Basisskriptum

ORACLE

PL/SQL

"Procedural Language extensions to SQL"

v 0.3

- Mag. Johannes Tumfart -

Schuljahr 2002/2003

Änderungen

Version	Datum	Änderungsbeschreibung
0.1	-	Ersterstellung im Schuljahr 2002/2003
0.2	Feb 2003	Übernahme des Skripts von Dr. Thomas Stütz (Vielen Dank!)
0.3	Feb. 2003	Einarbeitung Einführung in PLSQL (FAW)

Inhalt

1 Einleitung

Normalerweise werden SQL Statements sequentiell ausgeführt. Nach der Ausführung eines Statements wird vom DBMS entweder das Ergebnis oder ein SQLCODE zurückgegeben.

Durch PL/SQL (Procedural Language extension to SQL) ist es möglich SQL Statements zusammenzufassen und zu einem späteren Zeitpunkt als ‚Paket' auszuführen.

Ein solches Paket enthält die bereits bekannten SQL Statements, darüber hinaus aber auch Sprachelemente aus einer 3GL – wie Schleifen, Verzweigungen, Prozeduren und Funktionen. Es wird also SQL um eine prozedurale Komponente erweitert.

1.1 Warum PL/SQL

- PL-SQL erhielt die Mächtigkeit von SQL + Kontrollkonstrukten+ Ausnahmebehandlung (Fehlerbehandlung ähnlich ADA) + Cursor.
- Verlagerung von Code aus der Anwendung in die Datenbank (Business Rules & Trigger werden in PL-SQL implementiert) mittels gespeicherten Prozeduren (Stored Procedures)
- SQL und PL-SQL Code wird vorcompiliert und im Datenbankcache gehalten.
- Erhöhung der Effizienz der Softwarentwicklung im Datenbankbereich durch Annäherung von SQL an konventionelle prozedurale Programmiersprachen
- PL-SQL Prozeduren können in der Datenbank abgelegt werden und von sämtlichen Benutzern mit entsprechenden Rechten ausgeführt werden.
- Zusammenfassen von Prozeduren zu Modulen (PACKAGES) ist möglich.

Beispiele

1. Transaktion :

```
DECLARE
   qty_on_hand  NUMBER(5);   -- lokale Variable
BEGIN
   SELECT quantity INTO qty_on_hand FROM inventory
      WHERE product = 'TENNIS RACKET'
      FOR UPDATE OF quantity;

   -- Verzweigung
   IF qty_on_hand > 0 THEN  -- check quantity
         UPDATE inventory SET quantity = quantity - 1
           WHERE product = 'TENNIS RACKET';
         INSERT INTO purchase_record
           VALUES ('Tennis racket purchased', SYSDATE);
   ELSE
         INSERT INTO purchase_record
            VALUES ('Out of tennis rackets', SYSDATE);
   END IF;
   COMMIT;
END;
/
```

Verkauf von Tennis-Rackets wird nur dann zugelassen, wenn genügende Anzahl vorrätig ist.

2. Zählergesteuerte Schleife

```
DECLARE
    x    NUMBER := 100;   -- lokale initialisierte Variable
 BEGIN
    FOR i IN 1..10 LOOP
       IF MOD(i,2) = 0 THEN
          INSERT INTO temp VALUES (i, x, 'i is even');
       ELSE
         INSERT INTO temp VALUES (i, x, 'i is odd');
       END IF;
       x := x + 100;   -- Laufvar. selbst darf nicht veränd. werden
    END LOOP;
 COMMIT;
 END;
/
```

Hier werden in die Tabelle TEMP 10 Datensätze eingefügt. Die Spalte NUM_COL1 wird mit Werten von 1 bis 10 belegt. Die Spalte NUM_COL2 mit numerischen Werten von 100 beginnend in 100er Schritten. Wenn in NUM_COL1 eine gerade Zahl steht, dann wird in der Spalte CHAR_COL der Text mit gerader Zahl, sonst mit ungerader Zahl eingetragen.

Das Einfügen der Datensätze erfolgt mit einem anonymen PL/SQL Block, der zur Verarbeitung vom SQL*Plus - Prompt aus aufgerufen wird. Abarbeitung mit Kommando RUN oder /.

PL/SQL ist eine Alternative zur Benutzung "gewöhnlicher" Programmiersprachen (dem Wirt), in der die benötigten SQL-Routinen eingebettet werden (C, Java, Visual Basic, Cobol)

Prior to 1991 the only way to use procedural construct with SQL was to use PRO*C. This is where Oracle SQL statements were embedded into C code. The C code was precompiled to convert the SQL statements into library calls.

In 1991 PL/SQL 1.0 was released with Oracle Version 6.0. It was very limited in its capabilities.

PL/SQL Version 2.0 was released with Oracle Version 7.0 This was a major upgrade. It had support for stored packages, procedures, functions, PL/SQL tables, programmer defined records and package extensions.

PL/SQL Version 2.1 was released with Oracle Version 7.1. This enabled the use of stored functions within SQL statements and the creation of dynamic SQL by using the DBMS_SQL package. It was also possible to execute DDL statements from with PL/SQL programs.

PL/SQL Version 2.2 was released with Oracle Version 7.2. This implements a binary wrapped for PL/SQL programs to protect the code from prying eyes. It was also possible to schedule database jobs with the DBMS_JOB package.

PL/SQL Version 2.3 was released with Oracle Version 7.3. This version enhanced the capabilities of the PL/SQL tables and added file I/O functionality.

PL/SQL Version 8.0 was released with Oracle Version 8.0. This version supports the enhancements of Oracle 8, including Large Objects, object oriented design, nested tables and Oracle advanced queuing.

PL/SQL Version 9.0.1 was released with Oracle Version 9i. This enabled the use of Case statements and expressions, New Date/Time Types, Better Integration for LOB Datatypes and the MERGE Statement.

Tab. 1: The History of PL/SPLSQL

2 Architektur von PL/SQL

PL/SQL ist eine Komponente von Oracle, die PL/SQL-Blöcke und -Unterprogramme ausführt.

Zur Ausführung von PL/SQL gibt es die PL/SQL Engine, die eine eigene Komponente von vielen Oracle Produkten darstellt (Oracle Server, Forms, Reports, Menu,...).

PL/SQL ist also kein eigenständiges Produkt im Sinn von Designer, Developer, Report, etc.

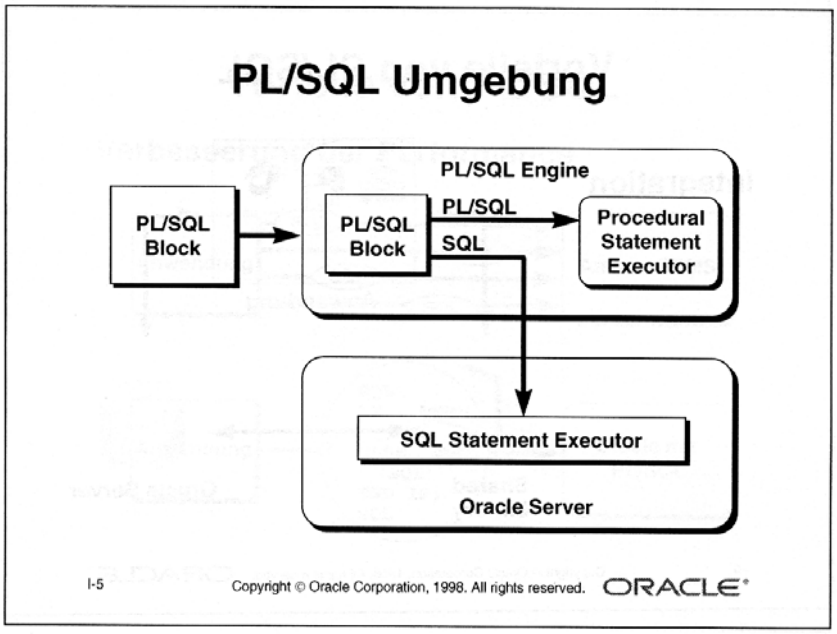


Abb. 1: PL/SQL - Umgebung

2.1 Blockstruktur

Ein PL/SQL Block hat 3 Teile: Deklarationsteil, Ausführungsteil und Exception-Handler. Nur der Ausführungsteil ist zwingend erforderlich.

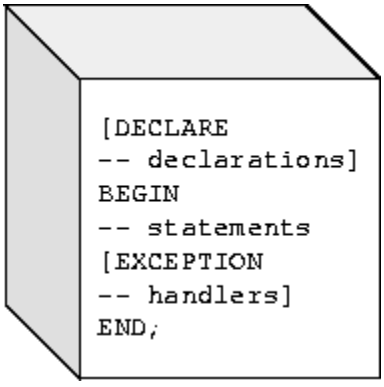


Abb. 2: PL/SQL-Blockstruktur

Sub-Blocks können im Ausführungsteil und Exception-Handler eines PL/SQL Blocks oder Unterprogramms, aber nicht im Deklarationsteil geschachtelt werden. Man kann aber lokale Unterprogramme im Deklarations-

teil eines jeden Blocks definieren. Aufgerufen werden diese aber nur aus dem Block, in dem sie definiert sind.

2.2 Arten von Blöcken

- Anonymer Block (unnamed block)

ist ein PL/SQL-Block, der in einer Applikation aufscheint, aber nicht benannt ist. (Für die Abarbeitung in SQL*Plus üblicherweise als SQL-Skript abgespeichert)

- Lokale Prozedur / Funktion

ist ein benannter PL/SQL-Sub-Block, der in einer Applikation deklariert und aufgerufen wird.

- Stored Procedure / Function

ist ein kompilierter PL/SQL Block, der in der Datenbank gespeichert und mit seinen Namen aus einer Applikation heraus aufgerufen wird.

Wenn eine Applikation einen solchen Block aufruft, lädt Oracle den kompilierten Code in die System Global Area. Die Komponenten PL/SQL Engine und SQL Statement Executor arbeiten zusammen, um die Statements innerhalb der Prozedur/Funktion zu bearbeiten.

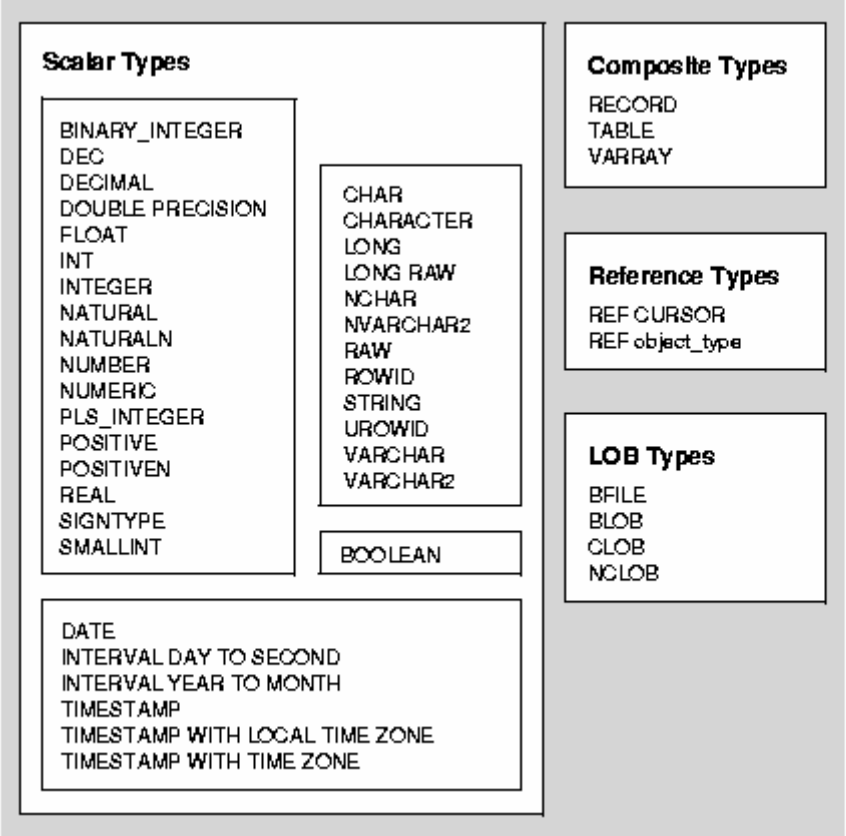
Hinweise:

- `BEGIN` und `END` sind optional, müssen aber verwendet werden, wenn
  1. eine `DECLARE`-Section verwendet wird.
  2. mehrere PL/SQL-Blöcke verwendet werden
- Mehrere PL/SQL-Blöcke ermöglichen die Verwendung unterschiedlicher Fehlerbehandlungsroutinen
- PL/SQL-Anweisungen werden mit einem Semikolon (";") abgeschlossen (Ausnahme: `BEGIN`, `DECLARE`).
- Kommentierung erfolgt mit:
  - "--" für den Rest der Zeile
  - "/*" bzw. "*/" für einen Block

3 Variable und Konstante

3.1 Datentypen

PL/SQL umfasst die Variablentypen: skalar, zusammengesetzt (composite), Referenz und LOB (large objects)



Tab. 2: Übersicht Datentypen (Quelle: Oracle Manual)

Datentypen sind z.B.

- number
- char
- varchar2
- date
- boolean
- rowid

Dabei ist zwischen Datentypen der Datenbank und Datentypen von PL/SQL zu unterscheiden. So wird bspw. der PL/SQL-Datentyp BOOLEAN von der ORACLE Datenbank nicht unterstützt.

3.2 Deklaration

Im Deklarationsteil werden Datentypdeklarationen der verwendeten Variablen und Konstanten durchgeführt, explizite Cursor definiert, um mengenorientierte 'select' Operationen durchzuführen und benutzerdefinierte Fehler bzw. Ausnahmen anzugeben.

Syntax:

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

Beispiele:

```
DECLARE
   birthdate       DATE;
   emp_count       NUMBER(3) := 0;
   part_no         NUMBER(4);
   in_stock        BOOLEAN;
```

Hinweise:

Im zweiten Beispiel wird die Variable initialisiert – man beachte den Zuweisungsoperator :=

Der Defaultwert einer Variablen ist NULL. Daher wird in den meisten Fällen eine Initialisierung unumgänglich sein.

Variablen können aus Ziffern, Buchstaben und Sonderzeichen gebildet werden. Es dürfen keine Operatoren oder Schlüsselwörter verwendet werden.

