

Datenbanksysteme

PL/SQL

Jan Haase

2025

Abschnitt 11

Themenübersicht

- Wiederholung Teil I

PL/SQL

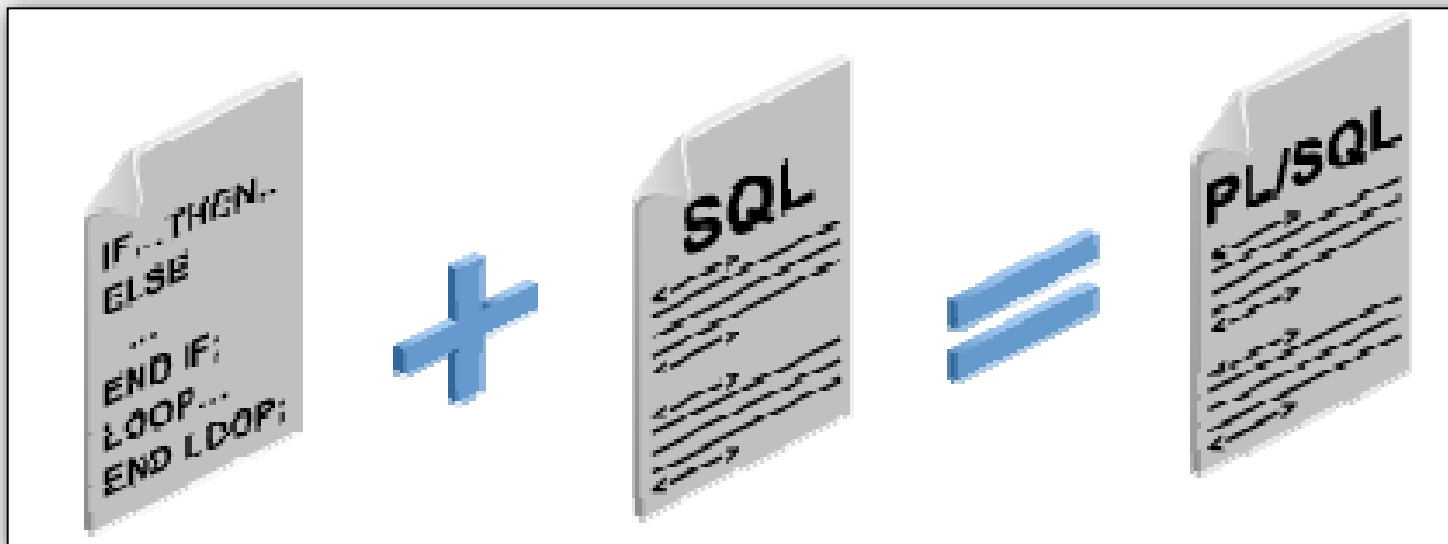
- DB-Tuning
- Stern-Schema-Modellierung
- Big Data, NoSQL
- Klausurvorbereitung

Themenübersicht

- Wiederholung Teil I
- **PL/SQL**
 - ➔ **Eigenschaften und Struktur**
 - Typen, Variablen, Kontrollstrukturen und Operatoren
 - Prozeduren und Funktionen
 - Ausnahmebehandlung
 - Trigger
 - Cursor
- DB-Tuning
- DB-Zugriff aus Programmiersprachen
- Big Data, NoSQL
- Business Intelligence, Data Warehouses
- Klausurvorbereitung

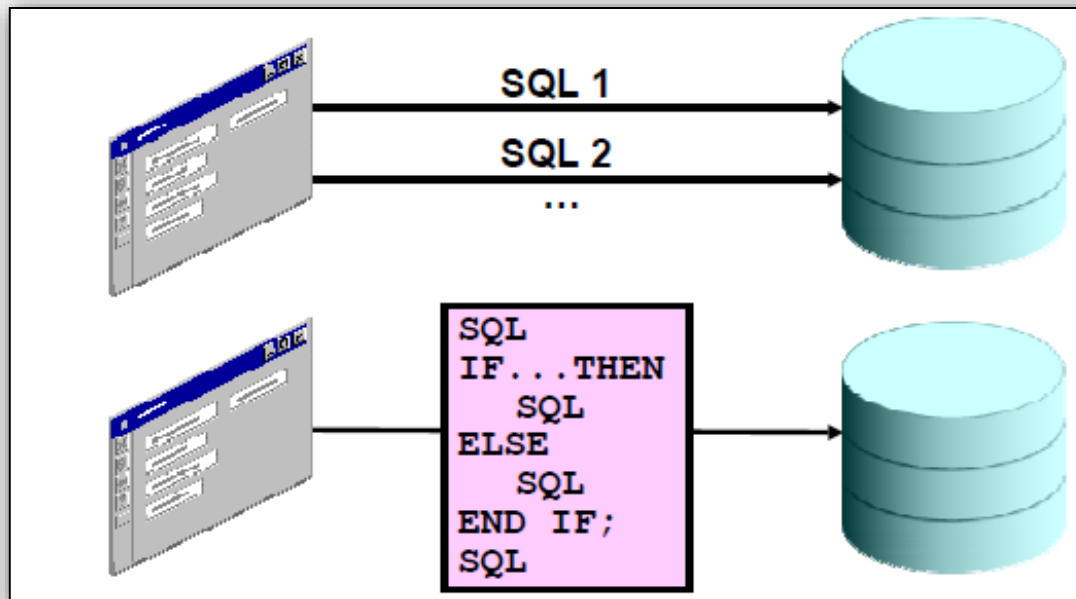
Was ist PL/SQL?

- PL/SQL steht für **Procedural Language / Structured Query Language**.
- Prozedurale **Erweiterung** der strukturierten Abfragesprache SQL.
- Oracle-Standardsprache für den Datenzugriff in relationalen Datenbanken.
- Integriert prozedurale Konstrukte nahtlos mit SQL.
- PL/SQL ist die Basis für SQL-Skripte, gespeicherte Prozeduren (Stored Procedures) und Trigger.



Eigenschaften von PL/SQL

- Definierte Blockstruktur für ausführbare Code-Einheiten
- Unterstützte prozedurale Konstrukte u. a.
 - Variablen, Konstanten und Typen
 - Kontrollstrukturen wie Bedingungsanweisungen und Schleifen
- Wiederverwendbare Programmeinheiten, die einmal erstellt und mehrfach ausgeführt werden
- Bessere Performanz durch Integration von prozeduralen Konstrukten mit SQL gegenüber dem DB-Zugriff aus prozeduralen Programmiersprachen



PL/SQL-Blockstruktur

Syntax:

DECLARE (optional)

Variablen, Cursor, benutzerdefinierte Exceptions

BEGIN (obligatorisch)

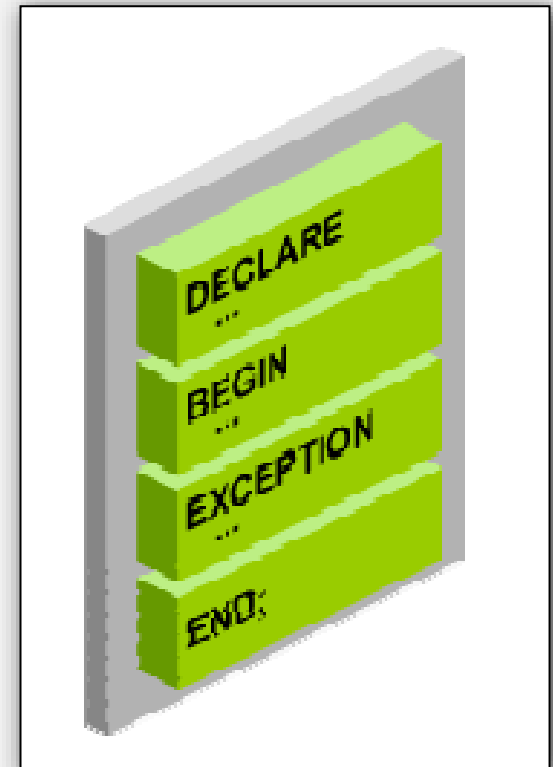
SQL-Anweisungen

PL/SQL-Anweisungen

EXCEPTION (optional)

Aktionen, die ausgeführt werden sollen,
wenn Fehler auftreten

END; (obligatorisch)



