Hinweise

- Dieses Skript beinhaltet evtl. Fehler, die von mir gewollt sind.
- Vermutlich gibt es in diesem Skript auch Fehler, die nicht von mir gewollt waren.
- Manche Folien / Beispiele sind unvollständig. Dies ist Absicht.
- Die Lösungen zu den Beispielen werden in der Vorlesung besprochen.

7. Aktive Inhalte in Oracle

PL /SQL und Trigger

Hinweis: Alle Graphiken und Abbildungen entstammen den Oracle Schulungsunterlagen der Oracle Academy.

Motivation

- SQL ***relational vollständig***, aber ***nicht berechnungsvollständig***
- Manchmal sind Aktionen wünschenswert,
  - die Schleifen oder Rekursion benötigen
  - die von der DB automatisch ausgeführt werden, z.B. beim Einfügen / Löschen von Datensätzen -> **Trigger**

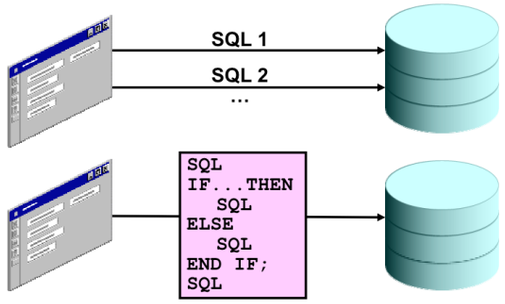
- Oracle bietet hierfür **PL/SQL, Java, C** als datenbankinterne Lösungen an

- **Programmieren von PL/SQL**
  - SQL-Schnittstelle
  - *SQL-Developer*
  - Oracle JDeveloper (*http://www.oracle.com/technetwork/developer-tools/jdev*)

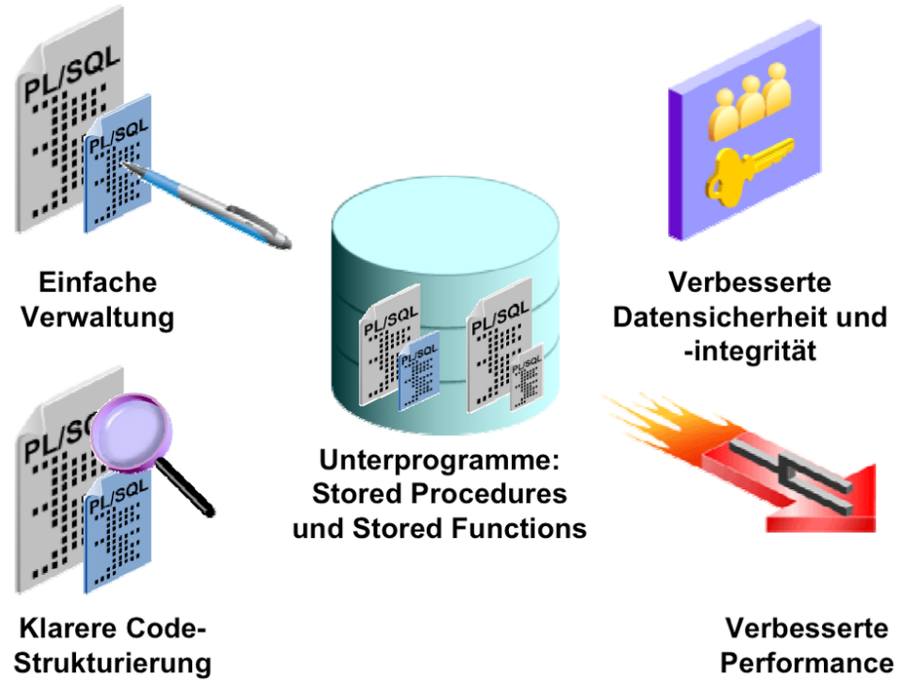
7.1 PL/SQL (Oracle)

PL/SQL

- PL/SQL = **P**rocedural **L**anguage extensions to the **S**tructured **Q**uery **L**anguage
- PL/SQL =
  - prozedurale Spracherweiterung von SQL
  - integriert prozedurale Konstrukte nahtlos in SQL
  - Oracles Standardsprache für den Datenzugriff bei relationalen Datenbanken
  - **berechnungsvollständige** Programmierschnittstelle innerhalb des DBMS



- PL/SQL
  - stellt eine **Blockstruktur** für ausführbaren Code bereit
  - bietet prozedurale Konstrukte, z.B.
    - Variablen, Konstanten, Datentypen, ...
    - Kontrollstrukturen (`IF ... THEN ... ELSE, LOOP, ...`)
    - Wiederverwendbare Programmblöcke,
      die einmal erstellt und mehrmals ausgeführt werden können



Universität
Augsburg
University

7.1.1 PL/SQL Grundlagen

PL/SQL Grundlagen

Universität Augsburg University

- Jeder Befehl wird mit einem **Semikolon** abgeschlossen
- **Groß- / Kleinschreibung**: keine Unterscheidung, aber *Konventionen*
  - **Großschreibung**
    - PL/SQL- und SQL-Schlüsselwörter (`BEGIN`, `SELECT`, ...)
    - Namen vordefinierter Funktionen (`SUBSTR`, ...)
    - vordefinierte Typen (`NUMBER`, ...)
  - **Kleinschreibung**
    - Datenbankobjekte (Tabellennamen, ...)
    - Variablennamen
- **Bezeichner**:
  - erstes Zeichen ist Buchstabe
  - besteht aus Buchstaben, Ziffern, $, #, _
  - ist bis zu 30 Zeichen lang
- **Kommentare**
  - Einzeilig: `-- Kommentar`
  - Mehrzeilig:
```
/*
   Kommentar
 */
```

PL/SQL Blockstruktur

- **`DECLARE`** (optional)
  - Variablen, Cursor, benutzerdefinierte Exceptions
- **`BEGIN`**
  - SQL-Anweisungen
  - PL/SQL-Anweisungen
- **`EXCEPTION`** (optional)
  - Aktionen, die durchgeführt werden, wenn Fehler auftreten
- **`END;`**

- **Blocktypen**

Anonym

```
[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;
```

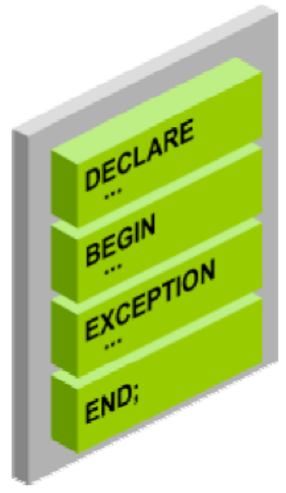
– Blöcke können Blöcke enthalten

– Deklarierte Variablen, Konstanten, Methoden, etc. sind in dem Block gültig, in dem sie deklariert wurden. Nach außen hin sind sie nicht sichtbar.

PL/SQL Grundlagen

- **Ausgabe:** Für die Ausgabe von beliebigen Werten in der „Standardausgabe“ gibt es

```
DBMS_OUTPUT.PUT(<string>);          -- schreibt den Eingabe-String
DBMS_OUTPUT.PUT_LINE(<string>);     -- bricht zusätzlich die Zeile
DBMS_OUTPUT.NEW_LINE();             -- bricht nur die Zeile um
```



- Die Ausgabe muss vorher in jeder Session „aktiviert“ werden:

```
SET SERVEROUTPUT ON;
```

- Bei Kompilierungsfehler lassen sich genauere Informationen über den Fehler ausgeben mit:

```
SHOW ERRORS;
```