Korrekte Bezeichner	Inkorrekte Bezeichner	
dd	mine&yours	ungültiges &
X	ebit-amount	ist auch verboten
t2	on/off	/ ebenfalls
phone#	user id	Leerezeichen auch nicht
credit_limit	Reservierte Wörter dürfen nicht als Variablennamen benutzt werden	
LastName		
oracle$number		

Tab. 3: Variablenbezeichner

Die folgenden beiden Beispiele sind äquivalent.

```
birthdate         DATE;
birthdate         DATE := NULL;
```

Bei Konstantendeklarationen wird das Schlüsselwort CONSTANT dem Datentyp vorangestellt:

```
credit_limit      CONSTANT NUMBER(8,2) := 5000.00;
```

Variable und Konstante sind nicht case sensitive.

3.2.1 Was passiert wenn der Typ einer Spalte geändert wurde !

Dies soll den Programmierer nicht stören, denn in PL-SQL gibt es die Möglichkeit Variablen in Abhängigkeit des Typs einer bestimmten Spalte zu deklarieren.

%TYPE

Ein Problem bei PL/SQL Programmen währe, wenn man bei einer bereits vorhandenen Datenbank einen Datentyp, z.B. einen String ändert (Länge, etc.). In diesem Fall müsste man das gesamte Programm umschreiben. Zu diesem Zweck gibt es das %TYPE Attribut.

Bsp.:

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

Die Variablen v_name und n_name nehmen automatisch den Datentyp von den Spalten „e_vname“ und „e_nname“ der Tabelle „EMP“ an. Die Variable „tmpstr“ nimmt den Datentyp von v_name an. Sofern nicht nicht zu viel an der Tabelle EMP ändert (z.B. Stringlänge) muss das Programm nicht umgeschrieben werden.

Es besteht aber auch die Möglichkeit Variablen in Abhängigkeit des Types einer anderen Variable zu deklarieren.

```
declare
nr                number(5) not null:=0;
nr2               nr%type
```

%ROWTYPE

Dieses Attribut wird verwendet, wenn man eine Struktur aufbauen will, die den selben Aufbau besitzen soll, wie ein Satz aus einer Tabelle oder View, welche durch einen Cursor angesprochen wird bzw. den selben Aufbau, den ein SELECT – Statement liefert, das in einem Cursor deklariert wird.

Bsp.:

```
DECLARE
  dept_row dept%ROWTYPE; --Normale Struktur auf DEPT bezogen
  cur_row c_dept%ROWTYPE;     --Struktur auf die Rückgabewerte des Cursors
  c_dept bezogen
                         --Der Cursor muss allerdings bereits definiert
  sein

BEGIN
....
  SELECT *
       INTO        dept_row
       FROM        dept
       WHERE deptno = 20;
...
END;
```

4 Ausdrücke

4.1 Operatoren

Folgende Operatoren können verwendet werden (in absteigender Prioritätenreihenfolge):

Operator	Operation	
**, NOT	exponentiation, logical negation	Exponentialoperator, logische Verneinung
+, –	identity, negation	Vorzeichen
*, /	multiplication, division	Multiplikation, Division
+, –, ||	addition, subtraction, concatenation	Addition, Subtraktion, Verkettung
=, !=, <, >, <=, >=,IS NULL, LIKE,BETWEEN,IN	comparison	Vergleich
AND	conjunction	Logisches UND
OR	inclusion	Logisches ODER

Tab. 4: Operatoren

Die nachfolgend angegebenen Funktionen sind nur eine (kleine) Auswahl der tatsächlich verfügbaren Funktionen.

4.2 Ausdrücke und Funktionen

	Number	Character	Conversion	Date	Obj Ref	Misc
SQLCODE SQLERRM	ABS ACOS ASIN ATAN ATAN2 **BITAND** CEIL COS COSH EXP FLOOR LN LOG MOD POWER ROUND SIGN SIN SINH SQRT TAN TANH TRUNC	ASCII CHR CONCAT INITCAP INSTR INSTRB LENGTH LENGTHB LOWER LPAD LTRIM NLS_INITCAP NLS_LOWER NLSSORT NLS_UPPER REPLACE RPAD RTRIM SOUNDEX SUBSTR SUBSTRB TRANSLATE TRIM UPPER	CHARTOROWID CONVERT HEXTORAW RAWTOHEX ROWIDTOCHAR **TO_BLOB** TO_CHAR **TO_CLOB** TO_DATE TO_MULTI_BYTE **TO_NCLOB** TO_NUMBER TO_SINGLE_BYTE	ADD_MONTHS **CURRENT_DATE** **CURRENT_TIMESTAMP** **DBTIMEZONE** **EXTRACT** **FROM_TZ** LAST_DAY **LOCALTIMESTAMP** MONTHS_BETWEEN NEW_TIME NEXT_DAY **NUMTODSINTERVAL** **NUMTOYMINTERVAL** ROUND **SESSIONTIMEZONE** SYSDATE **SYSTIMESTAMP** **TO_DSINTERVAL** **TO_TIMESTAMP** **TO_TIMESTAMP_LTZ** **TO_TIMESTAMP_TZ** **TO_YMINTERVAL** **TZ_OFFSET** TRUNC	DEREF REF VALUE **TREAT**	BFILENAME DECODE DUMP EMPTY_BLOB EMPTY_CLOB GREATEST LEAST NLS_CHARSET_DECL_LEN NLS_CHARSET_ID NLS_CHARSET_NAME NVL SYS_CONTEXT SYS_GUID UID USER USERENV VSIZE

Tab. 5: Built-In Functions

4.2.1 Logische Ausdrücke

- Logische Operatoren. AND, OR, NOT
- Vergleichsoperatoren =,!=,<=,=>,<,>(<>, ^=)
  IS [NOT] NULL, [NOT] LIKE, [NOT} BETWEEN… AND …, [NOT] IN

4.2.2 Numerische Ausdrücke

- **Operatoren**: +,-,*,/,** (**…Exponential z.B.: `5E3 = 5 10**3 = 5 1000 = 5000`)
- **Vorzeichen**: +,-

Numerische Funktionen:

- <number> := **ABS**(<number>)
- <numbQuotient> := **MOD**(<numbDividend>, <numbDivisor>)
  z.B. `MOD(13, 4)` → 1
- <numbPotenz> := **POWER**(<numbGrundzahl>, <numbHochzahl>)
  z.B. `POWER(5,2)` → $5^2 = 25$
- <number> := **ROUND**(<number> [, <numbNachkommastellen>]), Runden
- <number> := **TRUNC**(<number> [, <numbNachkommastellen>]), Abschneiden
- <number> := **SQRT**(<numbRadikant>), Quadratwurzel

4.2.3 String-Ausdrücke

Konkatenationsoperator: ||

Bsp.: 'otto' || 'kar' → 'ottokar'

Stringfunktionen:

- <string> := **SUBSTR**(<string>, <intAnfangsposition>, <intSchnittlänge>)
- <number> := **INSTR**(<string>, <suchstring>, <intBeginnSuche>)
- <string> := **UPPER**(<string>)
- <string> := **LOWER**(<string>)
- <number> := **LENGTH**(<string>)
- <string> := **INITCAP**(<string>)

4.2.4 Datums-Ausdrücke

Datumdifferenzen

<datum> +/- <wert> → <datum>

<datum> +/- <datum> → <wert>

```
SELECT sysdate-10 from dual;

SYSDATE-1
---------
05-OCT-01

SELECT sysdate - TO_DATE('05-10-2001','DD-MM-YYYY')
       FROM dual;

SYSDATE-TO_DATE('05-10-2001','DD-MM-YYYY')
------------------------------------------
                                10.3483449
```

Datumsfunktionen:

- <date> := **SYSDATE**

- <date> := **ADD_MONTHS** (<datum>, <intAnzMonate>)
  Bsp.: `ADD_MONTHS(sysdate, 12)` = "heute in einem Jahr"
- <number> := **MONTHS_BETWEEN**(<date1>, <date2>) errechnet die zwischen den beiden Datumswerten liegende Anzahl von Monaten. Falls die beiden Daten die gleiche Tageszahl (z.B. 02) oder den jeweiligen Monatsletzten (z.B. 31.05 und 30.04) beinhalten wird einen Ganzzahl zurückgegeben, ansonsten wird der in der Differenz enthaltene Monatsteil auf Basis von 31 Tagen umgerechnet.

4.3 Zuweisungen

4.3.1 Möglichkeiten

- varname := <ausdruck>;
- SELECT ... INTO varname1, varname2 FROM …
- FETCH cursorname INTO varname1 (→ Cursors)

Beispiele :

- Zuweisungsoperator :=

```
tax := price * tax_rate;
bonus := current_salary * 0.10;
amount := TO_NUMBER(SUBSTR('750 dollars', 1, 3));
valid := FALSE;   -- Ausgabe des Wertes FALSE ist nicht möglich!
```

- Zuweisung von Datenbankwerten an Variablen mittels select oder fetch

```
SELECT sal * 0.10 INTO bonus FROM emp WHERE empno = emp_id;
```

4.3.2 Typenprüfung

Zugewiesener Wert muss den gleichen oder kompatiblem Typ haben.

Implizite Konvertierung:

- char → number (Randbedingungen)
- char → date (Randbedingungen)
- number → char
- date → char

Kommentar aus ORACLE-Dokumentation:

> **Implicit versus Explicit Conversion**
>
> Generally, to rely on implicit datatype conversions is a poor programming practice because they can hamper performance and might change from one software release to the next. Also, implicit conversions are context sensitive and therefore not always predictable. Instead, use datatype conversion functions. That way, your applications will be more reliable and easier to maintain.

Explizite Konvertierung

TO_CHAR, TO_DATE, TO_NUMBER

- <date> := **TO_DATE**(<string>,<formatstring>)
- <string> := **TO_CHAR**(<date>, <formatstring>), oder
  <string> := **TO_CHAR**(<number>, <formatstring>)
- <number> := **TO_NUMBER**(<string>,<formatstring>)

FormatString	Bemerkung
DD, Dy, Day, DY, DAY	Tage
MM, Mon, Month, MON, MONTH	Monat: MON (3-stellige Kodierung, zB JAN, FEB) MONTH (in englisch geschriebene Monatsnamen)
YY, YYYY	Jahr (2 oder 4-stellig)
HH, HH12, HH24	Stunden (12 oder 24 Stunden)
MI	Minute
SS	Sekunden
0	Platzhalter für eine Ziffer
9	Ebenfalls Platzhalter für eine Ziffer, wobei führende Nullen unterdrückt werden
.	Platzhalter für den Dezimalpunkt
D	Platzhalter für das Dezimaltrennzeichen entsprechend der nld-Einstellungen
MI	Platzhalter für das Vorzeichen
G	Platzhalter für das Tausendertrennzeichen entsprechen der nls-Einstellung

Tab. 6: Übersicht wichtiger Formatstrings

4.3.3 Zuweisung mit Records

Beispiel:

```
DECLARE
  deptrec1 dept%ROWTYPE;
  deptrec2 dept%ROWTYPE;
BEGIN
  SELECT * INTO deptrec1 FROM dept WHERE deptno=10;
  deptrec2:=deptrec1;
  deptrec2.loc := 'WIEN';
  dbms_output.put_line(deptrec2.deptno);
  dbms_output.put_line(deptrec2.dname);
  dbms_output.put_line(deptrec2.loc);
  deptrec2 := (50,'ABT','WIEN');      --FEHLER
END;
```

Hinweise:

- Das SELECT-Statement darf nur einen Datensatz zurückgeben
  (Fehlermeldung: `ORA-01422: exact fetch returns more than requested number of rows`)
- Fehler in Zeile 11, da man einem Record nicht eine Liste von Werten zuweisen kann.
- Damit dbms_output funktioniert zuerst Umgebungsvariable setzen:
  `SET SERVEROUTPUT ON;`

5 Kontrollstrukturen

5.1 Einfache Verzweigung (IF-THEN)

Syntax:

```
IF <condition> THEN
   <sequence_of_statements>
END IF
```

Wie gewohnt wird <sequence_of_statements> nur ausgeführt, wenn die Auswertung der Bedingung TRUE ergibt. Bei FALSE, aber auch bei NULL, wird keine Aktion durchgeführt.

