Informationssysteme

Sommersemester 2016

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Datenbank-Zugriff via JDBC

Java Database Connectivity

- Bietet Schnittstelle für den Zugriff auf ein DBMS aus Java-Anwendungen

JDBC: Connect und einfache Anfrage

```
1  //registriere geeigneten Treiber (hier fuer Postgresql)
2  Class.forName("org.postgresql.Driver");
3  //erzeuge Verbindung zur Datenbank
4  Connection conn = DriverManager.getConnection(
5       "jdbc:postgresql://localhost/university",
6       "username", "password");
7
8  //erzeuge ein einfaches Statement Objekt
9  Statement stmt = conn.createStatement();
10
11       //mit execute Query koennen nun darauf Anfragen
     ausgefuehrt werden
12       //Ergebnisse in Form eines ResultSet Objekts
13       ResultSet rset = stmt.executeQuery("SELECT p.
     persnr from professoren p");
```

JDBC: Connect und einfache Anfrage

```
14          //dieses besitzt Metadaten
15          ResultSetMetaData metadata = rset.getMetaData();
16
17          //welche Attribute (Spalten) besitzen die
        Ergebnis-Tupel?
18          int column_count = metadata.getColumnCount();
19
20          for (int index=1; index<=column_count; index++) {
21                System.out.println("Spalte "+index+" heisst
        " +
22                      metadata.getColumnName(index));
23          }
24
25          //iteriere nun ueber Ergebnisse
26          while (rset.next()) {
27                System.out.println(rset.getString(1));
28          }
```

JDBC Treiber für Postgresql

`http://jdbc.postgresql.org/`

Siehe insbesondere Dokumentation dazu (mit Beispielen), sowie ganz allgemein die **Dokumentation zum Paket** `java.sql`:
`https://docs.oracle.com/javase/8/docs/api/index.html?java/sql/package-summary.html`

JDBC - wichtige Funktionalitäten

Laden des Treibers

- Kann auf verschiedene Weise erfolgen, z.B. durch explizites Laden mit dem Klassenlader:

```
Class.forName(DriverClassName);
```

Aufbau einer Verbindung

- Connection-Objekt repräsentiert die Verbindung zum DB-Server
- Beim Aufbau werden URL der DB, Benutzername und Passwort aus Strings übergeben (teilweise optional).

```
Connection conn =
                DriverManager.getConnection(url,
                                            login, password);
```

JDBC - wichtige Funktionalitäten (2)

Anweisungen

- Mit dem Connection-Objekt können u.a. Metadaten der DB erfragt und Statement-Objekte zum Absetzen von SQL-Anweisngen erzeugt werden
- Erzeugen einer SQL-Anweisung zur direkten (einmaligen) Ausführung

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

- PreparedStatement-Objekt erlaubt das Erzeugen und Vorbereiten von (parametrisierten) SQL-Anweisungen zur wiederholten Ausführung

```
PreparedStatement pstmt = conn.prepareStatement(
        "select * from personal where gehalt >= ?");
```

Schließen von Verbindungen, Statements, usw.

```
stmt.close();
conn.close();
```

JDBC - Anweisungen

Anweisungen (Statements)

- Werden in einem Schritt vorbereitet und ausgeführt

Die Methode executeQuery

- führt die Anfrage aus und liefert Ergebnis zurück

```
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery(
        "select pnr, name, gehalt
         from personal where gehalt>=40000");
//wir sehen gleich, wie man mit diesem ResultSet
arbeitet
```

JDBC - Anweisungen

Die Methode executeUpdate

- werden zur direkten Ausführung von UPDATE-, INSERT-, DELETE- und DDL-Anweisungen benutzt

```
Statement stmt = conn.createStatement();
int n = stmt.executeUpdate(
      "update personal
       set gehalt = gehalt * 1.10
       where gehalt < 20000");

//n enthaelt die Anzahl der aktualisierten Zeilen
```

JDBC - Prepared Anweisungen

PreparedStatement-Objekt

```
PreparedStatement pstmt;
double gehalt = 50000.00;
pstmt = conn.prepareStatement(
    "select * from personal where gehalt >= ?");
```