Themenübersicht

- Wiederholung Teil I
- **PL/SQL**
 - Eigenschaften und Struktur
 - ➡ **Typen, Variablen, Kontrollstrukturen und Operatoren**
 - Prozeduren und Funktionen
 - Ausnahmebehandlung
 - Trigger
 - Cursor
- DB-Tuning
- DB-Zugriff aus Programmiersprachen
- Big Data, NoSQL
- Business Intelligence, Data Warehouses
- Klausurvorbereitung

Blocktypen

Anonymer Block

```
[DECLARE]

BEGIN
    - statements

[EXCEPTION]

END;
```

Prozedur

```
PROCEDURE name
IS

BEGIN
    - statements

[EXCEPTION]

END;
```

Funktion

```
FUNCTION name
RETURN datatype
IS

BEGIN
    - statements
RETURN value;

[EXCEPTION]

END;
```

- Beispiel für einen anonymen Block:

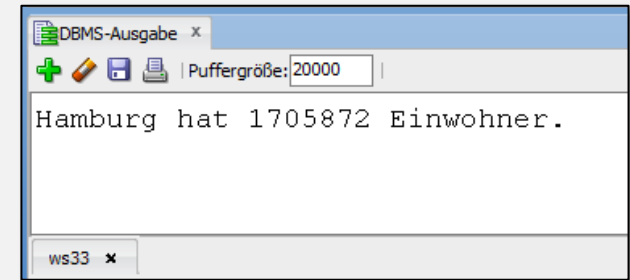
```
DECLARE
    F_Population NUMBER;
BEGIN
    SELECT Population INTO F_Population FROM City
    WHERE Name='Hamburg';
END;
```

Hier wird ein Wert aus einer SQL-Abfrage einer Variablen zugewiesen.

Jede PL/SQL-Anweisung wird mit einem Semikolon beendet.

Ausgaben in PL/SQL-Blöcken

- Verwenden Sie ein vordefiniertes Oracle-Package mit der entsprechenden Prozedur:
 - DBMS_OUTPUT.PUT_LINE
- Aktivieren Sie die Ausgabe mit dem Befehl
 - SET SERVEROUTPUT ON
 oder öffnen Sie die Ansicht DBMS-Ausgabe.



- Für das Beispiel:

```
SET SERVEROUTPUT ON;
```

```
DECLARE
```

```
    f_population NUMBER;
```

```
BEGIN
```

```
    SELECT Population INTO f_population FROM City
```

```
    WHERE Name = 'Hamburg';
```

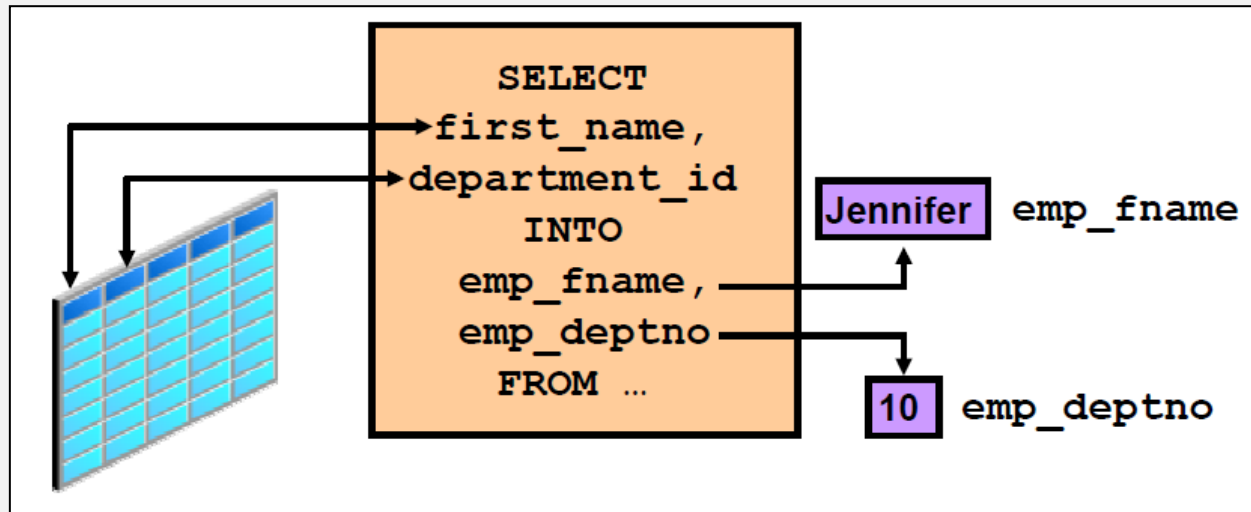
```
    DBMS_OUTPUT.PUT_LINE('Hamburg hat ' || f_population ||  
                          ' Einwohner.');
```

```
END;
```

Zeichenketten-Konkatenation

Variablen in PL/SQL

- Variablen können für folgende Aufgaben verwendet werden:
 - Temporäre Datenspeicherung
 - Bearbeitung gespeicherter Werte
 - Wiederverwendbarkeit



Bezeichner in PL/SQL

- Unterschied Variable \Leftrightarrow Bezeichner (Identifizier):
 - **Variablen** dienen als **Speicherort** für Daten. Daten sind im Speicher abgelegt. Variablen verweisen auf diesen Speicherort, über den Daten gelesen und bearbeitet werden können.
 - **Bezeichner** dienen der **Benennung** von PL/SQL-Objekten wie Variablen, Typen, Cursor und Unterprogrammen.
- Für Variablen-Bezeichner (Identifizier) gelten folgende Konventionen:
 - Müssen mit einem Buchstaben beginnen.
 - Können Buchstaben oder Zahlen enthalten.
 - Können Sonderzeichen wie Dollarzeichen, Unterstriche oder Nummernzeichen enthalten.
 - Dürfen eine Länge von maximal 30 Zeichen haben.
 - Dürfen keine reservierten Wörter enthalten.

Umgang mit Variablen in PL/SQL

- Variablen werden:
 - im deklarativen Bereich deklariert und initialisiert,
 - im ausführbaren Bereich verwendet und mit neuen Werten versehen,
 - als Parameter an PL/SQL-Unterprogramme übergeben und
 - zum Speichern der Ausgabe eines PL/SQL-Unterprogramms verwendet.

Deklaration:

```
Identifizier [CONSTANT] datatype [NOT NULL]
    [:= | DEFAULT expr];
```

- Beispiele:

```
DECLARE
    emp_hiredate    DATE;
    emp_Deptno      NUMBER(2) NOT NULL := 10;
    location        VARCHAR2(13) := 'Atlanta';
    c_comm          CONSTANT NUMBER := 1400;
```

Variablendeklaration im Detail

Syntax:

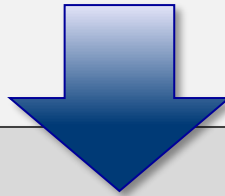
```
Identifizier [CONSTANT] datatype [NOT NULL]
    [:= | DEFAULT expr];
```

- **Identifizier** Name der Variablen
- **CONSTANT** Legt die Variable als Konstante fest, sodass ihr Wert nicht geändert werden kann. Konstanten müssen initialisiert werden.
- **Datatype** Steht für einen skalaren, zusammengesetzten, Referenz- oder Datentyp.
- **NOT NULL** Legt fest, dass die Variable einen Wert enthalten muss. NOT NULL-Variablen müssen initialisiert werden.
- **expr** Steht für einen PL/SQL-Ausdruck und kann ein Literal, eine andere Variable oder ein Ausdruck mit Operatoren und Funktionen sein.
- Die Zuweisung mit := und DEFAULT sind äquivalent.
- Nicht initialisierte Variablen haben den Wert NULL.

