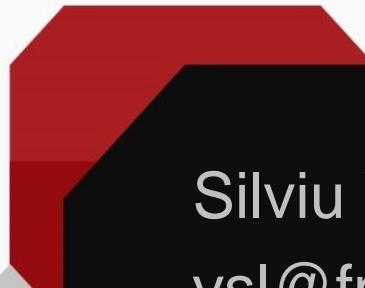


Curs 1: Prezentare

SGBD



Silviu Vasile
vsl@fmi.unibuc.ro

Informatii generale:

- Program:
 - Curs: 16:00 – 17:50 (vineri ?)
- Forma de evaluare:
 - Examen + **2 Partiale** (functii, triggeri)
- Cursurile vor contine notiunile necesare pentru rezolvarea exercitiilor din laborator – recommand sa fiti conectati la serverul de BD in timpul cursului
- Fara pauza
- Contact:
 - email: vsl@fmi.unibuc.ro
 - web: <https://moodle.unibuc.ro/course/view.php?id=3038>

Examen

- Partiale:
 - dupa laboratorul de subprograme;
 - Dupa laboratorul de triggeri.
- Examen sesiune
- Calcul nota finala: $N=(2*E+P1+P2)/4$
- Absenta la partial => $P_i=1$

Schema succinta a cursului:

- ◆ Generalitati
- ◆ Introducere in PL/SQL
- ◆ *Variabile in PL/SQL*
- ◆ Sintaxa PL/SQL
- ◆ *Tipuri de date (record, object, tablou indexat/imbricat, vector)*
- ◆ *Cursoare*

Schema succinta a cursului:

- ◆ ***Subprograme***
- ◆ **Pachete**
- ◆ **Exceptii. Tratarea erorilor**
- ◆ ***Triggeri***
- ◆ **Pachete sistem (DBMS_JOB, UTL_FILE, DBMS_SQL etc)**

Introducere:

- ◆ Ce pondere considerati ca au bazele de date in viata de zi cu zi?

Introducere:

- ◆ Ati folosit astazi o baza de date? La ce?

Introducere:

- ◆ Ce este o baza de date?

Introducere:

- ◆ Ce functionalitati trebuie sa indeplineasca o baza de date?

Introducere:

- ◆ Ce deosebiri considerati ca exista intre un tabel al unei baze de date si un «sac»?

Introducere:

- ◆ Ce este un SGBD? SQL?
- ◆ PL/SQL?

Introducere:

- ◆ Ce SGBD-uri cunoasteti?

Introducere:

◆ Ce este o baza de date relationala?

Introducere:

- ◆ De ce considerati ca intr-o baza de date sunt necesare mai multe tabele?

Bibliografie:

- **Modelarea bazelor de date, Ileana Popescu, 2001**
- **Oracle SQL By Example (4th Edition) Paperback – August 22, 2009**
- **Oracle Database 12c SQL Paperback – August 20, 2013**
- **Murach's Oracle SQL and PL/SQL for Developers, 2nd Edition**
- **Oracle PL/SQL Programming Paperback – February 16, 2014**
- **OCA Oracle Database 12c SQL Fundamentals I Exam Guide (Exam 1Z0-061) (Oracle Press)**
- **The Language of SQL: How to Access Data in Relational Databases Paperback – June 3, 2010**
- **Expert Oracle SQL: Optimization, Deployment, and Statistics Paperback – June 24, 2014**

Introducere în PL/SQL

Motivatie PL/SQL

- Structured Query Language (SQL) este limbajul standard pentru manipularea bazelor de date relationale
- Sa consideram urmatoarea cerere:

```
SELECT first_name, department_id, salary
FROM employees;
```
- Cerere: In functie de fiecare department si de salariul angajatiilor sa se acorde un bonus.

Imagine generala PL/SQL

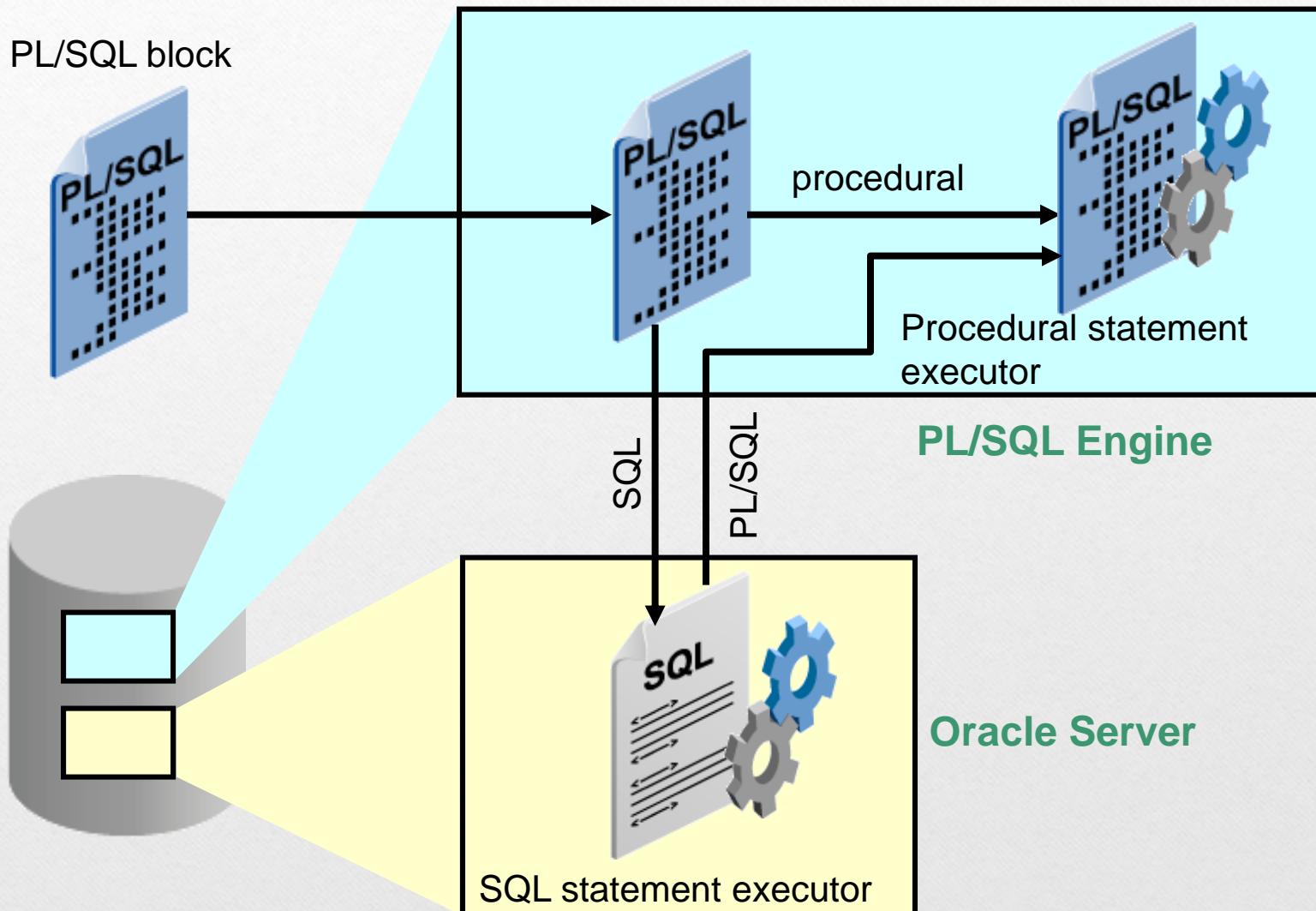
- PL/SQL:
 - Poate fi definit ca “Procedural Language extension to SQL”
 - Este limbajul procedural standard Oracle pentru manipularea bazelor de date relationale
 - Integreaza facilitatile unui limbaj procedural peste SQL



Generalitati PL/SQL

- PL/SQL:
 - Se caracterizeaza printr-o structura de bloc in care sunt integrate comenzile ce urmeaza a fi prelucrate. Mantinerea codului este mult mai usoar pe o astfel de structura.
 - Ofere facilitatile unui limbaj procedural:
 - Variabile, constante si tipuri de date
 - Structuri pentru adaugarea de conditii si pentru controlul executiei (case, if, for, loop, while)
 - Ofere facilitati pentru stocarea si reutilizarea codului.

Procesarea blocurilor PL/SQL

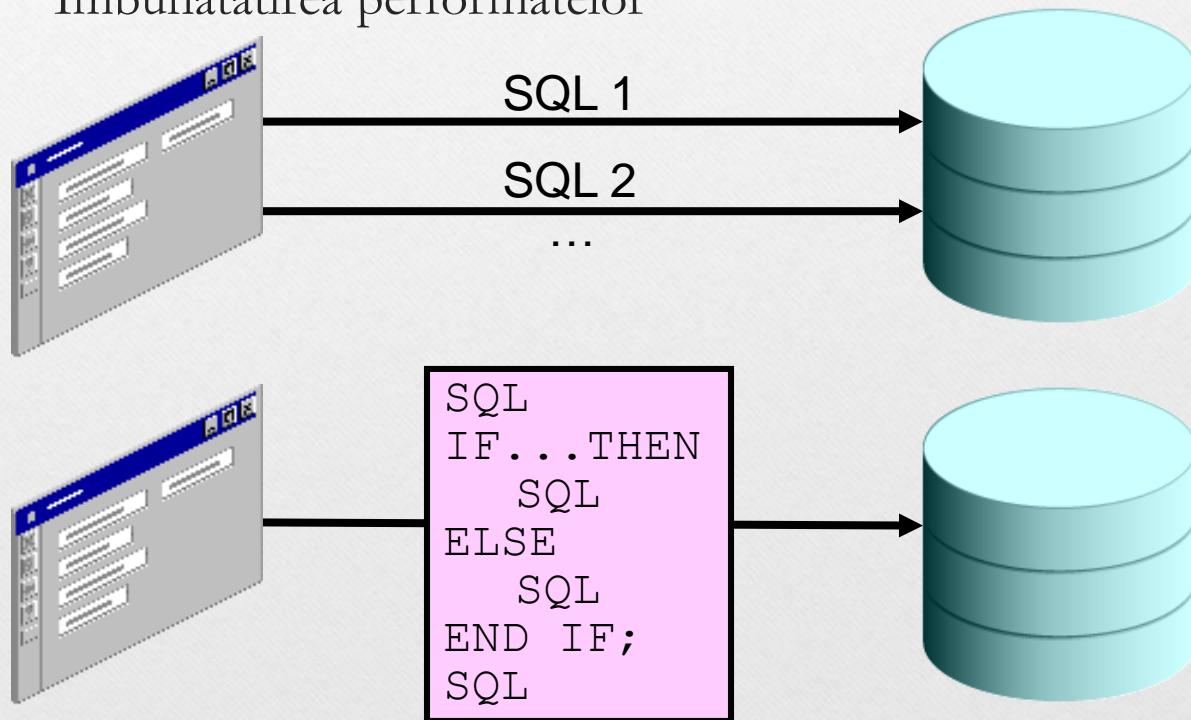


Procesarea blocurilor PL/SQL

- Bloc PL/SQL:
 - În general un bloc PL/SQL poate să contină sintaxa specifică unui limbaj procedural și comenzi SQL
 - Execuției blocului PL/SQL se intrerupe pentru execuția comenziilor SQL
 - O comandă SQL poate să contină/să implice execuția unui bloc PL/SQL
 - În ambele situații se așteaptă finalizarea contextului apelat și eventual utilizarea valorilor returnate în contextual apelant