- Symbol ? markiert hier freie Parameter
- Vor der Ausführung sind dann die Parameter einzusetzen.
- Durch Methoden entsprechend Datentyp, z.B.

```
pstmt.setDouble(1, gehalt);
```

https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html

JDBC - Prepared Anweisungen (2)

Ausführen einer Prepared-Anweisung als Anfrage

```
PreparedStatement pstmt;
double gehalt = 50000.00;
pstmt = conn.prepareStatement(
    "select * from personal where gehalt >= ?");
```

Vorbereitung und Ausführung

```
pstmt = con.prepareStatement(
        "delete from personal where name = ?");
pstmt.setString(1, "Maier");

int n = pstmt.executeUpdate();

//Methoden der Prepared-Anweisungen haben keine
Argumente
```

JDBC - Ergebnismengen und Cursor

Select-Anfragen und Ergebnisübergabe

- Jede JDBC-Methode, mit der man Anfragen an das DBMS stellen kann, liefert ResultSet-Objekte als Rückgabewert

```
ResultSet rset = stmt.executeQuery(
        "select pnr, name, gehalt
        from personal where gehalt >= " + gehalt);
```

- Cursor-Zugriff und Konvertierung der DBMS-Datentypen in passende Java-Datentypen erforderlich
- JDBC-Cursor ist durch die Methode **next()** der Klasse ResultSet implementiert

`https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html`

JDBC - Ergebnismengen und Cursor (2)

Cursor → getInt("pnr") ↓ getString("name") ↓ getDouble("gehalt") ↓

↓ next()

123	Maier	23352.00
456	Schulze	34553.00

Zugriff aus Java-Programm

```
while (rset.next()) {
    System.out.print(res.getInt("pnr")+"\t");
    System.out.print(res.getString("name")+"\t");
    System.out.println(res.getString("gehalt"));
}
```

JDBC - Versch. Typen von ResultSets

TYPE_FORWARD_ONLY

- nur Aufruf von **next()** möglich

TYPE_SCROLL_INSENSITIVE

- Scroll-Operationen sind möglich, aber Aktualisierungen der Datenbank verändern ResultSet nach seiner Erstellung nicht

JDBC - Versch. Typen von ResultSets (2)

TYPE_SCROLL_SENSITIVE

- Scroll-Operationen möglich und Änderungen in der Datenbank werden berücksichtigt

ResultSet lässt Änderungen zu oder nicht:

- CONCUR_UPDATABLE
- CONCUR_READ_ONLY

```
Statement stmt = conn.createStatement(
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
ResultSet rset = stmt.executeQuery(...);
rset.updateString("name", "Schmitt");
rset.updateRow();
```

JDBC - Zugriff auf Metadaten

Allgemeine Metadaten

- Klasse **DatabaseMetaData** zum Abfragen von DB-Informationen

Informationen über ResultSets

- JDBC bietet die Klasse **ResultSetMetaData**

```
ResultSets rset = stmt.executeQuery("select ...");
ResultSetMetaData rsmd = rset.getMetaData();
```

- Abfragen von Spaltenanzahl, Spaltennamen und deren Typen

```
int anzahlSpalten = rsmd.getColumnCount();
String spaltenName = rsmd.getColumnName(1);
String typeName = rsmd.getColumnTypeName(1);
```

