

Einführung PL-SQL



- PL/SQL - Die Programmiersprache
- PL/SQL Block
- Systeminterne Spalten
- Das Cursorkonzept
- Procedures und Packages
- Constraints
- Trigger

Die Programmiersprache

Warum PL-SQL ?

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

Syntaktische Elemente

Sprachelemente im PL-SQL Block

- Einfache und zusammengesetzte Operatoren
- Schlüsselwörter und Variablen
- Numerische Ausdrücke und Konstanten
- Selektion, Iteration, Sequenz - Sprachelemente
- Fehlerbehandlung

Operatoren

Einfache Operatoren

- +** addition operator
- subtraction/negation operator
- *** multiplication operator
- /** division operator
- =** relational operator
- <** relational operator
- >** relational operator
- (** expression or list delimiter
-)** expression or list delimiter
- ;** statement terminator
- %** attribute indicator
- ,** item separator
- .** component selector
- @** remote access indicator
- '** character string delimiter
- "** quoted identifier delimiter
- :** host variable indicator

Zusammengesetzte Operatoren

- **** exponentiation operator
- <>** relational operator
- !=** relational operator
- ~=** relational operator
- ^=** relational operator
- <=** relational operator
- >=** relational operator
- :=** assignment operator
- =>** association operator
- ..** range operator
- ||** concatenation operator
- <<** (beginning) label delimiter
- >>** (ending) label delimiter
- single-line comment indicator
- /*** (beginning) multiline comment delimiter
- */** (ending) multiline comment delimiter

Datentypen

einfache Datentypen

char	1..32767 bytes
varchar	
varchar2	
long	1.65535 bytes
number	1e-129...9.99e125
decimal	
float	
integer	
real	
smallint	
binary_integer	$-2^{31}..2^{31}-1$
natural	$0..2^{31}-1$
positiv	$0..2^{31}-1$
boolean	true false
date	4172 v. Chr-4712 n. Chr
raw	1..32767
rowid	18 bytes
mlslabel	4 bytes (Trusted Oracle)
raw mlslabel	255 bytes (Trusted Oracle)

zusammengesetzte Datentypen

TABLES

RECORDS

Datentypen

Zusammengesetzte Datentypen

Table (Array)

```
declare  
type text_tab is table of varchar2(80)  
           index by binary_integer;  
  
text      text_tab;  
abc       text_tab;  
  
begin  
text(1):='Hello world';  
text(2):='Good morning Vietnam';
```

Record

```
declare  
type mit_rec_type is record  
           (mit_name varchar2(50),  
            mit_geb   date,  
            mit_abtnr mitarb.mit_abtnr%type);  
  
mit_rec      mit_rec_type;  
  
begin  
mit_rec.mit_geb:=to_date('02-jan-65');
```

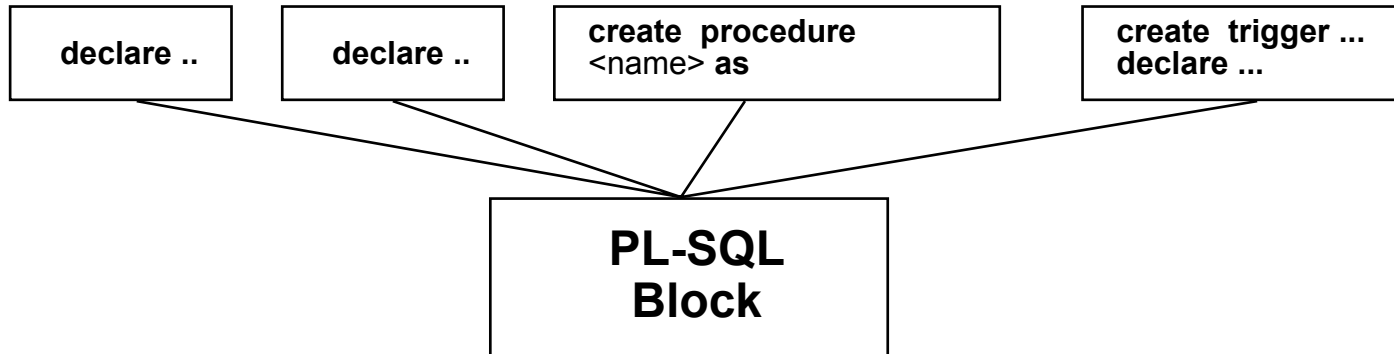
➞ Zusammengesetzte Datentypen müssen zuerst typdefiniert und können dann erst deklariert werden.

Einsatz im Oracle System

Wo wird PL-SQL eingesetzt ?

- 4-GL Prozeduren
- Anonyme PL-SQL Blöcke (Scripts)
- Gespeicherte PL-SQL-Programme
- DB-Trigger

Diese 4 Typen von Programmen benutzen die selbe PL-SQL-Blockstruktur unterscheiden sich jedoch bei der Erzeugung.



Teile eines PL-SQL Programms

Die Blockstruktur

Programmteile eines PL-SQL Blocks

- **Deklarationsteil**
- **Ausführungsteil**
- **Fehler - und Ausnahmebehandlungsteil**