Avantajele utilizarii blocurilor PL/SQL

- Utilizarea comenzilor specifice unui limbaj procedural
 - SQL= **ce sa facă** vs. PL/SQL=**ce sa facă + cum sa facă**
- Imbunatatirea performatelor



Avantajele utilizarii blocurilor PL/SQL

- Modularizarea codului
- Integrarea unor utilitare/programe externe
- Portabilitate
- Tratarea erorilor
- PL/SQL utilizeaza aceleasi tipuri de date din SQL (cu mici extensii) si comenzi specifice SQL

Structura blocului PL/SQL

- DECLARE (optional)
 - Variabile, cursoare, exceptii, tipuri de date locale
- BEGIN (mandatory)
 - Comenzi SQL
 - Sintaxa PL/SQL
- EXCEPTION (optional)
 - Ce actiuni sa intreprinda cand are loc o exceptie
- END; (mandatory)



Tipuri de blocuri PL/SQL

- Anonymous

```
[DECLARE]  
  
BEGIN  
    --statements  
  
[EXCEPTION]  
  
END;
```

Procedure

```
PROCEDURE name  
IS  
  
BEGIN  
    --statements  
  
[EXCEPTION]  
  
END;
```

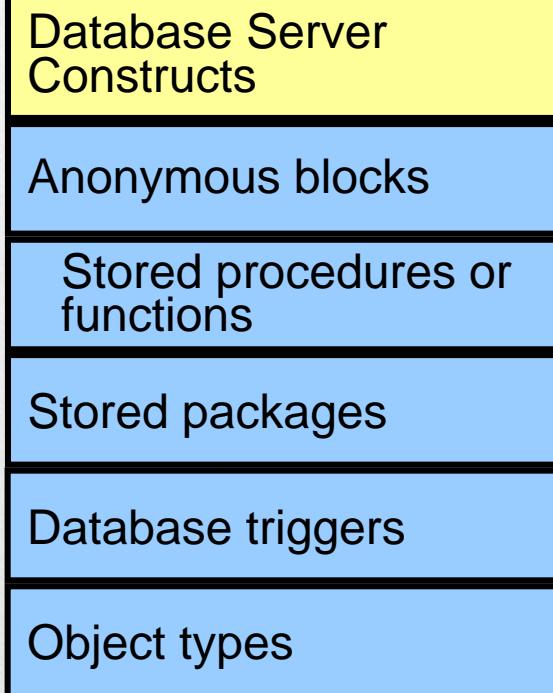
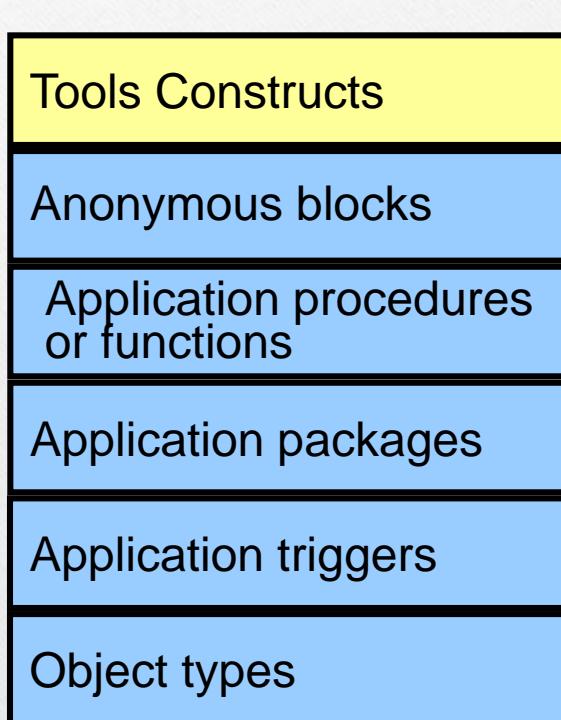
Function

```
FUNCTION name  
RETURN datatype  
IS  
  
BEGIN  
    --statements  
    RETURN value;  
  
[EXCEPTION]  
  
END;
```

Tipuri de blocuri PL/SQL

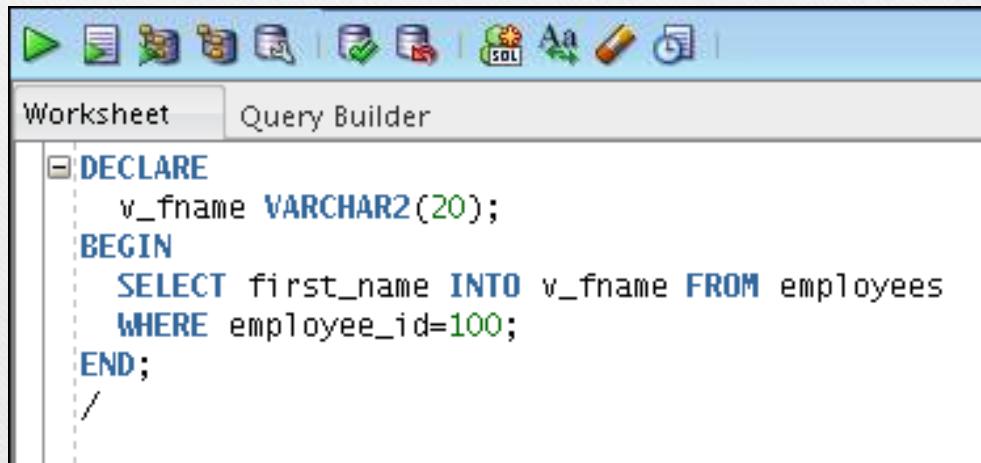
- Un program PL/SQL poate fi alcătuit din mai multe blocuri care pot fi independente sau încuibarite
- Pentru a defini un program pot fi utilizate 3 tipuri de blocuri:
 - **Anonime** sunt blocuri care nu primesc nume; se declară la un anumit moment și nu sunt stocate (se declara și se compileaza de fiecare dată cand sunt utilizate)
 - **Proceduri** - blocuri stocate care primesc nume
 - **Functii** - blocuri stocate care au nume + trebuie să returneze
- **Nota:** subprogramele pot fi reutilizate în diferite contexte;

Utilizare blocurilor PL/SQL



Structura unui bloc anonim

- Definirea unui bloc anonim in SQL Developer:



The screenshot shows the Oracle SQL Developer interface with the 'Worksheet' tab selected. The main area contains the following PL/SQL code:

```
DECLARE
    v_fname VARCHAR2(20);
BEGIN
    SELECT first_name INTO v_fname FROM employees
    WHERE employee_id=100;
END;
/
```

Executia unui bloc anonim

- Selectati butonul Run Script (F5) pentru a executa blocul anonim:

The screenshot shows the Oracle SQL Developer interface. The top menu bar has a 'Run Script (or F5)' button highlighted with a yellow box. Below the menu is a toolbar with various icons. The main workspace is titled 'Worksheet' and contains the following PL/SQL code:

```
DECLARE
    v_fname VARCHAR2(20);
BEGIN
    SELECT first_name INTO v_fname FROM employees
    WHERE employee_id=100;
END;
/
```

Below the code, the 'Script Output' window shows the result: 'anonymous block completed'. At the bottom of the output window, it says 'Task completed in 0.037 seconds'.

Afisarea rezultatului rularii unui bloc PL/SQL

1. Pentru a permite afisarea rezultatului rularii unui bloc PL/SQL se adauga urmatoarea comanda inaintea blocului:

```
SET SERVEROUTPUT ON
```

2. Pentru a afisa un anumit mesaj folositi urmatoarea procedura din pachetul DBMS_OUTPUT:

- DBMS_OUTPUT.PUT_LINE

```
DBMS_OUTPUT.PUT_LINE('The First Name of the  
Employee is ' || v_fname);  
...
```

Rezultatul rularii unui bloc PL/SQL

The screenshot shows the Oracle SQL Developer interface. In the top toolbar, there's a green play button icon followed by several other icons. To the right of the toolbar, it says "0.007 seconds". Below the toolbar, the tab "Worksheet" is selected. The main workspace contains the following PL/SQL code:

```
SET SERVEROUTPUT ON
DECLARE
    v_fname VARCHAR(20);
BEGIN
    SELECT first_name
    INTO v_fname
    FROM employees
    WHERE employee_id = 100;
    DBMS_OUTPUT.PUT_LINE('The First Name of the Employee is ' || v_fname);
END;
```

A yellow callout box with a black border and a black arrow points from the text "Press F5 to execute the command and PL/SQL block." to the green play button icon in the toolbar.

In the bottom left corner of the workspace, there's a small icon of a script and the text "Script Output". Below the workspace, the status bar displays "Task completed in 0.007 seconds". The output pane shows the results of the execution:

```
anonymous block completed
The First Name of the Employee is Steven
```

Quiz

- Un bloc PL/SQL *trebuie* sa contine urmatoarele 3 sectiuni:
 - **Declarativa**, care incepe cu DECLARE si se continua pana la partea executabila
 - **Executabila**, este marcata prin BEGIN si se termina cu END
 - **Tratarea erorilor**, este marcata prin cuvantul cheie EXCEPTION si este inclusa in partea executabila
- a. Adevarat
- b. Fals

Exercitii (I)

Ce subpuncte definesc un bloc PL/SQL valid?