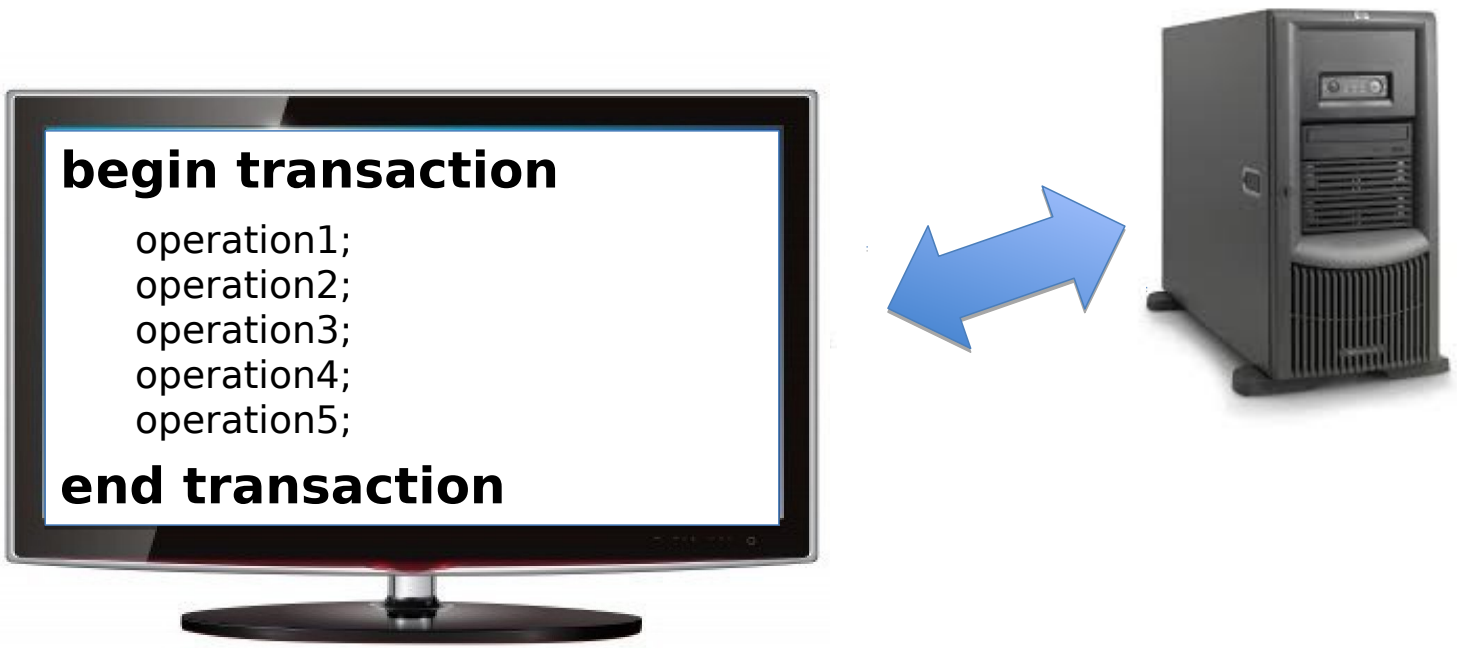
JDBC - Fehlerbehandlung

- Spezifikation der Ausnahmen, die eine Methode werfen kann, steht bei ihrer Deklaration (throws Exception)
- Wird Code in einem try-Block ausgeführt, werden im catch-Block Ausnahmen abgefangen.

```
try {
    //code ......
}
catch (SQLException e) {
  System.out.println("Es ist ein Fehler aufgetreten
  :");
  System.out.println("Msg: "+ e.getMessage());
  System.out.println("SQLState: "+ e.getSQLState())
  ;
  System.out.println("ErrorCode: "+ e.getErrorCode
  ());
  //und zum debuggen noch gleich dazu
  e.printStackTrace();
}
```

Anwendungsprogrammierung: Transaktionen

- Haben nun nicht nur eine einzelne SQL-Anweisung, sondern ganze Folge davon, je nach Anwendung.
- **Eine oder mehrere Anweisungen werden als Transaktion zusammengefasst bzw. betrachtet.** Z.B. Abheben von Geld am Geldautomat.
- Wird im Detail im **Kapitel über Transaktionen** betrachtet.



Anmerkung: SQL Injections

SQL Anfragen wird in Anwendung erstellt, wobei id eine Benutzereingabe ist

```
....”SELECT author, subject, text ” +
    ”FROM artikel WHERE ID=” + id
```

Aufruf z.B. durch Webserver `http://webserver/cgi-bin/find.cgi?ID=42`

SQL Injection zum Ausspähen von Daten

`http://webserver/cgi-bin/find.cgi?ID=42+UNION+SELECT+`
`login,+password,+’x’+FROM+user`

Führt zur SQL Anweisung:

select author, subject, text **from** artikel
where ID=42 **union select** login, password, 'x' **from** user;

Und andere Fälle bis hin zum Einschleusen von beliebigem Code auf Rechner + öffnen einer Shell (abh. von DBMS)

