

Java Database Connectivity-API (JDBC)



JDBC

Einführung

Verbindungsaufbau

Datenbankanweisungen

Arbeiten mit ResultSet

Typen

Metadaten

Transaktionen

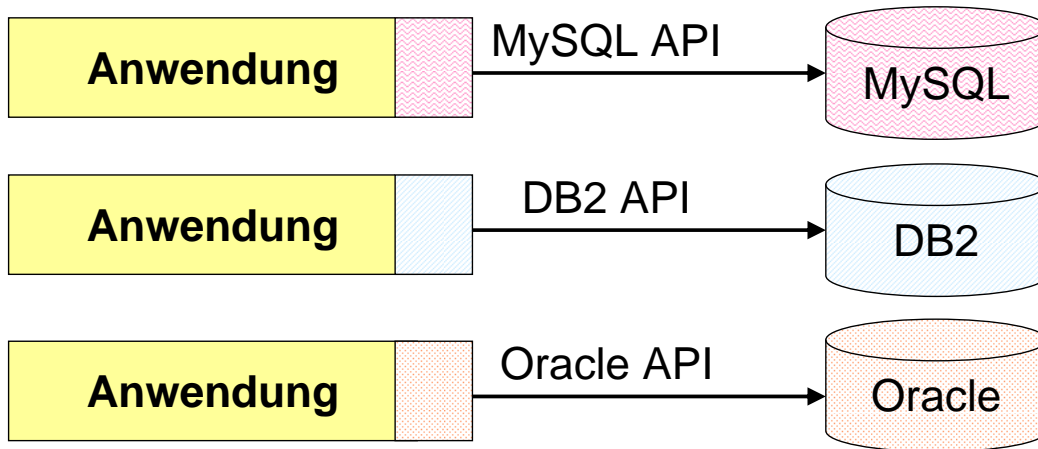
Zusammenfassung

Datenbanksystem Derby



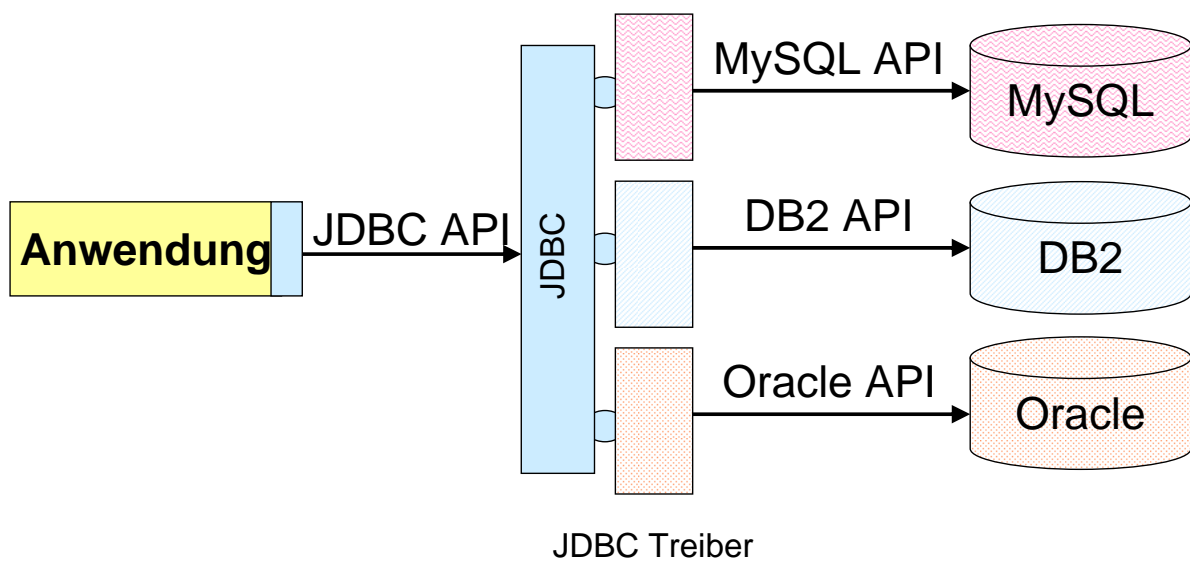
Motivation (1/2)

- **Problem: Zugriff auf ein DBMS ist Herstellerabhängig**



Motivation (2/2)