Deklarationsteil

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

Ausführungsteil bzw Programmkörper

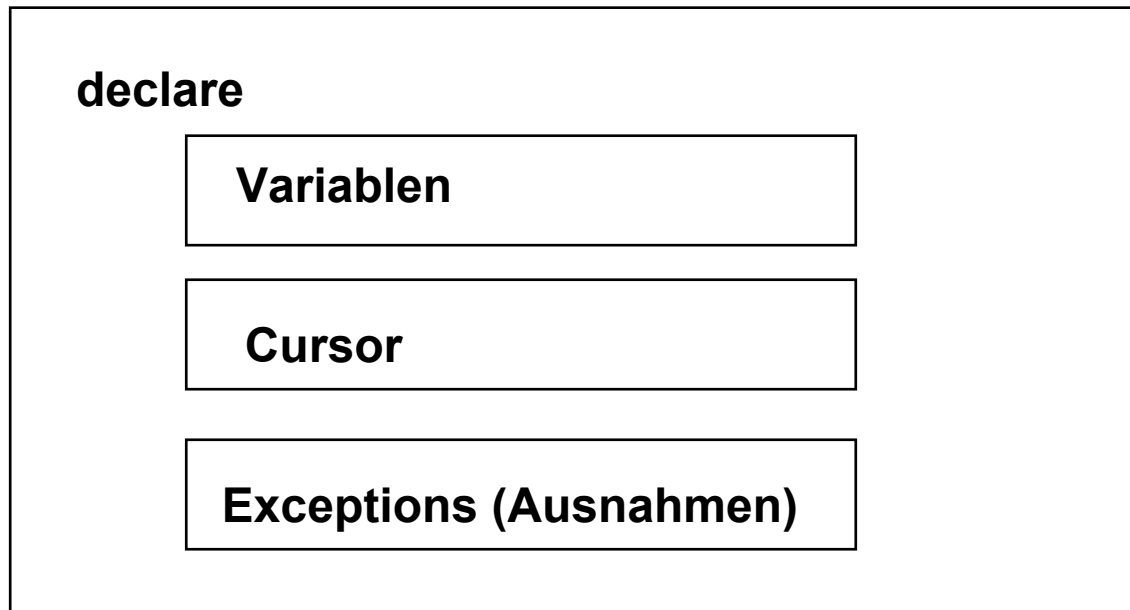
Der Ausführungsteil beinhaltet das eigentliche algorithmische Programm.

Fehler - und Ausnahmebehandlungsteil

In diesen Teil wird stets bei Fehlern und Ausnahmen verzweigt um aufgetretene Fehlersituationen bearbeiten zu können.

Der Deklarationsteil

Typischer Deklarationsteil eines PL-SQL Programms



↗ Variablen, Cursor und Exeptions können beliebig kombiniert werden, aber auch weggelassen werden.

Variablen

Im Deklarationsteil werden unter anderem **Datentypdeklarationen** der verwendeten **Variablen** und **Konstanten** durchgeführt.

einfache Beispiele für Variablen

declare

name_pls		varchar2(50);	
ok		boolean;	
wert	constant	number(9)	default 100;
saldo_pls		number (5)	not null:=0;

➤ Die Variablenlänge kann frei gewählt werden, soll jedoch 30 Zeichen nicht überschreiten und kein Schlüssel sein.

Variablen

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

korrekte Bezeichner

dd
X
t2
phone#
credit_limit
LastName
oracle\$number

inkorrekte Bezeichner

mine&yours ungültiges &
ebit-amount - ist auch verboten
on/off / ebenfalls
user id Leerezeichen auch nicht
Reservierte Wörter dürfen **nicht** als
Variablennamen benutzt werden

➤ PL-SQL ist **non case sensitive**

➤ **Achtung!** Ausdrücke wie $x < y$ **ohne Leerzeichen** schreiben(7.0)

Variablen

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 Types einer bestimmten Spalte zu deklarieren.

```
declare  
name_var1      mitarbeiter.mit_name%type
```

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
```

Ausnahmen und Fehler

Arten von Ausnahmen

- vom System vordefinierte Ausnahmen
- vom Programmierer definierte Ausnahmen
- sonstige Oracle Fehlersituationen, für die Oracle einen Fehlercode liefert

Vom **System vordefinierte Ausnahmen** müssen **nicht deklariert** können jedoch im Behandlungsteil berücksichtigt werden

Vom **Programmierer definierte Ausnahmen** müssen als (Ausnahmen) EXCEPTIONS deklariert werden.

P1_stop **exception;**

tomany **exception;**

In weiter Folge werden die Exceptions mit **RAISE** aufgerufen und im Fehlerbehandlungsteil behandelt.

Ausnahmen und Fehler

Sonstige **Oracle Fehlersituationen**, für die Oracle einen Fehlercode liefert können um nicht während des Programmlaufs ausdrücklich ermittelt werden zu müssen vom Programmierer mittels

pragma exception_init(*eigener Exceptionname, Oracle Fehlernummer*)

deklariert werden. Der Vorteil dieser Methode der Fehlerbehandlung besteht darin, daß die Exception automatisch bei Auftreten eines Fehlers behandelt wird. Dies geschieht wie wir noch sehen werden im Ausnahmebehandlungsteil des PL-SQL Blocks.

Beispiel

```
pragma exception_init(max_open_cursor, -1000)
```

Wir sehen, daß sämtliche Fehlernummern auch die der bereits vordefinierten Exceptions umbenannt werden können.

➤ Oracle Fehlernummern findet der Programmierer in der Hilfe.

➤ Laß Oracle für dich die Arbeit machen und verwende exception_init()

Ausnahmen und Fehler

Wichtige Exeptions !

Exception Name	ORACLE Error	SQLCODE Value
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
STORAGE_ERROR	ORA-06500	-6500
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
TRANSACTION_BACKED_OUT	ORA-00061	-61
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	OAR-01476	-1476

Das Cursorskonzept

Was ist ein Cursor ?

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

- ❶ Wie Variablen werden auch Cursor im Deklarationsteil **deklariert** und erhalten dort einen Namen und eine SELECT-Anweisung.
- ❷ Das **Öffnen** eines Cursors bewirkt die Ausführung des SELECT-Statements. Nach dem Öffnen des Cursors verwaltet dieser eigenständig einen Zeiger auf die aktuelle Zeile der selektierten Datenzeilen, wobei dieser Zeiger direkt nach dem Öffnen des Cursors auf die erste Datenzeile zeigt.
- ❸ Wurde bei der select Anweisung **FOR UPDATE** angegeben bewirkt das Öffnen des Cursors, daß alle selektierten Sätze **gesperrt** werden, und das Schließen die Aufhebung der Sperre
- ❹ **Schließen** des Cursors.

Das Cursorkonzept

Beispiel: Grundstruktur der Cursorverarbeitung mit einer herkömmlichen Schleife

```
DECLARE
```

```
CURSOR lcurs IS                                /* Deklariere Cursor */  
  SELECT    liefer_nr, liefer_dat, liefer_ref  
  FROM      liefer_tab  
  WHERE      liefer_tab='I';
```

