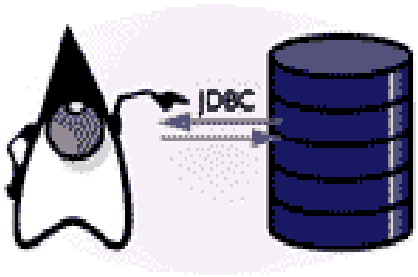
Skriptum

Grundlagen

JDBC mit ORACLE

"Java Database Connectivity"



v 0.5

- Thomas Stütz -

Schuljahr 2007/2008

Änderungen

Version	**Datum**	**Änderungsbeschreibung**
0.1	-	Ersterstellung im Schuljahr 2005/2006
0.4	Jänner 2007	Erweiterungen, Fehlerkorrekturen
0.5	Oktober 2007	Neuerungen Java 6
0.5b	Feber 2008	Batch – Verarbeitung; Ergänzung Driver Manager; lfd. Aktualisierungen und Fehlerkorrekturen
0.5c	März 2008	Ergänzungen im Kapitel „DataSource“

Inhalt

Quellen

Partl Hubert	Java – Einführung, Kursunterlage, Boku Wien http://www.boku.ac.at/javaeinf/jein.html sehr guter Einstieg in die Programmiersprache Java. Teile daraus wurden in dieses Skript übernommen.	2005
Joller, J.M.	JDBC - J2EE – Grundlagen und Praxis http://www.joller-voss.ch/java/notes/jdbc/jdbc_j2ee.html Ein hervorragendes Skript mir vielen Beispielen (inkl. Lösungen) basierend auf der Datenbank Cloudscape (jetziger Name: Derby , http://db.apache.org/derby/)	2004
Petkovic, Brüderl	Java in Datenbanksystemen – JDBC, SQLJ, Java DB-Systeme und -Objekte, Addison-Wesley	2002
Bonazzi, Stokol	Oracle und Java – Datenbankentwicklung für E-Commerce-Anwendungen	2002
Middendorf, Singer, Heid	Programmierhandbuch und Referenz für die JavaTM-2-Plattform, Standard Edition, 3. Auflage http://www.dpunkt.de/java//index.html	2002
Wiki-Book	Java Standard: JDBC http://de.wikibooks.org/wiki/Java_Standard:_JDBC	-
Louis, Müller	Das Java Codebook, Addison-Wesley	2005
Sauer, Jürgen	FH Regensburg Skriptum Datenbanken (http://fbim.fh-regensburg.de/~saj39122/DB/skript/DB_Skript_SS2006.doc) Skriptum Objektorientierte Programmierung (http://fbim.fh-regensburg.de/~saj39122/oop/index.html)	WS02/03
Sun	http://java.sun.com/docs/books/tutorial/jdbc/overview/index.html	01-2007
Sun	http://java.sun.com/docs/books/tutorial/jdbc/basics/index.html	01-2007
Sun	JDBC™ 4.0 Specification, JSR 221, Lance Andersen, Specification Lead, November 7, 2006 http://jcp.org/aboutJava/communityprocess/final/jsr221/index.html	11-2006
Ullenboom	Java ist auch eine Insel http://www.galileocomputing.de/openbook/javainsel6 Sehr aktuell; absolut empfehlenswert	2007

Weitere Ressourcen:

http://www.oracle.com/technology/tech/java/sqlj_jdbc/index.html

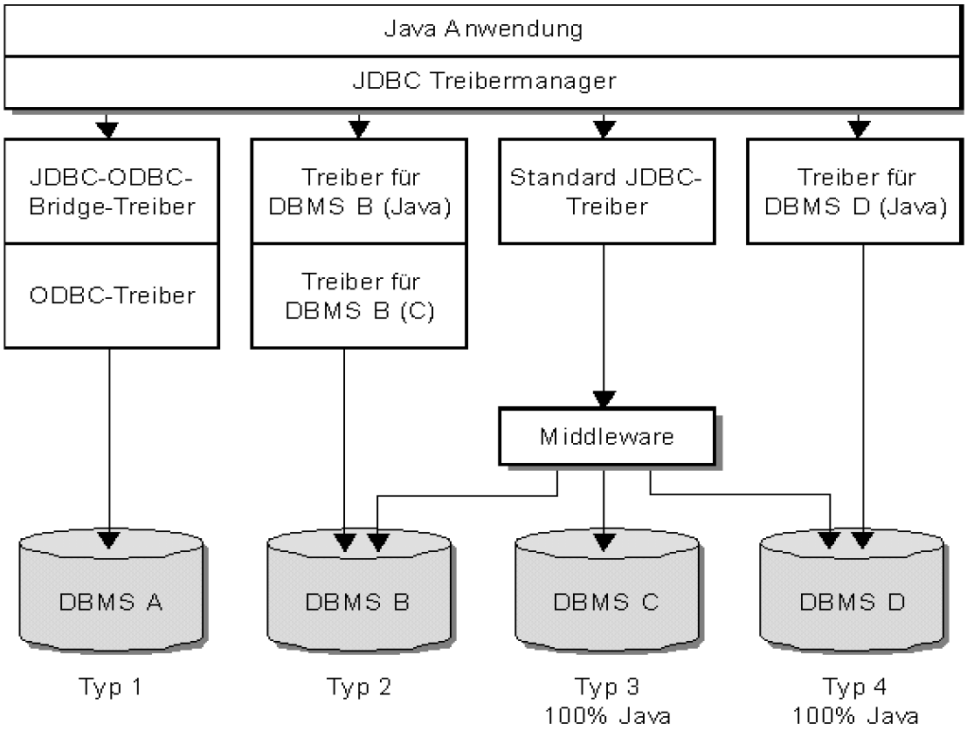
http://www.oracle.com/technology/hosted_doc/jdev/jdbc/overview-summary.html

http://www.onjava.com/pub/a/onjava/2001/12/19/oraclejdbc.html

http://www-db.stanford.edu/~ullman/fcdb/oracle/or-jdbc.html

1 Einsatz und Architektur

1.1 JDBC - Treiberkonzept



- Typ 1 (JDBC-ODBC-Brücke)
  - ✗ Zugriff auf andere Datenzugriffs-API z.B. ODBC (OpenDataBaseConnectivity)-Treiber z.B. der JDBC-ODBC Brückentreiber
  - ✗ Voraussetzung: Existenz eines ODBC-Treiber auf dem Clientrechner (native Bibliothek), daher eingeschränkte Portabilität
- Typ 2 (Native Treiber)
  - ✗ Sind tw. in Java und tw. in nativem Code geschrieben
  - ✗ Umwandlung in herstellerspezifische Anfragen (bei Oracle OCI)
  - ✗ Voraussetzung: Existenz nativer Bibliotheken auf dem Clientrechner, daher eingeschränkte Portabilität
- Typ 3
  - ✗ Benutzen eine Pure Java - Client
  - ✗ Kommunizieren mit einem Middleware-Server und benutzen ein datenbankunabhängiges Netzprotokoll.
  - ✗ Voraussetzung: Datenbankhersteller unterstützt dieses Protokoll.
- Typ 4 (Thin-Treiber)
  - ✗ Ein Pure Java – Treiber
  - ✗ benutzt ein Netzwerkprotokoll oder File I/O um mit spezifischen Datenquellen zu kommunizieren.
  - ✗ Vorteil: keine spezielle Software auf Clientseite nötig → Einsatz bei Applets

1.2 Datentypen

	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

TABLE B-5 Conversions Performed by `setObject` and `setNull` Between Java Object Types and Target JDBC Types

JDBC Typen --> Java Typen

JDBC Type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
BOOLEAN	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
CLOB	java.sql.Clob
BLOB	java.sql.Blob
ARRAY	java.sql.Array
DISTINCT	mapping of underlying type
STRUCT	java.sql.Struct
REF	java.sql.Ref
DATALINK	java.net.URL
JAVA_OBJECT	underlying Java class
ROWID	java.sql.RowId
NCHAR	String
NVARCHAR	String
LONGNVARCHAR	String
NCLOB	java.sql.NClob
SQLXML	java.sql.SQLXML

Java Typen --> JDBC Typen

Java Type	JDBC Type
String	CHAR, VARCHAR, LONGVARCHAR, NCHAR, NVARCHAR or LONGNVARCHAR
java.math.BigDecimal	NUMERIC
boolean	BIT or BOOLEAN
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[]	BINARY, VARBINARY, or LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.sql.Clob	CLOB
java.sql.Blob	BLOB
java.sql.Array	ARRAY
java.sql.Struct	STRUCT
java.sql.Ref	REF
java.net.URL	DATALINK
Java class	JAVA_OBJECT
java.sql.RowId	ROWID
java.sql.NClob	NCLOB
java.sql.SQLXML	SQLXML

2 Grundlegende Funktionen

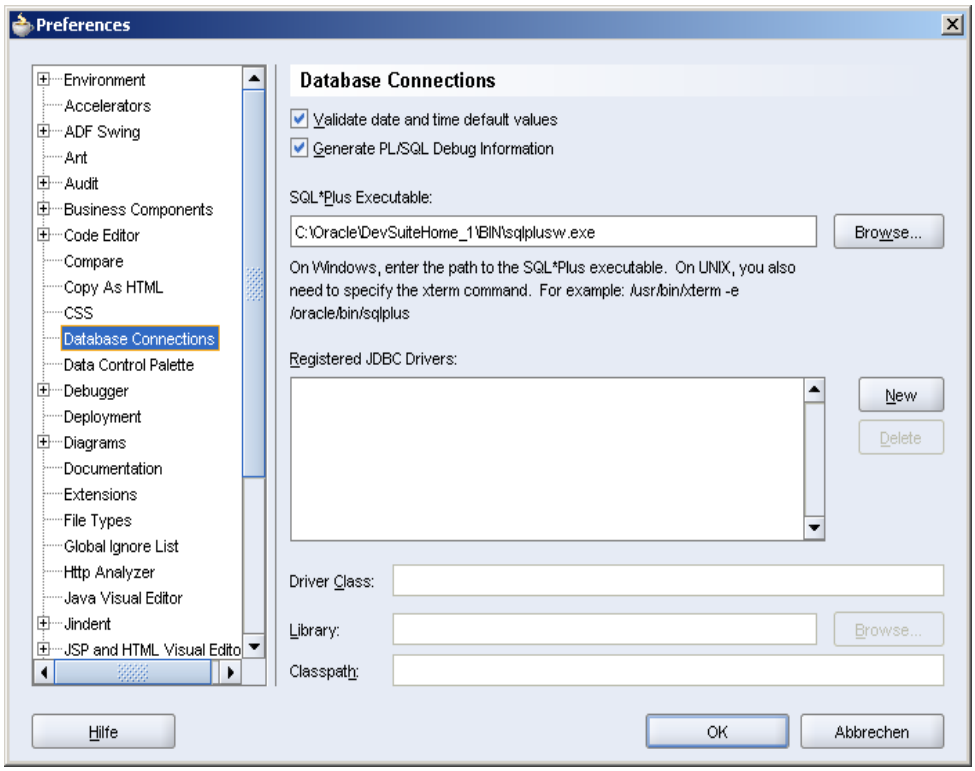
2.1 Voraussetzungen

2.1.1 SQL-Skripts in JDeveloper starten

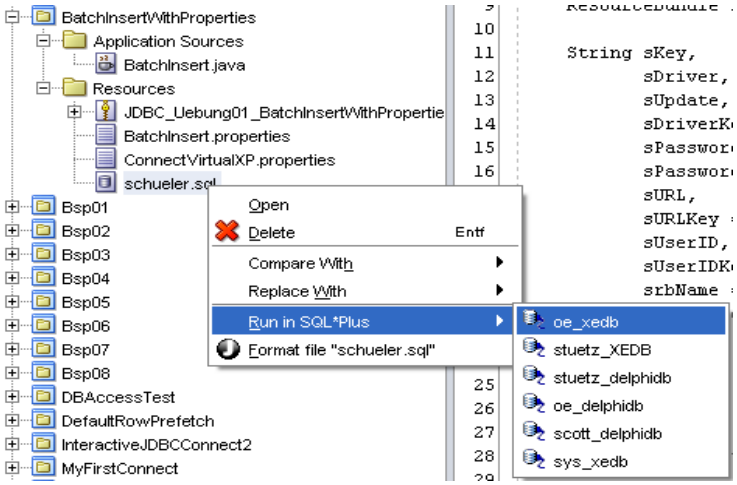
Man kann SQL*Plus aus der Jdeveloper-IDE aufrufen. Hierzu muss der Pfad einer auf dem System vorhandenen SQL*Plus – Installation (Instant-Client, Client, Oracle Forms) angegeben werden.

Tools – Preferences ... – Database Connections

mit Button „Browse ...“ den Pfad zu SQLPLUSW.EXE auswählen.