- **Lösung: Zwischenschicht**

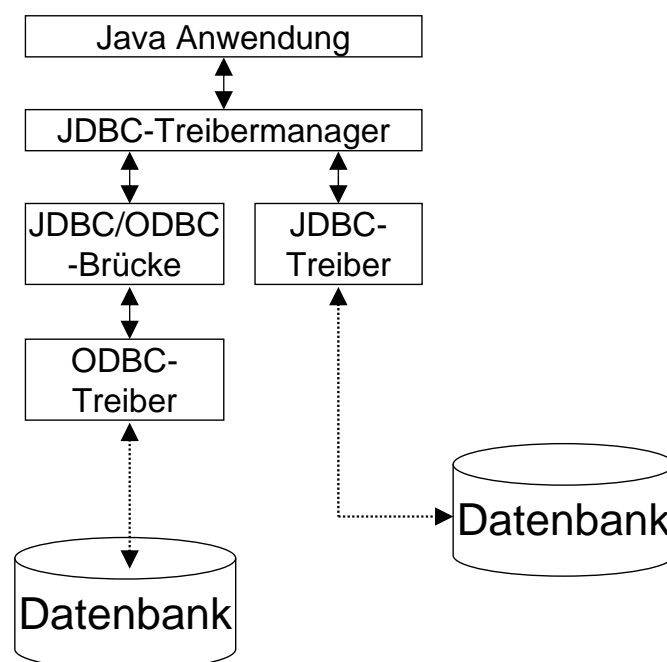


Entwicklung

- Entwicklung seit: 1995
- Erster Ansatz: Java erweitern
- Zweiter Ansatz: Treiber der Datenbankhersteller
- Vorbild: ODBC
- Unterschiede:
 - ODBC wenige Befehle, viele Optionen
 - JDBC viele einfache Methoden
 - ODBC nutzt void-Zeiger
 - Java kennt keine Zeiger
- Flexibilität: JDBC erlaubt beliebige SQL-Statements (Zeichenfolgen)
 - Anpassung an Datenbank möglich
 - Optimierung ✓
 - Bindung ✗



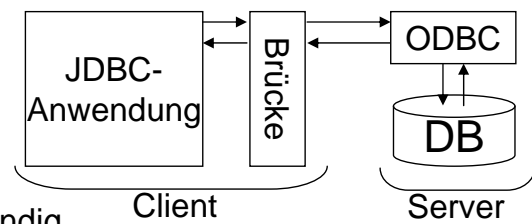
Architektur



Treiber (1/2)

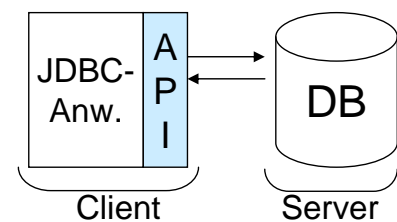
Typ 1: JDBC/ODBC-Brücke

- ODBC ist sehr weit verbreitet ✓
- Leistung ✗
- Wartung ✗
- Testen, Experimentieren
- kein eigener JDBC-Treiber notwendig
- Windows Plattformen



Typ 2: Partial Java Driver

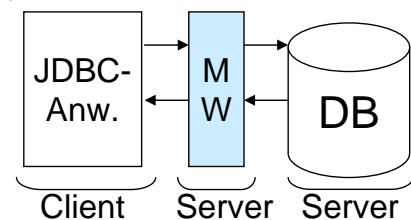
- konvertiert JDBC-Aufruf in DB-abhängigen API-Aufruf ✓
- schnell, weil API-Aufruf kompiliert ist ✓
- abhängig von DB + OS ✗
- Client braucht plattformabhängige API ✗



Treiber (2/2)

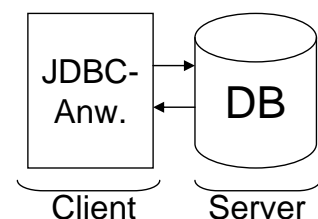
Typ 3: Reiner Java Treiber zu Middleware

- Keine plattformabhängigen Treiber am Client ✓
- DB-unabhängig ✓
- Flexibel, mehrere DB möglich ✓
- DB-abhängiger Code in Middleware



Typ 4: Reiner Java Treiber direkt zur DB

- JDBC in DB-spezifische Netzwerkaufufe verpackt
- Schnell ✓
- Keine plattformabhängigen Treiber am Client ✓
- Client braucht für verschiedene DB verschiedene Treiber ✗



Treiberinstallation

- Download
 - <http://developers.sun.com/product/jdbc/drivers>
 - Datenbankhersteller
- Installation
 - Eintragen in den Klassenpfad
- 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");`



Achtung: Ab Version Java 6 nicht mehr nötig!