```
liefer_rec  lcurs%ROWTYPE; /* Struktur mit Attributen einer Zeile */
```

```
BEGIN
```

```
OPEN lcurs;                                /* Öffne Cursor */
```

```
LOOP
```

```
  FETCH lcurs INTO liefer_rec;            /* Einlesen der Zeile */
```

```
  EXIT WHEN lcurs%NOTFOUND; /* Verlasse Einlesen wenn kein Datensatz */
```

```
END LOOP;                                /* mehr gefunden wird */
```







```
CLOSE lcurs;                                /* Schließe Cursor */
```

```
END;
```

Der Ausführungsteil

Der **Ausführungsteil** eines PL-SQL Programms enthält die eigentliche **Ablauflogik** eines PL-SQL Blocks. Hier werden in Abhängigkeit von verschiedenen Variablen, von Ergebnissen aus SQL-Befehlen und gemäß bestimmter Ausnahmefälle unterschiedliche Aktionen ausgeführt, Werte berechnet oder Daten innerhalb der Oracle Datenbank gelesen oder geändert.

Wichtige Bestandteile des Ausführungsteils

-  **Zuweisungen**
-  **Bedingte Verarbeitung (if .. then .. else ..)**
-  **Schleifenkonstrukte**
-  **Ausführung von SQL-Statements**
-  **FOR-Cursor-Schleife (Cursorabarbeitung)**
-  **Auslösen von Ausnahmen (Ausnahmebehandlung)**

Zuweisungen

```
DECLARE  
TYPE myrec IS RECORD  
(element1      NUMBER,  
 element2      DATE,  
 element3      CHAR );  
  
x      NUMBER;  
y      liefer tab.liefer_nr%TYPE;  
c      VARCHAR2 (5);  
rec    myrec;  
rec1   rec%TYPE;  
  
BEGIN  
  x:=10;  
  c:='Hello World';  
  y:='25';  
  rec:=rec1;  
  rec.element2:='10-MAY-96';  
  rec.element1:=rec1.element1*x;  
END;
```

Zuweisungen zwischen RECORDS
können nur zwischen RECORDS
gleichen Typs erfolgen.

Datumskonvertierung nur
möglich, wenn Datum im
Standardformat vorliegt.

➞ Der **Initialwert** sämtlicher deklartierter Variablen ist **NULL** !

Zuweisungen

Strukturkompatibilität

DECLARE

```
TYPE LieferRecType IS RECORD  
(liefer_nr NUMBER  
  liefer_datum DATE  
  liefer_ref NUMBER);
```

```
liefer_rec      liefer_tab%ROWTYPE; /* Gleiche Struktur wie LieferRecType */  
liefer_rec1     LieferRecType;
```

BEGIN

```
liefer_rec:=liefer_rec1;           /* ungültig weil RECORDS nicht typgleich sind */
```

```
liefer_rec.liefer_nr:= liefer_rec1.liefer_nr;      /* funktioniert */  
END;
```

➡ Arbeit mit ROWTYPE erleichtert Arbeit !

Zuweisungen

Tabellen

```
DECLARE  
  
TYPE LieferTabType IS TABLE OF liefer_tab.liefer_nr%TYPE  
    INDEX BY BINARY_INTEGER;  
  
liefer_tab          LieferTabType;  
  
BEGIN  
  
liefer_tab(17):=1;  
liefer_tab(-15):=liefer_tab(17)+1;  
  
END;
```

Sollen an eine Tabelle Werte zugewiesen werden so müssen diese mit einem Primärschlüssel in Klammern gekennzeichnet sein. Eine **Tabelle** ist kein Array sondern ein **einspaltiger Table** in der nur elementare Datentypen gespeichert werden können. Eine Tabelle wird jedoch anders gespeichert als ein Table.

Zuweisungen

Wichtig ist bei der **Verbindung mehrerer Operanden**, daß sie **typengleich** sind und daß **kein einziger Wert NULL** ist, sonst ist der **ganze Ausdruck NULL !!**

```
zkette:='Hello'||'World'||'!';  /* verwenden des Konkatenationsoperators */
```

```
zahl:=1+2*3+4;                /* als Ergebnis 11 */
```

```
zahl:=(1+2)*(3+4);            /* als Ergebnis 21 */
```

```
zahl:=zahl+1;
```

```
zahl:=NVL(zahl,0)+1;
```

```
zahl:=zahl+1;
```

ergibt NULL, wenn zahl der Wert NULL hat

```
zahl:=NVL(zahl,0)+1;
```

ergibt 1, wenn zahl den Wert 0 hat

Bedingte Ausführungen

IF... THEN... ELSE...

Um die bedingte Ausführung einer Anweisung zu erzwingen stehen die Konstrukte

IF .. THEN .. ELSE
IF .. THEN .. ELSIF
zur Verfügung

IF .. THEN .. ELSE

```
IF Bedingung THEN  
  ...  
ELSE  
  ...  
END IF;
```

IF .. THEN .. ELSIF

```
IF Bedingung THEN  
  ...  
ELSIF bed2 THEN  
  ...  
ELSIF bed3 THEN  
  ...  
END IF;
```


Schleifenkonstrukte

Unbedingte Schleifen

LOOP Schleife

LOOP .. END LOOP

LOOP

Anweisungen

IF bedingung **THEN EXIT;**




weitere Anweisungen

END LOOP;

LOOP Schleifen können geschachtelt werden. Nach dem Austritt aus der inneren Schleife mit EXIT wird nach dem Ende der inneren Schleife in der äußeren mit der Abarbeitung der Befehle fortgefahren.

Schleifenkonstrukte

Bedingte Schleifen

	FOR Schleife	feststehende Anzahl der Schleifendurchläufe
	WHILE Schleife	variable Anzahl der Schleifendurchläufe
	FOR CURSOR Schleife	zeilenweises Verarbeiten der Cursordaten

FOR variable IN (REVERSE) LB..UB LOOP ... END LOOP

```
FOR i IN 1..10 LOOP
```

```
  x:=i * 5;
```

```
  i:=11;      /* führt zu Fehler da Zuweisung an Schleifenvariable nicht erlaubt ist */
```

