Crash-Kurs PL/SQL

Martin Landvoigt

Martin@Landvoigt-Net.de
0171-1493 497

Übersicht

- Einführung: Warum Code in der DB?
- Ein Beispiel ...
- Die Umgebung: SQLPLUS, TOAD, Aqua Data Studio...

- Objekte mit PL/SQL:
  - Anonyme Codeblöcke
  - Stored Procedures, Functions und Packages / Trigger
- Die Sprache im Überflug:
  - Variablen / Blöcke und Exceptions
  - Operatoren / Kontrollstrukturen
  - Einbindung von SQL / Cursor etc.
  - E/A – Das schräge Output Modell

Einführung: Warum Code in der DB?

- Exklusive Funktionen: Manches geht nur hier
- Funktionale Kapselung
- Performance-Steigerung
- Applikationsentwicklung
  - Web-Anwendungen
  - Oracle Forms
- Security Implementierung
- Pflege von Altanwendungen

Ein Beispiel ...

- Replikation / Kopplung einer Zentralen DB mit Notebook-Instanzen
- Erstellung von DB-Objekten
- Fehlerbehandlung
- Automatische Erstellung von User-Accounts / Abgleich mit Zentrale
- Dynamisches Kopieren von beliebigen Tabelen

Die Umgebung: (1)

- Oracle DB-Server 7.3 – 10g
- Oracle Client / Oracle Net
  - Konfiguration TNSNAMES.ORA
- SQLplus
  - Command-Line und Windows-Vers.
  - Erweiterter Command Set
  - Batch-Prozessor
- Andere Front-Ends (Fortsetzung)

Die Umgebung (2)

- TOAD, SQL Worksheet, SQL Navigator
  - Beliebte Entwicklungs-Umgebungen
  - Editor, Sonderfunktionen …
- Aqua Data Studio
  - Java / kostenlos (privat & educational)
  - für viele DBMS verfügbar
  - erleichtert Entwicklung
  - Besonderheit: GO

Objekte mit PL/SQL

- Anonyme Codeblöcke
  - Skripte
  - Installation / Test einmal-Programme
- Stored Procedures & Functions
  - DB-Objekte mit und ohne Rückgabe …
  - Functions auch in SQL-Selects …
- Packages
  - Strukturierung mit Variablen / Security ..
- Trigger
  - Werden bei Ereignissen in Tabellen gestartet

Die Sprache im Überflug: Variablen

- Ähnliche Datentypen wie in ORACLE DB:
  - Char und Varchar2 (bis 32 KBytes)
  - Number (BCD mit bis zu 38 Stellen)
  - Date
  - Long
- Zusätzliche Datentypen:
  - BINARY_INTEGER
  - PLS_INTEGER
  - Boolean

Char Variablen (1)

- Char [(<Länge>)]
  - Zeichenkette fester Länge (in Bytes)
  - Maximale Länge 32767 Bytes
  - Maximale Länge in der Datenbank 255 Bytes (ab Oracle8: 2000 / Oracle 9: 4000)
  - Ggf. mit Leerzeichen aufgefüllt
- VARCHAR2 [(<Länge>)]
  - Zeichenkette variabler Länge (in Bytes)
  - Maximale Länge 32767 Bytes
  - Maximale Länge in der Datenbank 2000 Bytes (ab Oracle8: 4000)
  - Subtypen: STRING, VARCHAR

Vergleich zwischen CHAR und VARCHAR2

Bei Zuweisungen werden CHAR-Variablen mit Leerzeichen aufgefüllt. Das führt dazu, daß bei Vergleichen von CHAR-Werten nachfolgende Leerzeichen unberücksichtigt bleiben. Da in VARCHAR2 nachfolgende Leerzeichen so erhalten bleiben, wie sie zugewiesen werden, führt ein Vergleich zu anderen Ergebnissen:

- declare
  s1 char(5) := '12'; -- ohne nachfolgende Leerzeichen
  s2 char(5) := '12 '; -- mit nachfolgenden Leerzeichen
  => s1 = s2 ist TRUE
- declare
  s1 varchar2(5) := '12'; -- ohne Leerzeichen
  s2 varchar2(5) := '12 '; -- mit Leerzeichen
  => s1 = s2 ist FALSE

Beim Einfügen von Werten in die Datenbank, werden CHAR - Spalten mit Leerzeichen aufgefüllt, VARCHAR2-Spalten nicht. Sie behalten ihren Wert, der ihnen zuvor zugewiesen wurde. Nachfolgende Leerzeichen bleiben erhalten.

CHAR-Werte benötigen für jedes definierte Zeichen Speicherplatz. VARCHAR-Werte benötigen für jedes verwendete Zeichen Speicherplatz.

