parameter:
  - java –Djdbc.drivers=com.mysql.jdbc.Driver <Programm>
- Setzen der Systemeigenschaft "jdbc.drivers":
  - System.setProperty("jdbc.drivers", com.mysql.jdbc.Driver");
- Händisches Instanzieren der Treiber-Klasse:
  - Class.forName("com.mysql.jdbc.Driver");

Vorgehen

Typisch sind folgende Schritte:

1) Aufbau einer Verbindung zur Datenbank

2) Definition und ausführen von Datenbankkommandos

3) Verarbeiten der Ergebnisse

4) Ressourcen freigeben und Verbindung schließen

```
1.) Verbindung deklarieren
try {
    1.) Verbindung zur Datenquelle anfordern
```

```
    2.) SQL-Kommandos ausführen
```

```
    3.) Ergebnis verarbeiten
```

```
    4.) Ressourcen freigeben
} catch ( SQLException ) {
    Exception behandeln
} finally {
    try {
        4.) Verbindung schließen
    } catch (Exception) { Exception behandeln }
}
```

Klassen

für Schritt 1) und 4)

- `Driver` und `DriverManager`
- `Connection`
- `DataSource`

für Schritt 2)

- `Statement`
- `PreparedStatement`
- `CallableStatement`

für Schritt 3)

- `ResultSet`

JDBC

Einführung

Verbindungsaufbau

Datenbankanweisungen

Arbeiten mit ResultSet

Typen

Metadaten

Transaktionen

Zusammenfassung

Datenbanksystem Derby

Verbindungsaufbau

Verbindungsaufbau erfolgt mit

- Url zur Datenbank
- Benutername, Passwort

Datenbank Url

```
jdbc:<Datenbanktreiber>:<treiberspezifische Angaben>
```

Beispiele:

- MySql:
  jdbc:mysql://<host>:<port>/< Database >

- Derby:
  Embedded Mode: jdbc:derby:< Database >
  z.B.: jdbc:derby:C:/Users/hp/testDB

  Server Mode: jdbc:derby://<host>:<port>/< Database >
  z.B.: jdbc:derby://localhost:1527/testDB

DriverManager (1/2)

Achtung: Ab Version Java 6 nicht mehr nötig!

static-Methoden zur Verwaltung der Treiber, Verbindungsaufbau, Settings

Treiberverwaltung

```
static void registerDriver(Driver driver)
                        throws SQLException
static void deregisterDriver(Driver driver)
                        throws SQLException
static Enumeration<Driver> getDrivers()
```

Registrierung erfolgt üblicherweise im static-Initializers der Driver-Klasse;
es reicht daher, wenn Driverklasse geladen wird.

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

Beispiel: Registrieren des Treibers durch Laden der Driver-Klasse

DriverManager (2/2)

Verbindungsaufbau über Klasse `DriverManager`

- mit Url und optional Benutzer und Passwort
- liefert `Connection`-Objekt

```
public class DriverManager
        static Connection getConnection(  String url,
                                          String user,
                                          String password)
                           throws SQLException

        static Connection getConnection(String url)
                           throws SQLException
```

Beispiel:

```
String url = "jdbc:derby:C:/Users/hp/testDB";
Connection con = DriverManager.getConnection(url,"me","mypwd");
```

Settings: LogStream und Timeout

```
        static void setLogWriter(PrintWriter out)
        static PrintWriter getLogWriter()
        static void println(String message)
        static void setLoginTimeout(int seconds)
```

Beispiel Verbindungsaufbau

```
public static void main(String[] args) {

    String url = "jdbc:derby:C:/Users/hp/testDB;create=true";
    Connection conn = null;
    try {
        conn = DriverManager.getConnection(url);

        …


    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        try {
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Connection

`Connection` ist zentrales Objekt für Verbindung zur Datenbank

- Erzeugen von Statement-Objekten

```
Statement createStatement() throws SQLException
PreparedStatement prepareStatement(String sql) throws SQLException
CallableStatement prepareCall(String sql) throws SQLException
…
```

- Zugriff auf Datenbankinformationen (Metadaten)

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

- Transaktionen (lokale)

```
void commit() throws SQLException
void rollback() throws SQLException
void setAutoCommit(boolean autoCommit) throws SQLException
```

- Diverse Settings

```
void setReadOnly(boolean readOnly) throws SQLExcepti
void setTransactionIsolation(int level) throws SQLException
void setHoldability(int holdability) throws SQLException
```

```
TRANSACTION_READ_UNCOMMITTED
TRANSACTION_READ_COMMITTED
TRANSACTION_REPEATABLE_READ
TRANSACTION_SERIALIZABLE.
```

```
ResultSet.HOLD_CURSORS_OVER_COMMIT
ResultSet.CLOSE_CURSORS_AT_COMMIT
```

- Schließen