```
END LOOP;
```



In PL-SQL gibt es nur kopfgesteuerte Schleifen



Die FOR Cursor Schleife wird bei der Abarbeitung des Cursors behandelt.

Schleifenkonstrukte

WHILE bedingung LOOP ... END LOOP

DECLARE

durchlauf **BOOLEAN;**
durchlauf:=**TRUE;**

WHILE durchlauf LOOP

Anweisungen

durchlauf:=bedingung;

END LOOP;

An der Stelle von Durchlauf zwischen WHILE... LOOP stehen in der Regel irgendwelche boolschen Ausdrücke wie z. B. anzahl < 10

Achtung bei boolschen Ausdrücken ! Wenn ein Teil des Ausdrucks NULL ist verhält sich AND oder OR wie folgt !

AND	FALSE	TRUE	NULL
FALSE	FALSE	FALSE	FALSE
TRUE	FALSE	TRUE	NULL
NULL	FALSE	NULL	NULL

OR	FALSE	TRUE	NULL
FALSE	FALSE	TRUE	NULL
TRUE	TRUE	TRUE	TRUE
NULL	FALSE	TRUE	NULL

NOT	FALSE	TRUE	NULL
	TRUE	FALSE	NULL

Systeminterne Spalten

- `sequence_name.CURRVAL` Liefert den aktuellen Wert einer Sequenz
- `sequence_name.LEVEL` Bei Abfragen auf hierarchische Strukturen liefert LEVEL 1... für Wurzelknoten
2... für dessen Kinder usw.
- `sequence_name.NEXTVAL` Liefert den nächstfolgenden Wert einer Sequenz. (!!! Create Sequence)
- `NULL` Liefert den Wert NULL (kein Wert)
- `ROWID` Liefert eine eindeutige Zeilenadresse
- `ROWNUM` Liefert sequentielle Nummer für Ausgabe
1 gehört zur ersten Zeile die die WHERE Klausel erfüllt
- `SYSDATE` Systemdatum
- `UID, USER` Liefern interne Benutzernummern und -name

Ausführen von SQL-Statements

DECLARE

```
TYPE LieferRecType IS RECORD  
(liefer_nr      NUMBER  
  liefer_datum  DATE  
  liefer_ref    NUMBER);
```

```
liefer_rec      LieferRecType;  
ldatum         DATE;  
lref           NUMBER;
```

BEGIN

```
SELECT liefer_datum, liefer_ref INTO ldatum, lref  
FROM liefer_tab  
WHERE liefer_nr=1938;
```

```
SELECT liefer_nr, liefer_datum, liefer_ref INTO liefer_rec  
FROM liefer_tab  
WHERE liefer_knr=3454;  
END;
```

Ausführen von SQL-Statements

Prinzipiell kann jedes SQL-Statement in einem PL-SQL Programm auftauchen. Nicht nur SELECT sondern auch UPDATE, INSERT, DELETE. Eine Einschränkung ist bei einem SELECT Statement jedoch, dass das Ergebnis nur 1 Zeile lang sein darf. Die Behandlung von mehrzeiligen SELECT-Statements erfolgt mittels Cursor, der im nächsten Kapitel behandelt wird.

Ausführen von SQL-Statements

Fehler die bei der Ausführung eines SQL-Statements auftreten.

Ausnahmen die auftreten

Liefert eine SELECT Anweisung, die mit einer INTO Klausel versehen ist, mehr als einen Datensatz zurück, wird dir Ausnahme TOO_MANY_ROWS ausgelöst.

Wird kein Datensatz selektiert, ist der Inhalt der Variablen undefiniert und die Ausnahme NO_DATA_FOUND tritt auf.

Attribute(vollständig)

- ☞ **SQL%NOTFOUND** ist TRUE wenn die letzte SQL-Anweisung keine Datenzeile selektiert, eingefügt modifiziert oder gelöscht hat.
- ☞ Eine **SELECT Anweisung die keinen Datensatz findet** löst die Ausnahme NO_DATA_FOUND aus.
- ☞ **SQL%FOUND** ist TRUE, wenn **mindestens eine Datenzeile** behandelt wurde.
- ☞ **SQL%ROWCOUNT** liefert die **Anzahl** der in der letzten SQL-Anweisung verarbeiteten **Datenzeilen**.

Cursor

Deklaration eines Cursors

```
CURSOR cursorname [(parameter datatype[, parameter datatype]..)  
[IS query]
```

```
DECLARE  
CURSOR liefer_cur (lnrv NUMBER, lnrb NUMBER) IS  
SELECT *  
FROM liefer_tab  
WHERE liefer_nr BETWEEN lnrv AND lnrb;  
aktlnr NUMBER;  
oldlnr NUMBER;
```

```
BEGIN  
aktlnr:=10;  
oldlnr:=5;  
OPEN liefer_cur (10,20);  
OPEN liefer_cur (oldlnr, aktlnr);  
CLOSE liefer_cur;  
END;
```

Wie man hier sieht,
können einem Cursor
auch **Parameter**
mitgegeben werden!

Cursor

Die Verarbeitung des Cursors mit *LOOP ... EXIT... END LOOP* in Kombination mit *FETCH*, bei der der aktuelle Inhalt der gelesenen Zeile in eine **Struktur** des selben Typs gelesen wird kann mit der **FOR-CURSOR- Schleife** einfacher implementiert werden.

```
DECLARE
CURSOR c1 IS
    SELECT ename, sal, hiredate, deptno FROM emp;
....
BEGIN
    ...
    FOR emp_rec IN c1 LOOP
        ...
        salary_total := salary_total + emp_rec.sal;
    END LOOP;
END;
```

- Bei der FOR-Cursor Schleife geschieht das FETCH automatisch !
- Cursor wird automatisch geöffnet
- Erstellen eines Record in diesem Fall emp_rec (keine Deklaration nötig!)
- Schließen des Cursors bei Schleifenende

Cursor

Cursorattribute

<i>cursorname</i>.%NOTFOUND	ist TRUE wenn in der letzten fetch Anweisung nichts mehr gefunden wurde. Muß bei der FOR-Cursor Schleife nicht berücksichtigt werden.
<i>cursorname</i> %FOUND	ist TRUE , wenn bei der letzten FETCH Operation eine Datenzeile gelesen wurde.
<i>cursorname</i> %ISOPEN	ist TRUE wenn der Cursor bereits offen ist.
<i>cursorname</i> %ROWCOUNT	enthält die Anzahl der bereits gelesen Zeile. Achtung (1. Datensatz gelesen ... ROWCOUNT ist 0)

Anwendung von %rowcount

