

```
<div style="display: flex;  
justify-content: center; flex-  
direction: column; height:  
70%"> <h1>PL/SQL</h1>  
<h6>Presentation von  
Michael Stenz</h6> </div>
```

Allgemein

```
<ul> <li style="display: flex; align-items: center; list-style: none;
position: relative;"> <span style="position: absolute; left:
-0.95em;">●</span> <p style="margin-top: 0">Entwickelt von </p>
 </li> <li>Procedural
Language extensions to SQL</li> <li>Seit Oracle 6.0 (1991)</li>
<li>Erweiterung mit jeder Oracle Version</li> </ul>
```

History

- **Pro*C** (< 1991): Oracle SQL Statements in C
- **PL/SQL 1.0** (1991): Sehr limitiert
- **PL/SQL 2.1**: Prozeduren, Funktionen, Packages
- **PL/SQL 2.2**: Calling Stored Functions in SQL => DBMS_SQL package & DDL statements
- **PL/SQL 2.3**: Binary PL/SQL programs
- **PL/SQL 2.4**: File I/O Support & PL/SQL Table/Record improvements (wie Arrays)
- **Oracle 19c**: Polymorphic Table Functions in the same package ->

Warum PL/SQL?

- Erweiterung durch 3GL-Elemente (Schleifen, Bedingungen, Prozeduren, Funktionen...)
- Auslagerung von Code/Business-logic in die Datenbank (Datenintegrität 👍, Keine Redundanz 👍)
- Vorbereitung und Speicherung Code (Precompiled im Datenbankcache ⚡)
- Zeit und Aktions gesteuerte Ausführung (Trigger, Scheduler)
- Komplexe probleme -> einfache sub-programme
- ...

Wo es eingesetzt wird:

Architektur

- **PL/SQL Engine ->**
Komponente von Oracle die
PL/SQL Blöcke ausführt
- Engine führt proceduralen
Code aus und sendet SQL
Statements an den
Database Server

Blockstruktur

```
<< label >> (optional)
DECLARE      -- Declarative part (optional)
-- Declarations of local types, variables, & subprograms

BEGIN        -- Executable part (required)
-- Statements (which can use items declared in declarative part)

[EXCEPTION  -- Exception-handling part (optional)
-- Exception handlers for exceptions (errors) raised in executable part]
END;
```

```
<div style="display: flex; justify-content: center; flex-direction: column; height: 70%"> <h1>Basics & Datenstrukturen</h1> </div>
```


Blockstruktur

Anonymer Block (nicht benannt)

```
BEGIN  
-- Statements  
END;
```

Blockstruktur

Lokale Prozedur (nur innerhalb des Blocks)

```
DECLARE
    PROCEDURE my_proc IS
    BEGIN
        -- Statements
    END;
BEGIN
    my_proc; -- Nur innerhalb des Blocks aufrufbar
END;
```

Blockstruktur

Stored Procedure (in DB gespeichert)

```
CREATE OR REPLACE PROCEDURE my_proc IS
BEGIN
    -- Statements
END;
```

Variablen

```
identifizier [CONSTANT] datentyp [NOT NULL] [:= | DEFAULT ausdruck]
```

```
DECLARE
    emp_count NUMBER(3) := 0;
    part_no NUMBER(4);
    in_stock BOOLEAN;
BEGIN
    select count(*) into emp_count from emp; --Variablen mittels statements zuweisen
END;
```

%Type

Datentyp einer Spalte oder Variable
Verhindert Probleme bei Änderungen

```
DECLARE
  v_name emp.e_vname%TYPE
  n_name emp.e_nname%TYPE
  tmpstr v_name%TYPE
BEGIN
```

[DEMO](#)

%RowType

Datentyp einer Zeile einer Tabelle (Cursor)

```
DECLARE
    dept_row dept%ROWTYPE;
BEGIN
    SELECT * INTO dept_row FROM dept WHERE deptno = 10;
END;
```