Beispiel:

- s char(100) := '1234567890' 100 Bytes
- s varchar2(100) := '1234567890' 10 Bytes

Numerische Datentypen

- **NUMBER [(<Länge>, <Nachkommastellen>)]**
  - Numerische Werte von 1.0E-129 .. 9.99E125
  - maximale Länge 38
  - Nachkommastellen von -84 bis 127
    => automatische Rundung
  - Beispiel: NUMBER (5,2)
    => drei Vorkomma- und zwei Nachkommastellen
  - Subtypen: DECIMAL, INTEGER, REAL, SMALLINT
- **BINARY_INTEGER**
  - Ganze Zahlen von -2147483647 bis 2147483647
  - Verbraucht weniger Speicherplatz als NUMBER
  - Subtypen: NATURAL, NATURALN, POSITIVE, POSITIVEN
- **PLS_INTEGER**
  - Ganze Zahlen von -2147483647 bis 2147483647
  - Verbraucht weniger Speicherplatz als NUMBER
  - Schneller als BINARY_INTEGER und NUMBER

Deklaration von Variablen

Zuweisungen von Werten zu Variablen direkt bei der Deklaration erfolgen wie folgt:

```
declare
  v_zahl1 number(5) := 5;
  v_zahl2 number(5) DEFAULT 5;
```

Soll eine Variable zur Laufzeit niemals den Wert NULL annehmen dürfen, so ist dies bei der Deklaration anzugeben:

```
declare
  v_zahl1 number(5) NOT NULL := 5;
  v_zahl2 number(5) NOT NULL DEFAULT 5;
```

Die Zuweisung von NULL führt dann zu einem Laufzeitfehler.

Die Default-Werte sind bei der Deklaration zwingend erforderlich.

Deklaration von Konstanten

Beispiel:

```
declare
  v_zahl1 constant number(5) := 5;
  v_zahl2 constant number(5) DEFAULT 5;
```

Zuweisungen sind nicht möglich

Default-Werte sind zwingend erforderlich

Typ - Referenzierung

Unter Typ - Referenzierung versteht man die Deklaration einer Variablen in Abhängigkeit von einer Typ - Definition einer Tabelle oder Spalte in der Datenbank. Der Operator %TYPE referenziert einen Spaltentyp, %ROWTYPE einen Recordtyp aus einer Tabelle oder einem View.

Beispiel:

```
declare
  v_bezeichnung    ARTIKEL.BEZEICHNUNG%TYPE;
  v_auftrag_pos    AUFTRAG_POS%ROWTYPE;
begin
  select * into v_auftrag_pos
    from auftrag_pos
    where auftrag_nr = 8 and pos = 1;
  select bezeichnung into v_bezeichnung
    from artikel
    where artikel_nr = v_auftrag_pos.artikel_nr;
end;
```

Blöcke und Exceptions

Prinzip:

- **DECLARE**
  - Deklarationsteil
- **BEGIN**
  - Programmteil
- **EXCEPTION**
  - Ausnahmebehandlung
- **END;**

Der Start eines PL/SQL - Blocks wird mit dem Zeichen / zu Beginn der nächsten Zeile eingeleitet. Dabei wird der Block als Ganzes von der Datenbank verarbeitet. Der Deklarationsteil und die Ausnahmebehandlung sind optional. Es ist eine beliebige Schachtelung von Blöcken möglich.

Kommentare werden mit -- bis zum Ende der Zeile eingeleitet. Geklammerte /* und */ fügen mehrzeilige Kommentare ein.

Blöcke und Exceptions - Beispiel:

```
DECLARE
  v_einheit_kurz varchar2(10);
  v_bezeichnung varchar2(40);
BEGIN
  v_einheit_kurz := 'kg';
  v_bezeichnung := 'Kilogramm';
  insert into einheit (einheit_kurz, bezeichnung)
    values (v_einheit_kurz, v_bezeichnung);
EXCEPTION when DUP_VAL_ON_INDEX then
  /* Der Datensatz ist schon da
     => Bezeichnung neu setzen */
  update einheit set
    bezeichnung = v_bezeichnung
    where einheit_kurz = v_einheit_kurz;
END;
```

- Das Beispiel fügt in die Tabelle Einheit einen Datensatz ein. Existiert bereits eine Einheit mit der Kurzbezeichnung 'kg', so wird nur der Bezeichnungstext geändert. Dies erfolgt jedoch nur, weil EINHEIT_KURZ der Primärschlüssel ist.

Procedure in PL/SQL

- **PROCEDURE name [(parameter [, parameter])] IS**
- Deklarationsteil
- **BEGIN**
- Programmteil
- **EXCEPTION**
- Ausnahmebehandlung
- **END;**

Der prinzipielle Aufbau einer Prozedur entspricht dem eines PL/SQL - Blocks. Die Parameterliste ist optional. Falls Parameter definiert werden, so besteht die Definition aus der Parametervariablen, der Parameterart und dem Datentyp. Die Länge der Datentypen wird nicht mit angegeben, also z.B. varchar2 und nicht varchar2(80).

Es existieren, abhängig von der jeweiligen PL/SQL - Version, Einschränkungen bei der Verwendung von Datentypen als Parameter. So ist es in PL/SQL - Version 1 nicht möglich zusammengesetzte Typen zu übergeben.

Parameter

Arten von Parametern

- IN Nur Eingabe (DEFAULT)
- OUT Nur Ausgabe
- IN OUT Ein- und Ausgabe, call by value and result

DEFAULT-Parameter

Beispiel:

- procedure p (a in number, b in varchar2 default '0') is ...

