## Löschen von Datenbankobjekten

```
CREATE TYPE bla AS OBJECT (x NUMBER);
/
CREATE TABLE blatab OF bla;
SELECT table_name, table_type FROM user_object_tables;
-- name: blatab, type: bla
DROP TYPE bla FORCE;
SELECT table_name, table_type FROM user_object_tables;
-- nichts
CREATE TYPE bla AS OBJECT (x NUMBER);
/
CREATE TABLE blatab OF bla;
-- ORA-00955: name is already used by an existing object
SELECT * FROM user_objects WHERE object_name='BLATAB';
-- blatab - invalid
SELECT object_name, object_type, status from user_objects;
SELECT object_name, object_type, status from user_objects
 WHERE object_type = 'TABLE';
```

Problem: alle Tabellen löschen (Skript drop-all-tables)

- Verwendung von user_tables ignoriert invalide Tabellen.
- Nested Tables können nicht mit DROP TABLE gelöscht werden, sondern nur mit der Tabelle zu der sie gehören.
- Verwendung von user_objects where object_type = 'TABLE' bricht ab, wenn es eine (möglicherweise invalide, also in user_object_tables auch nicht mehr als nested gelistete) Nested Table löschen soll.





# 9.8 Fazit

- *Objektrelationale Tabellen* (Folie 221):
  Kompatibilität mit den grundlegenden Konzepten von SQL. U.a. Fremdschlüsselbedingungen von objektrelationalen Tabellen zu relationalen Tabellen.
- *Objektorientiertes Modell* (Folie 246):
  ... etwas kompliziert zu handhaben.
- *Object/Objekt-Relationale Views* (Folie 279):
  erlauben ein objektorientiertes externes Schema. Benutzer-Interaktionen werden durch Methoden und `INSTEAD OF`-Trigger auf das interne Schema umgesetzt. Implementierung auf relationaler Basis.
- *Objekttypen-Konzept* als Basis für (vordefinierte, in Java implementierte Klassen als) Datentypen zur Behandlung von nicht-atomaren Werten (XML (siehe Folie 353), Multimedia etc.).

# Kapitel 10 Embedded SQL

## Kopplungsarten zwischen Datenbank- und Programmiersprachen

- Erweiterung der Datenbanksprache um Programmierkonstrukte (z.B. PL/SQL)
- Erweiterung von Programmiersprachen um Datenbankkonstrukte: Persistente Programmiersprachen (Persistent Java)
- Datenbankzugriff aus einer Programmiersprache (JDBC)
- Einbettung der Datenbanksprache in eine Programmiersprache: “Embedded SQL” (C, Pascal, Java/SQLJ)





## 10.1 Embedded SQL: Grundprinzipien

... realisiert für C, Pascal, C++, Java (als SQLJ, siehe Folie 341) und weitere.

### Impedance Mismatch bei der SQL-Einbettung

- Typsysteme passen nicht zusammen
- Unterschiedliche Paradigmen: Mengenorientiert vs. einzelne Variablen

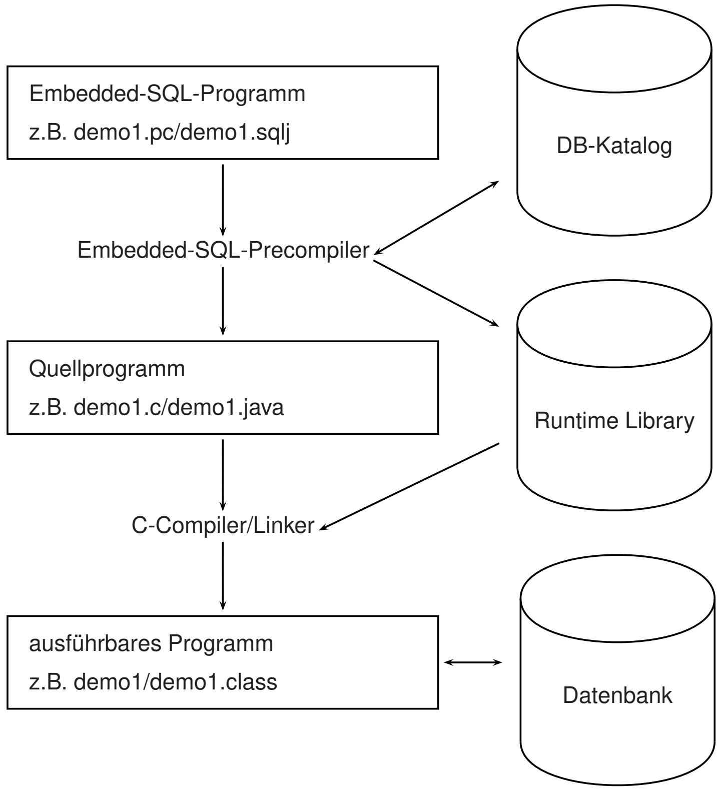
### Realisierte Lösung

- Abbildung von Tupeln bzw. Attributen auf Datentypen der Hostsprache,
- Iterative Verarbeitung der Ergebnismenge mittels Cursor.