```
void close() throws SQLException
```

JDBC

Einführung

Verbindungsaufbau

Datenbankanweisungen

Arbeiten mit ResultSet

Typen

Metadaten

Transaktionen

Zusammenfassung

Datenbanksystem Derby

Erzeugen von Statement-Objekten

Es gibt 3 Arten von Statement-Objekten

- `Statement`: normale Anweisungen
- `PreparedStatement`: vorkompilierte Statements mit Input-Parametern
- `CallableStatement`: zur Ausführung von StoredProcedures mit Input- und Output-Parametern

Statement-Objekte werden durch Connection erzeugt

```
Statement createStatement() throws SqlException
Statement createStatement(int resultSetType, int resultSetConcurrency)
    throws SQLException
```

```
PreparedStatement prepareStatement(String sql) throws SQLException
PreparedStatement prepareStatement(String sql,
                                   int resultSetType,
                                   int resultSetConcurrency,
                                   int resultSetHoldability)
    throws SQLException
```

```
CallableStatement prepareCall(String sql) throws SQLException
CallableStatement prepareCall(String sql,
                                   int resultSetType,
                                   int resultSetConcurrency,
                                   int resultSetHoldability)
    throws SQLException;
```

Ausführen von Statement-Objekten

Statement-Objekte erlauben die Ausführung von SQL-Anweisungen

SQL-Anweisungen werden als Strings übergeben

3 Methoden zur Ausführung:

```
ResultSet executeQuery(String sql) throws SQLException
```

- Ausführung von SELECT-Statements
- liefert ResultSet als Ergebnis

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

- Ausführung von UPDATE, INSERT, DELETE, CREATE, …
- Ergebnis ist die Anzahl der geänderten Zeilen

```
boolean execute(String sql) throws SQLException
```

- Ausführung von Anweisungen mit mehreren Resultaten
  - Mehrere Ergebnisse vorhanden

    ```
    boolean getMoreResults() throws SQLException
    ```

  - Zugriff auf Ergebnisse

Beispiel Statement und PreparedStatement

```
public static void main(String[] args) {
  …
  try {
    …
    java.sql.Statement stmt = conn.createStatement();
    stmt.executeUpdate(
      "create table Person (id INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY, " +
       "firstname VARCHAR(128), lastName VARCHAR(128), born INTEGER)");
    …
  } catch (SQLException e) {
…
```

PreparedStatement

PreparedStatement sind vorkompilierte Statements (effizienter)

werden mit SQL-Anweisung durch Connection erzeugt

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

können Input-Parameter haben

- werden mit ? im SQL-String gekennzeichnet und identifiziert durch Position

```
"INSERT INTO test VALUES (?, ?)";
```

- Setzen mit set-Operationen entsprechenden Typs

```
void set<Type>(int n, <Type> x)
```

- Löschen aller Parameterwerte.

```
void clearParameters()
```

Beginnt bei 1

Beispiel:

```
PreparedStatement stat;
stat = con.prepareStatement("INSERT INTO test VALUES (?, ?)");
stat.setString(1, "Hallo");
stat.setString(2, "World!");
stat.executeUpdate();
```

Beispiel Statement und PreparedStatement

```
public static void main(String[] args) {
  …
  try {
    …
    java.sql.Statement stmt = conn.createStatement();
    stmt.executeUpdate(
      "create table Person (id INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY, " +
       firstname VARCHAR(128), lastName VARCHAR(128), born INTEGER)");

    PreparedStatement pStmt =
        conn.prepareStatement("insert into Person values(?, ?, ?)");
    pStmt.setString(1, "Hermann");
    pStmt.setString(2, "Maier");
    pStmt.setInt(3, 1971);
    pStmt.executeUpdate();
    pStmt.setString(1, "Michael");
    pStmt.setString(2, "Walchhofer");
    pStmt.setInt(3, 1977);
    pStmt.executeUpdate();
    …

    stmt.execute("update Person set born=1973 where id=1");
    ResultSet rs = stmt.executeQuery("select id, lastName, born from Person");
    …

  } catch (SQLException e) {
…
```

CallableStatement

CallableStatement: Ausführen von Datenbankprozeduren (SQL stored procedures

SQL-Strings stellen Prozeduraufruf dar:

- Parameterlose Prozedur

```
{call procedure_name}
```

- Prozedur:

```
{call procedure_name[(?, ?, ...)]}
```

- Funktion:

```
{? = call procedure_name[(?, ?, ...)]}
```

Das Setzen der Parameter erfolgt analog zu den `PreparedStatements`

Output-Parameter

- müssen als solche registriert werden

```
void registerOutParameter(int parameterIndex, int sqlType)
        throws SQLException
```

- Werte können mit get-Methoden ausgelesen werden

```
<Type> get<Type>(int parameterIndex) throws SQLException
```

Batch-Updates

Absetzen mehrerer Update-Kommandos in einem Batch

➔ zur Verbesserung der Performanz

Kommandos werden mit