```
declare cur_for is select * from mitarbeiter order by mit_name;  
begin  
for c1_rec in cur_for  
  anzahl:=cur_for%rowcount;  
end loop;
```

Der Ausnahmebehandlungsteil

Wie wir bereits wissen erfolgt die Ausnahmebehandlung in PL-SQL ähnlich denen in ADA. Ausnahmen, ab jetzt heißen sie *exceptions*, müssen deklariert werden um als solche erkannt zu werden. Dies gilt nicht für vom System vordefinierte.

Exceptions sind entweder vordefiniert oder Äquivalente zu SQL Errorcodes (siehe PRAGMA EXCEPTION INT (name, nr);). Diese beiden Exceptionarten müssen während des Programmablaufs nicht berücksichtigt werden. Oracle 7.1 erledigt dies für uns. Sie werden lediglich im EXCEPTION PART (Ausnahmebehandlungsteil) abgearbeitet.

Viel interessanter sind hingegen die selbst definierten Exceptions. Sie werden deklariert, mittels RAISE aufgerufen meist nach Auswerten einer oder mehrerer Bedingungen und im Ausnahmebehandlungsteil bearbeitet.

Beispiel

```
declare
zuviel          exception;
max_op_cursors  exception;
pragma exception_init(max_op_cursors,-1000);
```

eigene Exception

maskierter
Oracle-
Fehlercode

Der Ausnahmebehandlungsteil

begin

.....

raise zuviel;

alle anderen Exceptions werden
automatisch behandelt

exception

when dup_val_on_index

when zuviel

when invalid_cursor or max_op_cursors

when others

then

then

then

then ...

alle Ausnahmen werden
hier behandelt

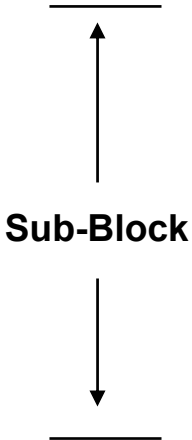
- Exceptions sind zwar global vom äußeren Block in seine Unterblöcke, werden sie jedoch überschrieben muß die Behandlung im Sub-Block erfolgen.
- Exceptions können in einem Block nur einmal deklariert werden.
- Werden sie in einem Sub-Block deklariert ist dies eine Art des Überschreibens, wobei es sich dann um eine andere Exception handelt auch wenn diese gleich heißt. Achtung ! Behandlung muß im Sub-Block erfolgen, sonst ist dies ein Fehler.
- Exeptions die in einem äußeren Block deklariert, nicht überschrieben und im Sub-Block aufgerufen wurden werden im Ausnahmebehandlungsteil des äußeren Blocks abgearbeitet.

Der Ausnahmebehandlungsteil

Beispiel aus der Oracle Hilfe

```
DECLARE
  past_due EXCEPTION;
  acct_num NUMBER;
BEGIN
  ...
  DECLARE
    past_due EXCEPTION; -- this declaration prevails
    acct_num NUMBER;
  BEGIN
    ....
    IF ... THEN
      RAISE past_due; -- this is not handled
    END IF;
    ...
  END;

EXCEPTION
  WHEN past_due THEN -- does not handle RAISED exception
    ...
END;
```



The diagram shows a vertical double-headed arrow between two horizontal lines, with the text "Sub-Block" centered between them, indicating the scope of the inner DECLARE and BEGIN/END blocks.

Procedures & Packages

- Allgemeines
- Anlegen einer Prozedur bzw. Funktion
- Aufrufen einer Prozedur bzw. Funktion
- Anlegen eines Pakets (Package) und dessen Körper (Body)
- Aufruf von Prozeduren aus einem Paket

Procedures & Packages

Bisher haben wir uns hauptsächlich mit PL-SQL Syntax und dem Aufbau von **anonymen PL-SQL Programmen** beschäftigt. Solche Programme können in SQL-Plus wie ganz gewöhnliche Abfragen gespeichert und ausgeführt werden.

Oracle 7.1 bietet aber noch andere Möglichkeiten der Speicherung an und erlaubt es PL-SQL-Programme zu parameterisieren. Dies sind **Prozeduren (Procedures)** und **Pakete (Packages)**, wobei **Packages** aus einem **Deklarationsteil** und einem Implementierungsteil bestehen und somit **Module** darstellen.

Prozeduren können auch **Werte zurückgeben**. Diese Prozeduren heißen **Funktionen**.

Funktionen und Prozeduren können auch gemischt in Pakete zusammengefaßt werden. Funktionen und Prozeduren können von jeder **beliebigen SQL-Sequenz** sowohl Statemen als auch PL-SQL Programm aufgerufen werden.

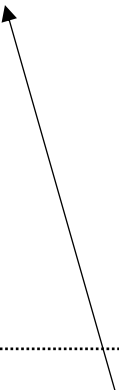
Ein Package ist vergleichbar mit einem Include-File, der Package-Body mit dem Implementierungsteil.

Der wesentliche Unterschied zwischen PL-SQL Prozeduren und anonymen PL-SQL-Blöcken ist die **Speicherung in der Datenbank und im Datenbankcache** nach dem ersten Aufruf.

Erstellen einer Prozedur

Erstellen einer Prozedur

```
CREATE PROCEDURE NEW_WORKER (Person_Name IN VARCHAR2)
AS
BEGIN
insert into WORKER (Name, Age, Lodging)
values (Person_Name, null, null);
END;
```



Parametertyp

Parametertypen

IN	Eingabeparameter (Call by Value)
OUT	Nur Ausgabeparameter
IN OUT	Ein- und Ausgabeparameter (ähnl. Call by Reference)

Erstellen einer Funktion

Erstellen einer Funktion

```
CREATE FUNCTION BALANCE_CHECK (Person_Name IN varchar2)
```

```
  RETURN NUMBER
```

Typ des
Rückgabewertes

```
  IS
```

```
    balance          NUMBER(10,2)
```

```
  BEGIN
```

```
    SELECT           SUM (decode(Action, 'BOUGHT', Amount,0))  
                    - SUM (decode(Action, 'SOLD', Amount, 0))
```