Der Parameter b ist optional, d.h. er muß beim Aufruf der Prozedur nicht angegeben werden. In diesem Fall erhält b Wert '0'.

Parameterübergabe

- Anhand der Position: p (10, 'hallo')
- Anhand der Namen:
  p (b => 10, a => 'hallo') oder
  p (a => 'hallo', b => 10)

Es werden Parameter normalerweise call by Value referenziert!

Ausnahme: **NOCOPY**

Beispiel für eine Prozedur

```
declare
    v_anzahl pls_integer;
procedure neue_einheit (
        p_einheit_kurz in varchar2,
        p_bezeichnung in varchar2) is
 begin
   insert into einheit
     (einheit_kurz, bezeichnung) values
     (p_einheit_kurz, p_bezeichnung);
 end;

begin
  select count(*)into v_anzahl
    from einheit
    where einheit_kurz = 'm';
    if v_anzahl = 0 then
      neue_einheit ('m', 'Meter');
    end if;
end;
```

Funktionen in PL/SQL

- **FUNCTION name [(parameter [, parameter])] RETURN type IS typ**
- Deklarationsteil
- **BEGIN**
- Programmteil
- **RETURN …**
- **EXCEPTION**
- Ausnahmebehandlung
- **END;**

Der Aufbau einer Funktion entspricht dem einer Prozedur mit der Ausnahme, daß Funktionen immer ein Ergebnis zurückliefern. Deshalb muß der Programmteil stets mit einem RETURN - Befehl beendet werden.

Beispiel für eine Funktion

```
declare
    function einheit_existiert (p_einheit_kurz in varchar2)
      return boolean is
    v_anzahl pls_integer;
    begin
      select count(*)into v_anzahl
        from einheit
        where einheit_kurz = p_einheit_kurz;
      return v_anzahl > 0;
    end;
begin
  if not einheit_existiert ('m') then
    insert into einheit
          (einheit_kurz, bezeichnung) values (m', 'Meter');
  end if;
end;
```

Vergleichsoperatoren

=	gleich
<>, ~=, !=, ^=	ungleich
<	kleiner
>	größer
<	kleiner oder gleich
>=	größer oder gleich
IS NULL	hat den Pseudo-Wert NULL
IS NOT NULL	hat nicht den Pseudo-Wert NULL
LIKE	Zeichenkettenvergleich mit Platzhaltern '%' und '_'
NOT LIKE	Umkehrung von LIKE
BETWEEN	Kurzschreibweise für <= und >=
IN	Mengenvergleich

Vergleiche mit NULL liefern immer das Ergebnis FALSE
Platzhalter '%' steht für beliebige Zeichenkette (auch leer)
Platzhalter '_' steht für genau ein beliebiges Zeichen

Beispiel für BETWEEN: if a between 1 and 10 then ...
Beispiel für IN: if a in (1,2,3,4) then ...

Weitere Operatoren

AND	logisches und
OR	logisches oder
NOT	logisches nicht
*	Multiplikation
**	Exponent
/	Division
+	Addition bzw. positives Vorzeichen
-	Subtraktion bzw. negatives Vorzeichen
||	Konkatenation zweier Zeichenketten

Prioritäten der Operatoren in Ausdrücken:

höchste Priorität NOT, **
+, - (als Vorzeichen)
/, *
+, -, ||
=, !=, <, >, <=, >=, IS NULL,LIKE,BETWEEN,IN
AND
geringste Priorität OR

Kontrollstrukturen: Zuweisung

- **Variable := Ausdruck;**

Beispiel:

- ```
  declare
    a number(5);
    b boolean;
  begin
    a := (3 + 5) / 8;
    b := a = 5;
      ...
    b := TRUE;
  end;
  ```

Man betrachte die folgende Zuweisung:

- a := b + c;

Die Variable a wird nach der Zuweisung den Wert NULL enthalten, wenn entweder b oder c den Wert NULL enthalten!
  ```

Bedingte Verzweigung

if … then … (elsif) … (else) … end if;

- if a = 5 then
  …
  elsif a = 6
  …
  else
  …
  end if;

Zwischen **then** und **end if** muß mindestens ein Befehl stehen (oder der leere Befehl: NULL;)!

Sprung (goto)

Definition von Sprungmarken:

- <<Sprungmarke>>

Eine Sprungmarke muß unmittelbar vor einem ausführbaren Befehl definiert werden (also nicht end, end if ...).

GOTO - Befehl

- GOTO Sprungmarke

Beispiel:

```
...
if not berechnen then
  goto ende;
end if;
...
<<ende>>
...
```

Einschränkungen

Eine GOTO - Anweisung kann nur zu einem Label verzweigen, das auf der gleichen oder einer höheren Verarbeitungsebene steht. So ist es z.B. nicht erlaubt, von außen in eine Verzweigung, Schleife oder Block zu springen.

Unbedingte Schleife (loop)

- **loop … end loop;**

Es handelt sich generell um eine Endlos - Schleife. Sie muß mit dem EXIT - Befehl verlassen werden.

Schleifenmarken

Um bei verschachtelten Schleifen angeben zu können, welche Schleife durch einen EXIT - Befehl verlassen werden soll, setzt man vor der Schleife eine entsprechende Sprungmarke und gibt sie beim EXIT - Befehl an. Das führt im folgenden Beispiel zum Verlassen beider Schleifen.