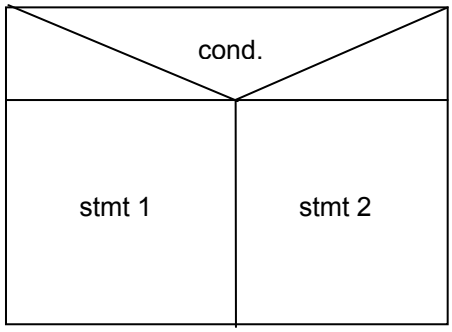
Beispiel:

```
IF sales > quota THEN
   bonus := compute_bonus(empid);    ---Aufruf einer Funktion
   UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
END IF;
```

5.2 Alternative Verzweigung (IF-THEN-ELSE)

Syntax:

```
IF <condition> THEN
   <sequence_of_statements1>
ELSE
   <sequence_of_statements2>
END IF
```



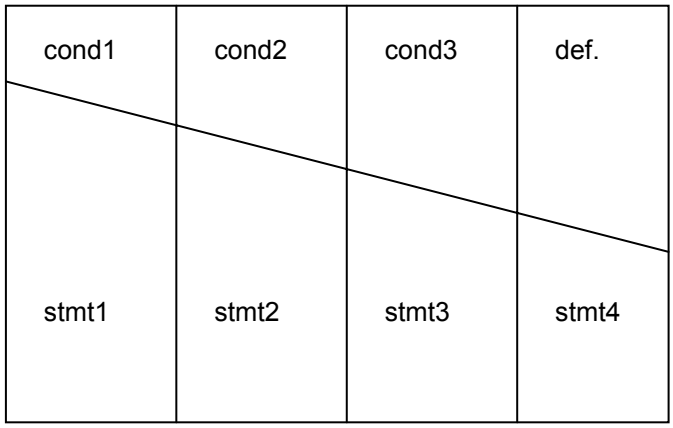
Beispiel:

```
IF trans_type = 'CR' THEN
   UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
   IF new_balance >= minimum_balance THEN
      UPDATE accounts SET balance = balance - debit WHERE ...
   ELSE
      RAISE insufficient_funds;     -- benutzerdef. Exception aufrufen
   END IF;
END IF;
```

5.3 Mehrfache Verzweigung (IF-THEN-ELSIF)

Syntax:

```
IF <condition1> THEN
   <sequence_of_statements1>
ELSIF <condition2> THEN
   <sequence_of_statements2>
ELSIF <condition3> THEN
   <sequence_of_statements3>
ELSE
   <sequence_of_statements4>
END IF
```



entspricht CASE-Statement

Beispiel:

```
IF sales > 50000 THEN
   bonus := 1500;
ELSIF sales > 35000 THEN
   bonus := 500;
ELSE
   bonus := 100;
END IF;

INSERT INTO payroll VALUES (emp_id, bonus, ...);
```

Die beiden folgenden Konstruktionen sind logisch äquivalent:

```
IF condition1 THEN
statement1;
ELSE
IF condition2 THEN
statement2;
ELSE
  IF condition3 THEN
  statement3;
  END IF;
END IF;
END IF;
```

```
IF condition1 THEN
statement1;
ELSIF condition2 THEN
statement2;
ELSIF condition3 THEN
statement3;
END IF;
```

5.4 Mehrfache Verzweigung (CASE)

5.4.1 CASE Statement als Verzweigung

Ein CASE-Statement (verfügbar ab Vers. 9) ist – verglichen mit einem IF-Statement – lesbarer und wird auch effizienter ausgeführt. Anstelle eines bool'schen Ausdrucks verwendet das CASE-Statement einen Selektor.

```
[<<label_name>>]
CASE selector
   WHEN expression1 THEN sequence_of_statements1;
   WHEN expression2 THEN sequence_of_statements2;
   ...
   WHEN expressionN THEN sequence_of_statementsN;
  [ELSE sequence_of_statementsN+1;]
END CASE [label_name];
```

```
IF grade = 'A' THEN
   dbms_output.put_line('Excellent');
ELSIF grade = 'B' THEN
   dbms_output.put_line('Very Good');
ELSIF grade = 'C' THEN
   dbms_output.put_line('Good');
ELSIF grade = 'D' THEN
   dbms_output. put_line('Fair');
ELSIF grade = 'F' THEN
   dbms_output.put_line('Poor');
ELSE
   dbms_output.put_line('No such grade');
END IF;
```

Wird keine expliziter ELSE-Zweig wie im obigen Bsp. angegeben, so fügt PL/SQL implizit folgende Klausel hinzu.

```
ELSE RAISE CASE_NOT_FOUND;
```

5.4.2 CASE-Statement ohne Selektor (Searched CASE)

Ein CASE-Statement kann auch wie folgt angewendet werden.

```
CASE
   WHEN grade = 'A' THEN dbms_output.put_line('Excellent');
   WHEN grade = 'B' THEN dbms_output.put_line('Very Good');
   WHEN grade = 'C' THEN dbms_output.put_line('Good');
   WHEN grade = 'D' THEN dbms_output.put_line('Fair');
   WHEN grade = 'F' THEN dbms_output.put_line('Poor');
   ELSE dbms_output.put_line('No such grade');
END CASE;
```

5.4.3 CASE-Statement als Ausdruck

Ein CASE-Konstrukt kann auch Bestandteil eines Ausdrucks sein.

```
DECLARE
   grade CHAR(1) := 'B';
   appraisal VARCHAR2(20);
BEGIN
   appraisal :=
      CASE grade
         WHEN 'A' THEN 'Excellent'
         WHEN 'B' THEN 'Very Good'
         WHEN 'C' THEN 'Good'
         WHEN 'D' THEN 'Fair'
         WHEN 'F' THEN 'Poor'
         ELSE 'No such grade'
      END;
END;
```

5.5 Schleifen

5.5.1 LOOP – Anweisung

Die einfachste Form einer Schleife (**Endlosschleife**):

Syntax:

```
LOOP
   <sequence_of_statements>
END LOOP
```

Die unbedingte Beendigung einer Schleife erfolgt mit dem EXIT – Statement. Es wird mit dem ersten Statement nach der END LOOP Anweisung fortgesetzt.

```
LOOP
   ...
   IF credit_rating < 3 THEN
      ...
      EXIT; -- exit loop immediately
   END IF;
END LOOP;

-- control resumes here
```

Das EXIT Statement darf nur innerhalb einer Schleife vorkommen. Es ist nicht möglich einen PL/SQL Block mit EXIT zu beenden. Daher ist die folgende Sequenz nicht gültig:

```
BEGIN
   ...
   IF credit_rating < 3 THEN
      ...
      EXIT; -- illegal
   END IF;
END;
```

5.5.2 EXIT WHEN - Anweisung

Beendet eine Schleife bedingt und ersetzt dabei ein IF kombiniert mit einem EXIT.

Die folgenden beiden Sequenzen sind logisch gleichwertig:

```
IF count > 100 THEN
    EXIT;
END IF;
```

```
EXIT WHEN count > 100;
```

5.5.3 WHILE – Schleife

Wenn die Auswertung der Bedingung TRUE ergibt wird die Schleife ausgeführt:

Syntax:

```
[<<loop_name>>]
WHILE <condition> LOOP
   <sequence_of_statements>
END LOOP
```

Beispiel:

```
WHILE total <= 25000 LOOP
   ...
   SELECT sal INTO salary FROM emp WHERE ...
   total := total + salary;
END LOOP;
```

Hinweis:

- Der `loop_name` wird in doppelten "<<" bzw. ">>" eingeschlossen.
- Die Benennung von Schleifen dient vor allem dazu mit EXITs gezielt aus verschachtelten Schleifen herauszuspringen:

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

5.5.4 FOR Schleife

Syntax:

```
[<<loop_name>>]
FOR <counter> IN [REVERSE] <lower_bound>..<higher_bound> LOOP
   <sequence_of_statements>
END LOOP;
```

Beispiel:

```
FOR i IN 1..3 LOOP          -- assign the values 1,2,3 to i
   <sequence_of_statements> -- executes three times
END LOOP;

FOR i IN REVERSE 1..3 LOOP  -- assign the values 3,2,1 to i
   <sequence_of_statements> -- executes three times
END LOOP;
```

Der Schleifenzähler wird automatisch als Integer - Variable angenommen und braucht daher nicht explizit in der Declare Section deklariert zu werden. Er ist nur innerhalb der Schleife bekannt und kann daher nicht außerhalb der Schleife verwendet werden. Daher ist folgende Sequenz ungültig:

```
FOR ctr IN 1..10 LOOP
   ...
END LOOP;

sum := ctr - 1; -- illegal
```

Auf Schleifenzähler darf innerhalb der Schleife keine Zuweisung erfolgen, Verwendung als Quelle ist erlaubt :

```
FOR ctr IN 1..10 LOOP
...
ctr := ctr + 2;   -- illegal
```

```
z := z + 2*ctr;   -- legal
END LOOP;
```

5.5.5 Sprungbefehle und Labels

Nur im äußersten Notfall :

```
BEGIN
...
   GOTO insert_row;
   ...
   <<insert_row>> -- Labeldeklaration
   INSERT INTO emp VALUES ...

END;
```

Eine Label innerhalb eines Schleifen- oder Verzweigungsblocks darf nicht von außen angesprungen werden !

6 Lesen von der Datenbank

6.1 Lesen einer einzelnen Zeile

SELECT INTO - Statement (siehe 1.Beispiel)

```
SELECT quantity INTO qty_on_hand FROM inventory
      WHERE product = 'TENNIS RACKET';
```

Wenn dieses SELECT-Statement mehrere Zeilen zurückgibt → Error `ORA-01422 Too many rows` → Cursor muss verwendet werden.

6.2 Cursors

Ein Cursor dient der Verwaltung des Zugriffs auf einen Satz von Datenzeilen, der das Ergebnis einer SELECT-Anweisung ist. Hiebei kann der Satz keine, eine oder mehrere Zeilen umfassen.

Oracle verwendet Work Areas, um SQL Statements auszuführen und Verarbeitungsinformation abzuspeichern. Ein PL/SQL Konstrukt namens Cursor benennt eine Work Area und erlaubt auf deren gespeicherte Information zuzugreifen.

Es gibt 2 Arten von Cursors: *implizite* und *explizite*. PL/SQL deklariert implizit einen Cursor für alle SQL-DML-Statements, sowie für Queries, die nur eine Zeile (s.o.) zurückgeben. Für Queries, die mehr als eine Zeile zurückgeben, ist explizit ein Cursor zu deklarieren, um jede Zeilen für sich bearbeiten zu können.

Beispiel:

```
DECLARE
   CURSOR c1 IS
   SELECT empno,ename,job FROM emp WHERE deptno = 20;
```

Die Menge der bei einer solchen Query zurück gegebenen Zeilen heißt Ergebnismenge (result set). Wie das Bild zeigt, ist ein expliziter Cursor ein Pointer auf die aktuelle Zeile (current row) in der Ergebnismenge. Dadurch kann das Programm die gefetchten Zeilen einzeln behandeln.

Result Set				
	7369	SMITH	CLERK	
	7566	JONES	MANAGER	
cursor →	7788	SCOTT	ANALYST	Current Row
	7876	ADAMS	CLERK	
	7902	FORD	ANALYST	

Abb. 3: Result Set und Current Row eines Cursors

Die Bearbeitung von Ergebnismengen entspricht in etwa einer Datei-Verarbeitung . Beispielsweise öffnet ein C-Programm eine Datei, bearbeitet Records und schließt dann die Datei. Analog dazu öffnet ein PL/SQL Programm einen Cursor, bearbeitet die von einer Query zurückgegebenen Zeilen und schließt dann den Cursor. Genau wie ein File-Pointer die aktuelle Position in eine offenen Datei markiert, zeigt ein Cursor auf die aktuelle Position in einer Ergebnismenge.

Syntax:

```
DECLARE
...
CURSOR cursor_name [(parameter[, parameter]...)]
   [RETURN return_type] IS
                       SELECT ... FROM ...
                         [FOR UPDATE [OF col_name][NOWAIT]];
```

wobei zu beachten ist, dass der Rückgabetyp des Cursors nur ein Record oder eine Zeile einer Tabelle sein kann.

Man verwendet OPEN,FETCH und CLOSE Statements zur Cursor-Kontrolle.

- Das OPEN Statement führt die mit dem Cursor verbundene Query aus, identifiziert die Ergebnismenge und stellt den Cursor vor die erste Zeile.

```
DECLARE
   CURSOR c1 IS SELECT ename, job FROM emp WHERE sal < 3000;
   ...
BEGIN
   OPEN c1;
   ...
END;
```

- Das FETCH Statement setzt den Cursor auf die nächste Zeile und holt die aktuelle Zeile aus dem Puffer.

```
FETCH c1 INTO my_empno, my_ename, my_deptno;
```

- Nach Bearbeitung der letzten Zeile deaktiviert das CLOSE Statement den Cursor.

```
CLOSE c1;
```

- Übergeben von Parametern an den Cursor

```
DECLARE
   emp_name emp.ename%TYPE;
   salary   emp.sal%TYPE;
   CURSOR c1 (name VARCHAR2, salary NUMBER) IS SELECT ...
```

- wobei der Cursor mit folgenden Statements geöffnet werden kann:

```
OPEN c1(emp_name, 3000);
OPEN c1('ATTLEY', 1500);
OPEN c1(emp_name, salary);
```

- Verwenden der BULK COLLECT – Klausel: Hiermit werden sämtliche Zeilen der Ergebnismenge sofort übergeben. Beim nachfolgenden Beispiel werden sämtliche Zeilen in eine Collection von Records geladen.

```
DECLARE
   TYPE DeptRecTab IS TABLE OF dept%ROWTYPE;
   dept_recs DeptRecTab;
   CURSOR c1 IS
      SELECT deptno, dname, loc FROM dept WHERE deptno > 10;
BEGIN
   OPEN c1;
   FETCH c1 BULK COLLECT INTO dept_recs;
END;
```

6.3 Attribute

Bei Verwendung eines Cursors werden einige zusätzliche Sprachelemente benötigt:

6.3.1 Cursorattribute

6.3.1.1 %FOUND %NOTFOUND (boolean)

Nach Eröffnung eines Cursors hat %FOUND den Wert NULL. Werden Zeilen gefunden, so hat es den Wert TRUE. Wenn das letzte FETCH keine Zeile zurückgibt, hat es den Wert FALSE.

```
DECLARE
       CURSOR c1    IS SELECT ename, sal, hiredate FROM emp
                    WHERE deptno = 20;
       my_ename     emp.ename%TYPE;
       my_sal       emp.sal%TYPE;
       ...
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO my_ename, my_sal, my_hiredate;
      IF c1%FOUND THEN -- fetch succeeded
         ...
      ELSE -- fetch failed, so exit loop
         EXIT;
      END IF;
     ...
   END LOOP;
   CLOSE c1;
END;
```

Das logische Gegenteil ist %NOTFOUND

6.3.1.2 %ISOPEN

prüft, ob ein Cursor offen ist

```
IF c1%ISOPEN THEN -- cursor is open
   ...
ELSE -- cursor is closed, so open it
   OPEN c1;
END IF;
```

6.3.1.3 %ROWCOUNT

zählt die Rows, die aus einem Cursor gefetcht worden sind.

```
LOOP
   FETCH c1 INTO my_ename, my_deptno;

   IF c1%ROWCOUNT > 10 THEN
      ...
   END IF;
   ...
END LOOP;
```

6.3.1.4 Impliziter Cursor

Neben den expliziten Cursors (die von Benutzern deklariert werden) gibt es auch implizite. Bei DELETE, INSERT oder UPDATE erstellt Oracle einen impliziten Cursor zur Verarbeitung des Statements.

Dieser Cursor ist anonym. Auf die Cursorattribute kann aber mit dem Schlüsselwort SQL zugegriffen werden.

Daher können auch die Attribute SQL%FOUND, SQL%NOTFOUND, SQL%ROWCOUNT und SQL%ISOPEN verwendet werden.