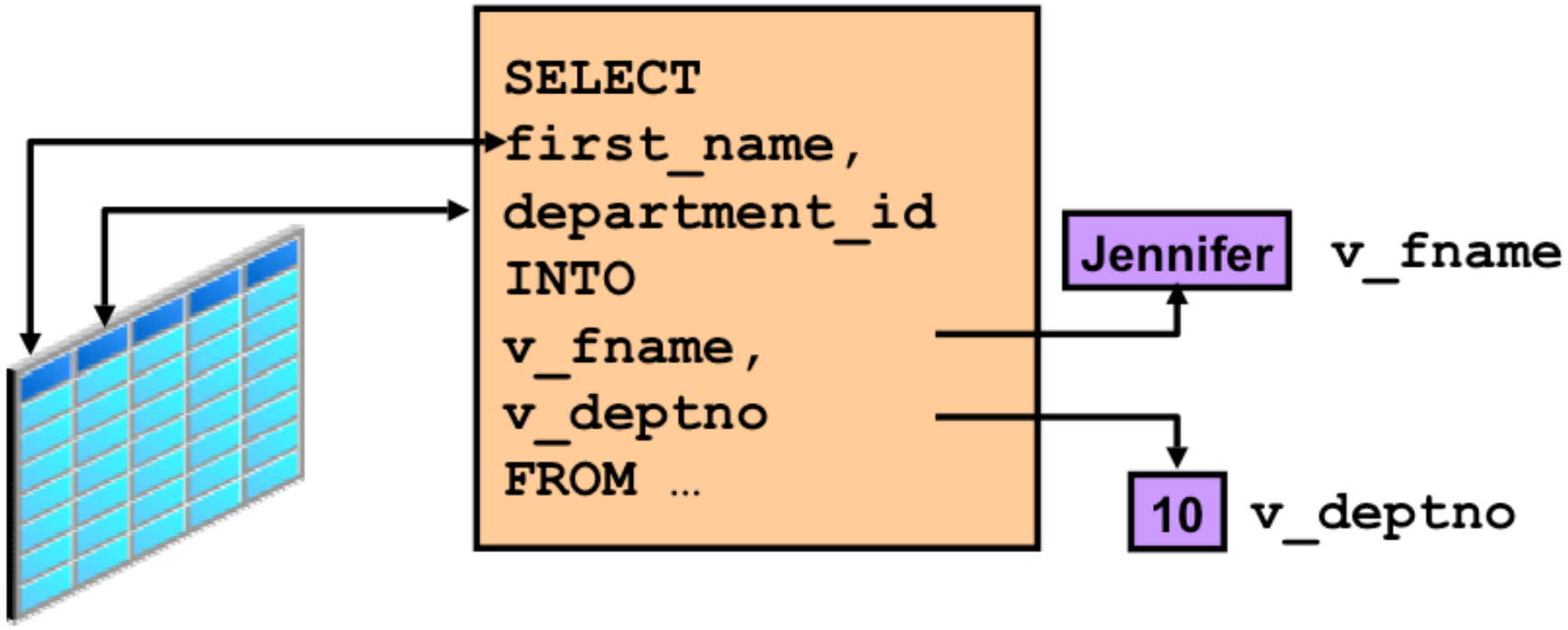
```
BEGIN
   DBMS_OUTPUT.PUT_LINE('My Name is Markus');
END;
```

Beispiel

7.1.2 Variablen und Datentypen

PL/SQL Variablen

- **Variablen**
  - Daten temporär speichern
  - Gespeicherte Werte bearbeiten
  - Wiederverwendbarkeit
  - müssen im `DECLARE` Bereich deklariert und initialisiert werden



PL/SQL Variablen

- **Syntax**

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

```
-- Beispiele
cal BOOLEAN;
unit VARCHAR2(2) NOT NULL DEFAULT 'ab';
pi CONSTANT FLOAT := 3.1415927;
```

– Ohne `DEFAULT` werden alle Variablen mit `NULL` initialisiert
– Wenn `NOT NULL` verwendet wird, muss ein Default-Wert angegeben werden
– `CONSTANT` deklariert eine Konstante
– Auch der Wert einer Konstante muss bei der Deklaration gesetzt werden

PL/SQL Beispiel

```
DECLARE
 v_myName VARCHAR2(20):= 'John';
BEGIN
 v_myName := 'Steven';
 DBMS_OUTPUT.PUT_LINE('My name is: '|| v_myName);
END;
/
```

PL/SQL Skalare Datentypen

- **Skalare Datentypen**:
  - Es werden die skalaren Datentypen aus (Oracle)-SQL übernommen.
  - Die Wertebereiche können sich allerdings unterscheiden!

Art	Typname	Wertebereich/Beschreibung
Ganzzahlen	`BINARY_INTEGER`	$-(2^{31}-1)\ldots+(2^{31}-1)$
	`PLS_INTEGER`	$-(2^{31}-1)\ldots+(2^{31}-1)$
Reelle Zahlen	`NUMBER(P, S)`	`P` $\in 0..38$, `S` $\in -84..127$ `P`: Vor-, `S`: Nachkommastellen
Zeichenketten bestimmter Länge	`VARCHAR2(L)`	$0 < L < 32768$ (in SQL nur 4000)
	`CHAR(L)`	$0 < L < 32768$ (in SQL nur 2000)
	`LONG(L)`	$0 < L < 32768$ (in SQL $2^{31}-1$)
	`NCHAR(L)`	$0 < L < 32768$ (in SQL nur 2000)
	`NVARCHAR2(L)`	$0 < L < 32768$ (in SQL nur 4000)
Binärdaten bestimmter Länge	`RAW(L)`	$0 < L < 32768$ (in SQL nur 255)
	`LONG RAW(L)`	$0 < L < 32768$ (in SQL $2^{31}-1$)
	`BFILE`	"Zeiger" auf Datei außerhalb des DBMS
	`BLOB`	binäre Daten bis zu 4 GB Länge
	`CLOB`	Textdaten bis zu 4 GB Länge, 1 Byte pro Zeichen
	`NCLOB`	Textdaten bis zu 4 GB Länge, Byte je Zeichen var.
Boolesche Werte	`BOOLEAN`	`TRUE, FALSE`
Zeitwerte	`DATE`	01.01.4712 v. Chr. 00:00:00 - 31.12.9999 23:59:59
	`TIMESTAMP(L) [WITH [LOCAL] TIME ZONE]`	`DATE` mit einer Auflösung von 10^{-L} Sekunden mit $0 \leq L \leq 9$ und optionaler Angabe von bei (`LOCAL` auf die DB normierter) Zeitzone
Datensatz-ID	`ROWID`	eindeutige ID jedes Datensatzes in der DB

PL/SQL Skalare Datentypen

- **`BINARY_INTEGER`**: keine Fehler beim Überlauf
- **`PLS_INTEGER`**: Fehler bei Überlauf

- **`BINARY_INTEGER`** besitzt Untertypen mit unterschiedlichen Wertemengen:

Typname	Wertebereich
NATURAL	$0 \ldots 2^{31} - 1$
NATURALN	$0 \ldots 2^{31} - 1$ NOT NULL
POSITIVE	$1 \ldots 2^{31} - 1$
POSITIVEN	$1 \ldots 2^{31} - 1$ NOT NULL
SIGNTYPE	$-1, 0, 1$

- **`NUMBER`** besitzt Untertypen mit identischen Wertemengen
  - `DEC, DECIMAL, DOUBLE PRECISION, FLOAT(P), INT, INTEGER, NUMERIC, REAL, SMALLINT`
- Leere Zeichenketten werden von PL/SQL wie `NULL`-Werte behandelt

PL/SQL Skalare Datentypen

- Definition eigener skalarer Datentypen:

```
SUBTYPE <neuer Typ> IS <vorhandener Typ>
```

- **Beispiel**:

```
-- Subtypen
DECLARE
  SUBTYPE VC10 IS VARCHAR2(10);
  v_myName VC10 := 'Stefan';
  v_name person.name%TYPE := 'Stefan';
BEGIN
  v_myName := 'Markus';
  DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myName);
  DBMS_OUTPUT.PUT_LINE('My name is: ' || v_name);
END;
```

PL/SQL Skalare Datentypen

Universität Augsburg University

%TYPE-Attribut

```
identifier      table.column_name%TYPE;
```

- Dient zum Deklarieren einer Variablen entsprechend:
  - der Definition einer Datenbankspalte
  - einer anderen deklarierten Variablen
- Erhält als Präfix:
  - die Datenbanktabelle und -spalte
  - den Namen der deklarierten Variablen
- Deklarationen dieser Art heißen ***verankert***.

