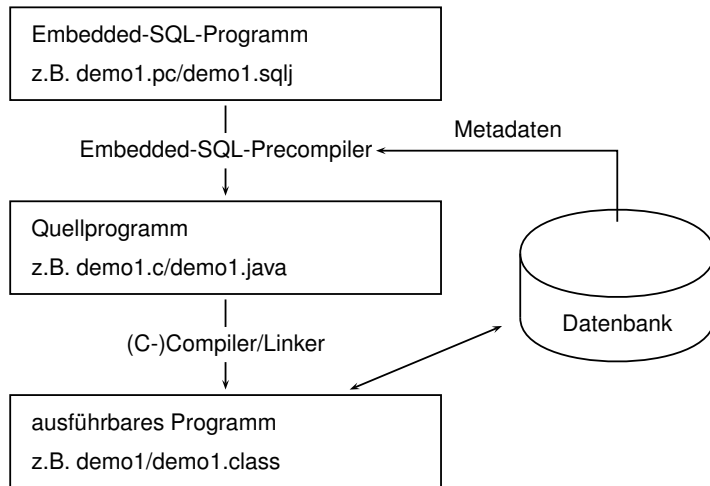


ENTWICKLUNG EINER EMBEDDED SQL-APPLIKATION



- SQLJ (siehe Folie 378): Zwischenschritt bei der Compilierung muss nicht separat ausgeführt werden.

10.2 Embedded SQL in C [Legacy]

Hinweis: dieser Abschnitt kann ausgelassen und durch SQLJ (Folie 378) ersetzt werden. Er ist nur noch für die Arbeit mit Legacy-Datenbanken relevant, die diese Technologie verwenden.

VERBINDUNGS-AUFBAU

Embedded-Anwendung: Verbindung zu einer Datenbank muss explizit hergestellt werden.

```
EXEC SQL CONNECT :username IDENTIFIED BY :passwd;
```

- username und passwd Hostvariablen vom Typ CHAR bzw. VARCHAR..
- Strings sind hier nicht erlaubt!

Äquivalent:

```
EXEC SQL CONNECT :uid;
```

wobei uid ein String der Form "name/passwd" ist.

HOSTVARIABLEN

- Kommunikation zwischen Datenbank und Anwendungsprogramm
- Output-Variablen übertragen Werte von der Datenbank zum Anwendungsprogramm
- Input-Variablen übertragen Werte vom Anwendungsprogramm zur Datenbank.
- jeder Hostvariablen zugeordnet: Indikatorvariable zur Verarbeitung von NULL-Werten.

- werden in der *Declare Section* deklariert:

```
EXEC SQL BEGIN DECLARE SECTION;
  int population;          /* host variable */
  short population\_ind;   /* indicator variable */
EXEC SQL END DECLARE SECTION;
```

- in SQL-Statements wird Hostvariablen und Indikatorvariablen ein Doppelpunkt (":") vorangestellt
- Datentypen der Datenbank- und Programmiersprache müssen kompatibel sein

INDIKATORVARIABLEN

Verarbeitung von Nullwerten und Ausnahmefällen

Indikatorvariablen für Output-Variablen:

- -1 : der Attributwert ist NULL, der Wert der Hostvariablen ist somit undefiniert.
- 0 : die Hostvariable enthält einen gültigen Attributwert.
- >0 : die Hostvariable enthält nur einen Teil des Spaltenwertes. Die Indikatorvariable gibt die ursprüngliche Länge des Spaltenwertes an.
- -2 : die Hostvariable enthält einen Teil des Spaltenwertes wobei dessen ursprüngliche Länge nicht bekannt ist.

Indikatorvariablen für Input-Variablen:

- -1 : unabhängig vom Wert der Hostvariable wird NULL in die betreffende Spalte eingefügt.
- >=0 : der Wert der Hostvariable wird in die Spalte eingefügt.

CURSORE

- Analog zu PL/SQL
- notwendig zur Verarbeitung einer Ergebnismenge, die mehr als ein Tupel enthält

Cursor-Operationen

- DECLARE <cursor-name> CURSOR FOR <sql statement>
- OPEN <cursor-name>
- FETCH <cursor-name> INTO <varlist>
- CLOSE <cursor-name>

Fehlersituationen

- der Cursor wurde nicht geöffnet bzw. nicht deklariert
- es wurden keine (weiteren) Daten gefunden
- der Cursor wurde geschlossen, aber noch nicht wieder geöffnet

Current of-**Klausel** analog zu PL/SQL

Beispiel

```
int main() {
    EXEC SQL BEGIN DECLARE SECTION;
        char cityName[50];    /* output host var */
        int cityEinw;        /* output host var */
        char* landID = "D";  /* input host var */
        short ind1, ind2;    /* indicator var */
        char* uid = "/";
    EXEC SQL END DECLARE SECTION;
    /* Verbindung zur Datenbank herstellen */
    EXEC SQL CONNECT :uid;
    /* Cursor deklarieren */
    EXEC SQL DECLARE StadtCursor CURSOR FOR
        SELECT Name, Einwohner
        FROM Stadt
        WHERE Code = :landID;
    EXEC SQL OPEN StadtCursor; /* Cursor oeffnen */
    printf("Stadt           Einwohner\n");
    while (1)
    {EXEC SQL FETCH StadtCursor INTO :cityName:ind1 ,
        :cityEinw INDICATOR :ind2;
        if(ind1 != -1 && ind2 != -1)
        { /* keine NULLwerte ausgeben */
            printf("%s      %d \n", cityName, cityEinw);
        }
    };
    EXEC SQL CLOSE StadtCursor; }
```

HOSTARRAYS

- sinnvoll, wenn die Größe der Antwortmenge bekannt ist oder nur ein bestimmter Teil interessiert.
- vereinfacht Programmierung, da damit häufig auf einen Cursor verzichtet werden kann.
- verringert zudem Kommunikationsaufwand zwischen Client und Server.

```
EXEC SQL BEGIN DECLARE SECTION;
    char cityName[50][20];    /* host array */
    int cityPop[20];          /* host array */
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT Name, Population
    INTO :cityName, :cityPop
    FROM City
    WHERE Code = 'D';
```

holt 20 Tupel in die beiden Hostarrays.

PL/SQL IN EMBEDDED-ANWEISUNGEN

- Oracle Pro*C/C++ Precompiler unterstützt PL/SQL-Blöcke.
- PL/SQL-Block kann anstelle einer SQL-Anweisung verwendet werden.
- PL/SQL-Block verringert Kommunikationsaufwand zwischen Client und Server
- Übergabe in einem Rahmen:

```
EXEC SQL EXECUTE
DECLARE
    ...
BEGIN
    ...
END;
END-EXEC;
```

DYNAMISCHES SQL

SQL-Anweisungen können durch Stringoperationen zusammengestellt werden. Zur Übergabe an die Datenbank dienen unterschiedliche Befehle, abhängig von den in der Anweisung auftretenden Variablen.

TRANSAKTIONEN

- Anwendungsprogramm wird als geschlossene Transaktion behandelt, falls es nicht durch COMMIT- oder ROLLBACK-Anweisungen unterteilt ist
- In Oracle wird nach Beendigung des Programms automatisch ein COMMIT ausgeführt
- DDL-Anweisungen generieren vor und nach ihrer Ausführung implizit ein COMMIT
- Verbindung zur Datenbank durch EXEC SQL COMMIT RELEASE; oder EXEC SQL ROLLBACK RELEASE; beenden.
- Savepoints: EXEC SQL SAVEPOINT <name>

MECHANISMEN FÜR AUSNAHMEBEHANDLUNG

SQLCA (SQL Communications Area)

Enthält Statusinformationen bzgl. der zuletzt ausgeführten SQL-Anweisung

```
struct sqlca {
    char    sqlcaid[8];
    long    sqlcabc;
    long    sqlcode;
    struct { unsigned short sqlerrml;
            char sqlerrmc[70];
    } sqlerrm;
    char    sqlerrp[8];
    long    sqlerrd[6];
    char    sqlwarn[8];
    char    sqlext[8];
};
```

Interpretation der Komponente sqlcode:

- 0: die Verarbeitung einer Anweisung erfolgte ohne Probleme.
- >0: die Verarbeitung ist zwar erfolgt, dabei ist jedoch eine Warnung aufgetreten.
- <0: es trat ein ernsthafter Fehler auf und die Anweisung konnte nicht ausgeführt werden.