- a. BEGIN
END;
- b. DECLARE
v_amount INTEGER(10);
END;
- c. DECLARE
BEGIN
END;
- d. DECLARE
v_amount INTEGER(10);
BEGIN
DBMS_OUTPUT.PUT_LINE(v_amount);
END;

Exercitii (II)

1. Definiti un bloc anonim care afiseaza mesajul “*Hello World*.”

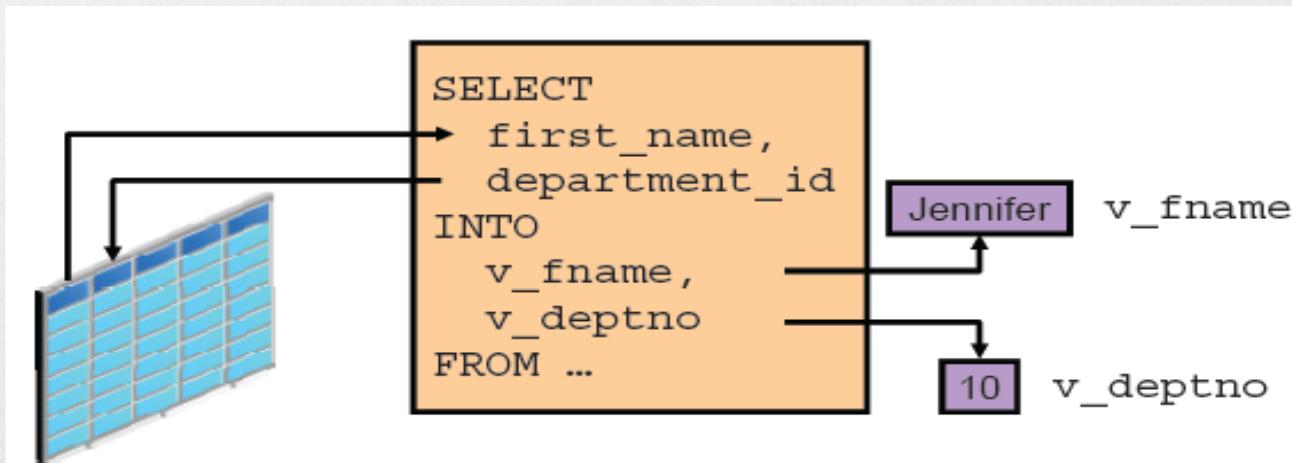
Exercitii pregaritoare curs 3:

1. Definiti un bloc anonim care citeste de la tastatura un cod de angajat si ii afiseaza numele.
2. Definiti un bloc anonim care citeste de la tastatura un nume de angajat si afiseaza salariul.

Variabile in PL/SQL

Utilizarea variabilelor

- Pentru stocarea temporara a datelor
- Manipularea informatiilor stocate
- (Re)Utilizarea in diferite contexte



Denumirea variabilelor

- Un nume de variabila:
 - Trebuie sa inceapa cu o litera
 - Poate sa includa litere si cifre
 - Poate sa includa caracterele speciale \$, _,#
 - Nu poate sa aiba o lungime mai mare de 30 de caractere
 - Nu poate sa includa cuvinte rezervate (ex: join, select etc)

Utilizarea variabilelor

- Variabilele sunt:
 - Declarate si optional initialize in sectiunea *DECLARE*.
 - Utilizate pentru a stoca valori in sectiunea executabila
 - Folosite drept parametrii ale subprogramelor
 - Utilizate drept valoare atribuita doar daca in prealabil au fost definite

Declararea si initializarea variabilelor

Sintaxa:

```
identifier [CONSTANT] datatype [NOT NULL]
      [:= | DEFAULT expr];
```

Exemple:

```
DECLARE
    v_hiredate      DATE;
    v_location       VARCHAR2(13) := 'Atlanta';
    v_deptno         NUMBER(2) NOT NULL := 10;
    c_comm           CONSTANT NUMBER := 1400;
```

Declararea si initializarea variabilelor

Observatie:

```
DECLARE
    v_myName  VARCHAR(20);
BEGIN
    DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myName );
    v_myName  := 'John';
    DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myName );
END;
/
```

```
DECLARE
    v_myName  VARCHAR2(20) := 'John';
BEGIN
    v_myName  := 'Steven';
    DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myName );
END;
/
```

Delimitatori in initializarea variabilelor

Exemplu:

```
DECLARE
    v_event VARCHAR2(15);
BEGIN
    v_event := q'!Father's day!';
    DBMS_OUTPUT.PUT_LINE('3rd Sunday in June is :
    ' || v_event );
    v_event := q'[Mother's day]';
    DBMS_OUTPUT.PUT_LINE('2nd Sunday in May is :
    ' || v_event );
END;
/
```

Output:

3rd Sunday in June is :Father's day
2nd Sunday in May is :Mother's day

Clasificarea variabilelor

- Variabile PL/SQL:
 - Scalare
 - Pointer
 - Colectii
 - Obiecte externe (adresa obiectelor LOB)
- Variabile de legatura (SQL – PL/SQL)
 - Utilizate pentru a pasa informatie intre cele 2 medii

Exemple tipuri de variabile

TRUE



15-JAN-09

Snow White
Long, long ago,
in a land far, far away,
there lived a princess called Snow
White...



256120.08

Atlanta

Info:

BOOLEAN, DATE, BLOB, VARCHAR2, CLOB, NUMBER, BFILE

Declararea variabilelor in PL/SQL

- Alegeti denumiri sugestive (x vs v_emp_id)
- Definiti cate o variabila pe linie
- Initializati variabilele care trebuie sa fie NOT NULL sau sunt declarate cu ajutorul CONSTANT
- Folositi pentru initializare “:=“ sau DEFAULT

```
v_myName VARCHAR2(20) := 'John';
```

```
v_myName VARCHAR2(20) DEFAULT 'John';
```

Declararea variabilelor in PL/SQL

- Evitati sa denumiti variabilele cu numele coloanelor care vor furniza valorile

```
DECLARE
    employee_id NUMBER(6);
BEGIN
    SELECT      employee_id
    INTO        employee_id
    FROM        employees
    WHERE       last_name = 'Kochhar';
END;
```

A yellow triangular warning sign with a red exclamation mark in the center, positioned next to the code snippet.

- Utilizati optiunea NOT NULL atunci cand o variabila trebuie sa aiba o valoare

```
pincode VARCHAR2(15) NOT NULL := 'Oxford';
```

Conventii privind denumirea variabilelor

PL/SQL Structure	Convention	Example
Variable	v_variable_name	v_rate
Constant	c_constant_name	c_rate
Subprogram parameter	p_parameter_name	p_id
Bind (host) variable	b_bind_name	b_salary
Cursor	cur_cursor_name	cur_emp
Record	rec_record_name	rec_emp
Type	type_name_type	ename_table_type
Exception	e_exception_name	e_products_invalid
File handle	f_file_handle_name	f_file

Variabile de tip scalar

```
DECLARE
    v_emp_job          VARCHAR2(9);
    v_count_loop       BINARY_INTEGER := 0;
    v_dept_total_sal  NUMBER(9,2)  := 0;
    v_orderdate        DATE := SYSDATE + 7;
    c_tax_rate         CONSTANT NUMBER(3,2) := 8.25;
    v_valid            BOOLEAN NOT NULL := TRUE;
```

Declararea cu ajutorul %TYPE

- În unele situații se dorește definirea unei variabile care să permită întotdeauna salvarea datelor care se regăsesc într-o coloană a unui tabel (indiferent de modificările pe care acesta le poate suferi)
- Se folosește %type pentru:
 - definirea unei variabile de tipul tabel.coloana%type;
 - definirea unei variabile de tipul altor variabile variabila%type;

Declararea cu ajutorul %TYPE

- Sintaxa:

```
identifier      table.column_name%TYPE;
```

- Exemple:

```
...
  v_emp_lname      employees.last_name%TYPE;
...
```

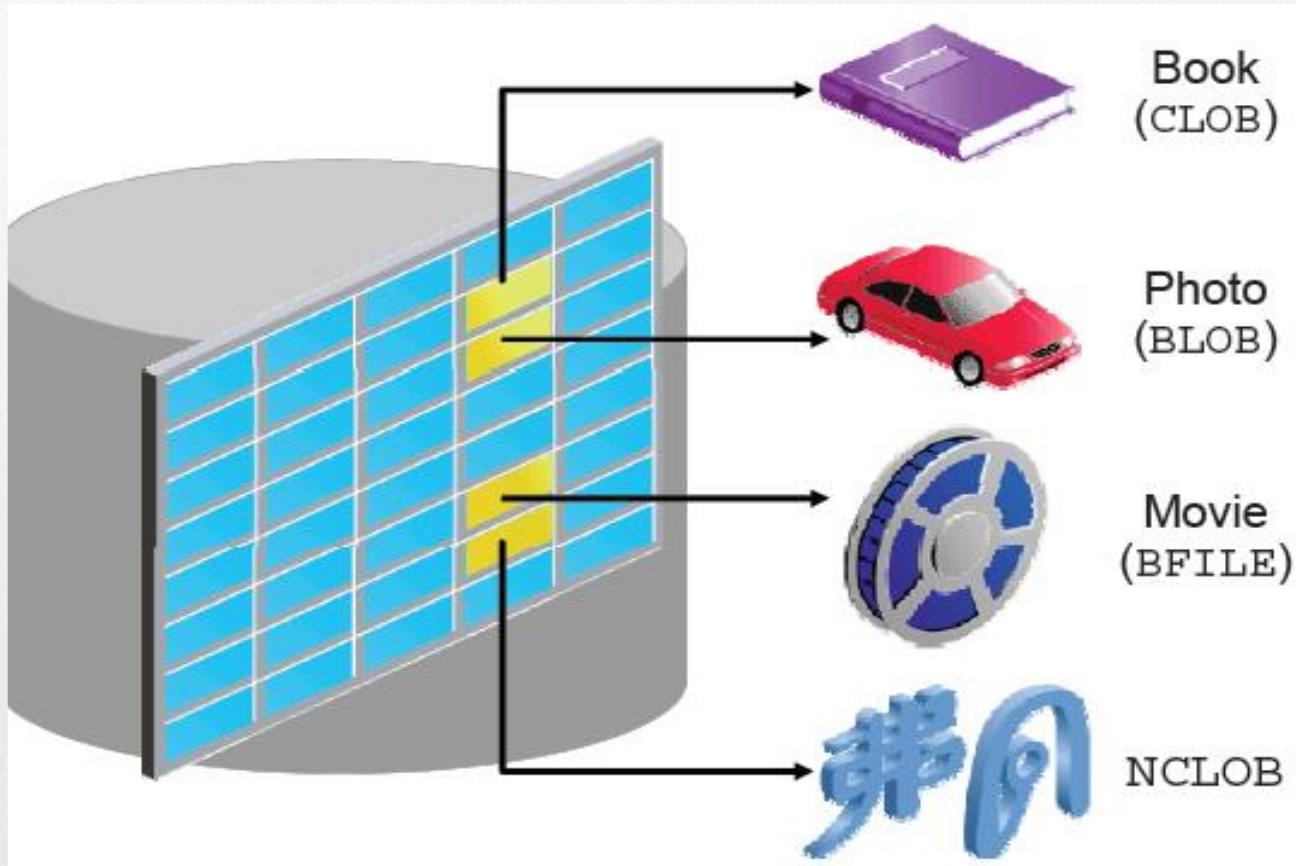
```
...
  v_balance        NUMBER(7,2);
  v_min_balance   v_balance%TYPE := 1000;
...
```