Beispiel für Umgang mit Variablen

```
DECLARE
  f_population NUMBER;
BEGIN
  SELECT Population INTO f_population
  FROM City WHERE Name='Hamburg';
  Dbms_Output.Put_Line('Hamburg hat ' || f_population ||
    ' Einwohner.');
```

END;



```
DECLARE
  f_city          CONSTANT VARCHAR2(7) := 'Hamburg';
  f_population NUMBER;
BEGIN
  SELECT Population INTO f_population
  FROM City WHERE Name=f_city;
  Dbms_Output.Put_Line(f_city||' hat ' || f_population ||
    ' Einwohner.');
```

END;

Begrenzungszeichen für Zeichenfolgenlitterale: Erläuterung

- Enthält Ihre Zeichenfolge ein Hochkomma, muss diesem ein weiteres Hochkomma voranstellen werden.
 - Beispiel: `event VARCHAR2(15) := 'Father' 's day' ;`
 - Das erste Hochkomma fungiert dabei als **Escape-Zeichen**. Besonders Zeichenfolgen, die aus SQL Anweisungen bestehen, werden dadurch recht kompliziert. Sie können jedes Zeichen als Begrenzungszeichen festlegen, das nicht bereits in der Zeichenfolge vorhanden ist.
- Sie beginnen die Zeichenfolge mit `q'`, wenn Sie ein Begrenzungszeichen verwenden möchten. Das Zeichen hinter der Notation ist das verwendete Begrenzungszeichen. Nachdem Sie das Begrenzungszeichen festgelegt haben, geben Sie Ihre Zeichenfolge ein. Schließen Sie das Begrenzungszeichen, und schließen Sie dann die Notation mit einem einfachen Anführungszeichen. Beispiel: `event := q'!Father's day!' ;`
- Klammern werden automatisch richtig behandelt. So können z. B. [und] als Begrenzungszeichen verwendet werden:
Beispiel: `event := q'[Mother's day]' ;`

Begrenzungszeichen für Zeichenfolgenlitterale: Beispiel

```
SET SERVEROUTPUT ON
DECLARE
  event VARCHAR2(15);
BEGIN
  event := q'!Father's day!';
  DBMS_OUTPUT.PUT_LINE('3rd Sunday in June is :
    '||event);
  event := q'[Mother's day]';
  DBMS_OUTPUT.PUT_LINE('2nd Sunday in May is :
    '||event);
END;
/
```

Definition
Begrenzungszeichen
mit q'

Begrenzungszeichen !

Begrenzungszeichen []

Trennt verschiedene Blöcke in
einem gemeinsamen SQL-Script

Ausgabe:

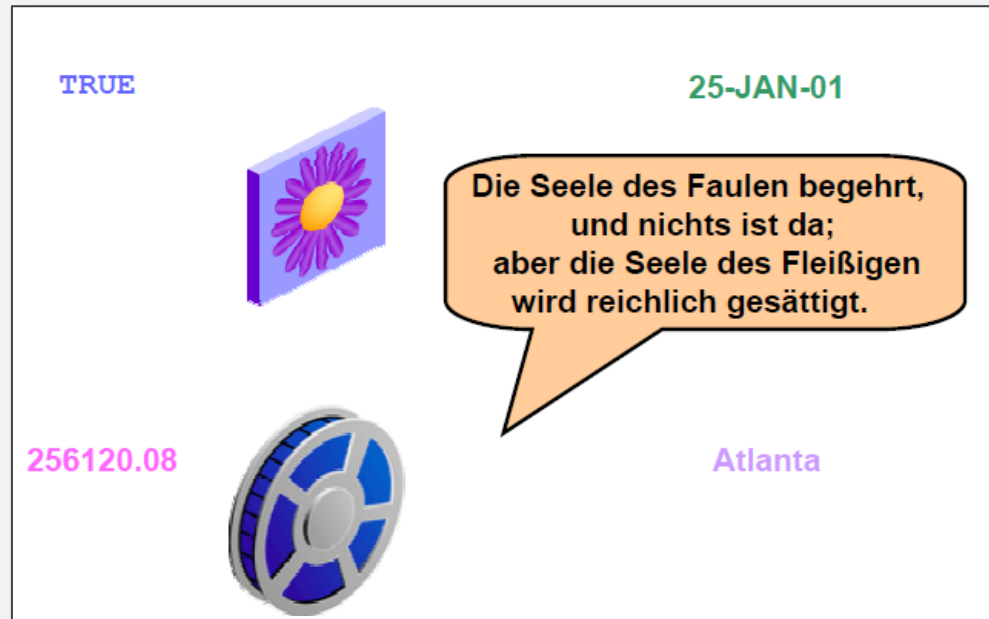
3rd Sunday in June is : Father's day
2nd Sunday in May is : Mother's day

Variablentypen

- **Skalar:** Einzelne Werte
- **Zusammengesetzt:** Zusammengesetzte Werte (Bündel von Variablen)
- **Referenz:** Referenzdatentypen enthalten Zeiger genannte Werte, die auf einen Speicherort verweisen.
- **LOB** (Large Objects): LOB-Datentypen enthalten Positionsanzeiger genannte Werte, die den Speicherort großer Objekte (zum Beispiel Grafiken) bezeichnen, die out-of-line gespeichert werden.

Variablentypen

- 256120.08 steht für den Datentyp **NUMBER** mit Dezimalstellen.
- Der Film steht für den Datentyp **BFILE**.
- Der Städtenamen Atlanta steht für den Datentyp **VARCHAR2**.



- TRUE steht für einen **Booleschen** Wert.
- 25-JAN-01 steht für den Datentyp **DATE**.
- Das Bild steht für den Datentyp **BLOB**.
- Der Text des Spruches kann für den Datentyp **VARCHAR2** oder **CLOB** stehen.

Richtlinien für Variablendeklaration und -initialisierung

- Beachten Sie die Benennungskonventionen.
- Verwenden Sie aussagekräftige Namen für Variablen.
- Initialisieren Sie als NOT NULL und CONSTANT definierte Variablen.
- Initialisieren Sie Variablen mit dem Zuweisungsoperator (:=) oder dem Schlüsselwort DEFAULT.
- Deklarieren Sie einen Identifier pro Zeile, um die Lesbarkeit und Code-Wartung zu verbessern.
- Verwenden Sie das NOT NULL-Constraint, wenn die Variable einen Wert enthalten muss.
- Vermeiden Sie Spaltennamen als Namen für Identifier.

```
DECLARE
    name VARCHAR2 (35) ;
BEGIN
    SELECT name
    INTO name
    FROM City
    WHERE LATITUDE =53.55;
END;
/
```



Skalare Datentypen (1/3)

- **CHAR** [(maximum_length)]
Basistyp für **Zeichendaten** fester Länge bis maximal 32.767 Byte. Wenn für maximum_length keine Maximallänge angegeben wird, wird die Default-Länge auf 1 eingestellt.
- **VARCHAR2** (maximum_length)
Basistyp für **Zeichendaten** variabler Länge bis maximal 32.767 Byte. Für VARCHAR2-Variablen und -Konstanten gibt es keine Default-Länge.
- **NUMBER** [(precision, scale)]
Basistyp für **Ziffern** mit einer festgelegten Anzahl der Nachkommastellen (scale) und Gesamtstellenzahl (precision). Die Anzahl der Gesamtstellen kann zwischen 1 und 38 betragen. Der gültige Wertebereich für die Nachkommastellenzahl ist -84 bis 127.
- **BINARY_INTEGER** bzw. **PLS_INTEGER**
Basistypen für **Ganzzahlen** zwischen -2.147.483.647 und 2.147.483.647. Die arithmetischen Operationen mit PLS_INTEGER- und BINARY_INTEGER-Werten sind schneller als mit NUMBER-Werten. Unterschiede der beiden Typen gering, z. B. Exceptions bei Overflows.

Skalare Datentypen (2/3)

- **BOOLEAN**

Basistyp, der einen von *drei* möglichen Werten speichert, die für **logische** Berechnungen verwendet werden: TRUE, FALSE oder NULL.

- **BINARY_FLOAT:**

Neuer Datentyp seit Oracle Database 10g für **Gleitkommazahlen** im IEEE 754-Format. Für die Speicherung des Wertes sind 5 Bytes erforderlich.

- **BINARY_DOUBLE:**

Neuer Datentyp seit Oracle Database 10g für **Gleitkommazahlen** im IEEE 754-Format. Für die Speicherung des Wertes sind 9 Bytes erforderlich.

Skalare Datentypen (3/3)

- **DATE:**

Basistyp für **Datumsangaben** und **Uhrzeiten**. DATE-Werte enthalten die Tageszeit in Sekunden seit Mitternacht. Der Datumsbereich liegt zwischen 4712 v. Chr. und 9999 n. Chr.

- **TIMESTAMP:**

Der Datentyp TIMESTAMP, eine Erweiterung des Datentyps DATE, speichert Jahr, Monat, Tag, Stunde, Minute, Sekunde und Sekundenbruchteil. Die Syntax lautet: TIMESTAMP[(precision)]. Der optionale Parameter precision gibt dabei die Stellenanzahl der Sekundenbruchteile im Feld SECONDS an. Sie können keine symbolischen Konstanten oder Variablen zur Angabe der Gesamtstellenzahl verwenden, sondern müssen ein ganzzahliges Literal zwischen 0 und 9 angeben. Der Default-Wert ist 6.