Vorteile ?

```
...
  emp_lname       employees.last_name%TYPE;
...
```

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

PL/SQL Komplexe Datentypen

Universität Augsburg University

Komplexe Datentypen

- Können mehrere Werte aufnehmen
- Es gibt zwei Arten:
  - **PL/SQL-Records**: Speichern von Werten unterschiedlicher Datentypen, jeweils aber nur ein Vorkommen
  - **PL/SQL-Collections**: Speichern von Werten desselben Datentyps

PL/SQL Komplexe Datentypen

Universität Augsburg University

PL/SQL-Records

- Müssen ein oder mehrere Felder eines skalaren, RECORD- oder INDEX BY Tabellendatentyps enthalten

1
```
TYPE type_name IS RECORD
    (field_declaration[, field_declaration]…);
```

2
```
identifier    type_name;
```

field_declaration:

```
field_name {field_type | variable%TYPE
           | table.column%TYPE | table%ROWTYPE}
           [[NOT NULL] {:= | DEFAULT} expr]
```

PL/SQL Komplexe Datentypen

Universität Augsburg University

- **Beispiel:**

Beispiel

```
-- im Deklarationsteil
TYPE COMPLEX IS RECORD (
   r NUMBER,
   i NUMBER
);

val COMPLEX;

...

-- im ausgeführten Teil: Zuweisung komponentenweise
val.r := -5.0;
val.i := 1.2;
```

PL/SQL Komplexe Datentypen

Universität Augsburg University

%ROWTYPE-Attribut

- Deklaration von Variablen entsprechend einer Collection von Spalten in einer Tabelle oder einem View
- Felder im Record erhalten den Namen und Datentyp der Spalten aus der Tabelle bzw. aus dem View

```
DECLARE
   identifier  reference%ROWTYPE;
```

Vorteile ?

Vorteile:

- **Anzahl und Datentypen der zu Grunde liegenden Spalten sind unwichtig**
- **Änderungen der Attributnamen oder -datentypen haben keinen Einfluss auf die PL/SQL Prozedur**

PL/SQL Komplexe Datentypen

Universität Augsburg University

- **Beispiel zu %ROWTYPE:**

Beispiel

```
DECLARE
  v_employee_number number:= 124;
  v_emp_rec   employees%ROWTYPE;
BEGIN
 SELECT * INTO v_emp_rec FROM employees
 WHERE  employee_id = v_employee_number;
 INSERT INTO retired_emps(empno, ename, job, mgr,
                  hiredate, leavedate, sal, comm, deptno)
   VALUES (v_emp_rec.employee_id, v_emp_rec.last_name,
           v_emp_rec.job_id, v_emp_rec.manager_id,
           v_emp_rec.hire_date, SYSDATE,
           v_emp_rec.salary,  v_emp_rec.commission_pct,
           v_emp_rec.department_id);
END;
/
```

PL/SQL Komplexe Datentypen

Universität Augsburg University

- **Records mit `%ROWTYPE` einfügen:**

```
...
DECLARE
  v_employee_number number:= 124;
  v_emp_rec retired_emps%ROWTYPE;
BEGIN
 SELECT employee_id, last_name, job_id, manager_id,
 hire_date, hire_date, salary, commission_pct,
 department_id INTO v_emp_rec FROM employees
 WHERE  employee_id = v_employee_number;
 INSERT INTO retired_emps VALUES v_emp_rec;
END;
/
SELECT * FROM retired_emps;
```

PL/SQL Komplexe Datentypen

Universität Augsburg University

- **Zeilen in einer Tabelle mit einem Record aktualisieren**

```
SET VERIFY OFF
DECLARE
  v_employee_number number:= 124;
  v_emp_rec retired_emps%ROWTYPE;
BEGIN
 SELECT * INTO v_emp_rec FROM retired_emps;
 v_emp_rec.leavedate:=CURRENT_DATE;
 UPDATE retired_emps SET ROW = v_emp_rec WHERE
  empno=v_employee_number;
END;
/
SELECT * FROM retired_emps;
```

PL/SQL Komplexe Datentypen

Universität Augsburg University

- **Array-Typen**

```
TYPE <name> IS {VARRAY | VARIABLE ARRAY} (max_size)
   OF <typ> [NOT NULL];
```

- Erstes Element im Array hat den Index 1
- `max_size` legt die maximale Größe des Arrays fest.
- Beim Erzeugen eines Arrays werden die Initialwerte der Elemente im Konstruktor angegeben.
- Zugriff auf einzelne Element erfolgt über einen Index

```
TYPE INTARRAY IS VARRAY(20) OF INTEGER;

val INTARRAY := INTARRAY(4, 1, 2); -- val(1) ist 4
                                   -- val(2) ist 1
                                   -- val(3) ist 2
```

7.1.3 Steuerstrukturen

Steuerstrukturen

- **IF**

```
IF <boolean_ausdruck1> THEN
    <befehle>
  [ELSIF <boolean_ausdruck2> THEN
    <befehle>]
...
  [ELSIF <boolean_ausdruck2> THEN
    <befehle>]
  [ELSE <befehle>]
END IF;
```

- **LOOP**

```
LOOP
  <befehle>
END LOOP;
```

- **Abbruch einer Schleife mit**

```
EXIT [WHEN <boolean_ausdruck>];
```

Steuerstrukturen

- **WHILE**

```
WHILE <boolean_ausdruck> LOOP
  <befehle>
END LOOP;
```

- **FOR**

```
FOR <zähler> IN [REVERSE] <anfang>..<ende> LOOP
  <befehle>
END LOOP;
```

- `<zähler>` ist implizit `BINARY_INTEGER`
- `<zähler>` ist nur innerhalb der Schleife deklariert und
- verdeckt möglicherweise vorhandene Variablen von außen
- Unter- und Obergrenze werden zu Beginn der Schleife ein Mal errechnet und bleiben dann unveränderlich
- `REVERSE`: Umgekehrter Schleifendurchlauf

Steuerstrukturen

- **GOTO:**
  Wird **GOTO** verwendet, springt das Programm sofort zu der Anweisung, an der die Marke gesetzt ist.

```
<<label>>
GOTO label;
```

- **EXIT: Schleifen mit Marken**
  Wird direkt vor eine **Schleife** eine Marke gesetzt, kann aus der Schleife heraus mit Hilfe von **EXIT** *an das Ende der Schleife* gesprungen werden, z.B für Beendigung von verschachtelten Schleifen.

- **NULL als Anweisung**
  NULL als Anweisung führt keine Aktion aus. Sie kann z.B. dazu dienen, deutlich zu machen, dass an ihrer Stelle nicht Code fehlt, sondern absichtlich nichts passiert.

```
-- GOTO vs. EXIT
DECLARE
 square BINARY_INTEGER;
BEGIN
 -- Beispiel für EXIT
 <<sqr>>      -- Marke für EXIT
 FOR i IN 1..20 LOOP
    square := i * i;
    DBMS_OUTPUT.PUT_LINE(square);
       IF square > 20 THEN
          EXIT sqr;
       END IF;
 END LOOP;

 DBMS_OUTPUT.PUT_LINE('EXIT erfolgreich beendet');

 -- Beispiel für GOTO
 FOR i IN 1..20 LOOP
   square := i * i;
   DBMS_OUTPUT.PUT_LINE(square);
      IF square > 20 THEN
         GOTO ende;
      END IF;

 <<ende>>       -- Marke für GOTO
 DBMS_OUTPUT.PUT_LINE('Das ist das Ende für GOTO');
END;
```

Steuerstrukturen

Universität Augsburg University

Beispiel

```
DECLARE
  up BINARY_INTEGER := 20;
  square BINARY_INTEGER;
  temp BINARY_INTEGER;

BEGIN
  <<sqr>>    -- Marke
  FOR i IN 1..up LOOP
    square := i * i;
    DBMS_OUTPUT.PUT_LINE(i || ' * ' || i || ' = ' || square);

    temp := 0;
    FOR j IN REVERSE 1..square LOOP
      temp := temp + j;
      IF temp > 100 THEN
        EXIT sqr;
      ELSIF temp = 100 THEN
        DBMS_OUTPUT.PUT_LINE('Summe von ' || j || ' bis ' ||
          square || ' ist 100');
      ELSE NULL;
      END IF;
    END LOOP;
  END LOOP;
END;
```

Universität
Augsburg
University

7.1.4 Exception Handling

Exception Handling

- **Exception-Handling** in der PL/SQL Blockstruktur

Anonym

```
[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;
```

- Exceptions können SQL-Fehler (z.B. Verletzung von Constraints) oder Programmfehler in PL/SQL sein.
- Um eine Exception abzufangen, die in einem Ausführungsblock auftritt, muss ein Exception-Block definiert werden.
- Tritt eine Exception auf, die nicht im gleichen Block abgefangen wird, wird sie an den umgebenden Block weitergegeben.
- Wird eine Exception im äußersten Block des Programms nicht abgefangen, wird von Oracle eine Fehlermeldung zurückgeliefert.