6.3.2 Weitere Attribute (Wiederholung)

6.3.2.1 %TYPE

stellt den Datentyp einer Variablen oder Datenbankspalte zur Verfügung. Dieser kann anstelle einer Typvereinbarung im Rahmen einer Variablen-, Feld- oder Parameterdeklaration verwendet werden.

```
DECLARE
   player players.playerno%TYPE;
```

Hier wird die Variable Player definiert. Sie soll denselben Datentyp haben wie die Spalte playerno aus der Tabelle players.

```
balance            NUMBER(7,2);
minimum_balance    balance%TYPE := 10.00;
```

minmum_balance hat denselben Datentyp wie balance und wird zusätzlich initialisiert.

Beispiel:

Gesucht sind die 5 höchstbezahlten Angestellten. Diese sollen in eine Tabelle temp gespeichert werden.

```
DECLARE
   CURSOR c1 IS SELECT ename, empno, sal FROM emp
                ORDER BY sal DESC; -- Der Teuerste zuerst
   my_ename     emp.ename%TYPE;
   my_empno     emp.empno%TYPE;
   my_sal      emp.sal%TYPE;

BEGIN
   OPEN c1;
   FOR i IN 1..5 LOOP
      FETCH c1 INTO my_ename, my_empno, my_sal;
      EXIT WHEN c1%NOTFOUND;  -- weniger als 5 Angestellte
      INSERT INTO temp VALUES (my_sal, my_empno, my_ename);
      COMMIT;
   END LOOP;
   CLOSE c1;
END;
```

6.3.2.2 %ROWTYPE

Stellt den Datentyp eines Records zur Verfügung. Dieser Record kann eine gesamte Datenzeile einer Tabelle oder die SELECT-Liste eines Cursors beinhalten. Spalten bzw. SELECT-List-Members und korrespondierende Felder im Record haben denselben Namen und denselben Datentyp.

```
DECLARE
   emp_rec      emp%ROWTYPE;   -- Recordformat der Tabelle emp
   CURSOR c1    IS SELECT deptno,dname,loc FROM dept;
   dept_rec     c1%ROWTYPE;    -- Recordformat des Cursors c1

BEGIN
SELECT * INTO emp_rec FROM emp WHERE ...
IF (emp_rec.deptno = 20) AND (emp_rec.sal > 2000) THEN
...
END IF;
.....
```

6.4 Die Klausel FOR UPDATE

Dient zum Sperren von Zeilen vor einem UPDATE oder DELETE. Durch Hinzufügen der Klausel FOR UPDATE werden die Zeilen der Ergebnismenge gesperrt, sobald der Cursor geöffnet wird.

Syntax:

```
SELECT ...
FROM ...
FOR UPDATE [OF <spalten_referenz>] [NOWAIT]
```

Hinweis:

- Die Klausel NOWAIT gibt einen ORACLE-Fehler zurück, wenn die Zeilen von einer anderen Session gesperrt sind.
- FOR UPDATE ist die letzte Klausel eines SELECT-Statements, sogar nach einem ev. GROUP BY.
- Werden mehrere Tabellen abgefragt, können Zeilensperren durch die Klausel FOR UPDATE **OF** auf bestimmte Tabellen begrenzt werden.

6.5 Die Klausel CURRENT OF

Nachdem die Klausel FOR UPDATE bestimmte Zeilen gesperrt hat, kann die aktuelle Zeile eines expliziten Cursors mit der Klausel CURRENT OF referenziert werden.

Syntax:

```
... WHERE CURRENT OF <cursor_name>
```

Beispiel:

```
DECLARE
   CURSOR sal_cursor IS
      SELECT …
      FOR UPDATE;
BEGIN
   …
FOR emp_record IN sal_cursor LOOP
   UPDATE emp
   SET sal = emp_record.sal * 1.10
      WHERE CURRENT OF sal_cursor;
   END LOOP;
   COMMIT;
END;
```

7 Cursor-FOR-Schleifen

Die meisten Fälle, die einen expliziten Cursor erfordern, kann man mittels einer Cursor-FOR-Schleife anstelle von OPEN, FETCH und CLOSE Statements vereinfachen.

Eine Cursor-FOR-Schleife deklariert implizit ihren Schleifen-Index als Record , der eine von der Datenbank gefetchte Zeile repräsentiert.

Nach (automatischem) Öffnen des Cursors werden die Zeilen der Ergebnismenge bei jedem Schleifendurchlauf in die Komponenten des Records gefetcht .Nach der Bearbeitung aller Zeilen wird der Cursor wieder geschlossen.

Beispiele :

1. Aus der Tabelle Players sollen die Nummern aller Spieler, die zwischen 1950 und 1960 geboren worden sind in eine Hilfstabelle übertragen werden:

   Klassische Implementierung mit OPEN,FETCH und CLOSE :

```
DECLARE
   player     players.playerno%TYPE;
   CURSOR     play_curs IS SELECT playerno FROM players
              WHERE year_of_birth BETWEEN 1950 and 1960;
BEGIN
   OPEN play_curs;
      LOOP
      FETCH play_curs INTO player;
      EXIT WHEN play_curs%NOTFOUND;
      INSERT INTO help_table VALUES (player);
   END LOOP;
   CLOSE play_curs;
END;
```

   Implementierung mit Cursor-FOR-Schleife (play_rec implizit als Record deklariert) :

```
DECLARE
CURSOR      play_curs IS SELECT playerno FROM players
            WHERE year_of_birth BETWEEN 1950 and 1960;
BEGIN
   FOR play_rec IN play_curs LOOP
      INSERT INTO help_table VALUES (play_rec.playerno);
   END LOOP;
END;
```

   Zum Ansprechen einzelner Komponenten des Records verwendet man die dot notation, in der ein dot (.) als Selektor der Komponenten dient.

2. Fülle eine Summentabelle für die Auftragssummen der Kunden sum_table(custid,custsum) → zwingende Verwendung von Aliasen :

```
DECLARE
   CURSOR sum_curs IS    SELECT c.custid cid,sum(o.total) csum
                         FROM customer c,ord o
                         WHERE c.custid = o.custid
                         GROUP BY c.custid;
BEGIN
   FOR sum_curs_rec IN sum_curs LOOP
      INSERT INTO sum_table
      VALUES (sum_curs_rec.cid, sum_curs_rec.csum);
   END LOOP;
END;
```

8 Cursor-Variable

8.1 Was sind Cursor-Variablen ?

Wie ein Cursor zeigt eine Cursor- Variable auf die aktuelle Zeile in der Ergebnismenge einer mehrzeiligen Abfrage. Aber Cursor unterscheiden sich von Cursor-Variablen wie Konstanten von Variablen. Während ein Cursor statisch ist, ist eine Cursor-Variable dynamisch, da sie nicht an eine bestimmte Abfrage (query) gebunden ist. Im Unterschied zu einem Cursor, kann eine Cursor-Variable für jegliche Typ-kompatible Abfrage geöffnet werden.

Cursor-Variable sind echte PL/SQL-Variablen, welchen neue Werte zugewiesen und die an Unterprogramme als Parameter übergeben werden können. Dadurch wird eine größere Flexibilität erreicht und eine einfache Möglichkeit die Datengewinnung zu zentralisieren.

Cursor- Variablen sind wie Zeiger in C, sie halten die Speicheradresse eines Objekts anstatt das Objekt selbst. Ein Cursor ist statisch, eine Cursor-Variable ist dynamisch. In PL/SQL hat eine Cursor- Variable einen Datentyp REF X, wobei REF für Referenz und X für die Klasse des Objekts steht.

8.2 Verwendung von Cursor-Variablen

Zur Ausführung einer mehrzeiligen Abfrage öffnet der Oracle Server einen unbenannten Arbeitsbereich (unnamed work area) zur Speicherung von Verarbeitungsinformationen. Um auf die Information zuzugreifen wird entweder der Arbeitsbereich explizit benannt (durch Verwendung eines expliziten Cursors) oder eine Cursor-Variable verwendet, die auf den Arbeitsbereich zeigt. Während ein Cursor sich immer auf denselben Arbeitsbereich bezieht, kann sich eine Cursor- Variable auf unterschiedliche Arbeitsbereiche beziehen. Aus diesem Grunde sind Cursor und Cursor-Variable nicht austauschbar.

Primär wird eine Cursor- Variable verwendet, um Abfrage-Ergebnisse zwischen gespeicherten PL/SQL-Programmen und verschiedenen Clients zu transportieren. Keinem gehört die Ergebnismenge, die beteiligten Programme bzw. Clients teilen sich einfach einen Zeiger auf den Arbeitsbereich der Abfrage, der die Ergebnismenge enthält.

Eine Cursor- Variable kann auf der Client-Seite deklariert werden, auf der Serverseite kann sie geöffnet und aus ihr gelesen werden, und dann wird das Lesen aus ihr auf der Client-Seite fortgesetzt.

8.3 Arbeiten mit Cursor-Variablen

8.3.1 Schritt 1: Definieren einer Cursor-Variablen

Zum Anlegen einer Cursor-Variablen müssen Sie zuerst einen REF CURSOR-Typ deklarieren, dann deklarieren Sie eine Variable von diesem Typ.

```
TYPE <ref_type_name> IS REF CURSOR [RETURN <return_type>]
```

<ref_type_name>	…	Typ-Bezeichner für nachfolgende Deklaration
<return_type>	...	repräsentiert eine Zeile oder einen Record einer Datenbanktabelle.

Beispiel:

Es wird ein Rückgabetyp spezifiziert, der eine Zeile der Datenbanktabelle dept darstellt.

```
DECLARE
   TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
```

REF CURSOR-Typen können stark (restriktiv, restrictive) oder schwach (nicht restriktiv, nonrestrictive) sein. Ein starker REF CURSOR-Typ spezifiziert einen Rückgabetyp, eine schwache Definition tut dies nicht. PL/SQL läßt einen starken Typ nur mit typ-kompatiblen Abfragen verknüpfen, wohingegen ein schwacher Typ mit jeder Abfrage verknüpft werden kann. Daher neigen starke REF CURSOR-Typen weniger zu Fehlern, schwache sind dagegen flexibler.

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;  -- strong
   TYPE GenericCurTyp IS REF CURSOR;                 -- weak
```

8.3.2 Schritt 2: Öffnen einer Cursor-Variablen

Im Normalfall wird eine Cursor-Variable geöffnet, indem die Cursor-Variable an eine Stored Procedure als formaler Parameter übergeben.

Beispiel:

Eine Cursor-Variable generic_cv (schwacher REF CURSOR) wird für die gewählte Abfrage geöffnet.

```
PROCEDURE open_cv (generic_cv IN OUT GenericCurTyp,choice NUMBER) IS
BEGIN
   IF choice = 1 THEN
      OPEN generic_cv FOR SELECT * FROM emp;
   ELSIF choice = 2 THEN
      OPEN generic_cv FOR SELECT * FROM dept;
   ELSIF choice = 3 THEN
      OPEN generic_cv FOR SELECT * FROM salgrade;
   END IF;
END;
```

8.4 Einschränkungen bei Cursor-Variablen

- Cursor-Variablen können nicht in einem Package deklariert werden, da sie keinen beständigen Status haben.
- Aufrufe von Fern-Programmen zur Übergabe von Cursor-Variablen von einem Server zu einem anderen, können nicht verwendet werden.
- In eine mit einer Cursor-Variable verknüpfte Abfrage darf kein FOR UPDATE-Klausel eingebaut werden.
- Einer Cursor-Variablen darf nicht NULL zugewiesen werden.
- Cursor-Variablen dürfen nicht in dynamic SQL verwendet werden.

9 Unterprogramme (Stored Procedures und Stored Functions)

9.1 Einführung

Zur Modularisierung unseres PL/SQL-Codes stehen uns Funktionen, Prozeduren und Packages zur Verfügung:

- Eine **Prozedur** ist ein benannter SQL-Block, der eine oder mehr Anweisungen ausführt. Daten können durch die Parameter-Liste an die Prozedur übergeben bzw. von der Prozedur erhalten werden.
- Eine **Funktion** is ein benannter SQL-Block, der einen Einzelwert zurückgibt und kann wie ein PL/SQL-Ausdruck aufgerufen werden. Funktionen und Prozeduren sind ident strukturiert, bis auf die in der Funktion enthaltene RETURN-Klausel
- Ein **Package** ist eine benannte Sammlung von Prozeduren, Funktionen, Typen und Variablen.
- Anonymer oder **unbenannter PL/SQL-Block**

Prozeduren und Funktionen werden von Oracle als **Unterprogramme** (Subprograms**)** bezeichnet. Stored Subprograms liegen in kompilierter Form in der Datenbank vor.

Unterprogramme sind PL/SQL Blöcke mit folgenden wesentlichen Eigenschaften:

- Haben einen eindeutigen Namen
- Können Parameter verarbeiten
- Können im Data Dictionary gespeichert werden
- Können von verschiedenen Usern verwendet werden

```
-- Header
PROCEDURE award_bonus (emp_id NUMBER) IS
   -- optionaler Deklarationsteil (optional declarative part)
   bonus        REAL;
   comm_missing EXCEPTION;

-- Ausführungsteil (executable part)
BEGIN
   SELECT comm * 0.15 INTO bonus FROM emp WHERE empno = emp_id;
   IF bonus IS NULL THEN
      RAISE comm_missing;
   ELSE
      UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
   END IF;

-- otionaler Exception-Handler
EXCEPTION
   WHEN comm_missing THEN
      ...
END award_bonus;
```

9.2 Parameter

9.2.1 Deklaration

```
 Name  Modus  Datentyp
```

9.2.2 Parametermodi

IN (default)	übergibt Werte an eine Prozedur. (call by value)
OUT	gibt Werte von einer Prozedur zurück . (call by reference)
IN OUT	übergibt Werte an eine Prozedur, die in der Prozedur geändert werden können und anschließend wieder zurückgegeben werden . (call by reference)

Tab. 7: Parametermodi

9.2.3 Parametertypen

Möglichst nur Grundtypen verwenden : NUMBER,CHAR,VARCHAR2,DATE, ...

CHAR,VARCHAR2 (ohne Größenangabe) gelten auch für Strings.

Beispiele:

1. **Prozedurdeklaration** für die Ausgabe aller Angestellten eines Departments

```
PROCEDURE get_emp_names (dept_num IN NUMBER) IS
   CURSOR ec (dno NUMBER) IS  SELECT ename FROM emp
                              WHERE deptno = dno;
BEGIN
   FOR ec_rec IN ec LOOP
      DBMS_Output.Put_Line('EMPNAME : '||ec_rec.ename);
   END LOOP;
END;
```

2. **Lokale Prozedur** in anonymem Block : Alle faulen Salesmen sollen hinausgeworfen werden.

