ORACLE9i: PL/SQL

Autoren: *Günter Preuner*

Thomas Thalhammer

Aktualisierte Fassung in Anlehnung an "Oracle 7: PL/SQL"

Folien von Stefan Rausch-Schott und Werner Retschitzegger, IFS

PL/SQL
Inhalt

Einführung
Schwächen von SQL

- **interaktives SQL** für viele DB-Applikationen **nicht ausreichend**
  - nur sequentielle Abarbeitung von SQL-Anweisungen
  - weder Schleifen noch bedingte Verzweigung
  - keine Berechnungsvollständigkeit
  - keine Modularisierung
- **PL/SQL** = prozedurale Erweiterung von SQL; **kombiniert** deklarative **Abfragesprache** mit prozeduraler **Programmiersprache**
  - Ausführung komplexer DB-Operationen innerhalb des DB-Servers
  - Kontrolle muß nicht nach jeder SQL-Operation an das Anwendungsprogramm zurückgegeben werden

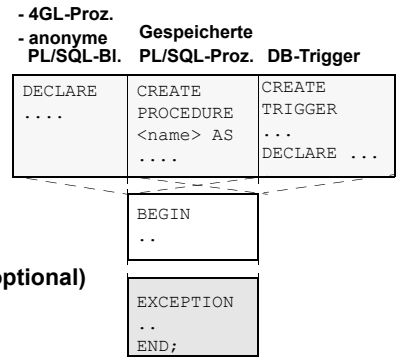
Einführung
Sprachkonstrukte

- **blockorientierte Sprache** - jeder Block best. aus
  - **Deklarationsteil (optional)**

    Variablen/Konstanten-Deklarationen
    explizite Cursor-Definitionen
    benutzerdefinierte Ausnahmen
  - **Ausführungsteil**

    eigentliches Programm
  - **Fehler- und Ausnahmebehandlungsteil (optional)**

    Bearbeitung von Fehlersituationen durch „exception handler"



Deklarationsteil
Variablen und Konstanten

- Deklaration von - Name
  - Datentyp (explizit bzw. implizit)
  - Vorbelegung (bei Variablen optional)

- **Variablendeklaration**

```
<var_name> <type> [[NOT NULL] :=|DEFAULT <expr>]
```

```
name_pls    VARCHAR2(50);
ok BOOLEAN  DEFAULT FALSE;
saldo_pls   NUMBER(5) NOT NULL := 0;
```

- **Konstantendeklaration**

```
<const_name> CONSTANT <type> :=|DEFAULT <expr>
```

```
wert CONSTANT NUMBER(8) DEFAULT 100;
pi   CONSTANT REAL := 3.14159;
```


Deklarationsteil
Elementare Datentypen

Datentyp	Subtyp	Beispiel	Wertebereich

CHAR
VARCHAR
VARCHAR2
 	
LONG
		text_1 CHAR(50);
		text_2 VARCHAR(80);
		text_3 VARCHAR2(120);
		
		t_long LONG;
			1 .. 2000 bytes
			1 .. 4000 bytes
			
			1 .. 2^{31}-1 bytes

NUMBER	 DECIMAL FLOAT INTEGER REAL SMALLINT	zahl_1 NUMBER (P, S); dez_zahl DECIMAL; komma_zahl FLOAT; ganz_zahl INTEGER; real_zahl REAL; kleine_zahl SMALLINT;	p = 1 .. 38, s = -84 .. 127 b = 1 .. 126 (bits) NUMBER(38) FLOAT(18)
BOOLEAN **DATE** **RAW** **LONG RAW** **ROWID**		flag1 BOOLEAN; geb_dat DATE; m_daten RAW; m_daten LONG RAW; zeile ROW_ID;	TRUE, FALSE 4712 v. Chr. - 4712 n.Chr. 1 .. 2000 bytes 1 .. 32760 bytes 10 bytes (18 Hex.-Ziffern)

Deklarationsteil
Implizite Datentypdefinition

- **Deklaration über bereits definierten Datentyp**

  eines Attributs einer Tabelle/View

```
name_var1   mitarbeiter.mit_name%type;
```

  einer bereits deklarierten Variable/Konstante

```
name_var2   name_var1%type;
```