Hilfe u.a. durch Benutzen von PreparedStatements

Übersicht unter: `http://de.wikipedia.org/wiki/SQL-Injection`

Weitere Call-Level-Interfaces (CLIs) für Postgresql

Für C++: libpqxx

- http://pqxx.org/

Für Ruby: pg

- https://rubygems.org/gems/pg

```
require 'pg'
conn = PG::Connection.open(:host => 'localhost',
         :dbname => 'university', :user => 'username',
         :password => 'my password')
res = conn.exec("select name from studenten")
res.each do |row|
    puts row['name']
end
```

Call-level-Interface (CLI) vs. Embedded SQL

- Unter Verwendung einer Bibliothek werden aus dem Anwendungsprogramm (Wirtssprache) Funktionen aufgerufen, die mit dem DBMS kommunizieren.
- **JDBC ist ein Beispiel für ein CLI**
- **Im embedded SQL werden hingegen SQL Anweisungen direkt in der Wirtssprache benutzt**.
- (Dennoch werden diese letztendlich durch Aufrufe von DBMS-spezifischen Bibliotheken realisiert)
- Hier nur kurz erwähnt. Syntax von Embedded SQL (SQLJ) nicht klausurrelevant.

Embedded SQL (ESQL)

Idee

- Benutze SQL-Anweisungen direkt im Programmcode
- Syntax in Java:

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

- Syntax in C oder C++

```
EXEC SQL <sql-statement>;
```

Zum Beispiel:

```
1  EXEC SQL
2  SELECT vorname, nachname
3  INTO :vorname, :nachname
4  FROM mitarbeitertabelle
5  WHERE pnr = :pnr;
```

SQLJ: Embedded SQL für Java

- Einbettung von SQL in Java
- Anweisungen der Form

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

Zum Beispiel:

```
#sql {INSERT INTO emp (ename, sal)
      VALUES ('Joe', 43000) };
```

SQLJ: Embedded SQL für Java

Idee

- SQL Anweisungen werden direkt im Java Code benutzt (embedded)
- Ein Precompiler übersetzt diesen gemischten Code (in `*.sqlj` Dateien) in normalen Java Code (in `*.java` Dateien).
- Java Code benutzt dann JDBC oder Implementierung ähnlich zum JDBC Konzept.

SQLJ: Vor- und Nachteile im Vergleich zu JDBC

Vorteile

- Einfacher (kompakter) Code
- Verwendung der gleichen Variablen in SQL und in Java

Nachteile

- Erfordert extra Übersetzung in "normales" Java.

Umsetzung/Unterstützung

- Wird von Oracle angeboten (im eigenen DBMS)
- Sonst kaum (nicht) anzutreffen

Stored Procedures / User-Defined Functions

Stored Procedures/UDFs

Bislang: Kommunikation mit DBMS via Anwendungsprogrammen: Einzelne Statements, Verarbeitung in Wirtssprache.
Manchmal ist es aber sinnvoll, Teile der Anwendung direkt im DBMS auszuführen und nicht via einzelnen SQL Statements.

Vorteile

- Daten müssen nicht erst auf dem DBMS zur Anwendung gebracht werden (und umgekehrt)
- Höhere Performance
- Code kann wiederverwendet werden (zwischen Anwendungen)

Nachteile

- Etwas aufwendiger zu erstellen.
- Debugging schwieriger.

SQL vs. SQL/PSM vs. PL/SQL bzw. PL/pgSQL

SQL

- Standard Query Language für Datenbanksysteme. Deklarativ.
- SQL Anweisungen können via Anwendungsprogrammierung (JDBC oder Embedded SQL) an DB geschickt werden.

PSM

- Persistent, Stored Modules (PSM), bzw. Sprache diese zu realisieren.
- Im SQL:2003 Standard definiert. Erlaubt es prozeduralen Code direkt innerhalb der DB zu schreiben.

PL/SQL bzw. PL/pgSQL

- Procedural Language/(PostgreSQL) Structured Query Language
- Prozedurale Sprache, benutzt in Oracle bzw. Postgresql
- Inspiriert von bzw. implementiert PSM.
- PL/SQL bzw. PL/pgSQL erlauben diese Anwendungslogik als Prozedur innerhalb des DBMS zu definieren.