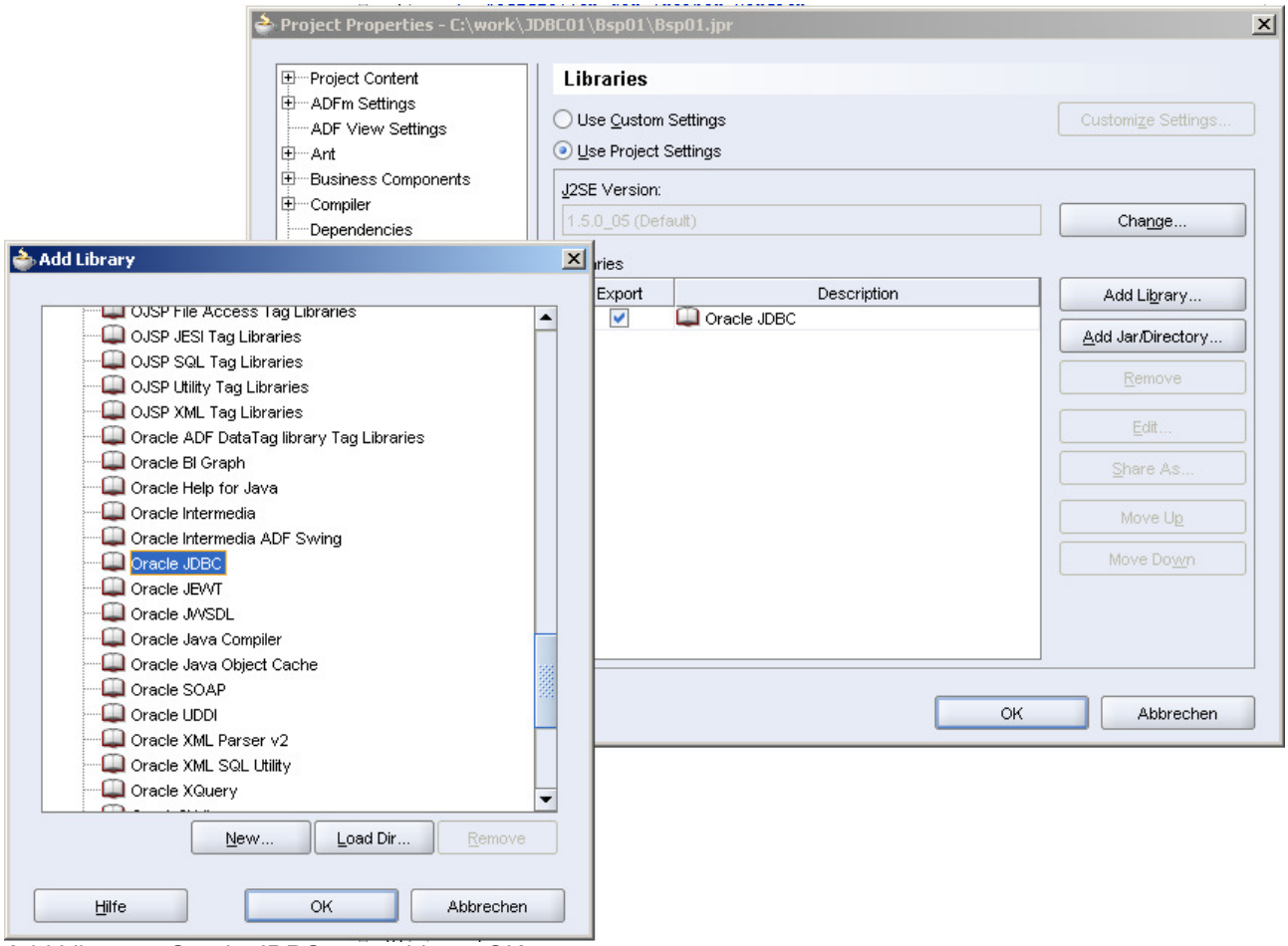
Nun kann ein vorhandenes SQL-Skript gestartet werden.



2.1.2 Verweis auf JDBC-Bibliothek

2 Möglichkeiten:

- Verweis auf `ojdbc14.jar` aus dem Projekt der IDE (z.B. im JDBC\LIB – Verzeichnis des Oracle – Clients.
- Hinzufügen der mitgelieferten Oracle JDBC – Library der JDeveloper-IDE



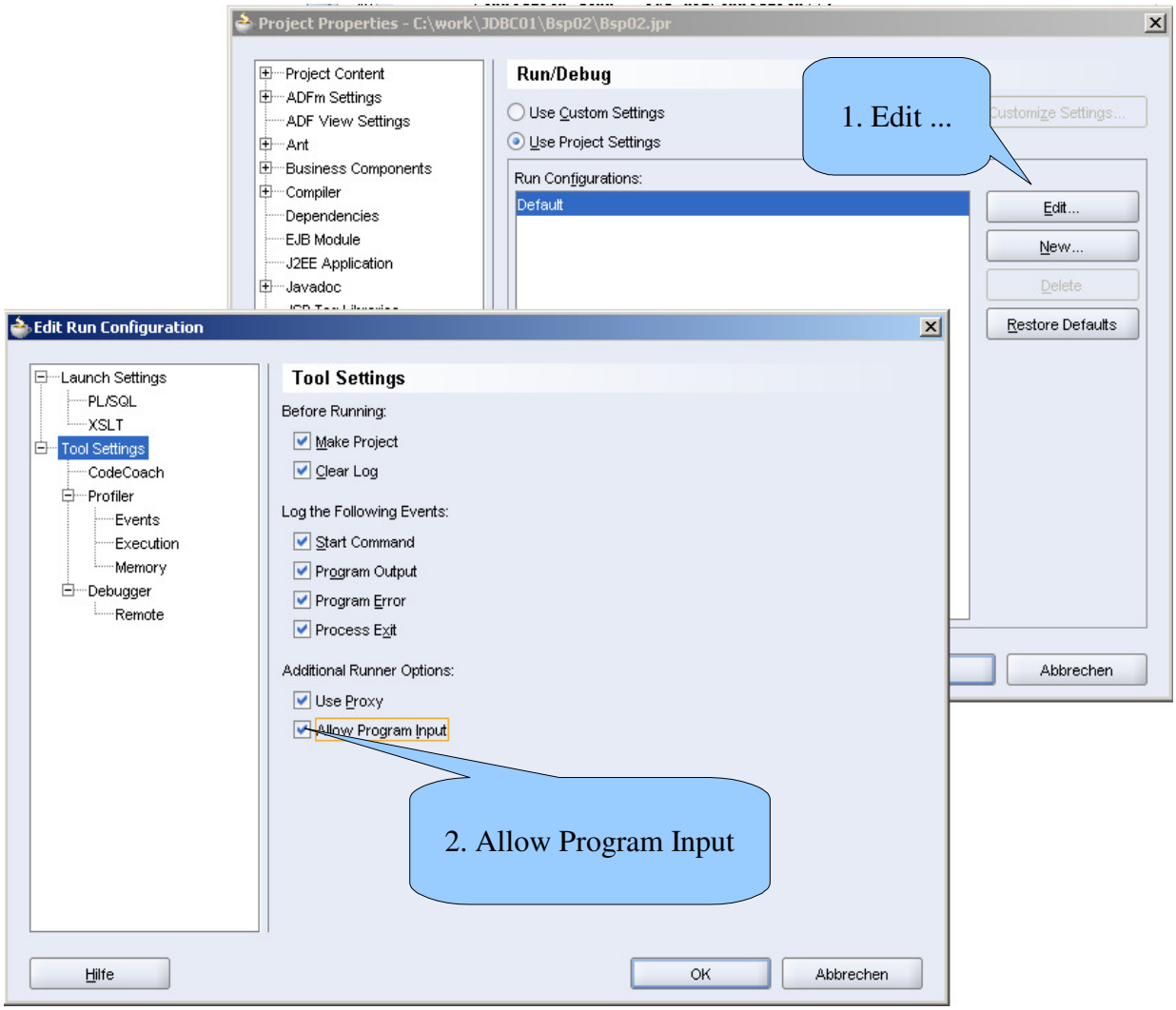
Add Library – Oracle JDBC auswählen – OK

2.1.3 JDeveloper – Benutzereingaben auf der Konsole

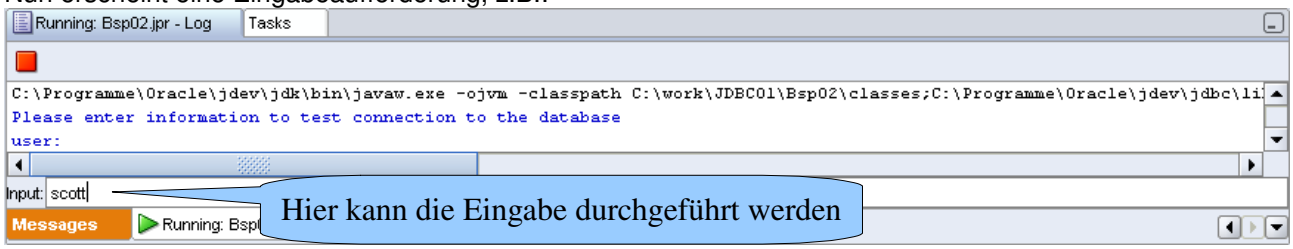
In JDeveloper muß zunächst eine Option gesetzt werden, um eine Eingabe in der Console zu ermöglichen[1]:

Version 10.1.3

Tools – Project Properties – Run/Debug – Edit ... – Allow Program Input



Nun erscheint eine Eingabeaufforderung, z.B.:



1 Vgl. http://forums.oracle.com/forums/thread.jspa?messageID=852894&#852894

2.2 Aufbau eines JDBC-Programms

2.2.1 Driver

Beim Einsatz einer Oracle-DB muss folgendes File in den CLASSPATH gestellt werden: `ojdbc14.jar`, `ojdbc5.jar` oder `ojdbc6.jar`

Dadurch werden die Packages verfügbar gemacht.

Import statement	Provides
`import java.sql.*;`	Standard JDBC packages
`import java.math.*;`	The `BigDecimal` and `BigInteger` classes. You can omit this package if you are not going to use these classes in your application.
`import oracle.jdbc.*;`	Oracle extensions to JDBC. This is optional.
`import oracle.jdbc.pool.*;`	`OracleDataSource`
`import oracle.sql.*;`	Oracle type extensions. This is optional.

Vgl. Oracle Database JDBC Developer's Guide and Reference

Vor dem Aufbau einer Connection, muss der JDBC-Treiber in das Programm geladen werden. Hierzu gibt es mehrere Möglichkeiten:

- mit der statischen Methode
  `Class.forName("Drivername");`
- oder mit dem Konstruktor
  `Class.ForName("Drivername").newInstance();`
- oder mit der statischen Methode
  `DriverManager.registerDriver( new Drivername() );`

Für Drivername kann bspw. Verwendet werden:

- `sun.jdbc.odbc.JdbcOdbcDriver`
- `oracle.jdbc.driver.OracleDriver`
- `com.mysql.jdbc.Driver`

2.2.1.1 *DriverManager*

`DriverManager` ist eine statische Klasse bei der der JDBC-Treiber der Datenbank registriert wird und mit der die statische Methode `getConnection` aufgerufen werden kann.

```
DriverManager.registerDriver(new OracleDriver());  // registrieren des Treibers
Connection conn = DriverManager.getConnection
  ("jdbc:oracle:thin:@hostname/databasename", username, password);
```

http://www.galileocomputing.de/openbook/javainsel7/javainsel_22_005.htm#mj730ba4edc8f42232f7127d9e51c2c803

siehe auch JDBC-Spec 3.0 und 4.0

[wird noch ergänzt]

Interessant ist die Möglichkeit ein Logging zu aktivieren:

```
...
DriverManager.registerDriver(new OracleDriver());
DriverManager.setLogWriter(new PrintWriter(System.out));
...
```

2.2.1.2 `DataSource`

Grundsätzlich wird von Sun empfohlen `DataSource` anstelle eines Treiber-Managers einzusetzen. Eine DataSource bietet mehr Möglichkeiten, u.a. in Bezug auf Connection Pooling.

Eine DataSource kann erstellt werden, indem

- eine herstellerabhängiger (vendor-specific) Treiber verwendet wird
- und/oder ein Namensdienst verwendet wird. Es muss kein Namensdienst verwendet werden.

Quelle: http://java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/datasource.html#998071

Ein DataSource-Objekt wird Verwendung eines Namensdienstes (z.B: openLDAP) eingesetzt. Bei Verbindung zu einer Oracle Datenbank wird empfohlen (auch wenn kein Namensdienst verwendet wird) DataSource-Objekte zu verwenden (siehe Beispiel im Kapitel „Connection – OracleDataSource“)

Beispiel für die JavaDB (Derby):

```
import org.apache.derby.client.*;
...
ClientDataSource dataSource = new ClientDataSource ();
dataSource.setServerName ("my_derby_database_server");
dataSource.setDatabaseName ("my_derby_database_name");
/*
//Binden an einen Namensdienst-Kontext
javax.naming.Context context = new javax.naming.InitialContext();
context.bind ("jdbc/my_datasource_name", dataSource);
*/
...
```

Quelle: http://db.apache.org/derby/javadoc/publishedapi/jdbc3/org/apache/derby/jdbc/ClientDataSource.html

[wird noch ergänzt]

2.2.2 Connection[1]

Die Klasse Connection dient zur Verbindung mit einer Datenbank. Die Treiber Methode `getConnection()` liefert ein Connection Objekt, welches folgende Aufgaben

hat:

- es ist verantwortlich für das Kreieren der Objekte:
  - Statement,
  - PreparedStatement und
  - CallableStatement (welches man im Zusammenhang mit Stored Procedures benötigt).
- es liefert die DatabaseMetadata Objekte.
- es kontrolliert Transaktionen mittels der commit()und der rollback() Methode.
- das Connection Objekt wird eingesetzt, um Isolation Levels in den Transaktionen zu setzen.

Erzeugt wird diese Verbindung vom JDBC DriverManager in der Form

```
Connection con = DriverManager.getConnection (URL, username, password);
```

1 Vgl. Partl, S.160 ff

Der URL hat eine der folgenden Formen:

- `jdbc:protocol:databasename`
  für eine lokale Datenbank mit JDBC-Driver
- `jdbc:protocol://hostname:port/databasename`
  für eine Datenbank auf einem anderen Rechner, mit JDBC-Driver
- `jdbc:odbc:datasourcename`
  für eine lokale ODBC-Datenbank, mit JDBC-ODBC-Driver
- `jdbc:odbc://hostname/datasourcename`
  für eine ODBC-Datenbank auf einem anderen Rechner, mit JDBC-ODBC-Driver

Der Username und das Passwort sind als String-Parameter anzugeben. Aus Sicherheitsgründen empfiehlt es sich, das Passwort nicht fix im Programm anzugeben sondern vom Benutzer zur Laufzeit eingeben zu lassen, am besten in einem TextField mit setEchoCharacter('*') oder in Swing mit einem JpasswordField. Eine gängige Variante besteht in der Verwendung von Property-Dateien [vgl. Kap. Batch-Connect]

Die wichtigsten Methoden der Klasse Connection sind:

- `createStatement();`
  erzeugt ein Statement für die Ausführung von SQL-Befehlen.
- `prepareStatement (String sql);`
  erzeugt ein Prepared Statement für die optimierte Ausführung von SQL-Befehlen.
- `prepareCall (String call);`
  erzeugt ein Callable Statement für die optimierte Ausführung einer in der Datenbank gespeicherten Stored Procedure.
- `DatabaseMetaData getMetaData()`
  liefert die DatabaseMetaData (DDL-Informationen) der Datenbank.
- `setReadOnly(true);`
  es werden nur Datenbank-Abfragen durchgeführt, der Datenbank-Inhalt wird nicht verändert, es sind daher keine Transaktionen notwendig und der Zugriff kann optimiert werden.
- `setReadOnly(false);`
  alle SQL-Statements sind möglich, also sowohl Abfragen als auch Updates (Default).
- `setAutoCommit(true);`
  jedes SQL-Statement wird als eine eigene Transaktion ausgeführt, commit() und rollback() sind nicht notwendig (Default).
- `setAutoCommit(false);`
  mehrere SQL-Statements müssen mit commit() und rollback() zu Transaktionen zusammengefasst werden.
- `Commit();`
  beendet eine Transaktion erfolgreich (alle Statements wurden ausgeführt).
- `Rollback();`
  beendet eine Transaktion im Fehlerfall (alle Änderungen seit Beginn der Transaktion werden rückgängig gemacht).
- `Close();`
  beendet die Verbindung mit der Datenbank.

