# Kapitel 5

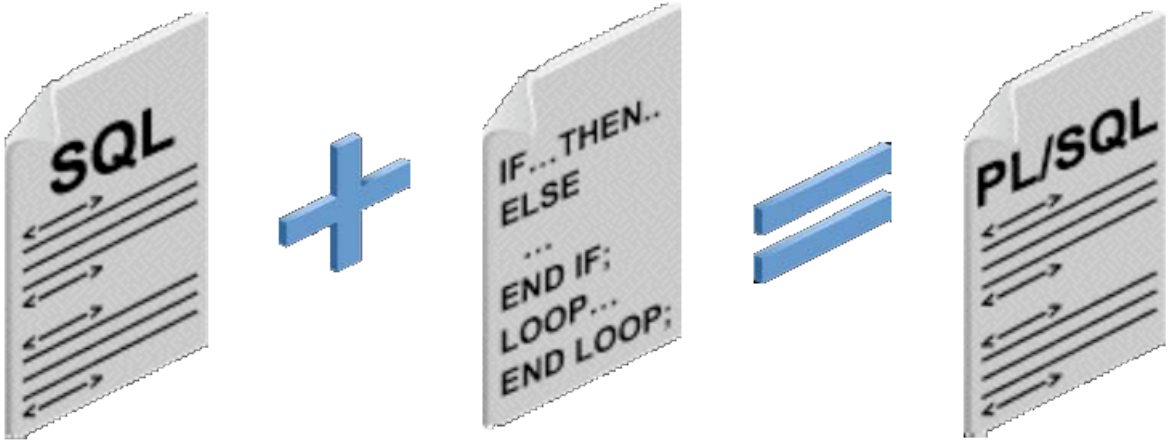
# Datenbankprogrammierung

# 5.1 Einführung in PL/SQL

## Was ist PL/SQL?

PL/SQL:

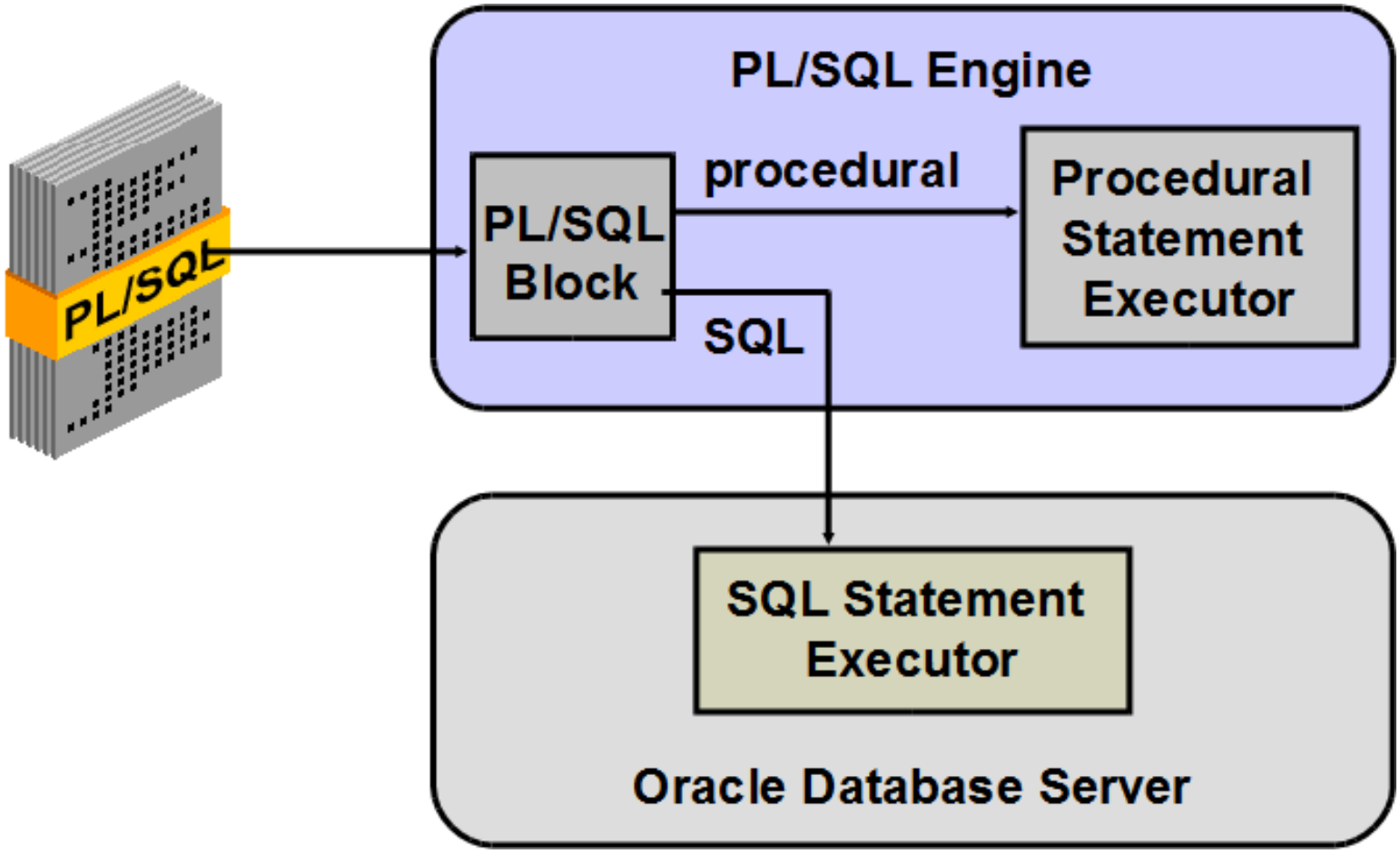
- steht für “Procedural Language”-Erweiterung von SQL
- ist die Standard-Datenbankprogrammiersprache von Oracle
- integriert SQL in prozedurale Konstrukte!



## PL/SQL

- bietet eine Block-Struktur für ausführbare Code-Einheiten
- bietet prozedurale Konstrukte an:
  - Variablen, Konstanten und Datentyp-Funktionen
  - Kontrollstrukturen wie bedingte Anweisungen und Schleifen
  - wiederverwendbare Programmeinheiten wie Datenbankprozeduren und -funktionen
- erlaut es, SQL-DML-Anweisungen inklusive Ein-Zeilen-SQL-Anfragen auszuführen
- bietet mit “Cursorn” Zugriff auf die Ergebnisse von Mehr-Zeilen-SQL-Anfragen

# PL/SQL-Umgebung



## PL/SQL-Block-Struktur

**declare** (optional)

Variablen, Cursor,
benutzer-definierte Ausnahmen

**begin**

- SQL-Anweisungen
- PL/SQL-Anweisungen

**exception** (optional)

Befehle, die ausgeführt werden,
wenn Fehler bzw. Ausnahmen
auftreten

**end**;

## PL/SQL-Block-Typen

### Anonym

```
[declare
 --Deklarationen]
begin
 --Anweisungen

[exception
 --Ausn.behandlung]
end;
```

### Prozedur

```
procedure Name

is
[--Deklarationen]
begin
 --Anweisungen

[exception
 --Ausn.behandlung]
end;
```

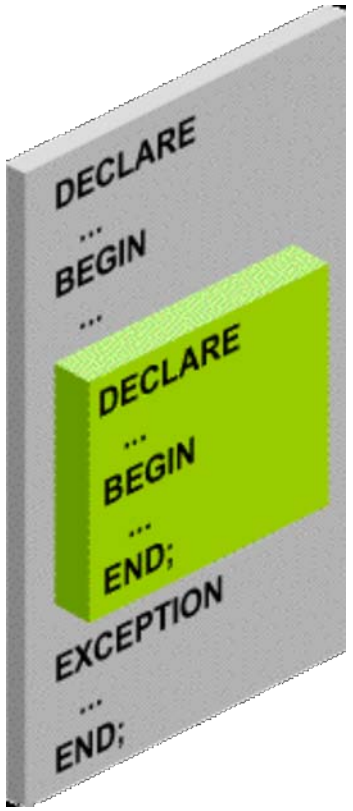
### Funktion

```
function Name
return Datentyp
is
[--Deklarationen]
begin
 --Anweisungen
return Rückgabewert;
[exception
 --Ausn.behandlung]
end;
```

## Verschachtelte PL/SQL-Blöcke

PL/SQL-Blöcke können verschachtelt sein.

- Ein ausführbarer Abschnitt (**begin** ... **end**) kann Blöcke (als Anweisungen) enthalten.
- Ein **exception**-Abschnitt kann verschachtelte Blöcke enthalten.
- Deklarationen gelten für den Block der Deklaration und alle darin enthaltenen Blöcke, soweit sie nicht durch Deklarationen gleichnamiger Bezeichner überschrieben werden.
- Verwendete Bezeichner beziehen sich auf die die innerste an der Verwendungsstelle gültige Deklaration des Bezeichners[1].

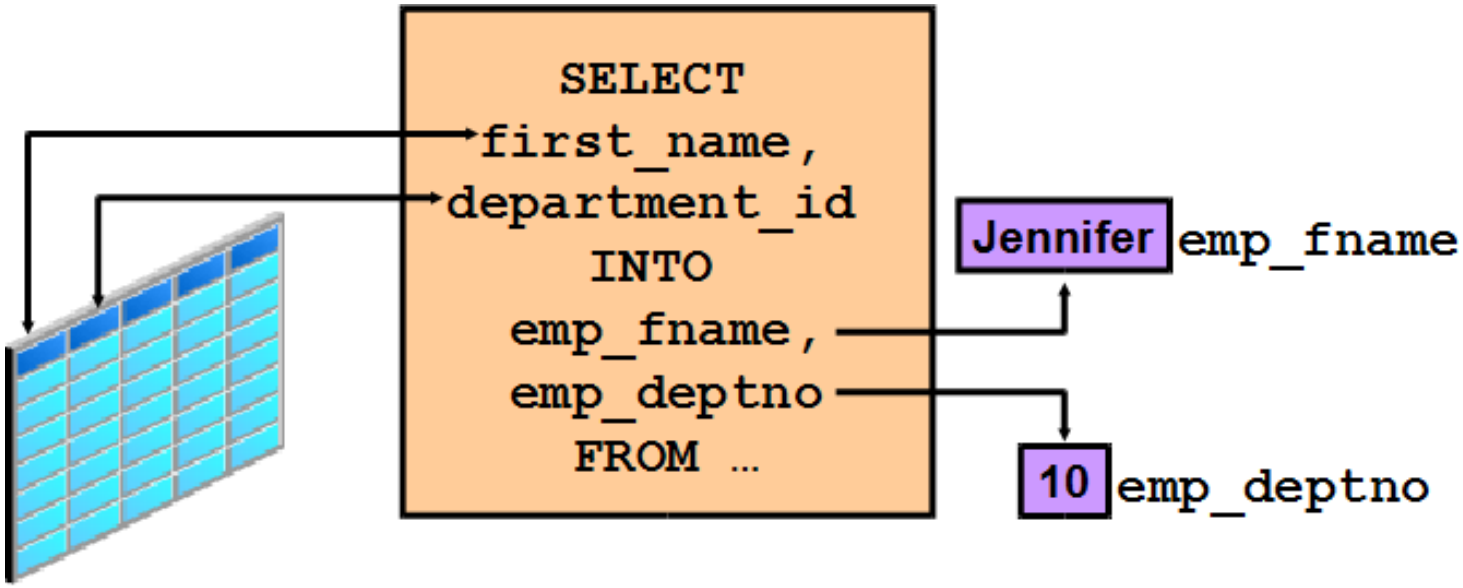



[1] außer man benennt Blöcke , z.B. «outer» **declare**... **begin**..., und versieht den Bezeichner mit dem Blocknamen als Präfix, z.B. outer.name