Records

Kann mehrere Variablen
verschiedener Datentypen
speichern
Deklaration mit %ROWTYPE
oder explizit

Datentypen

- SQL Types + PL/SQL Types
 - z.B BOOLEAN (Demo)
- Scalar (können subtypes haben)
- Composite
- Large Object (LOB)

Subtype

- Nur subset von von Basis Typ

```
SUBTYPE subtype_name IS base_type  
    { precision [, scale ] | RANGE low_value .. high_value } [ NOT NULL ]
```

[DEMO](#)

```
<div style="display: flex; justify-content: center; flex-direction: column; height: 70%"> <h1>Kontrollstrukturen</h1> </div>
```

IF-THEN-ELSIF

Klassische verzweigungen

```
IF <condition1> THEN
    <sequence_of_statements1>
ELSIF <condition2> THEN
    <sequence_of_statements2>
ELSE
    <sequence_of_statements3>
END IF;
```

CASE

Vergleichbar mit IF-ELSE - wird effizienter ausgeführt.
Hierbei wird der selector verwendet anstatt bool'schen Ausdrücken

```
CASE selector
  WHEN expression1 THEN
    sequence_of_statements1;
  WHEN expression2 THEN
    sequence_of_statements2;
  ...
  [ELSE
    sequence_of_statementsN;]
END CASE;
```

Searched CASE

Ohne Selector

```
CASE
  WHEN grade = 'A' THEN dbms_output.put_line('Excellent');
  WHEN grade = 'B' THEN dbms_output.put_line('Very Good');
  WHEN grade = 'C' THEN dbms_output.put_line('Good');
  WHEN grade = 'D' THEN dbms_output.put_line('Fair');
  WHEN grade = 'F' THEN dbms_output.put_line('Poor');
  ELSE dbms_output.put_line('Invalid grade');
END CASE;
```

Case als Ausdruck

```
grade := CASE  
  WHEN score >= 90 THEN 'A'  
  WHEN score >= 80 THEN 'B'  
  WHEN score >= 70 THEN 'C'  
  WHEN score >= 60 THEN 'D'  
  ELSE 'F'  
END;
```

Loop

Endlosschleife

Benötigt **EXIT;** statement zum beenden.

```
LOOP
    -- statements
    EXIT WHEN <condition>; -- Gleich wie If-Then EXIT;
END LOOP;
```

Benennung von Schleifen

Schleifen können benannt werden um mit **EXIT** oder **CONTINUE** gezielt zu springen.

```
<<loop1>>  
FOR i IN 1..10 LOOP  
    <<loop2>>  
    FOR j IN 1..10 LOOP  
        EXIT loop1 WHEN <condition>;  
    END loop;  
END loop;
```


While

```
WHILE <condition> LOOP  
    -- statements  
END LOOP;
```

For-Loop

Die zähler variable ist ein Integer, und muss nicht deklariert werden.

```
[<<loop_name>>]  
FOR <counter> IN [REVERSE] <lower_bound>..<higher_bound> LOOP  
    <sequence_of_statements> -- Counter kann innerhalb des loops nicht zugewiesen werden.  
END LOOP;
```

```
FOR i IN 1..10 LOOP  
    dbms_output.put_line(i);  
END LOOP;
```

```
<div style="display: flex; justify-content: center; flex-direction: column; height: 70%"> <h1>Cursors</h1> </div>
```

Cursor

Ein Cursor verwaltet den Zugriff auf einen Datensatz (ergebnis eines Select-Statements)

- 2 Arten:
 - Implicit: DML-Statements die nur eine Zeile returnen
 - Explicit: Queries die mehrere Zeilen returnen

Cursor

- PL/SQL öffnet einen Cursor
- SQL-Statement wird ausgeführt
- PL/SQL schließt den Cursor
- Variable für current row