- **Deklaration über Datensatzformat**

  einer Tabelle

```
ds_mit      mitarbeiter%rowtype;
```

  eines bereits deklarierten Cursors

```
CURSOR c1 IS SELECT ...;
ds_mit      c1%rowtype;
```


Deklarationsteil
Zusammengesetzte Datentypen (1)

Deklaration von **PL/SQL-Tabellen** und **PL/SQL-Records** in zwei Schritten

1. Deklaration des gewünschten zusammengesetzten Datentyps
2. Variablendeklaration mit Hilfe des zuvor deklarierten Typs

- **PL/SQL-Tabelle**

  = eindimensionales Array eines skalaren Datentyps
  (Index Ganzzahl oder Zeichenkette)

```
TYPE text_tab_type IS TABLE OF VARCHAR2(80) INDEX BY BINARY_INTEGER;
text text_tab_type;
abc  text_tab_type;
```

  - **Zugriff** auf einzelne Elemente über ganzzahligen (ev. negativen) Index

```
text(1) := 'PL/SQL ist die integrierte ';
text(2) := 'Datenbanksprache von ORACLE';
```

- falls einem Element noch kein Wert zugewiesen wurde, existiert es noch nicht
  -> bei Lesezugriff Fehlersituation NO_DATA_FOUND

Deklarationsteil
Zusammengesetzte Datentypen (2)

- **PL/SQL-Record**
  - besteht aus **beliebigen Elementen beliebiger Datentypen**
  - als Komponenten sind **skalare** als auch **zusammengesetzte DT** erlaubt

```
TYPE mit_rec_typ IS RECORD (mit_nameVARCHAR2(30),
  mit_geb       DATE,
  mit_abtnr     NUMBER(8));

mit_rec mit_rec_typ;
```

  - **Zugriff** auf einzelne Komponenten **über Pfade**

```
mit_rec.mit_geb := TO_DATE('01-feb-93');
```

  - Zuweisung eines Records auf Record gleichen Typs erlaubt
  - Zuweisung eines Records auf Record anderen Typs auch bei exakter Übereinstimmung aller Komponenten nicht möglich

Ausführungsteil
Anweisungen

- **erlaubte Anweisungen**
  - beliebige **DML-Anweisungen** mit Ausnahme von `EXPLAIN PLAN`

    `SELECT, INSERT, UPDATE, DELETE, LOCK TABLE, ...`
  - Anweisungen für die **Transaktionskontrolle**

    `COMMIT, ROLLBACK, SAVEPOINT, ...`
  - alle **SQL-Funktionen** inkl. aggregierender Funktionen
  - **SQL-Pseudospalten**

    `CURRVAL, LEVEL, NEXTVAL, ROWID, ROWNUM`
- **DDL-Anweisungen nicht erlaubt**

Ausführungsteil
PL/SQL-Kontrollstrukturen

- **Zuweisungen**

Zuweisungsausdruck	Beispiel
Konstanten	`var0 := 10`
Variablen	`var1 := var2`
arithmetische Ausdrücke	`var3 := var1*faktor`
SQL-Funktionen	`var4 := DECODE(var5, 10, 100)`
gespeicherte Funktionen	`var5 := meine_funk(var1)`
Bedingungen	`var6 := (gehalt > 3500)`

- **Bedingte Verarbeitung**

```
IF datum < '26-may-93' THEN
  INSERT INTO tab VALUES (...);
ELSIF name LIKE '%er' THEN
  a := vor_name;
ELSIF name LIKE 'A%' THEN
  a := name;
ELSE
  abc (datum, name, vor_name, status);
END IF;
```


Ausführungsteil
Schleifenkonstrukte

- **Unbedingte Schleife**

```
LOOP
   ...
   [EXIT [WHEN <bedingung>];]
   ...
END LOOP;
```

- **FOR-Schleife**

```
FOR <zaehler> IN [REVERSE] <start> .. <ende> LOOP
   ...
END LOOP;
```

  <zaehler> kann innerhalb der Schleife nur gelesen, nicht verändert werden;