```
void addBatch(String sqlCmd)
void addBatch()
```

angefügt

und mit

```
int[] executeBatch()
```

ausgeführt (Rückgabewert sind Anzahl der geänderten Zeilen pro Update)

Beispiel Batch-Updates

mit Statement

```
Statement stmt = conn.createStatement();
stmt.addBatch("insert into Person values('Mario', 'Scheiber', 1983)");
stmt.addBatch("insert into Person values('Rainer', 'Schönfelder', 1977)");
int[] upds = stmt.executeBatch();
```

mit PreparedStatement

```
PreparedStatement pStmt =
       conn.prepareStatement("insert into Person values(?, ?, ?)");

pStmt.setString(1, "Mario");
pStmt.setString(2, "Scheiber");
pStmt.setInt(3, 1983);
pStmt.addBatch();

pStmt.setString(1, "Rainer");
pStmt.setString(2, "Schönfelder");
pStmt.setInt(3, 1977);
pStmt.addBatch();
int[] upds = pStmt.executeBatch();
```

JDBC

Einführung

Verbindungsaufbau

Datenbankanweisungen

Arbeiten mit ResultSet

Typen

Metadaten

Transaktionen

Zusammenfassung

Datenbanksystem Derby

ResultSet

ResultSet stellt die Ergebnistabelle einer Abfrage dar

Es gibt 3 Arten von ResultsSets

- einfache: können nur sequentiell von vorne nach hinten durchlaufen werden
- scrollbare: erlaubt beide Richtungen und Positionierung
- scrollbare und änderbare: erlauben auch Updates

Unterscheidung erfolgt beim Erzeugen des Statements:

```
Statement createStatement(int resultSetType, int resultSetConcurrency)
        throws SQLException
```

```
ResultSet.TYPE_FORWARD_ONLY
ResultSet.TYPE_SCROLL_INSENSITIVE
ResultSet.TYPE_SCROLL_SENSITIVE
```

```
ResultSet.CONCUR_READONLY
ResultSet.CONCUR_UPDATABLE
```

`ResultSet.TYPE_FORWARD_ONLY`	– nur vorwärst lesend
`ResultSet.TYPE_SCROLL_INSENSITIVE`	– scrollbar vor und zurück, nur lesend
`ResultSet.TYPE_SCROLL_SENSITIVE`	– scrollbar, mit Updates
`ResultSet.CONCUR_READONLY`	– nur lesend
`ResultSet.CONCUR_UPDATABLE`	– erlaubt updates

Sequentieller Zugriff

`ResultSet` arbeitet mit Cursor auf aktuelle Zeile

- Weiterschalten auf nächste Zeile mit next

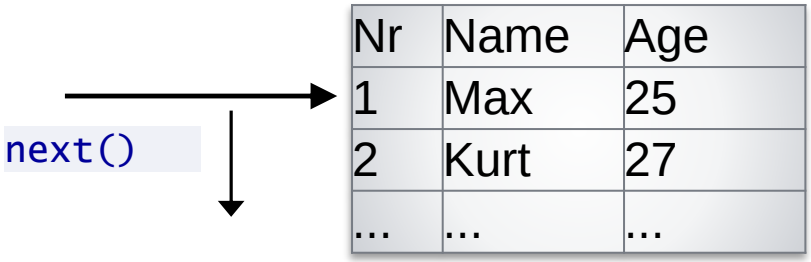
```
boolean next()
```

  - Anspringen der nächsten Zeile (begonnen wird **VOR** der ersten Zeile)
  - `true` solange noch eine gültige Zeile erreicht wurde.

- Zugriff mit get-Methoden auf Spaltenwerte der aktuellen Zeile

```
<Type> get<Type>(int column)
<Type> get<Type>(String colName)
int findColumn(String colName)
```

Beispiel:




```
getString(3) => 25
getString("Name") => Max
findColumn("Nr") => 1
```

Beispiel ResultSet

```
public static void main(String[] args) {
  …
  try {
      …
      ResultSet rs = stmt.executeQuery("select id, lastName, born from Person");
      while (rs.next()) {
        int nr = rs.getInt(1);
        String lastName = rs.getString("lastName");
        int born = rs.getInt("born");
        System.out.println(nr + ": " + lastName + " born: " + born);
      }

  } catch (SQLException e) {
    …
  }
}
```

Zugriff mit scrollable ResultSets

- `boolean first()`
  - Erste Zeile im ResultSet.
  - true wenn eine gültige Zeile erreicht wurde.
- `beforeFirst()`
  - Vor die erste Zeile im ResultSet.
- `boolean last()`
  - Letzte Zeile im ResultSet.
  - true wenn eine gültige Zeile erreicht wurde.
- `afterLast()`
  - Nach letzter Zeile im ResultSet.
- `boolean absolute(int row)`
  - Eine Zeile anspringen
    row > 0 ... von oben gezählt (1 erste Zeile, 2 zweite Zeile, ...)
    row < 0 ... von unten gezählt (-1 letzte Zeile, -2 vorletzte Zeile, ...)
  - true wenn eine gültige Zeile erreicht wurde.
- `boolean relative(int rows)`
  - Überspringen von rows Zeilen
- `int getRow()`
  - Nummer der aktuellen Zeile
- `boolean previous()`
  - Springen auf die vorherige Zeile

Beispiel scrollable ResultSet