Variabile de tip boolean in PL/SQL

- Unei variabile de tip boolean ii pot fi atribuite doar valorile TRUE, FALSE si NULL
- Pot fi folosite impreuna cu operatorii logici AND si OR
- Expresiile aritmetice, cele care prelucreaza tipuri de date caracter si date pot intoarce expresii de tip boolean

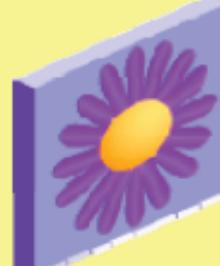
```
DECLARE
    flag BOOLEAN := FALSE;
BEGIN
    flag := TRUE;
END;
```

Tipuri de date LOB



Tipuri de date compuse

PL/SQL Record:

TRUE	23-DEC-98	ATLANTA	
------	-----------	---------	---

PL/SQL Collections:

1	SMITH	1	5000
2	JONES	2	2345
3	NANCY	3	12
4	TIM	4	3456

PLS_INTEGER VARCHAR2 PLS_INTEGER NUMBER

Variabile de legatura

- Generalitati
 - Pot fi definite de mediu (environment)
 - Sunt denumite variabile host
 - NU sunt variabile globale
 - Sunt definite in SQLDeveloper cu ajutorul VARIABLE
 - Pot fi utilizate atat in SQL, cat si in PL/SQL
 - Pot fi accesate dupa terminarea blocului PL/SQL in care sunt utilizate
 - Sunt prefixate de ":"
 - Sunt afisate in SQL cu ajutorul lui PRINT

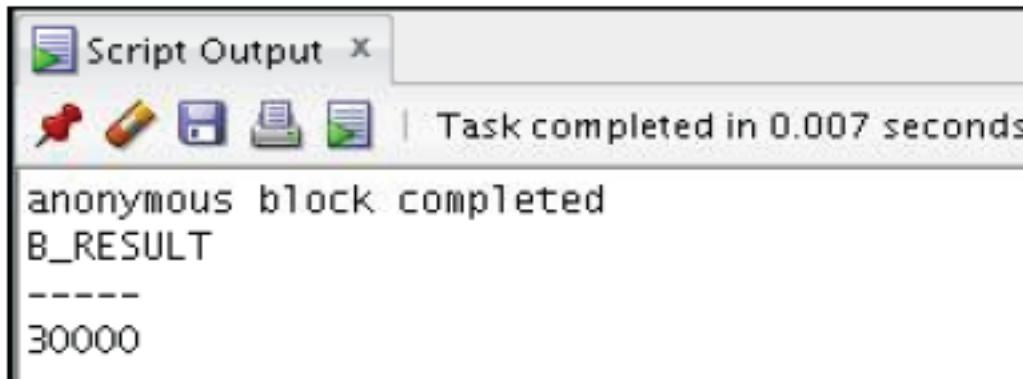
```
VARIABLE return_code NUMBER
```

```
VARIABLE return_msg    VARCHAR2(30)
```

Variabile de legatura

- Exemplu:

```
VARIABLE b_result NUMBER
BEGIN
    SELECT (SALARY*12) + NVL(COMMISSION_PCT,0) INTO :b_result
    FROM employees WHERE employee_id = 144;
END;
/
PRINT b_result
```



The screenshot shows the 'Script Output' window from Oracle SQL Developer. The window title is 'Script Output'. The status bar at the bottom right says 'Task completed in 0.007 seconds'. The main area displays the output of an anonymous block. It starts with 'anonymous block completed', followed by a header 'B_RESULT' with a dashed line below it, and then the value '30000'.

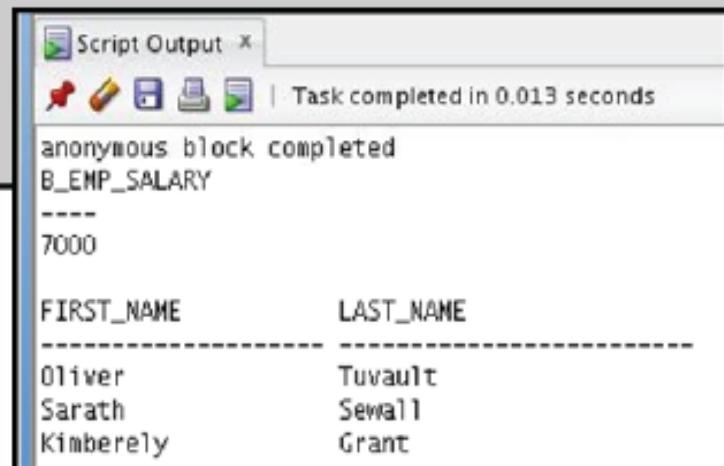
```
anonymous block completed
B_RESULT
-----
30000
```

Variabile de legatura

- Exemplu:

```
VARIABLE b_emp_salary NUMBER  
BEGIN  
    SELECT salary INTO :b_emp_salary  
    FROM employees WHERE employee_id = 178;  
END;  
/  
PRINT b_emp_salary  
SELECT first_name, last_name  
FROM employees  
WHERE salary=:b_emp_salary;
```

Output →



The screenshot shows the 'Script Output' window from Oracle SQL Developer. The window title is 'Script Output'. It displays the message 'Task completed in 0.013 seconds'. Below this, it shows the output of an anonymous block. The output starts with 'anonymous block completed' followed by a table with two columns: 'FIRST_NAME' and 'LAST_NAME'. The table has four rows with data: Oliver Tuvault, Sarah Sewall, Kimberly Grant, and Kimberly Grant.

FIRST_NAME	LAST_NAME
Oliver	Tuvault
Sarah	Sewall
Kimberly	Grant
Kimberly	Grant

Variabile de legatura - AUTOPRINT

- Exemplu:

The screenshot shows a PL/SQL block in the Worksheet tab of Oracle SQL Developer. The code is as follows:

```
VARIABLE h.emp_salary NUMBER
SET AUTOPRINT ON
DECLARE
    v_empno NUMBER(6):=&empno;
BEGIN
    SELECT salary INTO :b_emp_salary
    FROM employees WHERE employee_id = v_empno;
END;
```

A red box highlights the `SET AUTOPRINT ON` statement. A red arrow points from this statement to the `&empno;` placeholder in the `DECLARE` section. A yellow highlight covers the entire PL/SQL block.

A modal dialog box titled "Enter Substitution Variable" is displayed in the foreground. It contains a text input field labeled "EMPNO:" with the value "178" entered. At the bottom are "OK" and "Cancel" buttons.

Variabile case sensitive?

- În situații cu totul speciale este necesara definirea unor variabile case sensitive sau care să contină spații:

"begin date" DATE;

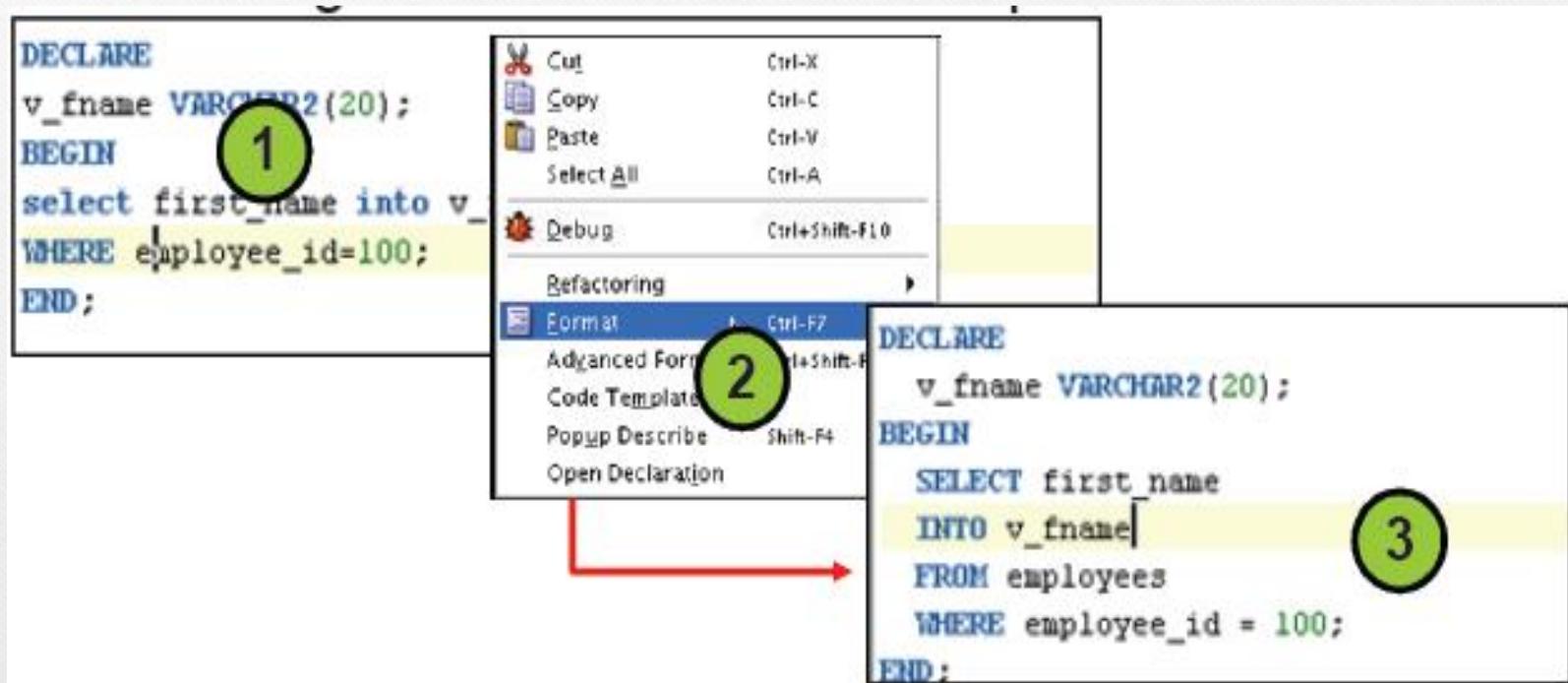
"end date" DATE;