Exception Handling

- **Abfangen von Exceptions: Exception-Block**

```
EXCEPTION
  WHEN <exception_name> [OR <exception_name>] THEN <befehle>;
  ...
  WHEN <exception_name> [OR <exception_name>] THEN <befehle>;
  [WHEN OTHERS THEN <befehle>;]
```

- **`WHEN OTHERS`** fängt alle Arten von Exceptions ab.
- Die Funktionen `SQLCODE` bzw. `SQLERRM` geben die Oracle-Fehlernummer bzw. die Oracle-Fehlermeldung zu einer Exception zurück.
- SQLERRM lässt sich auch mit einer Zahl als Parameter aufrufen und gibt dann die Fehlermeldung für den Fehler mit dieser Nummer zurück.

Exception Handling

- **Vordefinierte Exceptions** in Oracle:

Nr.	Name	Beschreibung
-1722	INVALID_NUMBER	Zeichenstring kann nicht in Zahl konvertiert werden
-1476	ZERO_DIVIDE	Division durch 0
	SELECT INTO	
100	NO_DATA_FOUND	SELECT-INTO-Anweisung liefert keine Zeile zurück
-1422	TOO_MANY_ROWS	SELECT-INTO-Anweisung liefert mehr als eine Zeile zurück
	Cursor	
-6511	CURSOR_ALREADY_OPEN	ein Cursor ist beim OPEN bereits geöffnet
-1001	INVALID_CURSOR	ein Cursor ist noch nicht geöffnet, wird aber bereits so verwendet, z. B. mit FETCH
~~-1001~~ -6504	ROWTYPE_MISMATCH	FETCH in Variable eines anderen Typs als ROWTYPE
	VArrays und Tabellen	
-6532	SUBSCRIPT_OUTSIDE_LIMIT	Verweis auf Index außerhalb des deklarierten Bereichs
-6533	SUBSCRIPT_BEYOND_COUNT	Verweis auf zu großen Index

Exception Handling

- **Exceptions selbst auslösen mit**

```
RAISE <exception_name>;
```

  Auslösen aller vor- und selbst definierter Exceptions.

- Zudem gibt es noch

```
RAISE_APPLICATION_ERROR(<nummer>, <meldung>[, <behalte_fehler>]);
```

  - `<nummer>` ist eine Fehlernummer zwischen -20999 und -20000
  - `<meldung>` ist die Fehlermeldung
  - `<behalte_fehler>` ist ein Boole‘scher Wert, der angibt, ob der Fehler zur bisher vorhandenen Fehlerliste hinzugefügt werden soll (`TRUE`), oder diese ersetzen soll (`FALSE`, der Standard).

Exception Handling

Universität Augsburg University

Benutzerdefinierte Exceptions

- werden wie Variablen im Deklarationsteil eines Blocks deklariert.

```
<name> EXCEPTION;
```

- gleiche Sichtbarkeit wie eine im gleichen Block definierte Variable.
- Exception kann mit `RAISE` ausgelöst und mit einer entsprechenden Exception-Klausel abgefangen werden.
- Mit Hilfe eines Pragmas kann man selbstdefinierten Exceptions bereits vorhandene Fehlercodes zuweisen:

```
PRAGMA EXCEPTION_INIT(<name>, <fehlercode>);
```

- Wenn jetzt die Exception `<name>` geworfen wird, kann sie mit einem `WHEN` abgefangen werden, das den vordefinierten Oracle-Fehler mit dieser Fehlernummer abfängt.

Exception Handling

Universität Augsburg University

Beispiel

```
DECLARE
  nothing_here EXCEPTION;
  PRAGMA EXCEPTION_INIT(nothing_here, 100);

BEGIN
  IF TRUE THEN
    RAISE nothing_here;
  END IF;

  EXCEPTION
    WHEN NO_DATA_FOUND THEN       -- NO_DATA_FOUND = 100
    DBMS_OUTPUT.PUT_LINE('nothing found');
END;
```

Universität
Augsburg
University

7.1.5 Das ‚SQL‘ in PL/SQL

Das ‚SQL' in PL/SQL

Universität Augsburg University

`SELECT`-Statements mit `INTO`

- können direkt im PL/SQL-Code ausgeführt werden
- Das Ergebnis der Abfrage muss genau eine Zeile sein!
- Speichern der Ergebnisse mit der `INTO`-Klausel
- Die `INTO`-Klausel befindet sich zwischen `SELECT` und `FROM`

Beispiel

```
DECLARE
  current DATE;
  id ROWID;

  TYPE MYTYPE IS RECORD (dt DATE, id ROWID);
  rec MYTYPE;

BEGIN
  SELECT SYSDATE, ROWID INTO current, id FROM dual;
  DBMS_OUTPUT.PUT_LINE('Zeit: ' || current);
  DBMS_OUTPUT.PUT_LINE('Row-ID: ' || id);

  SELECT SYSDATE, ROWID INTO rec FROM dual;
  DBMS_OUTPUT.PUT_LINE('Zeit: ' || rec.dt);
  DBMS_OUTPUT.PUT_LINE('Row-ID: ' || rec.id);
END;
```

Das ‚SQL' in PL/SQL

CURSOR

- **Abfragen, die mehr als eine Zeile zurück liefern benötigen *statische Cursor***.
- Damit können Ergebnisse von SELECT-Statements zeilenweise durchlaufen werden.

```
CURSOR <name> [(<par> <typ>, ..., <par> <typ>)] IS <SQL>;
OPEN <name>;
FETCH <name> INTO <var>;
...
FETCH <name> INTO <var>;
CLOSE <name>;
```

- Ein stat. Cursor wird bei der Deklaration fest an eine SELECT-Anfrage gebunden.
- Nach der Deklaration kann ein Cursor mit OPEN geöffnet und mit CLOSE geschlossen werden.
- Mit FETCH kann der Cursor zeilenweise durchlaufen werden.
- Die Spaltenwerte der aktuellen Zeile können den entsprechenden Variablen zugewiesen werden.
- Ein Cursor kann beliebig oft geöffnet, durchlaufen und geschlossen werden.
- Im SQL-Statement eines Cursors können bereits definierte PL/SQL-Variablen vorkommen.
- Ebenso können Cursor-Parameter verwendet werden (Typdeklaration ohne Angabe der Größe des Typs!)

CURSOR

- **Cursor** können auch in einer Cursor-FOR-Schleife durchlaufen werden:

```
FOR <name> IN <cursor> LOOP
...
END LOOP;
```

  - Die Schleifenvariable <name> ist vom gleichen Typ wie die zurückgelieferten Zeilen des Cursors.
  - Der Cursor wird automatisch geöffnet und geschlossen.

Das ‚SQL' in PL/SQL

Universität Augsburg University

CURSOR besitzen verschiedene Deklarationsattribute

- **`ROWTYPE`**
  entspricht dem gleichnamigen Attribut von Tabellen.

- **`ISOPEN`**
  gibt an, ob der Cursor im Moment geöffnet ist.

- **`ROWCOUNT`**
  liefert die Anzahl der Zeilen, die bisher mit Hilfe von `FETCH` zurückgeliefert wurden.

- **`FOUND`**
  gibt `TRUE` zurück, wenn das letzte `FETCH` eine Zeile zurückgegeben hat.

- **`NOTFOUND`**
  ist stets die Negation von FOUND.