```
-- ownex04pl.sql
DECLARE
   CURSOR salcurs IS   SELECT empno,ename,comm FROM emp
                       WHERE job = 'SALESMAN';

   PROCEDURE fire (eno IN NUMBER) IS
   BEGIN
      DELETE FROM emp
      WHERE empno = eno;
   END;

BEGIN
   DBMS_Output.New_Line;
   FOR salcurs_rec in salcurs LOOP
      IF nvl(salcurs_rec.comm,0) < 500 THEN
         fire (salcurs_rec.empno);
         DBMS_Output.Put_Line('Employee '||salcurs_rec.ename ||' fired!');
         END IF;
      END LOOP;
   END;
```

Ergebnis :

```
Employee ALLEN fired!
Employee TURNER fired!

PL/SQL procedure successfully completed.
```

3. **Lokale Funktion** in anonymem Block : Das Gehalt aller Mitarbeiter, die gleichviel wie der Spitzenverdiener einer bestimmten Abteilung verdienen,soll um 10% angehoben werden (ungerecht!).

```
-- ownex05pl.sql
DECLARE
   maxsal      emp.sal%TYPE;
   empname      emp.ename%TYPE;
   CURSOR maxcurs (best_sal IN NUMBER) IS
      SELECT ename FROM emp
      WHERE sal = best_sal;

   FUNCTION bestsal (dno IN NUMBER) RETURN NUMBER IS
      maxsal   emp.sal%TYPE; -- lokal
   BEGIN
      SELECT max(sal)INTO maxsal FROM emp
         WHERE deptno = dno;
      RETURN (maxsal);
   EXCEPTION
      WHEN OTHERS THEN RETURN NULL;
   END bestsal;

BEGIN
   maxsal := bestsal(20);
   DBMS_Output.New_Line;

   FOR maxcurs_rec in maxcurs(maxsal) LOOP
      DBMS_Output.Put_Line('Employee '||maxcurs_rec.ename||
      ' from '||TO_CHAR(maxsal)||' to '|| TO_CHAR(maxsal * 1.1));
   END LOOP;

   UPDATE emp SET sal = sal * 1.1
   WHERE sal = maxsal;
END;
/
```

Ergebnis :

```
Employee SCOTT from 3000 to 3300
Employee FORD from 3000 to 3300

PL/SQL procedure successfully completed.
```

9.3 Gespeicherte Unterprogramme (Stored Subprograms)

Alle ORACLE-Tools (wie z.B. Oracle Forms) mit eingebauter PL/SQL Engine können Unterprogramme für spätere, strikt lokale Ausführung speichern.

Damit aber Unterprogramme für alle Tools zugänglich werden , müssen diese in einer Oracle Datenbank abgespeichert werden.

Zum Erstellen und dauernden Abspeichern von Unterprogrammen in einer Oracle Datenbank verwendet man die CREATE PROCEDURE oder CREATE FUNCTION Statements, die man interaktiv von SQL*Plus oder vom Enterprise Manager absetzen kann.

Mit der Ausführung eines solchen Statements wird das Unterprogramm kompiliert (→ P Code) und im Hauptspeicher (Shared Pool) sowie in der DB abgespeichert.

Im Shared Pool (Shared Memory Area der DB) steht das Unterprogramm dann mehreren Prozessen zur Verfügung. Die PL/SQL-Engine arbeitet den P-Code interpretativ ab.

9.4 Erstellen einer Stored Procedure

Um eine Prozedur erstellen zu können, muss der Autor das Recht CREATE PROCEDURE haben.

Um eine Prozedur ausführen zu können, muss der Benutzer das Recht EXECUTE haben.

Syntax:

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

Die Prozedur selbst wird in einem Texteditor geschrieben. Dieser Text wird eingeleitet mit:

```
CREATE PROCEDURE <procedure_name> AS ...
```

Beispiel für eine vollständige Prozedur:

```
CREATE PROCEDURE get_emp_rec (emp_number IN emp.empno%TYPE,
                              emp_ret OUT emp%ROWTYPE) AS
BEGIN
   SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno
     INTO emp_ret
     FROM emp
    WHERE empno = emp_number;
END;
/
```

Der Slash aktiviert die Prozedur.

9.5 Stored Functions

Syntax:

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

Werden in analoger Weise verwendet, wie z.B. aus Pascal bekannt:

```
CREATE FUNCTION dept_salary (dnum NUMBER) RETURN NUMBER IS
   ...   -- lokale Variablendeklaration
BEGIN
   ...
   RETURN total_wages;
END dept_salary;
```

Vorsicht : Funktionen dürfen keine DML-Statements enthalten. Unterprogramme mit DML-Statements müssen als Prozeduren definiert werden.

9.6 Ausführung einer Stored Procedure / Function

Eine Stored Procedure kann

- Aus dem Rumpf einer anderen Prozedur aufgerufen werden

```
sal_raise(emp_id, 200);
```

- interaktiv aus einem Oracle Tool gestartet werden (z.B. SQL*Plus), beispielsweise unter Verwendung eines anonymen PL/SQL Blockes:

```
BEGIN
         sal_raise(1043, 200);
END;
```

oder unter Verwendung des EXECUTE Kommandos

```
EXECUTE sal_raise(1043, 200)
```

- aus einer Applikation heraus aufgerufen werden (z.B. Pro*C-Programm)

  Dabei ist darauf zu achten, dass das EXECUTE Kommando verwendet wird:

```
EXEC SQL EXECUTE
   BEGIN
      fire_emp(:empno);
      /* hier wird die Prozedur mit einer Host-Variablen aufgerufen */
   END;
END-EXEC;
```

Eine Stored Function kann in ähnlicher Weise verwendet werden wie built-in – Functions

```
SELECT scott.my_funcs_pkg.my_func(10,20) from dual;
```

9.7 Rekursive Funktion

Beispiel :

```
CREATE OR REPLACE PROCEDURE Show_Emps(eno IN NUMBER, dashes IN VARCHAR2) IS
   CURSOR empc IS   SELECT empno,ename FROM emp WHERE mgr = eno;
   next_dashes      VARCHAR2(10);

BEGIN
   -- Parameter dashes gibt Einrueckung an ,z.B. '----':

   FOR empcrec IN empc LOOP
      DBMS_Output.Put_Line (dashes||to_char(empcrec.empno)||
                            ' '||empcrec.ename);
      next_dashes := dashes||'--';
      Show_Emps(empcrec.empno,next_dashes);
   END LOOP;

   EXCEPTION
      WHEN NO_DATA_FOUND THEN NULL;
END;
/
```

Aufruf : `execute Show_Emps(7839,'');`

Ergebnis :

```
7566 JONES
--7788 SCOTT
----7876 ADAMS
--7902 FORD
----7369 SMITH
7698 BLAKE
--7499 ALLEN
--7521 WARD
--7654 MARTIN
--7844 TURNER
--7900 JAMES
7782 CLARK
--7934 MILLER

PL/SQL procedure successfully completed.
```

10 Fehlerbehandlungen

10.1 Einführung

Exceptions (Ausnahmebedingungen) sind von Java her bekannt.

Ausnahmebedingungen werden durch Fehler (oder Warnungen) verursacht. Gründe hierfür sind:

1. Fehlermeldungen hervorgerufen durch das System
2. Fehler, die durch den User verursacht werden
3. Fehler hervorgerufen durch die Programmlogik

Bsp.: erg:=a/b;

wenn b=0 → ZERO DIVIDE

Es gibt vier Arten von PL/SQL-Exceptions

- **Named system exceptions** (Vordefinierte Ausnahmebedingungen, Named Predefined Exceptions): Fehlermeldungen die in PL/SQL benannt sind und durch den PL/SQL-Code oder das DBMS aufgerufen werden
- **Unnamed system exceptions** (Nicht vordefinierte Ausnahmebedingungen, Named Predefined Exceptions): nur die häufigsten Fehlermeldungen besitzen eine PL/SQL-Bezeichnung
- **Named programmer defined exceptions** (Benannte benutzerdefinierte Ausnahmebedingungen, Named User-Defined Exceptions): werden im Deklarationsteil des PL/SQL-Codes angegeben und werden explizit durch den Programmierer im PL/SQL-Code aufgerufen
- **Unnamed programmer defined exceptions** (Unbenannte benutzerdefinierte Ausnahmebedingungen, Unnamed User-Defined Exceptions): Diese Ausnahmen werden direkt am Server im PL/SQL-Code deklariert und aufgerufen (durch die RAISE_APPLICATION_ERROR-Prozedur).

Wenn ein Fehler auftritt, wird die Ausnahmebedingung aktiviert. Die normale Exekution der Applikation wird beendet und die Kontrolle wird an den Exception-Handler, der die Ausnahmebedingung bearbeitet, übergeben.

Benutzerdefinierte Ausnahmebedingungen müssen explizit durch das RAISE Statement aktiviert werden.

Beispiel:

```
DECLARE
pe_ratio NUMBER(3,1);
BEGIN
   SELECT price/earnings INTO pe_ratio FROM stocks
      WHERE symbol = 'XYZ'; -- might cause division-by-zero error
   INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
   COMMIT;

EXCEPTION -- exception handlers begin
   WHEN ZERO_DIVIDE THEN -- handles 'division by zero' error
      INSERT INTO stats (symbol, ratio) VALUES ('XYZ', NULL);
      COMMIT;
   ...
   WHEN OTHERS THEN -- handles all other errors
   ROLLBACK;
END; -- exception handlers and block end here
```

10.2 Vordefinierte Ausnahmebedingungen

Exception	Erklärung	Oracle Error
at *str* line num	This is usually the last of a message stack and indicates where a problem occurred in the PL/SQL code.	ORA-06512
CURSOR_ALREADY_OPEN	ein Cursor ist bereits offen	ORA-06511
DUP_VAL_ON_INDEX	wenn versucht wird in eine Tabellenspalte, auf die ein unique Index gelegt worden ist, einen Wert einzufügen, der bereits existiert.	ORA-00001
INVALID_CURSOR	wenn eine ungültige Cursoroperation ausgeführt wird. Z.B. wenn versucht wird, einen nicht geöffneten Cursor zu schließen.	ORA-01001
INVALID_NUMBER	wenn in einem SQL Statement versucht wird eine Zeichenkette in eine Zahl zu konvertieren und die Kette ungültige Zeichen enthält. In Non- SQL -Statements wird in diesem Fall die Exception VALUE_ERROR gesetzt.	ORA-01722
NO_DATA_FOUND	wenn bei einem SELECT INTO Statement keine Zeilen zurückgegeben werden. Beim FETCH Statement wird diese Bedingung nicht gesetzt. AVG und SUM geben immer einen Wert oder NULL zurück, daher wird die Bedingung NO_DATA_FOUND in diesem Fall nicht gesetzt.	ORA-01403
PROGRAM_ERROR	wenn ein PL/SQL internes Problem entstanden ist.	ORA-06501
TOO_MANY_ROWS	wenn bei einem SELECT INTO mehr als eine Zeile zurückgegeben wird.	ORA-01422

VALUE_ERROR	wenn ein arithmetischer, ein conversion-, truncation- oder constraint-Error auftritt. Z.B. wenn ein Spaltenwert in eine Charactervariable gelesen wird und diese Variable kürzer deklariert worden ist als die Spalte. Das folgende Beispiel liefert einen VALUE_ERROR:
	`DECLARE`
	`  my_empno NUMBER(4);`
	`  my_ename CHAR(10);`
	`BEGIN`
	`  my_empno := 'HALL'; --raises VALUE_ERROR`
	`  --In SQL statements, INVALID_NUMBER is`
	`  --raised instead.`
		ORA-06502

ZERO_DIVIDE	Division durch 0	ORA-01476

Anmerkung : SQLCODE hat bei diesen vordefinierten Exceptions immer den negativen Wert des ORACLE-Fehlercodes : z.B. ORA-01403 → SQLCODE = - 1403

Tab. 8: Vordefinierte Ausnahmebedingungen

10.3 Benutzerdefinierte Fehlermeldungen

Benutzerdefinierte Fehlermeldungen, die an die Client Application zurückgegeben werden, können mittels der Built-In Procedure `Raise_Application_Error` generiert werden.

Syntax:

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

Diese Prozedur beendet die Ausführung des PL/SQL-Codes, führt ein Rollback durch und gibt eine benutzerspezifische Fehlermeldung zurück.

`Error_Number` muss im Bereich von –20000 bis –20999 liegen.

`Error_Text` ist ein bis zu 2K langer Fehlertext.

`Keep_Error_Stack` wird auf TRUE gesetzt, wenn der aktuelle Fehler dem Error Stack hinzugefügt werden soll (Defaultwert: FALSE).

Typischerweise wird Raise_Application_Error im Exception Handler verwendet.

Diese Art der Ausnahmebehandlung wird auch als "Unnamed Programmer Defined Exceptions" bezeichnet und wird oft eingesetzt um Fehler aus serverseitigem Code (Code in stored objects) zu kommunizieren. Besonders zum "debuggen" von kompliziertem serverseitigem PL/SQL-Code sind unbenannte benutzerdefinierte Ausnahmen sehr nützlich.

Beispiel:

Im folgenden Beispiel wird eine Prozedur mit den Parametern Mitarbeiter -Nummer und Betrag, um den der Gehalt dieses Mitarbeiters erhöht werden soll, erstellt. Wenn der aktuelle Gehalt unbekannt ist (NULL), soll eine Ausnahmebedingung gesetzt werden:

```
CREATE PROCEDURE raise_salary (emp_id NUMBER, increase NUMBER) AS
   current_salary      emp.sal%TYPE;
BEGIN
   SELECT sal INTO current_salary FROM emp
    WHERE empno = emp_id;
   IF current_salary IS NULL THEN
      /* Issue user-defined error message. */
      raise_application_error(-20101, 'Salary is missing');
   ELSE
      UPDATE emp SET sal = current_salary + increase
       WHERE empno = emp_id;
   END IF;
END raise_salary;
```

Im Fehlerfall wird in SQL*PLUS folgende Ausgabe erzeugt:

```
ERROR at line 1:
ORA-20101: Salary is missing
ORA-06512: at "RAISE_SALARY", line 8
ORA-06512: at line 1
```

10.4 Compile Time Errors

Wenn bei der Erstellung einer Procedure oder Function (create procedure ..) ein Kompilierfehler auftritt, erscheint die Message:

```
MGR-00072: Warning: Procedure proc-name created with compilation errors
```

Mit SHOW ERRORS kann dann der Fehler angezeigt werden.

10.5 Benutzerdefinierte Ausnahmebedingungen

10.5.1 Deklaration

Diese Kategorie von Ausnahmebedingungen muss deklariert und durch ein RAISE Statement aktiviert werden.