```
    INTO            balance
```

```
    FROM            ledger
```

```
    WHERE           Person=Person_Name;
```

```
  RETURN (balance);
```

```
END;
```

Zurückgeben
des Wertes

Ändern einer Funktion oder Prozedur

Wenn eine alte Version einer Prozedur oder Funktion **geändert** werden soll, dann genügt es nicht die Prozedur mit **CREATE** zu erstellen. Anstatt von CREATE wird **CREATE OR REPLACE** verwendet. Je nach dem **CREATE OR REPLACE PROCEDURE** oder **CREATE OR REPLACE FUNCTION**.

Überschreiben einer Prozedur

```
CREATE OR REPLACE PROCEDURE NEW_WORKER (Person_Name IN VARCHAR2)
AS
BEGIN
insert into WORKER (Name, Age, Lodging)
values (Person_Name, null, null);
END;
```

Ausführen einer Funktion oder Prozedur

Ausführen von der Kommandozeile

EXECUTE Mingeht (von, bis, mitarb_nr);

Der Aufruf einer Prozedur erfolgt in der Regel mit vollen Namen d. h. *ersteller_name.(paket.name).prozedurname(.....);*. Besser ist es jedoch gleich ein Synonym für die Prozedur zu vergeben.

Vergeben eines Synonyms

1) Ausführen der Prozedur
mike.my_procs.myname()

2) SQL Synonym
CREATE synonym mikesname
for mike.my_procs.myname()

3) Ausführen
mikesname();

Aufruf in PL-SQL Programmen

In ein PL-SQL-Programms wird eine **Prozedur** nur mit ihrem Namen aufgerufen und eine **Funktion** muß den Wert zurückgeben.

Package und Package Body

PL-SQL Pakete

Ein PL-SQL-Paket besteht in der Regel aus einem **Include File dem Package** und dessen **Implementierung dem Package Body**.

Die **Namen** von Package und Package Body müssen **gleich** sein.

Der **Package Body** beinhaltet **alle** Implementierungen der im Package **deklarierten Funktionen**.

Im **Package** deklarierte Funktion und Variablen sind **public**.

Im **Package Body** deklarierte Funktionen und Variablen sind **private**.

Zum Package Body gehört auch ein **Initialisierungsteil** der nach dem Compilieren des Packages **einmal ausgeführt** wird. Günstig für Erstinitialisierung von Variablen.

➡ **Package und Package Body werden unabhängig voneinander compiliert.**

Erstellen einer Package

Anlegen eines Packages

```
CREATE OR REPLACE PACKAGE LEDGER_PACKAGE
```

```
AS
```

```
    function BALANCE_CHECK (Person_Name IN VARCHAR2);
```

```
    function NEW_WORKER (Person_Name IN VARCHAR2);
```

```
END; -- oder END LEDGER_PACKAGE;
```

Es ist möglich nach dem **END** nach jeder Definition von Packages, Package Bodies, Functions und Procedures Namen anzugeben. Dies erhöht die Lesbarkeit des Codes.

Erstellen eines Package Bodies

```
CREATE OR REPLACE PACKAGE BODY LEDGER_PACKAGE  
AS
```

```
FUNCTION BALANCE_CHECK (Person_Name IN varchar2)  
  RETURN NUMBER  
IS  
  balance      NUMBER(10,2)  
BEGIN  
  SELECT      SUM (decode(Action, 'BOUGHT', Amount,0))  
               - SUM (decode(Action, 'SOLD', Amount, 0))  
  
  INTO        balance  
  FROM        ledger  
  WHERE       Person=Person_Name;  
  RETURN (balance);  
  EXCEPTION  
  WHEN NO_DATA_FOUND THEN  
    RAISE_APPLICATION_ERROR  
      (-20100, 'No bought and sold entries found');  
  
END BALANCE_CHECK;
```

Erstellen eines Package Bodies

```
PROCEDURE NEW_WORKER (Person_Name IN VARCHAR2)
AS
BEGIN
    INSERT INTO WORKER (Name, Age, Lodging)
    VALUES (Person_Name, null, null);
END NEW_WORKER;

END LEDGER_PACKAGE;
```

Fehler und Prozeduren

Anzeigen von Fehlern bei der Ausführung von PL-SQL Programmen und Prozeduren

Fehleranzeige

- show errors function <name>
- show errors procedure <name>
- select * from user_errors.

Beispiel

```
show errors procedure mikesname;
```

Anzeigen des Durchführungsplans (Execution Plan)

EXPLAIN PLAN