- **TIMESTAMP WITH TIME ZONE**

- **TIMESTAMP WITH LOCAL TIME ZONE**

- **INTERVAL YEAR TO MONTH**

- **INTERVAL DAY TO SECOND**

- Wird verwendet für die Variablendeklaration gemäß:
 - der Definition einer Datenbankspalte *oder*
 - einer anderen deklarierten Variablen
- Enthält als Präfix:
 - die Datenbanktabelle und –spalte *oder*
 - den Namen der deklarierten Variablen

In der Praxis sehr häufig anzutreffen, da **einfach und sicher!**


Syntax:

`Identifizier table.column_name%TYPE;`

- Beispiele:

```
...  
emp_lname      employees.last_name%TYPE;  
balance        NUMBER(7,2);  
min_balance    balance%TYPE := 1000;  
...
```

Zusammengesetzte Datentypen: Beispiele

| | | | |
|------|-----------|---------|-------------------------------------------------------------------------------------|
| TRUE | 23-DEC-98 | ATLANTA |  |
|------|-----------|---------|-------------------------------------------------------------------------------------|

Struktur einer
PL/SQL-Tabelle

| | |
|---|-------|
| 1 | SMITH |
| 2 | JONES |
| 3 | NANCY |
| 4 | TIM |

↑
PLS_INTEGER

↑
VARCHAR2

Struktur einer
PL/SQL-Tabelle

| | |
|---|------|
| 1 | 5000 |
| 2 | 2345 |
| 3 | 12 |
| 4 | 3456 |

↑
PLS_INTEGER

↑
NUMBER

Deklaration zusammengesetzter Datentypen mit RECORD

- Zuerst wird der Record-Typ deklariert:

```
TYPE typename IS RECORD (felddeklaration1
                        [, felddeklaration2,...]);
```

- **felddeklaration** steht dabei für die Deklaration des einzelnen Feldes und entspricht in der Syntax einer Variablendeklaration:

```
feldname datentyp [NOT NULL] [DEFAULT | := wert];
```

- Beispiel:

```
TYPE dept_rec_type is RECORD
    (deptnumber number(2),
      dname dept.dname%TYPE,
      loc varchar2(13));
```

- Anschließend kann dann eine Variable dieses Datentyps deklariert werden:

```
variable recordtyp;
```

- Defaultwerte oder NOT NULL sind hier nicht zulässig, da dies bei einem Record nur für einzelne Felder möglich ist.

- Beispiel: `dept_rec dept_rec_type;`

Referenzierung eines Records

- Ein einzelnes Feld in einem Record wird mit der Punktnotation angesprochen in der Form

`recordname.feldname`

- Beispiele:

```
emp_rec.empno := 4711;
```

```
IF emp_rec.ename = 'KING' THEN ....
```

```
IF emp_rec.comm IS NULL THEN ...
```

IF <Bedingung> THEN
ist eine einfache
Fallunterscheidung.
Details folgen.

- Ein **kompletter** Record kann nicht verglichen werden
(weder mit IS NULL noch mit Vergleichsoperatoren)!

Implizite Typdeklaration mit %ROWTYPE

- Mit %ROWTYPE wird die Struktur einer Datenbanktabelle komplett übernommen.
- Dadurch kann die Variablendeklaration in einem Schritt erfolgen:

```
variable tabellenname%ROWTYPE;
```

- Beispiel:

```
emp_rec emp%ROWTYPE;
```

```
dept_rec scott.dept%ROWTYPE;
```

- Die Variable **emp_rec** enthält nun die Felder **empno**, **ename**, **job**, **mgr**, **hiredate**, **sal**, **comm** und **deptno** mit den entsprechenden Datentypen und spiegelt damit die Tabelle emp komplett wider.

In der Praxis sehr häufig anzutreffen, da **einfach und sicher!**

Füllen von Records mit Select-Anweisungen

- Wir haben schon gesehen, wie skalare Variablen mit Select-Anweisungen befüllt werden können:

```
SELECT Population INTO f_population FROM City;
```

- Das funktioniert auch mit Records:

```
DECLARE
```

```
    Hamburg City%Rowtype;
```

```
BEGIN
```

```
    SELECT * INTO Hamburg FROM City WHERE Name =  
'Hamburg';
```

```
    Dbms_Output.Put_Line(Hamburg.Population);
```

```
END;
```

Kontrollstrukturen – Fallunterscheidungen

- Die aus anderen prozeduralen Programmiersprachen bekannte Fallunterscheidung mit **IF-THEN-ELSIF-ELSE** gibt es auch in PL/SQL.

- Form:

```

IF <Bedingung>
  THEN <Block>
  [ ELSIF <Bedingung> THEN <Block> ]
  ...
  [ ELSIF <Bedingung> THEN <Block> ]
  [ ELSE <Block> ]
END IF;

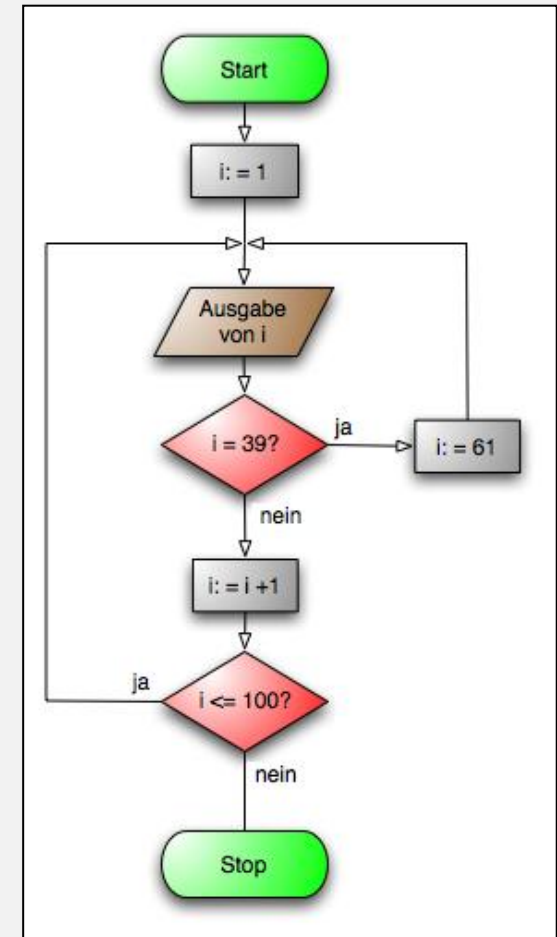
```

- Beispiel:

```

IF name IS NULL THEN
  zaehler:=zaehler+1;
  name2:='Unbekannt';
END IF;

```



Kontrollstrukturen – Einfache Schleifen

- Der einfachste Schleifentyp wird in PL/SQL mit **LOOP** gebildet.

- Form:

```
LOOP
```

```
    <Anweisungen>
```

```
END LOOP;
```



- Die Schleife kann mit EXIT oder EXIT WHEN <Bedingung> verlassen werden.

- Beispiel:

```
LOOP
```

```
    EXIT WHEN zaehler > anzahl;
```

```
    DBMS_OUTPUT.PUT_LINE(zaehler || '. ' || text);
```

```
    zaehler := zaehler + 1;
```

```
END LOOP;
```

Kontrollstrukturen – Zählschleifen

- Eine weitere bekannte Kontrollstruktur ist die **Zählschleife**.

- Form:

```
FOR <Laufvariable> IN [REVERSE] <Start> .. <Ende>
LOOP
    <Block>
END LOOP;
```



- Mit Reverse zählt die Schleife umgekehrt, also von Ende bis Anfang.

- Beispiel:

```
FOR zaehler IN 1 .. anzahl
LOOP
    DBMS_OUTPUT.PUT_LINE(zaehler || '. ' || text);
END LOOP;
```

Kontrollstrukturen – While-Schleifen

- Auch die **WHILE-Schleife** sollte bekannt sein.

- Form:

```
WHILE <Bedingung>  
  LOOP  
    <Block>  
  END LOOP;
```



- Beispiel:

```
WHILE zaehler <= anzahl  
  LOOP  
    DBMS_OUTPUT.PUT_LINE(zaehler || '. ' || text);  
    zaehler := zaehler + 1;  
  END LOOP;
```


Operatoren in PL/SQL

| | |
|------------------------------------------------------------------------------|--------------------------|
| <code>+, -</code> | Vorzeichen, Verneinung |
| <code>*, /</code> | Multiplikation, Division |
| <code>+, -</code> | Addition, Subtraktion |
| <code>**</code> | Exponentialoperator |
| <code> </code> | Konkatenation |
| <code>=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN</code> | Vergleiche |
| <code>NOT</code> | Logische Verneinung |
| <code>AND</code> | Logisches UND |
| <code>OR</code> | Logisches ODER |

Kommentare

- Stellen Sie einzeiligen Kommentaren zwei Bindestriche (--) voran.
- Setzen Sie mehrzeilige Kommentare zwischen die Symbole /* und */.