## Variablen

Variablen können benutzt werden für:

- temporäre Datenspeicherung
- Manipulation von gespeicherten Werten
- programminterne Wiederverwendung von Daten



## Deklarieren und Initialisieren von Variablen

Syntax:

*Bezeichner* [**constant**] *Datentyp* [**not null**]
[{ := | **default**} *Ausdruck*];

Beispiele für (skalare) Variablen:

```
declare
  emp_hiredate   date;
  emp_deptno     number(2) not null := 10;
  location       varchar2(13) default 'Atlanta';
  c_tax_rate     constant number(3,2) := 8.25;
  count_loop     binary_integer := 0;
  orderdate      date := sysdate+7;
  valid          boolean not null := TRUE;
  ...
```

## Deklarieren und Initialisieren von Variablen (Forts.)

PL/SQL-Bezeichner:

- sind nicht case-sensitiv
- müssen mit einem Buchstaben beginnen
- können Buchstaben und Zahlen enthalten
- können Sonderzeichen wie Unterstriche, '$' und '#' enthalten
- sind beschränkt auf 30 Zeichen Länge
- dürfen keine reservierten Wörter sein

(Skalare) Basisdatentypen:

- **char**[(*Maximallänge*)]
- **varchar2(***Maximallänge***)**
- **long**
- **long raw**
- **number** [(*Präzision*, *Skala*)]
- **binary_integer**
- **pls_integer**
- **binary_float**
- **binary_double**
- **date**
- **timestamp**
- **timestamp with time zone**
- **timestamp with local time zone**
- **interval day to second**
- **boolean**

## Deklarieren von Variablen mit Hilfe von %type

Syntax:

*Bezeichner* {*Tabelle.Spalten_Name*|*Variable*}**%type**;

Beispiele:

```
emp_lname    EMPLOYEES.last_name%type;
balance      number(7,2);
min_balance  balance%type := 1000;
```

Das Schlüsselwort **%type** wird verwendet, um eine Variable mit dem gleichen Datentyp zu deklarieren

- wie eine Spalte einer Tabelle in der Datenbank
- oder wie eine andere deklarierte Variable.

## Operatoren in PL/SQL

- genau wie in SQL:
  - logische
  - arithmetische
  - Konkatenation
  - ggf. mit Klammerung
- anders als in SQL:
  - Potenzierungs-Operator ($**$)

Beispiele:

- Erhöhe den Zähler für eine Beobachtung:

```
obs_count := obs_count+1;
```

- Setze den Wert eines booleschen Flags:

```
good_sal := sal between 50000 and 150000;
valid := (empno is not null);
```

## SQL-Funktionen in PL/SQL

- in prozeduralen Anweisungen verfügbar:
  - Zeilen-Funktionen auf Zahlen
  - Zeilen-Funktionen auf Strings
  - Zeilen-Funktionen auf Kalenderdaten/Zeitstempeln
  - Datentyp-Konvertierungsfunktionen, z.B. **to_date**
  - **greatest** und **least**
  - einige andere Funktionen
- in prozeduralen Anweisungen nicht verfügbar:
  - **decode**
  - Aggregierungsfunktionen

## SQL-Funktionen in PL/SQL (Forts.)

Beispiele:

- Lies die Länge einer Zeichenkette aus:

```
desc_size integer(5);
prod_description varchar2(70):='You can use this product with your
computers for mobile internet access.';
...
--  merke die Länge von  prod_description
desc_size:= length(prod_description);
```

- Konvertiere eine Zeichenkette in Kleinbuchstaben:

```
emp_name EMPLOYEES.last_name%type;
...
emp_name := lower(emp_name);
```

## Hinweise zur Syntax

- String- und Datumswerte sind von Apostrophs zu umschließen.
- Anweisungen können über mehrere Zeilen fortgesetzt werden.
- Einzeilige Kommentare werden durch zwei vorangestellte Bindestriche (--) gekennzeichnet.
- Mehrzeilige Kommentare werden mit "/*" und "*/" umschlossen.

Beispiel (übertriebene Kommentierung):

```
declare
   monthly_sal number(8,2);
   annual_sal   number(9,2);
   ...
begin  -- Anfang des ausführbaren Abschnitts
/* Berechne das jährliche Gehalt basierend auf dem
   vom Nutzer eingegebenen monatlichen Einkommen */
   annual_sal := monthly_sal*12;
end;  -- Das ist das Ende des Blocks
```

## SQL-Anweisungen in PL/SQL

- Ein-Zeilen-Anfragen: **select**
- DML: **insert**, **update**, **delete**
- Transaktionssteuerung: **commit**, **rollback**, **savepoint**

## select-Anweisungen in PL/SQL

Syntax:

**select** *Select-Liste*
**into** {*passende Liste von Variablennamen*}
| {*Name einer passenden Record-Variable*}
**from** ...

- Die **into**-Klausel wird benötigt.
- Anfragen dürfen nur eine Zeile zurückgeben.

Beispiel:

```
SET SERVEROUTPUT ON                                  [Client-abhängiger Befehl]
declare
   fname varchar2(25);
begin
   select first_name into fname
   from EMPLOYEES where employee_id = 200;
   DBMS_OUTPUT.PUT_LINE('First Name is : ' || fname);
end;
/                                        [Client-abhängige Ende-Kennzeichnung]
```

## select-Anweisungen in PL/SQL (Forts.)

Weitere Beispiele:

```
declare
   emp_hiredate   EMPLOYEES.hire_date%type;
   emp_salary     EMPLOYEES.salary%type;
begin
   select  hire_date, salary
   into    emp_hiredate, emp_salary
   from    EMPLOYEES
   where employee_id = 100;
   ...
end;
```

```
declare
   sum_sal  number(10,2);
   deptno   number not null := 60;
begin
   select  sum(salary)
   into    sum_sal
   from    EMPLOYEES
   where department_id = deptno;
   DBMS_OUTPUT.PUT_LINE('The sum of salaries is: '|| sum_sal);
end;
```

## Misslungene Namensgebung

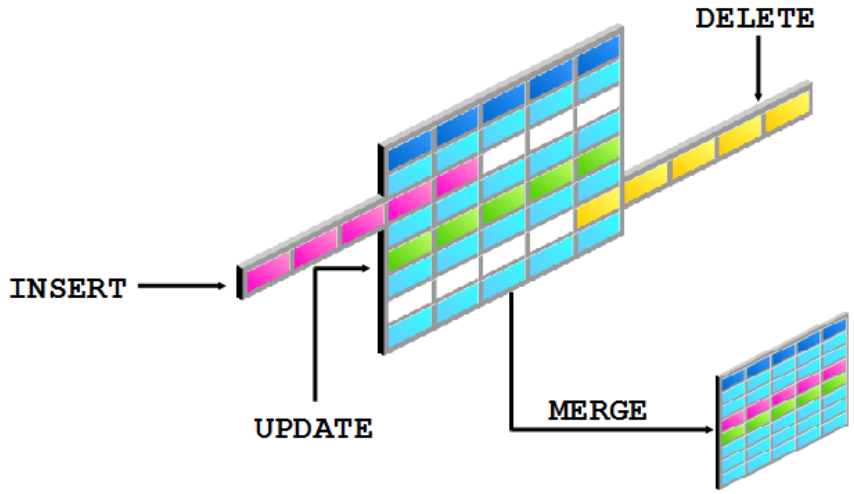
```
declare
   hire_date      EMPLOYEES.hire_date%type;
   sysdate        hire_date%type;
   employee_id    EMPLOYEES.employee_id%type := 176;
begin
   select  hire_date, sysdate
   into    hire_date, sysdate
   from    EMPLOYEES
   where employee_id = employee_id;
end;
```

```
→  declare
   *
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 6
```

## Namenskonventionen

- Man sollte Namenskonventionen verwenden, um Mehrdeutigkeiten in der **where**-Klausel zu vermeiden, z.B. empid statt employee_id.
- Insbesondere sollte vermieden werden, Spaltennamen der Datenbank als PL/SQL-Bezeichner zu verwenden.
- SQL-Syntax-Überprüfungen in PL/SQL-Programmen folgen bestimmten Vorrangregeln:
  - Namen von *Spalten* der Datenbanktabellen haben Vorrang vor Namen lokaler Variablen.
  - Namen lokaler Variablen und formaler Parameter haben Vorrang vor Namen von Datenbank-*Tabellen*.

# DML-Anweisungen in PL/SQL




Beispiel:

```
declare
   sal_increase   EMPLOYEES.salary%type := 800;
begin
   update EMPLOYEES
   set       salary = salary + sal_increase
   where  job_id = 'ST_CLERK';
end;
```

## Variablen/Datentypen (fortgesetzt): PL/SQL-Records

- Ein PL/SQL-**Record** besteht aus einer oder mehreren Komponenten von skalaren oder Record-Datentypen, sogenannten “Feldern”.
- Ein Record ist also eine heterogen zusammengesetzte Datenstruktur wie es sie in den meisten 3GL-Sprachen einschl. C und C++ gibt.
- Ein PL/SQL-Record ist vor allem geeignet, um eine ganze Daten-Zeile aus einem Anfrageergebnis bzw. für eine Tabelle aufzunehmen.

Beispiel: Deklaration eines Record-Datentypen und einer Record-Variablen, um Name, Beruf und Gehalt eines Angestellten zu speichern:

```
declare
  type emp_record_type is
  record
    (last_name   varchar2(25),
     job_id      varchar2(10),
     salary      number(8,2));
  emp_record  emp_record_type;
  ...
```

```
begin
  ...
  select  last_name, job_id, salary
  into    emp_record
  from    EMPLOYEES
  where employee_id = 621;
  ...
```

## Deklarieren von PL/SQL-Record-Datentypen und -Record-Variablen

Syntax zur Deklarationen eines Recordtyps:

**type** *Recordtyp_Name* **is**
**record** (*Feld_Beschreibung* [, *Feld_Beschreibung* ] ...);

*Feld_Beschreibung* :: wie Variablendeklaration

Syntax zur Deklaration einer Record-Variable:

*Bezeichner* { *Recordtyp_Name* | *Tabelle* **%rowtype** };

Mit dem Schlüsselwort **%rowtype** wird die Spaltenstruktur, d.h. Namen und Datentypen aller Spalten, der angegebenen Datenbank-Tabelle oder -Sicht als Recordtyp übernommen.