-
```
begin
  << aeussere_schleife >>
  loop
    ...
    loop
      ...
      exit aeussere_schleife;
    end loop
  end loop aeussere_schleife;
end;
```

WHILE - Schleife

In einer WHILE - Schleife wird eine Bedingung spezifiziert. Die Schleife wird dann so lange durchlaufen, bis diese Bedingung erfüllt ist. Die Bedingung wird vor jedem Schleifendurchlauf erneut überprüft.

- **while <boolscher Ausdruck> loop**
  ...
  end loop

FOR - Schleife

- **for Zählvariable in <linke Grenze>..<rechte Grenze> loop**
  ...
  end loop;

Die Zählvariable kann nur ganze Zahlen annehmen, wobei die Schrittweite immer eins beträgt. Rückwärts laufende Schleifen erfordern stattdessen die Angabe IN REVERSE statt IN.

Innerhalb der Schleife kann die Zählvariable nicht das Ziel einer Zuweisung sein. **ACHTUNG: Sie überdeckt eine bereits existierende Variable gleichen Namens in dem Block, in dem sich die Schleife befindet, d.h. der Wert der Zählvariablen ist nach dem END LOOP nicht verfügbar. Die Zählvariable wird nicht im DECLARE - Teil des PL/SQL - Blocks oder im Deklarationsteil einer Prozedur definiert. Die Deklaration erfolgt durch die Schleife selbst.**

Die Grenzen werden zur Laufzeit einmalig vor der Schleife berechnet, d.h. die Grenzen sind während der Schleifendurchläufe nicht veränderbar. Um eine FOR - Schleife vorzeitig zu verlassen, steht der EXIT - Befehl zur Verfügung.

For Each wird später mit Cursor Verarbeitung genannt.

Schleifen - EXIT - Befehl

Der EXIT - Befehl bewirkt das Verlassen der aktuellen Schleife. Es wird nach dem END LOOP - Befehl fortgefahren. Der Befehl ist nur im Schleifen - Kontext nutzbar, es ist nicht möglich, z.B. eine verschachtelte IF - Abfrage oder einen Block mit dem EXIT - Befehl zu verlassen. Dazu dient das GOTO - Statement.

EXIT WHEN - Befehl

- EXIT WHEN <boolscher Ausdruck>;

Die Schleife wird nur verlassen, wenn eine Bedingung erfüllt ist. Es ist dies eine Kurzschreibweise für:

- if <boolscher Ausdruck> then
  exit;
  end if;

Einbettung von SQL in PL/SQL

Prinzip

Da PL/SQL die prozedurale Erweiterung von SQL darstellt, können in PL/SQL fast alle Möglichkeiten von SQL benutzt werden. Alle SQL-DML - Befehle (select, insert, update, delete) sind direkt in PL/SQL verfügbar. Dabei dienen PL/SQL - Variablen dem Datenaustausch zwischen SQL und PL/SQL. Weiterhin ist die Verwendung der Befehle zur Transaktionssteuerung (commit, rollback ...) möglich.

DDL - Befehle müssen über dynamisches SQL durch Verwendung der Standard Package DBMS_SQL realisiert werden. (… Parse …) oder

EXECUTE IMMEDIATE – führt konstruierte Strings, auch DDL, aus

Alle SELECT-Befehle müssen eine INTO-Klausel enthalten, die definiert, welche Variablen das Ergebnis aufnehmen sollen.

Einbettung von SQL in PL/SQL

Beispiel:

```
declare
  v_bezeichnung einheit.BEZEICHNUNG%TYPE;
begin
  select bezeichnung into v_bezeichnung
        from einheit
        where einheit_kurz = 'm';
    ...
  insert into einheit (einheit_kurz, bezeichnung)
        values ('l', 'Liter');
  ...
  update auftrag set
        hinweis = hinweis || 'Neuer Artikel ' || v_bezeichnung;
end;
```

Cursor - Definition

- Cursor sind Datenstrukturen im Arbeitsspeicher, die für die Abarbeitung von SQL - Befehlen benötigt werden. Für jeden SQL - Befehl wird automatisch ein entsprechender Speicherbereich allokiert (impliziter Cursor).

- Für SELECT - Befehle, die mehrere Zeilen liefern, muß ein expliziter Cursor verwendet werden.

Cursor - Beispiel:

```
declare
  v_summe        number := 0;
  v_pos_preis    number;
  cursor c_auftrag_pos is
         select anzahl*preis from auftrag_pos;
begin
  OPEN c_auftrag_pos;
  loop
    FETCH c_auftrag_pos into v_pos_preis;
    exit when c_auftrag_pos%NOTFOUND;
    v_summe := v_summe + v_pos_preis;
  end loop;
  CLOSE c_auftrag_pos;
end;
```

Cursor – Parameter Beispiel