WHENEVER-Statement

spezifiziert Aktionen die im Fehlerfall automatisch vom DBS ausgeführt werden sollen.

```
EXEC SQL WHENEVER <condition> <action>;
```

<condition>

- **SQLWARNING** : die letzte Anweisung verursachte eine "no data found" verschiedene Warnung (siehe auch `sqlwarn`). Dies entspricht `sqlcode > 0` aber ungleich 1403.
- **SQLERROR** : die letzte Anweisung verursachte einen (ernsthaften) Fehler. Dies entspricht `sqlcode < 0`.
- **NOT FOUND** : `SELECT INTO` bzw. `FETCH` liefern keine Antworttupel zurück. Dies entspricht `sqlcode 1403`.

<action>

- **CONTINUE** : das Programm fährt mit der nächsten Anweisung fort.
- `DO flq proc_name>` : Aufruf einer Prozedur (Fehlerroutine); `DO break` zum Abbruch einer Schleife.
- **GOTO <label>** : Sprung zu dem angegebenen Label.
- **STOP**: das Programm wird ohne `commit` beendet (`exit()`), stattdessen wird ein `rollback` ausgeführt.

Kapitel 11

Java und Datenbanken

- Java: plattformunabhängig
- überall, wo eine *Java Virtual Machine (JVM)* läuft, können Java-Programme ablaufen.
- APIs: Application Programming Interfaces; Sammlungen von Klassen und Schnittstellen, die eine bestimmte Funktionalität bereitstellen.

Mehrere der bisher behandelten Aspekte können mit Java gekoppelt werden:

- Prozeduren und Funktionen, Member Methods: Java Stored Procedures (Folie 326),
- Objekttypen: Java Object Types (Folie 332) (so kann man beliebige Datenstrukturen implementieren und anbieten → XML),
- Low-Level-Infrastruktur für Datenbankzugriff aus Java: JDBC (Folie 338),
- Embedded SQL (intern basierend auf JDBC): SQLJ (Folie 378).

STANDARDISIERUNG

- JDBC ist ein Java-API, das (DB-produktunabhängigen) low-level-Datenbankzugriff aus Java erlaubt.
- SQLJ: ANSI-Standard als Teil des SQL-Standards, bestehend aus drei Teilen:
 - Part 0: Embedded SQL in Java (ANSI X3.135.10-1998, bzw ISO-Standard “Database Language SQL:1999 – Part 10: Object Language Bindings (SQL/OLB)”; siehe Abschnitt “SQLJ”)
 - Part 1: SQL routines using Java (ANSI NCITS 331.1-1999, siehe Abschnitt “Java in Stored Procedures”).
 - Part 2: SQL types using Java (ANSI NCITS 331.2-2000, siehe Abschnitt “Java in SQL-Objekttypen”, u.a. → XMLType).
 - Part 1 und 2 bilden zusammen Part 13 des SQL:1999-Standards (ISO/IEC 9075-13:2002) “SQL Routines and Types Using the Java Programming Language (SQL/JRT)”

Technische Hinweise/Settings im IFI [Juli 2018]

- benötigte jars (in CLASSPATH reinnehmen):


```
/afs/informatik.uni-goettingen.de/group/dbis/public/oracle/instantclient :
ojdbc8.jar, [orai18n.jar], [orai18n-mapping.jar]
/afs/informatik.uni-goettingen.de/group/dbis/public/oracle/lib :
xdb.jar, [xmlparserv2.jar - Konflikt mit XML-P-jars]
/afs/informatik.uni-goettingen.de/group/dbis/public/oracle/sqlj/lib :
runtime12.jar, translator.jar
```
- Unix settings für Oracle (incl. CLASSPATH):


```
/afs/informatik.uni-goettingen.de/group/dbis/public/oracle/.oracle_env
```
- Wenn JDBC-Verbindungsaufbau zu lange dauert:


```
java -Djava.security.egd=file:///dev/urandom
classfilename
```

 (JDBC-Verbindungen sind verschlüsselt. Normales /random basiert auf externen Ereignissen (Maus, Portanfragen), die an Poolrechnern nicht genügend vorliegen).

11.1 Java in Stored Procedures und Member Methods

- Oracle hat (seit 8i) eine eigene, integrierte JVM
 - keine GUI-Funktionalität
 - Java-Entwicklung außerhalb des DB-Servers
 - keine `main()`-Methode in Klassen, nur statische Methoden (= Klassenmethoden)
 - ab 9i Nutzung von Klassen als Objekttypen
 - kein Multithreading
 - DB-Zugriff über JDBC/SQLJ, dabei wird der serverseitige JDBC-Treiber verwendet (siehe Folien 338 und 378).
- Quelldateien (`.java`), Klassendateien (`.class`) oder Archive (`.jar`) können eingelesen werden.
- Shell: `loadjava`, `dropjava`
- DB: `CREATE JAVA SOURCE`, `DROP JAVA SOURCE`
- Einbettung in Prozedur/Funktion (*Wrapper*, *call spec*) (void-Methoden als Prozeduren, non-void als Funktionen)

LADEN VON JAVA-CODE PER SHELL

Außerhalb der DB wird eine Klasse geschrieben:

```
public class Greet
{ public static String sayHello (String name)
  { System.out.println("This is Java"); // Java output
    return "Hello " + name + "!"; // return value
  } }
```

[Filename: Java/Greet.java]

- Einlesen in die Datenbank mit `loadjava`.
Empfehlenswert ist hier ein Alias:

```
alias loadjava='loadjava -thin -u
uname/passwd@dbis'
```

 dann braucht das Passwort nicht angegeben zu werden:

```
dbis@s042> loadjava -r Greet.java
```
- `-r`: wird sofort kompiliert und Referenzen aufgelöst (sonst: on-demand zur Laufzeit)
- Einlesen von `.class`-Dateien (ohne `-r`):

```
dbis@s042> loadjava Greet.class
```

 Klappt nur, wenn bei der Datenbank dieselbe Java-Version, wie auf dem Rechner wo man es kompiliert hat, läuft.
- analog mit `.jar` (das Sourcen und/oder class-Files enthält)
- Löschen einer Java-Klasse: analog mit `dropjava`

```
Greet.java
```


ERZEUGEN VON JAVA-KLASSEN IN SQL

Klasse ausserhalb der DB entwickeln und dann in der DB generieren:

```
CREATE OR REPLACE JAVA SOURCE NAMED "Hello"
AS
// here also imports are allowed
public class Greet
{ public static String sayHello (String name)
  { System.out.println("This is Java"); // Java output
    return "Hello " + name + "!"; // return value
  }
};
/
```

[Filename: Java/Greet-Create.sql]

- Wichtig: in doppelte Anführungszeichen (Version 11g)
- dann mit dem auf der nächsten Folie beschriebenen Wrapper eine PL/SQL-Prozedur daraus erzeugen,
- Löschen mit

```
DROP JAVA SOURCE "Hello";
```

- Analog: Klassen als Binaries laden:

```
CREATE OR REPLACE JAVA CLASS USING
    BFILE(directory_object, filename);
CREATE OR REPLACE JAVA CLASS USING
    {BLOB|CLOB|BFILE} subquery;
```

EINBINDEN DES JAVA-CODES IN PL/SQL-FUNKTION/PROZEDUR

Innerhalb der Datenbank:

- Funktion als Wrapper (*call spec*):

```
CREATE OR REPLACE FUNCTION greet (person IN VARCHAR2)
RETURN VARCHAR2 AS
LANGUAGE JAVA
NAME 'Greet.sayHello (java.lang.String)
    return java.lang.String';
/
```

[Filename: Java/Greet.sql]

(Bei void-Methoden: Prozedur als Wrapper)

- Aufruf: `SELECT greet('Jim') FROM DUAL;`

```
GREET('JIM')
```

```
Hello Jim!
```

- Um die Java-Ausgabe auch zu bekommen, muss man sowohl das Java-Output-Buffering als auch den SQL-Output aktivieren:

```
CALL dbms_java.set_output(2000);
SET SERVEROUTPUT ON;
```

Beispiel: `SELECT greet(name) FROM COUNTRY;`

SYNTAX DES PROZEDUR/FUNKTIONS-WRAPPERS

```
CREATE [OR REPLACE]
{ PROCEDURE <proc_name>[(<parameter-list>)]
  | FUNCTION <func_name>[(<parameter-list>)]
  RETURN sql_type}
{IS | AS} LANGUAGE JAVA
NAME '<java_method_name>[(<java-parameter-list>)]
[return <java_type_fullname>]';
/
```

- Bei void-Methoden: Prozeduren,
- Bei non-void-Methoden: Funktionen,
- Die `<parameter-list>` muss der `<java-parameter-list>` entsprechen:
 - gleiche Länge,
 - sich entsprechende Parameter-Typen;
 Parameter-Typ-Mapping: siehe Abschnitt über JDBC
- Achtung: In der NAME-Spezifikation muss `return` klein geschrieben werden,
- Aufruf des Wrappers eingebettet aus SQL und PL/SQL in Anfragen, DML-Operationen, Prozeduren, Triggern, ...