## Verwenden eines %rowtype-Records

Beispiel: Ein Angestellter geht in den Ruhestand:

```
declare
   emp_rec  EMPLOYEES%rowtype;
begin
   select  * into emp_rec
   from   EMPLOYEES
   where employee_id = 124;
   --
   insert  into RETIRED_EMPS(empno, ename, job, mgr,
              hiredate, leavedate, sal, deptno)
   values (emp_rec.employee_id, emp_rec.last_name,
              emp_rec.job_id, emp_rec.manager_id,
              emp_rec.hire_date, sysdate,
              emp_rec.salary, emp_rec.department_id);
end;
```

## Verwenden eines %rowtype-Records (Forts.)

Das gleiche Beispiel anders formuliert:

```
declare
   emp_rec  RETIRED_EMPS%rowtype;
begin
   select  employee_id, last_name,
           job_id, manager_id,
           hire_date, sysdate,
           salary, department_id
   into    emp_rec
   from    EMPLOYEES
   where employee_id =124;
   --
   insert  into RETIRED_EMPS values emp_rec;
end;
```

## Verwenden eines %rowtype-Records (Forts.)

Und ein Update auf der eingefügten Zeile (beachte "**row**=" in **set**-Klausel):

```
declare
  emp_rec  RETIRED_EMPS%rowtype;
begin
  select  * into emp_rec
  from   RETIRED_EMPS
  where  empno =124;
  --
  emp_rec.leavedate :=
          trunc(add_months(sysdate,1), 'MONTH')-1;
  emp_rec.salary:= emp_rec.salary*1.025;
  --
  update RETIRED_EMPS set row = emp_rec
  where  empno = 124;
end;
```

## Attribute von SQL-Anweisungen
## (PL/SQL-Terminologie: “Implizite Cursor-Attribute”)

Durch die Verwendung spezieller, impliziter Attribute kann man die Ergebnisse von SQL-Anweisungen überprüfen.

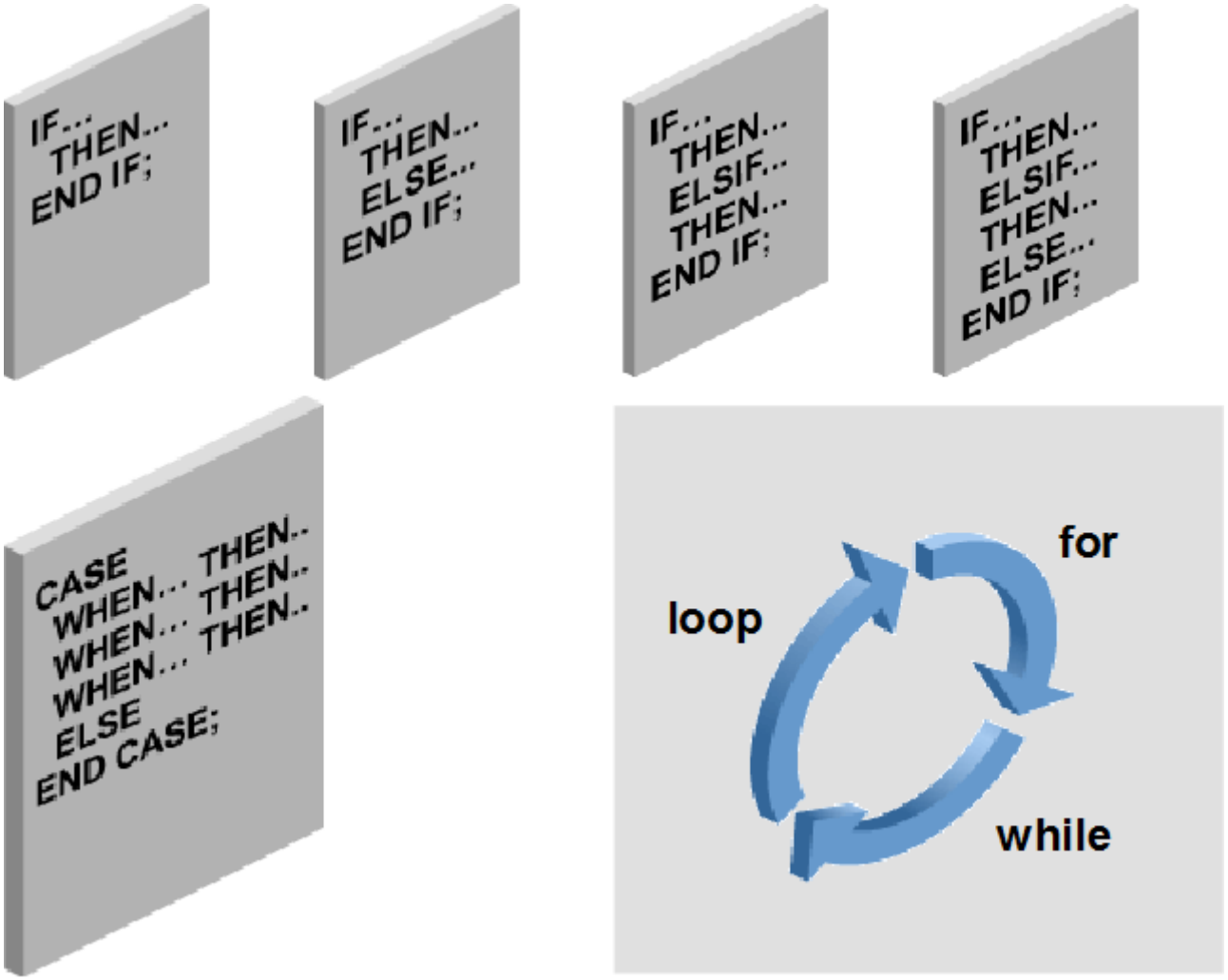
| | |
|---|---|
| SQL%FOUND | Boolesches Attribut, das TRUE liefert, wenn die letzte SQL-Anweisung mindestens eine Zeile zurückgegeben oder betroffen hat. |
| SQL%NOTFOUND | Boolesches Attribut, das TRUE liefert, wenn die letzte SQL-Anweisung keine Zeile zurückgegeben oder betroffen hat. |
| SQL%ROWCOUNT | Ein Integer-Wert, der die Anzahl der Zeilen angibt, die von der letzten SQL-Anweisung betroffen waren. |

## Attribute von SQL-Anweisungen (Forts.)

Beispiel: Lösche aus der Tabelle DEPARTMENTS alle Zeilen, die eine bestimmte location_id aufweisen, und gib die Anzahl der gelöschten Zeilen aus.

```
declare
   locid DEPARTMENTS.location_id%type := 176;
begin
   delete from DEPARTMENTS
   where location_id = locid;
   DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' row(s) deleted.');
end;
```

# Kontrollstrukturen in PL/SQL



## if-Anweisungen: Syntax

**if** *Bedingung* **then**
    *Anweisungen*;
[**elsif** *Bedingung* **then**
    *Anweisungen*;]
[**else**
    *Anweisungen*;]
**end if**;

## case-Anweisungen: Syntax

**case** *Selektor*
    **when** *Ausdruck1* **then** *Anweisungen1*
    **when** *Ausdruck2* **then** *Anweisungen2*
    ...
    **when** *AusdruckN* **then** *AnweisungenN*
    [**else** *AnweisungenN+1*]
**end case**;

(auch verfügbar: **case**-Ausdrücke)

# if-Anweisungen: Beispiel

```
declare
   myage number := 22;
begin
   if myage < 11 then
        DBMS_OUTPUT.PUT_LINE(' I am a child');
   elsif myage < 20 then
        DBMS_OUTPUT.PUT_LINE(' I am young');
   elsif myage < 30 then
        DBMS_OUTPUT.PUT_LINE(' I am in my twenties');
   elsif myage < 40 then
        DBMS_OUTPUT.PUT_LINE(' I am in my thirties');
   else
        DBMS_OUTPUT.PUT_LINE( 'I am always young');
   end if;
end;
```

## case-Anweisungen: Beispiel

```
declare
   deptid number := 100;
   avg_sal number(10,2);
begin
   select avg(salary) into avg_sal from EMPLOYEES
   where department_id = deptid;
   case deptid
   when 100 then
      update EMPLOYEES set salary = avg_sal
      where salary < avg_sal and department_id = 100;
      DBMS_OUTPUT.PUT_LINE('Congratulations!');
   when 200 then
      update EMPLOYEES set salary = avg_sal
      where salary > avg_sal and department_id = 200;
      DBMS_OUTPUT.PUT_LINE('So sorry!');
   else
      DBMS_OUTPUT.PUT_LINE('No changes!');
   end case;
end;
```

## loop-Schleifen: Syntax

**loop**
    *Anweisung1*;
    *Anweisung2*;
    ...
    **exit** [**when** *Bedingung*];
**end loop**;

**loop** wiederholt eine Folge von Anweisungen beliebig oft,
ausser die Schleife wird mit **exit** verlassen.

## loop-Schleifen: Beispiel

```
declare
   countryid   location.country_id%type := 'CA';
   locid       locations.location_id%type;
   counter     number(2) := 1;
   newcity     location.city%type := 'Montreal';
begin
   select max(location_id) into locid from locations
   where country_id = countryid;
   loop
      insert into locations(location_id, city, country_id)
      values ((locid + counter), newcity, countryid);
      counter := counter+1;
      exit when counter > 3;
   end loop;
end;
```

## while-Schleifen: Syntax

**while** *Bedingung* **loop**
    *Anweisung1*;
    *Anweisung2*;
    ...
**end loop**;

Die **while**-Schleife wiederholt Anweisungen, solange die *Bedingung* TRUE ist.

## while-Schleifen: Beispiel

```
declare
   countryid   location.country_id%type := 'CA';
   locid       locations.location_id%type;
   counter     number(2) := 1;
   newcity     location.city%type := 'Montreal';
begin
   select max(location_id) into locid from locations
   where country_id = countryid;
   while counter <= 3 loop
      insert into locations(location_id, city, country_id)
      values ((locid + counter), newcity, countryid);
      counter := counter + 1;
   end loop;
end;
```

## for-Schleifen: Syntax

**for** *Zähler* **in** [**reverse**] *Untere_Grenze*..*Obere_Grenze* **loop**
    *Anweisung1*;
    *Anweisung2*;
    ...
**end loop**;

- Eine **for**-Schleife wird benutzt, wenn die Anzahl der Iterationen vorher bekannt ist.
- Der Zähler wird eine implizit deklarierte Variable. Er ist außerhalb der Schleife undefiniert.
- Man darf den Zähler nicht als Ziel einer Zuweisung benutzen.

