## Functions in PGPLSQL

functions in psql
create or replace function funcName()
return **s** returntype as **$$**
begin
raise notice 'Hello Birb!';
end;
**$$** language langName
The two $ are always necessary. Also note the returns with an s and the language at the end.
Which **MUST** be a **PROCEDURAL LANGUAGE**, so c++ doesn't work here.

Parameters are handled like in any language
func(x bigint, y bigint)
you can also define multiple return types
func(variadic a numeric[])
or a generic return
func(param anyelement)

Variable Declaration:
returns void as $$
DECLARE
x bigint; y bigint;
BEGIN ....

Variable manipulation: x := 6 + 4

if: IF n = 0 THEN RETURN 1;
(optional)ELSE RETURN 2; END IF;
ELSIF also possible (note elsif not elseif)

case x when 1, 2 then msg := 'one or two'; end case;
essentially this checks if x is 1 **OR** 2

case when x between 0 and 10 then ....
similar but with a range, both can be simulated by if else.

Exceptions: BEGIN z:= x / y;
EXCEPTION WHEN division-by-zero
THEN z:= 0; (or error rather) END;
if you want to catch all: WHEN others THEN
often used after exception: RAISE; (show error)

For Loop: For var IN query LOOP
statements END LOOP;
for r in SELECT * FROM ang LOOP
RETURN NEXT r; END LOOP; RETURN; END;
note that the return next doesn't return you store it in a buffer
and return it at the end of the function.
for infinite loops: FOR i IN 1..__max LOOP;

update and insert: INSERT INTO ANG VALUES(...);
UPDATE ang set salary = salary + 500
where name = 'dashie';
interestingly, after the where name = 'dashie' you can use if not found then (handle error)
this allows for easier error handling.

queries: **execute** 'SELECT * from ang' || into result; return result; END;

comments are done by either – or /* */ for multiline

anonymous function: you can omit the name and just write do $$ ...

cursor: declare curs CURSOR FOR query;
BEGIN OPEN curs; LOOP do something CLOSE curs; END;
Cursors are essentially just iterables.
cursors can also be unbound curs1 refcursor
or they can be parameterized curs3 cursor(arg)

### PL/pgSQL: Datentypen

- PL/pgSQL übernimmt alle SQL Datentypen:
  - Boolean:
  - Zahlen: int, integer, number
  - Strings, Datum, etc.
  - Arrays: alle Datentypen gefolgt von „[]", z.B. int[]
  - Weitere: JSON, etc.
- ... ergänzt mit zusätzlichen Datentypen:
  - var5 angestellter.id%type; -- abgeleiteter col.-Typ
  - var6 angestellter%rowtype; -- abgeleitet von Tabelle
  - var7 record; -- generischer Record This is a simple entry inside a qurey aka 1 person
  - var8 anyelement; -- generischer Typ gemäss Fn.-Argument, vgl. nachfolgend
  - curs1 refcursor; curs2 cursor ...; -- vgl. nachfolgend

return types: all of the above AND void, SETOF type (array of a type), TABLE, Trigger

| | FUNCTION | PROCEDURE |
|---|---|---|
| Use in an expression | ✔ | ✖ |
| Return a value | ✔ | ✖ |
| Return values as OUT parameters | ✔ (PG Spezialität) | ✔ (PG v14) |
| Return a single result set | ✔ (as table fn.) | ✔ |
| Return multiple result sets | ✖ | ✔ |
| Contain transactions | ✖ | ✔ |
| Make it run using ... | EXECUTE | CALL |

also note that type%rowtype is used like this:
r ang%rowtype -> for r in select * from ang;

important to know, you can always use these functions to manipulate queries, for example
select upper(name) from ang;
depending on the function you can also
select generatetab(1,10)

- Es gibt zusätzlich IN, OUT, INOUT
  `create function foo(IN p1 type)...`
  - IN: call by value; Variablen oder Ausdrücke als Argument
  - OUT: call by reference; nur Variablen als Argument
  - INOUT: beides

cast: cast(input as type);
cast(record.id as text);

stored procedures are nothing but a chaining of functions:
- Schritt 1 in PL/pgSQL: siehe Beispiel 3 (SP-Funktion mit in/out –Parametern)
- Schritt 2: Deklaration in Java/JPA (aka Registrierung in JPA):

```
@NamedStoredProcedureQuery(
  name = "MySum", -- JPA-Objekt
  procedureName = "mysum", -- Name der SP-Fn. (DB-Objekt)
  parameters = {
    @StoredProcedureParameter(mode = ParameterMode.IN, type = Double.class, name = "x"),
    @StoredProcedureParameter(mode = ParameterMode.IN, type = Double.class, name = "y"),
    @StoredProcedureParameter(mode = ParameterMode.OUT, type = Double.class, name = "sum")
  }
)
```