Beispielsskizze:

```
String username="admin";
String password="geheim";
Connection conn = DriverManager.getConnection
  ("jdbc:odbc://hostname/databasename", username, password);
Statement stmt = con.createStatement();
conn.setReadOnly(true);
...
stmt.close();
conn.close();
```

Innerhalb einer Connection können mehrere Statements geöffnet werden, eventuell auch mehrere gleichzeitig.

2.2.3 Connection - OracleDataSource

Quelle: http://www.java2s.com/ExampleCode/Database-SQL-JDBC/OracleDataSourceDemo.htm

Für die Anbindung an Oracle-Datenbanken kann auch eine OracleDatasource erstellt werden.

```
/*
Java Programming with Oracle JDBC
by Donald Bales
ISBN: 059600088X
Publisher: O'Reilly
*/


import java.sql.*;
import oracle.jdbc.pool.*;

public class TestThinDSApp {

  public static void main(String args[]) throws ClassNotFoundException,
      SQLException {

    // These settings are typically configured in JNDI
    // so they a implementation specific
    OracleDataSource ds = new OracleDataSource();
    ds.setDriverType("thin");
    ds.setServerName("delphi.htl-leonding.ac.at");
    ds.setPortNumber(1521);
    ds.setDatabaseName("delphidb"); // sid
    ds.setUser("scott");
    ds.setPassword("tiger");

    Connection conn = ds.getConnection();

    Statement stmt = conn.createStatement();
    ResultSet rset = stmt
        .executeQuery("select 'Hello Thin driver data source tester '||"
            + "initcap(USER)||'!' result from dual");
    if (rset.next())
      System.out.println(rset.getString(1));
    rset.close();
    stmt.close();
    conn.close();
  }
}
```

2.2.4 Batch-Connect (Property-Dateien)

2.2.4.1 ResourceBundles

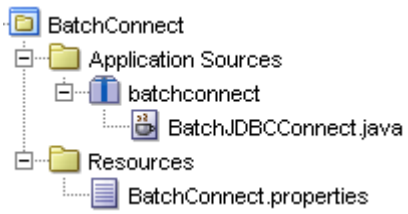
Quelle: Joller, S. 26ff

Wie schon besprochen, macht es kaum Sinn den Verbindungsaufbau fix zu programmieren.

Die generelle Technik mit diesem Problem umzugehen in Java ist der Einsatz von RessourceBundle und damit der Einsatz von Property Dateien oder ListResourceBundle.

"Batch," ist ein Begriff, den man aus den Mainframe Umgebungen kennt. "ein Programm läuft ohne Benutzerinteraktion", im Gegensatz zu interaktiven Programmen, bei denen der Benutzer Zusatzinfos eingeben muss.

Die Property-Datei ist eine Textdatei (z.B. `BatchConnect.properties`), die dem Projekt hinzugefügt wird:




```
#PropertiesResourceBundle für Connection Properties
CSDriver=oracle.jdbc.driver.OracleDriver
CSURL=jdbc:oracle:thin:@virtual-xp-prof:1521:xe
CSUserID=stuetz
CSPassword=passme
```

Der vollständige Code des folgenden Beispiels ist im Anhang „Metadaten einer Tabelle auslesen“ enthalten.

Beispiel: BatchConnect.java

```
package batchconnect;

import java.sql.*;
import java.util.*;

/**
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss GmbH
 * @author J.M.Joller
 * @version 1.0
 */
public class BatchJDBCConnect {
  Connection con;
  ResourceBundle rbConnect;
  ResultSet rs;
  ResultSetMetaData rsmd;
  Statement stmt;
  String sDriver,
         sDriverKey = "CSDriver",
         sPassword,
         sPasswordKey ="CSPassword",
         sQuery =
          "SELECT * FROM KaffeeListe " +
          "WHERE Kaffeesorte = 'MoJava'",
         srbName = "BatchConnect",
         sURL,
         sURLKey="CSURL",
         sUserID,
         sUserIDKey = "CSUserID";


  public BatchJDBCConnect()
  {
    try  // PropertyResourceBundle
    {
      rbConnect = ResourceBundle.getBundle( srbName );

      sDriver   = rbConnect.getString( sDriverKey );
      sPassword = rbConnect.getString( sPasswordKey );
      sURL      = rbConnect.getString( sURLKey );
      sUserID = rbConnect.getString( sUserIDKey );
    }
    catch( MissingResourceException mre )
    {
      System.err.println(
                  "ResourceBundle Problem " +
                   srbName + ", Programm wird abgebrochen." );
      System.err.println("Fehler: " + mre.getMessage() );
      return; // exit on error
    }

    try   //  JDBC Driver laden
    {     //  mit newInstance

    ...
    // JDBC - Operationen
    ...
}
```

2.2.4.2 Properties-Objekte

Können .properties Dateien im Plain-Text- und XML-Format einlesen und wieder schreiben.

Properties-Objekt (ab Java 6)

```
package at.stuetz.jdbc_demo;

import java.io.*;
```

```
import java.sql.*;
import java.util.*;

public class Mainclass {

    public static void main(String[] args) {
        try {
            String srbName="Connect.properties";
            Properties connProperties = new Properties();
            connProperties.load(new FileReader(srbName));
            //connProperties.load(new FileInputStream(srbName));
            connProperties.list(System.out);
            DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
            // ... JDBC stuff ...
        } catch (SQLException e) {
            System.out.println("ERROR: " + e.getMessage());
            // e.printStackTrace();
        } catch (FileNotFoundException e) {
            System.out.println("ERROR: " + e.getMessage());
            // e.printStackTrace();
        } catch (IOException e) {
            System.out.println("ERROR: " + e.getMessage());
            // e.printStackTrace();
        }
    }
}
```

Konsolenausgabe:

```
-- listing properties --
URL=jdbc:oracle:thin:@delphi:1521:delphidb
UserID=stuetz
Driver=oracle.jdbc.driver.OracleDriver
Password=passme
```

2.2.5 Statement

Die Klasse Statement dient zur Ausführung eines SQL-Statements oder von mehreren SQL-Statements nacheinander. Erzeugt wird dieses Statement von der Connection in der Form

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

Die wichtigsten Methoden der Klasse Statement sind:

- `ResultSet res = executeQuery( String sql );`
  führt ein SQL-Statement aus, das ein ResultSet als Ergebnis liefert, also z.B. ein SELECT-Statement.
- `int rowCount = executeUpdate( String sql );`
  führt ein SQL-Statement aus, das kein ResultSet liefert, also z.B. ein INSERT, UPDATE oder DELETE-Statement. Der Rückgabewert ist die Anahl der Records, für die der Befehl durchgeführt wurde, oder 0.
- `boolean isResultSet = execute( String sql );`
  führt ein SQL-Statement aus, das mehrere Ergebnisse (ResultSets oder UpdateCounts) liefern kann, die man dann mit den Methoden getResultSet, getUpdateCount und getMoreResults abarbeiten muss. Dies trifft nur in seltenen Spezialfällen zu.
- `setQueryTimeout ( int seconds );`
  setzt ein Zeitlimit für die Durchführung von Datenbankabfragen (in Sekunden).
- `Close();`
  beendet das Statement.

Beispiel für eine Abfrage:

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery
  ("SELECT ename, sal, job, deptno FROM emp");
...
rs.close();
stmt.close();
```

Beispiel für ein Update:

```
Statement stmt = con.createStatement();
int rowCount = stmt.executeUpdate
  ("UPDATE Employees" +
   "SET Salary = 5000.0 WHERE LastName = 'Partl' ");
...
stmt.close();
```

Innerhalb eines Statements darf zu jedem Zeitpunkt immer nur höchstens 1 ResultSet offen sein.

Beispiel für eine generische Funktion zur Abarbeitung von SQL-Statements[1]

```
public static void executeSQL(String sqlText) {
  Statement stmt = null;
  try {
    System.out.println("SQL> " + sqlText);
    stmt = conn.createStatement();
    if (stmt.execute(sqlText))  {
      /*
      ** When execute() returns a true
      ** A result set is available
      */
      ExecuteQuery.printResults(stmt.getResultSet());
    }
    else {
      /*
      ** When execute() returns a false
      ** An update count is available
      */
      System.out.println("" + stmt.getUpdateCount() + " row(s) altered");
    }
  }
  catch (Exception e) { e.printStackTrace(); }
  finally {
    try { stmt.close();
          stmt = null;
        } catch (Exception e) {}
  }
}
```

2.2.6 ResultSet

Ein ResultSet ist das Ergebnis einer SQL-Abfrage , im Allgemeinen das Ergebnis eines SELECT-Statements. Es enthält einen oder mehrere Records von Datenfeldern und bietet Methoden, um diese Datenfelder ins Java-Programm hereinzuholen.

Die wichtigsten Methoden der Klasse ResultSet sind:

- `boolean next()`
  setzt den "Cursor" auf den ersten bzw. nächsten Record innerhalb des ResultSet. Dessen Datenfelder können dann mit den getXXX-Methoden angesprochen werden.
- `boolean getBoolean( String name )`
  `int getInt ( String name )`
  `float getFloat ( String name )`
  `double getDouble ( String name )`
  `String getString ( String name )`
  usw. liefert den Wert des Datenfeldes mit dem angegebenen Namen (case-insensitive), wenn der Name in der Datenbank eindeutig ist.
- `boolean getBoolean( int n )`
  `int getInt ( int n )`
  `float getFloat ( int n )`
  `double getDouble ( int n )`
  `String getString ( int n )`
  usw. liefert den Wert des n-ten Datenfeldes im Ergebnis (von 1, nicht von 0 an gezählt); wenn man im SELECT-Befehl Feldnamen angegeben hat, dann in der angegebenen Reihenfolge; wenn man im SELECT-Befehl * angegeben hat, dann in der in der Datenbank definierten Reihenfolge.

1 Vgl. Bonazzi,Stokol S.425

- `java.sql.Date getDate ( String name )`
  `Time getTime ( String name )`
  `Timestamp getTimestamp ( String name )`
  `java.sql.Date getDate ( int n )`
  usw. liefert analog ein Datums- bzw. Zeit-Objekt. Dabei ist zu beachten, dass die Klasse java.sql.Date verschieden von java.util.Date ist, es gibt in diesen Klassen aber Methoden, um das eine Datum in das andere umzuwandeln (am einfachsten mit getDate und setDate).
- `boolean wasNull()`
  gibt an, ob das zuletzt gelesene Datenfeld leer war (d.h. den SQL-Nullwert enthielt).
- `close()`
  beendet die Abfrage bzw. schließt das ResultSet.
- `ResultSetMetaData getMetaData()`
  liefert die ResultSetMetaData (DDL-Informationen) zu diesem ResultSet.

Beispielskizze:

```
ResultSet rs = stmt.executeQuery
  ("SELECT LastName, Salary, Age, Sex FROM Employees");
System.out.println("List of all employees:");
while (rs.next()) {
  System.out.print(" name=" + rs.getString(1) );
  System.out.print(" salary=" + rs.getDouble(2) );
  System.out.print(" age=" + rs.getInt(3) );
  if ( rs.getBoolean(4) ) System.out.print(" sex=M");
  else System.out.print(" sex=F");
  System.out.println();
}
rs.close();
```

Anmerkungen:

Innerhalb eines Statements darf zu jedem Zeitpunkt immer nur höchstens 1 ResultSet offen sein. Wenn man mehrere ResultSets gleichzeitig braucht, muss man dafür mehrere Statements innerhalb der Connection öffnen (sofern das Datenbanksystem das erlaubt). Die Zugriffe auf die Felder sollen in der Reihenfolge erfolgen, wie sie von der Datenbank geliefert werden, also in der Reihenfolge, in der sie im SELECT-Befehl bzw. bei * im Record stehen. Bei JDBC 1.x können die Records auch nur in der Reihenfolge, in der sie von der Datenbank geliefert werden, verarbeitet werden (mit next).

Ab JDBC 2.0 enthält die Klasse ResultSet zahlreiche weitere Methoden, die auch ein mehrmaliges Lesen der Records (z.B. mit previous) und auch Datenänderungen in einzelnen Records (z.B. mit updateString, updateInt) erlauben.

Ansonsten kann man ein mehrmaliges Abarbeiten der Resultate erreichen, indem man sie in einer Liste oder einem Vector zwischenspeichert.

Beispielskizze:

```
ResultSet rs = stmt.getResultSet();
ResultSetMetaData md = rs.getMetaData();
int numberOfColumns = md.getColumnCount();
int numberOfRows = 0;
Vector rows = new Vector();
while (rs.next()) {
  numberOfRows++;
  Vector newRow = new Vector();
  for (int i=1; i<=numberOfColumns; i++) {
    newRow.addElement(rs.getString(i));
  }
  rows.addElement (newRow);
}
rs.close();
```

2.2.6.1 Scrollability

ab JDBC 2.0

Die Fähigkeit sich sowohl vor als auch zurück durch ein Result Set zu bewegen.

2.2.6.2 Positioning

ab JDBC 2.0

Man unterscheidet zwischen

- relativer Positionierung, und
- absoluter Positionierung

Die relative Positionierung ermöglicht es, sich eine gewisse Anzahl von Datensätzen im Result Set vor oder zurück von der momentanen Position zu bewegen.

Mittels der absoluten Positionierung kann man sich zu einer bestimmten Position (Zeilenummer) im Result Set bewegen.

2.2.6.3 Sensitivity

Muss bei scrollbaren bzw. positionierbaren Result Sets festgelegt werden. Darunter versteht man die Fähigkeit eines Result Sets festzustellen, ob und wo Änderungen der darunterliegenden Datenbank von außerhalb des Result Sets durchgeführt wurden.

2.2.7 Prepared Statement[1]

Wenn man viele Updates nacheinander ausführt und dazu jedesmal mit `executeUpdate` einen kompletten `INSERT`- oder `UPDATE`-SQL-Befehl an die Datenbank sendet, muss jedesmal wieder der SQL-Befehl interpretiert und dann ausgeführt werden. Bei einer großen Anzahl von ähnlichen Updates kann dies sehr viel unnötige Rechenzeit in Anspruch nehmen.

Um den Update-Vorgang zu beschleunigen, kann man in diesem Fall mit der Connection-Methode `prepareStatement` ein Muster für den SQL-Befehl an das Datenbanksystem senden, in dem die variablen Datenfelder mit Fragezeichen gekennzeichnet sind, und dann mit den Methoden der Klasse `PreparedStatement` nur mehr die Daten in diese vorbereiteten SQL-Statements "einfüllen".

Beispielskizze:

```
con.setAutoCommit(false);
PreparedStatement ps = con.prepareStatement
  ("UPDATE emp SET sal = ? WHERE ename = ? ");
for (int i=0; i<goodPerson.length; i++) {
  ps.setFloat (1, newSalary[i] );
  ps.setString (2, goodPerson[i] );
  ps.executeUpdate();
}
con.commit();
con.close();
```

2.3 Erste Schritte

2.3.1 Testen der JDBC-Verbindung

Oracle stellt folgendes Code zum Testen einer JDBC-Verbindung zur Verfügung (JdbcCheckup.java):

```
/*
* This sample can be used to check the JDBC installation.
* Just run it and provide the connect information. It will select
* "Hello World" from the database.
*/

// You need to import the java.sql and JDBC packages to use JDBC
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;

// We import java.io to be able to read from the command line
import java.io.*;

class JdbcCheckup {
    public static void main(String[] args) throws SQLException, IOException {
```

1 Vgl. Partl, S. 163 f

```
            // Prompt the user for connect information
            System.out.println("Please enter information to test connection"
                               + " to the database");
            String user;
            String password;
            String database;

            user = readEntry("user: ");
            int slash_index = user.indexOf('/');
            if (slash_index != -1) {
                password = user.substring(slash_index + 1);
                user = user.substring(0, slash_index);
            } else
                password = readEntry("password: ");
            database = readEntry("database(a TNSNAME entry): ");
            System.out.print("Connecting to the database...");
            System.out.flush();
            System.out.println("Connecting...");
            // Open an OracleDataSource and get a connection
            OracleDataSource ods = new OracleDataSource();
            ods.setURL("jdbc:oracle:oci:@" + database);
            ods.setUser(user);
            ods.setPassword(password);
            Connection conn = ods.getConnection();
            System.out.println("connected.");

            // Create a statement
            Statement stmt = conn.createStatement();

            // Do the SQL "Hello World" thing
            ResultSet rset = stmt.executeQuery("select 'Hello World' from dual");

            while (rset.next())
                System.out.println(rset.getString(1));
            // close the result set, the statement and connect
            rset.close();
            stmt.close();
            conn.close();
            System.out.println("Your JDBC installation is correct.");
        }

        // Utility function to read a line from standard input
        static String readEntry(String prompt) {
            try {
                StringBuffer buffer = new StringBuffer();
                System.out.print(prompt);
                System.out.flush();
                int c = System.in.read();
                while (c != '\n' && c != -1) {
                    buffer.append((char)c);
                    c = System.in.read();
                }
                return buffer.toString().trim();
            } catch (IOException e) {
                return "";
            }
        }
    }
```

Konsolenausgabe:

```
Please enter information to test connection to the database
user: scott
password: tiger
database(a TNSNAME entry): delphidb
Connecting to the database...Connecting...
connected.
Hello World
Your JDBC installation is correct.
Process exited with exit code 0.
```

Anmerkung:

Hier wird der OCI-Treiber ein in C implementierter Treiber (Typ 2 - Treiber) verwendet.

2.4 Lesender Zugriff auf DB

2.4.1 Auslesen eines Datensatzes aus der DB

Beispiel: Bsp03.java

```
//~--- JDK imports ------------------------------------------------------------
import java.sql.*;
import oracle.jdbc.*;
import java.io.*;

//~--- classes ----------------------------------------------------------------
public class Bsp03 {
    private String driver   = "oracle.jdbc.driver.OracleDriver";
    private String url      = "jdbc:oracle:thin:@virtual-xp-prof:1521:xe";
    private String user     = "stuetz";
    private String password = "passme";

    private Connection conn;

    //~--- constructors -------------------------------------------------------
    public Bsp03() {
         try {
             Class.forName(driver);
             conn = DriverManager.getConnection(url, user, password);
         } catch (Exception e) {
             System.err.println(e);
             System.exit(1);
         }
    }

    //~--- methods ------------------------------------------------------------
    public void getEmp() {
        try {
            int empno;

            BufferedReader br =
                  new BufferedReader(new InputStreamReader(System.in));
            System.out.print("Geben Sie eine EMPNO ein: ");
            empno = Integer.parseInt(br.readLine());

            String sql =
              "SELECT empno, ename, deptno FROM emp WHERE empno = " + empno;
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql);
            while (rs.next())  {
                System.out.print(rs.getInt(1) + " ");
                System.out.print(rs.getString(2) + " ");
                System.out.print(rs.getString("deptno") + " ");
                System.out.print(System.getProperty("line.separator"));
            }
            rs.close();
            stmt.close();
            conn.close();        } catch (SQLException e) {
            System.err.println("SQL-Fehler");
            System.err.println(e);
            System.exit(1);
        } catch (Exception e) {
            System.err.println(" Allgemeiner Fehler");
            System.err.println(e);
            System.exit(1);
        }

    }

    public static void main(String[] args) {
        Bsp03 bsp03 = new Bsp03();
        bsp03.getEmp();
    }
```

```
}
```

Konsolenausgabe:

```
Geben Sie eine EMPNO ein: 7499
7499 ALLEN      30
Process exited with exit code 0.
```

Erläuterung:

- `import java.sql.*;`

  macht das Package verfügbar.

- `conn = DriverManager.getConnection(url, user, password);`

  baut die Verbindung zu einer Datenbank auf.

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

  oder `PreparedStatement ...`

  gibt an, in welcher Form die SQL-Befehle zur Datenbank gesendet werden.

- `ResultSet rs = stmt.executeQuery (sql);`
- oder `int n = stmt.executeUpdate (sql);`

  führt einen SQL-Befehl aus (im ersten Fall eine Abfrage, im zweiten Fall eine Datenveränderung).

- Auf die Spalten kann entweder mit dem Index (`rs.getString(2)`) oder mir dem Spaltennamen (`rs.getString("deptno")`) zugegriffen werden. Die Verwendung der Indizess ist allerdings zu empfehlen, da performanter.

2.4.2 Prepared Statements und Parameter

Beispiel: Bsp04.java

```
...
try {
  int empno;

  ...

  String sql = "SELECT empno, ename, sal, deptno FROM emp WHERE empno = ?";
  PreparedStatement stmt = conn.prepareStatement(sql);
  stmt.setInt(1,empno); // dem ersten Parameter den Wert der
                        // Variablem empno zuweisen
  ResultSet rs = stmt.executeQuery();
  while (rs.next())  {
    System.out.print(rs.getString(1) + " ");        // empno
    System.out.print(rs.getString(2) + " ");        // ename
    System.out.print(rs.getString(3) + " EUR ");    // sal
    System.out.print(rs.getString("deptno") + " ");
    System.out.print(System.getProperty("line.separator"));
  }
  ...
  conn.close();
} catch (SQLException e) {
  ...
} catch (Exception e) {
  ...;
}
```

2.4.3 Anzahl der Zeilen in der Ergebnismenge

Quelle: http://www.dpunkt.de/java/Programmieren_mit_Java/Java_Database_Connectivity/33.html

Viele Anwendungen, wie z. B. Suchmaschinen, führen Recherchen in Datenbanken durch, die unter Umständen sehr viele Ergebniszeilen zurückliefern. Meist werden bei solchen Anwendungen nicht sofort alle Treffer

angezeigt, sondern es wird lediglich ein Ausschnitt aus der Ergebnismenge sowie die Gesamttrefferzahl dargestellt.

JDBC stellt keine vordefinierte Methode für die Ermittlung der Anzahl der Zeilen in einer Ergebnismenge bereit. Die Anzahl an Ergebniszeilen wird nicht im Cursor gespeichert, da die Zeilen nach Bedarf aus der Datenbank abgerufen werden. Dadurch steht sie erst endgültig fest, wenn der Cursor alle Zeilen aus der Datenbank zurückgeliefert hat.

Die Ermittlung der Gesamttrefferzahl einer Abfrage kann man auf drei Arten durchführen, abhängig von der verfügbaren JDBC-Version. Die einfachste, mit allen JDBC-Varianten kompatible Möglichkeit besteht darin, zunächst eine Abfrage auszuführen, in der nur die Anzahl der Treffer ermittelt wird. Anschließend werden in einer zweiten Anfrage die Daten selektiert:

```
...
Statement stmt = conn.creatStatement();
// 1. Abfrage: Anzahl Zeilen ermitteln
ResultSet rs = stmt.executeQuery(
                  "SELECT count(*) FROM titel");
rs.next();
long zeilen = rs.getLong(1);
rs.close();
// 2. Abfrage: Daten-Ausschnitt ausgeben
rs = stmt.executeQuery("SELECT * FROM titel t ORDER BY t.titel");
// Zugriff auf die Daten
...
```

Bei der zweiten Alternative werden die Positionierungs-Funktionen von JDBC-2.0 verwendet. Hierbei muss man den Cursor zunächst mit der Methode last() auf den letzten Datensatz positionieren und anschließend über die Methode getRow() die Nummer der Zeile in der Ergebnismenge ermitteln. Über die Methode first() kann das ResultSet anschließend wieder auf den ersten Satz positioniert werden:

```
// Initialisierung des Statement-Objekts
Statement stmt;
stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                            ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * FROM titel t ORDER BY t.titel");
// Positionierung auf den letzten Satz und Abfrage der Zeilen
rs.last();
long zeilen = rs.getRow());
// Erneute Positionierung auf den ersten Satz
rs.first();
// Zugriff auf die Daten
...
```

Als Typ der Ergebnismenge wurde TYPE_SCROLL_INSENSITIVE gewählt, da eine Positionierung für die Ermittlung der Zeilenanzahl möglich sein muss.

Als dritte Alternative kann man die Datensätze im Anwendungscode in einer while-Schleife selbst durchzählen. Das ist die effizienteste Methode, wenn man neben dem Zählen alle Daten in der Ergebnismenge auch sofort verarbeiten will:

```
ResultSet rs;
...
int count = 0;
while(rs.next()) {
  count++; // Zeilen-Zähler erhöhen
  // Hier Daten verarbeiten
}
...
```

Da eine positionierbare Ergebnismenge mehr Ressourcen als ein Standard-Cursor verbraucht, ist bei großen Datenmengen die erste Methode meist schneller, obwohl zwei SQL-Abfragen benötigt werden.

2.4.4 NULL-Check

Quelle: http://www.dpunkt.de/java/Programmieren_mit_Java/Java_Database_Connectivity/32.html#LABQUERYNULL

Ruft man die Daten über eine getXXX()-Methode ab, die das Ergebnis in einem primitiven Rückgabewert zurückliefert, wird vom Treiber ein Default-Wert zurückgegeben, falls in der Datenbank ein NULL-Wert enthalten ist. Bei der Abfrage von Zahlenwerten ist dieser Default-Wert 0, bei booleschen Typen false. Das heißt, beim Aufruf dieser Methoden kann man nicht unterscheiden, ob der Wert wirklich in der Datenbank gespeichert ist, oder ob es sich um einen vom Treiber generierten Default-Wert handelt, der als Ersatz für einen NULL-Wert zurückgegeben wird. Man kann jedoch auf folgende Arten herausfinden, ob in der Datenbank tatsächlich ein NULL-Wert gespeichert ist:

- Aufruf einer kompatiblen getXXX()-Methode, die den gespeicherten Wert als Verweistyp zurückliefert.
  Die Methode getObject() liefert für jeden Datentyp null zurück, falls es sich in der Datenbank um einen NULL-Wert handelt.
- Aufruf der Methode wasNull().
  Diese Methode hat keine Parameter und liefert true, wenn der zuletzt über eine getXXX() zurückgelieferte Wert in der Datenbank einen NULL-Wert hatte.

Folgendes Beispiel zeigt, wie bei einer Abfrage geprüft wird, ob ein gültiger Preis verfügbar ist:

```
float preis;
while(rs.next()) {
  System.out.println(rs.getString("titel"));
  preis = rs.getFloat("preis");
  // Ist der Preis in der Datenbank NULL?
  if(rs.wasNull())
    System.out.println("Noch kein Preis verfügbar !");
  else
    System.out.println("Preis: "+preis);
}
```

2.4.5 Rückgabe einer Ergebnismenge aufgrund eines NULL-Wertes

Quelle: Oracle Database JDBC Developer's Guide and Reference, S. 13-4

You cannot use a relational operator to compare NULL values with each other or with other values. For example, the following SELECT statement does not return any row even if the COMM column contains one or more NULL values.

```
PreparedStatement pstmt = conn.prepareStatement(
  "SELECT * FROM EMP WHERE COMM = ?");
pstmt.setNull(1, java.sql.Types.VARCHAR);
```

The next example shows how to compare values for equality when some return values might be NULL. The following code returns all the ENAMES from the EMP table that are NULL, if there is no value of 100 for COMM.