Soweit ist noch kein Datenbank-Zugriff aus den Methoden möglich. Dies wird durch JDBC ermöglicht (siehe Folie 338).

NULLWERTE AN JAVA ÜBERGEBEN

- wenn das Argument NULL ist, ist es in Java null,
- `return null` wird als SQL NULL-Wert interpretiert.

```
CREATE OR REPLACE JAVA SOURCE NAMED "Hello"
AS
public class Greet
{ public static String sayHello (String name)
  { System.out.println("This is Java");
    if (name != null)
      return "Hello " + name + "!";
    else return null;
  }
};
/
```

[Filename: Java/Greet-Null-Create.sql]

- wie vorher per Wrapper eine PL/SQL-Prozedur daraus erzeugen,
- `SELECT greet(NULL) FROM DUAL;`
- Anmerkung: in Oracle problemlos, in DB2 muss CREATE PROCEDURE mit GENERAL WITH NULLS bzw. SIMPLE WITH NULLS spezifiziert werden (→ Doku)

11.2 Java in SQL-Objekttypen

Man kann Java-Klassen als SQL-Typen registrieren. Die Java-Klasse muss das Interface `java.sql.SQLData` implementieren.

Methoden:

- `public String getSQLTypeName()`
liefert den entsprechenden SQL-Datentyp zurück
- `public void readSQL(SQLInput stream, String typeName) throws SQLException`
liest Daten aus der Datenbank und initialisiert das Java-Objekt
- `public void writeSQL(SQLOutput stream)`
bildet das Java-Objekt auf die Datenbank ab.
(vgl. Marshalling/Unmarshalling zwischen XML und Java in JAXB)

Diese drei Methoden werden nachher nicht vom Benutzer, sondern intern bei der Umsetzung aufgerufen.

BEISPIEL: JAVA-KLASSE GeoCoordJ

```
import java.sql.*;

public class GeoCoordJ implements java.sql.SQLData {
    private double latitude, longitude;

    public String getSQLTypeName() {
        return "GEOCOORD";
    }

    public void readSQL(SQLInput stream, String typeName)
        throws SQLException {
        latitude = stream.readDouble();
        longitude = stream.readDouble();
    }

    public void writeSQL(SQLOutput stream)
        throws SQLException {
        stream.writeDouble(latitude);
        stream.writeDouble(longitude);
    }
}
//... to be continued
```

- SCOTT.GEOCOORD: Name des SQL-Typs in Oracle
- Felder lesen/setzen in der Reihenfolge der SQL-Definition
- Lese-/Schreibmethoden: `stream.read/write<type>`

BEISPIEL CONT.: GeoCoordJ

```
//... continued
public String toString() { // for another example . . .
    return "GeoCoord(" + latitude + "," + longitude + ")";
}

public double distance(GeoCoordJ other) {
    return
        6370 *
        Math.acos(
            Math.cos(this.latitude/180*3.14) *
            Math.cos(other.latitude/180*3.14) *
            Math.cos(
                (this.longitude - other.longitude)
                /180*3.14
            ) +
            Math.sin(this.latitude/180*3.14) *
            Math.sin(other.latitude/180*3.14)
        );
}
}
```

[Filename: Java/GeoCoordJ.java]

dbis@s042> loadjava -r GeoCoordJ.java

SQL-WRAPPER-TYPE

```
CREATE OR REPLACE TYPE jgeocoord
AS OBJECT
EXTERNAL NAME 'GeoCoordJ'
LANGUAGE JAVA
USING SQLData
( latitude number external name 'latitude',
  longitude number external name 'longitude',
  MEMBER FUNCTION distance (other IN jgeocoord)
  RETURN NUMBER EXTERNAL
  NAME 'distance (GeoCoordJ) return double');
/
CREATE TABLE jcoordtable OF jgeocoord;
INSERT INTO jcoordtable VALUES (jgeocoord(50,0));
INSERT INTO jcoordtable VALUES (jgeocoord(-34,151));

SELECT x.distance(jgeocoord(51.5,0))
FROM jcoordtable x;
```

[Filename: Java/GeoCoordJ.sql]

X.DISTANCE(JGEOCOORD(51.5,0))

166.681667

16977.5482

Java-Objekte als Zeilenobjekte

- Tabelle sieht aus wie eine normale Tupel-/Objekttabelle:

DESC jcoordtable;

Name	Null?	Type
LATITUDE		NUMBER
LONGITUDE		NUMBER

- Einträge sehen aus wie ganz normale Zeilenobjekte:

SELECT * FROM jcoordtable;

LONGITUDE	LATITUDE
50	0
-34	151

Java-Objekte als Spaltenobjekte

- duplicate the "Mountain" table with Java coordinates:

```
CREATE TABLE mountainJcoord
AS (SELECT name, mountains, elevation, type,
        jgeocoord(m.coordinates.latitude,
                  m.coordinates.longitude) as coords
    FROM mountain m);

SELECT name, m.coords.distance(jgeocoord(51.5,0))
FROM mountainJcoord m
ORDER BY 2 DESC;
```

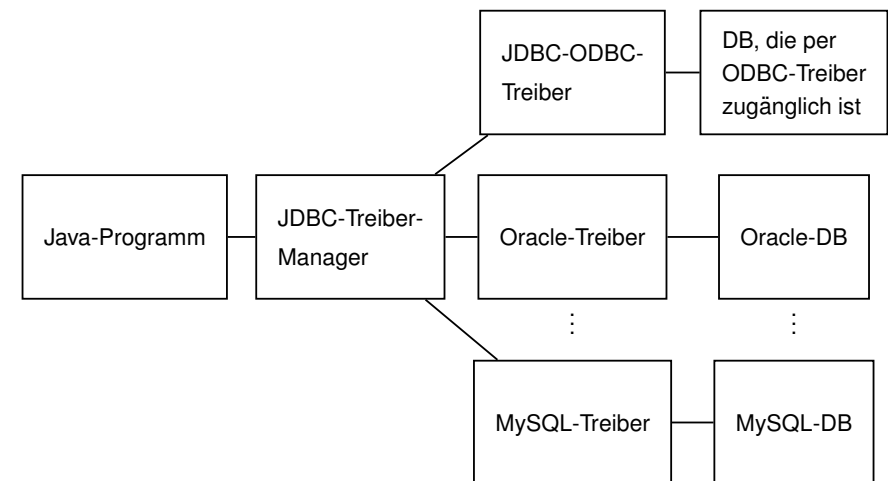
[Filename: Java/GeoCoordJ2.sql]

- select * from mountainjcoord :
Einträge sind Spaltenobjekte, die z.B. als JGEOCOORD(15.14, 120.3) ausgegeben werden.

11.3 JDBC (*Java Database Connectivity*): API für Low-Level-Datenbankzugriff

- Interface für den (entfernten) Datenbankzugriff von Java-Programmen aus,
- Teil des SDK (`java.sql.*`),
- Applikation kann unabhängig vom darunterliegenden DBMS programmiert werden,
- setzt die Idee von ODBC (Open DataBase Connectivity; ein 1992 entwickelter Standard zum Zugriff auf Datenbanken aus Programmiersprachen) auf Java um,
- gemeinsame Grundlage ist der X/Open SQL CLI (Call Level Interface) Standard.

JDBC-ARCHITEKTUR



JDBC-ARCHITEKTUR

- Kern: Treiber-Manager (`java.sql.DriverManager`)
- darunter: Treiber für einzelne DBMS'e

JDBC-API

- flexibel:
 - Applikation kann unabhängig vom darunterliegenden DBMS programmiert werden
- "low-level":
 - Statements werden durch Strings übertragen
 - im Gegensatz zu SQLJ (später) keine Verwendung von Programmvariablen in den SQL-Befehlen (d.h. Werte müssen explizit eingesetzt werden)

Darauf aufbauend:

- Embedded SQL für Java (SQLJ)
- direkte Darstellung von Tabellen und Tupeln in Form von Java-Klassen

JDBC-FUNKTIONALITÄT

- Aufbau einer Verbindung zur Datenbank (`DriverManager`, `Connection`)
- Versenden von SQL-Anweisungen an die Datenbank (`Statement`, `PreparedStatement` und `CallableStatement`)
- Verarbeitung der Ergebnismenge (`ResultSet`)

JDBC-TREIBER-MANAGER

java.sql.DriverManager

- verwaltet (registriert) Treiber
- wählt bei Verbindungswunsch den passenden Treiber aus und stellt Verbindung zur Datenbank her.
- Es wird nur ein DriverManager benötigt.