- **WHILE-Schleife**

```
WHILE <bedingung> LOOP
   ...
END LOOP;
```

Cursorkonzept

- **Cursor = Handle auf privaten Arbeitsbereich** einer SQL-Anweisung („Private SQL Area“)
- **Impliziter Cursor**

  für jede DDL/DML-Anweisung einschließlich SELECT, das nur eine einzige Zeile liefert

  wird automatisch vom DB-Server definiert, wenn benötigt
- **Expliziter Cursor**

  für mengenorientierte Abfragen, die mehr als eine Zeile liefern

  ermöglicht **zeilenweise Abarbeitung** des Abfrageergebnisses

  muss im Programm explizit deklariert werden

Cursorkonzept
Vorgangsweise

4 Schritte

- **Deklaration** eines Cursors
- **Öffnen** des Cursors

  veranlasst Ausführung der SQL-Anweisung im Server, ohne jedoch Daten in den Programmbereich zu transferieren
- **Übertragen** von Datensätzen aus der DB in den Programmbereich

  mit dem SQL-Befehl `FETCH` wird der jeweils nächste Datensatz übertragen und spezifizierten Variablen zugeordnet
- **Schließen** des Cursors

  definiertes Beenden der Leseoperation und Freigeben aller Ressourcen

Cursorkonzept
Deklaration

- Deklaration von - **Name**
  - **Eingabeparameter** (optional)
  - (beliebig komplexe) **SELECT-Anweisung**

```
CURSOR c1 (name_var VARCHAR2) IS
  SELECT mit_name, mit_geb, mit_abtnr
  FROM mitarbeiter
  WHERE mit_name LIKE name_var
  ORDER BY mit_name;
```

- Spezifikation von **Default-Werten** für Eingabeparameter möglich

```
CURSOR c2 (low INTEGER DEFAULT 0, high INTEGER DEFAULT 99) IS
...;
```

- generell **keine Beschränkung für Anzahl** der aktiven Cursor pro PL/SQL-Programm bzw. pro Session;
  max. Anzahl offener Cursors über DB-Parameter `MAX_OPEN_CURSORS` definierbar

Cursorkonzept
Cursor-Variablen

zur Überprüfung des Status eines Cursors

Variable	Wert
`%FOUND`	‘wahr‘, wenn letzte `FETCH`-Op. Datensatz gefunden hat
`%NOTFOUND`	‘wahr‘, wenn letzte `FETCH`-Op. keinen Datensatz gefunden hat
`%ROWCOUNT`	Anzahl bereits gefundener Datensätze
`%ISOPEN`	‘wahr‘, wenn der Cursor geöffnet ist

- bei **expliziten** Cursorn werden diese an den Cursornamen angehängt

```
FETCH c1 INTO ...
IF c1%FOUND THEN ...
```

- auch bei **impliziten** Cursorn verwendbar, obwohl kein Cursorname vorhanden

```
UPDATE mitarbeiter
SET mit_gehalt = mit_gehalt * 1.05
WHERE mit_abtnr = 40;
```

Cursorkonzept
Cursor-Verarbeitung (1)

```
DECLARE
   ...
   CURSOR C1 IS SELECT * FROM mitarbeiter
      ORDER BY mit_name;
   c1_rec C1%ROWTYPE;

BEGIN
   ...
   OPEN C1;
   ...
   LOOP
    FETCH C1 INTO c1_rec;
    EXIT WHEN C1%NOTFOUND;
    ...
   END LOOP;
   ...
   anzahl := C1%ROWCOUNT;
   CLOSE C1;
   ...
```


Cursorkonzept
Cursor-Verarbeitung (2)

- **FOR-Cursor-Schleife**

  implizite Durchführung von
  - Erstellung einer Record-Datensatzstruktur gemäß Cursor-Deklaration
  - Öffnen des Cursors
  - Übertragen der Datensätze aus der DB in die Record-Struktur
  - Schließen des Cursors und Beenden der Schleife

```
DECLARE
   CURSOR c1 IS SELECT * FROM mitarbeiter
                ORDER BY mit_name;
   ...
BEGIN
   FOR c1_rec IN c1 LOOP
      ...
      anzahl := c1%ROWCOUNT;
      ...
   END LOOP;
   ...
```