- Beispiel:

```
DECLARE
annual_sal NUMBER (9,2);
BEGIN -- Beginn the executable section
    /* Compute the annual salary beased on
    the monthly salary input from the user */
    annual_sal := monthly_sal * 12;
END; -- This is the end of the block
/
```

Übung 1

1. Bringen Sie den PL/SQL-Block, der die Einwohnerzahl von Hamburg ausgibt, zum Laufen und erzeugen Sie nach diesem Schema weitere Ausgaben (z. B. „Die Hauptstadt von Deutschland ist Berlin.“).
2. Erstellen Sie eine Tabelle „Belege“ mit den Feldern
 - Belegnummer (Ganzzahlig, Primärschlüssel),
 - Belegtext und
 - Betrag.
3. Schreiben Sie einen PL/SQL-Block, der in einer Schleife von 1 bis zu einem fest vorgegebenen n zählt und nur bei ungeradem n Belege in die Tabelle nach dem folgenden Schema einträgt:

| Belegnummer | Belegtext | Betrag | |
|-------------|-----------|--------------------|--------------------|
| 1 | Beleg 1 | 120 | |
| 3 | Beleg 3 | 160 | |
| 5 | Beleg 5 | 200 | |
| ... | | | |
| n | Beleg n | $100 + 20 \cdot n$ | (falls n ungerade) |

Themenübersicht

- Wiederholung Teil I
- **PL/SQL**
 - Eigenschaften und Struktur
 - Typen, Variablen, Kontrollstrukturen und Operatoren
- ➡ **Prozeduren und Funktionen**
 - Ausnahmebehandlung
 - Trigger
 - Cursor
- DB-Tuning
- DB-Zugriff aus Programmiersprachen
- Big Data, NoSQL
- Business Intelligence, Data Warehouses
- Klausurvorbereitung

PL/SQL-Prozeduren: Einführung

- **Prozeduren** fassen PL/SQL-Anweisungen zusammen, haben einen **Namen** und können **parametrisiert** werden.
- Vergleichbar mit Prozeduren/Methoden in anderen Programmiersprachen.
- **Funktionen** sind Prozeduren mit **Rückgabewert**.
- Prozeduren und Funktionen werden vom Oracle-DBMS in kompilierter Form gespeichert und verwaltet.
- Ihre Ausführung kann an Rechte geknüpft werden.
- Sie können manuell gestartet werden.
- Über Trigger können Prozeduren auch automatisiert angestoßen werden, sobald bestimmte Ereignisse eintreten (z. B. Einfügen eines Datensatzes).

PL/SQL-Prozeduren: Definieren und ausführen

Syntax:

```
CREATE [OR REPLACE] PROCEDURE <Prozedurname>
    [(<Parameterliste>)] IS
    <LokaleVariablen>
    BEGIN
        <Prozedurrumpf>
    END;
```

- Beispiel:

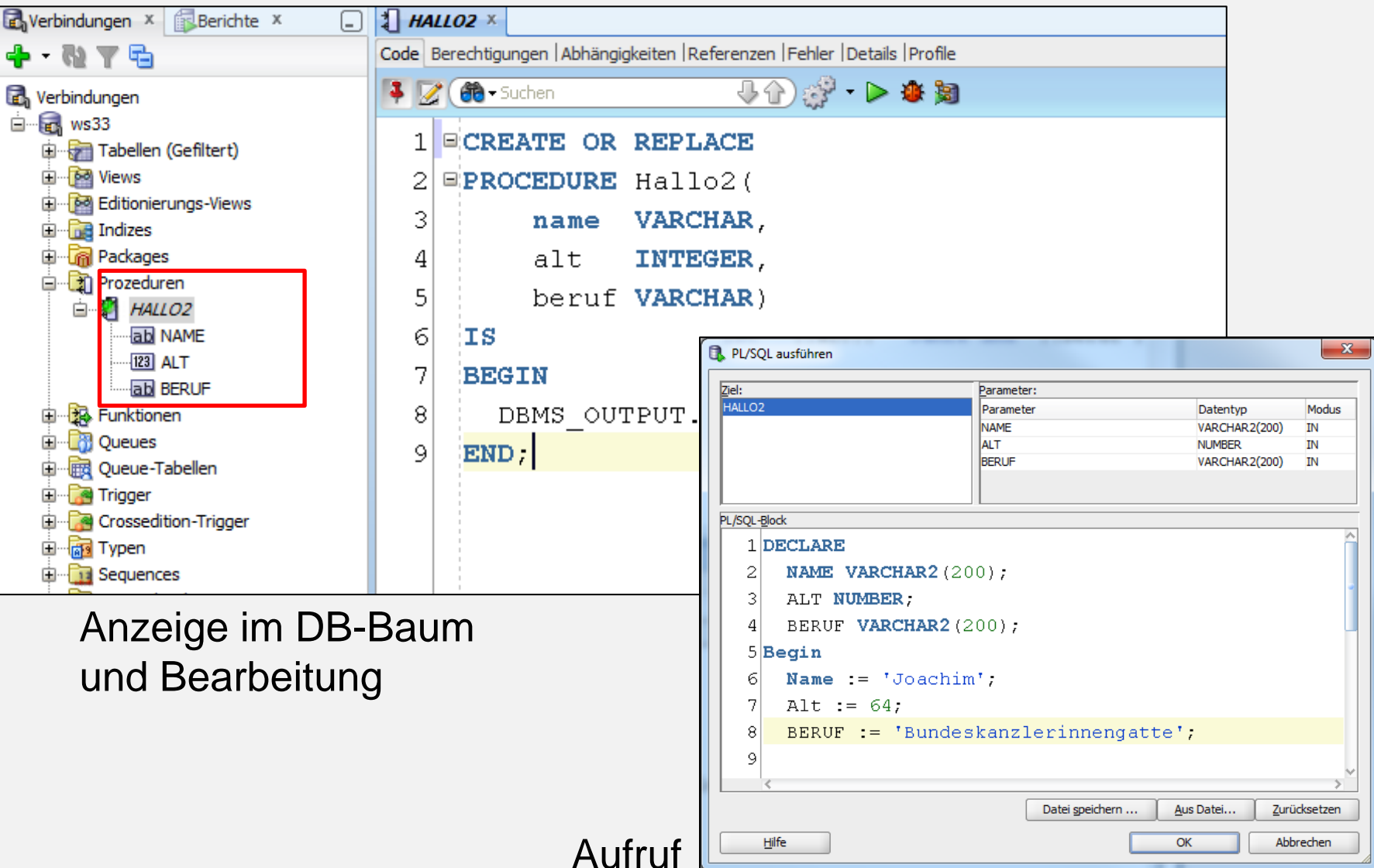
```
CREATE OR REPLACE PROCEDURE Hallo2
    (name VARCHAR, alt INTEGER, beruf VARCHAR)    IS
BEGIN
    DBMS_OUTPUT.PUT_LINE(name||' ist ' ||alt||
        ' Jahre und ' ||beruf || ' von Beruf.');
```

END;

- Prozedur ausführen:

```
EXECUTE Hallo2('Olaf', 66, 'Bundeskanzler');
```

Prozedurbearbeitung und –aufruf im SQL Developer



The screenshot displays the SQL Developer interface. On the left, the 'Verbindungen' (Connections) pane shows a tree structure for 'ws33'. Under 'Prozeduren' (Procedures), the procedure 'HALLO2' is highlighted with a red box, and its parameters 'NAME', 'ALT', and 'BERUF' are listed below it. The main 'Code' editor shows the definition of the 'HALLO2' procedure:

```

1 CREATE OR REPLACE
2 PROCEDURE Hallo2 (
3     name VARCHAR,
4     alt  INTEGER,
5     beruf VARCHAR)
6 IS
7 BEGIN
8     DBMS_OUTPUT.
9 END;

```

Below the code editor, the 'PL/SQL ausführen' (Execute PL/SQL) dialog is open. It shows the 'Ziel:' (Target) as 'HALLO2'. The 'Parameter:' section contains a table with the following data:

| Parameter | Datentyp | Modus |
|-----------|---------------|-------|
| NAME | VARCHAR2(200) | IN |
| ALT | NUMBER | IN |
| BERUF | VARCHAR2(200) | IN |

The 'PL/SQL-Block' section contains the following code:

```

1 DECLARE
2     NAME VARCHAR2(200);
3     ALT  NUMBER;
4     BERUF VARCHAR2(200);
5 Begin
6     Name := 'Joachim';
7     Alt  := 64;
8     BERUF := 'Bundeskanzlerinnengatte';
9

```

At the bottom of the dialog, there are buttons for 'Datei speichern ...', 'Aus Datei...', 'Zurücksetzen', 'Hilfe', 'OK', and 'Abbrechen'.