⇒ Klasse DriverManager:

- nur static Methoden (operieren auf Klasse)
- Konstruktor ist private (keine Instanzen erzeugbar)

Benötigte Treiber müssen angemeldet werden:

```
DriverManager.registerDriver(driver*)
```

Im Praktikum für den Oracle-Treiber:

```
DriverManager.registerDriver  
    (new oracle.jdbc.driver.OracleDriver());
```

erzeugt eine neue Oracle-Treiber-Instanz und "gibt" sie dem DriverManager.

VERBINDUNGS-AUFBAU

- DriverManager erzeugt offene Verbindungs-Instanz:

```
Connection conn = DriverManager.getConnection  
    (<jdbc-url>, <user-id>, <passwd>);
```

oder

```
DriverManager.getConnection(<jdbc-url>, <props>);
```

(Login-Daten aus Datei, via java.util.Properties).

- Datenbank wird eindeutig durch JDBC-URL bezeichnet

JDBC-URL:

- jdbc:<subprotocol>:<subname>
- <subprotocol>: Treiber und Zugriffsmechanismus
- <subname> bezeichnet Datenbank

Bei uns:

```
jdbc:oracle:<driver-name>:  
    @//<IP-Address DB Server>:<Port>/<Service>
```

```
String url =  
    'jdbc:oracle:thin:@/xxx.xxx.xxx.xxx:1521/dbis';
```

(die aktuelle URL steht hoffentlich auf dem Aufgabenblatt)

Verbindung beenden: `conn.close();`

VERSENDEN VON SQL-ANWEISUNGEN

Statement-Objekte

- werden durch Aufruf von Methoden einer bestehenden Verbindung `<connection>` erzeugt.
- `Statement`: einfache SQL-Anweisungen ohne Parameter
- `PreparedStatement`: Vorcompilierte Anfragen, Anfragen mit Parametern
- `CallableStatement`: Aufruf von gespeicherten Prozeduren

KLASSE "STATEMENT"

```
Statement <name> = <connection>.createStatement();
```

Sei `<string>` ein SQL-Statement *ohne Semikolon*.

- `ResultSet <statement>.executeQuery(<string>)`:
SQL-Anfragen an die Datenbank. Dabei wird eine Ergebnismenge zurückgegeben.
- `int <statement>.executeUpdate(<string>)`:
SQL-Statements, die eine Veränderung an der Datenbasis vornehmen (einschliesslich CREATE PROCEDURE etc).
Der Rückgabewert gibt an, wieviele Tupel von der SQL-Anweisung betroffen waren.
- `boolean <statement>.execute(<string>)` – Sonstiges:
 - Generierung und Aufrufe von Prozeduren/Funktionen (siehe `CallableStatements`),
 - Statement dynamisch als String erzeugt, und man weiß nicht, ob es eine Query oder ein Update ist,
 - "true" wenn das (erste) Ergebnis ein `ResultSet` ist; "false" sonst (siehe später).

Ein `Statement`-Objekt kann beliebig oft wiederverwendet werden, um SQL-Anweisungen zu übermitteln.

Mit der Methode `close()` kann ein `Statement`-Objekt geschlossen werden.

BEHANDLUNG VON ERGEBNISMENGEN

Klasse "ResultSet" (Iterator-Pattern):

```
ResultSet <name> = <statement>.executeQuery(<string>);
```

- virtuelle Tabelle, auf die von der "Hostsprache" – hier also Java – zugegriffen werden kann.
- ResultSet-Objekt unterhält einen Cursor, der mit `<result-set>.next();` auf das nächste (bzw. am Anfang auf das erste) Tupel gesetzt wird.
- `<result-set>.next()` liefert den Wert `false` wenn alle Tupel gelesen wurden.

```
ResultSet countries =
    stmt.executeQuery("SELECT Name, Code, Population
                      FROM Country");
```

Name	<u>code</u>	Population
Germany	D	83536115
Sweden	S	8900954
Canada	CDN	28820671
Poland	PL	38642565
Bolivia	BOL	7165257
..

BEHANDLUNG VON ERGEBNISMENGEN

- Zugriff auf die einzelnen Spalten des Tupels unter dem Cursor mit `<result-set>.get<type>(<attribute>)`
- `<type>` ist dabei ein Java-Datentyp,

SQL-Typ	get-Methode
INTEGER	getInt
REAL, FLOAT	getFloat
BIT	getBoolean
CHAR, VARCHAR	getString
DATE	getDate
TIME	getTime

`<getString>` funktioniert immer (*type casting*).

- `<attribute>` kann entweder durch Attributnamen, oder durch die Spaltennummer gegeben sein.


```
countries.getString("Code");
countries.getInt("Population");
countries.getInt(3);
```
- Bei `get<type>` werden die Daten des Ergebnistupels (SQL-Datentypen) in Java-Typen konvertiert.

Beispiel-Code

```

import java.sql.*;
class jdbcCities {
public static void main (String args [])
    throws SQLException {
// ORACLE-Treiber laden
DriverManager.registerDriver
    (new oracle.jdbc.driver.OracleDriver());
// Verbindung zur Datenbank herstellen
String url =
    "jdbc:oracle:thin:@xxx.xxx.xxx.xxx:1521/dbis";
Connection conn =
    DriverManager.getConnection(url,"scott","tiger");
// Anfrage an die Datenbank
Statement stmt = conn.createStatement();
ResultSet rset =
    stmt.executeQuery("SELECT Name, Population FROM City");
while (rset.next()) { // Ergebnis verarbeiten
    String s = rset.getString(1);
    int i = rset.getInt("Population");
    System.out.println (s + " " + i + "\n");
}
rset.close(); stmt.close(); conn.close();
}}

```

[Filename: Java/JdbcCities.java]

Verbindungsaufbau zu Oracle mit Properties-File

- conn.props (Dateiname ist frei wählbar) ist eine Datei (bzw. Datei im .jar-Archiv), die folgendermassen aussieht:

```

#conn.props:
## fuer normales JDBC
url=jdbc:oracle:thin:@//xxx.xxx.xxx.xxx:1521/dbis
user=scott
password=tiger
### fuer SQLJ dasselbe nochmal
sqlj.url=jdbc:oracle:thin:@//xxx.xxx.xxx.xxx:1521/dbis
sqlj.user=scott
sqlj.password=tiger

```

[Filename: Java/conn.props – muss jeder selber schreiben]

- Eine Datei dieser Form kann man in eine Instanz von java.util.Properties einlesen (ein key-Value-Hash) und dann mit getProperty() oder automatisch abfragen.
- conn.props ggf. mit ins .jar legen.

Verbindungsaufbau mit Properties-File

```
import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;
class jdbcCities {
    public static void main (String args [])
        throws Exception { // um kein try machen zu muessen
// Oracle-Treiber laden
DriverManager.registerDriver
    (new oracle.jdbc.driver.OracleDriver());
// Verbindung zur Datenbank herstellen
Properties props = new Properties();
props.load(new FileInputStream("conn.props"));
Connection conn = DriverManager
    .getConnection(props.getProperty("url"), props);
// Anfrage an die Datenbank
Statement stmt = conn.createStatement();
ResultSet rset =
    stmt.executeQuery("SELECT Name, Population FROM City");
while (rset.next()) {
    // Verarbeitung der Ergebnismenge
    String s = rset.getString(1);
    int i = rset.getInt("Population");
    System.out.println (s + " " + i); }
    rset.close(); stmt.close(); conn.close();
}}
```

[Filename: Java/jdbcCities.java]

BEHANDLUNG VON ERGEBNISMENGEN

JDBC-Datentypen

- JDBC steht zwischen Java (Objekttypen) und SQL (Typen mit unterschiedlichen Namen).
- java.sql.Types definiert *generische* SQL-Typen, mit denen JDBC arbeitet:

Java-Typ	JDBC-SQL-Typ in java.sql.Types
java.lang.String	CHAR, VARCHAR
java.math.BigDecimal	NUMBER, NUMERIC, DECIMAL
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	FLOAT, DOUBLE
java.sql.Date	DATE (Tag, Monat, Jahr)
java.sql.Time	TIME (Stunde, Minute, Sekunde)

Diese werden auch verwendet, um Meta-Daten zu verarbeiten.

BEHANDLUNG VON ERGEBNISMENGEN

Im Fall von allgemeinen Anfragen weiß man oft nicht, wieviele Spalten eine Ergebnismenge hat, wie sie heißen, und welche Typen sie haben.