## for-Schleifen: Beispiel

```
declare
   countryid   location.country_id%type := 'CA';
   locid       locations.location_id%type;
   newcity     location.city%type := 'Montreal';
begin
   select max(location_id) into locid from locations
   where country_id = countryid;
   for i in 1..3 loop
      insert into locations(location_id, city, country_id)
      values ((locid+i), newcity, countryid);
   end loop;
end;
```

# Verschachtelte Schleifen und Labels

```
...
begin
   <<Outer_loop>>
   loop
      counter := counter+1;
       exit when counter > 10;
      <<Inner_loop>>
      loop
         ...
          exit Outer_loop when total_done = 'YES';
         --  verlässt beide Schleifen
         exit when inner_done = 'YES';
         --  verlässt nur die innere Schleife
         ...
      end loop Inner_loop;
      ...
   end loop Outer_loop;
end;
```

# Datenbank-Prozeduren und -Funktionen

| Anonymer Block | Unterprogramme<br>(d.h. Prozeduren und Funktionen) |
|---|---|
| unbenannte PL/SQL-Blöcke | benannte PL/SQL-Blöcke |
| jedes Mal kompiliert | nur einmal kompiliert |
| nicht gespeichert | in der Datenbank gespeichert |
| können nicht<br>von anderen Anwendungen<br>aufgerufen werden | sind benannt und können deshalb<br>von anderen Anwendungen<br>aufgerufen werden |
| geben keinen Wert zurück | Funktionen müssen einen Wert<br>zurückgeben |
| können keine Parameter<br>übergeben bekommen | können Parameter<br>übergeben bekommen |

## Prozeduren und Funktionen: Syntax

**create** [**or replace**] **procedure** *Prozedur_Name*
  [(*Parameter1* [*Modus1*] *Datentyp1* [*Initialisierung1*],
  *Parameter2* [*Modus2*] *Datentyp2* [*Initialisierung2*],
  ...)] [2]
**is**|**as**
  *Prozedur_Rumpf*; -- ist ein PL/SQL-Block

**create** [**or replace**] **function** *Funktions_Name*
  [(*Parameter1* [*Modus1*] *Datentyp1* [*Initialisierung1*],
  *Parameter2* [*Modus2*] *Datentyp2* [*Initialisierung2*],
  ...)] [2]
**return** *Datentyp*
**is**|**as**
  *Funktions_Rumpf*; -- ist ein PL/SQL-Block, der mindestens ein
                    -- **return**-Statement enthält

[2] Falls das **execute**-Recht an der Prozedur/Funktion weitergegeben werden soll, wird hier noch eine **authid**-Klausel benötigt:
Die Ausführung erfolgt dann mit den Rechten des Definierenden zur Laufzeit (**authid definer**, voreingestellt; jedoch: ohne die Rechte seiner Rollen!) oder mit den Rechten des Aufrufenden (**authid current_user**, sicherer).
Deshalb in den Übungen, vor allem zu Dynamic SQL (Kap. 5.3), hier hinzufügen: **authid current_user**

## Prozeduren: Beispiel

```
...
create table DEPT as select * from DEPARTMENTS;
create procedure add_dept
    (dept_id DEPT.department_id%type,
     dept_name DEPT.department_name%type)
is
begin
    insert into DEPT(department_id, department_name)
    values(dept_id, dept_name);
    DBMS_OUTPUT.PUT_LINE('Inserted '|| SQL%ROWCOUNT ||' row(s)');
end;
```

## Aufruf einer Prozedur: Beispiel

```
begin
   add_dept (280, 'ST-Curriculum');
end;

select department_id, department_name
from DEPT where department_id = 280;
```

```
Inserted 1 row
PL/SQL procedure successfully completed.
```

| DEPARTMENT_ID | DEPARTMENT_NAME |
|---|---|
| 280 | ST-Curriculum |

## Löschen einer Prozedur/Funktion

```
drop procedure add_dept;   -- oder drop function ...
```

## Funktionen: Beispiel

```
create or replace function check_sal      -- Gehalt überdurchschnittlich?
   (empno EMPLOYEES.employee_id%type)
return boolean is
   dept_id  EMPLOYEES.department_id%type;
   sal      EMPLOYEES.salary%type;
   avg_sal  EMPLOYEES.salary%type;
begin
   select  salary, department_id into sal, dept_id from EMPLOYEES
   where employee_id = empno;
   select  avg(salary) into avg_sal from EMPLOYEES
   where department_id = dept_id;
   return (sal>avg_sal);
exception ...
   ...
```

## Aufrufe einer Funktion: Beispiele

```
begin
   for k in 50..200 loop
      DBMS_OUTPUT.PUT_LINE('Checking employee with id '||k||': ');
      if (check_sal(k) is null) then
         DBMS_OUTPUT.PUT_LINE('null due to exception');
      elsif (check_sal(k)) then
         DBMS_OUTPUT.PUT_LINE('Salary > average');
      else
         DBMS_OUTPUT.PUT_LINE('Salary <= average');
      end if;
   end loop;
end;
```

## Parameter-Übergabe

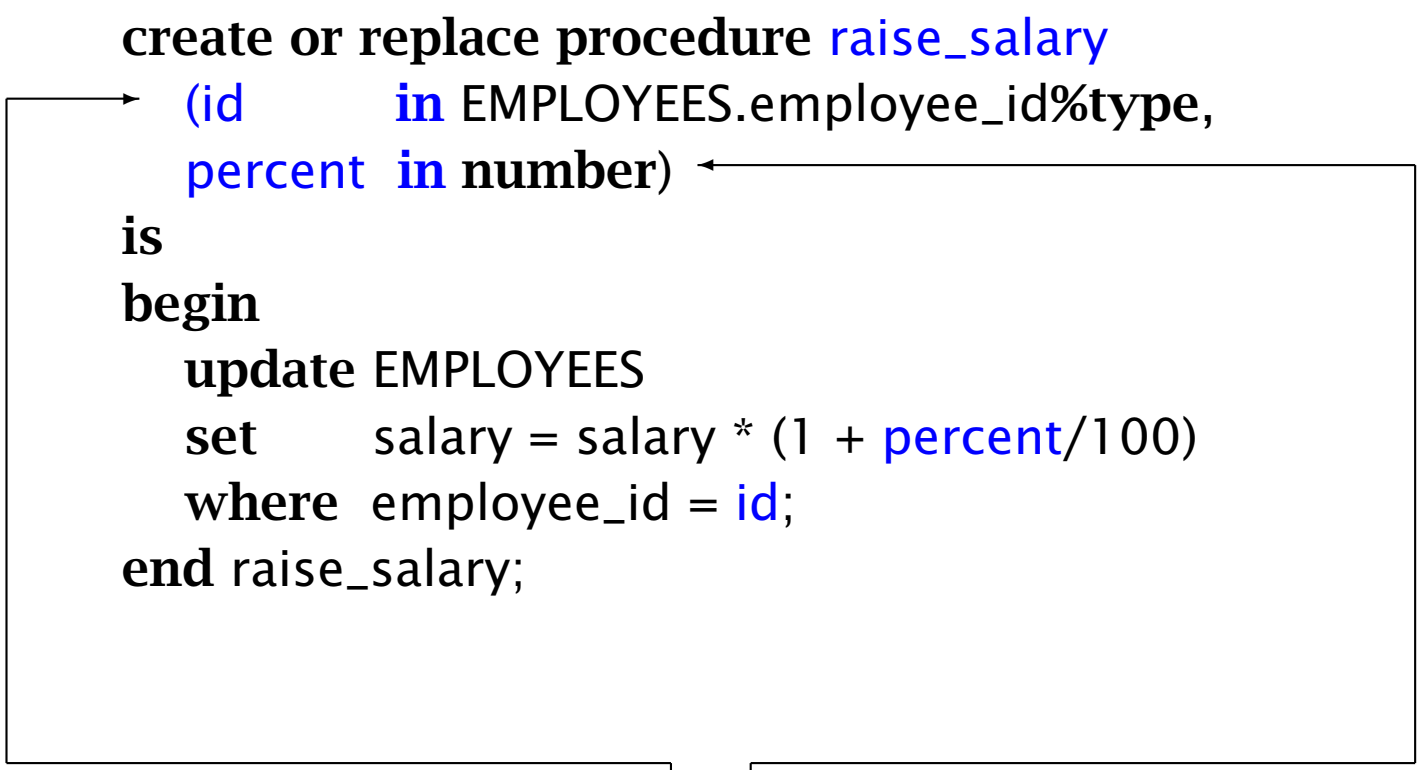
Übergabe-Modi:

- Ein **in**-Parameter (voreingestellt) stellt einem Unterprogramm einen Wert zur Verfügung.
- Ein **out**-Parameter muss beim Aufruf eine Variable sein, in der das Unterprogramm einen Wert zurückliefert.
- Ein **in-out**-Parameter muss ebenfalls eine Variable sein, die beim Aufruf einen Wert zur Verfügung stellt und nach der Rückkehr einen ggf. geänderten Rückgabe-Wert beinhaltet.

Übergabe-Syntax beim Aufruf:

- *Positional:* Alle aktuellen Parameter werden in der gleichen Reihenfolge wie die deklarierten (formalen) Parameter angegeben.
- *Benannt:* Die aktuellen Parameter werden in willkürlicher Reihenfolge angegeben, aber mit dem Zuweisungs-Operator (=>) den formalen Parametern zugeordnet. Zu formalen Parametern mit Initialisierung dürfen aktuelle Parameter weggelassen werden.