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.) Ergebnisse 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
 - Benuternamen, Passwort

- Datenbank Url

`j dbc: <Datenbanktreiber>: <treiberspezifische Angaben>`

Beispiele:

- MySQL:
`j dbc:mysql://<host>:<port>/< Datenbankname>`
- JavaDB (Derby):
`j dbc:derby:url_to_Database` z.B. `j dbc:derby://localhost:1527/TestDB`
`j dbc:derby:Database`



DriverManager (1/2)

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.

Beispiel:

Registrieren des Treibers durch Laden der Driver-Klasse */

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

Achtung: Ab Version Java 6 nicht mehr nötig!



DriverManager (2/2)

▪ Verbindungsaufbau

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

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

Beispiel:

```
String url = "jdbc:odbc:wombat";
Connection con =
    DriverManager.getConnection(url, "oboy", "12Java");
```

▪ Settings: LogStream und Timeout

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



Beispiel Verbindungsaufbau

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

    String url = "jdbc:derby:c:/Derby/test;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 SQLException  
void setTransactionIsolation(int level) throws SQLException  
void setHoldability(int holdability) throws SQLException
```

- 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
- liefert true, wenn Ergebnis ResultSet ist
- Zugriff auf Ergebnisse mit:

```
ResultSet getResultSet() throws SQLException  
boolean getMoreResults() throws SQLException
```



PreparedStatement

- *PreparedStatement* sind vorkompilierte Statements (effizienter)
- werden mit SQL-Anweisung durch Connection erzeugt

```
PreparedStatement preparedStatement(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)`
 - Setzen des Parameters an der Stelle n (1 .. m).
 - Löschen aller Parameterwerte.
`void clearParameters()`

Beispiel:

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



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 PreparedStatement
- 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
```



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, firstname VARCHAR(128), " +
            "lastName VARCHAR(128), born INTEGER)");

        PreparedStatement pStmt =
            conn.prepareStatement("insert into Person values(?, ?, ?, ?)");
        pStmt.setInt(1, 1);
        pStmt.setString(2, "Hermann");
        pStmt.setString(3, "Mayer");
        pStmt.setInt(4, 1971);
        pStmt.executeUpdate();
        pStmt.setInt(1, 2);
        pStmt.setString(2, "Michael");
        pStmt.setString(3, "Walchofer");
        pStmt.setInt(4, 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) {
        ...
    }
}
```



Beispiel ResultSet

- Wie viele Zeilen hat ein ResultSet?

```
Connection con;
...
Statement stat = con.createStatement();
ResultSet result = stat.executeQuery("SELECT ...");
int rowAmount;

result.last();
rowAmount = result.getRow();
result.beforeFirst();

// Mit ResultSet arbeiten
while (result.next()) {
    ...
}
```



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(5, 'Mario', 'Scheiber', 1983)");
stmt.addBatch("insert into Person values(6, 'Rainer', 'Schönfelder', 1977)");
int[] upds = stmt.executeBatch();
```

- mit PreparedStatement

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

pstmt.setInt(1, 6);
pstmt.setString(2, "Mario");
pstmt.setString(3, "Scheiber");
pstmt.setInt(4, 1983);
pstmt.addBatch();

pstmt.setInt(1, 7);
pstmt.setString(2, "Rainer");
pstmt.setString(3, "Schönfelder");
pstmt.setInt(4, 1977);
pstmt.addBatch();
int[] upds = pstmt.executeBatch();
```



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



Sequentieller Zugriff

- ResultSet arbeitet mit Cursor auf aktuelle Zeile
 - Zugriff mit get-Methoden auf Spaltenwerte der aktuellen Zeile

```
Typ> get<Type>(int spalte)
<Type> get<Type>(String spaltenName)
int findColumn(String spaltenName)
```
 - 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.