Vorteile von Stored Procedures / User Defined Functions

- **Ausführungspläne können vor-übersetzt werden**, sind **wiederverwendbar**
- **Anzahl der Zugriffe** des Anwendungsprogramms auf das DBMS werden reduziert, ebenso wie Menge an Daten, die zwischen Anwendung und DBMS hin und her geschickt wird.
- Prozeduren sind als **gemeinsamer Code** für verschiedene Anwendungsprogramme nutzbar
- Es wird ein **höherer Isolationsgrad** der Anwendung von dem DBMS erreicht.

Beispiel

```
CREATE  FUNCTION wieVieleVL (_matrnr int)
                                    RETURNS int AS $$
DECLARE
    qty int;
BEGIN
    SELECT COUNT(*) INTO qty
    FROM hoeren h
    WHERE h.matrnr = _matrnr;

    RETURN qty;
END;
$$ LANGUAGE plpgsql;
```

```
select *
from wieVieleVL(28106);
```

Liefert Ergebnis: 4

Unterschied Stored Procedures und UDFs

Generell

- UDF = user-defined function
- Stored procedure muss explizit mit CALL aufgerufen werden (`SQL EXEC CALL name`)
- UDF kann direkt in SQL ohne CALL benutzt werden

In Postgresql

- In Postgres gibt es allerdings keinen Unterschied zwischen stored procedures und UDFs
- `EXEC CALL` gibt es in Postgres nicht.
- UDF wird aufgerufen in SELECT statements, z.b.
  select from myFunction(44234234);

UDFs in Postgresql

Verschiedene Arten von UDFs

- Query language Funktionen (SQL)
- Procedural language Funktionen (PL/pgSQL, Perl, ...)
- Interne Funktionen
- C Funktionen
- PL/Java erlaubt auch die Nutzung von Java (`http://pgfoundry.org/projects/pljava/`)

`https://www.postgresql.org/docs/current/static/xfunc.html`

Query Language (SQL) Funktionen

Mit Hilfe des Schlüsselworts **LANGUAGE** wird angegeben welche Sprache zur Definition dieser Funktion benutzt wurde, hier SQL.

- Diese Funktion hat den Rückgabewert `void`
- Sie ist definiert als einfaches SQL delete Statement und besitzt auch keine Eingabeparameter.

```
CREATE FUNCTION clean_emp() RETURNS
void AS $$
    DELETE FROM emp
    WHERE salary < 0;
$$ LANGUAGE SQL;
```

Query Language Funktionen (2)

- Diese Funktion hat als Parameter ein Tupel der Relation **emp**, die neben Name des Mitarbeiters, dessen Gehalt (Salary), Alter und Raum (Als Point-Objekt) enthält.
- Der Rückgabewert ist Typ `numeric`

```
INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double_salary(emp)
 RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;
```

Query Language Funktionen (3)

```
CREATE FUNCTION addtoroom (professoren)
RETURNS int AS $$
select $1.raum+1 ;
$$ LANGUAGE SQL
```

Anwendung/Aufruf:

select name, addtoroom(professoren.*) from professoren;

Query Language Funktionen (4)

Hier wird eine Funktion definiert, die ein Dummy-Tupel für einen neuen Professor erzeugt (gemäß der Relation `professoren`):

```
CREATE FUNCTION neuerProf()
RETURNS professoren
AS $$
    SELECT 1 as persnr, text 'Unbekannt' AS name,
           text 'C3' as rang, 123 as raum;
$$ LANGUAGE SQL;
```

Anwendung zum Beispiel:
insert into professoren (select * from neuerProf());

Query Language Funktionen (5)

Diese Funktion hat mehrere Eingaben und mehrere Ausgaben:

```
CREATE FUNCTION sum_n_product
       (x int, y int, OUT sum int, OUT product int)
AS $$ SELECT $1 + $2, $1 * $2
$$ LANGUAGE SQL;
```

Beispielaufruf:

```
 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

Query Language Funktionen (6)

Rückgabewerte: Einzelne Zeilen vs. Tabellen

```
CREATE FUNCTION alleProfs()
RETURNS professoren
as $$
select * from professoren;
$$ LANGUAGE SQL;
```

Beispielaufruf:

```
select * from alleProfs();
```

```
 persnr |   name    | rang | raum
--------+-----------+------+------
   2125 | Sokrates | C4   |  226
(1 row)
```

Was macht `select alleProfs();` ?

Tabellen als Rückgabewerte: Table Functions

```
CREATE FUNCTION getProfs(int)
RETURNS TABLE(persnr int) AS $$
 SELECT persnr from professoren p
 WHERE p.persnr < $1 ;
$$ LANGUAGE SQL;


select * from getProfs(2130);

 persnr
--------
   2125
   2126
   2127
(3 rows)
```

PL/pgSQL

Anstelle von SQL in den vorherigen Beispielen wird nun PL/pgSQL betrachtet.

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

PL/pgSQL

Anstelle von SQL in den vorherigen Beispielen wird nun PL/pgSQL betrachtet.

```
CREATE FUNCTION sales_tax(subtotal real)
RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL

```
CREATE FUNCTION
    concat_selected_fields(in_t sometablename)
RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL

Funktion mit zwei Eingabeparametern und zwei Ausgabeparametern:

```
CREATE FUNCTION
            sum_n_product(x int, y int,
                  OUT sum int, OUT prod int)
AS $$
BEGIN
     sum := x + y;
     prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL: SELECT INTO

SQL Anfragen, die nur eine Zeile zurückliefern, können direkt in Variablen eingelesen werden, z.B.

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
```

Falls mehrere Ergebnisse geliefert werden, wird die erste Zeile benutzt. Durch die Angabe von STRICT, also

```
SELECT * INTO STRICT myrec FROM emp
WHERE empname = myname;
```

wird darauf geachtet, dass es nur genau ein Ergebnis gibt (ansonsten wird eine Exception geworfen).

Siehe **EXECUTE** für dynamische Anfragen und **PERFORM** für Anfragen ohne Ergebnis: `https://www.postgresql.org/docs/current/static/plpgsql-statements.html`

PL/pgSQL: Kontrollstrukturen

PL/pgSQL bietet die übliche Auswahl and Kontrollstrukturen wie IF-Statements und Schleifen (LOOP WHILE, FOR), EXIT (=break), CONTINUE

```
LOOP
    IF count > 0 THEN
        EXIT;
    END IF;
END LOOP;
```

PL/pgSQL: Kontrollstrukturen und SQL Anfragen

Über Anfrageergebnisse iterieren

```
CREATE OR REPLACE FUNCTION testit() RETURNS int as
$$
DECLARE
     myprofs RECORD;
     myint int = 0;
BEGIN
FOR myprofs in SELECT * FROM professoren
                        WHERE persnr<2130
LOOP
  myint = myprofs.persnr + myint;
END LOOP;
return myint;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL: IF Statement

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- dann muss die Zahl wohl NULL sein ...
    result := 'NULL';
END IF;
```

Siehe auch CASE statements.

PL/pgSQL: Weitere Befehle - RAISE NOTICE

Zum Ausgeben von Meldungen/Warnungen oder einfach zur zum Debuggen Ihreres PL/pgSQL Codes:

```
RAISE NOTICE 'Parameter x = % und y = %', x, y
```

PL/pgSQL: Exception Handling

Syntax

```
....
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
     ...
```

Beispiel

```
BEGIN
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
      --- bzw. aequivalent: WHEN SQLSTATE '22012' THEN
END;
```

Zusammenfassung UDFs bzw. PL/pgSQL

- Die Implementierung von Teilen der Anwendungslogik direkt im Datenbanksystem kann einige Vorteile haben. Z.B. Wiederverwendbarkeit von Code und dass Daten nicht bewegt werden müssen.
- Realisiert im DBMS durch Stored Procedures bzw. User Defined Functions (UDFs)
- Basierend auf prozeduraler Sprache (PSM=Persistent Stored Modules)
- Haben uns PL/pgSQL als Beispiel genauer angeschaut

Ausblick

- Integritätskontrolle im DBMS durch Trigger: Dort werden in PL/pgSQL Prozeduren geschrieben, die beim Eintreffen eines Ereignisses ausgeführt werden.