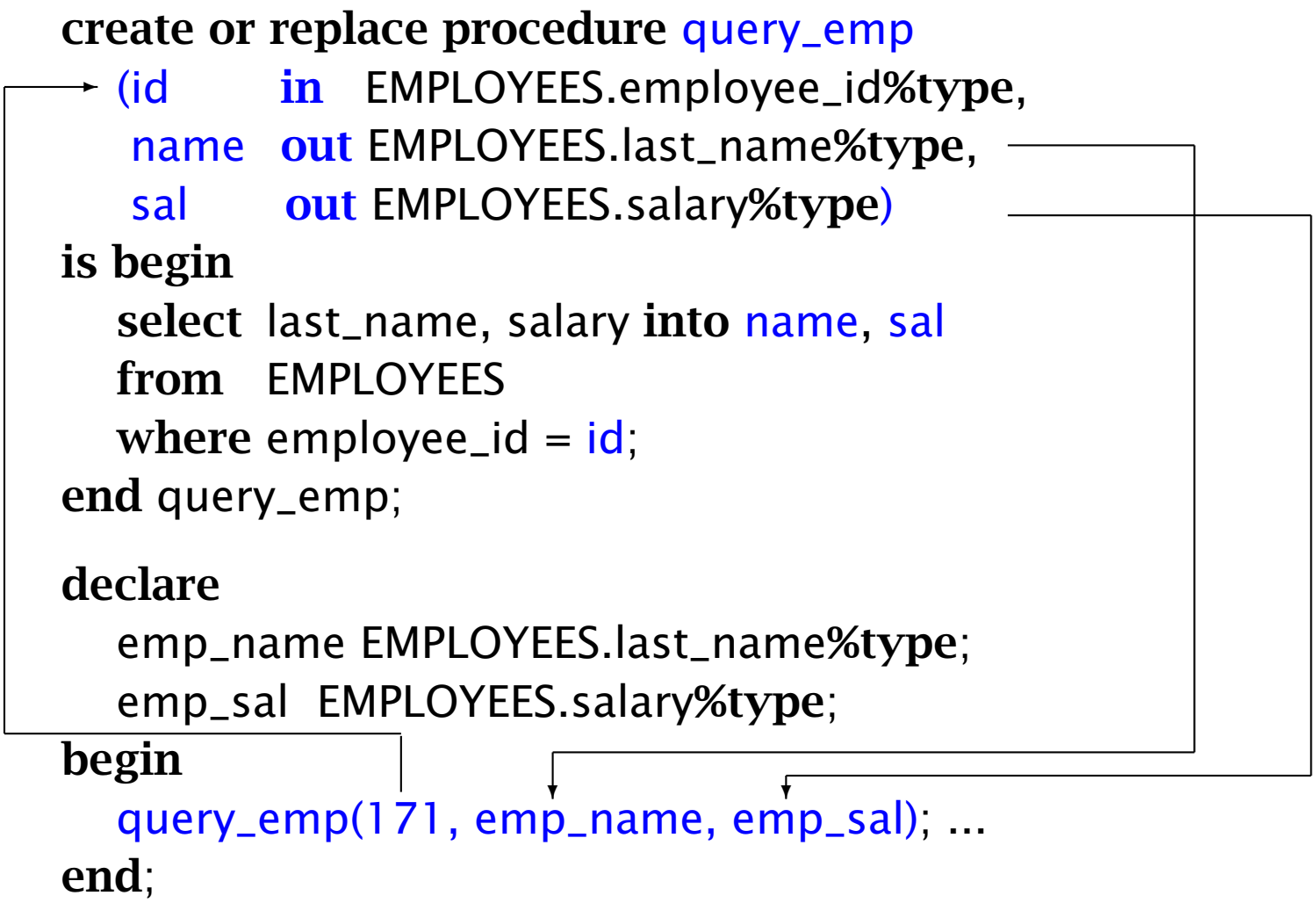
## Parameter-Übergabe: Beispiele



```
create or replace procedure raise_salary
   (id       in EMPLOYEES.employee_id%type,
   percent  in number)
is
begin
   update EMPLOYEES
   set      salary = salary * (1 + percent/100)
   where  employee_id = id;
end raise_salary;
```

```
execute raise_salary(176,10)          [stand-alone-Aufruf vom Client]
```

## Parameter-Übergabe: Beispiele (Forts.)



```
create or replace procedure query_emp
   (id      in  EMPLOYEES.employee_id%type,
    name    out EMPLOYEES.last_name%type,
    sal     out EMPLOYEES.salary%type)
is begin
   select last_name, salary into name, sal
   from   EMPLOYEES
   where employee_id = id;
end query_emp;

declare
   emp_name EMPLOYEES.last_name%type;
   emp_sal  EMPLOYEES.salary%type;
begin
   query_emp(171, emp_name, emp_sal); ...
end;
```

## Parameter-Übergabe: Beispiele (Forts.)

Aufrufende Umgebung


```
create or replace procedure format_phone
  (phone_no in out varchar2)
is
begin
    phone_no := '(' || substr(phone_no,1,3) ||
                ')' || substr(phone_no,4,3) ||
                '-' || substr(phone_no,7);
end format_phone;

begin
    ... tel:='8006330575'; format_phone(tel); ...
end;
```

## Parameter-Übergabe: Beispiele (Forts.)

```
create or replace procedure add_dept
   (name DEPARTMENTS.department_name%type:='Unknown',
    loc     DEPARTMENTS.location_id%type default 1700)
is begin
   insert into DEPARTMENTS(department_id,department_name,location_id)
   values (DEPARTMENTS_SEQ.nextval, name, loc);
end add_dept;                    -- nächste id erzeugt durch ein Sequenz-Objekt
```

- Übergabe durch positionale Notation:

  ```
  execute  add_dept('TRAINING', 2500)
  ```

- Übergabe durch benannte Notation; mögliche Aufrufe:

  ```
  execute  add_dept(loc=>2400, name =>'EDUCATION')
  execute  add_dept(loc=>1200)
  execute  add_dept
  ```

## Verwendung von PL/SQL-Funktionen

- Aufrufe als Teil von PL/SQL-Ausdrücken
  - Zuweisung des Ergebnisses an eine Host-(Client-)Variable

    **variable** salary **number**
    **execute** :salary := get_annsal(100) [3]
  - Zuweisung des Ergebnisses an eine lokale Variable

    **declare** sal EMPLOYEES.salary**%type**;
    **begin**
      sal := get_annsal(100); ...
    **end**;
  - Parameter für ein anderes Unterprogramm

    **execute** DBMS_OUTPUT.PUT_LINE(get_annsal(100))

- Oder (eingeschränkte) Aufrufe in SQL-Anweisungen

  **select** ..., get_annsal(manager_id) **from** DEPARTMENTS;

[3] get_annsal sei eine Funktion, die zu einer employee_id das jährliche Gehalt berechnet

## Benutzer-definierte Funktionen in SQL-Anweisungen

Vorteile:

- PL/SQL-Funktionen können SQL erweitern, wenn Berechnungen zu komplex, zu schwierig oder gar nicht in SQL verfügbar sind;
- sie können die Effizienz erhöhen, wenn sie in der **where**-Klausel verwendet werden, um Daten zu filtern – im Gegensatz zum Filtern in der Anwendung
- und sie können Daten-Werte manipulieren

Aufruf-Stellen wie bei eingebauten Zeilen-Funktionen:

- in der **select**-Liste einer Anfrage
- in Bedingungen der **where**- und **having**-Klauseln
- in **order-by**- und **group-by**-Klauseln
- in der **values**-Klausel einer **insert**-Anweisung
- in der **set**-Klausel einer **update**-Anweisung

## Benutzer-definierte Funktionen in SQL-Anweisungen (Forts.)

Einschränkungen:

- PL/SQL-Funktionen, die aus SQL aufgerufen werden sollen:
  - müssen in der Datenbank gespeichert sein
  - dürfen nur **in**-Parameter mit SQL-Datentypen akzeptieren, keine PL/SQL-spezifischen Typen
  - müssen SQL-Datentypen zurückgeben, keine PL/SQL-spez. Typen
- Beim Aufruf
  - müssen Parameter positional notiert werden
  - muss der Aufrufer auch “Owner” der Funktion sein oder dafür das **execute**-Recht besitzen

## Benutzer-definierte Funktionen in SQL-Anweisungen (Forts.)

Außerdem kontrolliert der Server zur Laufzeit auf Seiteneffekte:

Funktionen, die aufgerufen werden von...

- ...einer **select**-Anweisung, können keine DML-Anweisungen enthalten
- ...einer **update** oder **delete**-Anweisung auf einer Tabelle $T$, können nicht dieselbe Tabelle $T$ anfragen oder manipulieren (s. folg. Bsp.)
- ...SQL-Anweisungen, können Transaktionen nicht beenden (d.h. sie können keine **commit**- oder **rollback**-Anweisung ausführen)

Beachte: Auch Aufrufe von Unterprogrammen, die diese Einschränkungen verletzen, sind in solchen Funktionsausführungen nicht erlaubt.

## Benutzer-definierte Funktionen in SQL-Anweisungen (Forts.)

```
create or replace function strange_fct(sal number)
   return number is
begin
   insert into EMPLOYEES(employee_id, last_name,
            email, hire_date, job_id, salary)
   values(1111,'Frost', 'jfrost@company.com', sysdate, 'SA_MAN', sal);
   return (sal+100);
end;

update EMPLOYEES set salary = strange_fct(2000)
where  employee_id = 170;

→  update EMPLOYEES set salary = strange_fct(2000)
   *
ERROR at line 1:
ORA-04091: table EMPLOYEES is mutating,
trigger/function may not see it
ORA-06512: at "STRANGE_FCT", line 4
```

## Prozeduren/Funktionen im Data Dictionary

Informationen für PL/SQL-Prozeduren/-Funktionen werden in folgenden Sichten des Data Dictionaries gespeichert:

- Die Namen von eigenen Prozeduren/Funktionen stehen in USER_OBJECTS.

  ```
  select  object_name
  from    USER_OBJECTS
  where object_type = 'PROCEDURE'     oder     'FUNCTION';
  ```

- Der Quellcode steht in USER_SOURCE; z. B.:

  ```
  select  text
  from    USER_SOURCE
  where name='ADD_DEPARTMENT' and type='PROCEDURE'
  order   by line;
  ```