Beispiel:

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

next()

Nr	Name	Age
1	Max	25
2	Kurt	27
...



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) {
        ...
    }
}
```



ResultSet

- ResultSet stellt die Ergebnistabelle einer Abfrage dar
- Es gibt unterschiedliche Arten von ResultSets
 - *einfache*: können nur sequentiell von vorne nach hinten durchlaufen werden
 - *scrollbare*: erlaubt beide Richtungen und Positionierung
 - *änderbare*: erlauben auch Änderungen in der Datenbank
 - *sensitive*: zeigen Änderungen in Datenbank an
- 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	- einfacher ResultSet, nur lesend
ResultSet.TYPE_SCROLL_INSENSITIVE	- scrollbar, Änderungen in DB nicht angezeigt
ResultSet.TYPE_SCROLL_SENSITIVE	- scrollbar, Änderungen in DB werden angezeigt
ResultSet.CONCUR_READONLY	- nur lesend
ResultSet.CONCUR_UPDATABLE	- erlaubt updates



Zugriff mit scrollable ResultSets

`boolean next()`

- Nächste Zeile.

`boolean previous()`

- Vorherige Zeile.

`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 isBeforeFirst(), boolean isFirst,`
`boolean isLast(), boolean isAfterLast()`

- Prüfmethode



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);
```

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

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

- 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", "Christoph");  
updRs.updateString("lastName", "Gruber");  
updRs.updateInt("born", 1976);  
updRs.insertRow();  
  
// delete first row  
updRs.absolute(1);  
updRs.deleteRow();
```



Einführung
Verbindungsaufbau
Datenbankanweisungen
Arbeiten mit ResultSet

Typen

Metadaten
Transaktionen
Zusammenfassung
Datenbanksystem Derby



Typen

- Standard-Type-Mapping 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 (PreparedStatement)

	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	SQXML	
String	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	



JDBC Typ ⇒ Java Typ (ResultSet)

	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	SQXML		
getBytes	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x															
getDate												x	x	x				x		x															
getTime												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	x					x												x	
getCharacterStream											x	x	x	x	x	x	x					x												x	
getNCharacterStream											x	x	x	x	x	x	x					x												x	
getClob																							x											x	
getNClob																																		x	
getBlob																							x												
getArray																																			
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



- 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**

`ResultSetMetaData getMetaData()` throws `SQLException`

- Zugriff auf Spalteninformation

```
String getColumnName(int column)
int getColumnType(int column)
String getColumnTypeName(int column)
String getTableName(int column)
...
```

Nr	Name	Title	...	Age
1	Max	Mag.	...	25
2	Kurt	DI	...	27
...

`SELECT Nr, Name, Age FROM users;`

```
getColumnCount()           => 3
getColumnName(1)          => "Nr"
getColumnType(3)          => 4
getColumnTypeName(3)      => "int"
getTableName(2)           => "users"
```

Nr	Name	Age
1	Max	25
2	Kurt	27
...



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` ausgeschaltet 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 Database Derby



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



Arbeiten mit Derby

- Umgebungsvariablen
 - JAVA_HOME= Pfad zur Java-Installation
 - CLASSPATH
 - + C:\Programme\Sun\JavaDB\lib\derby.jar
 - + C:\Programme\Sun\JavaDB\lib\derbytools.jar
 - PATH
 - + C:\Programme\Sun\JavaDB\frameworks\embedded\bin
- Systeminformationen
 - java org.apache.derby.tools.sysinfo

Verzeichnis der Derby-
Installation

```
Administrator: Java Command Shell
C:\daten>java org.apache.derby.tools.sysinfo
----- Java Information -----
Java Version:      1.6.0
Java Vendor:      Sun Microsystems Inc.
Java home:        C:\Program Files\Java\jre1.6.0
Java specification version: 1.6
----- Derby Information -----
JRE - JDBC: Java SE 6 - JDBC 4.0
[C:\Program Files\Java\jdk1.6.0\db\lib\derby.jar] 10.2.1.7 - (453926)
[C:\Program Files\Java\jdk1.6.0\db\lib\derbytools.jar] 10.2.1.7 - (453926)
----- License Information -----
```



Arbeiten mit Derby

- Kommandozeilenwerkzeug
 - java org.apache.derby.tools.ij