### Auswirkungen auf die Hostsprache

- Struktur der Hostsprache bleibt unverändert,
- Spezielle Anweisungen zum Verbindungsaufbau,
- Jede SQL-Anweisung kann eingebettet werden,
- Verwendung von “Hostvariablen” (der umgebenden Programmiersprache) in SQL-Statements,
- SQL-Anweisungen wird `EXEC SQL` (oder sonstwas) vorangestellt.

## Entwicklung einer Embedded SQL-Applikation




- SQLJ (siehe Folie 341): Zwischenschritt bei der Compilierung muss nicht separat ausgeführt werden.



Praktikum: Datenbankprogrammierung in SQL/ORACLE

# 10.2 Embedded SQL in C [Legacy]

*Hinweis: dieser Abschnitt kann ausgelassen und durch SQLJ (Folie 341) ersetzt werden. Er ist nur noch für die Arbeit mit Legacy-Datenbanken relevant, die diese Technologie verwenden.*

## Verbindungsaufbau

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



Praktikum: Datenbankprogrammierung in SQL/ORACLE

## 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[35];    /* 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[35][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 verringt 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.
- API’s: 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 300),
- Objekttypen: Java Object Types (Folie 305)
(so kann man beliebige Datenstrukturen implementieren und anbieten → XML),
- Low-Level-Infrastruktur für Datenbankzugriff aus Java: JDBC (Folie 309),
- Embedded SQL (intern basierend auf JDBC): SQLJ (Folie 341).

## 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 309 und 341).
- 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 -u` *uname/passwd*`'`

  dann braucht das Passwort nicht angegeben zu werden:

  `dbis@s042> loadjava -r Greet.java`
- `-r`: wird sofort compiliert 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 compiliert 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 Namensspezifikation 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 309).



Praktikum: Datenbankprogrammierung in SQL/ORACLE

# 11.2 Java-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 longitude, latitude;

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

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

 public void writeSQL(SQLOutput stream)
   throws SQLException {
     stream.writeDouble(longitude);
     stream.writeDouble(latitude);
 } //... 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 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 geocoord
AS OBJECT EXTERNAL
NAME 'GeoCoordJ'
LANGUAGE JAVA USING SQLData(
longitude number external name 'longitude',
latitude number external name 'latitude',
MEMBER FUNCTION distance (other IN GeoCoord)
  RETURN NUMBER EXTERNAL
  NAME 'distance (GeoCoordJ) return double');
/
CREATE TABLE geoTable OF geocoord;
INSERT INTO geoTable VALUES (geocoord(10,20));
INSERT INTO geoTable VALUES (geocoord(20,30));

SET SERVEROUTPUT ON
CALL dbms_java.set_output(2000);

SELECT g.distance(geocoord(0,51)) FROM geoTable g;
```

[Filename: Java/GeoCoordJ.sql]