Das ‚SQL' in PL/SQL

Universität Augsburg University

Beispiele

```
-- Cursor ohne Parameter
DECLARE
                -- Cursor ohne Parameter
                CURSOR emp_c IS
                  SELECT first_name, last_name FROM employee;
                var01 VARCHAR2(20);
                var02 VARCHAR2(20);
BEGIN
      -- alle Zeilen lesen
      OPEN emp_c;
      LOOP
                FETCH emp_c INTO var01, var02;
                EXIT WHEN emp_c%NOTFOUND;
                DBMS_OUTPUT.PUT_LINE(var01 || ' ' || var02);
      END LOOP;
      CLOSE emp_c;
END;
```

```
-- Cursor, der Variable benutzt
DECLARE
                -- Cursor, der Variable benutzt
                varSalary INTEGER := 0;
                CURSOR emp_c IS
                  SELECT first_name, last_name
                  FROM employee
                  WHERE salary = varSalary;
                var01 VARCHAR2(20);
                var02 VARCHAR2(20);
BEGIN
      -- nur eine Zeile lesen
      OPEN emp_c;
      FETCH emp_c INTO var01, var02;
      DBMS_OUTPUT.PUT_LINE(var01 || ' ' || var02 || ' ist ein armes Schwein.');
      CLOSE emp_c;

      -- alle Zeilen mit einer Cursor-FOR-Schleife lesen
      FOR line IN emp_c LOOP
                DBMS_OUTPUT.PUT_LINE(line.first_name || ' ' || line.last_name
                        || ' ist ein armes Schwein.');
      END LOOP;
END;
```

```
-- Cursor mit Parameter
DECLARE
                -- Cursor mit Parameter
                CURSOR with_par (varSalary INTEGER := 0) IS SELECT first_name, last_name FROM employee WHERE salary = varSalary;
                var01 VARCHAR2(20);
                var02 VARCHAR2(20);
BEGIN
  -- Aufruf mit Parameter
      OPEN with_par(1000);
      FETCH with_par INTO var01, var02;
      IF with_par%FOUND THEN     -- wenn Tupel vorhanden
            DBMS_OUTPUT.PUT_LINE(var01 || ' ' || var02 || ' ist immer noch ein armes Schwein.');
      END IF;
      CLOSE with_par;

      -- Aufruf mit Parameter: Standardwert verwenden
      FOR line IN with_par LOOP
                DBMS_OUTPUT.PUT_LINE(line.first_name || ' ' || line.last_name || ' ist ein armes Schwein.');
      END LOOP;
END;
```

Das ‚SQL' in PL/SQL

Universität Augsburg University

`INSERT, UPDATE, DELETE`-Statements

- einfach als Befehle in PL/SQL.
- keine Cursor notwendig.

Beispiel

```
DECLARE
   v_id BINARY_INTEGER := 3;
   name VARCHAR(20) := 'markus';
   job_id INTEGER;
   v_first_name VARCHAR2(20) := 'florian';
   v_last_name VARCHAR(20) := 'wenzel';
BEGIN
      -- Gehalt neu setzen
      UPDATE employee SET salary = 100000 WHERE first_name = name;

      -- neuen Angestellten einfuegen
      INSERT INTO employee (id, first_name, last_name)
                  VALUES (v_id, v_first_name, v_last_name);

      -- loesche Angestellten mit JOB_ID = 1
      job_id := 1;
      DELETE FROM employee WHERE job_id = job_id;
END;
```

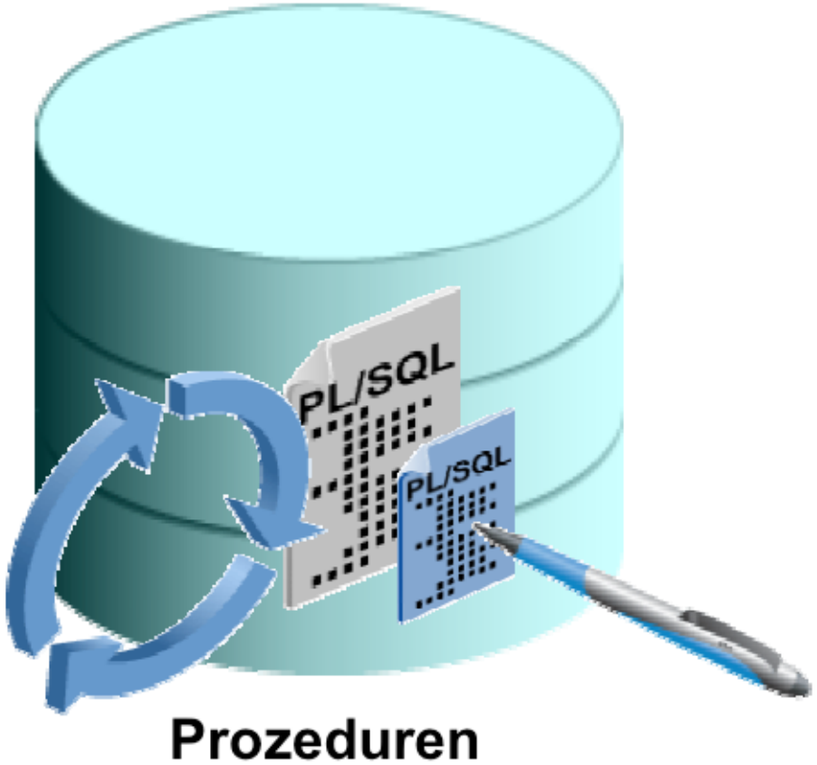
Das ‚SQL' in PL/SQL

Universität Augsburg University

- PL/SQL unterstützt nur eine Teilmenge von SQL: die DML
  - SELECT, INSERT, UPDATE, DELETE
- Es gibt zwei *normale* Arten, SQL-Befehle in PL/SQL zu verwenden:
  - direkt im Programmcode
  - mit einem explizit definierten Cursor

- Bei eingebettetem SQL in PL/SQL ist zu beachten:
  - Tabellen- und Spaltennamen werden automatisch erkannt.
  - Hat ein Parameter den gleichen Namen wie eine Tabelle oder Spalte, so wird er als Tabellen- bzw. Spaltennamen verwendet.
  - PL/SQL-Variablen sollten daher immer eindeutig sein.
  - Parameter sind nur bei Ausdrücken möglich, z.B. Argumenten von WHERE-Klauseln.
  - D.h. auch, dass Tabellen- und Spaltennamen nicht parametrisierbar sind.

7.1.6 Stored Functions und Stored Procedures



Prozeduren

Stored Functions & Procedures

- Bislang nur anonyme Blöcke, d.h. Blöcke ohne Namen, die nicht von außerhalb des Blocks angesprochen werden können.

Anonym

```
[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;
```

- Stored Functions bzw. Stored Procedures sind *benamte* Blöcke.
  - ***Stored Functions*** vergleichbar mit Funktionen einer Programmiersprachen. Haben immer einen Rückgabewert.
  - ***Stored Procedures*** analog zu Prozeduren oder Methoden ohne Rückgabewert.

- Programmierung in *PL/SQL*, C oder Java.

Stored Functions & Procedures

- **Stored Procedure:** Benannter Block ohne Rückgabewert

```
PROCEDURE <Name> (
 <Parameter>,
 ...,
 <Parameter>)
```

- Durch die Verwendung von **`OUT`**-Parametern können auch aus Prozeduren Werte zurückgeliefert werden, obwohl sie keinen Rückgabewert besitzen.

Stored Functions & Procedures

Universität Augsburg University

- **Stored Functions:** Benannter Block mit Rückgabewert

```
FUNCTION <Name> (
  <Parameter>,
  ...,
  <Parameter>)
 RETURN <Typ>
```

- <Typ> ist der Rückgabetyp.
- Durch die Verwendung von **OUT**-Parametern können zusätzlich zum Rückgabewert weitere Ergebnisse zurückgeliefert werden.