```
Administrator: Java Command Shell - java org.apache.derby.tools.ij
Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\daten>java org.apache.derby.tools.ij
ij version 10.2
ij> connect 'jdbc:derby:c:/Derby/test;create=true';
ij>
```

Verbinden zu, und erzeugen einer Datenbank

```
Administrator: Java Command Shell - java org.apache.derby.tools.ij
C:\daten>java org.apache.derby.tools.ij
ij version 10.2
ij> connect 'jdbc:derby:c:/Derby/test';
ij> create table Person (id INTEGER PRIMARY KEY, name VARCHAR(128));
0 rows inserted/updated/deleted
ij> describe person;
COLUMN_NAME      |TYPE_NAME|DEC&|NUM&|COLUM&|COLUMN_DEF|CHAR_OCTE&|IS_NULL&
-----
ID                |INTEGER  |0   |10  |10   |NULL      |NULL      |NO
NAME              |VARCHAR  |    |    |128  |NULL      |256       |YES
2 rows selected
ij>
```

Erzeugen einer Tabelle

Beschreibung einer Tabelle



Arbeiten mit Derby

```
Administrator: Java Command Shell - java org.apache.derby.tools.ij
ij> insert into Person values(123, 'Max Muster');
1 row inserted/updated/deleted
ij> select * from Person;
ID      |NAME
-----
123     |Max Muster
1 row selected
ij> update Person set name='Hugo Muster' where id=123;
1 row inserted/updated/deleted
ij> select * from Person;
ID      |NAME
-----
123     |Hugo Muster
1 row selected
ij> drop table person;
0 rows inserted/updated/deleted
ij>
```

Abfragen von Datensätzen

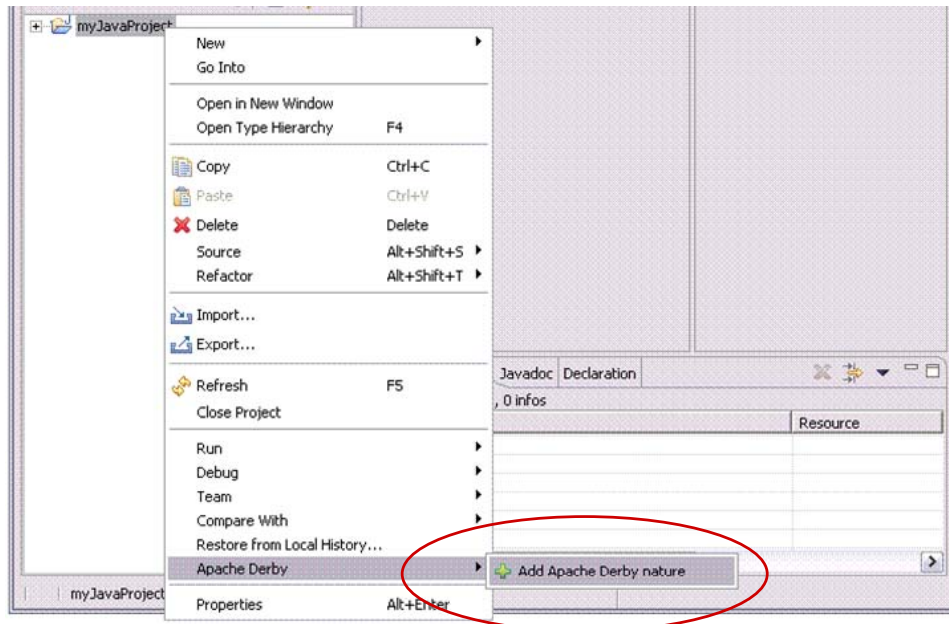
Aktualisieren eines Datensatzes

Löschen einer Tabelle



Derby Eclipse Plug-in

- Derby Plug-in für Eclipse
http://db.apache.org/derby/derby_downloads.html
- Installation wie jedes andere Plug-in (ins plugin-Verzeichnis kopieren)
- Hinzufügen von Derby ins aktuelle Projekt



Derby Driver und Connection-String

- Derby Driver

`org.apache.derby.jdbc.ClientDriver`

- Derby Connection-String

`jdbc:derby://localhost:1527/Name_of_DB`

**Derby Network-Server
muss gestartet sein!**

`jdbc:derby:Name_of_DB`

**Zugriff auf Files: Nur
eine Verbindung
möglich!**