Anzeige im DB-Baum
und Bearbeitung

Aufruf

PL/SQL-Funktionen

- Eine Funktion ist der Prozedur ähnlich. Die Funktion liefert jedoch einen Wert zurück, die Prozedur hingegen nicht.

Syntax:

```
CREATE [OR REPLACE] FUNCTION <Funktionsname>
    (<Parameterliste>) RETURN <Ergebnistyp> IS
    <LokaleVariablen>
BEGIN
    <Funktionsrumpf>
END;
```

- Beispiel:

```
CREATE OR REPLACE FUNCTION Addiere (zahl1 INTEGER, zahl2 INTEGER)
RETURN INTEGER IS
    ergebnis INTEGER DEFAULT 0;
BEGIN
    ergebnis := zahl1 + zahl2;
    RETURN ergebnis;
END;
```

Es muss ein Wert zurückgegeben werden, sonst kommt es zu einem Laufzeitfehler.

Themenübersicht

- Wiederholung Teil I
- **PL/SQL**
 - Eigenschaften und Struktur
 - Typen, Variablen, Kontrollstrukturen und Operatoren
 - Prozeduren und Funktionen
- ➡ **Ausnahmebehandlung**
 - Trigger
 - Cursor
- DB-Tuning
- DB-Zugriff aus Programmiersprachen
- Big Data, NoSQL
- Business Intelligence, Data Warehouses
- Klausurvorbereitung

Anwendungsfehler: Vordefinierte Fehler

- Bei Laufzeitfehlern werden vordefinierte Fehler (**Exceptions**) geworfen, die im Exception-Block abgefangen werden können.



- Beispiel:

```
CREATE OR REPLACE PROCEDURE exTest IS
    I INTEGER DEFAULT 0;
BEGIN
    I := I/I;
    DBMS_OUTPUT.PUT_LINE('Nicht Erreicht');
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE(' || SQLCODE || ' : ' ||
SQLERRM) ;
END;
```

Abfangen &
behandeln

- Ergebnis:

-1476 : : ORA-01476: Divisor ist Null

SQLCODE SQLERRM – Error Message

Exceptions: Definieren und auslösen

- Exceptions können auch selbst definiert werden.



```
CREATE OR REPLACE FUNCTION noHeinz(name VARCHAR)
    RETURN VARCHAR IS
    heinzException EXCEPTION;
BEGIN
    IF name='Heinz' THEN RAISE heinzException;
    ELSE RETURN name;
    END IF;
END;
```

Exception
definieren

Exception
auslösen

Exceptions: Behandeln

Syntax:

```
WHEN <Exceptiontyp1>
    THEN <Block1>
...
[WHEN <ExceptiontypN>
    THEN <BlockN>]
WHEN OTHERS
    THEN <Block>]
```



● Beispiel:

```
CREATE OR REPLACE FUNCTION noHeinz (name VARCHAR2)
RETURN VARCHAR2 IS HeinzException EXCEPTION;
BEGIN
    IF name='Heinz' THEN RAISE HeinzException;
    ELSE Return name;
    END IF;
EXCEPTION
    WHEN HeinzException THEN RETURN 'Ein Heinz! ';
    WHEN OTHERS THEN RETURN 'Fehler: ';
END;
```

Anwendungsfehler: Fehler mittels Funktion auslösen

- Mit `RAISE_APPLICATION_ERROR` können Fehler erzeugt werden, die dann abgefangen werden können.



Syntax:

`RAISE_APPLICATION_ERROR(<Nummer>,<Fehlertext>)`

Die Nummer muss zwischen -20999 und -20000 liegen.

- Beispiel:

```
CREATE OR REPLACE FUNCTION noHeinz (name VARCHAR2)
RETURN VARCHAR2 IS
BEGIN
    IF name='Heinz' THEN
        RAISE_APPLICATION_ERROR(-20001,'Heinz ist nicht
erlaubt!');
    ELSE Return name;
    END IF;
END;
```

Themenübersicht

- Wiederholung Teil I
- **PL/SQL**
 - Eigenschaften und Struktur
 - Typen, Variablen, Kontrollstrukturen und Operatoren
 - Prozeduren und Funktionen
 - Ausnahmebehandlung
- ➡ **Trigger**
 - Cursor
- DB-Tuning
- DB-Zugriff aus Programmiersprachen
- Big Data, NoSQL
- Business Intelligence, Data Warehouses
- Klausurvorbereitung

Trigger: Einführung

- Datenbank-**Trigger** reagieren auf **definierte Ereignisse**.
- Wird ein Ereignis ausgelöst, so startet der Trigger **automatisch** und arbeitet ein PL/SQL Programm ab.
- Ereignisse können z. B. Änderungen an Tabellen (UPDATE, INSERT, ...) sein.
- Trigger können tückisch sein, wenn man nicht (mehr) weiß, dass sie existieren, und sich das System unerwartet verhält.



Trigger: Syntax

Syntax:

```
CREATE [OR REPLACE] TRIGGER <Triggername>
{BEFORE | AFTER} ← Vorher oder nachher abarbeiten ?
{INSERT | DELETE | UPDATE} [OF {Spaltenliste}] ← Worauf reagieren?
[OR {INSERT | DELETE | UPDATE} [OF {Spaltenliste}]] (DML Befehl)
...
[OR {INSERT | DELETE | UPDATE} [OF {Spaltenliste}]]
ON <Tabellenname> ← Bei welcher Tabelle aktiv werden?
[FOR EACH ROW] ← Für jeden adressierten Datensatz wiederholen?
[WHEN <Bedingung>] ← Optional: Bestimmte Bedingung muss erfüllt sein
<PL/SQL-Block>;
```



Trigger: Zugriff auf alte und neue Werte

- Die Angabe von **FOR EACH ROW** und die Nutzung von **BEFORE** ermöglicht auch den genauen Zugriff auf die **geänderten Werte**.
- Dafür gibt es zwei spezielle Variablen
 - :NEW und
 - :OLD,die jeweils vom Typ <Tabellenname>%ROWTYPE sind.

:NEW.FieldName

:OLD.FieldName



Trigger: Beispiel (1/3)

- Beim Einfügen eines Datensatzes soll ein Protokoll mitgeschrieben werden.

- Definition der Protokolltabelle:

```
CREATE TABLE Zooprotokoll(
    nr INT,
    wann DATE,
    wer VARCHAR(255),
    gehege INTEGER,
    tiername VARCHAR(12),
    PRIMARY KEY(nr)
);
```

- Definition einer Sequenz:

```
CREATE SEQUENCE zoozaehler INCREMENT BY 1 START WITH 1;
```

- Abfragen des aktuellen [nächsten] Standes der Sequenz in PL/SQL:

```
SELECT zoozaehler.CURRVAL [zoozaehler.NEXTVAL] FROM
DUAL;
```

Trigger: Beispiel (2/3)

Definition des Triggers:

```
CREATE OR REPLACE TRIGGER neuesTier
BEFORE INSERT ON TIER
FOR EACH ROW
DECLARE
    datum DATE;
    nutzer USER_USERS.USERNAME%Type;
BEGIN
    SELECT SYSDATE INTO datum FROM DUAL;
    SELECT USERNAME INTO nutzer FROM USER_USERS;
    INSERT INTO Zooprotokoll VALUES
        (zoozaehler.NEXTVAL, datum, nutzer, :NEW.gehege,
        :NEW.name) ;
END;
```

Trigger: Beispiel (3/3)

- Dann werden folgende Statements ausgeführt:

```
INSERT INTO Tier VALUES (1, 'Pooh', 'Baer', 1);
INSERT INTO Tier VALUES (2, 'Paddington', 'Baer', 1);
INSERT INTO Tier VALUES (3, 'Shir Khan', 'Tiger', 2);
```

- Ergebnis des Triggers:

```
SELECT * FROM Zooprotokoll;
```

| NR | WANN | WER | GEHEGE | TIERNAME |
|----|----------|---------|--------|------------|
| 2 | 28.10.14 | HERMANN | 1 | Pooh |
| 3 | 28.10.14 | HERMANN | 1 | Paddington |
| 4 | 28.10.14 | HERMANN | 2 | Shir Khan |

Instead-of-Trigger: Einführung und Syntax

- Trigger wird abgearbeitet, nicht die Folgen des auslösenden Ereignisses.
- Mögliche Anwendung:
 - INSERT in eine VIEW. Der Trigger regelt dann, wie in eine VIEW Daten einzufügen sind.