```
  G.DISTANCE(GEOCOORD(0,51))
--------------------------
                5530.99686
                4708.68206
```

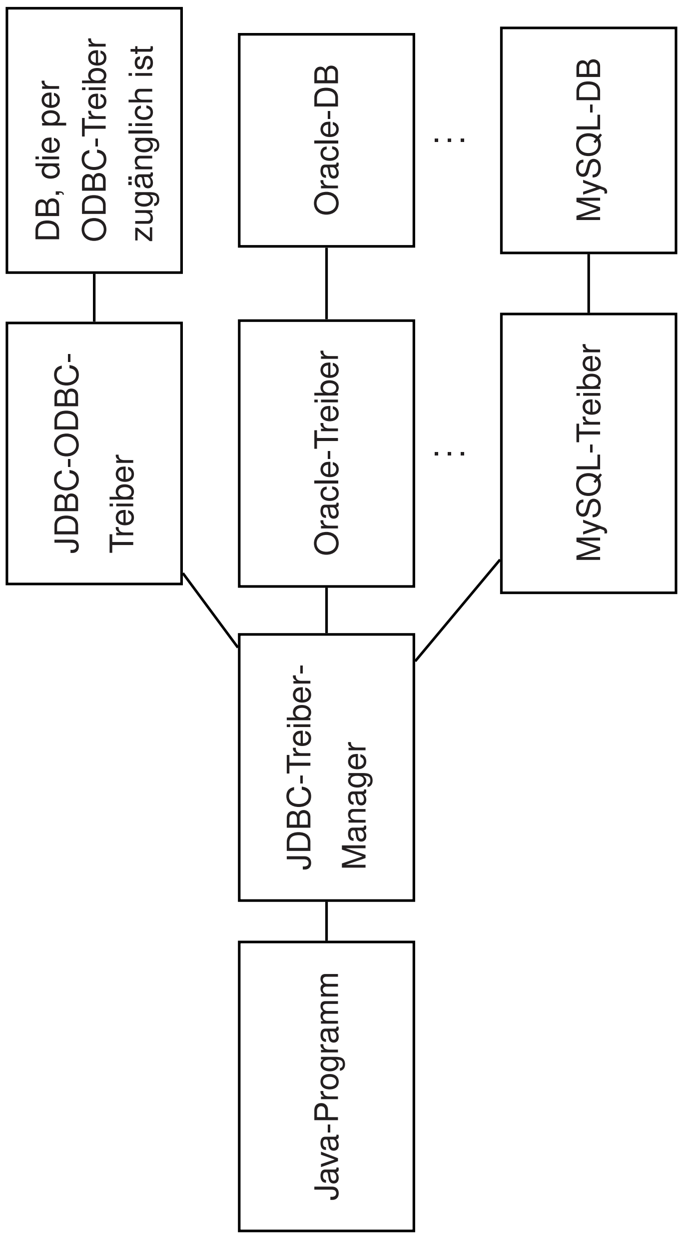




## 11.3 JDBC (*Java Database Connectivity*): API für Low-Level-Datenbankzugriff

- 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 erzeugen)

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.

## VERBINDUNGSAUFBAU

- `DriverManager` erzeugt offene Verbindungs-Instanz:

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

  oder

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

  (Login-Daten in externer Datei, `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>:<SID>
```

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

Verbindung beenden: `conn.close();`



Praktikum: Datenbankprogrammierung in SQL/ORACLE

## 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()) {
      // Verarbeitung der Ergebnismenge
   String s = rset.getString(1);
   int i = rset.getInt("Population");
   System.out.println (s + " " + i + "\n");
   }
  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.



Praktikum: Datenbankprogrammierung in SQL/ORACLE

# 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 der *aktuellen Zeile* eines ResultSets

```
ResultSetMetaData rsetmetadata = rset.getMetaData();
int numCols = rsetmetadata.getColumnCount();
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.*;
class jdbcSelect {
  public static void main (String args [])
   throws SQLException {
 DriverManager.registerDriver(new
         oracle.jdbc.driver.OracleDriver());
 String url = "jdbc:oracle:thin:/*hier korrekt fortsetzen
 Connection conn =
    DriverManager.getConnection(url,"scott","tiger");
 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.wasNull()) 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
```

## Auslesen von Objekten

```
dbis@c42> java jdbcSelect Mountain
```

- Für Instanzen von Objekttypen wird immer “null” ausgegeben.
- Objekte mit getObject auslesen
- Objekte sind vom Typ oracle.sql.STRUCT
  → testen und ggf. casten
- Objekttyp-Name: String name = x.getSQLTypeName()
- attribute: Object[] attrs = x.getAttributes()
  enthält dann Strings, Zahlwerte, oder wieder Objekte





## Beispiel: Auslesen von Objekten

[Filename: Java/jdbcSelectObj.java]

```
import java.sql.*;
class jdbcSelectObj {
  public static void main (String args []) throws SQLException {
   Connection conn = getConn();
   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++) {
     Object value = rset.getObject(i);
     if (rset.wasNull()) System.out.print("null  ");
     else if (value instanceof oracle.sql.STRUCT)
     { System.out.print("(" +
         ((oracle.sql.STRUCT)value).getSQLTypeName() + ": ");
       Object[] attributes =
         ((oracle.sql.STRUCT)value).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 SQLException
 {
   DriverManager.registerDriver(new
         oracle.jdbc.driver.OracleDriver());
   String url = "jdbc:oracle:thin:/*hier korrekt fortsetzen
   return DriverManager.getConnection(url,"scott","tiger");
```

## 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 ausgeführt werden soll.
- Abhängig von `<string>` ist nur eine der (parameterlosen!) Methoden
  - `<prepared-statement>.executeQuery()`,
  - `<prepared-statement>.executeUpdate()` oder
  - `<prepared-statement>.execute()`