- Rückgabe des Ergebnisses mit Hilfe von

```
RETURN <Wert vom Rueckgabetyp>;
```

Funktions-/Prozedur Parameter

- Benannte Blöcke unterstützen Parameter, die beim Aufruf des Blocks übergeben werden.

```
<Name> [<Modus>] <Typ> [ := <Default>]
```

  - <Name> ist der Bezeichner
  - <Modus> ist der Verwendungstyp (Default IN-Parameter)
    - **IN** (Default): Nur-Lese-Parameter. Diese Parameter können innerhalb des Blocks nur gelesen werden. Einziger Modus, bei dem nach der Typdeklaration ein Standardwert angegeben werden kann.
    - **OUT**: Ein OUT-Parameter kann nicht gelesen werden. Ihm können neue Werte zugewiesen werden. Ein OUT-Parameter muss eine Variable sein. Der Wert wird nur geschrieben, wenn keine Exception auftritt.
    - **IN OUT**: IN OUT Parameter können gelesen und geschrieben werden. Müssen stets Variablen sein.
  - <Typ> ist der Datentyp
    - Es sind nicht alle Datentypen erlaubt, insbesondere keine Cursor.
    - Hier muss auf Cursor-Variablen ausgewichen werden.
  - <Default> ist ein Standardwert

Stored Functions & Procedures

- **Erstellen von Funktionen und Prozeduren**
  Um einzelne Funktionen / Prozeduren in der Datenbank zu speichern, müssen sie mit **`CREATE`** erzeugt werden.

```
CREATE PROCEDURE add_up(val01 NUMBER, val02 IN NUMBER)
IS
  summe NUMBER;
BEGIN
   summe := val01 + val02;
  DBMS_OUTPUT.PUT_LINE('Summe: ' || summe);
END add_up;

-- Prozedur / Funktion löschen
DROP FUNCTION | PROCEDURE ...
```

Beispiel

- **`CREATE OR REPLACE`** erstellt eine neue Prozedur bzw. ersetzt eine bereits vorhandene.
- Selbst definierte Funktionen / Prozeduren werden genauso verwendet wie vordefinierte.
- Dabei muss das Ergebnis einer Funktion immer als Wert weiterverarbeitet werden.
- Parameter werden beim Aufruf in Klammern angegeben.
- Werden keine Parameter angegeben, können die Klammern weggelassen werden.

Stored Functions & Procedures

Universität Augsburg University

```
CREATE FUNCTION add_function(val01 NUMBER := 3,
                             val02 IN NUMBER,
                             count_neg_values OUT NUMBER)
  RETURN NUMBER
IS
  result NUMBER;
BEGIN
  IF val01 < 0 AND val02 < 0 THEN
    count_neg_values := 2;
  ELSIF val01 < 0 OR val02 < 0 THEN
    count_neg_values := 1;
  ELSE
    count_neg_values := 0;
  END IF;

  result := val01 + val02;

  RETURN result;
END add_function;   -- Angabe des Funktionsnamens optional
```

Beispiel

Aufruf einer Funktion in einem PL/SQL-Block:

```
-- Signatur von add_function, siehe vorherige Folie
FUNCTION add_function(val01 NUMBER := 3,
                      val02 IN NUMBER,
                      count_neg_values OUT NUMBER) RETURN NUMBER
```

Beispiel

```
DECLARE
  result NUMBER;
  count_neg_values NUMBER;

BEGIN
  result := add_function(
            val02 => 0,
            count_neg_values => count_neg_values);

  DBMS_OUTPUT.PUT_LINE('Summe: ' || result);
  DBMS_OUTPUT.PUT_LINE('Summanden kleiner 0: ' || count_neg_values);
END;
```

Aufruf von Funktionen und Prozeduren

- Funktionen und Prozeduren werden wie folgt im SQL-Developer aufgerufen:

```
EXEC <Prozedur>
```

- In Java dient dazu die Methode `prepareCall(String call)`, die zur Klasse `Connection` im Package `java.sql` gehört.

- **Funktionen in SQL-Ausdrücken:**

```
CREATE OR REPLACE FUNCTION tax(p_value IN NUMBER)
 RETURN NUMBER IS
BEGIN
   RETURN (p_value * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
FROM employees
WHERE department_id = 100;
```

Aufruf von Funktionen und Prozeduren in SQL-Ausdrücken - Einschränkungen

- ausschließlich `IN`-Parameter erlaubt
- nur SQL-Datentypen, keine PL/SQL-spezifischen Typen
- User muss `EXECUTE`-Privileg besitzen
- In SELECT-Anweisungen dürfen die Funktionen keine DML-Anweisungen enthalten

Universität
Augsburg
University

7.1.7 Pakete

Pakete

- Pakete / Packages sind Sammlungen verschiedener PL/SQL Objekte.
- Strenge Unterscheidung zwischen ***Spezifikation*** und ***Implementierung*** (Rumpf).

- **Package-Spezifikation** = Schnittstelle nach außen
  - Nur auf die dort definierten Objekte (Variablen, Konstanten, etc.) kann von außerhalb des Packages zugegriffen werden.
  - Spezifikation eines Packages mittels

```
-- Spezifikation
PACKAGE <Paketname> IS
  <Deklaration>
  ...
  <Deklaration>
END <Paketname>;
```

- **Package-Rumpf**
  - Enthält die Implementierungen der in der Spezifikation angegebenen Deklarationen

- **Zugriff auf Package-Elemente**
  - Der Zugriff auf Package-Elemente, die in der Spezifikation definiert wurden, erfolgt von außerhalb des Packages mit dem Package-Namen

```
-- Implementierung (Rumpf)
PACKAGE BODY <Paketname> IS
    <Deklarationen>
    ...
    <Implementierung>
END <Paketname>;
```

Pakete

- In Oracle gibt es eine große Zahl verschiedener ***PL/SQL-Standardpakete***
- Alle Pakete sind zu finden in

Oracle PL/SQL Packages and Types Reference

- Einige Pakete sind:
  - **`DBMS_JOB`:** Automatische Ausführung von PL/SQL-Anweisungen zu einer bestimmten Zeit.
  - **`DBMS_OUTPUT`**: Für Ausgaben.
  - **`DBMS_PIPE`**: Kommunikation zwischen verschiedenen Sessions mittels Pipes.
  - **`DBMS_SQL`**: Ermöglicht die Angabe dynamischer SQL-Befehle.
  - **`URL_HTTP`**: Aufbau von HTTP-Verbindungen, z.B. für einen Web-Service oder zum Lesen einer URL, um die darin befindlichen Information zu verarbeiten.

7.2 Trigger

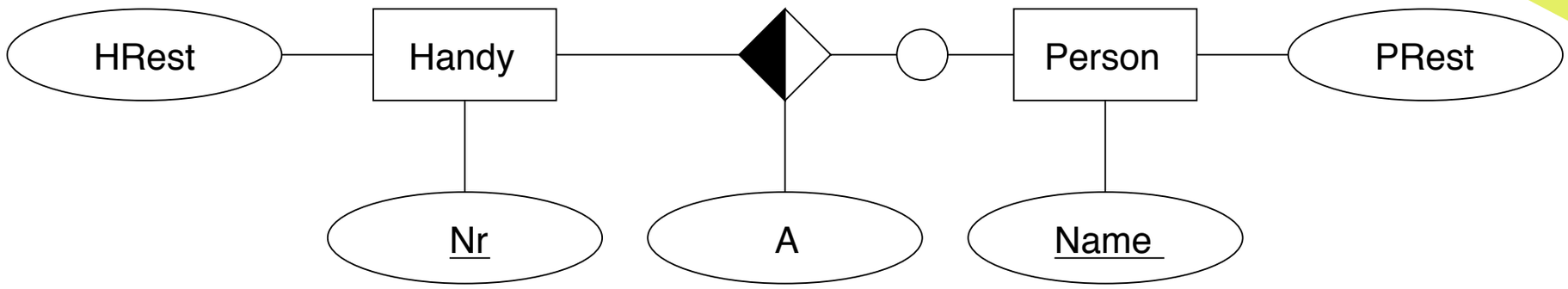
Relationships: n:1

eine Teilnahme optional (1)

Universität Augsburg University

Motivation: Folie 25 aus Kapitel 4

Manche Handys haben keinen Besitzer,
aber alle Personen haben mindestens ein Handy




```
CREATE TABLE Person (
 Name   ... PRIMARY KEY,
 PRest  ... NOT NULL
 CHECK (SELECT COUNT(*)
          FROM R
          WHERE PName = Name) > 0)
);

CREATE TABLE Handy (
 Nr     ... PRIMARY KEY,
 HRest  ... NOT NULL
);
```