```
[SET STATEMENT ID = 'text']  
[INTO [schema.]table[@dblink]]  
FOR statement
```

Sicherstellen der Integrität

Integrität

Integrität ist gleichbedeutend mit **Korrektheit oder Widerspruchsfreiheit**.
Integrität bedeutet **Datenqualitätssicherung**.

Arten der Integrität

-  **Ablaufintegrität**
-  **Semantische Integrität**
-  **Entitätsintegrität**
-  **Referentielle Integrität**

Ablaufintegrität

Greifen mehrere Benutzer bzw. Prozesse gleichzeitig bzw. konkurrierend auf Datenbankobjekte zu, so muß sichergestellt werden, daß sich nach den Zugriffen die Datenbank abhängig von Zugriffsberechtigungen und -zeitpunkten wieder in einem korrekten Zustand befindet. Dies erledigt Oracle 7.1 selbst.

Sicherstellen der Integrität

Semantische Integrität

Es muß sichergestellt sein, daß die **Eingaben** die der Benutzer macht **korrekt** sind. In Oracle wird dies mittels *Constraints* realisiert.

Entitätsintegrität

Es muß gewährleistet sein, daß jeder **Datensatz** über ein oder mehrere Felder **eindeutig** identifiziert werden kann. Der **Primärschlüssel** in Oracle *PRIMARY KEY* wird durch **Constraints** sichergestellt.

Referentielle Integrität

Die **referentielle Integrität** stellt eine logische Verbindung über Feldinhalte zwischen abhängigen Tabellen dar und wird mit **Foreign Key und Primary Key realisiert**. Sie überwacht INSERT/UPDATE und DELETE für diese Tabellen mittels allgemein gültiger Definitionen. *PRIMARY* und *FOREIGN KEY* Constraint

Sicherstellen der Integrität

Constraints

Constraints sind Bedingungen die zu Spalten einer Tabelle zusätzlich definiert werden, nämlich der **Form**

gewöhnlich

CONSTRAINT Constraintname Constrainttyp Bedingung;

Constrainttypen sind

NOT NULL, UNIQUE, CHECK, DEFAULT

Von der gewöhnlichen Struktur verschieden

CONSTRAINT Constraintname **PRIMARY KEY** (Keyattribute);

CONSTRAINT Constraintname **FOREIGN KEY** (Attribut in dieser Tabelle)
REFERENCES (Attribut in der Partnertabelle);

Constraints werden bei jedem Ereignis (insert, update, delete) ausgelöst.

Constraints beziehen sich sofern sie **nicht direkt nach einem Attribut** geschrieben werden (**ohne Komma**) auf die **gesamte Tabelle**.

Semantische Integrität

Constraints für semantische Integrität

- NOT NULL **Spalte muß in jeder Zeile einen Wert enthalten**
- UNIQUE **Keine Zwei Zeilen einer Spalte können den selben Wert besitzen. (NULL ist möglich)**
- CHECK **Regeldefinition für eine Spalte**
z. B. $1000 \leq \text{Wert in der Spalte} \leq 2000$
- DEFAULT **Standardwert für Wert in der Spalte**

CREATE TABLE Teil

```
(TeileNr      CHAR (10), TBez      CHAR (30), TArt CHAR (10),  
  TDatum     DATE          DEFAULT SYSDATE,  
  TBestand   NUMBER  
  CONSTRAINT CHKTBestand CHECK TBestand >= TMinBestand,  
  TMinBestand NUMBER);
```

Wie aus dem Beispiel ersichtlich ist es nicht notwendig CONSTRAINTS zu benennen, macht die Verwaltung der Constraints jedoch einfacher. Man könnte die Zeile

TDatum DATE DEFAULT SYSDATE, auch als

TDatum DATE CONSTRAINT DEFTDatum DEFAULT SYSDATE, schreiben

Entitätsintegrität und Referentielle Integrität

Eigenschaften der Spalten

- **PRIMARY KEY** Jede Zeile wird durch den Inhalt dieser Spalten eindeutig identifiziert (impliziert NOT NULL)
- **FOREIGN KEY** Muß in beiden Tabellen, sowohl in der übergeordneten als auch in der abhängigen Tabelle berücksichtigt werden.

In der im vorigen Kapitel erstellten Tabelle fehlt sowohl der Primärschlüssel als auch mögliche Fremdschlüssel in anderen Tabellen. Zur Demonstration der Fremdschlüsselbehandlung wird eine zweite Tabelle Erzeugstruktur eingeführt.

Entitätsintegrität und Referentielle Integrität

Übergeordnete Tabelle

```
CREATE TABLE Teil
(TeileNr          CHAR (10),
TBez             CHAR (30),
TArt             CHAR (10),
    CONSTRAINT      PkTeil          PRIMARY KEY (TeileNr),

TDatum           DATE
    CONSTRAINT      DefTdatum      DEFAULT SYSDATE,

TBestand         NUMBER
    CONSTRAINT      CHKTBestand    CHECK TBestand >= TMinBestand,

TMinBestand      NUMBER,
    CONSTRAINT      FKErzStruktur  FOREIGN KEY (TeileNr)
                                REFERENCES ErzStruk (UTeil));
```

Entitätsintegrität und Referentielle Integrität

Abhängige Tabelle

CREATE TABLE *ErzStruk*

(OTeil **CHAR (10),**
CONSTRAINT FKOTeil

FOREIGN KEY (OTeil)
REFERENCES *Teil*(TeileNr),

UTeil **CHAR (30),**
CONSTRAINT FKUTeil

FOREIGN KEY (UTeil)
REFERENCES *Teil*(TeileNr),





Menge **NUMBER,**

CONSTRAINT PKErzStruktur **PRIMARY KEY** (OTeil, UTeil));

Zu beachten ist, daß die **Syntax des FOREIGN KEY Constraint** in beiden Tabellen die **selbe** ist, jedoch die **Attribute vertauscht** sind.

Verwaltung der Constraints

Operationen auf Constraints

-  Disable
-  Enable
-  Add
-  Drop

Diese Aktionen werden MIT ALTER TABLE ausgeführt, denn ein Constraint kann für den Fall der Modifikation als Spalte betrachtet werden.

```
ALTER TABLE Teil DISABLE CONSTRAINT CHKTBestand;  
ALTER TABLE Teil ENABLE CONSTRAINT CHKTBestand;  
ALTER TABLE Teil DROP CONSTRAINT CHKTBestand;  
ALTER TABLE Teil ADD  
CONSTRAINT CHKTBestand CHECK TBestand >= TMinBestand,
```

Beispiel

Tabellen

lieferant:={lnr (Primary Key), firma (Unique) }

bestellung={bnr (Primary Key), lnr (!=0) }

lieferposition={bnr, lpnr, lnr}

Bedingungen für Tabellen

- ❶ Wird ein Lieferant gelöscht sollen seine zugehörigen Bestellungen gelöscht werden.
- ❷ Wird eine Bestellung gelöscht, so soll dies auch an die Tabelle Lieferposition weitergeleitet werden.
- ❸ Es darf kein Lieferant gelöscht werden, solange noch offene Posten für ihn vorhanden sind

Beispiel

```
CREATE TABLE lieferant (  
  lnr NUMBER(5) CONSTRAINT pk_lnr PRIMARY KEY,  
  firma VARCHAR2 (50) NOT NULL CONSTRAINT unq_Firma UNIQUE);
```

```
CREATE TABLE bestellung (  
  bnr NUMBER (5) CONSTRAINT pk_bnr PRIMARY KEY,  
  lnr NUMBER(5) NOT NULL  
    CONSTRAINT fk_lnr REFERENCES lieferant(lnr) ON DELETE CASCADE);
```

 Bestellungen werden durch