Cursor

```
DECLARE
  CURSOR c1 IS SELECT ename, job FROM emp;
  my_ename emp.ename%TYPE;
  my_job emp.job%TYPE;
BEGIN
  OPEN c1; -- ???
  -- ???
  FETCH c1 INTO my_ename, my_job;
  CLOSE c1; -- ???
END;
```

Cursor

```
DECLARE
  CURSOR c1 IS SELECT ename, job FROM emp;
  my_ename emp.ename%TYPE;
  my_job emp.job%TYPE;
BEGIN
  OPEN c1; -- Führt das Select-Statement aus
  -- ???
  FETCH c1 INTO my_ename, my_job;
  CLOSE c1; -- ???
END;
```

Cursor

```
DECLARE
  CURSOR c1 IS SELECT ename, job FROM emp;
  my_ename emp.ename%TYPE;
  my_job emp.job%TYPE;
BEGIN
  OPEN c1; -- Führt das Select-Statement aus
  -- Setzt Cursor auf nächste Zeile & holt current row aus dem Puffer
  FETCH c1 INTO my_ename, my_job;
  CLOSE c1; -- ???
END;
```


Cursor

```
DECLARE
  CURSOR c1 IS SELECT ename, job FROM emp;
  my_ename emp.ename%TYPE;
  my_job emp.job%TYPE;
BEGIN
  OPEN c1; -- Führt das Select-Statement aus
  -- Setzt Cursor auf nächste Zeile & holt current row aus dem Puffer
  FETCH c1 INTO my_ename, my_job;
  CLOSE c1; -- Schließt den Cursor
END;
```

Cursor mit Parameter

```
DECLARE
CURSOR c_product (low_price NUMBER, high_price NUMBER)
IS
    SELECT *
    FROM products
    WHERE price BETWEEN low_price AND high_price;
BEGIN
    OPEN c_product(100, 200);
    -- Statements
    CLOSE c_product;
END;
```

Wichtige Cursor Attribute

Attribut	return type	Amount
%FOUND	Booolean	Wenn letztes Fetch eine Zeile Zurückgiebt
%NOTFOUND	Boolean	Gegenteil
%ISOPEN	Boolean	Ob der Cursor noch offen ist
%ROWCOUNT	number	Anzahl an gefetchten Rows

Cursor-FOR-Loop

Automatisches öffnen, fetchen, schließen und loopen

```
DECLARE
    CURSOR c1 IS SELECT ename, job FROM emp;
BEGIN
    FOR emp_rec IN c1 LOOP
        dbms_output.put_line(emp_rec.ename || ' ' || emp_rec.job);
    END LOOP;
END;
```

```
<div style="display: flex; justify-content: center; flex-direction: column; height: 70%"> <h1>Procedures, Functions, Package, Trigger, Jobs...</h1> </div>
```

Gruppen

- Aufrufgesteuert
 - Procedure -> Kein Rückgabewert
 - Function -> Rückgabewert
- Aktionsgesteuert
 - Trigger -> durch DML, DDL oder DB operations "getriggert"
- Zeitgesteuert
 - Jobs -> Zeitgesteuerte Aufgaben

Procedures

- Kein Rückgabewert + Kann Parameter haben
- Mittels **CALL**, **EXEC** oder **anonymen Block** aufrufbar (**ACHTUNG:** EXEC funktioniert nur in SQL*Plus oder SQL Developer)

```
CREATE [OR REPLACE] PROCEDURE proc_name [(param_list)] IS
    [lokale Deklarationen]
BEGIN
    ...
[EXCEPTION]
END;
```

Functions

- Rückgabewert + Kann Parameter haben
- Mittels **SELECT** oder **PL/SQL** aufrufbar
- Keine DML-Statements (müssen als procedure definiert werden)

```
CREATE [OR REPLACE] FUNCTION func_name [(param_list)] RETURN datatype IS
    [lokale Variablen]
BEGIN
    . . .
RETURN var_name; -- für Rückgabewert
[EXCEPTION]
END [func_name]
```


Demo - Fibonacci

[DEMO](#)