```
declare
  v_summe       number := 0;
  v_pos_preis   number;
  cursor c_auftrag_pos
         (p_auftrag_nr in auftrag_pos.auftrag_nr%TYPE) is
         select anzahl*preis
           from auftrag_pos
           where auftrag_nr = p_auftrag_nr;
begin
  OPEN c_auftrag_pos (4711);
  loop
    FETCH c_auftrag_pos into v_pos_preis;
    exit when c_auftrag_pos%NOTFOUND;
    v_summe := v_summe + v_pos_preis;
  end loop;
  CLOSE c_auftrag_pos;
end;
```

Cursor - Strukturvariablen -Beispiel

```
declare
  v_summe          number := 0;
  v_auftrag_pos    auftrag_pos%ROWTYPE;
  cursor c_auftrag_pos
         (p_auftrag_nr in auftrag_pos.auftrag_nr%TYPE) is
         select *
           from auftrag_pos
           where auftrag_nr = p_auftrag_nr;
begin
  OPEN c_auftrag_pos (4711);
  loop
    FETCH c_auftrag_pos into v_auftrag_pos;
    exit when c_auftrag_pos%NOTFOUND;
    v_summe := v_summe + v_auftrag_pos.anzahl *
                         v_auftrag_pos.preis;
  end loop;
  CLOSE c_auftrag_pos;
end;
```

Cursor - Attribute

Jeder Cursor hat vier Statusattribute, auf die mit cursor%ATTRIBUT zugegriffen wird.

Bei impliziten Cursorn erfolgt der Zugriff mit SQL%ATTRIBUT.

%OPEN steht bei impliziten Cursorn nicht zur Verfügung.

Um den aktuellen Datensatz eines expliziten Cursors mit einem UPDATE-Befehl zu ändern, kann man die **CURRENT OF - Klausel** in der WHERE-Bedingung verwenden.

- **%FOUND**
  Gibt an, ob der letzte FETCH - Befehl einen Satz gefunden hat => TRUE
  Vor dem ersten Fetch NULL
- **%NOTFOUND**
  Gibt an, ob der letzte FETCH - Befehl einen Satz gefunden hat => FALSE
  Vor dem ersten Fetch NULL
- **%ROWCOUNT**
  Liefert die Anzahl der mit FETCH gelesenen Zeilen
  Vor dem ersten FETCH auf 0
- **%ISOPEN**
  Gibt an, ob ein Cursor geöffnet ist

Beispiel mit CURRENT OF

```
declare
  v_auftrag    auftrag%ROWTYPE;
  cursor c_auftrag
          (p_auftrag_nr in auftrag_pos.auftrag_nr%TYPE) is
          select *
            from auftrag
            where auftrag_nr = p_auftrag_nr;
            for update of status;
begin
  OPEN c_auftrag (4711);
  loop
    FETCH c_auftrag into v_auftrag;
    exit when c_auftrag_pos%NOTFOUND;
    if v_auftrag.status = 'A' then
      bearbeite_auftrag (v_auftrag);
      update auftrag set status = 'E'
              where current of c_auftrag;
    end if;
  end loop;
  CLOSE c_auftrag;
end;
```

Cursor - FOR - Schleifen

- Durch eine Cursor - FOR - Schleife wird ein (impliziter) Cursor automatisch geöffnet, alle Sätze der Ergebnismenge des zugehörigen SELECT - Befehls gelesen und der Cursor wieder geschlossen. Die Befehle OPEN, FETCH und CLOSE sind dann nicht notwendig.

- Die Schleifenvariable hat den Recordtyp des Cursors.

Cursor - FOR - Beispiel 1:

```
declare
  v_summe    number := 0;
  cursor c_auftrag_pos
        (p_auftrag_nr in auftrag_pos.auftrag_nr%TYPE) is
        select *
          from auftrag_pos
          where auftrag_nr = p_auftrag_nr;
begin
  for v_auftrag_pos in c_auftrag_pos(4711) loop
    v_summe := v_summe + v_auftrag_pos.anzahl *
                              v_auftrag_pos.preis;
  end loop;
end;
```

Cursor - FOR - Beispiel 2:

- Impliziter Cursor

```
declare
  v_summe number := 0;
begin
  for v_auftrag_pos in (select * from auftrag_pos) loop
    v_summe := v_summe + v_auftrag_pos.anzahl *
                                v_auftrag_pos.preis;
  end loop;
end;
```

Das schräge Output Modell

PL/SQL hat keine eingebauten E/A Anweisungen. Diese werden über SQL oder über Packages realisiert:

- **dbms_output**
- utl_file

Bildschirmausgaben in SQL*Plus oder SQL Worksheet einschalten:

- set serveroutput on
- Mit dbms_output in den Puffer schreiben

Eine Zeile schreiben:

- dbms_output.put_line (a in varchar2);
- dbms_output.put_line (a number);
  - begin
    dbms_output.put_line ('Dies ist ein Test!');
    end;

Die dbms_output - Puffergröße einstellen

- Die Defaultgröße beträgt 20.000 Bytes. Maximum ist 1.000.000 Bytes.
- Beispiel: Einstellen der dbms_output-Puffergröße auf 1.000.000:
- dbms_output.enable (1000000);

Exceptions

- Tritt zur Laufzeit ein Fehler auf, so wird eine **Exception** ausgelöst. Jeder Exception ist eine Datenbank - Fehlernummer zugeordnet.

- Durch den **Exception Handler** kann im Programm auf das Auslösen einer Exception reagiert werden. Für die wichtigsten Fehlerfälle existieren eine Reihe von **vordefinierten Exceptions**. Darüber hinaus können **weitere Exceptions** frei definiert werden.

- Das Auslösen einer Exception führt **immer** zum Verlassen des **aktuellen Blocks**. Soll der Fehlerzustand an den übergeordneten Block weitergegeben werden, so muß dieses mit dem Befehl **RAISE** erfolgen.

- Tritt ein Laufzeitfehler **ohne zugehörigen Exception Handler** auf, so wird der aktuelle Block mit dem Fehlerzustand verlassen und die Fehlerbehandlung erfolgt **von neuem im übergeordneten Block**.

Exceptions - Beispiel