Instanz der Klasse `ResultSetMetaData` enthält Metadaten über das vorliegende `ResultSet`:

```
ResultSetMetaData <name> = <result-set>.getMetaData();
```

erzeugt ein `ResultSetMetaData`-Objekt, das Informationen über die Ergebnismenge enthält:

- `int getColumnCount()`:
Spaltenanzahl der Ergebnismenge
- `String getColumnLabel(int)`:
Attributname der Spalte `<int>`
- `String getTableName(int)`:
Tabellenname der Spalte `<int>`
- `int getColumnType(int)`:
JDBC-Typ der Spalte `<int>`
- `String getColumnTypeName(int)`:
Unterliegender DBMS-Typ der Spalte `<int>`

BEHANDLUNG VON ERGEBNISMENGEN

- keine NULL-Werte in Java:

```
<resultSet>.wasNull()
```

testet, ob der zuletzt gelesene Spaltenwert NULL war.

Beispiel: Ausgabe aller Zeilen eines ResultSets

```
ResultSetMetaData rsetmetadata = rset.getMetaData();
int numCols = rsetmetadata.getColumnCount();
while (rset.next()) {
    for(int i=1; i<=numCols; i++) {
        String returnValue = rset.getString(i);
        if (rset.wasNull())
            System.out.println ("null");
        else
            System.out.println (returnValue);
    }
}
```

- Mit der Methode `close()` kann ein `ResultSet`-Objekt explizit geschlossen werden.

Beispiel: Auslesen einer beliebigen Tabelle

```

import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;
class JdbcSelect {
    public static void main (String args [])
        throws Exception {
        DriverManager.registerDriver(new
            oracle.jdbc.driver.OracleDriver());
        Properties props = new Properties();
        props.load(new FileInputStream("conn.props"));
        Connection conn = DriverManager
            .getConnection(props.getProperty("url"), props);
        Statement stmt = conn.createStatement();
        ResultSet rset =
            stmt.executeQuery("SELECT * FROM " + args[0]);
        ResultSetMetaData rsetmetadata = rset.getMetaData();
        int numCols = rsetmetadata.getColumnCount();
        while (rset.next()) {
            for(int i=1; i<=numCols; i++) {
                String value = rset.getString(i);
                if (rset.isNull()) System.out.print("null");
                else System.out.print(value);
                System.out.print(" ");
            }
            System.out.println();
        }
        rset.close(); stmt.close(); conn.close();
    }
}
[Filename: Java/JdbcSelect.java]

```

dbis@c42> java JdbcSelect City

Zeilenobjekte entsprechen normalen Tabellen (vgl. Folie 336):

dbis@c42> java JdbcSelect jcoordtable

Auslesen von Objekten: benutzerdefinierte SQL-Typen

dbis@c42> java jdbcSelect mountain

- Für Instanzen von Objekttypen wird "null" ausgegeben, oder man erhält sogar einen Fehler:
java.lang.NoClassDefFoundError: oracle/xdm/XMLType
(... interessant, dass JDBC da etwas mit XML macht - aber was?
TODO: kann man das produktiv nutzen?)
- **Objekte mit getObject(*n*) auslesen**
(behandelt auch Strings, Zahlen etc. korrekt)
- Instanzen von **benutzerdefinierten SQL-Typen** haben als SQL-Typ (mit ResultSetMetaData.getColumnType(i)) **java.sql.Types.STRUCT** und implementieren das Interface **java.sql.STRUCT**.
- Objekttyp-Name: String name = x.getSQLTypeName()
- attribute: Object[] attrs = x.getAttributes()
enthält dann Strings, Zahlwerte, oder wieder Objekte

Beispiel: Auslesen von SQL-Objekten

[Filename: Java/JdbcSelectObj.java]

```

import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;
class JdbcSelectObj {
    public static void main (String args []) throws Exception {
        Connection conn = getConn(); // unten rausgescrollt
        Statement stmt = conn.createStatement();
        ResultSet rset =
            stmt.executeQuery("SELECT * FROM " + args[0]);
        ResultSetMetaData rsmd = rset.getMetaData();
        int numCols = rsmd.getColumnCount();
        while (rset.next()) {
            for(int i=1; i<=numCols; i++) {
                Object value = rset.getObject(i);
                // System.out.print(rsmd.getColumnType(i));
                if (rset.isNull()) System.out.print("null ");
                else if (rsmd.getColumnType(i) == java.sql.Types.STRUCT)
                { java.sql.Struct s = (java.sql.Struct)value;
                  System.out.print(s.getSQLTypeName() + "( ");
                  Object[] attributes = s.getAttributes();
                  // attributes
                  for (int j = 0; j < attributes.length; j++)
                      System.out.print(attributes[j] + " ");
                  System.out.print(")");
                }
                else System.out.print(value + " "); }
            System.out.println(); }
        rset.close(); stmt.close(); conn.close(); }
private static Connection getConn() throws Exception
{
    DriverManager.registerDriver(new
        oracle.jdbc.driver.OracleDriver());

```

Auslesen von Objekten: Instanzen von Java-Klassen

- Für Instanzen von Java-Klassen (GeoCoordJ) in **Spaltenobjekten** (d.h. nicht für die Zeilenobjekte von Folie 336, die wie Tupel behandelt werden) wird eine Instanz der entsprechenden Klasse erzeugt:
 - GeoCoordJ.java/.class liegt zugreifbar im "Java"-Subdirectory.
 - sonst: java.sql.SQLException: ...
ClassNotFoundException: GeoCoordJ
 - Man kann dort irgendeine Klasse hinlegen, die den entsprechenden Namen hat und getSQLTypeName(), readSQL(), und writeSQL() und alle Methoden, die man tatsächlich benutzen möchte (auch das können mehr sein, als im Original), implementiert.
- java JdbcSelectObj mountainJCoord ruft generisch im normalen print-Zweig deren toString() auf.
- Wenn man Methoden/Felder einer Instanz einer Java-Klasse verwenden will, muss man entsprechend testen (Java-Typen haben als java.sql.Types bei Oracle die Konstante 2008) und explizit casten.

Beispiel: Auslesen von Objekten

[Filename: Java/JdbcSelectObjCast.java]

```

import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;
class JdbcSelectObjCast {
    public static void main (String args []) throws Exception {
        Connection conn = getConn(); // unten rausgescrollt
        Statement stmt = conn.createStatement();
        ResultSet rset =
            stmt.executeQuery("SELECT * FROM " + args[0]);
        ResultSetMetaData rsmd = rset.getMetaData();
        int numCols = rsmd.getColumnCount();
        while (rset.next()) {
            for(int i=1; i<=numCols; i++) {
                Object value = rset.getObject(i);
                // System.out.print(rsmd.getColumnType(i));
                if (rset.isNull()) System.out.print("null ");
                else if (rsmd.getColumnType(i) == 2008 &&
                    value instanceof GeoCoordJ) // <<<< test
                { GeoCoordJ gc = (GeoCoordJ)value; // <<<< cast
                    // System.out.print(gc);
                    System.out.print("java-geocoord:" + gc);
                }
                else if (rsmd.getColumnType(i) == java.sql.Types.STRUCT)
                { System.out.print(
                    ((java.sql.Struct)value).getSQLTypeName() + "( ");
                    Object[] attributes =
                    ((java.sql.Struct)value).getAttributes();
                    // attributes
                    for (int j = 0; j < attributes.length; j++)
                        System.out.print(attributes[j] + " ");
                    System.out.print(")");
                }
            }
        }
    }
}

```

DYNAMISCHES SQL: GENERISCHES EXECUTE()

(selten verwendet)

Häufig: <string> dynamisch generiert. Wenn man nicht weiß, ob es eine Query oder ein Update ist, kann man mit execute() generisch vorgehen.

- boolean <statement>.execute(<string>),
(nur für Statement, nicht für PreparedStatement und CallableStatement)
- "true" wenn das (erste) Ergebnis ein ResultSet ist; "false" sonst.
- ResultSet getResultSet(): Falls das (erste) Ergebnis eine Ergebnismenge ist, wird diese zurückgegeben; falls kein Ergebnis mehr vorhanden, oder das (erste) Ergebnis ein Update-Zähler ist: null zurückgeben.
- int getUpdateCount(): Falls das (erste) Ergebnis ein Update-Zähler ist, wird dieser ($n \geq 0$) zurückgegeben; falls kein Ergebnis mehr vorhanden, oder das (erste) Ergebnis eine Ergebnismenge ist, wird -1 zurückgegeben.

PREPARED STATEMENTS

PreparedStatement <name> =

```
<connection>.prepareStatement(<string>);
```

- SQL-Anweisung <string> wird vorcompiliert.
- damit ist die Anweisung fest im Objektzustand enthalten
- effizienter als Statement, wenn ein SQL-Statement häufig (mit ggf. verschiedenen Parametern) ausgeführt werden soll.