```
PreparedStatement pstmt = conn.prepareStatement("SELECT ENAME FROM EMP
  WHERE COMM =? OR ((COMM IS NULL) AND (? IS NULL))");
pstmt.setBigDecimal(1, new BigDecimal(100));
pstmt.setNull(2, java.sql.Types.VARCHAR);
```

2.5 Datenänderungen (DML und DDL)

2.5.1 Einzelne Änderungen (DELETE, UPDATE, INSERT)

Beispiel: Bsp05.java

```
...
try {
  String ename = "JONES";
  String sql = "UPDATE emp SET sal = 3000 WHERE ename = '" + ename +"'";
  Statement stmt = conn.createStatement();
  int rowCount = stmt.executeUpdate(sql);
  System.out.println(
  rowCount + " Gehaltserhöhung durchgeführt.");
  stmt.close();
  conn.close();
} catch (SQLException e) {
...
```

An die Variable rowCount wird bei DML-Statements die Anzahl der geänderten Zeilen zurückgegeben. Bei DDL-Statements ist der Wert immer NULL.

Mehrere Zeilen können wie folgt geändert werden.

Beispiel: Bsp06.jpr

```
...
Statement stmt = conn.createStatement();
int rowCount = 0;
for (int i = 0; i < goodPerson.size(); i++)  {
  Employee e = (Employee)goodPerson.get(i);
  rowCount = rowCount + stmt.executeUpdate
   ("UPDATE emp SET sal = " + e.getSal()*1.1 +
    " WHERE ename = '" + e.getEname() +"'");
}

System.out.println(
rowCount + " Gehaltserhöhungen durchgeführt.");
stmt.close();
...
```

2.5.2 Batch Updates

Quelle: Frischalowski, Böttcher: Java 6 Programmierhandbuch, entwickler press 2007, S. 966

Beispiel:

```
import java.sql.*;
public class BatchMode
{
  public BatchMode()
  {
    try
    {
      Class.forName("com.mysql.jdbc.Driver");
    }
    catch(ClassNotFoundException cnfEx)
    {
```

```
      System.exit(1);
    }
    try
    {
      Connection verbMySQL = DriverManager.getConnection(
        "jdbc:mysql://localhost/Kunden?user=root&password=mysql");
      Statement stmtMySQL = verbMySQL.createStatement();
      stmtMySQL.addBatch("CREATE TABLE Artikel(ID INTEGER, Name"+
                         " VARCHAR(30), PRIMARY KEY(ID))");
      stmtMySQL.addBatch("INSERT INTO Artikel VALUES(1, 'Schrauben')");
      stmtMySQL.addBatch("INSERT INTO Artikel VALUES(1, 'Muttern')");
      stmtMySQL.addBatch("INSERT INTO Artikel VALUES(2, 'Nägel')");
      stmtMySQL.addBatch("INSERT INTO Artikel VALUES(3, 'Dübel')");
      try
      {
        int resultate[] = stmtMySQL.executeBatch();
      }
      catch(BatchUpdateException buEx)
      {
        System.out.println("Batch-Fehler");
        int resultate[] = buEx.getUpdateCounts();
        for(int anzahl: resultate)
        {
          if(anzahl == Statement.EXECUTE_FAILED)
            System.out.println("Fehler");
          else
            System.out.println(anzahl);
        }
      }
    }
    catch(SQLException sqlEx)
    {
      System.out.println("SQL-Fehler");
    }
  }
  public static void main(String args[])
  { new BatchMode(); }
}
```

2.5.3 Bulk Updates mit Prepared Statements

```
import java.sql.*;
...
try {
  Class.forName("oracle.jdbc.driver.OracleDriver");
  String username="user";
  String password="geheim";
  Connection conn= DriverManager.getConnection
      ("jdbc:oracle:thin:@localhost:1521:xedb",username,password);
  con.setAutoCommit(false);
  Employee emp;
  String sql = "UPDATE emp SET sal = ? WHERE ename = ?";
  PreparedStatement pstmt = conn.prepareStatement(sql);
  for (int i = 0; i < goodPerson.size(); i++)  {
    emp = (Employee)goodPerson.get(i);
    pstmt.setDouble(1,emp.getSal());
    pstmt.setString(2,emp.getEname());
```

```
        pstmt.executeUpdate();
      }
      pstmt.close();
      conn.commit();
      conn.close();
    } catch ...
```

2.5.4 Ermitteln von automatisch generierten Schlüsseln

Beim Einfügen eines Datensatzes in eine Tabelle werden die Primärschlüsselwerte oft automatisch vergeben. Entweder durch Autowert-Felder (wie bei MySQL oder MS-SQL-Server) aber auch Verwendung von Sequenzen (wie bei Oracle). Für den Anwendungsprogrammierer ist es in diesen Fällen jedoch vorteilhaft den automatisch generierten Schlüssel sofort auslesen zu können.

Bsp. (siehe im Anhang für den vollständigen Code)

Folgende Tabelle steht zur Verfügung

```
CREATE TABLE person(
    id INTEGER PRIMARY KEY
,   name VARCHAR2(25)
,   loc VARCHAR2(25)
);
```

Das Feld ID wird durch einen Trigger und eine Sequenz automatisch befüllt.

Programmcode (ohne Ausnahmebehandlung)

```
...
Connection conn = // Connection herstellen
String sql ="INSERT INTO PERSON (name, loc) VALUES ('edi','leonding')";
String generatedColumns[]={"ID"};
PreparedStatement pstmt = conn.prepareStatement(sql,generatedColumns);
pstmt.executeUpdate();
ResultSet rs = pstmt.getGeneratedKeys();
if (rs != null) {
  if (rs.next()) {
    BigDecimal generatedKey = rs.getBigDecimal(1);
    System.out.println("Generierter Key: " + generatedKey);
  }
}
...
```

Beim Vorbereiten des preparedStatements wird zusätzlich zum SQL-Statement noch ein String-Array mit den Bezeichnungen der Spalten (für die die Werte automatisch generiert werden) übergeben.

ACHTUNG: Es muss als DB mind 10.2 verwendet werden und als JDBC-Treiber ebenfalls 10.2

2.6 Scrollable Result Sets

Quelle: http://de.wikibooks.org/wiki/Java_Standard:_JDBC

(ab JDBC 2.0)

2.6.1 Positionieren in einem Scrollable Result Set

Eine komfortable Datensatznavigation ermöglichen folgende Connection-Methoden:

- `Statement createStatement(int resultSetType, int resultSetConcurrency);`
- `Statement createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability);`

mit

`resultSetType:`

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

`resultSetConcurrency:`

- `ResultSet.CONCUR_READ_ONLY`
- `ResultSet.CONCUR_UPDATABLE`

`resultSetHoldability:`

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

Die genaue Bedeutung dieser Parameter soll hier nicht erläutert werden. Für ein scrollbares ResultSet ist der Parameter resultSetType auf ResultSet.TYPE_SCROLL_INSENSITIVE oder ResultSet.TYPE_SCROLL_SENSITIVE zu setzen.

Schon kann der Datensatzcursor fast beliebig positioniert werden:

`next()`	Bewegt den Datensatzcursor zum nächsten Datensatz
`previous()`	Bewegt den Datensatzcursor zum vorherigen Datensatz
`first()`	Bewegt den Datensatzcursor zum ersten Datensatz
`last()`	Bewegt den Datensatzcursor zum letzten Datensatz
`afterLast()`	Bewegt den Datensatzcursor hinter den letzten Datensatz
`beforeFirst()`	Bewegt den Datensatzcursor vor den ersten Datensatz
`absolute(int n)`	Bewegt den Datensatzcursor auf den n-ten Datensatz
`relative(int n)`	Bewegt den Datensatzcursor relativ zur momentanen Position

Beispiel: Bsp08.java

```
import java.sql.*;
import java.util.*;

...
        try {
            Class.forName(driver);

            Connection conn = DriverManager.getConnection(url, user, password);
            Statement  stmt =
                conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
            ResultSet rs = stmt.executeQuery("SELECT * FROM leute");

            // Erster Datensatz
            rs.next();
            System.out.println(rs.getInt("id") + "\t"
                               + rs.getString("vorname") + "\t"
                               + rs.getString("nachname") + "\t"
                               + rs.getDate(4));

            // Letzter Datensatz
            rs.afterLast();
            rs.previous();
            System.out.println(rs.getInt("id") + "\t"
                               + rs.getString("vorname") + "\t"
                               + rs.getString("nachname") + "\t"
                               + rs.getDate(4));

            // 2. Datensatz
            rs.absolute(2);
            System.out.println(rs.getInt("id") + "\t"
                               + rs.getString("vorname") + "\t"
                               + rs.getString("nachname") + "\t"
                               + rs.getDate(4));

            // 1. Datensatz
            rs.relative(-1);
```

```
                System.out.println(rs.getInt("id") + "\t"
                                   + rs.getString("vorname") + "\t"
                                   + rs.getString("nachname") + "\t"
                                   + rs.getDate(4));

                // Letzter Datensatz
                rs.absolute(-1);
                System.out.println(rs.getInt("id") + "\t"
                                   + rs.getString("vorname") + "\t"
                                   + rs.getString("nachname") + "\t"
                                   + rs.getDate(4));
                System.out.println("Tuple = " + rs.getRow());
                rs.close();
                stmt.close();
                conn.close();
            } catch (ClassNotFoundException e) {
                System.err.println("Class Error");
                e.printStackTrace();
            } catch (SQLException e) {
                System.err.println("SQL Error");
                e.printStackTrace();
            } catch (Exception e) {
                e.printStackTrace();
                System.exit(1);
            }
        }
    }
```

Konsolenausgabe:

```
1 Hansi  Hubsi  1999-12-01
4 Mili   Miker  1948-11-08
2 Willy  Wupp   1964-02-22
1 Hansi  Hubsi  1999-12-01
4 Mili   Miker  1948-11-08
Tuple = 4
```

Methoden zur Überprüfung der momentanen Position:

`boolean isBeforeFirst()`	TRUE, falls Position vor der ersten Zeile
`boolean isAfterLast()`	TRUE, falls Position nach der letzten Zeile
`boolean isFirst()`	TRUE, falls Position in der ersten Zeile
`boolean isLast()`	TRUE, falls Position in der letzten Zeile
`int getRow()`	gibt die Nummer der momentanen Zeile oder 0 im Falle einer nicht-gültigen Zeilennummer zurück.

2.6.2 DML-Operationen in einem Scrollable Result Set

Quelle: Oracle Database JDBC Developer's Guide and Reference, S. 19-11ff

2.6.2.1 DELETE

```
...
rs.absolute(5);
rs.deleteRow();
...
```

Achtung: Die gelschte Zeile bleibt im Result Set bestehen, auch wenn sie aus der Datenbank gelöscht wurde.

2.6.2.2 UPDATE

```
Statement stmt = conn.createStatement
          (ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT empno, sal FROM emp");
if (rs.absolute(10)) // (returns false if row does not exist)
{
```

```
  rs.updateString(1, "28959");
  rs.updateFloat("sal", 100000.0f);
  rs.updateRow();
}
// Changes are made permanent with the next COMMIT operation.
```

2.6.2.3 INSERT

Schritte:

1. Gehe zur Einfügezeile (insert-row) mittels der `moveToInsertRow` Methode (Nach dem Einfügen kann mit `moveToCurrentRow` wieder zur aktuellen Zeile zurückkehren).
2. Mit den jeweiligen updateXXX – Methoden werden die Daten in die einzelnen Spalten geschrieben.
   ```
   rs.updateString(1, "mystring");
   rs.updateFloat(2, 10000.0f);
   ```
   Anstelle der Spaltennummer kann auch der Spaltenname verwendet werden.
3. Mit der insertRow-Methode werden die Änderungen in die Datenbank eingefügt.

Beispiel

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

ResultSet rs = stmt.executeQuery("SELECT empno, sal FROM emp");

rs.moveToInsertRow();
rs.updateString(1, "28959");
rs.updateFloat("sal", 100000.0f);
rs.insertRow();
// Changes will be made permanent with the next COMMIT operation.
rs.moveToCurrentRow(); // Go back to where we came from...
...
```

2.6.3 Fetch Size

Quelle: Oracle Database JDBC Developer's Guide and Reference, S. 19-15ff

By default, when Oracle JDBC runs a query, it retrieves a result set of 10 rows at a time from the database cursor. This is the default Oracle row-prefetch value. You can change the number of rows retrieved with each trip to the database cursor by changing the row-prefetch value.

JDBC 2.0 also enables you to specify the number of rows fetched with each database round trip for a query, and this number is referred to as the fetch size. In Oracle JDBC, the row-prefetch value is used as the default fetch size in a statement object. Setting the fetch size overrides the row-prefetch setting and affects subsequent queries run through that statement object.

Fetch size is also used in a result set. When the statement object run a query, the fetch size of the statement object is passed to the result set object produced by the query. However, you can also set the fetch size in the result set object to override the statement fetch size that was passed to it.

Note: Changes made to the fetch size of a statement object after a result set is produced will have no affect on that result set.

The result set fetch size, either set explicitly, or by default equal to the statement fetch size that was passed to it, determines the number of rows that are retrieved in any subsequent trips to the database for that result set. This includes any trips that are still required to complete the original query, as well as any refetching of data into the result set. Data can be refetched, either explicitly or implicitly, to update a scroll-sensitive or scroll-insensitive/updatable result set.

2.6.3.1 Setzen der Fetch Size

```
void setFetchSize(int rows) throws SQLException
int getFetchSize() throws SQLException
```

2.6.3.2 Sichtbarkeit von internen und externen Änderungen für JDBC

Table 19–1 Visibility of Internal and External Changes for Oracle JDBC

Result Set Type	Can See Internal DELETE?	Can See Internal UPDATE ?	Can See Internal INSERT?	Can See External DELETE?	Can See External UPDATE?	Can See External INSERT?
forward-only	no	yes	no	no	no	no
scroll-sensitive	yes	yes	no	no	yes	no
scroll-insensitive	yes	yes	no	no	no	no

Notes:

- Remember that explicit use of the refreshRow method, is distinct from the concept of visibility of external changes.
- Remember that even when external changes are visible, as with UPDATE operations underlying a scroll-sensitive result set, they are not detected. The result set rowDeleted, rowUpdated, and rowInserted methods always return false.