Ausnahmebehandlung
Arten von Ausnahmen

- **Systemdefinierte Ausnahmen** müssen nicht deklariert werden
- **Benutzerdefinierte Ausnahmen**

```
max_open_cursors   EXCEPTION;
pl_stop            EXCEPTION;
zuviel             EXCEPTION;
zuwenig            EXCEPTION;
```

Aktivierung durch `RAISE ...` im Ausführungsteil

Zuordnung von **ORACLE-Fehlercodes** möglich („Ausnahmeinitialisierung“)
-> verhalten sich dadurch wie systemdefinierte Ausnahmen

```
PRAGMA EXCEPTION_INIT (max_open_cursors, -1000);
```


Ausnahmebehandlung
Systemdefinierte Ausnahmen

Ausnahme /Fehlername	ORACLE-Fehlernummer
cursor_already_open	ora-06511
dup_val_on_index	ora-00001
invalid_cursor	ora-01001
invalid_number	ora-01722
login_denied	ora-01017
no_data_found	ora-01403
not_logged_on	ora-01012
program_error	ora-06501
storage_error	ora-06500
timeout_on_resource	ora-00051
too_many_rows	ora-01422
transaction_backed_out	ora-00061
value_error	ora-06502
zero_divide	ora-01476

Ausnahmebehandlung
„Exception Handler“ (1)

- Definition von **Exception-Handlern**, die eine oder mehrere Fehlersituationen bearbeiten

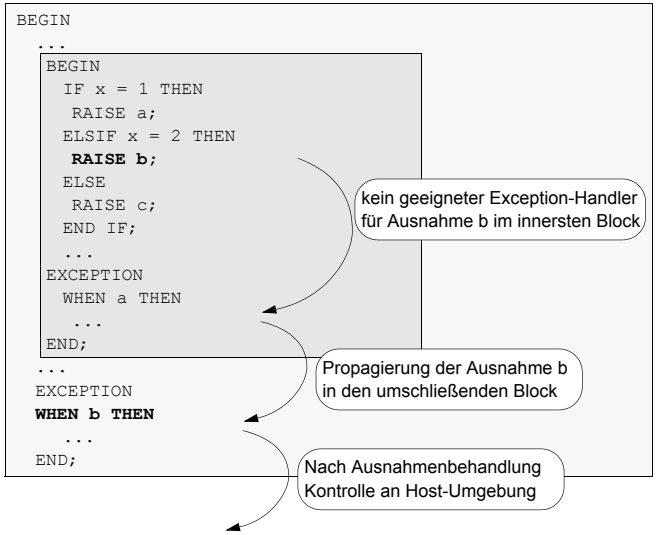
  dadurch einheitliche Behandlung aller Arten von Ausnahmen

  `WHEN <ausnahme_name> THEN <PL/SQL-Anweisungen>`

  Name von - systemdefinierter Ausnahme
  - benutzerdefinierter Ausnahme

- Fehlersituationen, für die kein entsprechender Exception-Handler definiert ist, benutzen den `OTHERS`**-Exception-Handler**, falls definiert
- Ausnahmen, die von keinem Exception-Handler behandelt werden, werden **an den nächsten übergeordneten PL/SQL-Block propagiert**
- **nach Ausführung** des Exception-Blocks wird PL/SQL-Block verlassen und mit der **1. Anweisung des übergeordneten Blocks** fortgefahren

Ausnahmebehandlung
„Exception Handler“ (2)



Ausnahmebehandlung
SQLCODE u. SQLERRM

- Innerhalb des Ausnahmebehandlungsteils kann auf **Funktionen** `SQLCODE` **u.** `SQLERRM` zurückgegriffen werden, die ORACLE-Fehlercode und zugehörigen Fehlertext liefern

  - `SQLCODE = 0` ... kein Fehler
  - `SQLCODE < 0` ... Fehler; Anweisung nicht ausgeführt
  - `SQLCODE > 0` ... Warnung: `100` = `NO_DATA_FOUND` (entspricht internem ORACLE-Code `ORA-1403`)

- diese Funktionen sind **nicht direkt in SQL-Anweisungen verwendbar**

```
EXCEPTION
   ...
   WHEN OTHERS THEN
    err_no := sqlcode;
    err_text := sqlerrm;
    INSERT INTO fehler(f_nr, f_text) VALUES(err_no, err_text);
```