```
declare
    ...
  begin
    ...

    exception when OTHERS then
      dbms_output.put_line ('Fehler:' || sqlerrm);
  end;
```

Vordefinierte Exceptions

Exception Name	Fehlernr.	Beschreibung
CURSOR_ALREADY_OPEN	ORA-06511	Cursor bereits geöffnet
DUP_VAL_ON_INDEX	ORA-00001	Schlüssel doppelt eingetragen
INVALID_CURSOR	ORA-01001	Cursor nicht geöffnet
INVALID_NUMBER	ORA-01722	bei impliziter Typkonvertierung
LOGIN_DENIED	ORA-01017	Benutzername oder Paßwort falsch
NO_DATA_FOUND	ORA-01403	SELECT liefert kein Ergebnis
NOT_LOGGED_ON	ORA-01012	Keine Verbindung zur Datenbank
PROGRAM_ERROR	RA-06501	Interner PL/SQL - Fehler
ROWTYPE_MISMATCH	ORA-06504	Strukturvariablen inkompatibel
STORAGE_ERROR	ORA-06500	Speicherprobleme
TIMEOUT_ON_RESOURCE	ORA-00051	Datenbank-Sperre
TOO_MANY_ROWS	ORA-01422	SELECT liefert mehr als eine Zeile
VALUE_ERROR	ORA-06502	meistens: Zeichenkette zu kurz definiert
ZERO_DIVIDE	ORA-01476	durch 0 geteilt

Verschachtelung von Exceptions

Eine Exception gilt für den Block, in dem sie definiert ist oder wenn der Fehler in einem Subblock auftritt und dort nicht behandelt wird. Der Fehlerzustand wird mit dem Befehl RAISE weitergegeben.

Beispiele:

- ```
  begin
    declare
      a number(1);
    begin
      a := 55;
    end;
  exception when VALUE_ERROR then
    ...
  end;
  ```
- ```
  begin
    declare
      a number(1);
    begin
      a := 55;
    exception when VALUE_ERROR then
      raise VALUE_ERROR;
    end;
  exception when VALUE_ERROR then
    ...
  end;
  ```

User Defined Exceptions

Beispiel:

```
declare
  v_bezeichnung          einheit.bezeichnung%TYPE;
  EINHEIT_FEHLER    EXCEPTION;

begin
  select bezeichnung into v_bezeichnung
         from einheit
         where einheit_kurz = 'm';
  if v_bezeichnung <> 'Meter' then
    raise EINHEIT_FEHLER;
  end if;
exception when EINHEIT_FEHLER then
  ...
end;
```

Zuordnung von Datenbank - Fehlernummern zu Exceptions

Über **PRAGMA** - Anweisungen können Datenbank - Fehlernummern Exceptions über die bereits vordefinierten hinaus zugeordnet werden. Alle Oracle - Fehlernummern können hier verwendet werden. Dabei stehen die Fehlernummern -20000 bis -20999 zusätzlich für die Programmierung frei zur Verfügung.

Der Befehl **raise_application_error** erzeugt einen Fehlerzustand unter Angabe der Fehlernummer und des Fehlertextes. Dies ist zur Verwendung der freien Fehlernummern -20000 bis -20999 gedacht.

Beispiel:

```
declare
  v_bezeichnung    einheit.bezeichnung%TYPE;
  EINHEIT_FEHLER   EXCEPTION;
  PRAGMA EXCEPTION_INIT (EINHEIT_FEHLER, -20100);
begin
  select bezeichnung into v_bezeichnung
           from einheit
           where einheit_kurz = 'm';
  if v_bezeichnung <> 'Meter' then
    raise_application_error (-20100, Bezeichnung falsch');
  end if;
exception when EINHEIT_FEHLER then
  ...
end;
```

PL/SQL-Tables (Arrays)

PL/SQL-Tables sind dynamische eindimensionale Arrays. Die Indizierung der Elemente erfolgt über beliebige ganze Zahlen (max. 10 Stellen, Typ BINARY_INTEGER).

Es ist keine sequentielle Indizierung notwendig. So lassen sich wahlfreie Zugriffe realisieren, in dem z.B. eine Auftragsnummer als Index verwendet wird, um Daten eines Auftrags als Element des Arrays abzulegen.

Als Elementtypen sind alle PL/SQL-Typen zugelassen. Zur Deklaration einer Array-Variablen muß zunächst die Deklaration eines Typs erfolgen, die das Array beschreibt:

```
DECLARE
TYPE table_typ IS TABLE OF datentyp [ NOT NULL ]
        INDEX BY BINARY_INTEGER;
variable table_typ;
```

PL/SQL-Tables Beispiele

```
DECLARE

  TYPE auftrag_table_typ IS TABLE OF auftrag%ROWTYPE
                INDEX BY BINARY_INTEGER;

  auftrag_table auftrag_table_typ;

  TYPE auftrag_bearbeitet_table_typ IS TABLE OF boolean
                INDEX BY BINARY_INTEGER;

  auftrag_bearbeitet_table  auftrag_bearbeitet_table_typ;
```

Verwendung von PL/SQL - Tables

Indizierung

- variable(index) := wert;
- wert := variable(index);

Beispiele

- auftrag_bearbeitet_table (4711) := TRUE;
- if not auftrag_bearbeitet_table (4711) then ...
- select * into auftrag_table (4711)
  from auftrag
  where auftrag_nr = 4711;

Ein Element wird dadurch erzeugt, dass ihm ein Wert zugewiesen wird. Der Zugriff auf ein nicht vorhandenes Element führt zu einer Exception NO_DATA_FOUND.

Jedes Element muß einzeln erzeugt werden. Das komplette oder teilweise Laden von Datenbank-Tabellen in eine PL/SQL-Table ist seit möglich 9i über Bulk-Operationen. Jede Zeile musste bis dahin einzeln über einen Fetch in die Table geschrieben werden. Allerdings gibt es Einschränkungen bei Bulk-Operationen.

PL/SQL - Table - Attribute (1)

EXISTS(n)

- Prüft, ob ein Element existiert.

Beispiel:

```
if not auftrag_bearbeitet_table.exists(4711) then
  auftrag_bearbeitet_table (4711) := FALSE;
end if;
```

COUNT

- Zählt die Elemente der Table.

Beispiel:

```
v_anzahl := auftrag_bearbeitet_table.count;
```

FIRST und LAST

- Bestimmt den ersten bzw. letzten Index der Table.
- Liefert NULL bei einer leeren Table

Beispiel:

```
v_start := auftrag_bearbeitet_table.first;
```

PL/SQL - Table - Attribute (2)

PRIOR(n) und NEXT(n)

- Bestimmt zu einem Index den nächsten bzw. vorherigen Index, zu dem ein Element existiert. Liefert NULL, wenn kein solches Element existiert.

Beispiel:

- v_naechster := auftrag_bearbeitet_table.next (v_start);

DELETE, DELETE(n), DELETE(n,m)

- Löschen aller Elemente einer Table (ohne Parameter)
- Löschen eines Elementes einer Table (ein Parameter)
- Löschen eines Bereiches von Elementen einer Table (zwei Parameter)

Beispiel:

- auftrag_bearbeitet_table.delete (0,100);
  Löscht alle Elemente mit dem Index 0 bis 100.

Alle Elemente einer PL/SQL-Table durchlaufen

```
declare
  TYPE auftrag_table_typ IS TABLE OF auftrag%ROWTYPE
                INDEX BY BINARY_INTEGER;
  auftrag_table auftrag_table_typ;
  v_index BINARY_INTEGER;