```
Statement scrStmt =
    conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
ResultSet scrRs = scrStmt.executeQuery("select id, lastName, born from Person");

// put out rows after 2nd
scrRs.absolute(2);
while (scrRs.next()) {
    int nr = scrRs.getInt(1);
    String lastName = scrRs.getString("lastName");
    int born = scrRs.getInt("born");
    System.out.println(nr + ": " + lastName + " born: " + born);
}

// put out rows in reverse order
scrRs.afterLast();
while (scrRs.previous()) {
    int nr = scrRs.getInt(1);
    String lastName = scrRs.getString("lastName");
    int born = scrRs.getInt("born");
    System.out.println(nr + ": " + lastName + " born: " + born);
}
```

Updates mit ResultSet

ResultSets und zugrundeliegende Datenquelle kann geändert werden
Änderbare ResultSets erzeugt durch createStatement mit

```
Statement stmt = con.createStatement(
                 ResultSet.TYPE_SCROLL_SENSITIVE,
                 ResultSet.CONCUR_UPDATABLE);
```

```
void update<Type>(int column, <Type> val)
void update<Type>(String colName, <Type> val)
```

Änderunungen im ResultSet mit update-Methoden (in aktueller Zeile)
Änderungen in Datenbank mit

updateRow()
- um Änderung in aktueller Zeile in Datenquelle zu schreiben

deleteRow()
- um aktuelle Zeile zu löschen

insertRow()
- um neue Zeile anzufügen

Anfügen von neuen Zeilen

- Springen auf spezielle Zeile mit **moveToInsertRow()**
- Datenänderungen mit Updates
- Einfügen mit **insertRow()**

```
rs.moveToInsertRow();
rs.updateInt(2, 3857);
…
rs.insertRow();
```

Beispiel Update bei ResultSets

```
Statement updStmt = conn.createStatement(
                                ResultSet.TYPE_SCROLL_SENSITIVE,
                                ResultSet.CONCUR_UPDATABLE);

ResultSet updRs = updStmt.executeQuery("select id,firstName,lastName,born from Person");

updRs.absolute(3);
while (updRs.previous()) {
    int born = updRs.getInt("born");
    // decrease born by 1
    updRs.updateInt("born", born - 1);
    updRs.updateRow();
}

// insert one new row
updRs.moveToInsertRow();
updRs.updateInt(1, 5);
updRs.updateString("firstName", "Chritoph");
updRs.updateString("lastName", "Gruber");
updRs.updateInt("born", 1976);
updRs.insertRow();

// delete first row
updRs.absolute(1);
updRs.deleteRow();
```

JDBC

- Einführung
- Verbindungsaufbau
- Datenbankanweisungen
- Arbeiten mit ResultSet
- Typen
- Metadaten
- Transaktionen
- Zusammenfassung
- Datenbanksystem Derby

Typen

Standard-Typemapping zwischen SQL und JAVA

SQL Type	Java Type
CHAR, VARCHAR, LONGVARCHAR	String
NUMERIC, DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT, DOUBLE	double
BINARY, VARBINARY, LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Java Typ ⇨ JDBC Typ

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	BOOLEAN	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	ARRAY	BLOB	CLOB	STRUCT	REF	DATALINK	JAVA_OBJECT	ROWID	NCHAR	NVARCHAR	LONGNVARCHAR	NCLOB	SQLXML
String	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x									x	x	x		
java.math. BigDecimal	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
Boolean	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
Byte	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
Short	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
Integer	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
Long	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
Float	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
Double	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
byte[]															x	x	x																
java.sql. Date												x	x	x				x		x													
java.sql. Time												x	x	x					x														
java.sql. Timestamp												x	x	x				x	x	x													
java.sql.Array																					x												
java.sql.Blob																						x											
java.sql.Clob																							x										
java.sql.Struct																								x									
java.sql.Ref																									x								
java.net.URL																										x							
Java class																											x						
java.sql.RowId																												x					
java.sql.NClob																																x	
java.sql.SQLXML																																	x

JSR 221, JDBC Specification 4, November 7, 2006, Page 198

JDBC Typ ⇨ Java Typ

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	BOOLEAN	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	DATALINK	STRUCT	JAVA_OBJECT	ROWID	NCHAR	NVARCHAR	LONGNVARCHAR	NCLOB	SQLXML
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x	x														x					
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x	x																			
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x	x																			
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x	x																			
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x	x																			
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x	x																			
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x	x																			
getBoolean	x	x	x	x	x	x	x	x	x	X	X	x	x	x																			
getString	x	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x					x				x	x	x		
getNString	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x					x				X	X	x		
getBytes															X	X	x																
getDate												x	x	x				X		x													
getTime												x	x	x					X	x													
getTimestamp												x	x	x				x	x	X													
getAsciiStream												x	x	X	x	x	x				x											x	
getBinaryStream															x	x	X					x											x
getCharacterStream												x	x	X	x	x	x				x								x	x	x	x	x
getNCharacterStream												x	x	x	x	x	x				x								x	x	X	x	x
getClob																					X											x	
getNClob																					x											X	
getBlob																						X											
getArray																							X										
getRef																								X									
getURL																									X								
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x
getRowId																												X					
getSQLXML																																	X

JSR 221, JDBC Specification 4, November 7, 2006, Page 199

JDBC

Einführung

Verbindungsaufbau

Datenbankanweisungen

Arbeiten mit ResultSet

Typen

Metadaten

Transaktionen

Zusammenfassung

Datenbanksystem Derby

Metadaten (1/2)

Zugriff auf Metadaten über Datenbank über Connection-Objekt

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

DatabaseMetaData erlaubt Zugriff auf Informationen über Datenbanken