"exc thrown" BOOLEAN DEFAULT TRUE;

- Aceste variabile trebuie să fie întotdeauna referite cu ajutorul "*var*"
- NU se recomanda (utilizarea este greoaie)

Formatarea unui bloc anonim

- Selectati butonul Format (CTRL+F7) pentru a formata blocul:



Comentarea codului

- Prefixarea liniei cu “--”
- Plasarea blocului intre simbolurile /* si */

```
DECLARE
  ...
  v_annual_sal NUMBER (9,2);
BEGIN
  /* Compute the annual salary based on the
     monthly salary input from the user */
  v_annual_sal := monthly_sal * 12;
  --The following line displays the annual salary
  DBMS_OUTPUT.PUT_LINE(v_annual_sal);
END;
```

Functii SQL in PL/SQL

- Într-un bloc PL/SQL sunt disponibile toate functiile single-row
- NU sunt disponibile functiile grup si functia DECODE

```
v_desc_size INTEGER(5);
v_prod_description VARCHAR2(70) := 'You can use this
product with your radios for higher frequency';

-- get the length of the string in prod_description
v_desc_size:= LENGTH(v_prod_description);
```

```
v_tenure:= MONTHS_BETWEEN (CURRENT_DATE, v_hiredate);
```

Utilizarea secventelor in PL/SQL

1. Incepand cu 11g este posibila utilizarea secventelor intr-un bloc:

```
DECLARE
    v_new_id NUMBER;
BEGIN
    v_new_id := my_seq.NEXTVAL;
END;
```

2. Inainte de 11g:

```
DECLARE
    v_new_id NUMBER;
BEGIN
    SELECT my_seq.NEXTVAL INTO v_new_id FROM Dual;
END;
```

Conversia tipurilor de date (implicite, explicite)

```
-- implicit data type conversion
```

```
v_date_of_joining DATE:= '02-Feb-2000';
```

```
-- error in data type conversion
```

```
v_date_of_joining DATE:= 'February 02,2000';
```

```
-- explicit data type conversion
```

```
v_date_of_joining DATE:= TO_DATE('February  
02,2000','Month DD, YYYY');
```

Utilizarea variabilelor in cadrul blocurilor imbricate

```
DECLARE
    v_outer_variable VARCHAR2(20) := 'GLOBAL VARIABLE';
BEGIN
    DECLARE
        v_inner_variable VARCHAR2(20) := 'LOCAL VARIABLE';
    BEGIN
        DBMS_OUTPUT.PUT_LINE(v_inner_variable);
        DBMS_OUTPUT.PUT_LINE(v_outer_variable);
    END;
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

Output: LOCAL VARIABLE
GLOBAL VARIABLE
GLOBAL VARIABLE

Utilizarea variabilelor in cadrul blocurilor imbricate

```
DECLARE
    v_father_name VARCHAR2(20) := 'Patrick';
    v_date_of_birth DATE := '20-Apr-1972';
BEGIN
    DECLARE
        v_child_name VARCHAR2(20) := 'Mike';
        v_date_of_birth DATE := '12-Dec-2002';
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Father''s Name: ' || v_father_name);
        DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth); ←
        DBMS_OUTPUT.PUT_LINE('Child''s Name: ' || v_child_name);
    END;
    →DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
END;
/
```

Exercitii

Determinati valorile variabilelor v_message, v_total_comp, v_comm, outer.v_comm la pozitiile indicate:

```
BEGIN <<outer>>
DECLARE
    v_sal      NUMBER(7,2)  := 60000;
    v_comm     NUMBER(7,2)  := v_sal * 0.20;
    v_message  VARCHAR2(255) := ' eligible for commission';
BEGIN
    DECLARE
        v_sal      NUMBER(7,2)  := 50000;
        v_comm     NUMBER(7,2)  := 0;
        v_total_comp  NUMBER(7,2) := v_sal + v_comm;
    BEGIN
        → v_message := 'CLERK not' || v_message;
        outer.v_comm := v_sal * 0.30;
    END;
    → v_message := 'SALESMAN' || v_message;
END;
END outer;
/
```

Sintaxa PL/SQL

Comenzi SQL in PL/SQL

- Cum sunt procesate comenziile SQL:
 - SELECT – cum se salveaza rezultatul returnat
 - UPDATE/DELETE in cadrul blocului PL/SQL
 - LCD (COMMIT, ROLLBACK, SAVEPOINT) in PL/SQL

Sintaxa SELECT

Sintaxa:

```
SELECT  select_list
INTO    {variable_name[, variable_name] ...
        | record_name}
FROM    table
[WHERE  condition];
```

Obs: clauza INTO este obligatorie; cererea trebuie sa intoarca o singura linie;

```
DECLARE
  v_fname VARCHAR2(25);
BEGIN
  SELECT first_name INTO v_fname
  FROM employees WHERE employee_id=200;
  DBMS_OUTPUT.PUT_LINE(' First Name is : '||v_fname);
END;
```

Sintaxa SELECT

Exemplu1:

```
DECLARE
    v_emp_hiredate    employees.hire_date%TYPE;
    v_emp_salary       employees.salary%TYPE;
BEGIN
    SELECT    hire_date, salary
    INTO      v_emp_hiredate, v_emp_salary
    FROM     employees
    WHERE    employee_id = 100;
    DBMS_OUTPUT.PUT_LINE ('Hire date is :|| v_emp_hiredate);
    DBMS_OUTPUT.PUT_LINE ('Salary is :|| v_emp_salary);
END;
```

Sintaxa SELECT

Exemplu2:

```
DECLARE
    v_sum_sal    NUMBER(10,2);
    v_deptno     NUMBER NOT NULL := 60;
BEGIN
    SELECT SUM(salary) -- group function
    INTO v_sum_sal FROM employees
    WHERE department_id = v_deptno;
    DBMS_OUTPUT.PUT_LINE ('The sum of salary is ' || v_sum_sal);
END;
```

Obs1: ! Curs 2: V_sum_sal := SUM(employees.salary); ???

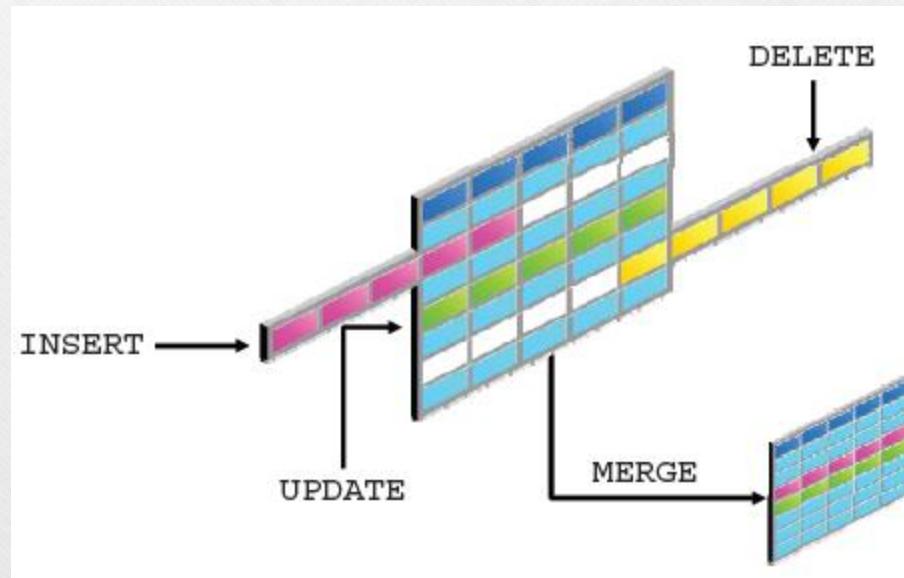
Obs2: ! Curs 2:

```
DECLARE
hire_date employees.hire_date%TYPE;
employee_id employees.employee_id%TYPE := 176;
BEGIN
SELECT hire_date INTO hire_date
FROM employees
WHERE employee_id = employee_id;
END;
```

DML in PL/SQL

Modificările intr-un tabel dintr-o baza de date pot fi implementate cu ajutorul:

- INSERT
- UPDATE
- DELETE
- MERGE



DML in PL/SQL – INSERT/UPDATE

```
BEGIN
    INSERT INTO employees
    (employee_id, first_name, last_name, email,
     hire_date, job_id, salary)
    VALUES(employees_seq.NEXTVAL, 'Ruth', 'Cores',
           'RCORES', CURRENT_DATE, 'AD_ASST', 4000);
END;
```

```
DECLARE
    sal_increase    employees.salary%TYPE := 800;
BEGIN
    UPDATE      employees
    SET          salary = salary + sal_increase
    WHERE        job_id = 'ST_CLERK';
END;
```

DML in PL/SQL – DELETE/MERGE

Exemplu:

```
DECLARE
    deptno    employees.department_id%TYPE := 10;
BEGIN
    DELETE FROM    employees
    WHERE    department_id = deptno;
END;

BEGIN
MERGE INTO copy_emp c
    USING employees e
    ON (e.employee_id = c.empno)
WHEN MATCHED THEN
    UPDATE SET
        c.first_name      = e.first_name,
        c.last_name       = e.last_name,
        c.email           = e.email,
        . . .
WHEN NOT MATCHED THEN
    INSERT VALUES(e.employee_id, e.first_name, e.last_name,
        . . .,e.department_id);
END;
```

Cursoare predefinite

- Un cursor este un pointer catre o zona privata de memorie (definit de SGBD)
- Este utilizat pentru a controla/manipula rezultatul unei cereri
- Sunt 2 tipuri de variabile cursor:
 - Implicite (definite automat)
 - Explicite (definite de utilizator)

Atributele unui cursor

- Când se procesează o comandă LMD, motorul SQL deschide un cursor implicit.
- Atributele scalare ale cursorului implicit
 - SQL%ROWCOUNT
 - SQL%FOUND
 - SQL%NOTFOUND
 - SQL%ISOPEN

furnizează informații referitoare la ultima comandă INSERT, UPDATE, DELETE sau SELECT INTO executată.

- Înainte ca Oracle să deschidă cursorul SQL implicit, atrbutele acestuia au valoarea null.

Atributele unui cursor

- Exemplu:

```
DECLARE
    v_rows_deleted VARCHAR2(30);
    v_empno employees.employee_id%TYPE := 176;
BEGIN
    DELETE FROM employees
    WHERE employee_id = v_empno;
    v_rows_deleted := (SQL%ROWCOUNT || ' row deleted.');
    DBMS_OUTPUT.PUT_LINE (v_rows_deleted);
END;
```

Atributele unui cursor

- Exemplu:

```
DECLARE  
  
    TYPE alfa IS TABLE OF NUMBER;    beta alfa;  
  
BEGIN  
  
    SELECT cod_artist BULK COLLECT INTO beta FROM artist;  
  
    FORALL j IN 1..beta.COUNT  
  
        INSERT INTO tab_art SELECT cod_artist,cod_opera  
  
            FROM     opera  
  
            WHERE   cod_artist = beta(j);  
  
    FOR j IN 1..beta.COUNT LOOP  
  
        DBMS_OUTPUT.PUT_LINE ('Pentru artistul ' || beta(j) || ' au  
fost inserate ' || SQL%BULK_ROWCOUNT(j) || ' inregistrari');  
  
    END LOOP;  
  
    DBMS_OUTPUT.PUT_LINE ('Numarul total este '||SQL%ROWCOUNT);  
  
END;
```

Comenzi de control a executiei

- IF
- CASE
- LOOP, WHILE, FOR
- Generalitati

Clauza IF

- Un program PL/SQL poate executa diferite porțiuni de cod, în funcție de rezultatul unui test (predicat). Instrucțiunile care realizează acest lucru sunt cele condiționale (IF, CASE).
- Structura instrucțiunii IF în PL/SQL este similară instrucțiunii IF din alte limbaje procedurale, permitând efectuarea unor acțiuni în mod selectiv, în funcție de anumite condiții. Instrucțiunea IF-THEN-ELSIF are următoarea formă sintactică:

```
IF condiție1 THEN
    secvența_de_comenzi_1
[ELSIF condiție2 THEN
    secvența_de_comenzi_2]
...
[ELSE
    secvența_de_comenzi_n]
END IF;
```

Clauza IF - exemplu

```
DECLARE
    v_myage number:=&m;
BEGIN
    IF v_myage < 11 THEN
        DBMS_OUTPUT.PUT_LINE(' I am a child ');
    ELSIF v_myage < 20 THEN
        DBMS_OUTPUT.PUT_LINE(' I am young ');
    ELSIF v_myage < 30 THEN
        DBMS_OUTPUT.PUT_LINE(' I am in my twenties');
    ELSIF v_myage < 40 THEN
        DBMS_OUTPUT.PUT_LINE(' I am in my thirties');
    ELSE
        DBMS_OUTPUT.PUT_LINE(' I am always young ');
    END IF;
END;
```

Clauza CASE

- Permite implementarea unor condiții multiple; are următoarea formă sintactică:

```
[<<eticheta>>]
CASE test_var
    WHEN valoare_1 THEN secvența_de_comenzi_1;
    WHEN valoare_2 THEN secvența_de_comenzi_2;
    ...
    WHEN valoare_k THEN secvența_de_comenzi_k;
    [ELSE altă_secvență; ]
END CASE [eticheta];
```

- Se va executa secvența_de_comenzi_p, dacă valoarea selectorului test_var are valoare_p. Selectorul test_var poate fi o variabilă sau o expresie complexă care poate conține chiar și apeluri de funcții.
- Clauza ELSE este optională

Clauza CASE - exemplu

```
DECLARE
    v_zi  CHAR(2) := UPPER('&p_zi');
BEGIN
    CASE
        WHEN v_zi = 'L' THEN
            DBMS_OUTPUT.PUT_LINE('Luni');
        WHEN v_zi = 'M' THEN
            DBMS_OUTPUT.PUT_LINE('Marti');
        ...
        WHEN v_zi = 'D' THEN
            DBMS_OUTPUT.PUT_LINE('Duminica');
        ELSE DBMS_OUTPUT.PUT_LINE('Este o eroare!');
    END CASE;
END;
```

Clauze iterative

- LOOP
- WHILE
- FOR

```
LOOP
    statement1;
    . . .
    EXIT [WHEN condition];
END LOOP;
```

```
WHILE condition LOOP
    statement1;
    statement2;
    . . .
END LOOP;
```

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
    statement1;
    statement2;
    . . .
END LOOP;
```

Clauze iterative exemple

```
DECLARE
    v_countryid      locations.country_id%TYPE := 'CA';
    v_loc_id         locations.location_id%TYPE;
    v_counter        NUMBER(2)  := 1;
    v_new_city       locations.city%TYPE := 'Montreal';
BEGIN
    SELECT MAX(location_id) INTO v_loc_id FROM locations
    WHERE country_id = v_countryid;
    LOOP
        INSERT INTO locations(location_id, city, country_id)
        VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
        v_counter := v_counter + 1;
        EXIT WHEN v_counter > 3;
    END LOOP;
END;
```

Clauze iterative exemple

```
DECLARE
    v_countryid    locations.country_id%TYPE := 'CA';
    v_loc_id        locations.location_id%TYPE;
    v_new_city      locations.city%TYPE := 'Montreal';
    v_counter       NUMBER := 1;
BEGIN
    SELECT MAX(location_id) INTO v_loc_id FROM locations
    WHERE country_id = v_countryid;
    WHILE v_counter <= 3 LOOP
        INSERT INTO locations(location_id, city, country_id)
        VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
        v_counter := v_counter + 1;
    END LOOP;
END;
```

Clauze iterative exemple

```
DECLARE
    v_countryid    locations.country_id%TYPE := 'CA';
    v_loc_id        locations.location_id%TYPE;
    v_new_city      locations.city%TYPE := 'Montreal';
BEGIN
    SELECT MAX(location_id) INTO v_loc_id
        FROM locations
        WHERE country_id = v_countryid;
    FOR i IN 1..3 LOOP
        INSERT INTO locations(location_id, city, country_id)
        VALUES( (v_loc_id + i), v_new_city, v_countryid );
    END LOOP;
END;
```

Clauza CONTINUE

- Care este ultima valoare a lui **v_total** afisata?

```
DECLARE
  v_total SIMPLE_INTEGER := 0;
BEGIN
  FOR i IN 1..10 LOOP
    v_total := v_total + i;
    dbms_output.put_line('Total is: ' || v_total);
    CONTINUE WHEN i > 5;
    v_total := v_total + i;
    dbms_output.put_line('Out of Loop Total is:' || v_total);
  END LOOP;
END;
```

Exercitii

- Se considera *emp* drept o copie a tabelului *employees* in care a fost adaugata coloana stars in care se trece cate o '*' pentru fiecare 1000 din salariu?
- Comentati rezolvarea urmatoare:

```
DECLARE
    v_empno emp.employee_id%TYPE := 176;
    v_asterisk emp.stars%TYPE := NULL;
    v_sal emp.salary%TYPE;
BEGIN
    SELECT NVL(ROUND(salary/1000), 0) INTO v_sal
    FROM emp WHERE employee_id = v_empno;
    FOR i IN 1..v_sal LOOP
        v_asterisk := v_asterisk || '*';
    END LOOP;
    UPDATE emp SET stars = v_asterisk
    WHERE employee_id = v_empno; COMMIT;
END;
```

Tipuri de date compuse

Agenda:

- Tipul de date RECORD
- Tipul de date colectie:
 - TABLOU INDEXAT
 - TABLOU IMBRICAT
 - VECTORI

Sintaxa SELECT

Tipul *RECORD* oferă un mecanism pentru prelucrarea înregistrărilor. Înregistrările au mai multe câmpuri ce pot fi de tipuri diferite, dar care sunt legate din punct de vedere logic.

Inregistrările trebuie definite în doi pași:

- se definește tipul *RECORD*;
- se declară înregistrările de acest tip.

Obs:

- Un *RECORD* poate să aibă un camp de tip *RECORD*;
- Este un mod elegant de a salva o linie returnată de o cerere;
- Poate constitui (prin varianta stocată) un tip de date pentru coloana unui tabel.

Sintaxa RECORD

Definire si utilizare:

```
TYPE type_name IS RECORD  
      (field_declaration, field_declaration) ;
```

```
identifier    type_name;
```

field_declaration:

```
field_name {field_type | variable%TYPE  
           | table.column%TYPE | table%ROWTYPE}  
           [ [NOT NULL] { := | DEFAULT} expr]
```

Sintaxa RECORD

Forma generală:

```
TYPE nume_tip IS RECORD
  (nume_camp1 {tip_camp / variabilă%TYPE /
    nume_tabel.coloană%TYPE / nume_tabel%ROWTYPE}
  [ [NOT NULL] {:= / DEFAULT} expresie1],
  (nume_camp2 {tip_camp / variabilă%TYPE /
    nume_tabel.coloană%TYPE / nume_tabel%ROWTYPE}
  [ [NOT NULL] {:= / DEFAULT} expresie2],...);
```

Field1 (data type)	Field2 (data type)	Field3 (data type)
employee_id number(6)	last_name varchar2(25)	job_id varchar2(10)
→ 100	King	AD_PRES

Observatii

- Dacă un câmp nu este inițializat atunci implicit se consideră că are valoarea *NULL*. Dacă s-a specificat constrângerea *NOT NULL*, atunci obligatoriu câmpul trebuie inițializat cu o valoare diferită de *NULL*.
- Pentru referirea câmpurilor individuale din înregistrare se prefixează numele câmpului cu numele înregistrării.
- Pot fi asignate valori unei înregistrări utilizând comenziile *SELECT*, *FETCH* sau instrucțiunea clasică de atribuire. De asemenea, o înregistrare poate fi asignată altiei înregistrari de același tip.
- Componentele unei înregistrări pot fi de tip scalar, *RECORD*, *TABLE*, obiect, colecție (dar, nu tipul *REF CURSOR*).
- *PL/SQL* permite declararea și referirea înregistrărilor imbicate.
- Numărul de câmpuri ale unei înregistrări nu este limitat.
- **Înregistrările nu pot fi comparate** (egalitate, inegalitate sau *null*).

Înregistrările pot fi parametri în subprograme și pot să apară în clauza *RETURN* a unei funcții.

%ROWTYPE vs RECORD

```
DECLARE  
    identifier reference%ROWTYPE;
```

- tipul *RECORD* permite specificarea tipului de date pentru câmpuri și permite declararea câmpurilor sale;
- atributul *%ROWTYPE* nu cere cunoașterea numărului și tipurilor coloanelor tabloului.
- se poate insera (*INSERT*) o linie într-un tabel utilizând o înregistrare. Nu mai este necesară listarea câmpurilor individuale, ci este suficientă utilizarea numelui înregistrării.
- se poate reactualiza (*UPDATE*) o linie a unui tabel utilizând o înregistrare. Sintaxa *SET ROW* permite să se reactualizeze întreaga linie folosind conținutul unei înregistrări.
- într-o înregistrare se poate regăsi și returna informația din clauza *RETURNING* a comenziilor *UPDATE* sau *DELETE*.