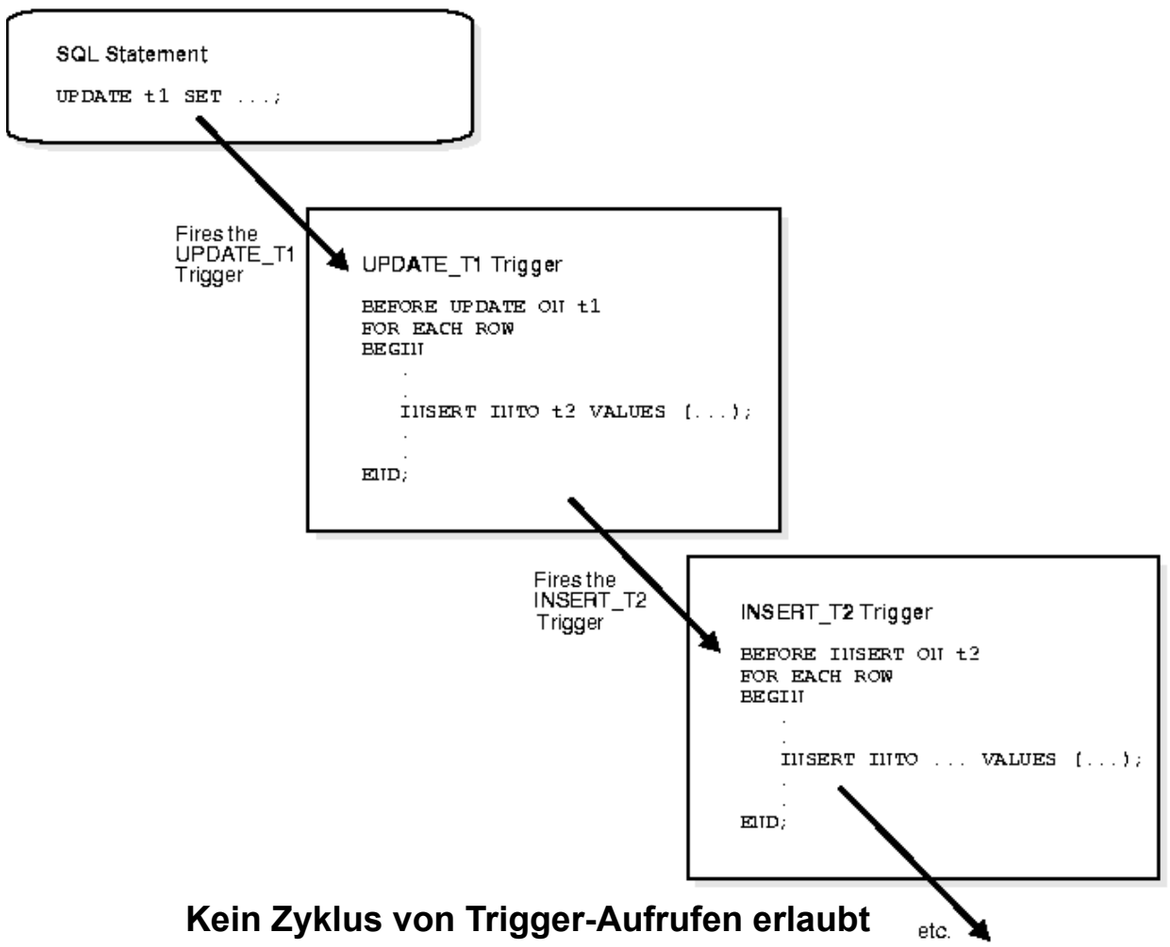
```
CREATE TABLE R (
 HNr    ... PRIMARY KEY REFERENCES Handy(Nr),
 PName  ... NOT NULL REFERENCES Person(Name),
 A      ... NOT NULL
);
```

Anmerkung: Kaum eine DB beherrscht Subqueries in CHECK-Klauseln.

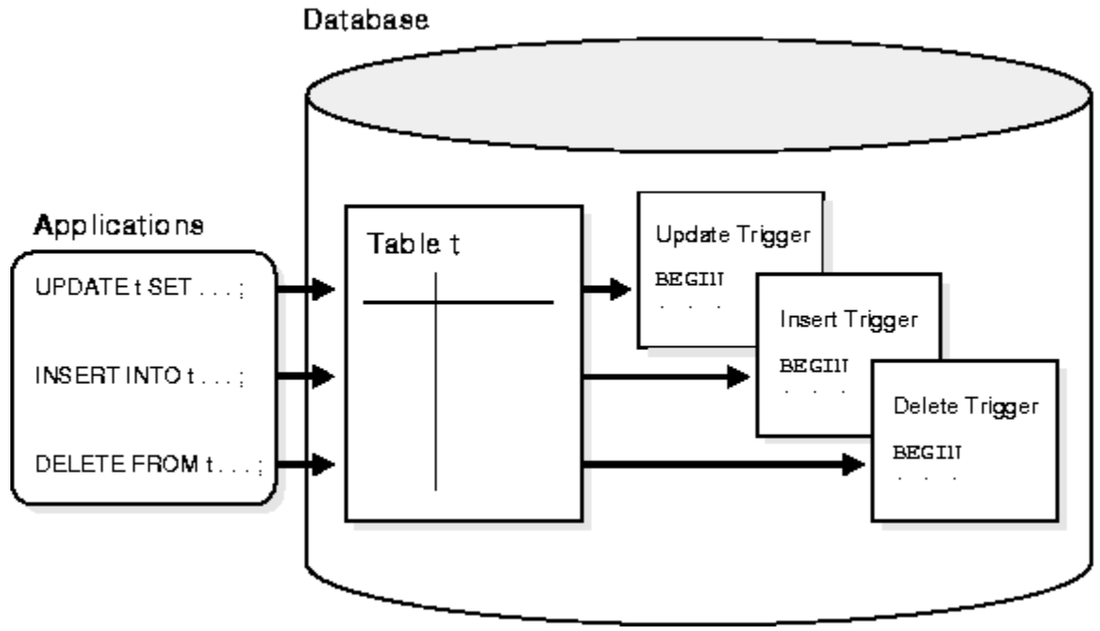
Wie kann dieses Problem gelöst werden?
Antwort: Trigger

Trigger

- Trigger sind spezielle Prozeduren in Oracle
- Ein Trigger ist eine spezielle Form einer Stored Procedure
- Trigger sind referenziert mit einer Tabelle oder einem View
- Trigger werden (je nach Art) automatisch nach einem **`INSERT`**, **`UPDATE`**, oder **`DELETE`** Statement ausgeführt.
- Trigger sind wesentlich mächtiger als `CHECK`-Klauseln



Kein Zyklus von Trigger-Aufrufen erlaubt



Trigger

Universität Augsburg University

Syntax

```
CREATE [ OR REPLACE ] TRIGGER <Name>
    { BEFORE | AFTER | INSTEAD OF }
    <Event>
    [ WHEN (<Bedingung>) ]
    { <PL/SQL-Block> | CALL <Prozedurname> } ;
```

- **<Name>**: Name des Triggers
- Zeitpunkt der Ausführung
  - **BEFORE: vor der auslösenden Aktion (nicht auf Views). Kann nur die neuen Zeilenwerte ändern**
  - **AFTER: nach der auslösenden Aktion (nicht auf Views). Kann weder alte noch neue Zeilenwerte verändern**
  - **INSTEAD OF: anstelle der auslösenden Aktion (nur auf Views). Kann weder alte noch neue Zeilenwerte ändern**
- **<Event>**: Event, das den Trigger auslöst
  - DML: Events durch die Manipulation von Daten
  - DDL
  - Events, die die Datenbank betreffen
- **WHEN (<Bedingung>)**: Trigger wird nur dann ausgeführt, wenn <Bedingung> zutrifft. Dabei kann die Bedingung alte und neue Zeilenwerte beinhalten.