```
public interface DatabaseMetaData extends Wrapper {
    String getDatabaseProductName() throws SQLException;
    boolean supportsTransactions() throws SQLException;
    ResultSet getProcedures(String catalog, String schemaPattern,
          String procedureNamePattern) throws SQLException;
    ResultSet getTables(String catalog, String schemaPattern,
          String tableNamePattern, String types[]) throws SQLException;
    ResultSet getSchemas() throws SQLException;
    ResultSet getCatalogs() throws SQLException;
    ResultSet getColumns(String catalog, String schemaPattern,
    String tableNamePattern, String columnNamePattern) throws SQLException;
    ResultSet getPrimaryKeys(String catalog, String schema, String table)
          throws SQLException;
    …
}
```

Metadaten (2/2)

Zugriff über Metadaten über ResultSet, z.B. Tabellen

```
ResultSet getTables(String catalog, String schemaPattern,
                    String tableNamePattern, String types[])
        throws SQLException;
```

ResultSet enthält Zeilen mit Beschreibung der Tabellen mit

- Tabellenname table: TABLE_NAME
- Tabellenschema: TABLE_SCHEM
- Typ der Tabelle TABLE_TYPE
- …

Beispiel:

```
conn = DriverManager.getConnection(dbUrl);
DatabaseMetaData metaData = conn.getMetaData();
ResultSet tableMetaData = metaData.getTables(null, null, null, null);
while (tableMetaData.next()) {
    System.out.println(
          tableMetaData.getString("TABLE_NAME") + ", " +
          tableMetaData.getString("TABLE_TYPE") + ", " +
          tableMetaData.getString("TABLE_SCHEM")
    );
}
```

Keine Einschränkung der Suche!

```
SYSALIASES, SYSTEM TABLE, SYS
SYSCHECKS, SYSTEM TABLE, SYS
SYSTABLES, SYSTEM TABLE, SYS
SYSTRIGGERS, SYSTEM TABLE, SYS
SYSUSERS, SYSTEM TABLE, SYS
SYSVIEWS, SYSTEM TABLE, SYS
...
PERSONS, TABLE, APP
```

JDBC

Einführung

Verbindungsaufbau

Datenbankanweisungen

Arbeiten mit ResultSet

Typen

Metadaten

Transaktionen

Zusammenfassung

Datenbanksystem Derby

Transaktionen

Connection unterstützt Transaktionen

Auto Commit:

- Bei AutoCommit ist jede Anweisung eine abgeschlossene Transaktion
- Kann mit setAutoCommit bei Connection ausgeschalten werden

```
<Connection>.setAutoCommit(boolean autoCommit)
```

- Abfragen:

```
boolean <Connection>.getAutoCommit()
```

Abschliessen einer Transaktion:

```
<Connection>.commit()
```

Rücksetzen im Fehlerfall (z.B.: SQLException):