2.7 JDBC RowSets

vgl. Oracle Database JDBC Developer's Guide and Reference, S. 20-1ff

vgl. http://java.sun.com/j2se/1.5/pdf/jdbc-rowset-tutorial-1.5.0.pdf

[Quelle des nachfolgenden Artikels: http://www.tutego.com/blog/javainsel/2006/07/resultsets-in-bohnen-durch-rowset.html Autor: Christian Ullenboom]

ResultSets in Bohnen durch RowSet

1.9 ResultSets in Bohnen durch RowSet

Ein ResultSet haben wir als Informant für Zeileninformationen kennengelernt. Mit einem Cursor konnten wir über Zeilen laufen und auch scrollen, wenn positionierbare Cursor angemeldet waren. Updates waren ebenfalls möglich gewesen. Ein ResultSet ist allerdings kein Datencontainer, vergleichbar mit einer Liste, die alle Daten bei sich hat. Das ResultSet muss immer Verbindung mit der Datenbank haben und ein Aufruf von next() könnte sich die Zeileninformationen immer von der Datenbank besorgen. Ein ResultSet ist also kein Container, der Daten speichert. Aus diesem Grunde implementiert ResultSet auch nicht die Schnittstelle Serializable.

1.9.1 Die Schnittstelle RowSet

Mittlerweile gibt es die Schnittstelle javax.sql.RowSet, die von interessanten Klassen implementiert wird. Ein RowSet ist ein ResultSet – das heißt die Schnittstelle RowSet erbt von ResultSet – und schreibt weiteres Verhalten vor. In erster Linie ist es als Container für Daten gedacht. Da es ein ResultSet ist,

kann es natürlich alles, was ein normales ResultSet auch kann; mittels getXXX() Daten besorgen und mit updateXXX() Daten aktualisieren. Aber als Container verhält es sich wie eine JavaBean und an sie lassen sich Listener hängen. Sie informieren, wann etwa Elemente eingeführt oder gelöscht werden. Falls es etwa eine Visualisierung gibt, kann sie sich abhängig von den Veränderungen immer informieren lassen und die neuen Werte anzeigen. Zwar kann ein scrollendes ResultSet auch den Cursor nach oben bewegen und ihn an eine beliebig Position setzten, das RowSet kann es aber immer, auch wenn das ResultSet diese Eigenschaft nicht hat.

Zusätzlich lässt sich ein RowSet auch serialisieren und ist prinzipiell nicht an Datenbanken gebunden und die Daten können auch von einer Excel-Tabelle kommen. Jedes RowSet hat seine eigenen Meta-Daten, die durch ein RowSetMetaDataImpl-Objekt implementiert werden.

1.9.2 Implementierungen von RowSet

Um ein RowSet aufzubauen gibt es unterschiedliche Implementierungen. Da es ein Aufsatz auf die JDBC-API ist, ist es auch nicht mit einem Treiber verbunden, sondern kann unabhängig vom Treiber implementiert werden – die Umsetzung ist generisch. Auch gibt es nicht nur ein RowSet, sondern Sun unterscheidet zwischen verschiedenen Typen. Sie sind als Unterschnittstellen von javax.sql.RowSet im Paket javax.sql.rowset deklariert:

• JDBCRowSet ist ein kleiner Wrapper um das ResultSet, um es als JavaBeans zugänglich zu machen. Eine Verbindung zur Datenbank muss bestehen.

• Ein CachedRowSet benötigt initial eine Verbindung zur Datenbank, um mit einer SQL-Anweisung automatisch alle Daten zu lesen. Anschließend ist keine Verbindung zur Datenbank nötig, wobei geänderte Daten später zurückgespielt werden können. Das ist perfekt in den Fällen, wenn Daten auf ein mobiles Endgerät wandern, dort Änderungen erfahren und dieser später wieder eingespielt werden sollen.

• Ein WebRowSet erweitert CachedRowSet und bildet Daten und Operationen in XML ab, um sie zum Beispiel für Web Services übertragen zu können.

• Der FilteredRowSet ist ein WebRowSet und ermöglicht zusätzliche Selektion mit Prädikaten.

• Ein JoinRowSet ist ebenfalls ein spezieller WebRowSet. Der Daten unterschiedlicher RowSet wie über ein SQL-JOIN verbinden kann.

Sun liefert seit Java 5 Implementierungen der RowSet-Schnittstellen mit; sie enden auf -Impl. Einige Datenbankhersteller liefern eigene Implementierungen mit aus.

1.9.3 Der Typ CachedRowSet

Um ein CachedRowSet aufzubauen, welches keine dauerhafte Verbindung zur Datenbank benötigt, ist zunächst ein einfaches CachedRowSetImpl-Objekt mit dem Standardkonstruktor aufzubauen. Anschließend ist dem Objekt zu sagen, aus welcher Datenbank welche Daten zu entnehmen sind. Wie üblich muss mit vorher der Treiber geladen sein.

com/javatutor/insel/jdbc/CachedRowSetDemo.java, main()

```
CachedRowSet crset = new CachedRowSetImpl();

crset.setDataSourceName( "UllisDS" );

crset.setCommand( "SELECT * FROM Customer" );

crset.execute();
```

Kommen die Daten nicht aus einer DataSource, bestimmt setUrl() die JDC-URL. In beiden Fällen können setUsername() und setPassword() Benutzername und Passwort angeben. Damit die Bean weiß, welche Daten sie aufnehmen soll, ist eine SQL-Anweisung zu formulieren. Der Aufruf execute() füllt das CachedRowSet. (Mit einem Logger lässt sich gut beobachten, dass die Datenbankverbindung auf- und dann wieder abgebaut wird.) Falls eine Verbindung zur Datenbank schon besteht, kann bei execute() auch ein Connection-Objekt übergeben werden. Das CachedRowSet führt dann die gesetzte Anweisung über die bestehende Connection aus und überträgt die Daten aus der Datenbank in die Bean.

Bekannt aus ResultSet lässt sich mit next() durch die Ergebnismenge eines RowSets iterieren.

```
while ( crset.next() )
   System.out.println( crset.getString(1) );
crset.close();
```

Eine Position zurück geht previous(). Absolute Positionierung ist erlaubt und mit der Funktion absolute(int position) möglich. Die aktuelle Zeile liefert getRow() und die Anzahl geladener Zeilen size(). Mit den Funktion updateXXX()

lassen sich Zeilen ändern und vor eine Bewegung des Cursors mit updateRow() als getan vermerken. Änderungen müssen allerdings mit setConcurrency() angekündigt sein, denn der Standardmodus ist ResultSet.CONCUR_READ_ONLY.

```
crset.setConcurrency( ResultSet.CONCUR_UPDATABLE );
```

Änderungen werden an die Datenbank nur mit einer speziellen Funktion zurückgeschrieben, die in der Regel am Ende aller Operationen aufgerufen wird: acceptChanges(). Spätestens dann muss es wieder eine Verbindung zur Datenbank geben.

```
crset.acceptChanges();
```

Veränderte Werte werden dann in der Datenbank aktualisiert und überschrieben. Wiederum ist auch eine Variante mit Connection-Parameter implementiert, die die Daten zu einer existierenden Datenbankverbindung schreibt.

1.9.4 Der Typ WebRowSet

Das WebRowSet schreibt zusätzlich zur Ober-Schnittstelle CachedRowSet nur zwei Arten von Operationen vor: readXml() und writeXml(). Die Lese-Methoden übertragen aus unterschiedlichen Datenquellen ¬– etwa InputStream oder Reader – die Daten auf das WebRowSet. Die Schreib-Methoden schreiben das RowSet in einen OutputStream oder Writer.

```
com/javatutor/insel/jdbc/WebRowSetDemo.java, main()
WebRowSet data = new WebRowSetImpl();
data.setDataSourceName( "UllisDS" );
data.setCommand( "SELECT * FROM Customer" );
data.setMaxRows( 2 );
data.execute();
data.writeXml( System.out );
data.close();
```

Die (gekürzte) Ausgabe ist:

```
<?xml version="1.0"?>
<webRowSet xmlns="http://java.sun.com/xml/ns/jdbc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/jdbc
http://java.sun.com/xml/ns/jdbc/webrowset.xsd">
<properties>
```

```
<command>SELECT * FROM Customer</command>
<concurrency>1008</concurrency>
<datasource>UllisDS</datasource>
<escape-processing>true</escape-processing>
<fetch-direction>1000</fetch-direction>
<fetch-size>0</fetch-size>
<isolation-level>2</isolation-level>
<key-columns>
</key-columns>
<map>
</map>
<max-field-size>0</max-field-size>
<max-rows>2</max-rows>
<query-timeout>0</query-timeout>
<read-only>true</read-only>
<rowset-type>ResultSet.TYPE_SCROLL_INSENSITIVE</rowset-type>
<show-deleted>false</show-deleted>
<table-name>Customer</table-name>
<url><null/></url>
<sync-provider>
<sync-provider-name>com.sun.rowset.providers.RIOptimisticProvider</sync-provi-
der-name>
<sync-provider-vendor>Sun Microsystems Inc.</sync-provider-vendor>
<sync-provider-version>1.0</sync-provider-version>
<sync-provider-grade>2</sync-provider-grade>
<data-source-lock>1</data-source-lock>
</sync-provider>
</properties>
<metadata>
<column-count>5</column-count>
<column-definition>
<column-index>1</column-index>
<auto-increment>false</auto-increment>
<case-sensitive>false</case-sensitive>
<currency>false</currency>
<nullable>0</nullable>
<signed>true</signed>
<searchable>true</searchable>
<column-display-size>11</column-display-size>
<column-label>ID</column-label>
<column-name>ID</column-name>
<schema-name>PUBLIC</schema-name>
<column-precision>10</column-precision>
```

```
<column-scale>0</column-scale>
<table-name>CUSTOMER</table-name>
<catalog-name></catalog-name>
<column-type>4</column-type>
<column-type-name>INTEGER</column-type-name>
</column-definition>
<column-definition> ... </column-definition>
<column-definition> ... </column-definition>
<column-definition> ... </column-definition>
<column-definition> ... </column-definition>
</metadata>
<data>
<currentRow>
<columnValue>0</columnValue>
<columnValue>Laura</columnValue>
<columnValue>Steel</columnValue>
<columnValue>429 Seventh Av.</columnValue>
<columnValue>Dallas</columnValue>
</currentRow>
<currentRow>
<columnValue>1</columnValue>
<columnValue>Susanne</columnValue>
<columnValue>King</columnValue>
<columnValue>366 - 20th Ave.</columnValue>
<columnValue>Olten</columnValue>
</currentRow>
</data>
</webRowSet>
```

2.8 JDBC - Metadaten

Mit den Klassen DatabaseMetaData und ResultSetMetaData kann man Informationen über die

- Datenbank (`DatabaseMetaData`) bzw. das
- ResultSet (`ResultSetMetaData`)

erhalten, also z.B. welche Tabellen (Relationen) definiert sind, wie die Datenfelder heißen und welchen Typ sie haben, und dergleichen.

2.8.1 Datenbank-Metadaten

Damit Daten über die Datenbank erfragt werden können, muss ein DatabaseMetaData Objekt kreiert werden. Dies geschieht mit folgender Anweisung, vorausgesetzt, man ist mit einer Datenbank verbunden:

```
DatabaseMetaData dbmd = con.getMetaData();
```

Jetzt ist nur noch eine passende Methode zu finden, um die Daten zu erhalten. Der Haken an der Geschichte ist, dass es ungefähr 150 Methoden in der `DatabaseMetaData` Klasse gibt. Diese alle auch nur einigerma-

ßen zu kennen, ist kaum wünschenswert. Aber man sollte einfach einmal nach schauen, für was es so Methoden gibt.

- viele dieser Methoden liefern ein ResultSet. Aus diesem muss dann die gewünschte Information heraus geholt werden.
- einige der Methoden, welche Informationen über die Datenbank oder die Tabellen liefern, verwenden recht eigenartige Namensmuster. Es kann auch sein, dass man die Informationen nur erhält, falls die Groß- / Kleinschreibung beachtet wird, je nach DBMS.

Die wichtigsten Informationen erhält man jedoch in der Regel sehr einfach. Dies umfasst beispielsweise:

- Namen der Datenbank
- Name des Treibers
- Version
- maximale Anzahl möglicher gleichzeitiger Datenbankverbindungen
- SQL Konformität

Viele Programme benötigen überhaupt keine Metadaten. Es kann auch sein, dass eine bestimmte Datenbank eine gewünschte Information nicht liefert, nicht liefern kann.

Manche `DatabaseMetaData` Methoden stellen Informationen über SQL Objekte (z.B. eine Tabelle) zur Verfügung. Mit der Methode `getPrimaryKeys` kann beispielsweise ausgelesen werden, welche Spalten dem Primärschlüssel angehören.

Beispiel

```
Connection conn = null;
DatabaseMetaData dbmd;
ResultSet rsMetaData;
String table = "emp";
...
conn = DriverManager.getConnection(sURL, SUserID, sPassword);
dbmd = conn.getMetaData();
rsMetaData = dbmd.getPrimaryKeys(null,null,table.toUpperCase());

// im Resultset befinden sich sämtliche Spalten des Primärschlüssels
while (rsMetaData.next()) {

   // in der 4. Spalte findet man den Namen einer Schlüsselspalte
   System.out.println("Spalte : " + rsMetaData.getString(4)";

   // in der 5. Spalte ist die Anordnung der Spalte im Resultset
   System.out.println("Sequenz: " + rsMetaData.getString(5)";
}
rsMetaData.close();
conn.close();
```

2.8.2 Resultset-Metadaten

Den Zugriff auf die `ResultSetMetaData` erhält man mit der Methode `getMetaData` im ResultSet. Ein paar typische Methoden der Klasse ResultSetMetaData sind:

`int getColumnCount()`
Anzahl der Datenfelder

`String getColumnName ( int n )`
welchen Namen das n-te Datenfeld hat

`int getColumnType ( int n )`
welchen der SQL-Datentypen das n-te Datenfeld hat (siehe die statischen Konstanten in der Klasse Types)

`boolean isSearchable ( int n )`
ob das n-te Datenfeld ein Suchfeld ist, das in der WHERE-Klausel angegeben werden darf

`int getColumnDisplaySize ( int n )`
wie viele Zeichen man maximal für die Anzeige des n-ten Datenfeldes braucht

`String getColumnLabel ( int n )`
welche Bezeichnung man für das n-te Datenfeld angeben soll (eventuell verständlicher als der Datenfeldname)

`String getSchemaName ( int n )`
in welchem Datebank-Schema (DDL) das n-te Datenfeld definiert ist

`String getTableName ( int n )`
zu welcher Tabelle (Relation) das n-te Datenfeld gehört

Anmerkung: Der Oracle-JDBC-Treiber unterstützt die Methoden `getSchemaName` und `getTableName` nicht, da diese von der Datenbank Oracle 10g nicht unterstützt werden.