Trigger

- **DML-Event**

```
DELETE | INSERT | UPDATE [ OF <Spalte>, ... , <Spalte> ]
[ OR DELETE | INSERT | UPDATE [ OF <Spalte>, ... , <Spalte>] * ]

ON
  {
    <Tabelle>
    |
    [ NESTED TABLE <NestedTable-Spalte> OF ] <View>
  }

[ REFERENCING
  [ OLD [ AS ] <Name> ]
  [ NEW [ AS ] <Name> ]
  [ PARENT [ AS ] <Eltern> ]
]
[ FOR EACH ROW ]
```

Parameter

- **`DELETE, INSERT, UPDATE:`**
  - Gibt an, bei welcher Art von DML-Event der Trigger ausgelöst wird.
  - Es sind mehrere Angaben möglich, die durch **`OR`** verbunden werden.
  - Update-Trigger können auf spezielle Spalten definiert werden, von Änderungen in anderen Spalten werden sie nicht ausgelöst.
  - `UPDATE` und `INSTEAD OF` schließen sich gegenseitig aus.
- Innerhalb eines Triggers sind die parameterlosen Booleschen Funktionen
  - `DELETING`, `INSERTING` und `UPDATING`
    definiert. Sie geben an, durch welche Art von Event der Trigger ausgelöst wurde.

- Mit **`ON`** `<Tabelle>` bzw. **`ON`** `<View>` wird angegeben, auf welche Tabelle bzw. welchen View sich der Trigger bezieht.
- Durch die Option **`NESTED TABLE`** und die Angabe der Nested-Table-Spalte kann der Trigger auch auf Events einer Nested Table eines Views reagieren.

Trigger

- **`REFERENCING:`**
  - Mit diesem Teil der Klausel kann festgelegt werden, unter welchen Namen in der **WHEN**-Klausel und im PL/SQL-Block auf die alten (**OLD**) und neuen (**NEW**) Zeilenwerte und – bei Nested Tables – auf die Zeilenwerte der Parent-Tabelle (**PARENT**) zugegriffen werden kann.
  - Mindestens eine Angabe der drei Optionen muss vorhanden sein, bei mehreren Angaben spielt die Reihenfolge keine Rolle.
- **`FOR EACH ROW:`**
  - Der Trigger wird für jede Zeile ausgeführt, die geändert wird.
  - Ohne diese Angabe wird der Trigger nur einmal beim Aufruf des auslösenden Statements ausgelöst - wenn die optionale `WHEN`-Bedingung erfüllt wird.
  - `INSTEAD OF`-Trigger werden stets für jede Zeile ausgeführt
- **`Trigger-Kategorien:`**
  - Trigger auf **Anweisungsebene** enthalten **nicht** die Klausel **FOR EACH ROW.**
    - Trigger wird durch das auslösende Ereignis nur einmal ausgelöst
    - Trigger hat **keinen Zugang zu den Spaltenwerten** der einzelnen vom Trigger betroffenen Zeilen
  - Trigger auf **Zeilenebene** enthalten die Klausel **FOR EACH ROW**
    - Trigger auf Zeilenebene wird für jede vom Trigger betroffene Zeile ausgelöst
    - Trigger kann auf die Ausgangswerte und neuen Werte der Spalten zugreifen

Trigger

- **DDL-Event**

```
<DDL-Event> [ OR <DDL-Event> OR ... ]

ON { [ <Schema> . ] SCHEMA
     | DATABASE
   }
```

Es wird ein Event mit dem entsprechenden Namen ausgelöst, wenn einer der folgenden SQL-Befehle verwendet wird:

```
ALTER (nicht bei ALTER DATABASE)
ANALYZE
ASSOCIATE STATISTICS
AUDIT
COMMENT
CREATE (nicht bei CREATE DATABASE und CREATE CONTROLFILE)
DISASSOCIATE STATISTICS
DROP
GRANT
NOAUDIT
RENAME
REVOKE
TRUNCATE
DDL: Wird bei jedem spezifischen DDL-Event ausgelöst.
```

- **Datenbank-Event**

```
<Datenbank-Event> [ OR <Datenbank-Event> OR ... ]

ON { [ <Schema> . ] SCHEMA
     | DATABASE
}
```

Mögliche Datenbank-Events:

`SERVERERROR`	Fehler auf dem Server (nur `AFTER`)
`LOGON`	beliebiger Login hat stattgefunden (nur `AFTER`)
`LOGOFF`	beliebiger Logoff hat stattgefunden (nur `BEFORE`)
`STARTUP`	DB-Server gestartet (nur `AFTER`) und zus. mit `DATABASE`)
`SHUTDOWN`	DB-Server wird beendet (nur `BEFORE` und zus. mit `DATABASE`)
`SUSPEND`	eine Transaktion wird unterbrochen (nur `AFTER`)
`DB ROLE CHANGE`	Wechsel der aktiven Rolle in einer *Data Guard*-Konfiguration (nur `AFTER`)

Trigger - Beispiel

Zwei Tabellen

```
TAB1 (ID1 INTEGER, Text1 VARCHAR(200))
TAB2 (ID2 INTEGER, Text2 VARCHAR(200))
```

Wann immer nun ein Datensatz mit ID1 > 200 in TAB1 eingefügt wird, soll automatisch davon eine Kopie in TAB2 abgelegt werden.

```
CREATE OR REPLACE TRIGGER duplikate
  AFTER INSERT ON tab1
  FOR EACH ROW
  WHEN (new.ID1 > 200)
    /* new !!!, PL/SQL Code*/
  BEGIN
    INSERT INTO tab2 VALUES (:new.ID1,:new.Text1);  -- :new
  END;
```

Bei Triggern gibt es immer zwei Variablen, die im PL/SQL-Code verwendet werden können:

- **:new** die *neue* Zeile der Tabelle (bei **INSERT** und **UPDATE**).
- **:old** die *alte* Zeile der Tabelle (bei **DELETE** und **UPDATE**).

Die Doppelpunkte vor den Namen zeigen an, dass die Variablen von ”außen” stammen.

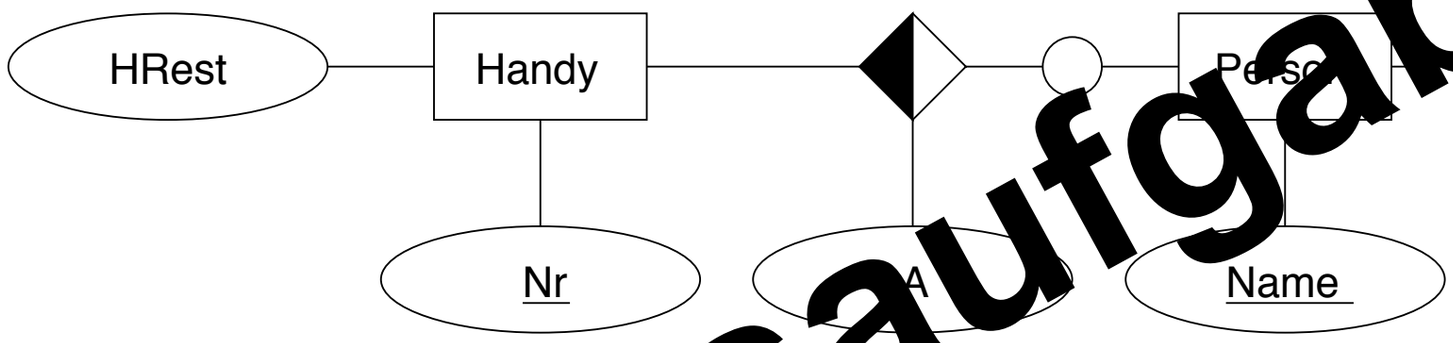
Relationships: n:1

eine Teilnahme optional (1)

Universität Augsburg University

Folie 25 aus Kapitel 4

Manche Handys haben keinen Besitzer,
aber alle Personen haben mindestens ein Handy



Übungsaufgabe

```
CREATE TABLE Person (
 Name   ... PRIMARY KEY,
 PRest  ... NOT NULL
 CHECK ((SELECT COUNT(*)
         FROM R
         WHERE PName = Name) > 0)
);

CREATE TABLE Handy (
 Nr     ... PRIMARY KEY,
 HRest  ... NOT NULL
);
```

```
CREATE TABLE R (
 HNr    ... PRIMARY KEY REFERENCES Handy(Nr),
 PName  ... NOT NULL REFERENCES Person(Name),
 A      ... NOT NULL
);
```

Anmerkung: Kaum eine DB beherrscht Subqueries in CHECK-Klauseln.

Wie kann dieses Problem gelöst werden?
Antwort: Trigger

Mögliche Klausuraufgaben