- Abhängig von <string> ist nur eine der (parameterlosen!) Methoden

```
– <prepared-statement>.executeQuery() oder
```

```
– <prepared-statement>.executeUpdate()
```

anwendbar.

PREPARED STATEMENTS: PARAMETER

- Eingabeparameter werden durch “?” repräsentiert

```
PreparedStatement giveCountryPop =
```

```
conn.prepareStatement("SELECT Population
                        FROM Country
                        WHERE Code = ?");
```

- “?”-Parameter werden mit

```
<prepared-statement>.set<type>(<pos>, <value>);
```

 gesetzt, bevor ein PreparedStatement ausgeführt wird.
- <type>: Java-Datentyp,
- <pos>: Position des zu setzenden Parameters,
- <value>: Wert.

Beispielsequenz:

```
giveCountryPop.setString(1, "D");
```

```
ResultSet rset = giveCountryPop.executeQuery();
```

```
if (rset.next()) System.out.print(rset.getInt(1));
```

```
giveCountryPop.setString(1, "CH");
```

```
ResultSet rset = giveCountryPop.executeQuery();
```

```
if (rset.next()) System.out.print(rset.getInt(1));
```

PreparedStatement (Cont'd)

- Nullwerte werden gesetzt durch
`setNULL(<pos>, <sqlType>);`
`<sqlType>` bezeichnet den **JDBC-Typ** dieser Spalte.
- nicht sinnvoll in Anfragen (Abfrage nicht mit “= NULL” sondern mit “IS NULL”), sondern z.B. bei INSERT-Statements oder Prozeduraufrufen etc.

Beispiel: PreparedStatement

```
import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;
class JdbcCountryPop {
    public static void main (String args [])
        throws Exception {
        DriverManager.registerDriver(new
            oracle.jdbc.driver.OracleDriver());
        Properties props = new Properties();
        props.load(new FileInputStream("conn.props"));
        Connection conn = DriverManager
            .getConnection(props.getProperty("url"), props);
        PreparedStatement giveCountryPop =
            conn.prepareStatement(
                "SELECT Population FROM Country WHERE Code = ?");
        giveCountryPop.setString(1,args[0]);
        ResultSet rset = giveCountryPop.executeQuery();
        if(rset.next()) {
            int pop = rset.getInt(1);
            if (rset.isNull()) System.out.println("null");
            else System.out.println(pop); }
        else System.out.println("No existing country code");
        conn.close();
    }}
}
```

[Filename: Java/JdbcCountryPop.java]

dbis@c42> java JdbcCountryPop D

dbis@c42> java JdbcCountryPop X

ERZEUGEN VON FUNKTIONEN, PROZEDUREN ETC.

- Erzeugen von Prozeduren und Funktionen mit

```
<statement>.executeUpdate(<string>);
```

(<string> von der Form CREATE PROCEDURE ...)

```
s = 'CREATE PROCEDURE bla() IS BEGIN ... END';
```

```
stmt.executeUpdate(s);
```

CALLABLE STATEMENTS: GESPEICHERTE PROZEDUREN

Der *Aufruf der Prozedur* wird als CallableStatement-Objekt erzeugt:

- Aufrufsyntax von Prozeduren bei den verschiedenen Datenbanksystemen unterschiedlich

⇒ JDBC verwendet eine *generische* Syntax per Escape-Sequenz (Umsetzung dann durch Treiber)

```
CallableStatement <name> =
```

```
<connection>.prepareCall("{call <procedure>}");
```

```
CallableStatement cstmt =
```

```
conn.prepareCall("{call bla()}");
```

CALLABLE STATEMENTS MIT PARAMETERN

```
s = 'CREATE FUNCTION
```

```
distance(city1 IN Name, city2 IN Name)
```

```
RETURN NUMBER IS BEGIN ... END';
```

```
stmt.executeUpdate(s);
```

- Parameter:

```
CallableStatement <name> =
```

```
<connection>.prepareCall("{call <procedure>(?,...,?)}");
```

- Rückgabewert bei Funktionen:

```
CallableStatement <name> =
```

```
<connection>.prepareCall
```

```
("{? = call <procedure>(?,...,?)}");
```

```
cstmt = conn.prepareCall("{? = call distance(?,?)}");
```

- Für OUT-Parameter sowie den Rückgabewert muss zuerst der **JDBC-Datentyp** der Parameter mit

```
<callable-statement>.registerOutParameter
```

```
(<pos>, java.sql.Types.<type>);
```

registriert werden.

```
cstmt.registerOutParameter(1, java.sql.Types.NUMERIC);
```

CALLABLE STATEMENTS MIT PARAMETERN

- Vorbereitung (s.o.)

```
cstmt = conn.prepareCall("{? = call distance(?,?)}");
cstmt.registerOutParameter(1, java.sql.Types.NUMERIC);
```

- IN-Parameter werden über set<type> gesetzt:

```
cstmt.setString(2, "Gottingen");
cstmt.setString(3, "Berlin");
```

- Aufruf mit

```
cstmt.execute();
```

- Lesen des OUT-Parameters mit get<type>:

```
int distance = cstmt.getInt(1);
```

Beispiel: CallableStatement

```
import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;
class JdbcCallProc {
    public static void main (String args [])
        throws Exception {
        DriverManager.registerDriver(new
            oracle.jdbc.driver.OracleDriver());
        Properties props = new Properties();
        props.load(new FileInputStream("conn.props"));
        Connection conn = DriverManager
            .getConnection(props.getProperty("url"), props);

        CallableStatement call =
            conn.prepareCall("{? = call greet(?)}");
        call.registerOutParameter(1, java.sql.Types.VARCHAR);
        call.setString(2, args[0]);
        call.execute();
        String answer = call.getString(1);
        System.out.println(answer);
        conn.close();
    }
}
```

[Filename: Java/JdbcCallProc.java]

Wenn die Funktion "Greet" (vgl. Folie 329) für den User verfügbar ist:

```
dbis@c42> java JdbcCallProc Joe
```

FOLGE VON (UPDATE) STATEMENTS VERARBEITEN

- jedes Statement mit execute() einzeln abzuschicken kostet Zeit.

Statement.executeBatch(<string>)

- <statement>.addBatch(<string>): Statement (keine Query, nur DML-Updates und DDL-Statements) zum Batch dazunehmen,
 - ... beliebig oft ... und dann
 - int [] <statement>.executeBatch(): alle Statements ausführen; ergibt ein Array mit updateCount-Werten,
 - clearBatch()
- ⇒ Folge *verschiedener* Statements erzeugen und ausführen lassen.

PreparedStatement.executeBatch()

- mit conn.prepareStatement(<string>) mit Parameter-?-Liste erzeugen
 - Parameter mit <preparedStatement>.set<type>(<pos>, <value>) setzen,
 - <preparedStatement>.addBatch(): Gesetzte Werte zum Batch dazunehmen,
 - ... beliebig oft ... und dann
 - int [] <preparedStatement>.executeBatch(): Statement für alle Parametertupel ausführen; ergibt ein Array mit updateCount-Werten,
 - clearBatch()
- ⇒ Folge desselben Statements mit verschiedenen Parametern ausführen lassen.
- con.setAutoCommit(true) (true ist Default) ist dann auch praktisch.

CallableStatement.executeBatch()

- analog.

TRANSAKTIONSSTEUERUNG

Per Default ist für eine Connection der Auto-Commit-Modus gesetzt:

- implizites Commit nach jeder ausgeführten Anweisung (Transaktion besteht also nur aus einem Statement)
- `con.setAutoCommit(false)` schaltet den Auto-Commit-Modus aus und man muss explizite Commits ausführen.

Dann hat man die folgenden Methoden:

- `con.setSavepoint(String name)` (setzt Sicherungspunkt)
- `con.commit()` (macht Änderungen persistent)
- `con.rollback(<savepoint>)` (nimmt alle Änderungen [bis zu <savepoint>] zurück).