2.9 Fehlerbehandlung[1]

Exceptions sind ein wichtiges Konzept in Java. Auch JDBC unterstützt Exceptions und es lohnt sich diese zu kennen. Man findet in vielen Foren im Internet Fragen zu diesem Thema.

Wichtig ist, dass man sich beim Entwickeln von Anwendungen, speziell Datenbankanwendungen, vergewissert, dass die Qualität stimmt. Man möchte kaum, dass die Buchungen auf dem Bankkonto nicht korrekt sind.

Man kann sehr leicht Warnungen oder Fehler in seinen Programmen generieren, vermutlich ist das schon manchmal passiert:

Folgende Fälle sind bspw. anzuführen:

1. Löschen von Daten, die es nicht gibt.
2. Einfügen von Datensätzen mit einem Schlüssel, der bereits vergeben ist.
3. UPDATE einer Zeile die nicht vorhanden ist.
4. DROP einer Tabelle, welche nicht vorhanden ist
5. UPDATE einer Zeile, mit beispielsweise einer Zeichenkette, die länger ist als bei der Definition der Tabelle angegeben.
6. SQL Syntax Fehler (DBMS spezifisch)

Hier geht es um drei Level :

- SQL Exceptions
- SQL Warnings
- Data Truncations, also Fälle, in denen die Daten im vorgesehenen Datentyp oder dessen Länge (Anzahl Zeichen) nicht Platz finden.

2.9.1 SQL Exception

Viele der im Package java.sql Package vorhandenen Methoden werfen eine SQLException. Diese müssen mit `try{ } catch() { }` Blöcken abgefangen werden, also genauso wie irgend eine andere Exception.

Damit werden beispielsweise SQL Fehler oder Treiber Fehler erkannt und gemeldet.

Neben der `getMessage()` Methode aus der Klasse Throwable kennt SQLException weitere Methoden, welche weitere Informationen liefern:

- `getSQLState()` liefert eine Zustandskennzeichnung gemäss der X/Open SQL Spezifikation. In den DBMS Manuals stehen normalerweise Erklärungen, was das Werfen einer dieser Exceptions im konkreten Fall / DBMS bedeutet.
- `getErrorCode()` liefert einen DBMS Anbieter spezifischen Fehlercode.
- `getNextException()` zeigt die nächste SQLException oder null falls es keine weiteren mehr gibt. Da bei einem Datenbankzugriff sehr viel schief laufen kann, können Sie damit eine umfassende Liste der möglichen Fehler erstellen.

1 Vgl. Joller, S. 79 ff

- `setNextException()` gestattet dem Programmierer weitere SQLExceptions einer Exception Kette hinzuzufügen.

Diese Methoden funktionieren genau so wie man erwarten kann. Typischerweise hat man folgende Codefragmente in seinen Programmen:

```
try {
  // DB Code
} catch ( SQLException SQLe) {
  while( SQLe != null) {
    // doHandling
    SQLe = SQLe.getNextException();
  }
} // end catch
```

Tipp: falls man Wert darauf legt bei SQL Fehlern die SQL Anweisung sehen zu können, sollte man jeweils die Anweisung `Connection.nativeSQL(ihrQueryString)` in die Behandlung der Exception (catch) einbauen.

2.9.2 SQL Warnings

Eine SQLWarning ist eine Unterklasse der SQLException Klasse. Sie wirft aber keine Exception, sondern es liegt am Programmierer diese Warnungen explizit abzufragen und anzuzeigen.

`Connections`, `Statements` und `ResultSets` kennen alle eine `getWarnings()` Methode. Es gibt auch eine `clearWarnings()` Methode, um ein mehrfaches Lesen der gleichen Warnung zu vermeiden. Die `SQLWarning` Klasse selber kennt zwei Methoden `getNextWarning()` und `setNextWarning()`.

Eine SQLWarning gleicht einer traditionellen Compiler Warnung: irgend etwas stimmt nicht, aber der Effekt ist nicht so, dass deswegen das Programm nicht lauffähig wäre. Es hängt schlicht von Kontext ab, ob man etwas unternehmen muss oder einfach darüber hinweg sehen kann.

Statements löschen die Warnungen automatisch, sobald sie erneut ausgeführt werden. ResultSets löschen die Warnung immer dann, wenn eine neue Zeile gelesen wird. Connections können sich so oder so verhalten. Die Dokumentation sagt nichts darüber aus. Sicherer sind Sie, falls Sie in diesem Fall `clearWarnings()` Methode anwenden. sobald Sie die Warnung, die Sie erhalten, analysiert haben.

Ein typisches Codefragment könnte folgendermaßen aussehen:

```
try {
  ...
  stmt = con.createStatement();
  sqlw = con.getWarnings();
  while( sqlw != null) {
    // handleSQLWarnings
    sqlw = sqlw.getNextWarning();
  }
  con.clearWarnings();
  stmt.executeUpdate( sUpdate );
  sqlw = stmt.getWarnings();
  while( sqlw != null) {
    // handleSQLWarnings
    sqlw = sqlw.getNextWarning();
  }
} // end try
catch ( SQLException SQLe) {
  ...
} // end catch
```

2.9.3 Data Truncation

DataTruncation entspricht in etwa einer SQL Warnung. Falls eine solche Ausnahme beim Lesen auftritt, wird eine Warnung gesetzt. Falls sie in einem write/update Vorgang auftritt, wird eine Exception geworfen mit SQLState of 01004.

Data Truncation bedeutet, dass weniger Information gelesen oder geschrieben wird, als verlangt wird. Einige Datenbanktreiber akzeptieren Daten, welche grösser sind, als der vorhandene Platz in der entsprechenden Spalte, im entsprechenden Zielfeld. Aber diese zu grossen Daten müssen irgendwie gekürzt werden, um abgespeichert zu werden.

Sie können mit den folgenden Methoden Informationen über die Daten und Data Truncation erfahren: `getDataSize(), getIndex(), getParameter(), getRead()` und `getTransferSize()`.

2.9.4 Navigieren durch SQLExceptions

ab Java 5 bzw. JDBC 4 (Quelle: JDBC 4.0 Specification)

Es ist möglich, dass während der Ausführung eines SQL-Statements mehrere Ausnahmen (exceptions) geworfen werden. D.h. wird eine `SQLException` abgefangen, können jedoch noch weitere `SQLExceptions` mit der ursprünglich geworfenen `SQLException` verkettet sein.

Die Methode `SQLException.getCause` kann rekursiv aufgerufen werden (bis NULL zurückgegeben wird), um die Kette der abhängigen `SQLExceptions` durch zulaufen.

Der nachfolgende Code demonstriert das Navigieren durch solche verketteten `SQLExceptions`.

Beispiel:

```
catch(SQLException ex) {
  while(ex != null) {
    System.out.println("SQLState:" + ex.getSQLState());
    System.out.println("Error Code:" + ex.getErrorCode());
    System.out.println("Message:" + ex.getMessage());
    Throwable t = ex.getCause();
    while(t != null) {
      System.out.println("Cause:" + t);
      t = t.getCause();
    }
     ex = ex.getNextException();
  }
}
```

oder Verwendung einer For-Each Schleife

```
catch(SQLException ex) {
  for(Throwable e : ex ) {
    System.out.println("Error encountered: " + e);
  }
}
```

2.10 Stored Procedures

Beispiel:

```
CallableStatement cstmt= conn.prepareCall("{ ? = call mondial_statistic.get_gdp(?)}");
cstmt.registerOutParameter(1, OracleTypes.CURSOR);
cstmt.setInt(2,Integer.parseInt(jtf_percent.getText()));
cstmt.execute();
ResultSet rs= ((OracleCallableStatement) cstmt).getCursor(1);
while(rs.next()) {
...
}
```

2.11 Transaktionen (Commit)

2.12 Verwendung besonderer Datentypen (Date, LOBs)

2.12.1 DATE

2.12.2 CLOB

in Oracle 10g können CLOBs wie Strings behandelt werden

(siehe hierzu http://www.oracle.com/technology/sample_code/tech/java/codesnippet/jdbc/clob10g/handling-clobsinoraclejdbc10g.html)

2.12.3 BLOB

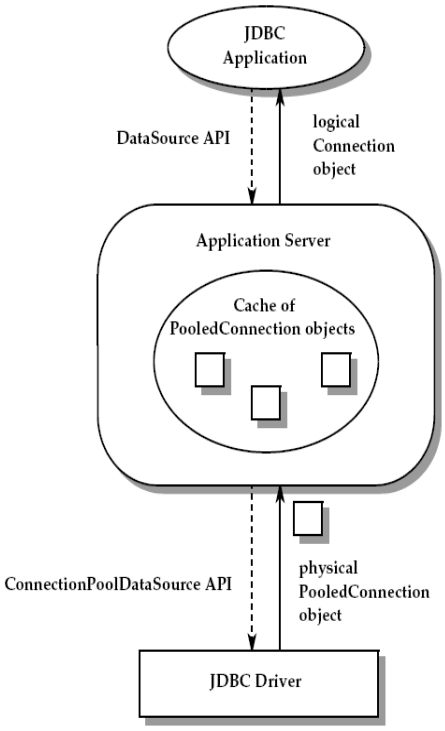
2.13 Connection Pooling

Der Aufbau einer Datenbankverbindung ist relativ zeitaufwendig (man spricht hier von **kostenintensiv**).

Bei überschaubare Programmen ist dies kein Problem, denkt man aber an z.B. Webanwednungen mit sehr vielen (> 100) internen Threads bzw. Servets die auf die Datenbank zugreifen. Die Threads/Servets sind sehr kurzlebig und würden immer DB-Verbindungen auf- und abbauen. Der DB-Server würde bald überlastet sein.

Daher gibt es Connection-Pools, bei der begrenzte Mengen an „echten“ (physikalischen) Datenbankverbindungen von einem Programm je nach Bedarf mit `getConnection()` angefordert, für SQL-Befehle benutzt und anschließend wieder mit `close()` geschlossen werden. Die Verbindung wird jedoch nur virtuell geschlossen, in Wirklichkeit bleibt sie bestehen. und kann beim nächsten Anfordern mit `getConnection()` wieder ohne zeitaufwendigen Neuaufbau zugeteilt werden.

Die Applikation sollte von diesem Connection Pooling nichts merken, der Programmcode darf sich nicht ändern.



Quelle: JDBC™ 4.0 Specification, JSR 221, Lance Andersen, Specification Lead, November 7, 2006

2.14 JDBC in Applets

2.15 JDBC Versionen

2.15.1 JDBC 2.0

2.15.2 JDBC 3.0

2.15.3 JDBC 4.0

3 Übungen und Beispielspool

3.1 Übungen

Ausgeben von Tabellen, Spaltenbezeichnung mittels Index und mittels Namen. Es werden auch Summenfunktionen abgefragt (Ziel der Übung: nicht eine Schleife zum Einlesen aller Datensätze sondern die Verwendung der SQL-Aggregationsfunktion SUM)

Verwendung von Prepared-Statements. Bei der Ausgabe werden Filterkriterien verwendet (Nur jene Angestellten, die mit M beginnen).

Ändern und Einfügen von Daten in einer Tabelle. (wahlweise bereits mittels GUI)

DML, DDL

Erstelle eine Tabelle. Die Daten sind in einem String-Array SQLData gespeichert und werden in die DB geladen. Vgl. Joller Übung1 (Beispiel1 im Code), bzw. in Application *Joller* das Projekt *CreateTableUndInsert*.

DatabaseMetaData

Erstelle eine Funktion die folgendes aus den Datenbank-Metaden ausliest:

- Namen der Datenbank
- Name des Treibers
- Version
- maximale Anzahl möglicher gleichzeitiger Datenbankverbindungen
- SQL Konformität

vgl. hierzu auch http://java.sun.com/products/jdbc/driverdevs.html

ResultSetMetadata

Erstellen einer generischen Print-Funktion. Es werden die Anzahl der Spalten des Result-Sets ausgelesen, deren Namen sowie deren Datentypen. Anschließend wird eine Methode printResultSet erstellt, die die Resultsets von SQL-Statements, die am Bildschrirm eingegeben werden und anschließend ausgeführt werden, am Bildschirm formatiert wieder ausgibt.

Fehlerbehandlung

Erweitere Dein Programm XXX um eine Fehlebehandlung. Zeige dabei auch das SQL-Statemant an, dass die Fehlermeldung verursacht hat (`Connection.nativeSQL(ihrQueryString)`)

Eine GUI-Anwendung (vgl. Abt) nach dem MVC-Konzept.

Die Bilder der Klassenmitglieder werden in die DB importiert und anschließend mittels eine GUI-Anwendung dargestellt.

3.2 Beispielspool

Joller: ScrollResult (http://www.joller-voss.ch/java/notes/jdbc/jdbc_j2ee.html)
die Übung ist ab S. 155 zu finden.