  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.

```
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.*;
class jdbcCountryPop {
  public static void main (String args [])
   throws SQLException {
 DriverManager.registerDriver(new
         oracle.jdbc.driver.OracleDriver());
 String url = "jdbc:oracle:thin:/*hier korrekt fortsetzen
 Connection conn =
    DriverManager.getConnection(url,"scott","tiger");

 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.wasNull()) System.out.print("null");
  else System.out.print(pop);
  }
 else System.out.print("Kein zulaessiger Landescode");
 System.out.println();
 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.*;
class jdbcCallProc {
  public static void main (String args [])
   throws SQLException {
 DriverManager.registerDriver(new
         oracle.jdbc.driver.OracleDriver());
 String url = "jdbc:oracle:thin:/*hier korrekt fortsetzen
 Connection conn =
    DriverManager.getConnection(url,"scott","tiger");

 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 303) für den User scott/tiger verfügbar ist:

```
dbis@c42> java jdbcCallProc Joe
```

## ABFRAGEN VON ERGEBNISSEN BEI EXECUTE()

Häufig: <string> dynamisch generiert.

- erste Annahme: liefert nur ein Ergebnis.
- boolean <statement>.execute(<string>),
  boolean <prepared-statement>.execute(),
  boolean <callable-statement>.execute()
- "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.





## FOLGE VON ERGEBNISSEN VERARBEITEN

- SQL-Statements, die mehrere Ergebnisse nacheinander zurückliefern
  Hinweis: Statement wie oben darf keinen Strichpunkt enthalten, darf also keine Sequenz sein.
  $\Rightarrow$ dieser Fall tritt also nur seltenst ein ...
  [Rest der Folie ist optional]
- getMoreResults(): schaltet zum nächsten Ergebnis. true, wenn das nächste Ergebnis eine Ergebnismenge ist, false, wenn es ein Update-Zähler ist, oder keine weiteren Ergebnisse.
- alle Ergebnisse verarbeitet:

```
((<stmt>.getResultSet() == null) &&
 (<stmt>.getUpdateCount() == -1))
```

  bzw.

```
((<stmt>.getMoreResults() == false) &&
 (<stmt>.getUpdateCount() == -1))
```

## FOLGE VON ERGEBNISSEN VERARBEITEN (2)

```
stmt.execute(StatementWithUnknownResultSequence);
while (true) {
    int rowCount = stmt.getUpdateCount();
    if (rowCount > 0) { // update, n Tupel geaendert
       System.out.println("Rows changed = " + count);
       stmt.getMoreResults();
       continue; }
    if (rowCount == 0) { // update, aber nichts geaendert
       System.out.println("No rows changed");
       stmt.getMoreResults();
       continue; }
    // sonst: query
    ResultSet rs = stmt.getResultSet();
    if (rs != null) {
       .....  // verarbeite Metadaten
       while (rs.next())
          { ....}  // verarbeite Ergebnismenge
       stmt.getMoreResults();
       continue;
    }
    break;
}
```





## FOLGE VON STATEMENTS VERARBEITEN

- man kann natürlich jedes einzeln abschicken ...

executeBatch()

- `<statement>.addBatch(<string>)`: Statement (keine Query) zum Batch dazunehmen,
- ... beliebig oft ... und dann
- `int[] <statement>.executeBatch()`: alle Statements ausführen; ergibt ein Array mit updateCount-Werten,
- `clearBatch()`

## TRANSAKTIONSSTEUERUNG

Sei `con` eine Instanz der Klasse `Connection`.
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.





# 11.4 JDBC in Java Stored Procedures

In Stored Procedures verwendet man ebenfalls JDBC-Technologie mit dem serverseitigen JDBC-Treiber von Oracle (`jdbc:default:connection:`). User/Password werden natürlich nicht angegeben, 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));
    rset.close();
    pstmt.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 -u user/passwd -r getCountryData.java
```

Definition und Ausführung des Wrappers in der DB:

```
CREATE 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');
83536115
```