Parameter

Syntax: <name> <modus> <datentyp>

Typen: Grundtypen ohne Größenangaben

Modi:

Mode	Description
IN	übergibt den Wert (call by value)
OUT	gibt Werte von einer Prozedur zurück (call by reference)
IN OUT	übergibt den Wert und kann geändert werden (call by reference)

IN/OUT Example

```
CREATE OR REPLACE PROCEDURE format_phone
    (p_phone_no IN OUT VARCHAR2 ) IS
BEGIN
    p_phone_no := '(' || SUBSTR (p_phone_no,1,3) ||
                  ')' || SUBSTR (p_phone_no,4,3) ||
                  '-' || SUBSTR (p_phone_no,7);

END format_phone;
-----
DECLARE
    v_phone_no VARCHAR2(10) := '+431234567';
BEGIN
    format_phone(v_phone_no);
    DBMS_OUTPUT.PUT_LINE(v_phone_no); -- (+43) 123-4567
END;
```

Trigger

- Auf Schema oder Datenbank Ebene
- Vor oder Nach DDL/DML-Statements oder DB-operationen
- Nutzvoll für:
 - Logging
 - Datenintegrität
 - Automatisierung (z.B. Berechnungen)
 - ...

Trigger Syntax

Jobs

Code in Zeitintervallen, Daten... ausführen

Signatur:

```
DBMS_JOB.SUBMIT(  
    job          OUT  BINARY_INTEGER,  
    what         IN   VARCHAR2,  
    next_date    IN   DATE DEFAULT sysdate,  
    interval     IN   VARCHAR2 DEFAULT 'null',  
    no_parse     IN   BOOLEAN DEFAULT FALSE,  
    instance     IN   BINARY_INTEGER DEFAULT any_instance,  
    force        IN   BOOLEAN DEFAULT FALSE  
);
```

Packages

- Sammlung von Typen, Stored-Procedures, Stored-Functions
- Wie Klassen in Java
- **Package Specification** (Interface) und **Package Body** (Implementation)
- Modularität slides.md:586 👍, Datenkapselung 👍, Performance 👍
- ...

Package - Struktur

1. Public Variables
2. Public Procedures
3. Private Procedures
4. Private Variables
5. Local Variables

Package - Syntax

Spezifikation

```
CREATE [OR REPLACE] PACKAGE <package_name>  
IS | AS --Synonym  
    öffentliche typ- und object-deklarationen,  
    unterprogramm-spezifikationen  
END <package_name>;
```

Package - Syntax

Body

```
CREATE [OR REPLACE] PACKAGE BODY <package_name>  
IS | AS  
    öffentliche typ- und object-deklarationen,  
    unterprogramm-spezifikationen  
END package_name;
```

```
<div style="display: flex; justify-content: center; flex-direction: column; height: 70%"> <h1>Exceptions/Errors</h1> </div>
```

Exceptions

- Named system exceptions (z.B. siehe Image)
- Unnamed system exceptions
- Named programmer defined exceptions
- Unnamed programmer defined exceptions

Benutzerdefinierte Exceptions

Error_Number muss im Bereich von -20000 bis -20999 liegen

```
Raise_Application_Error (Error_Number, Error_Text, [Keep_Error_Stack])
```

Catch Exceptions

Exceptions können im **Exception** Block gefangen werden
SQLCODE/SQLERRM geben den letzten Fehlermessage/code zurück

```
...  
EXCEPTION  
    WHEN exception_name1 THEN -- handler sequence_of_statements1  
    WHEN exception_name2 THEN -- another handler sequence_of_statements2  
    ...  
    WHEN OTHERS THEN -- optional handler sequence_of_statements3  
END;
```

BEGIN

END;

Quellen

- <https://www.oracle.com/database/technologies/appdev/plsql.html>
- <https://docs.oracle.com/>
- [PL/SQL Basicscriptum - Mag. Johannes Tumfart](#)
- [Einführung PL/SQL - Mag. Johannes Tumfart](#)