Beispiel: Transaktionen in JDBC

```
import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;
class JdbcTransactions {
    public static void main (String args [])
        throws Exception {
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        Properties props = new Properties();
        props.load(new FileInputStream("conn.props"));
        Connection conn1 = DriverManager
            .getConnection(props.getProperty("url"), props);
        conn1.setAutoCommit(false);
        Statement stmt1 = conn1.createStatement();
        stmt1.execute("DROP TABLE TTEST");
        stmt1.execute("CREATE TABLE TTEST(A VARCHAR2(4))");
        stmt1.execute("COMMIT");
        stmt1.execute("INSERT INTO TTEST VALUES('1BLA')");
        System.out.println ("select from conn1:");
        ResultSet rset1 = stmt1.executeQuery("SELECT A FROM TTEST");
        while (rset1.next()) {
            String s = rset1.getString(1);
            System.out.println (s); }
        Connection conn2 = DriverManager
            .getConnection(props.getProperty("url"), props);
        // Anfrage an die Datenbank
        Statement stmt2 = conn2.createStatement();
        stmt2.execute("INSERT INTO TTEST VALUES('2FOO')");
        System.out.println ("select from conn2:");
        ResultSet rset2 = stmt2.executeQuery("SELECT A FROM TTEST");
        while (rset2.next()) {
            String s = rset2.getString(1);
            System.out.println (s); }
        stmt1.execute("COMMIT");
        conn1.close();
        conn2.close();
    }
}
```

FEHLERBEHANDLUNG IN ANWENDUNGEN

- JDBC-Aufrufe werfen ggf. `SQLExceptions`.
- Nicht geschlossene Verbindungen bleiben offen.
- SQL-Ausführung in try-catch-Block mit finally einbetten:

```
Connection con = null;
Statement stmt = null;
ResultSet rset = null;
try {
    ... con, stmt, rset aufbauen und verarbeiten ...
} catch (SQLException e) { e.printStackTrace(); }
finally { rset.close(); stmt.close(); con.close(); }
```

FEHLERBEHANDLUNG, EFFIZIENZ

- Wenn die Java-Objekte `conn`, `stmt`, `rset` nur lokal sind, baut der Garbage-Collector sie auch schnell genug ab und schließt die Verbindungen.
 - Bei häufigem Kontakt zu derselben Datenbank ist es effizienter, `Connection`-Objekte nur einmal zu erzeugen/aufzubauen und dann in einem Pool zu verwalten:
`org.apache.commons.pool.impl.GenericObjectPool<T>`
 - `conn.close()` gibt die `Connection` dann an den Pool zurück.
 - In diesem Fall werden bei Fehlern liegengeliebene `Connections` ohne try-catch-finally nicht zurückgegeben, und bleiben offen.
- ⇒ bei 150 (default) `Connections` blockiert der Server:
`java.sql.SQLRecoverableException: I/O-Fehler: Got minus one from a read call`

FEHLERBEHANDLUNG - DEMO

```

import java.sql.*;
import java.util.HashSet;
import java.io.FileInputStream;
import java.util.Properties;
class JdbcConnectionOverflow {
    public static void main (String args [])
        throws Exception {
        DriverManager.registerDriver(new
            oracle.jdbc.driver.OracleDriver());
        Properties props = new Properties();
        props.load(new FileInputStream("conn.props"));
        String url = props.getProperty("url");
        Connection conn;
        Statement stmt;
        ResultSet rset;
        int i = 0;
        HashSet<Connection> s = new HashSet<Connection>();
        while (true) {
            try { Thread.sleep(200); i++; System.out.println(i);
                conn = DriverManager.getConnection(url, props);
                s.add(conn);
                stmt = conn.createStatement();
                rset = stmt.executeQuery("select * from qwertz");
            } catch (SQLException e) { e.printStackTrace(); }
            catch (InterruptedException ex) {
                Thread.currentThread().interrupt(); }
        }}}
/* ADMIN only:
select username, count(*) from V$SESSION group by username; */

```

[Filename: Java/JdbcConnectionOverflow.java]

**TECHNISCHER HINWEIS FALLS
JDBC-VERBINDUNGS-AUFBAU STOCKT**

Es kann vorkommen, dass JDBC-Aufrufe plötzlich lange dauern (und beim nächsten Versuch wieder schnell gehen usw.).

Siehe <https://blog.dbi-services.com/connect-times-to-the-db-suddenly-become-very-slow-using-sqlcl/>

Der Oracle JDBC Driver benötigt eine Zufallszahl, um die Verbindungsdaten zu verschlüsseln. Die übliche Art, wie diese mit /dev/random generiert werden, kann stocken.

Abhilfe: Aufruf von java mit:

```
java -Djava.security.egd=file:///dev/urandom java-file
argumente
```


11.4 SQL-Datenbank-Zugriff in Java Stored Procedures

- Java Stored Procedures: JDBC mit dem serverseitigen JDBC-Treiber von Oracle (jdbc:default:connection:).
- User/Password nicht angeben, da es bereits in der DB abläuft:

```
import java.sql.*;
public class GetCountryData{
    public static void getPop (String code)
        throws SQLException {
    String sql =
        "SELECT name,population FROM country WHERE code = ?";
    try {
        Connection conn = DriverManager.getConnection
            ("jdbc:default:connection:");
        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setString(1, code);
        ResultSet rset = pstmt.executeQuery();
        if (rset.next()) System.out.println(rset.getString(2));
        conn.close();
    }
    catch (SQLException e) {
        System.err.println(e.getMessage());
    }}}
```

[Filename: Java/GetCountryData.java]

JAVA-KLASSE IN PL/SQL-PROZEDUR EINBINDEN

Laden in die Datenbank:

```
loadjava -r GetCountryData.java
```

Definition und Ausführung des Wrappers in der DB:

```
CREATE OR REPLACE
    PROCEDURE getPopulation (code IN VARCHAR2) IS
    LANGUAGE JAVA
    NAME 'GetCountryData.getPop(java.lang.String)';
/
```

[Filename: Java/getCountryData.sql]

... Output aktivieren:

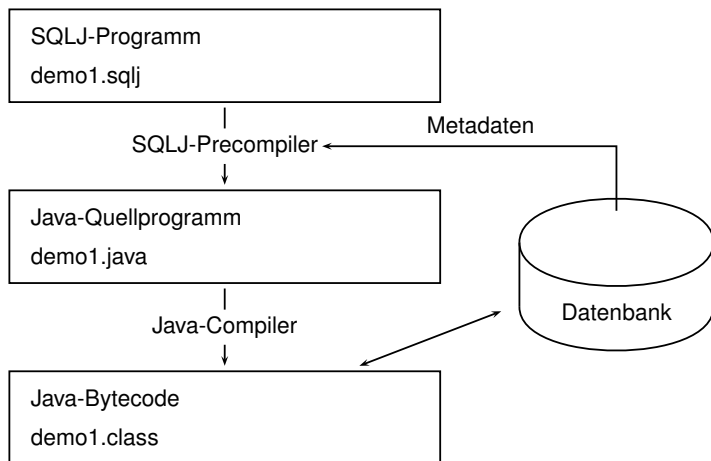
```
SET SERVEROUTPUT ON;
CALL dbms_java.set_output(2000);
```

```
EXEC getPopulation('D');
80219712
```

11.5 SQLJ

Realisierung des "Embedded SQL"-Konzeptes für Java:

- Standardisierte Spracherweiterung,
- Eingebettete SQLJ-Aufrufe werden vom Precompiler in pures Java übersetzt und dabei auf JDBC-Aufrufe abgebildet.



- **Oracle:** sqlj enthält den Precompiler und Compiler. Der Aufruf von sqlj demo1.sqlj erzeugt demo1.java und demo1.class.
- die Quelldatei muss die Endung .sqlj haben.
- Wenn man demo1.java anschaut, findet man die Umsetzung via JDBC.

Aktueller Stand (2017)

- SQLJ wird als "deprecated" (veraltet) bezeichnet.
- sqlj ruft translator.jar auf (von Oracle-Webseiten).
- Mit Java 8 zur Zeit (Juni 2017) nicht lauffähig (Klasse sun/io/CharToByteConverter wird nicht gefunden, existiert in Java 8 nicht mehr)
- aktuelle Installation im CIP-Pool: im sqlj-Skript wird der Classpath auf Java 7 gesetzt.

... interne Java-Version bei Oracle

(Stand Juni 2017, Oracle 12c)

```

SELECT dbms_java.get_ojvm_property (PROPSTRING
    =>'java.version')
FROM dual;
  
```

```

DBMS_JAVA.GET_OJVM_PROPERTY (PROPSTRING=>' JAVA.VERSION' )
-----
  
```

1.6.0_151

- 12c: JDK 7 wird auch unterstützt, aber JDK 6 ist noch Default.