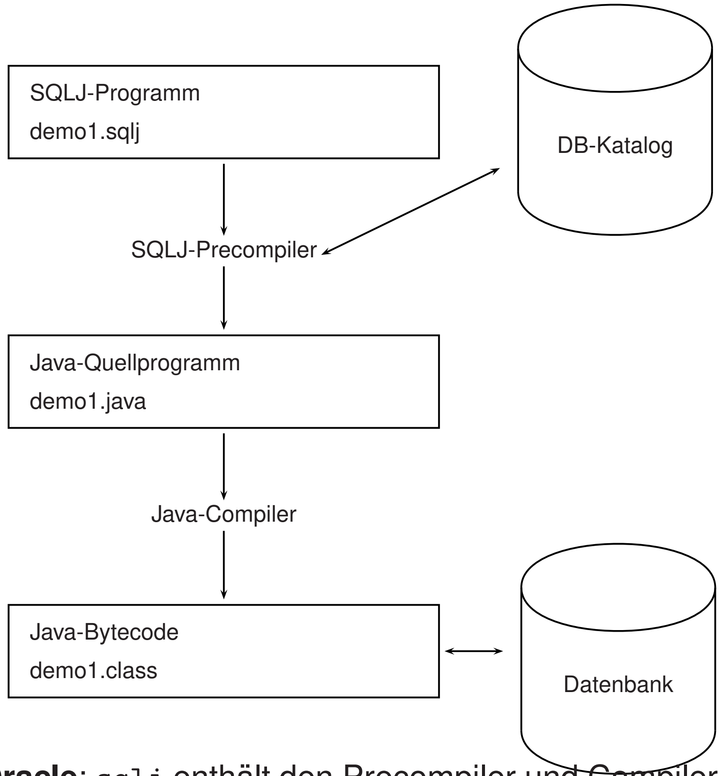


## 11.5 SQLJ

Realisierung des “Embedded SQL”-Konzeptes für Java:

- Standardisierte Java-DB Schnittstelle, verwendet JDBC
- ANSI-Standard als Teil des SQL-Standards, beteiligt waren u.a. Oracle, Sun, IBM, Microsoft (ANSI x.3.135.10-1998)
- Besteht aus drei Teilen:
  - Part 0: Embedded SQL in Java.
  - Part 1: SQL routines using Java (siehe Abschnitt “Java in Stored Procedures”).
  - Part 2: SQL types using Java (Java-Klassen als SQL Datentypen (u.a. → XMLType)).
- SQLJ Part 0: SQL-in-Java
- SQLJ Part 1 & 2: Java-in-SQL
- Eingebettete SQLJ-Aufrufe werden als pures Java übersetzt und werden auf JDBC-Aufrufe abgebildet.
- Hier: Beschränkung auf SQLJ Part 0, Part 1 als Java Stored Procedures

## ENTWICKLUNG EINER SQLJ-APPLIKATION




- **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.





## 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};
```

# VERBINDUNGSAUFBAU 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);
//--------------
```





# VERBINDUNGSAUFBAU ZU ORACLE

## Kompaktere Variante

- `connect.properties` ist eine Datei (bzw. Datei im .jar-Archiv), die folgendermassen aussieht:

```
#connect.properties:
sqlj.url=jdbc:oracle:thin:@xxx.xxx.xxx.xxx:1521:dbis
sqlj.user=<user>
sqlj.password=<passwd>
```

[Filename: Java/connect.properties – muss jeder selber schreiben]

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

- `<JavaClass>.class` ist eine Klasse, die im Dateisystem/jar-Archiv im selben Verzeichnis wie `connect.properties` liegt (der Name dieser Klasse dient nur dazu, `connect.properties` 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'};
```



Praktikum: Datenbankprogrammierung in SQL/ORACLE

## 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
  - Weiterschaltung mit next()
- Positionsiteratoren:
  - Spaltenzugriff über Position
  - Weiterschaltung mit FETCH ... INTO

## ITERATOREN MIT BENANNTEN SPALTEN

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

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

class sqljIteratorExample {
 public static void main (String args []){
  // Datenbank-Verbindung aufbauen
  try {
   Oracle.connect(sqljIteratorExample.class, "connect.properties'
   // Deklaration des benannten Iterators mit Spaltennamen
   #sql iterator CountryIter(String name, int population);
   // Iteratorobjekt wird definiert
   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."); }
   // Schliessen des Iterators
   cIter.close();
  }
  catch (SQLException e) {
   System.err.println(e.getMessage()); }
 }}
```

[Filename: Java/sqljIteratorExample.sqlj]





## POSITIONSITERATOREN

```
// Verbindungsaufbau, Deklaration Positionsiterator
Oracle.connect(Countries.class, "connect.properties");
#sql iterator CountryPosIterator(String, int);
// Hilfsvariablen
String name = "";
int pop = 0;

// Iteratorobjekt wird definiert
CountryPosIterator cIter;
// 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.");
}
// Schliessen des Iterators
cIter.close();
```

# 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};
```