Ausnahmebehandlung
raise_application_error

- Auslösen **benutzerdefinierter Fehlercodes** und -meldungen und Rückgabe an den aufrufenden Block
- **Fehlersituation** wird durch `raise_application_error (errc, errmsg)` **ausgelöst**
- Fehlercode benutzerdefiniert, muss zwischen -20000 und -20999 liegen

```
SELECT sal INTO curr_sal FROM emp WHERE empno = emp_id;
IF curr_sal IS NULL THEN
   raise_application_error(-20101, 'Salary is missing');
```

- kann nur in gespeicherten Prozeduren/Funktionen verwendet werden

Gespeicherte PL/SQL-Programme
Prozeduren - Funktionen - Pakete

- **Prozeduren und Funktionen**
  - Speicherung in kompilierter Form in der DB
  - bei Aufruf Ausführung im DB-Server bessere Performanz
- **Pakete**
  - zur Zusammenfassung mehrerer Prozeduren/Funktionen in einem Modul
  - ebenfalls Speicherung in der DB
  - Kapselung von DB-Operationen, zugehörigen Datenstrukturen und Zugriffsprivilegien

Gespeicherte PL/SQL-Programme
Prozeduren und Funktionen (1)

Definition einer gespeicherten Prozedur/Funktion durch DDL-Anweisungen
`create procedure` bzw. `create function`

- **PL/SQL-Compiler** prüft Syntax und Semantik
- **SQL-Anweisungen werden geparst**
  (Prüfen von Objekt-Namen, Spaltennamen, Zugriffsberechtigungen)
- **Sourcecode** u. **übersetztes PL/SQL-Programm** werden im DD (Source in View `user_source`) **gespeichert** (auch bei fehlerhafter Kompilierung)
- **Fehler** werden im DD in View `user_errors` **protokolliert**
- Ermittlung der Fehlerursache (alternativ)
  - in SQL*PLUS `SHOW ERRORS FUNCTION/PROCEDURE <funcname/procname>`
  - in SQL `SELECT * FROM user_errors WHERE name = <proc_name>`

Gespeicherte PL/SQL-Programme
Prozeduren und Funktionen (2)

```
CREATE [OR REPLACE] PROCEDURE <p_name>
    (<p1> [IN|OUT|IN OUT] <datatype>, ...)
AS
   [<lokale Deklarationen>]
   BEGIN
    <PL/SQL-Code>
   [EXCEPTION
    <exception handler>]
   END;

CREATE [OR REPLACE] FUNCTION <f_name>
    (<p1> [IN|OUT|IN OUT] <datatype>, ...)
    RETURN <datatype>
AS
   [<lokale Deklarationen>]
   BEGIN
    <PL/SQL-Code>
   [EXCEPTION
    <exception handler>]
   END;
```

default

Gespeicherte PL/SQL-Programme
Prozeduren und Funktionen (3)

```
CREATE PROCEDURE raise_salary (emp_id INTEGER, increase REAL) AS
   current_salary REAL;
   salary_missing EXCEPTION;
BEGIN
   SELECT sal INTO current_salary FROM emp WHERE empno = emp_id;
   IF current_salary IS NULL THEN
    RAISE salary_missing;
   ELSE
    UPDATE emp SET sal = sal + increase WHERE empno = emp_id;
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
    INSERT INTO emp_audit VALUES (emp_id, 'keine solche Angestellten#');
   WHEN salary_missing THEN
    INSERT INTO emp_audit VALUES (emp_id, 'Gehalt ist NULL');
END raise_salary;
```

Aufruf in PL/SQL: `raise_salary(emp_num, amount);`
Aufruf in SQL*PLUS: durch anonymen Block oder mit `CALL raise_salary(...)`

Gespeicherte PL/SQL-Programme
Prozeduren und Funktionen (4)

- **Default-Werte** für Parameter zulässig

```
PROCEDURE create_dept (
new_dname CHAR DEFAULT 'TEMP',
new_loc   CHAR DEFAULT 'TEMP') IS ...
```

