Struktur eines PL/SQL Blocks

DECLARE	
<Deklarationsteil>	Definition und Initialisierung[1] der im Block benutzten Variablen und Cursors
BEGIN	
<Ausführungsteil>	Wertzuweisungen, Operationen und Kontrollflußanweisungen
EXCEPTION	
<Ausnahmebehandlung>	liefert eine speziell angepaßte Fehlerbehandlung
END;	

[1] Variable automatisch „0“

CURSOR

mengenorientierung (SQL) + datensatzorientierung (PS)

Cursor sind immer dann nötig, wenn Daten über **SELECT** ausgelesen werden, aber mehr als ein Datensatz vorhanden ist.

Cursor können innerhalb eines PL/SQL-Blocks oder innerhalb einer Stored Procedure **deklariert(1)** werden. **Öffnen(2)** des Cursors führt das **SELECT**-Statement komplett aus, übermittelt aber noch kein Argebnis an das Client-Programm. Der Client kann mittels **FETCH**-Anweisung(3) jeden Eintrag aus der Ergebnisliste einzeln sequentiell lesen und verarbeiten. Die Ergebnisliste steht bis **CLOSE CURSOR(4)** zur Verfügung

Beispiel:

```
DECLARE
i    pls_integer;

CURSOR delete_ds(grenze number default 299) IS
     SELECT    pnr
     FROM      personal
     WHERE     pnr > grenze;

BEGIN
     OPEN delete_ds(299);
     LOOP
          FETCH delete_ds
          INTO I;
          EXIT WHEN delete_ds%NOTFOUND;
          DELETE FROM personal
          WHERE pnr = i;
     END LOOP;
     CLOSE delete_ds;
END;
/
```

Im Zuge der Cursoröffnung wird dem Cursor ein Parameter mitgegeben, der die SQL-Anweisung innerhalb des Cursors definiert. %NOTFOUND ist true wenn die letzte Fetch-Anweisung keinen Datensatz mehr lieferte.

Schleifen (For-Cursor):

Beispiel:

```
DECLARE
fg   constant  number(4)      :=3000;
pa   constant  number(7,5)    :=0.00987;
gehalt         number(10,5);

CURSOR umsatz_cursor IS
     SELECT    *
     FROM      T_umsatz;

umsatz_var     umsatz_cursor%rowtype;  --lt. Ge nicht nötig?!

BEGIN
     FOR umsatz_var IN umsatz_cursor
     LOOP
          gehalt    := fg+pa*umsatz_var.umsatz;
          INSERT INTO    T_gehalt
          VALUES (umsatz_var.umsatz, gehalt);
     END LOOP;
END;
/
```

Vorteile der **FOR**-Schleife:

- Cursor wird automatisch geöffnet und nach der Operation geschlossen.
- Bei jedem Schleifendurchlauf wird der Datensatzzeiger um 1 nach unten bewegt (like **FETCH**-Anweisung).
- %NOTFOUND wird automatisch mitgeprüft.

Rowid:

Jeder Datensatz jeder Tabelle innerhalb einer Oracle-Datenbank besitz eine eindeutige Rowid.

```
SELECT rowid, pnr FROM personal;

ROWID                  PNR
------------------     ------
00000035.0000.0002     123
00000035.0001.0002     127
00000035.0002.0002     130
00000035.0003.0002     131
```

Feld kann nur gelesen, nicht beschrieben werden.

Beispiel:

```
DECLARE
i         pls_integer;
neu_pnr   pls_integer;

CURSOR auto_pnr IS
     SELECT    rowid
     FROM      personal
     FOR UPDATE;    --Die zu ändernde Tabellenspalte ist
                    --eindeutig durch die Selektion spezifiziert

BEGIN
     OPEN auto_pnr;
     new_pnr:=1;
     LOOP
          FETCH auto_pnr
          INTO I;
          EXIT WHEN auto_pnr%notfound;

          UPDATE    personal
          SET       ident_pnr=neu_pnr
          WHERE     rowid=i;
          neu_pnr:=neu_pnr+1;
     END LOOP
END;
/
```

Trigger

Trigger sind Proceduren, die mit Tabellen verknüpft sind und beim Auftreten bestimmter Ereignisse (meist best. SQL-Befehle) automatisch ausgeführt werden. Trigger werden je nach Vereinbarung vor oder nach einer SQL-Anweisung ausgeführt, entweder 1 mal pro Anweisung oder je 1 mal pro angesprochener Tabellenzeile (**FOR EACH ROW**).