```
DECLARE
past_due    EXCEPTION;
```

Diese Art der Ausnahmebehandlung wird auch als "Named Programmer Defined Exceptions" bezeichnet.

10.5.2 Exception-Handler

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

Beispiel:

wie obiges Beispiel mit benutzerdefinierten Ausnahmebedingungen :

```
CREATE PROCEDURE raise_salary (emp_id NUMBER, increase NUMBER) AS
   current_salary    emp.sal%TYPE;
   unknown_sal       EXCEPTION;

BEGIN
   SELECT sal INTO current_salary FROM emp
    WHERE empno = emp_id;
   IF current_salary IS NULL THEN
      /* Issue user-defined error message. */
      raise unknown_sal;
   ELSE
      UPDATE emp SET sal = current_salary + increase
       WHERE empno =emp_id;
   END IF;

EXCEPTION
      WHEN unknown_sal THEN
         UPDATE EMP SET SAL = 0
          WHERE EMPNO = EMP_ID;

END raise_salary;
```

Will mane einen Programmabbruch verhindern, obwohl ein Fehler aufgetreten ist, so sind verschachtelte BEGIN-Blöcke (nested blocks) zu verwenden.

10.6 Verwendung von SQLCODE und SQLERRM

Die SQLCODE Funktion gibt den Error-Code des letzten ausgeführten SQL Statements zurück.

(Alles o.k. --> Error-Code = 0)

Beispiel: Der letzte SQL Code soll in einer Fehlertabelle gespeichert werden:

Ausschnitt aus dem Execption- Handler:

```
WHEN OTHERS THEN
   sqlcode_var := SQLCODE;
   INSERT INTO ERROR_TABLE VALUES (sqlcode_var);
```

Wenn der zu einem Fehlercode gehörende Fehlertext relevant ist, dann kann die Funktion SQLERRM verwendet werden.

```
Message := SQLERRM (code);
INSERT INTO CODE_TABLE VALUES (code, message);
```

Anmerkung: Beide Funktionen, sowohl SQLCODE als auch SQLERRM können nicht direkt in SQL – Statements verwendet werden.

10.7 Zusammenfassung: Wann verwendet man benannte oder unbenannte Exceptions

1. Wird die Logik client-seitig ausgeführt und der Fehler client-seitig verursacht, so kann eine Ausnahme benannt werden.
2. Wird der Code serverseitig ausgeführt (Stored Procedure, DB-Trigger), so kann der Name der selbstbenannten bzw. selbst-definierten Ausnahme nicht an den Client übergeben werden → Man verwendet unbenannte Exceptions.

11 Packages

11.1 Überblick

- Packages sind eine Gruppierung von logisch zusammenhängenden PL/SQL Typen, Objekten, Stored Procedures und Stored Functions.
- Packages bestehen aus zwei Teilen:
    - Spezifikation: Schnittstelle zu den Anwendungen. Deklariert die Typen, Variablen, Konstanten, Exceptions, Cursor und Unterprogramme, die zur Verfügung gestellt werden sollen.
    - Body: Definiert Cursor und Unterprogramme vollständig und implementiert so die Spezifikation.
- Das Package selbst kann weder aufgerufen, parametrisiert noch verschachtelt werden. Einmal erstellt und kompiliert können die Inhalte von vielen Anwendungen gemeinsam genutzt werden.
- Erlaubt Oracle8 mehrere Objekte auf einmal in den Hauptspeicher zu lesen. Sobald ein PL/SQL-Konstukt eines Packages aufgerufen wird, wird das gesamte Package in den Hauptspeicher geladen. Somit benötigen spätere Aufrufe zusammenhängender Konstrukte keine Plattenzugriffe.

11.2 Vorteile von Packages

11.2.1 Modularität

Logisch zusammenhängende Programmstrukturen werden in einem benannten Modul verkapselt. Jedes Package ist einfach zu verstehen, und die Schnittstelle zwischen Packages ist einfach, klar und gut definiert.

11.2.2 Leichtere Anwendungsentwicklung

Alles was zu Anfang benötigt wird ist die Schnittstelleninformation in der Package-Spezifikation. Eine Spezifikation kann ohne ihren Body kodiert und kompiliert werden. Danach können gespeicherte Unterprogramme, die das Package referenzieren, ebenso kompiliert werden. Der Package-Body braucht solange nicht vollständig definiert zu werden, bis die Anwendung fertiggestellt werden soll.

11.2.3 Datenkapselung

Eine Unterscheidung zwischen öffentlichen (sichtbar und zugänglich; public) und privaten (verborgen und nicht zugänglich) Konstrukten ist möglich. Das Package verbirgt die Definition der privaten Konstrukte, sodass nur das Package und nicht die Anwendung betroffen ist, wenn die Definition verändert wird. Durch das Verbergen der Implementierung wird auch die Integrität des Package geschützt.

11.2.4 Zusätzliche Funktionalität

Zusammengefasste öffentliche Variablen und Cursor bestehen für die Dauer einer Sitzung. So können sie von allen Unterprogrammen. die in der Umgebung laufen, gemeinsam genutzt werden. Ebenso können Daten über Transaktionen hinweg aufrechterhalten werden, ohne sie in die Datenbank speichern zu müssen.

11.2.5 Bessere Performance

Sobald ein zusammengefasstes Unterprogramm zum ersten Mal aufgerufen wird, wird das ganze Package in den Hauptspeicher geladen. Auf diesem Wege erfordern spätere Aufrufe zusammenhängender Unterprogramme keine weiteren Plattenzugriffe. Zusammengefasste Unterprogramme stoppen auch kaskadierende Abhängigkeiten und vermeiden unnötige Kompilierung.

11.2.6 Overloading

Packages erlauben das Überladen von Prozeduren und Funktionen. Das bedeutet, dass mehrere Unterprogramme mit gleichem Namen in demselben Package angelegt werden können, die in Anzahl oder Datentyp unterschiedliche Parameter akzeptieren.

11.3 Anwendung von Packages

Beispiel: Es wird ein Package employee_management erstellt, welches eine Stored Function und zwei Stored Procedures enthält.

```
CREATE PACKAGE employee_management AS
FUNCTION hire_emp (name VARCHAR2, job VARCHAR2, mgr NUMBER, hiredate DATE,
                   sal NUMBER, comm NUMBER, deptno NUMBER) RETURN NUMBER;
PROCEDURE fire_emp (emp_id NUMBER);
PROCEDURE sal_raise (emp_id NUMBER, sal_incr NUMBER);
END employee_management;
```

Für Packages sind dieselben Rechte notwendig, wie im Abschnitt über Procedures besprochen.

```
CREATE PACKAGE BODY Employee_management AS
   FUNCTION Hire_emp (Name VARCHAR2, Job VARCHAR2,
      Mgr NUMBER, Hiredate DATE, Sal NUMBER, Comm NUMBER,
      Deptno NUMBER) RETURN NUMBER IS
       New_empno    NUMBER(10);

-- This function accepts all arguments for the fields in
-- the employee table except for the employee number.
-- A value for this field is supplied by a sequence.
-- The function returns the sequence number generated
-- by the call to this function.

   BEGIN
      SELECT Emp_sequence.NEXTVAL INTO New_empno FROM dual;
      INSERT INTO Emp_tab VALUES (New_empno, Name, Job, Mgr,
         Hiredate, Sal, Comm, Deptno);
      RETURN (New_empno);
   END Hire_emp;

   PROCEDURE fire_emp(emp_id IN NUMBER) AS

-- This procedure deletes the employee with an employee
-- number that corresponds to the argument Emp_id. If
-- no employee is found, then an exception is raised.

   BEGIN
      DELETE FROM Emp_tab WHERE Empno = Emp_id;
      IF SQL%NOTFOUND THEN
      Raise_application_error(-20011, 'Invalid Employee
         Number: ' || TO_CHAR(Emp_id));
   END IF;
END fire_emp;

PROCEDURE Sal_raise (Emp_id IN NUMBER, Sal_incr IN NUMBER) AS

-- This procedure accepts two arguments. Emp_id is a
-- number that corresponds to an employee number.
-- SAL_INCR is the amount by which to increase the
-- employee's salary. If employee exists, then update
-- salary with increase.

   BEGIN
      UPDATE Emp_tab
         SET Sal = Sal + Sal_incr
         WHERE Empno = Emp_id;
      IF SQL%NOTFOUND THEN
         Raise_application_error(-20011, 'Invalid Employee
            Number: ' || TO_CHAR(Emp_id));
      END IF;
   END Sal_raise;
```

```
END Employee_management;
```

Voraussetzung für die Implementierung des obigen Beispiels ist die Erstellung folgender Sequenz:

```
SQL> CREATE SEQUENCE Emp_sequence
   > START WITH 8000 INCREMENT BY 10;
```

(obiges Beispiel wurde der ORACLE-Dokumentation

11.4 Entwicklung eines Package

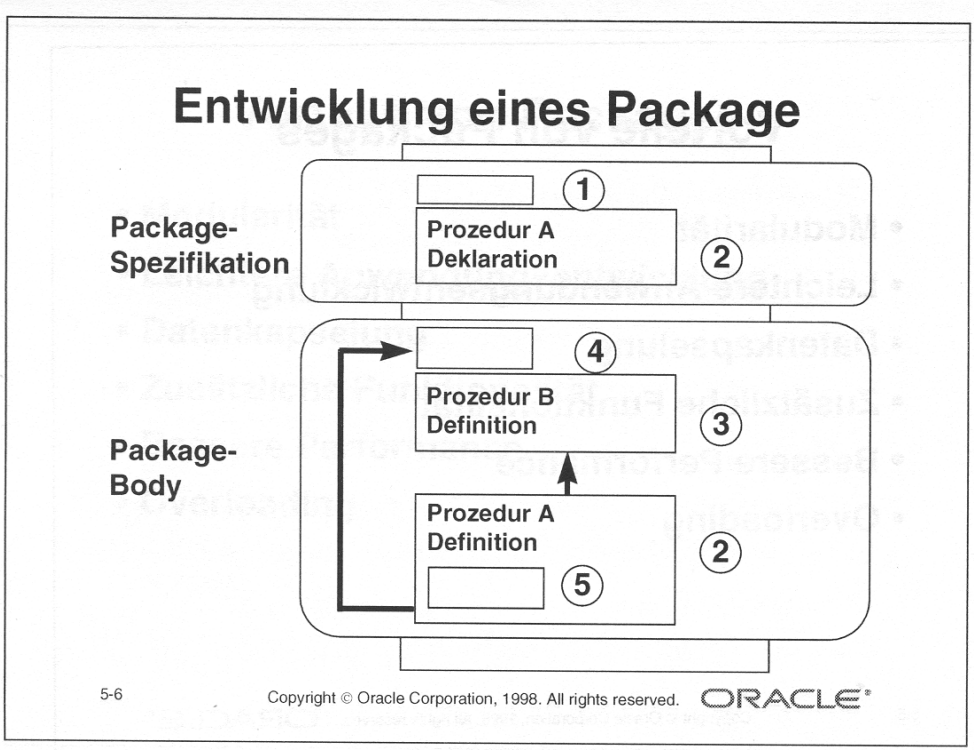


Abb. 4: Entwicklung eines Package

I. Öffentliche (globale) Variable

2. Öffentliche Prozedur

3. Private Prozedur

Ein Package wird in zwei Teilen angelegt: Eine Package-Spezifikation und ein Package-Body. Öffentliche (public) Package-Konstrukte sind in der Package-Spezifikation deklariert und im Package-Body definiert. Private Package-Konstrukte sind ausschließlich innerhalb des Package-Body definiert.

Gültigkeit des Konstrukts	Beschreibung	Plazierung innerhalb des Package
Public	Kann von jeder Oracle Server-Umgebung referenziert werden.	Deklariert innerhalb der Package-Spezifikation und definiert innerhalb des Package-Body.
Private	Kann nur von anderen Konstrukten referenziert werden, die Teil des selben Package sind.	Deklariert und definiert innerhalb des Package-Body.

Tab. 9: Gültigkeit eines PL/SQL-Konstrukts in Packages

Hinweis: Der Oracle Server speichert die Spezifikation und den Body eines Package getrennt in der Datenbank. Das erlaubt die Änderung der Definition eines Programmkonstrukts im Package-Body ohne andere Schemaobjekte, die das Programmkonstrukt aufrufen oder referenzieren, ungültig zu machen.

4. Private Variable (lokal. innerhalb des Package)

5. Lokale Variable (lokal, innerhalb der Prozedur)

Sichtbarkeit des Konstrukts	Beschreibung
Lokal	Eine Variable oder ein Unterprogramm, innerhalb eines anderen Unterprogramms definiert. (Kann durch andere Anwendungen nicht (referenziert werden und ist nur für den umgebenden Block sichtbar.) Lokal, innerhalb des Package: Eine Variable oder ein Unterprogramm kann innerhalb des Package Body definiert werden und kann somit von anderen Objekten desselben Packages referenziert werden. Lokal, innerhalb der Prozedur: eine Variable kann innerhalb einer Prozedur oder Funktion deklariert werden und kann somit auch nur innerhalb dieser referenziert werden-
Global	Eine Variable oder Unterprogramm, das außerhalb des Package referenziert (und verändert) werden kann.

Tab. 10: Sichtbarkeit eines PL/SQL-Konstrukts in Packages

11.5 Syntaxkonstrukte bei Packages

11.5.1 Anlegen einer Package-Spezifikation

Syntax:

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

Beispiel:

```
CREATE OR REPLACE PACKAGE comm_package
IS
   g_comm. NUMBER := 10;    -- initialisiert mit 10

   PROCEDURE reset_comm (v_comm IN NUMBER);