ANWEISUNGEN IN SQLJ

- Anfragen:

```
#sql anIterator
  = {SELECT name, population FROM country};
```

 wobei anIterator ein (auch per SQLJ) geeignet definierter Iterator ist.
- DML und DDL:

```
#sql{<statement>;}
```
- Prozeduraufrufe:

```
#sql{CALL <proc_name>[(<parameter-list>)]};
```
- Funktionsaufrufe:

```
#sql <variable>=
  {VALUES(<func_name>[(<parameter-list>)])};
```
- Aufruf unbenannter Blöcke:

```
#sql {BEGIN ... END};
```

VERBINDUNGS-AUFBAU ZU ORACLE

Ausführliche Variante

```
import java.sql.*;
import oracle.sqlj.runtime.Oracle;
//-----
import sqlj.runtime.*;
import sqlj.runtime.ref.DefaultContext;
:
String url =
  "jdbc:oracle:thin:@/xxx.xxx.xxx.xxx:1521/dbis";
String user = "...";
String passwd = "...";
DriverManager.registerDriver
  (new oracle.jdbc.driver.OracleDriver());
Connection con =
  DriverManager.getConnection(url,user,passwd);
DefaultContext ctx = new DefaultContext(con);
DefaultContext.setDefaultContext(ctx);
Oracle.connect(url, user, passwd);
//-----
```

VERBINDUNGS-AUFBAU ZU ORACLE

Kompaktere Variante

- verwendet `conn.props`

```
import java.sql.*;
import oracle.sqlj.runtime.Oracle;
:
Oracle.connect(<JavaClass>.class, "conn.props");
:
```

- `<JavaClass>.class` ist eine Klasse, die im Dateisystem/jar-Archiv im selben Verzeichnis wie `conn.props` liegt (der Name dieser Klasse dient nur dazu, `conn.props` zu finden!).

HOSTVARIABLEN

- Verwendung von Variablen einer Host-Sprache (hier Java) in SQL-Statements
- Dient dem Datenaustausch zwischen Datenbank und Anwendungsprogramm
- in SQLJ-Statements wird Hostvariablen ein Doppelpunkt (":") vorangestellt
- Datentypen der Datenbank- und Programmiersprache müssen kompatibel sein (siehe JDBC)

In Host-Variablen schreiben:

```
int countries;
#sql{SELECT COUNT(*) INTO :countries FROM country};
```

Aus Host-Variablen lesen:

```
int population = 75000000;
#sql{UPDATE country SET population = :population
      WHERE code='D'};
```

ITERATOREN

- Allgemein: Design-Pattern, sequenzieller Zugriff auf alle Objekte, die in einem Container enthalten sind
- Hier: Iteratoren bilden das Cursor-Konzept auf SQLJ ab.
- Iteratoren mit benannten Spalten:
 - Spaltenzugriff über Spaltennamen
`<var>:= <iterator>.namei()`
 - Weberschaltung mit `<iterator>.next()`
- Positionsiteratoren:
 - Spaltenzugriff über Positionen,
 - dabei Weberschaltung
 - im embedded-SQL-escape
`#sql{FETCH :<iterator>
 INTO :<var>1, ... , :<var>n}`

(Syntax im Stil des Datenbankzugriffs bei Netzwerkdatenbanken im CODASYL-Standard (1964))

ITERATOREN MIT BENANNTEN SPALTEN

Hierbei erhalten die Attribute des Iterators Namen ("Schema"):

```
import java.sql.*;
import oracle.sqlj.runtime.Oracle;

class sqljNamedIteratorExample {
    public static void main (String args []){
        try {
            // Datenbank-Verbindung aufbauen
            Oracle.connect(sqljNamedIteratorExample.class,
                "conn.props");
            // Deklaration des Iterators mit Spaltennamen und Typen
            #sql iterator CountryIter(String name, int population);
            // Iteratorinstanz definieren
            CountryIter cIter;
            // Initialisieren des Iterators mit der SQL-Anweisung
            #sql cIter = {SELECT name, population FROM country};
            // Abarbeitung der Ergebnismenge durch Iteration
            while (cIter.next()) {
                System.out.println(cIter.name() + " has " +
                    cIter.population() + " inhabitants."); }
            cIter.close();
        }
        catch (SQLException e) {
            System.err.println(e.getMessage()); }
    }
}
```

[Filename: Java/sqljNamedIteratorExample.sqlj]

POSITIONSITERATOREN

```
import java.sql.*;
import oracle.sqlj.runtime.Oracle;
class sqljPosIteratorExample {
public static void main (String args []){
try {
// Datenbank-Verbindung aufbauen
Oracle.connect(sqljPosIteratorExample.class, "conn.props");
// Deklaration des Iterators nur mit Typen
#sql iterator CountryPosIterator(String, int);
// Hilfsvariablen der Hostsprache
String name = "";
int pop = 0;
CountryPosIterator cIter; // Iteratorinstanz definieren
// Initialisieren des Iterators mit der SQL-Anweisung
#sql cIter = {SELECT name, population FROM country};
// Abarbeitung der Ergebnismenge durch Iteration
while (true) { // hole naechsten Datensatz
#sql{FETCH :cIter INTO :name,:pop};
//Ende des Iterators erreicht?
if(cIter.endFetch()) break;
System.out.println(name + " has " +
pop + " inhabitants."); }
cIter.close();
} catch (SQLException e) { System.err.println(e.getMessage())
}}
}
```

[Filename: Java/sqljPosIteratorExample.sqlj]

VERGLEICH: JDBC UND SQLJ

JDBC

- Call-Level-Schnittstelle
- Dynamisches SQL
- Fehlererkennung erst zur Laufzeit
- Hohe Flexibilität

```
int countries;
Statement stmt = con.createStatement();
String query = "SELECT COUNT(*) FROM country";
ResultSet rset = stmt.executeQuery(query);
rset.next();
countries = rset.getInt(1);
```

SQLJ

- Embedded SQL
- Statisches SQL
- Fehlererkennung bereits zur Übersetzungszeit
- Kompakte Syntax

```
int countries;
#sql{SELECT COUNT(*) INTO :countries FROM country};
```

11.6 Weitere SQL/Oracle-Werkzeuge

- seit ORACLE8i (1999; i= internet)
Mit eingebauter Java Virtual Machine, Zugriff auf das Filesystem,
Oracle-Web Server/Internet Application Server (seit 9i):
HTML-Seiten werden abhängig vom Datenbankinhalt erstellt.
- mit den Paketen IAS, Internet File System Server wachsen Datenbank und Betriebssystem zunehmend zusammen.
- seit ORACLE9i: Integration aus der XML-Welt (*XMLType*):
XPath, XSLT, DOM, XML Schema.
... siehe weitere Folien.
- ORACLE10g: grid computing
Oracle Rules Manager für Aktive Ereignis-basierte Regeln

ENTWICKLUNGSLINIE ORACLE

- 1977: Gründung durch Larry Ellison, Bob Miner, Ed Oates und Bruce Scott
- 1979: erstes Produkt
- 1992: Oracle 7
- letzte 7er (1997): 7.3.4: erste JDBC-Version
- 1997/1998: Oracle 8 (bis 8.0.4): Objekttypen, Nested Tables
- 3.1999: Oracle 8i/8.1.5 (i = Internet); JVM, Java Stored Procedures & Member Methods, SQLJ
- 2.2001: Oracle 8.1.6: ein bisschen XML-Support (als Java-Tools)
- 6.2001: Oracle 9i: Java-Klassen als Object Types, Vererbung
- 5.2002: 9i-R2/9.2.0: verbesserter XML-Support (*XMLType*)
- 2003: Oracle 10g (g = Grid); Recycle Bin, XPath, XQuery
- 2007: 11g
- 2013: 12c: VARCHAR2 bis zu 32767 Bytes; Container-DB mit bis zu 252 Pluggable-DBs

Kapitel 12

SQL und XML

12.1 XML: “Extensible Markup Language”

... mehr als nur “Language”: Datenmodell, viele Sprachen

- Instantiierung von SGML (vgl. HTML)
- **Semantische** Tags

⇒ wer HTML kennt, weiß, wie XML “aussieht”.

- Baumstruktur
- Elemente (Name, Attribute und Inhalt)
- rekursiver Aufbau

⇒ abstrakter Datentyp mit Konstruktoren und Operationen.

- Navigation im Baum
- vgl. Pfadausdrücke in Java, OQL, SQL (z.B. x.coordinates.latitude)
- Pfadausdrücke in Unix: (z.B. /home/may/teaching/dbp/folien.tex)

⇒ Adressierungssprache “XPath”

XML: BEISPIEL

```
<country id="D" capital="cty-Germany-Stuttgart">
  <name>Germany</name>
  <total_area>356910</total_area>
  <population>83536115</population>
  <encompassed continent="europe">100</encompassed>
  <ethnicgroup name="German">95.1</ethnicgroup>
  <ethnicgroup name="Italians">0.7</ethnicgroup>
  <religion name="Roman Catholic">37</religion>
  <religion name="Protestant">45</religion>
  <language name="German">100</language>
  <border country="F">451</border>
  <border country="A">784</border>
  <border country="CZ">646</border>
  :
```