- es wird Funktionalität in die DB ausgelagert
- es können Aktionen ausgelöst werden, wenn zuvor defionierte Ereignisse eintreten
- Netzlast sinkt, da der DB-Server beschäftigt wird
- Wartungsfreundlicher
- ermöglicht, die Überprüfung aus der Anwendung in die DB zu übertragen

Ziele:

Sicherung der Integrität der DB

- Semantische I. (Richtigkeit der Eingabe)
- Schlüsselintegrität (Eindeutige Identifikation)
- Referenzielle Integrität (Fremschlüsselintegrität)
- Operationale I. (Mehrbenutzerbetrieb)

Probleme:

- Erschwerte Wartung
- Kaskadierungseffekt (Trigger aktiviert Trigger)
- Aktivierungsreihenfolge manchmal schwer durchschaubar (mehrere Trigger pro Tabelle)
- Fehlerpotential, vor allem in der **WHEN**-Klausel (ein Trigger eines Typs pro Tabelle)

Syntax:

```
CREATE[OR REPLACE] TRIGGER [user.]triggername

{BEFORE|AFTER|INSTEAD OF}

     {DELETE
     |INSERT
     |UPDATE [OF column[, column]...]}
     [OR
     {DELETE
     |INSERT
     |UPDATE [OF column [,column]...]}]...
     ON [user.]{TABLE|VIEW}

[FOR EACH ROW]      --Zeilentrigger, sonst: Anweisungstrigger
[WHEN Bedingung]

BEGIN

Anweisungsblock

END;
/
```

Besonderheiten:

:old und :new

In Zeilentriggern kann man auf alte und neue Werte der aktuell zu verarbeitenden Zeile zugreifen. In PL/SQL-Blocks muß der Doppelpunkt voranstehen, in WHEN Bedingungen nicht. Logischerweise ist bei INSERT nur :new und bei DELETE nur :old verfügbar. Bei UPDATE-Triggern ist sowohl :new als auch :old da.

Beispiel:

```
INSERT INTO verlauf
(datum, pnr, pnr_neu)         --optionale Angabe
VALUES
(sysdate, :new.pnr, :old.pnr);
```

aber:

```
WHEN new.preis > old.preis;
```

Trigger können natürlich auch **STORED PROCEDURES** (Siehe Seite 8) aufrufen.

Beispiel allgmein:

Bei jeder Änderung der Tabelle personal wird ein Eintrag in die Tabelle verlauf getätigt.

```
CREATE TABLE verlauf(
datum     date,
pnr       number(4),
pnr_neu   number(4));

CREATE OR REPLACE TRIGGER verlauf_tr BEFORE
UPDATE ON personal
FOR EACH ROW
BEGIN
INSERT INTO verlauf
VALUES
(sysdate, :old.pnr, :new.pnr);
END;
/
```

STORED PROCEDURES

- Menge von SQL oder PL/SQL-Kommandos
- Code statisch, kann ohne Neukompilation verwendet werden
- Plattform-unabhängig

Syntax:

CREATE [OR REPLACE] PROCEDURE[.user**]** procedure

[(argument **[IN**|**OUT**|**IN OUT]** datatype

[,argument **[IN**|**OUT**|**IN OUT]** datatype**]...)]**

{IS|**AS}{**block|external program**};**

IN ist für Input, **OUT** für Output, **IN OUT** ist beides (vgl. Parameter in Delphi). Default ist **IN**. block ist der PL/SQL-Block der ausgeführt werden soll.

Es können in PL/SQL auch Funktionen erstellt werden.

Syntax:

```
CREATE [OR REPLACE] FUNCTION[.user] function
[(argument [IN|OUT|IN OUT] datatype
[,argument [IN|OUT|IN OUT] datatype]...)]
RETURN datatype                              --wie in PS
{IS|AS}{block|external program};
```

Beim Einsatz von Funktionen nicht die Wertzuweisung im block vergessen (ähnlich Delphi Result:=...)

RETURN pnr;

Beispiel:

```
CREATE OR REPLACE FUNCTION auswerten
RETURN number
IS
     anz_mitarbeiter number;

BEGIN

     SELECT COUNT(pnr)
     INTO anz_mitarbeiter
     FROM personal;
     RETURN anz_mitarbeiter;

END;

CREATE OR REPLACE TRIGGER test AFTER

INSERT OR DELETE ON personal

BEGIN

     INSERT INTO info(datum, ma)

     VALUES (sysdate, auswerten)

END;
/
```

Parameterübergabe:

```
CREATE OR REPLACE PROCEDURE insert_personal(

pnr             smallint,
name            varchar2,
vorname         char       :=NULL,   -- Standardparameter 4x!
geh_stufe       char       :=NULL,
abt_nr          char       :=NULL,
krankenkasse    char       :=NULL)

IS
BEGIN
     INSERT INTO personal
     (pnr, name, vorname, geh_stufe, abt_nr, krankenkasse)
     VALUES
     (pnr, name, vorname, geh_stufe, abt_nr, krankenkasse)
END;
/
```

Übergabe per Position:

```
EXECUTE insert_personal(120,'kruse','Stefan','it3','d12','TKK');
```

→ Die Datenbank erwartet eine vollständige Parameterliste.

Übergabe per Name:

```
EXECUTE insert_personal(pnr=>122, name=>'kunze');
```

→ einzelne Parameter möglich.

Objekt = Daten + Methoden
(-Klassen)

Relationales Modell ←→ Objektmodell
(VARRAYs, LOBs, Objekte)

Script Kap 5+6