END comm_package;
```

11.5.2 Anlegen des Package-Body

Syntax:

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

Beispiel:

```
CREATE OR REPLACE PACKAGE BODY comm_package
IS
   FUNCTION validate_comm(v_comm IN NUMBER) RETURN BOOLEAN
   IS
      v_max_comm NUMBER;
   BEGIN
      SELECT MAX(comm)
        INTO v_max_comm
        FROM emp;
      IF v_comm > v_max_comm. THEN
         RETURN(FALSE);
      ELSE
         RETURN(TRUE);
      END IF;
   END validat_comm;

   PROCEDURE reset_comm (v_comm IN NUMBER)
   IS
      v_valid BOOLEAN;
   BEGIN
      v_valid := validate_comm(v_comm);
      IF v_valid = TRUE THEN
         g_comm := v_comm;
      ELSE
         RAISE_APPLICATION_ERROR(-20210,'Invalid commission');
      END IF;
   END reset_comm;
END comm_package;
```

11.5.3 Referenzieren öffentlicher Variablen und öffentlicher Prozeduren

```
comm_package.g_comm := 5

comm_package.reset_comm(8)
```

11.5.4 Löschen von Packages

Syntax:

Löscht der Package-Spezifikation und den Body:

```
DROP PACKAGE package_name
```

Löscht den Body:

```
DROP PACKAGE BODY package_name
```

Anmerkung: Folgende Themen wurden nicht behandelt:

- Beständigkeit von Package-Variablen, -Cursor, -Tabellen und –Records.
- Overloading (Besprechung im Detail)
- Vorwärts-Deklaration
- Definition eines Initialisierungsteils
- Einschränkungen bei Packages (z.B. kein INSERT, UPDATE, DELETE erlaubt)
- Reinheitsgrad einer Package-Funktion (PRAGMA RESTRICT_REFERENCES)

12 Ausgabe von Messages bei Prozeduren, Funktionen und Triggern

In Oracle gibt es ein Public Package DBMS_Output mit dem Messages ausgegeben werden können.

Voraussetzung für die Ausgabe ist, dass das SQL*Plus-Kommando SET SERVEROUTPUT ON abgesetzt worden ist.

12.1 Enable

Zur Verwendung der Package muss die Buffergröße definiert werden.

```
DBMS_OUTPUT.Enable(buffer_size IN INTEGER);
```

(Buffergrößen zwischen 2000 (default) und 1000000)

12.2 Put, Put_Line, New_Line

Die Put und Putline Proceduren sind überladen; es können daher IN Parameter der Typen NUMBER, VARCHAR2 oder DATE verwendet werden:

```
DBMS_Output.Put (item IN NUMBER);
DBMS_Output.Put (item IN VARCHAR2);
DBMS_Output.Put (item IN DATE);
DBMS_Output.Put_Line (item IN NUMBER);
DBMS_Output.Put_Line (item IN VARCHAR2);
DBMS_Output.Put_Line (item IN DATE);
DBMS_Output.New_Line;
```

Put_Line wird mit einem end_of_line ergänzt. Üblicherweise wird es daher zur Ausgabe ganzer Zeilen verwendet. Put zur Ausgabe von Teilen einer Zeile (Vergleichbar mit write, writeln aus Pascal).

Beispiel:

```
CREATE FUNCTION dept_salary (dnum NUMBER) RETURN NUMBER IS
   CURSOR emp_cursor IS SELECT sal, comm FROM emp WHERE deptno = dnum;
   total_wages    NUMBER(11, 2) := 0;
   counter        NUMBER(10) := 1;  -- debug counter
BEGIN
   FOR emp_record IN emp_cursor LOOP
      emp_record.comm := NVL(emp_record.comm,0);
      total_wages := total_wages + emp_record.sal + emp_record.comm;
      DBMS_Output.Put_Line('Loop number = '||counter||'; Wages = '||
      to_char(total_wages));        -- Debug line
      counter := counter + 1;       -- Increment debug counter
   END LOOP;

   DBMS_Output.New_Line;
   DBMS_Output.Put_Line ('Total wages = ' ||to_char(total_wages));
   RETURN total_wages;
END dept_salary;
```

Aufruf mit:

```
SET SERVEROUTPUT ON
```

```
VARIABLE salary NUMBER;

EXECUTE :salary := dept_salary(20);
```

13 DB-Trigger

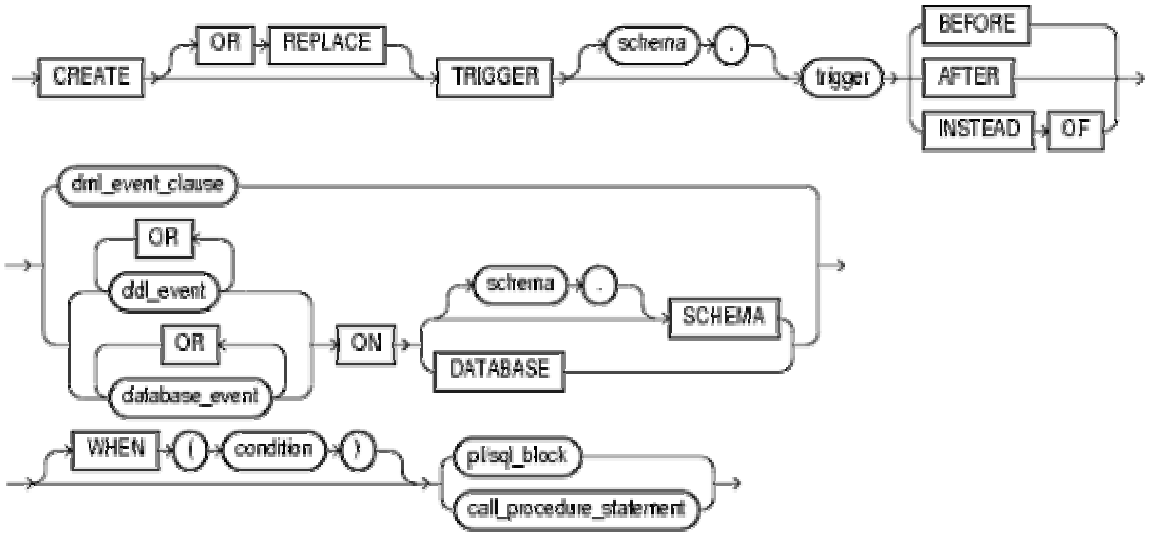
Datenbank-Trigger sind den Stored Procedures/Stored Functions ähnlich. Allerdings unterscheiden sie sich durch den Zeitpunkt Ihres Aufrufs.

Trigger können bei folgenden Ereignissen aufgerufen werden: Bei Ausführung von ....

- DML statements (DELETE, INSERT, UPDATE),
- DDL statements (CREATE, ALTER, DROP) und
- Database operations (SERVERERROR, LOGON, LOGOFF, STARTUP, SHUTDOWN)

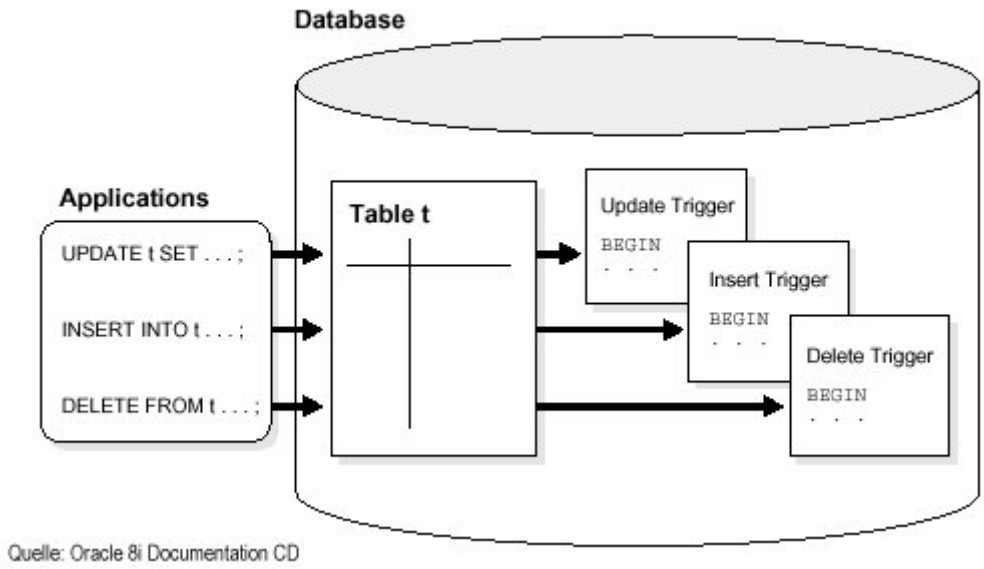
Ein Triger ist entweder ein gespeicerter PL/SQL-Block oder eine PL/SQL, C oder Java-Prozedur.

Syntaxdiagramm:



13.1 DML Triggers

Werden verwendet um bei INSERT-, UPDATE- und DELETE-Statements bestimmte Aktionen auszuführen.



Quelle: Oracle 8i Documentation CD

Trigger werden im Gegensatz zu Stored Procedures und Functions nicht explizit aufgerufen sondern automatisch beim Ausführen der entsprechenden INSERT-, UPDATE- bzw. DELETE-Statements.

Es gibt zwei Arten von Triggern:

- Nicht-Datensatzbezogene Trigger (Statement-Trigger)
- Datensatzbezogene Trigger (Row-Trigger, ForEachRow-Trigger)

Außerdem unterscheidet man noch zwischen:

- Vorab-Triggern (Before-Trigger)
- Danach-Triggern (After-Trigger)

Komponenten eines Triggers:

Tiggerkomponenten	entsprechende Syntax
❶ **Trigger-Name** ❷ **Trigger-Zeitpunkt** ❸ **Trigger-Ereignis** ❹ **Trigger-Typ** ❺ **Trigger-Restriktion** ❻ **Trigger Rumpf**	**CREATE OR REPLACE TRIGGER** <Name> **BEFORE | AFTER** **INSERT OR UPDATE [OF** <spalte1,..**] OR DELETE ON** <Tabellenname> **[FOR EACH ROW]** oder befehlsorientiert **WHEN** <PRÄDIKAT> PL-SQL Bock

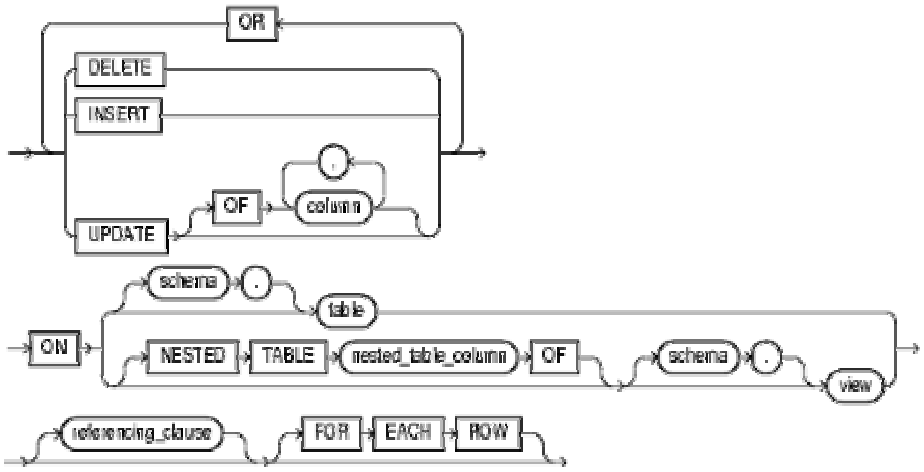
Schema:

```
CREATE OR REPLACE TRIGGER <Trigger-Name>
<Zeitpunkt> <Ereignis> ON <Tabellenname>
<für Statement oder jeden Datensatz>
<Ausführungsbedingung>
```

Syntax:

```
CREATE [OR REPLACE] TRIGGER triggername
  {BEFORE|AFTER|INSTEAD OF}
  {INSERT|UPDATE|DELETE [OF col_name1[,col_name2,...]]}
  [OR {INSERT|UPDATE|DELETE [OF col_name1[,col_name2,...]]}…]
  ON tab_name
  [REFERENCES [OLD [AS] old_refname] [NEW AS new_refname]]
  [FOR EACH ROW [WHEN bedingung]]
pl-sql-block
```

Syntaxdiagramm:

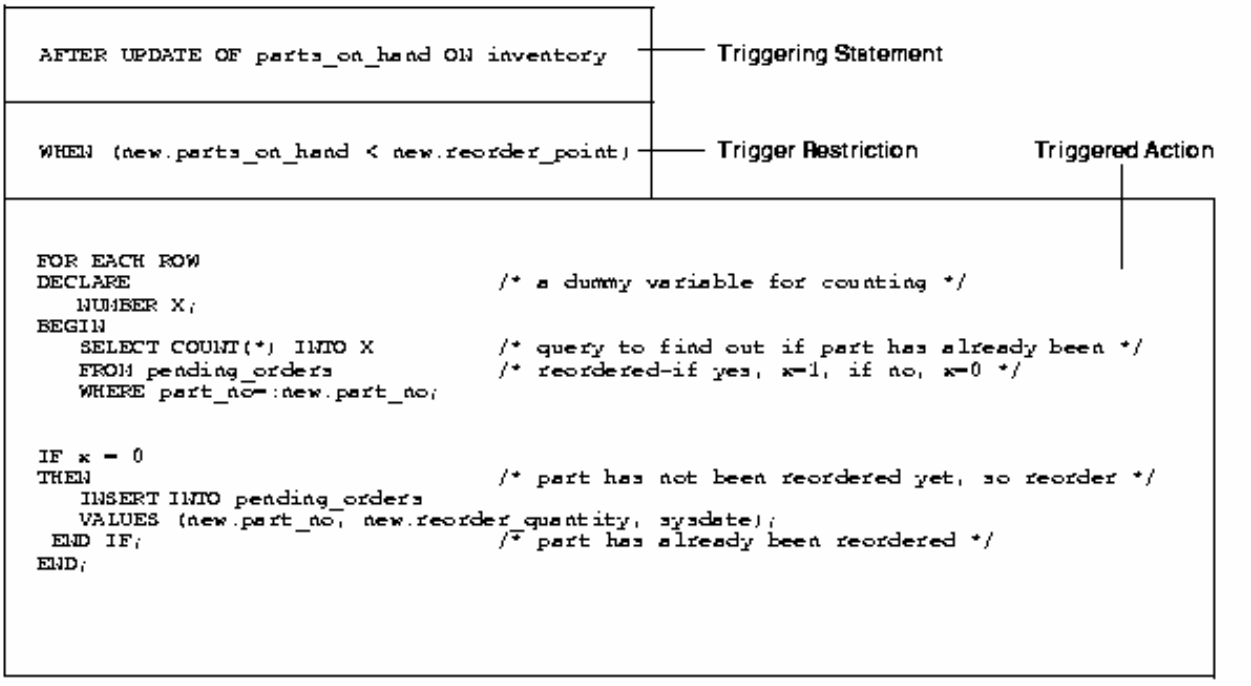


Beispiel 1:

```
CREATE TRIGGER audit_sal
   AFTER UPDATE OF sal ON emp
   FOR EACH ROW
BEGIN
   INSERT INTO emp_audit VALUES ...
END;
```

Hinweise:

- DB-Trigger bestehen aus drei Teilen:
  - a triggering event or statement
  - a trigger restriction
  - a trigger action



- die Spaltenliste in der
  `[OR {INSERT|UPDATE|DELETE [OF col_name1 [,col_name2,...]]}…]` – Zeile gilt nur für

die UPDATE-Klausel. Bei z.B. `AFTER UPDATE OF ename` wird der Trigger nur abgefeuert, wenn die Spalte ename verändert wird.

- Der ForEachRow-Trigger wird für jede Zeile ausgelöst, aber nur wenn die Bedingung wahr ist.
- Mit Hilfe der Schlüsselwörter `:new` und `:old` kann auf die geänderten bzw. ursprünglichen Werte der betroffenen Datensätze zugegriffen werden

  Bsp.: `… WHEN :new.ename = 'ALLEN';`

- Nur mit `FOR EACH ROW` kann auf die `:old` werte zugegriffen werden.
- Die Schlüsselwörter `:new` und `:old` können mit der REFERENCES-Klausel umbenannt werden.
- Innerhalb des Programms kann man mit den Schlüsselwörtern inserting, updating, bzw. deleting auf die jeweils verantwortliche Änderungsanweisung reagieren.

  Bsp.:
  ```
  IF INSERTING THEN …
  IF UPDATING THEN …
  IF DELETING THEN …
  ```

- Aktivieren/Deaktivieren eines Triggers mit
  ```
  ALTER TRIGGER trigger_name {ENABLE | DISABLE}
  ```
- Löschen eines Triggers mit
  ```
  DROP TRIGGER trigger_name
  ```
- Informationen über Trigger abrufen
  ```
  SELECT Trigger_type, Triggering_event, Table_name
    FROM USER_TRIGGERS
   WHERE Trigger_name = 'XXX';


  TYPE              TRIGGERING_STATEMENT       TABLE_NAME
  ----------------- -------------------------- ------------
  AFTER EACH ROW    UPDATE                     EMP



  SELECT Trigger_body
    FROM USER_TRIGGERS
   WHERE Trigger_name = 'XXX';


  TRIGGER_BODY
  ------------------------------------------
  DECLARE
     x NUMBER;
  BEGIN
     SELECT COUNT(*) INTO x
     ...
  END;
  ```

Beispiel 2:

```
CREATE TRIGGER at AFTER UPDATE OR DELETE OR INSERT ON emp
DECLARE
   typ CHAR(8);
   hour NUMBER;
BEGIN
   IF updating
   THEN typ := 'update'; END IF;
   IF deleting THEN typ := 'delete'; END IF;
   IF inserting THEN typ := 'insert'; END IF;
   hour := TRUNC((SYSDATE - TRUNC(SYSDATE)) * 24);
   UPDATE stat_tab
      SET rowcnt = rowcnt + stat.rowcnt
    WHERE utype = typ
      AND uhour = hour;
   IF SQL%ROWCOUNT = 0 THEN
      INSERT INTO stat_tab VALUES (typ, stat.rowcnt, hour);
   END IF;
EXCEPTION
   WHEN dup_val_on_index THEN
      UPDATE stat_tab
         SET rowcnt = rowcnt + stat.rowcnt
       WHERE utype = typ
         AND uhour = hour;
END;
```

13.2 DDL-Trigger

DDL-Trigger können auf Datenbankebene (`DATABASE`) oder Schema-Ebene (`SCHEMA`) deklariert werden.

Nachfolgend sind Beispiele für auslösende DDL-events angeführt:

`CREATE, ALTER, DROP, TRUNCATE, GRANT, REVOKE, DDL…`

Das DDL-Ereignis `DDL` löst bei irgendeinem der vohrher angeführten Events aus.

Beispiel: Nachfolgend wird ein Trigger erstellt, der nach dem Erstellen eines DB-Objektes im eigenen Schema einen PL/SQL-Block audführt.

```
CREATE TRIGGER audit_db_object AFTER CREATE
ON SCHEMA
< pl/sql_block >
```

Beispiel: Mit nachfolgenden Trigger wird das Löschen von DB-Objekten im Schema scott verhindert und eine Fehlermeldung wird ausgegeben:

```
CREATE OR REPLACE TRIGGER drop_trigger
BEFORE DROP ON scott.SCHEMA
BEGIN
RAISE_APPLICATION_ERROR (
num => -20000,
msg => 'Cannot drop object');
END;
/
```

13.3 Trigger für DB-Operationen

I.a. können Trigger für DB-Operationen ebenfalls auf DB-Ebene oder Schema-Ebene deklariert werden.

Nachfolgend sind Beispiele für auslösende DB-events angeführt:

`SERVERERROR, LOGON, LOGOFF, STARTUP, SHUTDOWN, SUSPEND` (falls eine Transaktion vom Server nicht ausgeführt werden kann)

Beispiel:

```
CONNECT system/manager
GRANT ADMINISTER DATABASE TRIGGER TO scott;

CONNECT scott/tiger
CREATE TABLE audit_table (
   seq      NUMBER,
   user_at  VARCHAR2(10),
   time_now DATE,
   term     VARCHAR2(10),
   job      VARCHAR2(10),
   proc     VARCHAR2(10),
   enum     NUMBER
);

CREATE OR REPLACE PROCEDURE foo (c VARCHAR2) AS
BEGIN
   INSERT INTO Audit_table (user_at) VALUES(c);
END;
CREATE OR REPLACE TRIGGER logontrig AFTER LOGON ON DATABASE
-- Just call an existing procedure. The ORA_LOGIN_USER is a function
-- that returns information about the event that fired the trigger.
   CALL foo (ora_login_user)
/
```

14 Native Dynamic SQL

14.1 Einleitung

In manchen Fällen ist es notwendig zur Laufzeit eine SQL-Abfrage zu erzeugen, d.h. das SQL-Statement kann sich bei jeder Programmausführung (und auch innerhalb eines Programmlaufes) ändern. In einem solchen Fall spricht man von einem dynamischen SQL-Statement.

Diese SQL-Statements werden in Strings gespeichert, welche durch das Anwendungsprogramm zusammengesetzt werden können. Diese Strings müssen ein gültiges (valid) SQL-Statement oder PL/SQL-Block enthalten. Durch sogenannte bind arguments können Parameter an das Statement übergeben werden.

z.B.: `'DELETE FROM emp WHERE sal > :my_sal AND comm < :my_comm'`

Ab der Version 7.1 konnten bereits dynamische Statements durch Verwendung des Packages DBMS_SQL erstellt werden. Da dies jedoch gewisse Probleme mit sich brachte (u.a. eine schlechte Performance), wurde ab Version 8.1 dynamisches SQL direkt in PL/SQL integriert. Daher spricht man von Native Dynamic SQL (NDS)

14.2 Anwendung von Native Dynamic SQL

NDS wird verwendet, da:

- Da DDL-Statements (CREATE,...), DCL-Statements (GRANT,…) und Session Control Statements in PL/SQL nicht statisch ausgeführt werden können.
- Um eine größere Flexibilität bei der Abfragegestaltung zu erreichen.
- Da die Verwendung von DBMS-SQL nicht performant genug ist.

14.3 Das EXECUTE IMMEDIATE Statement

EXECUTE IMMEDIATE bereitet das Statement vor (parsing) und führt es auch sofort aus.

Syntax:

```
EXECUTE IMMEDIATE dynamic_string
[INTO {define_variable[, define_variable]... | record}]
[USING [IN | OUT | IN OUT] bind_argument
[, [IN | OUT | IN OUT] bind_argument]...];
```

Beispiele:

```
DECLARE
   sql_stmt       VARCHAR2(100);
   plsql_block    VARCHAR2(200);
   my_deptno      NUMBER(2)          := 50;
   my_dname       VARCHAR2(15)       := 'PERSONNEL';
   my_loc         VARCHAR2(15)       := 'DALLAS';
   emp_rec        emp%ROWTYPE;

BEGIN
   sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
   EXECUTE IMMEDIATE sql_stmt USING my_deptno, my_dname, my_loc;

   sql_stmt := 'SELECT * FROM emp WHERE empno = :id';
   EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING 7788;

   EXECUTE IMMEDIATE 'DELETE FROM dept
   WHERE deptno = :n' USING my_deptno;

   plsql_block := 'BEGIN emp_stuff.raise_salary(:id, :amt); END;';
```

```
   EXECUTE IMMEDIATE plsql_block USING 7788, 500;

   EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt NUMBER)';

   sql_stmt := 'ALTER SESSION SET SQL_TRACE TRUE';
   EXECUTE IMMEDIATE sql_stmt;
END;
```

14.4 OPEN, FETCH und CLOSE Statements

Für Multi-Row-Abfragen sind Cursors zu verwenden.

14.4.1 Öffnen eines CURSORS

Syntax:

```
OPEN {cursor_variable | :host_cursor_variable} FOR dynamic_string
[USING bind_argument[, bind_argument]...];
```

Beispiel: Einer Cursor-Variablen wird ein dynamisches SQL-Statement zugewiesen.

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR; -- define weak REF CURSOR type
   emp_cv   EmpCurTyp; -- declare cursor variable
   my_ename VARCHAR2(15);
   my_sal   NUMBER      := 1000;

BEGIN
   OPEN emp_cv FOR -- open cursor variable
   'SELECT ename, sal FROM emp WHERE sal > :s' USING my_sal;
   ...
END;
```

14.4.2 Durchführen eines FETCH

Syntax:

```
FETCH {cursor_variable | :host_cursor_variable}
INTO {define_variable[, define_variable]... | record};
```

Beispiel: (Fortführung des obigen Codes)

```
LOOP
FETCH emp_cv INTO my_ename, my_sal; -- fetch next row
EXIT WHEN emp_cv%NOTFOUND; -- exit loop when last row is fetched
-- process row
END LOOP;
```

14.4.3 Schließen des Cursors

Syntax:

```
CLOSE {cursor_variable | :host_cursor_variable};
```

Beispiel: (Fortführung des obigen Codes)

```
LOOP
FETCH emp_cv INTO my_ename, my_sal;
EXIT WHEN emp_cv%NOTFOUND;
-- process row
END LOOP;
CLOSE emp_cv; -- close cursor variable
```

Weitere Beispiele

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR;
   emp_cv EmpCurTyp;
   emp_rec emp%ROWTYPE;
   sql_stmt VARCHAR2(100);
   my_job VARCHAR2(15) := 'CLERK';
BEGIN
   sql_stmt := 'SELECT * FROM emp WHERE job = :j';
   OPEN emp_cv FOR sql_stmt USING my_job;
   LOOP
      FETCH emp_cv INTO emp_rec;
      EXIT WHEN emp_cv%NOTFOUND;
      -- process record
   END LOOP;
   CLOSE emp_cv;
END;
```

14.5 Spezielle Probleme

14.5.1 Festlegen von Parameter Modes

***** noch offen *****

14.5.2 Verwendung doppelter Bind Arguments

Dem ersten Platzhalter im SQL-Statement wird automatisch dem ersten bind arguments in der USING-Klausel zugewiesen usw.

Beispiel:

```
DECLARE
   a NUMBER := 4;
   b NUMBER := 7;
BEGIN
   plsql_block := 'BEGIN calc_stats(:x, :x, :y, :x); END;'
   EXECUTE IMMEDIATE plsql_block USING a, b;
   ...
END;
```

15 Nützliche Statements:

- Überprüfen, welche Funktionen und Prozeduren vorhanden sind:
```
SELECT object_type, object_name FROM user_objects
WHERE object_type = 'PROCEDURE' OR object_type = 'FUNCTION';
```

- Den Code einer Funktion oder einer Prozedur ausgeben:
```
SELECT text FROM all_source WHERE name = 'func_name|proc_name';
```

- Ausgeben der Parameter einer Funktion oder Prozedur in der ORACLE-Konsole:
```
DESC func_name|proc_name
```

- Ausgewählte Tabellen und Views des Data-Dictionaries

Tabelle/View	Bemerkung
cat	Alle Tabellen des Users
tab	Alle Tabellen des Users (nur mehr aus Kompatibilitätsgründen vorhanden)
col	Sämtliche Spalten der Tabellen eines Users
ind	Sämtliche Indizes der Tabellen eines Users
user_constraints	Sämtliche Constraints eines Users Name Null? Type OWNER NOT NULL VARCHAR2(30) CONSTRAINT_NAME NOT NULL VARCHAR2(30) CONSTRAINT_TYPE VARCHAR2(1) TABLE_NAME NOT NULL VARCHAR2(30) SEARCH_CONDITION LONG R_OWNER VARCHAR2(30) R_CONSTRAINT_NAME VARCHAR2(30) ... (Liste der Attribute nicht vollständig)
user_cons_columns	Spalten der Constraints eines Users Name Null? Type OWNER NOT NULL VARCHAR2(30) CONSTRAINT_NAME NOT NULL VARCHAR2(30) TABLE_NAME NOT NULL VARCHAR2(30) COLUMN_NAME VARCHAR2(4000) POSITION NUMBER

Weitere Tabellen/Views: user_triggers, user_views, all_views, obj, seg, syn,…

Nicht behandelt wurden:

- Transaktionen (mit SAVEPOINT, ROLLBACK und COMMIT)
- INSTEAD-OF-Trigger