Syntax:

```
CREATE [OR REPLACE] TRIGGER <Triggername>  
INSTEAD OF  
{INSERT | DELETE | UPDATE} [OF {Spaltenliste}]  
ON <Viewname>  
[FOR EACH ROW]  
<PL/SQL-Block>;
```

Instead of Trigger: Beispiel (1/2)

- Gegeben sei folgende VIEW:

```
CREATE OR REPLACE VIEW Gesamt AS
  SELECT Tier.Name, Tier.Art, Art.MinFlaeche,
         Gehege.GNr, Gehege.GName
  FROM Tier, Art, Gehege
 WHERE Gehege.GNr = Tier.Gehege
        AND Tier.Art = Art.Gattung;
```

Instead of Trigger: Beispiel (2/2)

```
CREATE OR REPLACE TRIGGER tierInGesamt
INSTEAD OF
  INSERT ON GESAMT
FOR EACH ROW
  DECLARE
    zaehler INTEGER;
  BEGIN
    SELECT COUNT(*) INTO zaehler FROM Gehege
                                WHERE Gehege.GNr = :NEW.GNr;
    IF zaehler = 0 /* dann neues Gehege */ THEN
      INSERT INTO Gehege VALUES (:NEW.GNr, :NEW.GName, 50);
    END IF;
    SELECT COUNT(*) INTO zaehler FROM Art
                                WHERE Art.Gattung=:NEW.Gattung;
    IF zaehler=0 /* dann neue Art */ THEN
      INSERT INTO Art VALUES (:NEW.Gattung, :NEW.MinFlaeche);
    END IF;
    INSERT INTO Tier VALUES (:NEW.Gnr,:NEW.TName, :NEW.Gattung);
  END;
```

Was macht
dieser Trigger?

Themenübersicht

- Wiederholung Teil I
- **PL/SQL**
 - Eigenschaften und Struktur
 - Typen, Variablen, Kontrollstrukturen und Operatoren
 - Prozeduren und Funktionen
 - Ausnahmebehandlung
 - Trigger
- ➡ **Cursor**
- DB-Tuning
- DB-Zugriff aus Programmiersprachen
- Big Data, NoSQL
- Business Intelligence, Data Warehouses
- Klausurvorbereitung

Cursor: Einführung

- **Cursor** erlauben es, das Ergebnis einer (SQL)-Abfrage **schrittweise** abzuarbeiten.
- Es ist nicht möglich, auf alle Zeilen des Ergebnisses gleichzeitig zuzugreifen.
- Der aktuelle Zugriff ist immer nur auf **eine Zeile** möglich, wobei man den Cursor anweisen kann, zur nächsten Ergebniszeile zu gehen.
- Mit einem Cursor kann man durch eine Ergebnismenge „durchiterieren“.



Cursor: Definition und Nutzung

- Cursor werden als **spezieller Datentyp** für eine konkrete Anfrage definiert:

```
CURSOR <Cursorname> [ (Parameterliste) ] IS  
  <Datenbankanfrage>;
```
- Durch die Definition eines Cursors wird die zugehörige Anfrage noch **nicht ausgeführt**. Dies passiert erst bei der Nutzung des Cursors, z. B. durch

```
OPEN <Cursorname> [ (Argumente) ] ;
```
- Die Ergebnisse des Cursors können dann **zeilenweise** eingelesen werden. Zum Einlesen wird eine Variable benötigt, die zum Ergebnis des Cursors passt. Der zugehörige Record-Typ heißt `<Cursorname>%ROWTYPE`, sodass eine passende Variable durch `<Variablenname>
<Cursorname>%ROWTYPE;` deklariert wird.
- Das eigentliche **Einlesen** einer Ergebniszeile geschieht mit dem Befehl

```
FETCH <Cursorname> INTO <Variablenname>;
```
- Danach kann der erhaltene Record wie bereits bekannt bearbeitet werden.
- Nachdem ein Cursor abgearbeitet wurde, ist er mit `CLOSE <Cursorname>;` wieder zu **schließen**.

Cursor: Beispiel (Einzelzugriff auf Daten)

- Cursor definieren und Variable für Daten deklarieren:

```
DECLARE
```

```
    CURSOR continent_cur IS  
        SELECT name FROM continent;  
    my_name continent.name%Type;
```

- Cursor öffnen und Daten holen:

```
BEGIN
```

```
    OPEN continent_cur;                -- Cursor öffnen  
    FETCH continent_cur INTO my_name;  -- erste Zeile holen  
    dbms_output.put_line(my_name);  
    FETCH continent_cur INTO my_name;  -- zweite Zeile holen  
    dbms_output.put_line(my_name);  
    CLOSE continent_cur;               -- Cursor schließen
```

```
END;
```

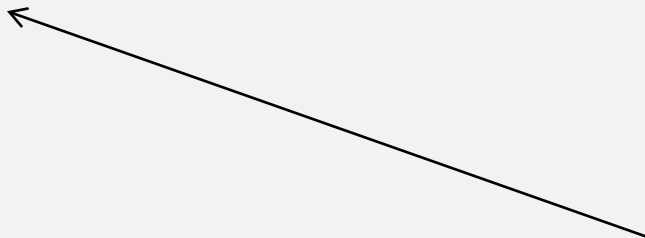
Verbesserungsidee?

Cursor: Beispiel (Iterative Verarbeitung)

- Alle Zeilen durchiterieren:

```
SET SERVEROUTPUT ON
DECLARE
    CURSOR continent_cur IS
        SELECT name FROM continent;
BEGIN
    FOR myrec IN continent_cur
    LOOP
        dbms_output.put_line(myrec.name) ;
    END LOOP;
END;
```

Schleife, bis Cursor am
Ende angelangt ist



Übung 2

1. Füllen Sie die Tabelle „Belege“ aus der letzten Übungsaufgabe mit weiteren Belegen, auch mit gerader Belegnummer. Lassen Sie dabei Lücken bei den Belegnummern. Löschen Sie auch einige Belege mit ungerader Nummer.
2. Schreiben Sie eine PL/SQL Funktion, welche die Anzahl der Lücken in den Belegnummern zurückgibt.
Beispiel: Belege 1, 3, 4, 5, 9 vorhanden → 2 und 6 bis 8 fehlen → 4 Lücken.
3. Schreiben Sie einen Trigger, der mittels dieser Funktion bei Änderungen an der Tabelle die Anzahl der Lücken ausgibt oder in eine zusätzliche Tabelle loggt.

Zusatzaufgabe für alle schnellen Teilnehmer:

4. Schreiben Sie eine weitere PL/SQL Prozedur, welche die Lücken mit Datensätzen füllt und dann als Belegtext „Lueckenfueller“ einsetzt und bei Betrag 0.
5. Schreiben Sie eine PL/SQL Prozedur, welche alle Lückenfüller wieder aus der Tabelle entfernt.

Übung 3 (Aus einer Klausur)

Ein Marktforschungsinstitut hat erhoben, welche Baumärkte welche Produkte verkaufen. Dazu wurde auch noch erhoben, welche Personen bei welchem Baumarkt einkaufen und welche Personen welche Produkte bevorzugt im Baumarkt kaufen. Dazu wurden die folgenden drei Tabellen in einer Datenbank angelegt und es werden auszugsweise Daten dargestellt:

Bevorzugt

| Kunde | Produkt |
|---------|-----------|
| Schulze | Schrauben |
| Müller | Werkzeug |
| Müller | Blumen |
| Schmidt | Möbel |
| Schmidt | Lampen |
| Horst | Blumen |
| Meier | Werkzeug |

Baumarktkunden

| Kunde | Baumarkt |
|---------|-----------|
| Müller | Hornbach |
| Schulze | Hornbach |
| Müller | OBI |
| Schmidt | Marktkauf |
| Horst | TOOM |

Produktangebot

| Baumarkt | Produkt |
|----------|-----------|
| Hornbach | Schrauben |
| Hornbach | Werkzeug |
| OBI | Werkzeug |
| OBI | Möbel |
| TOOM | Lampen |
| TOOM | Blumen |
| Bauhaus | Schrauben |

Es ist folgendes bekannt: Die Geschäftsleitung von OBI will Marktanteile von den anderen Baumärkten erobern und sich dafür als Komplettversorger im Bereich Baumarkt positionieren.