- Bei Aufruf können **Aktualparameter weggelassen** werden -> Defaultwert gilt

```
create_dept();
create_dept('MARKETING'); -- wird new_dname zugewiesen
create_dept('MARKETING', 'NEW YORK');
create_dept('NEW YORK'); --wird NICHT der new_loc zugewiesen
```

- Aktualparameter "von links nach rechts" zugewiesen;
  Umgehung durch **Parameterbindung mit Nennung des Formalparameters**

```
create_dept(new_loc => 'NEW YORK');
```


Gespeicherte PL/SQL-Programme
Pakete (1)

- **Paketspezifikation** definiert
  - Schnittstellen für Prozeduren/Funktionen
  - globale Variablen/Konstanten
  - globale Ausnahmen
  - globale Cursor
  - **globale Variablen/Konstanten** werden einmal initialisiert u. existieren bis zum Logout aus der DB
  - Benutzer mit **Zugriffsrecht** auf ein Paket können darin definierte Prozeduren/ Funktionen ausführen und globale Variablen, etc. verwenden
- **Paket-Rumpf** implementiert die in der Spezifikation definierten Prozeduren/ Funktionen
  - Definition von **lokalen Variablen/Konstanten, Prozeduren/Funktionen, Ausnahmen u. Cursorn**, die nur innerhalb des Pakets bekannt sind

Gespeicherte PL/SQL-Programme
Pakete (2)

- **Definition einer Paketspezifikation bzw. eines Paket-Rumpfes** analog zur Definition von Standalone-Prozeduren/Funktionen durch DDL-Anweisungen

```
CREATE [OR REPLACE] PACKAGE <paketname> AS
   <globale Typ- u. Objektdeklarationen>
   <Prozedur- & Funktionsdeklarationen>
END [<paketname>];

CREATE [OR REPLACE] PACKAGE BODY <paketname> AS
   <lokale Typ- u. Objektdeklarationen>
   <Prozedur- & Funktions-Rümpfe>
[BEGIN
   <Initialisierungsanweisungen>]
END [<paketname>];
```

gleicher Name

- **Überladen** von Prozedur- bzw. Funktionsnamen im gleichen Paket **möglich**

Gespeicherte PL/SQL-Programme
Pakete (3)

```
CREATE PACKAGE emp_actions AS
   TYPE EmpRecTyp IS RECORD (emp_id INTEGER, salary REAL);
   CURSOR desc_salary RETURN EmpRecTyp;
   PROCEDURE hire_employee (ename CHAR, job CHAR, mgr NUMBER,
    sal NUMBER, comm NUMBER, deptno NUMBER);
   PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
   CURSOR desc_salary RETURN EmpRecTyp IS
      SELECT empno, sal FROM emp ORDER BY sal DESC;

   PROCEDURE hire_employee (ename CHAR, job CHAR, mgr NUMBER,
                            sal NUMBER, comm NUMBER, deptno NUMBER) IS
   BEGIN
      INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename, job, mgr,
                              SYSDATE, sal, comm, deptno);
   END hire_employee;

   PROCEDURE fire_employee (emp_id NUMBER) IS
   BEGIN
      DELETE FROM emp WHERE emp_no = emp_id;
   END fire_employee;
END emp_actions;
```

Gespeicherte PL/SQL-Programme
Anwendung

- **Aufruf von Standalone-Prozeduren und -Funktionen** durch

```
<ersteller_name>.<proz_name>(<aktuelle parameter>)
```

```
scott.raise_salary(hansi_id, 100);
```

- **Aufruf von Paket-Prozeduren und -Funktionen** durch

```
<ersteller_name>.<paket_name>.<proz_name>(<aktuelle parameter>)
```

```
scott.emp_actions.hire_employee('Hansi', hansi_job, hansi_mgr, 12000,
hansi_comm, 10);
```

Verwendung von **Synonymen** zur Vereinfachung

```
CREATE SYNONYM hire_emp FOR scott.emp_actions.hire_employee;
```

- beliebige PL/SQL-Programme der gleichen Session haben **auf die im Paket definierten globalen Variablen/Konstanten** über Punktnotation **Zugriff**

```
FOR csal_rec IN emp_actions.desc_salary LOOP
...
```