Exemple

Exemplu1:

```
DECLARE
    TYPE t_rec IS RECORD
        (v_sal number(8),
         v_minsal number(8) default 1000,
         v_hire_date employees.hire_date%type,
         v_rec1 employees%rowtype);
    v_myrec t_rec;
BEGIN
    v_myrec.v_sal := v_myrec.v_minsal + 500;
    v_myrec.v_hire_date := sysdate;
    SELECT * INTO v_myrec.v_rec1
        FROM employees WHERE employee_id = 100;
    DBMS_OUTPUT.PUT_LINE(v_myrec.v_rec1.last_name || ' ' ||
        to_char(v_myrec.v_hire_date) || ' '|| to_char(v_myrec.v_sal));
END;
```

Exemple

Exemplu2:

```
...
DECLARE
v_employee_number number:= 124;
v_emp_rec retired_emps%ROWTYPE;
BEGIN
SELECT employee_id, last_name, job_id, manager_id,
hire_date, hire_date, salary, commission_pct,
department_id INTO v_emp_rec
FROM employees
WHERE employee_id = v_employee_number;
INSERT INTO retired_emps VALUES v_emp_rec;
END;
/
```

Exemple

Exemplu3:

```
DECLARE
v_employee_number number:= 124;
v_emp_rec retired_emps%ROWTYPE;
BEGIN
SELECT * INTO v_emp_rec
FROM retired_emps
WHERE empno = v_employee_number;
v_emp_rec.leavedate:= CURRENT_DATE;
UPDATE retired_emps
SET ROW = v_emp_rec
WHERE empno=v_employee_number;
END;
/
```

Colecții

- Uneori este preferabil să fie prelucrate simultan mai multe variabile de același tip. Tipurile de date care permit acest lucru sunt colecțiile. Fiecare element are un indice unic, care determină poziția sa în colecție.
- În PL/SQL există trei tipuri de colecții:
 - tablouri indexate (index-by tables);
 - tablouri imbricate (nested tables);
 - vectori (varrays sau varying arrays).

Collecții

Tipul *index-by table* poate fi utilizat **numai** în declarații *PL/SQL*.

Tipurile *varray* și *nested table* pot fi utilizate atât în declarații *PL/SQL*, cât și în declarații la nivelul schemei (de exemplu, pentru definirea tipului unei coloane a unui tabel relațional)

DECLARE

```
TYPE tab_index IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
TYPE tab_imbri IS TABLE OF NUMBER;
TYPE vector IS VARRAY(15) OF NUMBER;
v_tab_index tab_index;
v_tab_imbri tab_imbri;
v_vector vector;

BEGIN
    v_tab_index(1) := 72;
    v_tab_index(2) := 23;
    v_tab_imbri := tab_imbri(5, 3, 2, 8, 7);
    v_vector := vector(1, 2);
END;
```

Observatii colecții

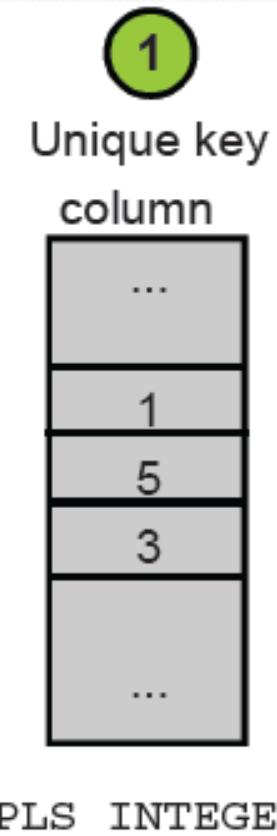
- Deoarece colecțiile nu pot fi comparate (egalitate sau inegalitate), ele nu pot să apară în clauzele *DISTINCT*, *GROUP BY*, *ORDER BY*.
- Tipul colecție poate fi definit într-un pachet.
- Tipul colecție poate să apară în clauza *RETURN* a unei funcții.
- Colecțiile pot fi parametri formali într-un subprogram.
- Accesul la elementele individuale ale unei colecții se face prin utilizarea unui indice.

Tablouri indexate - *associative array*

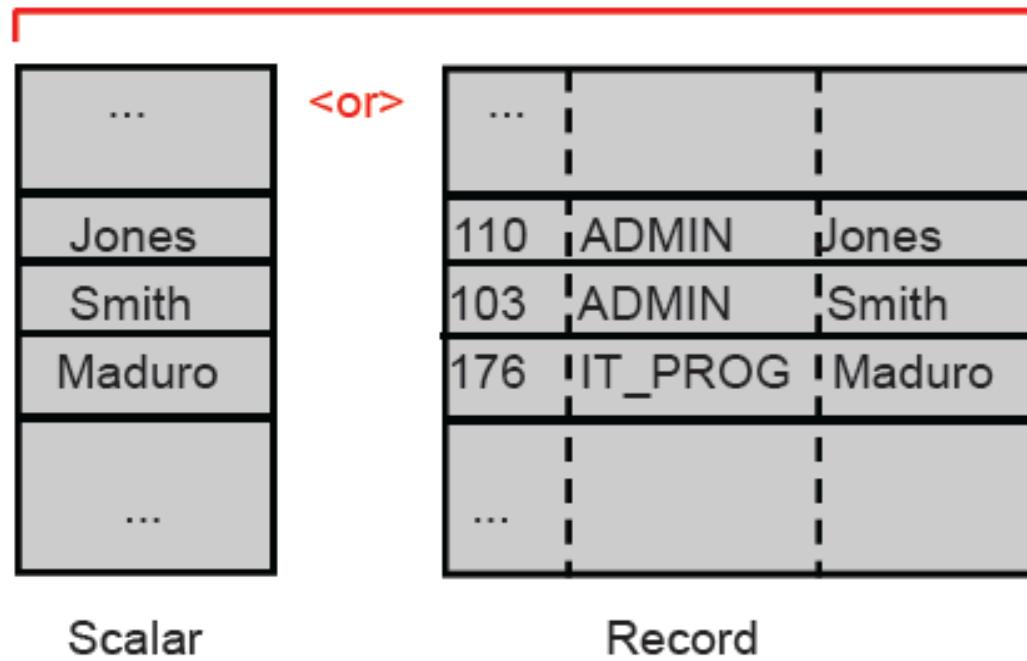
- Tabloul indexat *PL/SQL* are două componente: o coloană ce cuprinde cheia primară (integer/string) pentru acces la liniile tabloului și o coloană care include valoarea efectivă a elementelor tabloului

Key	Values
1	JONES
2	HARDEY
3	MADURO
4	KRAMER

Structura tablou indexat



2
---- Values column ----



Declarare tip date tablou indexat

Declararea tipului *TABLE* se face respectând următoarea sintaxă:

```
TYPE nume_tip IS TABLE OF  
  {tip_coloană / variabilă%TYPE /  
   nume_tabel.coloană%TYPE [NOT NULL] /  
   nume_tabel%ROWTYPE}  
INDEX BY tip_indexare;
```

Identifierul *nume_tip* este numele noului tip definit care va fi specificat în declararea tabloului *PL/SQL*, iar *tip_coloană* este un tip scalar simplu (**sau alt tip definit anterior**).

Initial unicul tip de indexare acceptat era *INDEX BY BINARY_INTEGER*, ulterior au fost adăugate următoarele opțiuni pentru *tip_indexare*: *PLS_INTEGER*, *NATURAL*, *POSITIVE*, *VARCHAR2(n)* sau chiar indexarea după un tip declarat cu *%TYPE*.

Nu sunt permise indexările *INDEX BY NUMBER*, *INDEX BY INTEGER*, *INDEX BY DATE*, *INDEX BY VARCHAR2*, *INDEX BY CHAR(n)* sau indexarea după un tip declarat cu *%TYPE* în care intervene unul dintre tipurile enumerate anterior

Observații:

- Elementele unui tablou indexat **nu sunt într-o ordine particulară** și pot fi inserate cu chei arbitrate.
- Deoarece nu există constrângeri de dimensiune, **dimensiunea tabloului se modifică dinamic**.
- Tabloul indexat *PL/SQL* **nu poate fi inițializat** în declararea sa.
- Un tablou indexat **neinițializat este vid** (nu conține nici valori, nici chei).
- Un element al tabloului este nedefinit atât timp cât nu are atribuită o valoare efectivă.
- Inițial, **un tablou indexat este nedens**. După declararea unui tablou se poate face referire la liniile lui prin precizarea valorii cheii primare.
- Dacă se face referire la o linie care nu există, atunci se produce excepția *NO_DATA_FOUND*.
- Dacă se dorește contorizarea numărului de linii, trebuie declarată o variabilă în acest scop sau poate fi utilizată **o metodă asociată tabloului**.
- Deoarece numărul de linii nu este limitat, operația de adăugare de linii este restricționată doar de dimensiunea memoriei alocate.

Exemple

```
...
DECLARE
  TYPE ename_table_type IS TABLE OF
    employees.last_name%TYPE
    INDEX BY PLS_INTEGER;
  TYPE hiredate_table_type IS TABLE OF DATE
    INDEX BY PLS_INTEGER;
  ename_table ename_table_type;
  hiredate_table hiredate_table_type;
BEGIN
  ename_table(1) := 'CAMERON';
  hiredate_table(8) := SYSDATE + 7;
  IF ename_table.EXISTS(1) THEN
    INSERT INTO ...
  ...
END;
/
...
```

Exemple

```
DECLARE
    TYPE org_table_type IS TABLE OF organizator%ROWTYPE
        INDEX BY BINARY INTEGER;
    org_table  org_table_type;
    i          NUMBER;
BEGIN
    IF org_table.COUNT <>0 THEN
        i := org_table.LAST+1;
    ELSE i:=1;
    END IF;
    org_table(i).cod_org := 752;
    org_table(i).nume := 'Grigore Ion';
    org_table(i).adresa := 'Calea Plevnei 18 Sibiu';
    org_table(i).tip := 'persoana fizica';
    INSERT INTO organizator VALUES (org_table(i));
    org_table.DELETE; -- sterge toate elementele
    DBMS_OUTPUT.PUT_LINE('Dupa aplicarea metodei DELETE
        sunt'||TO_CHAR(org_table.COUNT)||' elemente');
END;
```

Metode asociate

- EXISTS
- COUNT
- FIRST
- LAST
- PRIOR
- NEXT
- DELETE

Descriere

- EXISTS (*n*) – boolean; verifica daca exista componenta *n*;
- COUNT – numarul de elemente din colectie;
- FIRST – NULL sau indicele primei componente;
- LAST – NULL sau indicele ultimei componente ;
- PRIOR (*n*) – indicele componentei anterioare;
- NEXT (*n*) – indicele componentei urmatoare;
- DELETE, DELETE (*n*) , DELETE (*m, n*) – indicele utilizata pentru stergerea de componente;
- EXTEND (*n*) , TRIM (*n*) – adauga/sterge *n* componente;
- LIMIT – intoarce numarul maxim de elemente al unei colectii;

Exemple

```
DECLARE
  TYPE dept_table_type
  IS
    TABLE OF departments%ROWTYPE INDEX BY VARCHAR2(20);
  dept_table dept_table_type;
  -- Each element of dept_table is a record
BEGIN
  SELECT * INTO dept_table(1) FROM departments
  WHERE department_id = 10;
  DBMS_OUTPUT.PUT_LINE(dept_table(1).department_id ||
  ' || '
  dept_table(1).department_name || ' ' ||
  dept_table(1).manager_id);
END;
```

Vectori

Vectorii (*varray*) sunt structuri asemănătoare vectorilor din limbajele *C* sau *Java*. Spre deosebire de tablourile indexate, vectorii au o dimensiune maximă (constantă) stabilită la declarare. În special, se utilizează pentru modelarea relațiilor *one-to-many*, atunci când numărul maxim de elemente din partea „*many*“ este cunoscut și ordinea elementelor este importantă.