- Schritt 3: Call it!

```
StoredProcedureQuery query = this.em.createNamedStoredProcedureQuery("MySum");
query.setParameter("x", 1.23d);
query.setParameter("y", 4.56d);
query.execute();
Double sum = (Double) query.getOutputParameterValue("sum");
```

some good to know things: plain SQL is more efficient.
write variables lower case for sql
use cast over typename -> not select date '2022-06-07'

## Triggers

- sind DB-Objekte und immer einer Tabelle zugeordnet
- werden in Stored Procedures programmiert
- haben keine Parameter
- können nicht direkt aufgerufen werden
- werden vom DBMS beim Eintreten eines Events aufgerufen
- haben bei der Ausführung die Rechte ihres Owners

- Trigger can pass parameters to function
- FOR EACH [statement | row]
- Events: INSERT, UPDATE, DELETE, TRUNCATE, INSTEAD OF
- Function executes BEFORE or AFTER changes

statement is once, row means once per row, aka for the entire table.

- Before triggers can change contents of new row
- After triggers can only respond to what has happened e.g. Foreign Keys
- Return values of AFTER triggers will be ignored
- Triggers execute in alphabetical order

The INSTEAD Trigger can be use to avoid crashes:
-> INSTEAD OF UPDATE
-> ON UPDATE DO INSTEAD
for example, trying to update a read-only view -> INSTEAD OF UDPATE

- DDL für Triggers

```
CREATE | DROP | ALTER TRIGGER
  mytrigger …
  ON mytable …
```

- Syntax-Beispiel CREATE TRIGGER:

```
CREATE TRIGGER mytrigger
  AFTER INSERT OR UPDATE
  ON mytable
  FOR EACH ROW
  EXECUTE PROCEDURE mytriggerfn();
```

- Syntax-Beispiel Trigger-Fn. passend zu Trigger:

```
CREATE FUNCTION mytriggerfn()
RETURNS TRIGGER -- << Eigenes Schlüsselwort!
AS $$
  <<body>>
$$ language plpgsql;
```

  - Eine Trigger-Fn. hat keine Fn.-Parameter
  - Diese werden über Trigger-Fn.-Variablen übergeben, u.a.:
    - `TG_NARGS` Anzahl Parameter
    - `TG_ARGV[]` Array von Parametern als TEXT

### PL/pgSQL: Trigger-Fn.-Variablen

- TG_NAME Name des Triggers (TG)
- TG_WHEN BEFORE oder AFTER
- TG_LEVEL ROW od. STATEMENT
- TG_OP INSERT, UPDATE, DELETE, (TRUNCATE)
- TG_RELID OID der Tabelle
- TG_RELNAME Name der Tabelle
- TG_TABLE_SCHEMA Schema der Tabelle

```
CREATE OR REPLACE FUNCTION dt_trigger_func()
  RETURNS TRIGGER AS $$
BEGIN
  IF (TG_OP = 'INSERT') THEN
    NEW.creation_date := now();
  ELSIF (TG_OP = 'UPDATE') THEN
    NEW.modification_date := now();
  END IF;
  RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER dt_trigger
BEFORE INSERT OR UPDATE
ON mytable
FOR EACH ROW
EXECUTE PROCEDURE dt_trigger_func();
```

BEFORE-Trigger :
Falls RETURN null → wird Operation abgebrochen

Return types: RETURN NEW -> returns a new table/row
RETURN OLD -> returns the old table/row
(but could change other rows!)
RETURN NULL -> cancel operation.

running order: before statement, before row, after row, after statement ->
and of course alphabetically.

inside the Trigger functions you can use the variables that don't matter aka can be ANY -> user
or the entered user from the trigger -> NEW.user
or explicitly the old one -> OLD.user
and last user defined stuff like -> SELECT 'I'
which just places an I as the variable
or something like now() for timestamps.

Triggers make the database slower and harder to maintain.
some databases therefore let you disable them if you want.
On a table basis.
Also watch out for cascading effects of triggers they might cause something else to be deleted.

Stored Procedures are really helpful for security
They have all the prviliges, but only allow the user to do what the creator has predefined.

## JSON | SQL

testfdsgggggggggggggggggddddddddddddddddddddddddddddddddddggg