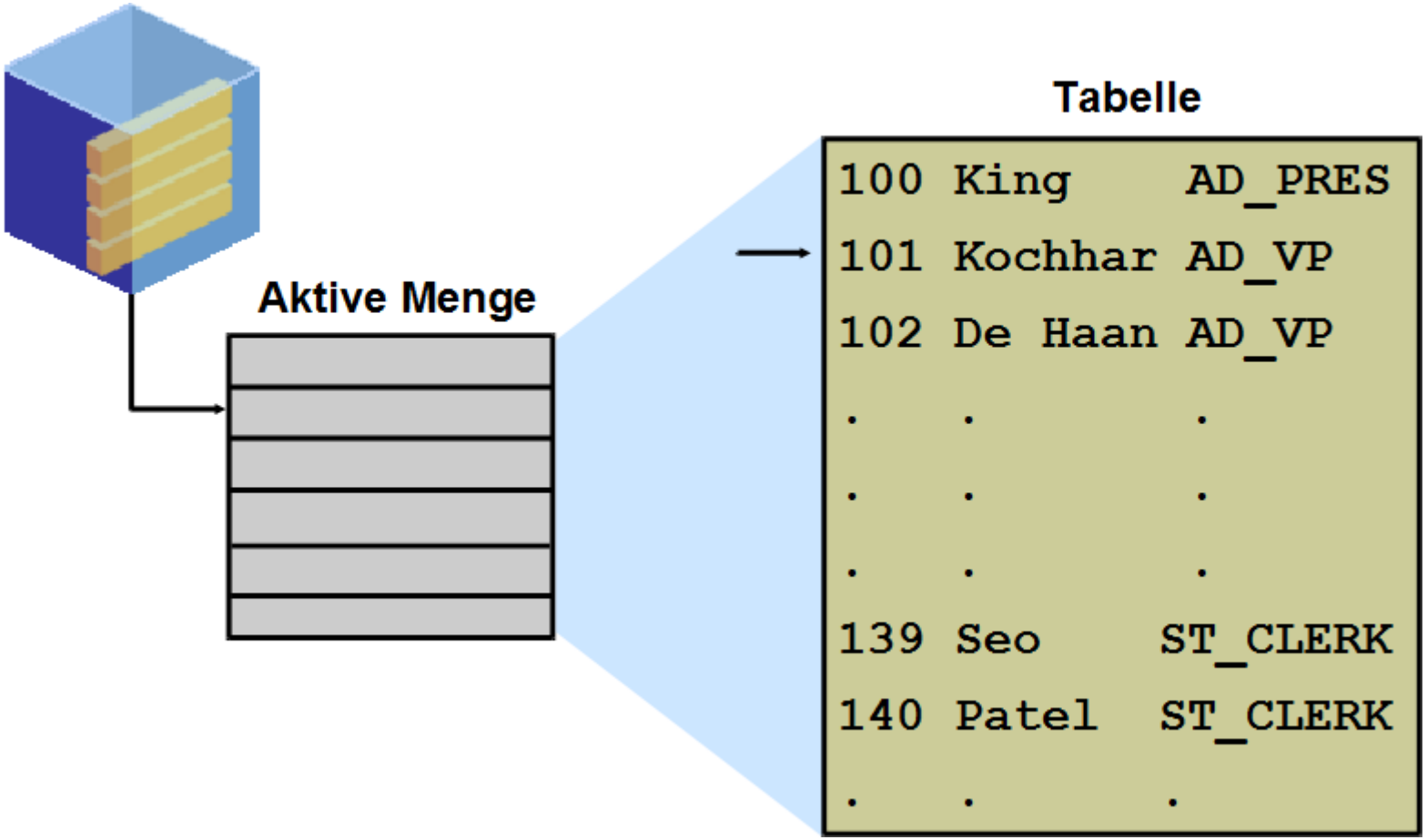
- In den ALL_-Sichten findet man auch noch die Prozeduren/Funktionen, von deren Ownern man das **execute**-Recht bekommen hat.
- USER|ALL_ERRORS zeigt die PL/SQL-Kompilier-Fehler an.
- Eine Rekompilierung ist durch Neueingabe der Definition oder mit **alter** {**procedure**|**function**} *Name* **compile** möglich.

## 5.2 Cursor in PL/SQL

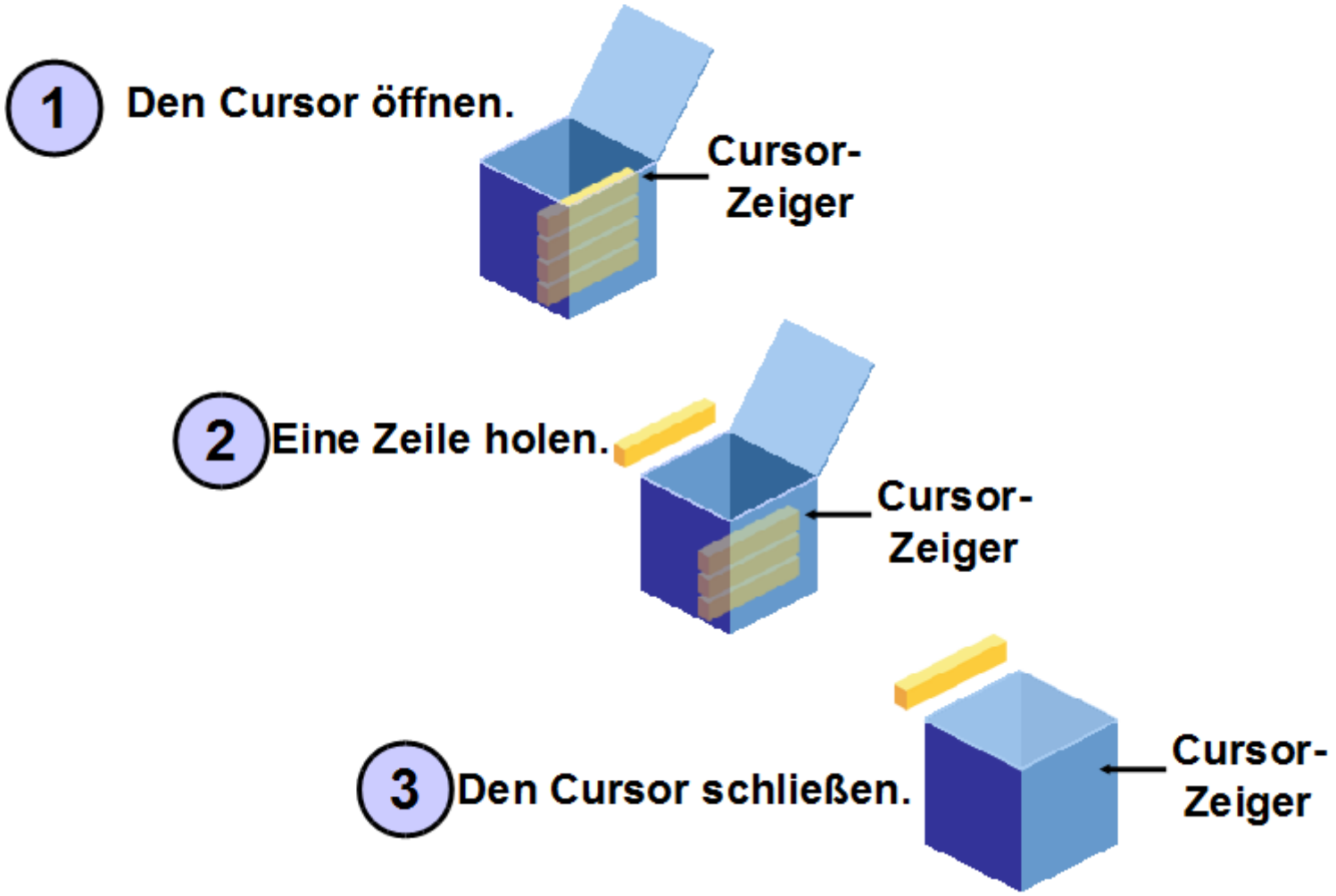
Jede SQL-Anweisung, die vom Oracle-Server ausgeführt wird, hat einen individuellen **Cursor**, der mit ihr verbunden ist und durch den man ihre (sequentielle tupelweise) Ausführung kontrollieren kann:

- Implizite Cursor: werden von PL-SQL für alle DML- und PL/SQL-**select**-Anweisungen deklariert und verwaltet
- Explizite Cursor: werden vom Programmierer deklariert u. verwaltet

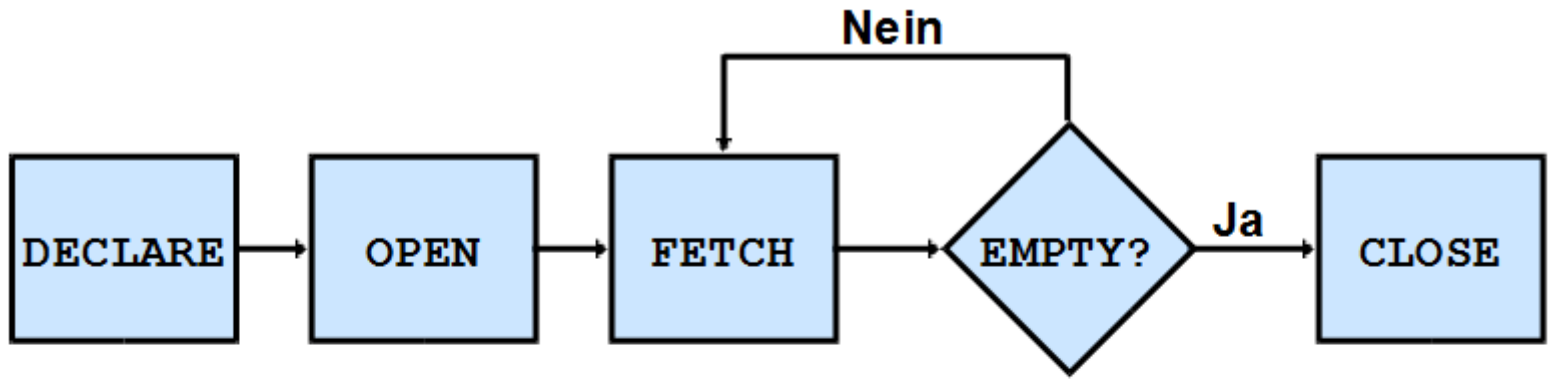
# Kontrollieren von expliziten Cursorn



## Kontrollieren von expliziten Cursorn (Forts.)



## Kontrollieren von expliziten Cursorn (Forts.)




- Erstelle einen benannten SQL-Bereich
- Identifiziere die aktive Menge
- Lade die aktuelle Zeile in Variablen
- Teste auf Existenz von Zeilen
- Kehre zu FETCH zurück, wenn Zeilen gefunden
- Gib die aktive Menge frei

## Deklarieren eines Cursors

Syntax:

**cursor** *Cursor_Name* **is**
**select**-*Anweisung*;

Beispiele:

```
declare
   cursor emp_cursor is
      select  employee_id, last_name from EMPLOYEES
      where department_id = 30;
   ...
   locid number:= 1700;
   cursor dept_cursor is
      select  * from DEPARTMENTS
      where location_id = locid;
   ...
```

## Verwenden eines Cursors

```
declare
   cursor emp_cursor is
      select  employee_id, last_name from EMPLOYEES
      where department_id = 30;
    empno EMPLOYEES.employee_id%TYPE;
    lname  EMPLOYEES.last_name%TYPE;
begin
   open emp_cursor;
   loop
      fetch emp_cursor into empno, lname;
      exit when emp_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(empno||' '||lname');
   end loop;
   close emp_cursor;
end;
```

## Cursor und Records

Man kann die aktuelle Zeile eines Cursors auch in einen PL/SQL-Record als Zeilenpuffer laden.

```
declare
   cursor emp_cursor is
      select  employee_id, last_name from EMPLOYEES
      where department_id = 30;
   emp_rec emp_cursor%ROWTYPE;
begin
   open emp_cursor;
   loop
      fetch emp_cursor into emp_rec;
   ...
```

## Cursor-for-Schleifen

Syntax:

```
for Record_Name in Cursor_Name loop
    Anweisung1;
    Anweisung2;
   ...
end loop;
```

- Die Cursor-**for**-Schleife ist eine Abkürzung, um explizite Cursor zu verarbeiten.
- Öffnen, Laden, Beenden und Schließen sind implizit enthalten.
- Auch der Record wird implizit (wie mit **%rowtype**) deklariert.
- Mit einer Unteranfrage statt des Cursornamens ist es nicht einmal mehr nötig, einen (nur einmal verwendeten) Cursor zu deklarieren.