Übung 3 (Aus einer Klausur)

1. Legen Sie die Tabellen an und füllen Sie diese exemplarisch.
2. OBI will deshalb alle Produkte anbieten, für die Kunden bevorzugt in irgendeinen der Baumärkte gehen. Sobald eine Kundenumfrage ergibt, dass ein Kunde ein Produkt bevorzugt in einem Baumarkt kauft und dieses in der Datenbank gespeichert wird, dann soll dieses Produkt auch entsprechend bei OBI angeboten werden. Dieses soll von der Datenbank automatisch auch bei zukünftigen Datensätzen berücksichtigt werden.
3. Als zweite Maßnahme will die Geschäftsleitung möglichst alle Kunden aggressiv bewerben, die bei der Konkurrenz einkaufen. Dazu wird eine Tabelle Werbung mit den Namen der Kunden aufgebaut, in die alle potentiellen Kunden aufgenommen werden sollen. Wann immer ein Kunde eines anderen Baumarktes in die Tabelle Baumarktkunden hinzukommt, soll dieser in die Werbungstabelle übernommen werden.
4. Schreiben Sie mit Hilfe eines Cursors eine PL/SQL Prozedur, welche als Eingabeparameter den Namen eines Baumarktes erhält. Die Prozedur soll alle Produkte der Kunden ermitteln, die nicht in diesem Baumarkt einkaufen. Zum Schluss soll die Prozedur noch ausgeben, wie viele Produkte ausgegeben wurden.

Übung 4:

Die Neubank speichert in zwei Tabellen Daten über ihre Kunden. In der Kunden Tabelle werden der Name des Kunden sowie sein aktueller Kontosaldo sowie sein Dispolimit gespeichert. In einer Transaktionstabelle werden die Kontenbewegungen in Form von Sendername, Empfängername, Betrag, Datum gespeichert.

1. Realisieren die beiden Tabellen und sorgen Sie dafür, dass die folgenden Geschäftsanforderungen umgesetzt werden.
2. Der Kontosaldo eines Kunden soll aus den Transaktionen wie folgt berechnet werden. Überall wo der Kunde als Empfänger auftaucht wird der Betrag bei ihm gutgeschrieben und dort wo er als Absender auftritt wird der Betrag bei ihm reduziert. Sofern der Sender oder Empfänger nicht in den Stammdaten vorkommt wird diese Seite als extern angenommen und nicht weiter verrechnet. Entwickeln Sie als Teil der Lösung auch eine Überweisungsprozedur, die es erlaubt einen Betrag zwischen zwei Kunden auszutauschen. Die Parameter sind Name1, Name2 und Betrag.

Übung 4

3. Bei Überweisungen soll zusätzlich noch das Dispolimit geprüft werden. Stellen Sie sicher, dass Überweisungen zurückgewiesen werden, wenn dadurch der Kunde seinen Dispo überschreiten würde.
4. Die Neubank möchte Kunden durch eine attraktive Verzinsung gewinnen. Sie zahlt deshalb jährlich einen Zins von 1,5 Prozent auf das Saldo. Sofern der Kunde im Minus ist fallen Dispozinsen in Höhe von 3 Prozent an. Schreiben Sie eine Prozedur, die zur Zinsberechnung und Aktualisierung der Salden verwendet werden kann.
5. Die Neubank möchte aufgrund interner Regelungen Überweisungen nur während der Geschäftszeiten der Bank Mo-Fr 9-17 Uhr erlauben. Entwickeln Sie einen Mechanismus der Überweisungen zurückweist wenn sie nicht in diesem Zeitraum erstellt werden. Überweisungen müssen nicht nur durch die Überweisungsfunktion ausgelöst werden, es kann auch in der Transaktionstabelle zu Änderungsoperationen kommen.

PL/SQL-Tabellen

- **PL/SQL-Tabellen** sollten nicht mit Datenbanktabellen verwechselt werden
- Sie existieren **nur im Hauptspeicher**.
- Eine PL/SQL-Tabelle besteht aus zwei Komponenten, die Spalten entsprechen, aber nicht benannt werden.
 - Die **erste Spalte** hat den Datentyp **BINARY_INTEGER** und dient der **Identifizierung** der einzelnen Elemente in der Tabelle.
 - Die **zweite Spalte** dient der Aufnahme von **Werten** und stellt die eigentliche Variable dar. Hier kann auch ein Record verwendet werden.
- Eine PL/SQL Table kann **dynamisch** wachsen (keine festgelegte Größe).

- Typdefinition und –deklaration im Beispiel:

```
TYPE num_table_type IS TABLE OF NUMBER
INDEX BY BINARY_INTEGER;
empno_table num_table_type;
```

```
TYPE emp_rec_table_type IS TABLE OF emp%ROWTYPE
INDEX BY BINARY_INTEGER;
emp_table emp_rec_table_type;
```

Auslesen von Tabelleneinträgen

- Die einzelnen Elemente einer PL/SQL-Tabelle werden über ihren Index angesprochen in der Form

`variablenname(i) ,`

wobei i dem Wert in der Primärschlüsselspalte entspricht. i muss nicht bei 1 beginnen und kann auch negativ sein. Lücken sind zulässig. Daher ist ein wahlfreier Zugriff möglich.

Bei einem Record-Datentyp werden die Felder wieder mit der Punktnotation nach Angabe des Index angesprochen.

- Beispiel:

```
v_empno := emp_table(3);  
IF emp_table(2).empno = 4711 THEN ...  
emp_table(4).ename := 'Scotty';
```

Methoden für PL/SQL-Tabellen

- Form:

`tablename.methodenname [(Parameter)]`

- Verfügbare Methoden:

EXISTS(*n*): Gibt TRUE zurück, wenn das Element mit Index *n* existiert, sonst FALSE.

COUNT: Gibt die Anzahl der Elemente einer Table zurück.

FIRST / LAST: Gibt den niedrigsten bzw. höchsten Index einer Table zurück oder NULL, wenn die Table leer ist.

PRIOR(*n*) / NEXT(*n*): Gibt den Index zurück, der vor (PRIOR) bzw. nach (NEXT) *n* in der Table liegt.

DELETE: Löscht alle Elemente der Tabelle.

DELETE(*n*): Löscht das Element mit Index *n*.

DELETE(*m*, *n*): Löscht alle Elemente mit Index zwischen *m* und *n*.

Einlesen einer ganzen Tabelle mit Bulk Collect

```
DECLARE
    TYPE nr_liste IS TABLE OF emp.empno%TYPE;
    nr nr_liste;
    num number;

BEGIN
    SELECT empno BULK COLLECT INTO nr FROM emp WHERE deptno = 20;
    FORALL i IN nr.FIRST .. nr.LAST
        loop
            UPDATE emp SET sal = sal* 1.1 WHERE empno = nr(i);
        end loop;

END;
/
```

Per FORALL werden alle UPDATE Statements auf einmal übertragen und nicht einzeln! -> Performancegewinn

FORALL kann immer nur ein einziges DML Statement enthalten (!)

Zusammenfassung (1/2)

- Eigenschaften und Struktur
 - DECLARE, BEGIN, EXCEPTION, END
- Typen, Variablen, Kontrollstrukturen und Operatoren
 - Anonymer Block, Prozedur, Funktion
 - Variablen: Typen, Deklaration und Verwendung
 - %TYPE, RECORD, %ROWTYPE
 - Kontrollstrukturen: IF, LOOP, WHILE
 - Operatoren
 - Kommentare
- Prozeduren und Funktionen
 - Erstellung und Verwendung
- Ausnahmebehandlung (EXCEPTION)
 - Definieren und auslösen
 - Behandeln

Zusammenfassung (2/2)

- Trigger
 - Definierte Ereignisse bei INSERT, UPDATE, DELETE
 - Definition und Nutzung
 - FOR EACH ROW, :NEW :OLD
 - INSTEAD OF
- Cursor
 - Definition und Nutzung; CURSOR, OPEN, FETCH, CLOSE