Gespeicherte PL/SQL-Programme
Testen

- vordefiniertes PL/SQL-Paket `dbms_output` stellt **Debug-Prozeduren** zur Verfügung

`put(var)`	... schreibt Wert beliebiger Var. in aktuelle Zeile
`put_line(var)`	... schreibt Wert beliebiger Var. in neue Zeile
`new_line`	... beginnt neue Zeile

- **Aktivierung der Ausgabe** in SQL*Plus mit `set serveroutput on`

```
CREATE OR REPLACE PROCEDURE b_p1 (nr NUMBER) AS
   ...
BEGIN
   dbms_output.put_line('Beginn Prozedur b_p1');
   FOR i IN 1..10 LOOP
    ...
    dbms_output.put('Anzahl: ');
    dbms_output.put(anzahl);
    dbms_output.new_line;
   ...
```


Dynamisches SQL
Motivation

- **bisher: statische Einbindung** von SQL-Statements in PL/SQL-Blöcke
  keine Generierung und Ausführung von SQL-Statements “on the fly” möglich
- flexiblere Ergänzung: **Dynamisches SQL** (in Oracle: “Native Dynamic SQL”): SQL-Statements nicht vordefiniert, sondern zur Laufzeit generiert
- **beliebige DDL- und DML-Statements** (d.h., auch Statements, die in Standard-PL/SQL unzulässig sind)
- **Unterscheidung:**
  einfache Statements (DDL, Datenänderungen, Select mit einem Ergebnistupel)
  Statements mit Cursor (Select mit einer Menge von Ergebnistupeln)
- **Besondere Zugriffsrechte** für Benutzer, die DDL-Statements via dynamischem SQL benutzen wollen (`CREATE TABLE ...`)

Dynamisches SQL
einfache Statements

- **Statement als Zeichenkette** definieren und mit **EXECUTE IMMEDIATE** ausführen lassen
- **Platzhalter** für eigentliche Wert innerhalb der Zeichenkette zulässig
- Bei Ausführung müssen aktuelle Werte für Platzhalter angegeben werden

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

- auch DDL-Statements aufrufbar

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

Dynamisches SQL
einfache Statements: Variablenbindung

- Angabe von **Variablen** oder **Konstanten** für Platzhalter
- **Übergabe** von Werten an das Statement oder **Übernahme** von Werten vom Statement
- Binden von Variablen im **Parametermodus** `IN, OUT` oder `IN OUT` (Default: `IN`)

```
sqlstmt := 'DELETE FROM dept WHERE deptno = :dept';
EXECUTE IMMEDIATE sqlstmt USING deptnumber;
```
(`deptnumber` wird als Input-Parameter gebunden)

```
sqlstmt := 'DELETE FROM dept WHERE deptno = 20 RETURNING loc
INTO :x';
EXECUTE IMMEDIATE sqlstmt USING OUT location;
```
(`location` als Output-Parameter)

Dynamisches SQL
einfache Statements: Schemaobjekte

- **Platzhalter** dürfen **nicht für Schemaobjekte** verwendet werden

  unzulässiges Statement:
```
EXECUTE IMMEDIATE 'DELETE FROM :table WHERE :attr = :val';
```
- Lösung: **dynamisches Ermitteln** der Zeichenkette

```
CREATE PROCEDURE delete_rows
         (table_name IN VARCHAR2, condition IN VARCHAR2) AS
IF (condition IS NULL) THEN
   ...
ELSE
   EXECUTE IMMEDIATE
     'DELETE FROM ' || table_name || ' WHERE ' || condition;
END IF;
```


Dynamisches SQL
Statements mit Cursor

- **Ablauf:**
  1. Deklaration des Cursors im Deklarationsteil
  2. Öffnen des Cursors für ein bestimmtes Statement
  3. Lesen der Ergebnistupel
  4. Schließen des Cursors

```
TYPE RefCur IS REF CURSOR;
emp_cv      RefCur;
...
OPEN emp_cv FOR
'SELECT ename, sal FROM emp WHERE sal > :s' USING my_sal;
LOOP
   FETCH emp_cv INTO my_ename, my_sal;
   EXIT WHEN emp_cv%NOTFOUND;
END LOOP;
CLOSE emp_cv;
```