# Verwendung von Cursor-for-Schleifen

```
declare
   cursor emp_cursor is
      select  employee_id, last_name from EMPLOYEES
      where department_id = 30;
begin
   for emp_rec in emp_cursor loop
      DBMS_OUTPUT.PUT_LINE(emp_rec.employee_id||' '||emp_rec.last_name);
   end loop;
end;
```

*oder:*

```
begin
   for emp_rec in
      (select  employee_id, last_name from EMPLOYEES
       where department_id = 30)
   loop
      DBMS_OUTPUT.PUT_LINE(emp_rec.employee_id||' '||emp_rec.last_name);
   end loop;
end;
```

## Explizite Cursor-Attribute

Um Status-Informationen über einen Cursor zu bekommen, gibt es folgende Cursor-Attribute:

| Attribute | Typ | Beschreibung |
|---|---|---|
| %ISOPEN | **boolean** | liefert TRUE, wenn der Cursor offen ist |
| %NOTFOUND | **boolean** | liefert TRUE, wenn das<br>letzte **fetch** keine Zeile zurückgegeben hat |
| %FOUND | **boolean** | liefert TRUE, wenn das<br>letzte **fetch** eine Zeile zurückgegeben hat;<br>Komplement von %NOTFOUND |
| %ROWCOUNT | **number** | liefert die Gesamtzahl<br>von bisher zurückgegebenen Zeilen |

## Verwendung von Cursor-Attributen

```
    if not emp_cursor%ISOPEN then                    -- Cursor noch nicht geöffnet?
       open emp_cursor;
    end if;
    loop
       fetch emp_cursor...
```

```
declare
    empno   EMPLOYEES.employee_id%TYPE;
    ename   EMPLOYEES.last_name%TYPE;
    cursor  emp_cursor is
       select employee_id, last_name from EMPLOYEES;
begin
    open emp_cursor;
    loop                                                -- nur max. 10 Ergebnisse
       fetch emp_cursor into empno, ename;
       exit when emp_cursor%ROWCOUNT>10
                    or emp_cursor%NOTFOUND;
       DBMS_OUTPUT.PUT_LINE(to_char(empno) ||' '|| ename);
    end loop;
    close emp_cursor;
end;
```

## Cursor mit Parametern

Syntax:

**cursor** *Cursor_Name*
  [(*Parameter_Name Datentyp*, ...)]
**is**
  **select**-*Anweisung*;

- Aktuelle Parameter-Werte werden an einen Cursor übergeben, wenn der Cursor geöffnet und die Anfrage ausgeführt wird.
- Man kann einen expliziten Cursor mehrmals mit verschiedenen Parametern und somit mit verschiedenen aktiven Mengen öffnen.

**open** *Cursor_Name*(*Parameter_Wert*, ...);

## Verwenden von Cursorn mit Parametern

```
declare
   cursor emp_cursor(deptno number) is
      select  employee_id, last_name
      from    EMPLOYEES
      where department_id = deptno;
    dept_id   number;
    lname     varchar2(15);
begin
   ...
   open emp_cursor(10);
   ...
   close emp_cursor;
   ...
   open emp_cursor(20);
   ...
```

## Spezialität: Die where-current-of-Klausel

Syntax:

**where current of** *Cursor*;

Beispiel:

```
...
update EMPLOYEES
set      salary = ...
where current of emp_cursor;
...
```

- Mit der **where-current-of**-Klausel kann man ein Update oder eine Löschung auf die Tabellenzeile beziehen, die der aktuellen Zeile eines Cursors zugrundeliegt (sofern erstere eindeutig ist).
- Bei Verwendung dieser Klausel muss man allerdings an die Cursor-Anfrage die **for-update** - Klausel anhängen, um die Zeilen für den Schreibzugriff zu sperren.

## 5.3 Dynamisches SQL

"Dynamic SQL" ist zu verwenden, wenn eine (beliebige) SQL-Anweisung oder ein PL/SQL-Block erzeugt werden soll, deren/dessen Struktur erst *zur Laufzeit* festliegt oder sich während der Laufzeit ändern kann, z.B. wenn die Tabellen- oder Spaltennamen variabel sind.

In der Anwendung (im PL/SQL-Programm) wird die Anweisung als String zusammengebaut und dann mit **execute immediate** ausgeführt[4].

Syntax:

**execute immediate** *Anweisungsstring*
[**into** { *Variable1* [, *Variable2* ...] } | *Record* ]
[**using** [*Modus*] *BindungsargumentA* [, [*Modus*] *Bindungsarg.B* ...];

- Die **into**-Klausel wird für Ein-Zeilen-Anfragen verwendet und legt fest, wohin die Ergebniswerte eingetragen werden sollen.
- Der Anweisungsstring darf Platzhalter ("Bindungsvariablen") für Spaltendaten enthalten, an die erst bei Ausführung Argumente gebunden werden. Die **using**-Klausel gibt die Bindungsargumente für die Platzhalter an. Der Modus für solche Argumentübergaben ist **in**, falls nicht explizit als {**in**|**out**|**in out**} angegeben.

[4] und dabei auch compiliert – also erst zur Laufzeit !

## Dynamisches SQL mit einer DDL-Anweisung

- Beispiel: Definition einer Prozedur zum Erstellen einer Tabelle

```
create procedure create_my_table(
   table_name varchar2, col_specs varchar2 )
is
begin
   execute immediate
      'create table MY_' || table_name || ' (' || col_specs || ')';
end;
```

- Beispiel: Der nachfolgende Aufruf dieser Prozedur ...

```
begin
    create_my_table(
      'EMPLOYEE_NAMES',
      'id number(4) primary key, name varchar2(40)' );
end;
```

- ... erzeugt eine Tabelle MY_EMPLOYEE_NAMES mit Spalten id / name.

## Dynamisches SQL mit einer DML-Anweisung

- Beispiel: Löschen aller Zeilen aus einer beliebigen Tabelle
  und Rückgabe einer Meldung

```
create function make_empty( table_name varchar2 )
return varchar2 is
begin
   execute immediate 'delete from ' || table_name;
   return (SQL%ROWCOUNT||' rows deleted from '||table_name);
end;

begin DBMS_OUTPUT.PUT_LINE(
   make_empty('EMPLOYEE_NAMES'));
end;
```

## Dynamisches SQL mit einer DML-Anweisung (Forts.)

- Beispiel: Einfügen von zwei Zeilen mit zwei Spalten in eine Tabelle

```
create procedure add_2x2rows(table_name varchar2,
   id number, stringA varchar2, stringB varchar2)
is
   stmt varchar2(50);
begin
   stmt:= 'insert into ' || table_name || ' values (:1, :2)';
   execute immediate stmt using id, stringA;
   execute immediate stmt using id+1, stringB;
end;
```

- Hier werden in der dynamischen SQL-Anweisung Platzhalter (alphanumerische Namen beginnend mit ':') benutzt, um zur Laufzeit Eingabewerte aus den Argumenten in der **using**-Klausel zu bekommen (oder Ausgabewerte dorthin zurückzugeben).
- Ohne String-Variable stmt und ohne Platzhalter müsste das erste Statement übrigens lauten:

```
execute immediate 'insert into ' || table_name ||
    ' values (''''||to_char(id)||''',''''||stringA||''''')';
```

## Dynamisches SQL mit einem PL/SQL-Block

```
create function get_annsal(emp_id number)
return number
is                                                    -- get_emp s. Folgeseite
   plsql varchar2(200) :=
      'declare'||
      '    emprec EMPLOYEES%rowtype; '||
      'begin '||
      '    emprec := get_emp(:empid); '||
      '    :res := emprec.salary*12; '||
      'end;' ;
   result number;
begin
   execute immediate plsql using in emp_id, out result;
   return result;
end;

execute DBMS_OUTPUT.PUT_LINE(get_annsal(100))
```

## Dynamisches SQL mit einer Ein-Zeilen-Anfrage

```
create function get_emp(emp_id number)
return EMPLOYEES%rowtype
is
   stmt varchar2(200);
   emprec EMPLOYEES%rowtype;
begin
   stmt := 'select * from EMPLOYEES ' || 'where employee_id = :id';
   execute immediate stmt into emprec using emp_id;
   return emprec;
end;

declare
   emprec EMPLOYEES%rowtype := get_emp(100);
begin
   DBMS_OUTPUT.PUT_LINE('Employee: ' || emprec.last_name);
end;
```

# Dynamisches SQL für Mehr-Zeilen-Anfragen bzw. Cursor

```
declare
   type CurTyp is ref cursor;
   csrvar      CurTyp;
   emp_rec     EMPLOYEES%rowtype;
   sql_stmt    varchar2(200);
   my_job      varchar2(10) := 'DB_PROF';
begin
   sql_stmt := 'select * from EMPLOYEES where job_id = :j';
   open csrvar for sql_stmt using my_job;
   loop
        fetch csrvar into emp_rec;
        exit when csrvar%NOTFOUND;
        ...
   end loop;
   close csrvar;
end;
```

- benötigt eine Cursor-*Variable*! (csrvar vom Typ **ref cursor**)
- Diese wird erst durch die erweiterte **open**-Anweisung an eine SQL-Anfrage gebunden.