4 Anhang

4.1 Testen der JDBC-Umgebung (mittels JDBC-ODBC-Bridge)

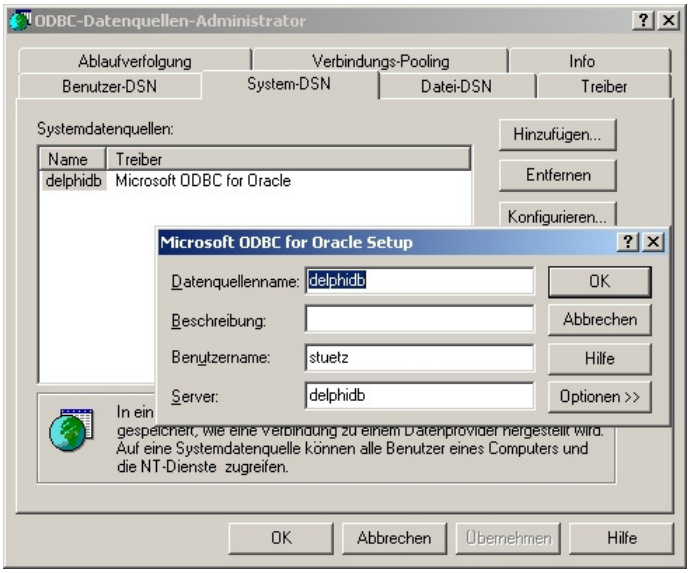
Quelle: vgl. Vorlesungsskript Objeltorientierte Programmierung (Sauer, Jürgen)

```
// You need to import the java.sql package to use JDBC
import java.sql.*;
// We import java.io to be able to read from the command line
import java.io.*;

class JdbcCheckup
{
  public static void main (String args [])
       throws SQLException, IOException
  {
    // Load the Oracle JDBC driver

    try {
         // Class.forName("oracle.jdbc.driver.OracleDriver");
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        }
    catch (Exception e)
      {
       System.out.println ("Fehler: " + e.getMessage () + "\n");
      }

    // DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
    // Prompt the user for connect information
    System.out.println (
     "Please enter information to test connection to the database");
    String user;
    String password;
    String database;
    user = readEntry ("user: ");
    int slash_index = user.indexOf ('/');
    if (slash_index != -1)
    {
      password = user.substring (slash_index + 1);
      user = user.substring (0, slash_index);
    }
    else
      password = readEntry ("password: ");
    database = readEntry ("database (a TNSNAME entry): ");
    System.out.print ("Connecting to the database...");
    System.out.flush ();
    System.out.println ("Connecting...");
    Connection conn =
      DriverManager.getConnection ("jdbc:odbc:" + database,
           user, password);
    // Create a statement
    Statement stmt = conn.createStatement ();
    // Do the SQL "Hello World" thing
    ResultSet rset = stmt.executeQuery ("select 'Hello World' from dual");
    while (rset.next ())
      System.out.println (rset.getString (1));
    System.out.println ("Your JDBC installation is correct.");
    // close the resultSet
    rset.close();
    // Close the statement
    stmt.close();
    // Close the connection
    conn.close();
  }
  // Utility function to read a line from standard input
  static String readEntry (String prompt)
  {
    try
```

```
        {
          StringBuffer buffer = new StringBuffer ();
          System.out.print (prompt);
          System.out.flush ();
          int c = System.in.read ();
          while (c != '\n' && c != -1)
          {
            buffer.append ((char)c);
            c = System.in.read ();
          }
          return buffer.toString ().trim ();
        }
        catch (IOException e)
        {
          return "";
        }
      }
    }
```

Voraussetzungen für dieses Programm:

1. Die TNSNAMES.ORA muss korrekt für die Datenbank konfiguriert werden
2. Eine ODBC-Datenquelle muss angelegt werden



Bildschirmausgabe:

```
Please enter information to test connection to the database
user: stuetz
password: passme
database (a TNSNAME entry): delphidb
Connecting to the database...Connecting...
Hello World
Your JDBC installation is correct.
Process exited with exit code 0.
```

4.2 Feststellen der Version des verwendeten JDBC-Treibers

Beispiel: Bsp01.java

```
//~--- JDK imports ------------------------------------------------------------
import java.sql.*;
import oracle.jdbc.*;
```

```
import oracle.jdbc.pool.OracleDataSource;

//~--- classes ----------------------------------------------------------------
public class Bsp01 {
  private final String driver   = "oracle.jdbc.driver.OracleDriver";
  private final String url      = "jdbc:oracle:thin:@delphi:1521:delphidb";
  private final String user     = "scott";
  private final String password = "tiger";
  private Connection conn;

  //~--- constructors -------------------------------------------------------
  public Bsp01() {
     try {
       Class.forName(driver);
       conn = DriverManager.getConnection(url, user, password);

       // Create Oracle DatabaseMetaData object
       DatabaseMetaData meta = conn.getMetaData();

       System.out.println("JDBC-Treiber Version ist " + meta.getDriverVersion());
     } catch (Exception e) {
       System.err.println(e);
       System.exit(1);
     }
  }

  //~--- methods ------------------------------------------------------------
  public static void main(String[] args) {
     Bsp01 bsp01 = new Bsp01();
  }
}
```

Konsolenausgabe:

```
JDBC-Treiber Version ist 10.1.0.4.2
Process exited with exit code 0.
```

4.3 Auslesen eines automatisch generierten Schlüssels

Connect.properties

```
#PropertiesResourceBundle für Connection Properties
driver=oracle.jdbc.driver.OracleDriver
url=jdbc:oracle:thin:@delphi.htl-leonding.ac.at:1521:delphidb
user=stuetz
pwd=passme
```

Anmerkung:

bld_person.sql

```
DROP TABLE person;
DROP SEQUENCE person_seq;
CREATE TABLE person(
    id INTEGER PRIMARY KEY
,   name VARCHAR2(25)
,   loc VARCHAR2(25)
);
/
CREATE sequence person_seq
 START WITH 1;
/
CREATE OR REPLACE TRIGGER person_autokey
 before INSERT ON "PERSON"
 FOR EACH ROW
BEGIN
  IF inserting THEN
    IF :NEW."ID" IS NULL THEN
      SELECT person_seq.nextval
      INTO :NEW."ID"
      FROM dual;
    END IF;
```

```
  END IF;
END;
```

OracleAutokey.java

```
/*
 * Titel: Abfragen von auto-generiereten Schlüsseln
 * Autor: Thomas Stuetz
 * Datum: Oktober 2007
 *
 * Quellen:
 *  http://forums.oracle.com/forums/thread.jspa?threadID=336591
 *  http://www.onjava.com/lpt/a/1058
 */

package at.stuetz.jdbc_demo;

import java.io.*;
import java.math.*;
import java.sql.*;
import java.util.*;

public class OracleAutokey {
    static Properties connProperties;

    public static void main(String[] args) {
        try {
            String srbName="Connect.properties";
            connProperties = new Properties();
            connProperties.load(new FileReader(srbName));
            //connProperties.list(System.out);
        } catch (FileNotFoundException e) {
            System.out.println("ERROR: file not found!");
            System.out.println("ERROR: " + e.getMessage());
            // e.printStackTrace();
        } catch (IOException e) {
            System.out.println("ERROR: IO-exception");
            System.out.println("ERROR: " + e.getMessage());
            // e.printStackTrace();
        }
        try {
            Class.forName(connProperties.getProperty("driver"));
        } catch (ClassNotFoundException e) {
            System.out.println("ERROR: no JDBC-Driver class found!");
            System.out.println("ERROR: " + e.getMessage());
            // e.printStackTrace();
        }
        try {
            Connection conn = DriverManager.getConnection(
                                connProperties.getProperty("url"),
                                connProperties.getProperty("user"),
                                connProperties.getProperty("pwd"));
          String sql ="INSERT INTO PERSON (name, loc) VALUES ('edi','leonding')";
            String generatedColumns[]={"ID"};
            PreparedStatement pstmt = conn.prepareStatement(sql,generatedColumns);
            pstmt.executeUpdate();
            ResultSet rs = pstmt.getGeneratedKeys();
            if (rs != null) {
                if (rs.next()) {
                    BigDecimal generatedKey = rs.getBigDecimal(1);
                    System.out.println("Generierter Key: " + generatedKey);
                }
            }
        } catch (SQLException e) {
            System.out.println("ERROR: failed to connect!");
            System.out.println("ERROR: " + e.getMessage());
            // e.printStackTrace();
        }
    }
}
```

Konsolenausgabe:

```
Generierter Key: 1
```

Eine weitere Lösung

Quelle: http://www.tutorials.de/forum/java/208113-oracle-jdbc-workaround-fuer-getgeneratedkeys.html

```
/**
 *
 */
package de.tutorials;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;

import oracle.jdbc.pool.OracleDataSource;

/**
 * @author Tom
 *
 */
public class OracleJDBCExample {

        /**
         * SQL> create table foo(id int not null, data varchar(32));
         *
         * Table created.
         *
         * SQL> create sequence seq_foo start with 1 increment by 1;
         *
         * Sequence created.
         *
         * @param args
         */
        public static void main(String[] args) throws Exception {
                OracleDataSource ods = new OracleDataSource();
                ods.setServerName("localhost");
                ods.setPortNumber(1521);
                ods.setUser("scott");
                ods.setPassword("foobar");
                ods.setDriverType("thin");
                ods.setDatabaseName("orcl");

                Connection con = ods.getConnection();
                try {
                        // PreparedStatement pstmt = con.prepareStatement("INSER
  INTO foo VALUES (seq_foo.nextval,?)",Statement.RETURN_GENERATED_KEYS);
                        // pstmt.setString(1,"Foooooo");
                        // --> Führt zu:
                        // Exception in thread "main" java.sql.SQLException: Nicht
  unterstützte Funktion

                        //Hier der Workaround
                        Statement stmt = con.createStatement();
                        ResultSet rs = stmt
                                        .executeQuery("select seq_foo.nextval from
  dual");
                        rs.next();

                        int generatedKey = rs.getInt(1);
                        stmt.close();
                        rs.close();


                        PreparedStatement pstmt = con
                                        .prepareStatement("INSERT INTO foo VALUES
  (?,?)");
                        pstmt.setInt(1,generatedKey);
                        pstmt.setString(2,"Foo:"+System.currentTimeMillis());

                        pstmt.execute();
```

```
                        pstmt.close();

                } finally {
                        if (con != null) {
                                con.close();
                        }
                }
        }
}
```

4.4 Metadaten einer Tabelle auslesen

BatchConnect.properties

```
#PropertiesResourceBundle für Connection Properties
CSDriver=oracle.jdbc.driver.OracleDriver
CSURL=jdbc:oracle:thin:@virtual-xp-prof:1521:xe
CSUserID=stuetz
CSPassword=passme
```

Beispiel: BatchConnect.java

```
package batchconnect;

import java.sql.*;
import java.util.*;

/**
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss GmbH
 * @author J.M.Joller
 * @version 1.0
 */
public class BatchJDBCConnect {
  Connection con;
  ResourceBundle rbConnect;
  ResultSet rs;
  ResultSetMetaData rsmd;
  Statement stmt;
  String sDriver,
         sDriverKey = "CSDriver",
         sPassword,
         sPasswordKey ="CSPassword",
         sQuery =
          "SELECT * FROM KaffeeListe " +
          "WHERE Kaffeesorte = 'MoJava'",
         srbName = "BatchConnect",
         sURL,
         sURLKey="CSURL",
         sUserID,
         sUserIDKey = "CSUserID";


  public BatchJDBCConnect()
  {
    try  // PropertyResourceBundle
    {
      rbConnect = ResourceBundle.getBundle( srbName );

      sDriver   = rbConnect.getString( sDriverKey );
      sPassword = rbConnect.getString( sPasswordKey );
      sURL      = rbConnect.getString( sURLKey );
      sUserID = rbConnect.getString( sUserIDKey );
    }
    catch( MissingResourceException mre )
    {
      System.err.println(
                  "ResourceBundle Problem " +
                   srbName + ", Programm wird abgebrochen." );
      System.err.println("Fehler: " + mre.getMessage() );
      return; // exit on error
    }
```

```
try   //  JDBC Driver laden
{     //  mit newInstance
    Class.forName( sDriver ).newInstance();
}
catch( Exception e )  // error
{
  System.err.println(
              "Datenbanktreiber konnte nicht geladen werden.");
  return;
} // end catch

try  {
  con = DriverManager.getConnection ( sURL,
                                      sUserID,
                                      sPassword);

  DatabaseMetaData dbmd = con.getMetaData();
  System.out.println(
    "DBMS: " +
     dbmd.getDatabaseProductName()  + ", " +
     dbmd.getDatabaseProductVersion() );

  System.out.println(
    "Driver: " +
     dbmd.getDriverName()  + ", " +
     dbmd.getDriverVersion() );

  stmt = con.createStatement();
} catch ( SQLException SQLe)  {
  System.err.println( "Der Verbindungsaufbau zu " +
                       sURL + " schlug fehl:" );
  System.err.println( SQLe.getMessage() );
  System.err.println( "SQL State: " +
                       SQLe.getSQLState() );

  if( con != null){
    try { con.close(); }
    catch( Exception e ) {}
  }

  return;
} // end catch

try {
  rs = stmt.executeQuery( sQuery );

  if( rs.next() ) // erste Zeile
  {               // Falls DAten vorliegen
    rsmd = rs.getMetaData();
    System.out.println();

    int i = rsmd.getColumnCount();

    for( int ndx = 1; ndx <= i; ndx++ )
    {
      System.out.println(
       "Column Name: " +
        rsmd.getColumnName( ndx ) + "." );
      System.out.println(
       "Column SQL Type: " +
        rsmd.getColumnTypeName( ndx ) + "." );
      System.out.println(
       "Column Java Class Equivalent: " +
        rsmd.getColumnClassName( ndx ) + ".\n" );
    }

  } // end if( result.next() )

} // end try
catch (Exception e) { e.printStackTrace(); }
finally
```

```
      {
        try { stmt.close(); }
        catch( Exception e ) {}

        try { con.close(); }
        catch( Exception e ) {}
      } // end finally clause

    } // end constructor

    public static void main (String args[]) {
      new BatchJDBCConnect();
    } // end main
  } // end class BatchJDBCConnect
```

Konsolenausgabe:

```
DBMS: Oracle, Oracle Database 10g Express Edition Release 10.2.0.1.0 - Beta
Driver: Oracle JDBC driver, 10.1.0.4.2

Column Name: EINTRAG.
Column SQL Type: NUMBER.
Column Java Class Equivalent: java.math.BigDecimal.

Column Name: MITARBEITER.
Column SQL Type: VARCHAR2.
Column Java Class Equivalent: java.lang.String.

Column Name: WOCHENTAG.
Column SQL Type: VARCHAR2.
Column Java Class Equivalent: java.lang.String.

Column Name: TASSEN.
Column SQL Type: NUMBER.
Column Java Class Equivalent: java.math.BigDecimal.

Column Name: KAFFEESORTE.
Column SQL Type: VARCHAR2.
Column Java Class Equivalent: java.lang.String.
```