```
<Connection>.rollback()
```

```
Connection conn;
...
try {
  conn.setAutoCommit(false);
  Statement stat = conn.createStatement();
  stat.executeUpdate("INSERT ...");
  stat.executeUpdate("INSERT ...");
  stat.executeUpdate("UPDATE ...");
  conn.commit();
} catch (SQLException e) {
  conn.rollback();
}
```

JDBC

Einführung

Verbindungsaufbau

Datenbankanweisungen

Arbeiten mit ResultSet

Typen

Metadaten

Transaktionen

Zusammenfassung

Datenbanksystem Derby

Zusammenfassung

Datenbankunabhängigkeit

- Zwischenschicht
- Treiberschnittstelle (mind. SQL 92)
  - 4 Treiberarten:
    JDBC -> ODBC
    Teilweise Java
    Nur Java zu einer Middleware
    Nur Java zur Datenbank
- Einfachere Programmentwicklung

Beliebige SQL-Kommandos absetzbar

- Optimierung / Datenbankabhängigkeit

Arten von Statements

- `java.sql.Statement`
  - Statisch oder vom Benutzer frei wählbar (Achtung: SQL injection)
- `java.sql.PreparedStatement`
  - Vorbereitete Statements (Sicher gegen SQL injection, schnell)
- `java.sql.CallableStatement`
  - Ausführen von SQL stored procedures

Weiteres

`DataSource` als Ersatz für DriverManager (bereits bei JDBC 3.0)

- erlaubt Auffinden von Datenquellen über JNDI
- ermöglicht Connection-Pooling
- ermöglicht verteilte Transaktionen

Neuerungen in JDBC 4.0 (Java 6.0)

- Spezifiziert in JSR-221
- Automatisches Laden des Treibers beim Verbindungsaufbau
- SQL:2003
- Unterstützung großer Objekte (CLOB, BLOB)
- Mehr Datentypen (SQLXML)
- Neue Exceptions
  - SQLTransientException
  - SQLRecoverableException
  - SQLNonTransientException
  - …
- Java Datebase Derby

JDBC

- Einführung
- Verbindungsaufbau
- Datenbankanweisungen
- Arbeiten mit ResultSet
- Typen
- Metadaten
- Transaktionen
- Zusammenfassung
- Datenbanksystem Derby

Derby (Java DB) Installation

Download

http://db.apache.org/derby

Umgebungsvariablen

- DERBY_HOME = Pfad zur Derby-Installation
- PATH - Anfügen von bin-Verzeichnis der Derby-Installation
- CLASSPATH – Anfügen von
  - %DERBY_HOME%\lib\derby.jar
  - %DERBY_HOME%\lib\derbytools.jar

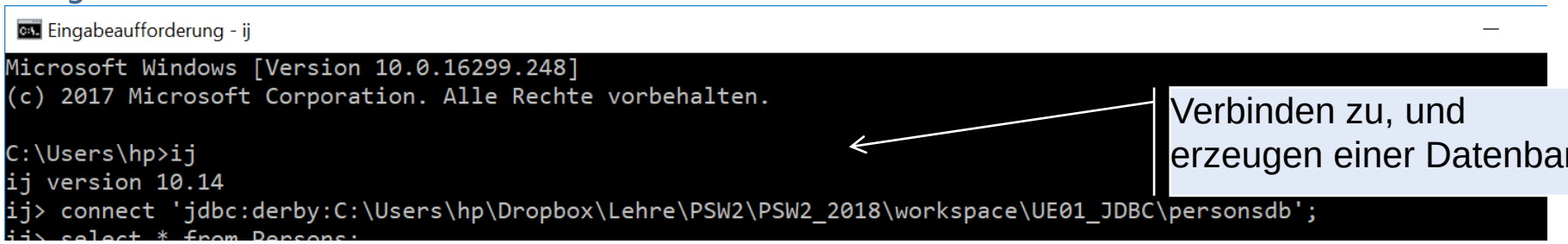
Verzeichnis der Derby-Installation

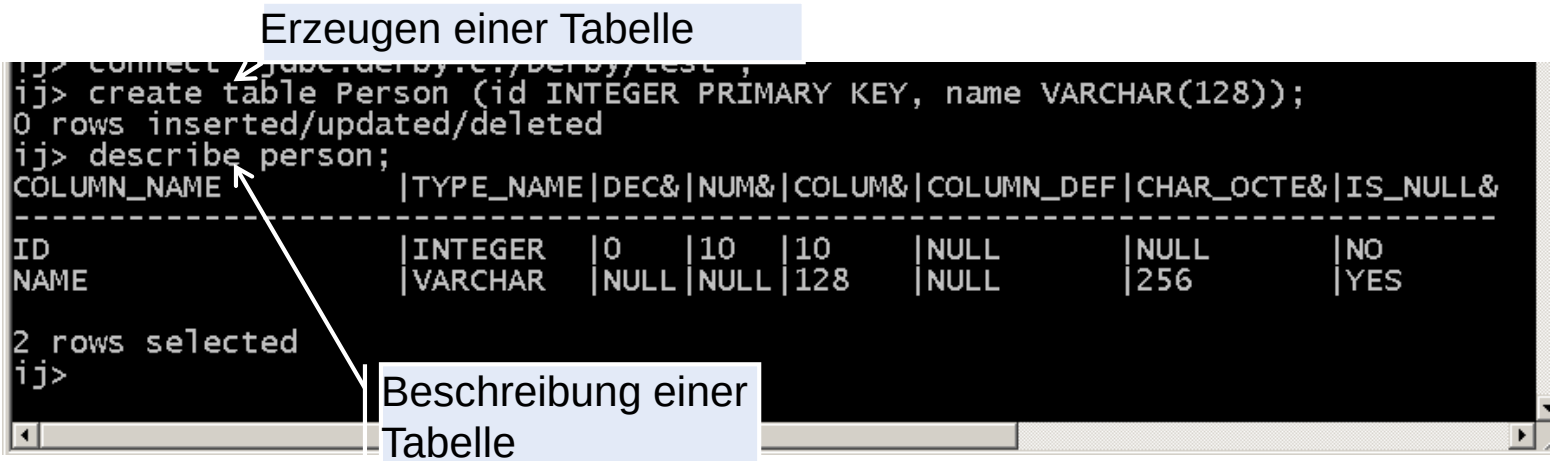
```
>set DERBY_HOME=C:\Programme\Derby\