## Dynamisches SQL für Mehr-Zeilen-Anfragen bzw. Cursor (Forts.)

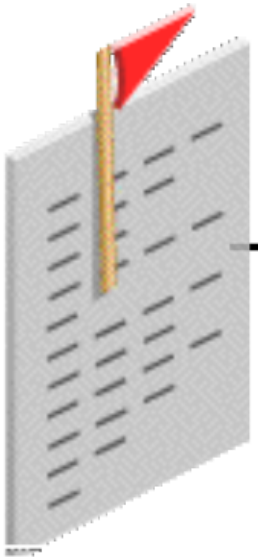
Es gibt zwei Typisierungsstufen für Cursor-Variablen:

```
create procedure process_data is
   type GenCurTyp is ref cursor;          -- schwach getypte Cursor-Variable
   type EMP_CurTyp is ref cursor
        return EMPLOYEES%rowtype;         -- stark getypte Cursor-Variable
                                          -- (nur für Cursor mit passendem Ergebnistyp verwendbar)
   ...
   generic_cv  GenCurTyp;
   emp_cv      EMP_CurTyp;
begin
   open generic_cv  for select ... from DEPARTMENTS where ...;
   open emp_cv      for select * from EMPLOYEES where ...;
                                          -- dann beide wie normale Cursor verwendbar
   ...
end;
```

# 5.4 Ausnahmebehandlung in PL/SQL

- Eine **Ausnahme (Exception)** ist ein Fehler, der während der Ausführung eines PL/SQL-Programms auftritt.
- Eine Exception kann ausgelöst ("geworfen") werden:
  - implizit vom Oracle-Server
  - explizit von einem Programm
- Eine Exception kann behandelt werden:
  - indem sie mit einem "Handler" abgefangen wird
  - oder indem sie an die aufrufende Umgebung weitergeleitet wird

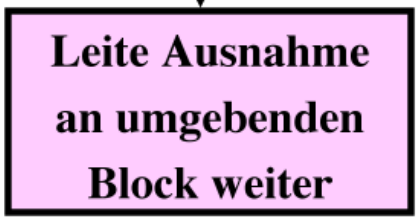
# Behandeln von Exceptions



Ausnahme wird in einem Block ausgelöst


nein


ja


## Abfangen von Exceptions

Beispiel:

```
declare
   lname varchar2(15);
begin
   select last_name into lname from EMPLOYEES
   where first_name = 'John';
   DBMS_OUTPUT.PUT_LINE('John''s last name is: ' || lname);
exception
   when TOO_MANY_ROWS then
        DBMS_OUTPUT.PUT_LINE('Your query retrieved multiple rows.
        Ask your programmer to use a cursor.');
end;
```

## **Abfangen von Exceptions** (Forts.)

Syntax:

**exception**
**when** *Exception1* [**or** *Exception1b* ...] **then**
*Anweisung11*;
*Anweisung12*;
...
[**when** *Exception2* [**or** *Exception2b* ...] **then**
*Anweisung21*;
*Anweisung22*;
...]
[**when others then**
*Anweisung91*;
*Anweisung92*;
...]

Zum Abfangen werden also Namen für die Exceptions benötigt.

## Arten von Exceptions

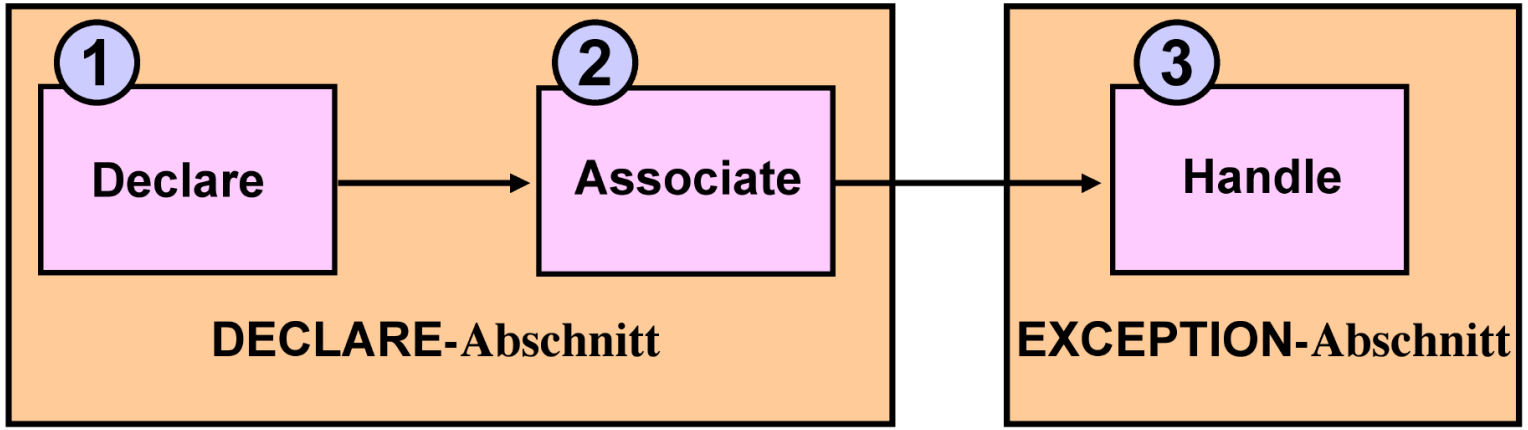
Implizit ausgelöst:

- vordefinierte Oracle-Server-Fehler haben bereits Namen, z.B.:
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - INVALID_CURSOR
  - ZERO_DIVIDE
  - DUP_VAL_ON_INDEX
- nicht-vordefinierte Oracle-Server-Fehler haben noch keine Namen

Explizit ausgelöst:

- benutzerdefinierte Ausnahmen, werden mit Namen deklariert

# Abfangen von nicht-vordefinierten Oracle-Server-Fehlern




**Benenne die Ausnahme.**

**Code “pragma exception_init”:**

**Assoziere den Namen mit einer Oracle-Fehlernummer**

**Behandle die ausgelöste Ausnahme.**

## Abfangen von nicht-vordefinierten Oracle-Server-Fehlern (Forts.)

Beispiel: Der Oracle-Server-Fehler mit der Nummer -01400 (meldet "cannot insert **null**") soll abgefangen werden.

```
declare
   insert_excep exception;                                    (1)
   pragma exception_init (insert_excep, -01400);              (2)
begin
   insert into DEPARTMENTS
      (department_id, department_name) values (280,null);
exception                                                     (3)
    when insert_excep then
       DBMS_OUTPUT.PUT_LINE('Fehler beim Einfuegen.');
       DBMS_OUTPUT.PUT_LINE('Das System meldet: ' || SQLERRM);
end;
```

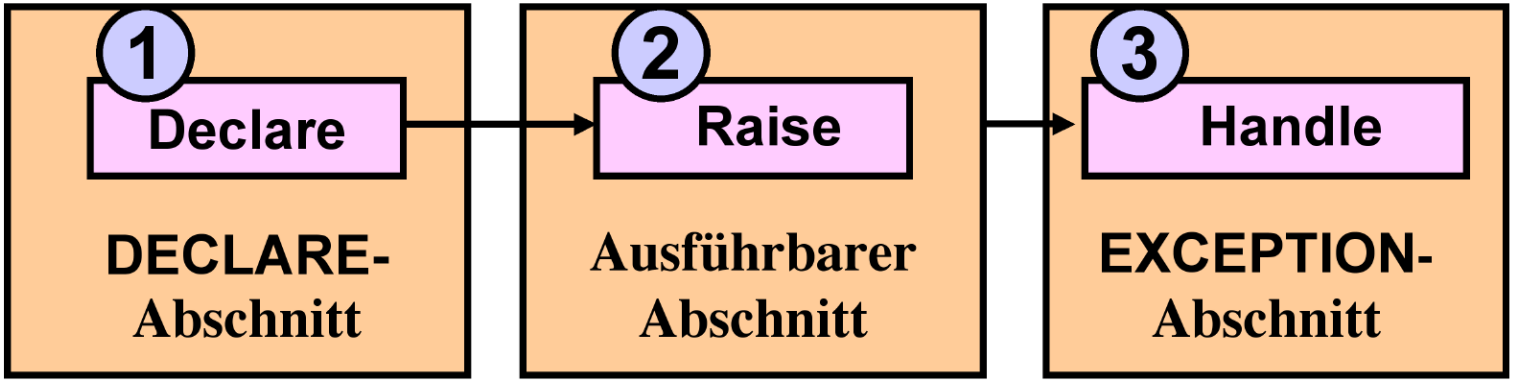
## Hilfsfunktionen zum Abfangen von Exceptions

- SQLCODE: gibt den numerischen Wert des Fehler-Codes zurück
- SQLERRM: gibt die Nachricht zurück, die mit der Fehlernummer verknüpft ist
  (Beide dürfen nicht direkt in einer SQL-Anweisung verwendet werden.)

Beispiel: Protokollierung sonstiger Fehler

```
declare
   error_code      number;
   error_message varchar2(255);
begin
   ...
exception
   ...
 when others then
    rollback;
    error_code := SQLCODE;
    error_message := SQLERRM;
    insert into MY_ERRORLOG_TABLE (e_user, e_date, e_code, e_message)
       values(user, sysdate, error_code, error_message);
end;
```

## Abfangen von benutzerdefinierten Exceptions




**Benenne die Ausnahme.**

**Löse die Ausnahme explizit durch Verwendung des raise-Befehls aus.**

**Behandle die ausgelöste Ausnahme.**

## Abfangen von benutzerdefinierten Exceptions (Forts.)

Beispiel:

```
declare
   invalid_department exception;          <-- 1
   name    varchar2(20);
   deptno number;
begin
   ...
   update DEPARTMENTS
   set       department_name = name
   where  department_id = deptno;
   if SQL%NOTFOUND then
      raise invalid_department;           <-- 2
   end if;
   commit;
exception                                  <-- 3
   when invalid_department then
      DBMS_OUTPUT.PUT_LINE('No such department id.');
end;
```

## Weiterleiten von Exceptions aus einem Unterblock

Unterblöcke können entweder eine Exception behandeln oder diese an den sie umschließenden Block weiterleiten.

```
declare
   ...
   no_rows  exception;
   integrity  exception;
   pragma exception_init (integrity,-2292);
begin
   for c_record in emp_cursor loop
      begin
         select ...
         update ...
         if SQL%NOTFOUND then
            raise no_rows;
         end if;
      end;
   end loop;
exception
   when integrity then ...
   when no_rows then ...
end;
```

## Die raise_application_error-Prozedur

- Syntax: **raise_application_error**(*Fehlernummer, Fehlermeldung*);
- Man kann diese Prozedur benutzen, um benutzerdefinierte Fehlernummern und -meldungen aus gespeicherten Prozeduren/Funktionen an die aufrufende Anwendung zurückzugeben.
- Die Rückgabe erfolgt so, als ein Oracle-Server-Fehler aufgetreten ist. SQLCODE und SQLERRM liefern die o.g. Argumente ab.
- Wenn in der aufrufenden Anwendung kein Abfangen erfolgt, sehen diese Fehler für den Benutzer der Anwendung wie Oracle-Server-Fehler aus, aber mit vom Entwickler der gespeicherten Prozeduren/Funktionen vorbereiteten Fehlermeldungen.
- So kann letzterer verhindern, dass Exceptions ganz unbehandelt (voll kryptisch) gemeldet werden. (Trotzdem nicht benutzerfreundlichst.)
- Die Fehlernummer kann aus dem Bereich -20000 ... -20999 gewählt werden.

## Die raise_application_error-Prozedur (Forts.)

Im ausführbaren Abschnitt:

```
begin
   ...
   delete from EMPLOYEES
   where manager_id = v_mgr;
   if SQL%NOTFOUND then
      raise_application_error(-20202, 'No manager to be deleted.');
   end if;
   ...
```

Oder im **exception**-Abschnitt:

```
   ...
   select * into emp_rec
   where employee_id = manager_id;
   ...
exception
   when NO_DATA_FOUND then
      raise_application_error(-20212, 'There is no self-managing employee.');
end;
```