```
CONSTRAINT fk_lnr REFERENCES lieferant(lnr) ON DELETE CASCADE);
```

gelöscht, wobei *lieferant*(lnr) der Primärschlüssel der Tabelle Lieferant ist.

Beispiel

```
CREATE TABLE lieferposition(  
  bnr NUMBER(5)  
    CONSTRAINT fk_bnr REFERENCES bestellung(bnr) ON DELETE CASCADE,  
  lpnr NUMBER(3),  
  lnr NUMBER(5) CONSTRAINT fk_lieferpos_lnr REFERENCES lieferant(lnr) ,  
  CONSTRAINT pk_bnr_lpnr PRIMARY KEY (bnr, lpnr));
```

- Wird eine Bestellung gelöscht, so wird dies an Lieferposition mit
CONSTRAINT fk_bnr REFERENCES *bestellung(bnr)* ON DELETE CASCADE,
weitergeleitet.
- Nichtlöschen des Lieferanten bei offenen Lieferposten wird durch
lnr **NUMBER(5) CONSTRAINT fk_lieferpos_lnr REFERENCES *lieferant(lnr)* (*restricted*),**
realisiert.
- bnr + lpnr ... sind gemeinsam der Primärschlüssel

Referentielle Integrität

Der **Foreign Key** besteht aus einer oder mehreren Spalten, die auf den Parent Key verweisen.

Der **Parent Key** ist ein UNIQUE oder PRIMARY KEY, der von einem Foreign Key der gleichen oder einer anderen Tabelle referenziert wird.

Die **abhängige Tabelle** enthält FOREIGN KEYS.

Die **übergeordnete Tabelle** wird von abhängigen Tabellen referenziert.

Referentielle Integrität

RESTRICTED

verhindert das löschen von Parent Key Werten, solange noch ein zugehörige Zeile existiert.

SET TO NULL

Wird Parent Key geändert oder gelöscht, werden alle zugehörigen Werte in der abhängigen Tabelle auf NULL gesetzt.

SET TO DEFAULT

Wie SET TO NULL jedoch zurücksetzen auf Standardwerte

ON UPDATE CASCADE

Wird der Parent Key geändert werden alle zugehörigen Werte in der Abhängigen Tabelle auf den gleichen Wert geändert.

ON DELETE CASCADE

Wird eine Zeile mit dem Parent Key gelöscht, so werden alle zugehörigen Zeilen der abhängigen Tabelle gelöscht.

Trigger

Trigger sind eine spezielle Form von **PL-SQL Prozeduren**. Sie werden nicht durch expliziten Aufruf, sondern durch das **Eintreten** eines bestimmten **Ereignisses** ausgeführt. Ein Trigger kann sich auf die Ereignisse **insert**, **update**, **delete** beziehen und **vor** oder **nach** dem Ereignis. Sie dienen genauso wie CONSTRAINTS zur Sicherung der Integrität.

Beispiele für den Einsatz von Triggern

- ☞ Über Trigger kann sichergestellt werden, daß bestimmte Benutzer nur zwischen 08:00 und 16:00 Uhr Daten eingeben.
 - ☞ Trigger werden nur bei den Aktionen aufgeführt für die sie definiert sind. CONSTRAINTS hingegen bei allen Ereignissen. Alle **Ereignisse** sind **insert**, **update**, **delete**.
 - ☞ Mit Triggern können zur Integritätssicherung verteilte Datenbestände eingesetzt werden.
- ➞ Trotzdem soll versucht werden die Integrität über Constraints zu sichern.

Trigger

Anlegen eines Trigger

Triggerkomponenten

- ① Trigger-Name
- ② Trigger-Zeitpunkt
- ③ Trigger-Ereignis
- ④ Trigger-Typ
- ⑤ Trigger-Restriktion
- ⑥ Trigger Rumpf

entsprechende Syntax

```
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 Block
```

Prinzipiell ist ein Trigger **befehlsorientiert** und wird **einmal** zum entsprechenden Zeitpunkt ausgeführt. Ein **zeilenorientierter Trigger** (for each row) wird bei **jedem zu bearbeitenden Datensatz** zum entsprechenden Zeitpunkt ausgeführt. **Wird FOR EACH ROW nicht angegeben ist der Trigger ein befehlsorientierter Trigger.** Bei zeilenorientierten Triggern ist es möglich mit :OLD- bzw. :NEW.spaltenname auf die alten bzw. neuen Werte zuzugreifen.

Beispiel zu Trigger

Beispiel

```
CREATE OR REPLACE TRIGGER Teil_LegalEntryTime  
BEFORE DELETE OR INSERT OR UPDATE ON Teil  
WHEN (USER!='SYSTEM')  
DECLARE  
    not_a_legal_entry EXCEPTION;  
BEGIN  
    IF TO_CHAR (SYSDATE, 'HH24:MI') NOT BETWEEN '08:00' AND '17:00' THEN  
        RAISE not_a_legal_entry;  
    EXEPTION  
    WHEN not_a_legal_enty THEN  
        RAISE APPLICATION_ERROR (-20000, 'Entries only between 08:00 and 17:00');  
END;
```

Zusammentreffen von Constraint und Trigger

- ➊ Nachdem ein Update,- Insert- oder Delete Befehl abgesetzt wurde, werden zunächst sämtliche Sperren und Kennungen gesetzt.
- ➋ Danach werden befehlsorientierte Before-Trigger ausgeführt.
- ➌ Für jede angesprochene Zeile wird
 - ➊ der jeweilige zeilenorientierte Before-Trigger ausgeführt
 - ➋ die Datenänderung durchgeführt und anhand der definierten Constraints überprüft sowie
 - ➌ der jeweilige zeilenorientierte After-Trigger ausgeführt.
- ➍ Dann wird der befehlsorientierte After Trigger ausgeführt.
- ➎ Die Abarbeitung wird durch ein nochmaliges Prüfen der in den Constraints definierten Integritätsbedingungen und das Aufheben der Sperren und Kennungen beendet.

Wird während des Ablaufs ein Fehler festgestellt und daher zu irgend einem Zeitpunkt abgebrochen, wird der Ablauf mit einem ROLLBACK beendet.