begin
  v_index := auftrag_table.first;
  while v_index is not NULL loop
    bearbeiten des Elementes auftrag_table(v_index)
    v_index := auftrag_table.next (v_index);
  end loop;
end;
```

Packages und andere PL/SQL-Objekte

Anzeige einer Stored Procedure, Function oder Package

- desc [FUNCTION | PROCEDURE | PACKAGE | PACKAGE BODY] ...
- select text
  from user_source
  where name = ' PROZEDUR_TEST'
  order by line;

Anzeige aller Stored Procedures, Functions und Packages

- select object_name, status
  from user_objects
  where object_type = 'PROCEDURE'
  order by object_name;
- select object_name, status
  from user_objects
  where object_type = 'FUNCTION'
  order by object_name;
- select object_name, status
  from user_objects
  where object_type like 'PACKAGE%'
  order by object_name;

Status INVALID

Hat eine Prozedur, Funktion oder Package den Status "invalid", so ist das Objekt zwar in der Datenbank abgelegt, es ist aber nicht übersetzt. Der nächste Aufruf führt automatisch zum erneuten Übersetzen. Es ist möglich, dass das Programm Fehler enthält oder ein abhängiges Objekt geändert wurde.

Manuelles erneutes Übersetzen:

- alter procedure prozedur_name compile;
- alter function funktions_name compile;
- alter package package_name compile;
- alter package body package_name compile;

Bestimmen des Status:

- select * from user_objects where status = 'INVALID';

Verwendung von Stored Functions in SQL

Eine Stored Function kann prinzipiell in SQL-Befehlen verwendet werden. Es gibt jedoch eine Reihe von Einschränkungen, die von den Seiteneffekten der Funktion abhängig sind. Funktionen aus Packages müssen zur Verwendung in SQL immer eine PRAGMA RESTRICT_REFERENCES-Anweisung enthalten.

Einschränkungen

- Die Funktion darf keine UPDATE, INSERT oder DELETE-Befehle enthalten
- Funktionen, die Package-Variablen verwenden, können nicht remote oder parallel ausgeführt werden
- Nur Funktionen, die über SELECT-Befehle aufgerufen werden, können Package-Variablen schreiben
- Die Parameter und der Rückgabewert müssen Datenbanktypen besitzen
- OUT und IN OUT-Parameter sind nicht erlaubt

Packages - Details

- Eine Package ist eine Zusammenfassung von PL/SQL-Objekten. Sie ist unterteilt in Header und Body. Beide Teile sind getrennte Datenbank-Objekte. Der Header enthält den öffentlichen Teil (Spezifikation), der von außen sichtbar ist. Der Body ist der private Teil (Programm). In ihm werden die im Header deklarierten Prozeduren und Funktionen implementiert.

Vorteile von Packages

- Modularität
- Information Hiding
- Persistente Variablen und Cursor

Nachteil:

- Ein erneutes Übersetzen einer Package kann zu Problemen führen. Enthält die Package globale Variablen oder Cursor, so verlieren diese Variablen ihren Wert. Das führt entweder zu einem Laufzeitfehler für alle Sessions, die diese Package verwenden oder zu einer erneuten Initialisierung. Ggf. müssen die Anwender die aktuelle Session beenden und neu starten.

Package Header

CREATE [OR REPLACE] PACKAGE package_name AS

öffentliche Deklarationen von:

- Typen
- Variablen
- Konstanten
- Cursors
- Exceptions
- Funktionen
- Prozeduren

END package_name;

- Die öffentlich deklarierten Objekte können mit der Punkt-Notation direkt angesprochen werden. Die Variablen und Cursor, die im Header deklariert sind, sind persistent für jede Session, d.h. sie sind während der gesamten Dauer einer Session gültig.

Package Body

CREATE [OR REPLACE] PACKAGE BODY package_name AS

private Deklarationen von:

- Typen
- Variablen
- Konstanten
- Cursors
- Exceptions

Implementierung der öffentlichen und privaten Prozeduren und Funktionen

- [BEGIN]
- Initialisierungsblock
- END package_name;

Die privat deklarierten Objekte sind ebenfalls persistent für jede Session. Sie sind jedoch von außen nur über Prozeduren und Funktionen indirekt verwendbar. Der Initialisierungsblock wird einmalig beim ersten Zugriff ausgeführt. Die Köpfe der öffentlich deklarierten Prozeduren und Funktionen müssen im Body und Header identisch sein.

Overloading

Es ist möglich, Prozeduren und Funktionen mit dem selben Namen, aber unterschiedlichen Parametern zu deklarieren. Anhand der Aufruf-Parameter wird dann automatisch die richtige Prozedur/Funktion bestimmt.

Beispiel:

```
create package ovl as
  procedure ausgabe (p in varchar2);
  procedure ausgabe (p in boolean);
end;
create package body ovl as
  procedure ausgabe (p in varchar2) is
  begin
    dbms_output.put_line ('Ausgabe => ' || p);
  end;
  procedure ausgabe (p in boolean) is
  begin
    if p then
      dbms_output.put_line ('Ausgabe => TRUE');
    else
      dbms_output.put_line ('Ausgabe => FALSE');
    end if;
  end;
end;
```

Verwendung von Package-Funktionen in SQL

Zur Verwendung einer Package-Funktion in SQL ist ein Pragma RESTRICT_REFERENCES zu deklarieren. Damit erfolgt die Definition der Einschränkungen der Funktion manuell (im Gegensatz zu Stored Functions). Die Pragma-Anweisung steht im Package Header nach der Deklaration der entsprechenden Funktion.

- PRAGMA RESTRICT_REFERENCES (funktions_name, WNDS [, WNPS] [, RNDS] [, RNPS])
  - WNDS writes no database state
  - WNPS writes no package state
  - RNDS reads no database state
  - RNPS reads no package state

Beispiel:

- create or replace package perf as
- procedure start_timer;
  procedure stop_timer;
  function get_time return number; -- Zeit in Sekunden
  pragma restrict_references (get_time, WNDS, WNPS, RNDS);
- end perf;

Standard Packages

- Mit Oracle wird eine große Anzahl an Packages, die ihrerseits jeweils n Objekte beinhalten. Z.B. Replikation, Piping u.v.m
- Standard Packages sind unter dem User SYS abgelegt.
- Mit einem Objekt-Browser und der Dokumentation können diese genutzt werden

Security

- Alle DB-Objekte liegen in einem Schema, normalerweise der Owner
- Die Ausführungsrechte von Statements werden zu Compile-Time ermittelt.

- Der Owner hat alle Rechte und kann andere User und Rollen berechtigen.
  - GRANT EXECUTE ON pack_x TO mla;

- Zur Runtime gilt nur das Execute Recht.

Weiterführende Literatur

- Original Doku von Oracle
- http://www.oracle.com/technology/index.html
- Bücher von Steven Feuerstein
- http://de.wikipedia.org/wiki/PL/SQL
- Oracle PL/SQL Users Guide
- Tutorial für PL/SQL
- Tutorial für PL/SQL auf Deutsch
- Oracle FAQ: knowledge base PL/SQL
- Ask Tom