>set CLASSPATH=%DERBY_HOME%\lib\derby.jar;%DERBY_HOME%\lib\derbytools.jar;%CLASSPATH%

>set PATH=%DERBY_HOME%\bin;%PATH%
```

Arbeiten mit Derby

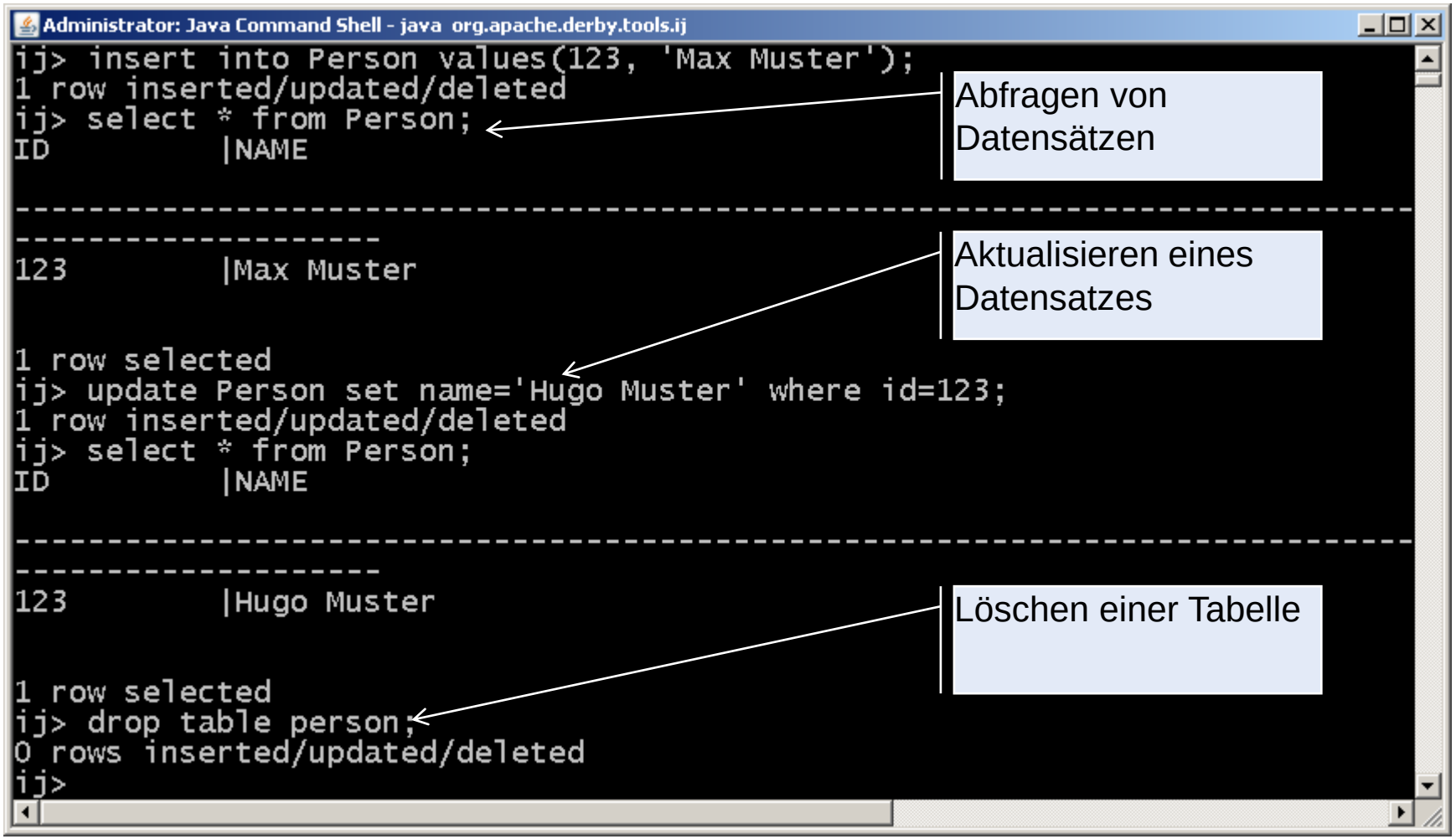
Kommandozeilenwerkzeug

- ij



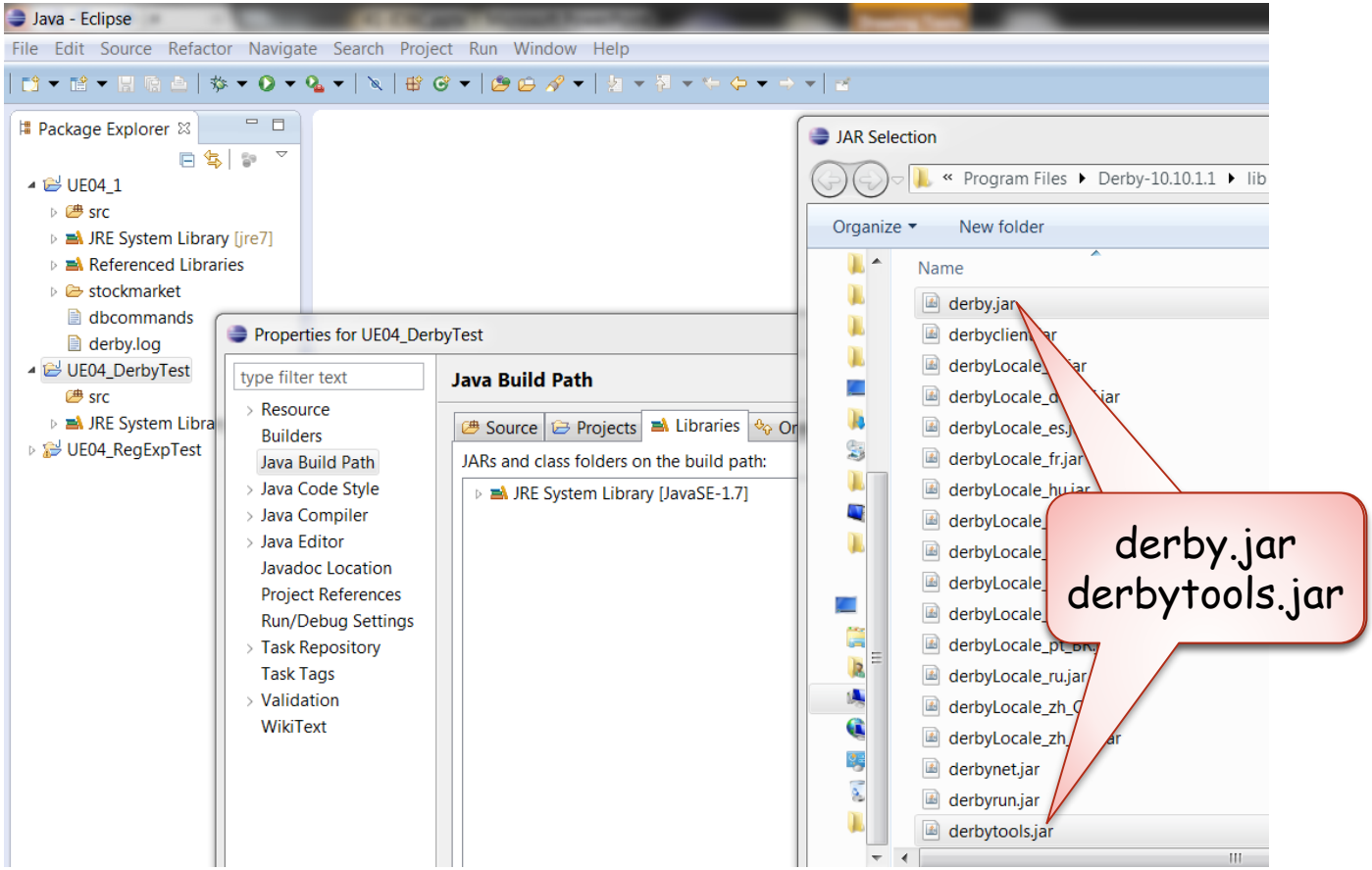


Arbeiten mit Derby



Verwendung von Derby in Eclipse

Anfügen von Derby-Libraries im Java Build Path



Derby Embedded Mode

Embedded Mode = Derby wird aus Applikations VM gestartet

- Starten durch Laden der Klasse **`org.apache.derby.jdbc.EmbeddedDriver`**
- in main-Methode

```
public static void main(String[] args) {
    String driver = "org.apache.derby.jdbc.EmbeddedDriver";
    String dbName = "jdbcDemoDB";
    String connectionURL = "jdbc:derby:" + dbName + ";create=true";
    try {
        /*
         * * Load the Derby driver.* When the embedded Driver is used this
         * action start the Derby engine.* Catch an error and suggest a
         * CLASSPATH problem
         */
        Class.forName(driver);
        System.out.println(driver + " loaded. ");
    } catch (java.lang.ClassNotFoundException e) {
        System.err.print("ClassNotFoundException: ");
        System.err.println(e.getMessage());
        System.out
                .println("\n    >>> Please check your CLASSPATH variable   <<<\n");
        return;
    }
    Connection conn = null;
    try {
        conn = DriverManager.getConnection(connectionURL);
        System.out.println("Connected to database ");
```

Started die Derby-Datenbank im Embedded-Mode