Vectorii reprezintă **structuri dense**. Fiecare element are un index care dă poziția sa în vector și care este folosit pentru accesarea elementelor particulare. Limita inferioară a indicelui este 1. Vectorul poate conține un număr variabil de elemente, de la 0 (vid) la numărul maxim specificat obligatoriu în definiția sa.

Tipul de date vector este declarat utilizând sintaxa:

TYPE nume_tip IS

{ VARRAY | VARYING ARRAY } (lungime_maximă)

OF tip_elemente [NOT NULL];

Vectori

```
DECLARE
    TYPE secenta IS VARRAY(5) OF VARCHAR2(10);
    v_sec  secenta := secenta ('alb', 'negru',
'rosu', 'verde');
BEGIN
    v_sec (3) := 'rosu';
    v_sec.EXTEND; -- adauga un element null
    v_sec(5) := 'albastru';
    -- extinderea la 6 elemente va genera
    -- eroarea ORA-06532
    v_sec.EXTEND;
END;
```

Tablouri imbricate

Tablourile imbricate (*nested table*) sunt tablouri indexate a căror dimensiune nu este stabilită. Numărul maxim de linii ale unui tablou imbricat este dat de capacitatea maximă 2 GB.

Un tablou imbricat este o mulțime neordonată de elemente de același tip. Valorile de acest tip:

- pot fi stocate în baza de date,
- pot fi prelucrate direct în instrucțiuni *SQL*
- au excepții predefinite proprii.

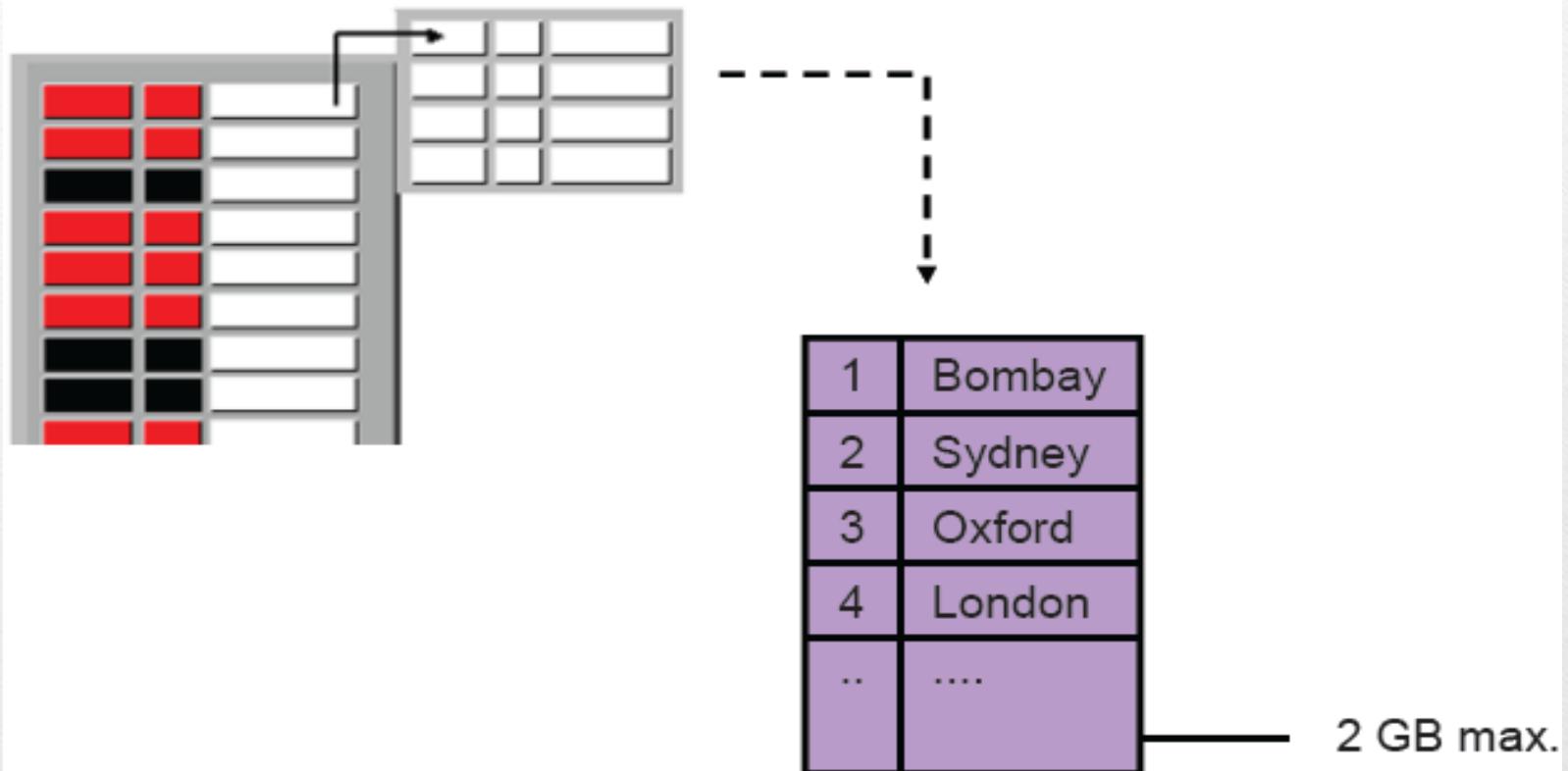
Sistemul *Oracle* nu stochează liniile unui tablou imbricat într-o ordine particulară. Dar, când se regăsește tabloul în variabile *PL/SQL*, liniile vor avea indici consecutivi începând cu valoarea 1. Inițial, aceste tablouri sunt structuri dense, dar se poate ca în urma prelucrării să nu mai aibă indici consecutivi.

Comanda de declarare a tipului de date tablou imbricat are sintaxa:

TYPE nume_tip IS TABLE OF tip_elemente [NOT NULL];

Identifierul *nume_tip* reprezintă numele noului tip de date tablou imbricat, iar *tip_elemente* este tipul fiecărui element din tabloul imbricat, care poate fi un tip definit de utilizator sau o expresie cu *%TYPE*, respectiv *%ROWTYPE*.

Tablouri imbricate



Exemple

```
DECLARE
    TYPE numartab IS TABLE OF NUMBER;
    -- se creeaza un tablou cu un singur element
    v_tab_1  numartab := numartab(-7);
    -- se creeaza un tablou cu 4 elemente
    v_tab_2  numartab := numartab(7,9,4,5);
    -- se creeaza un tablou fara nici un element
    v_tab_3  numartab := numartab();
BEGIN
    v_tab_1(1) := 57;
    FOR j IN 1..4 LOOP
        DBMS_OUTPUT.PUT_LINE (v_tab_2(j) || ' ');
    END LOOP;
END;
```

Se observă că singura diferență sintactică între tablourile indexate și cele imbricate este absența clauzei *INDEX BY BINARY_INTEGER*. Mai exact, dacă această clauză lipsește, tipul este tablou imbricat.

Observatii:

- Spre deosebire de tablourile indexate, vectorii și tablourile imbricate pot să apară în definirea tabelelor bazei de date
- Tablourile indexate pot avea indice negativ, domeniul permis pentru index fiind $-2147483647..2147483647$, iar pentru tabele imbricate domeniul indexului este $1..2147483647$
- Tablourile imbricate, spre deosebire de tablourile indexate, pot fi prelucrate prin comenzi *SQL*.
- Tablourile imbricate trebuie inițializate și/sau extinse pentru a li se adăuga elemente.

Initializare sau nu?

```
DECLARE
  TYPE alfa IS TABLE OF VARCHAR2(50);
  tab1 alfa; -- creeaza un tablou (atomic) null
  tab2 alfa := alfa(); --tabloul nu este null, e initializat
BEGIN
  IF tab1 IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('tab1 este NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('tab1 este NOT NULL');
  END IF;
  IF tab2 IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('tab2 este NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('tab2 este NOT NULL');
  END IF;
END;
```

În urma execuției acestui bloc se obține următorul rezultat:

```
tab1 este NULL
tab2 este NOT NULL
```

Tipuri de date stocate

Tipurile de date tablou imbricat, vector si object(record) pot fi utilizate drept câmpuri în tabelele bazei. Aceasta presupune că fiecare înregistrare din tabelul respectiv conține un obiect de tip colecție. Înainte de utilizare, tipul trebuie stocat în dicționarul datelor, deci trebuie declarat prin comanda:

CREATE TYPE nume_tip AS {TABLE | VARRAY} OF tip_elemente;

CREATE TYPE nume_tip AS OBJECT(camp1 type1, ...);

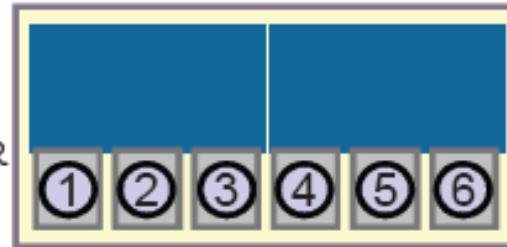
După crearea tabelului (prin comanda ***CREATE TABLE***), pentru fiecare câmp de tip tablou imbricat din tabel este necesară clauza de stocare:

NESTED TABLE nume_câmp STORE AS nume_tabel;

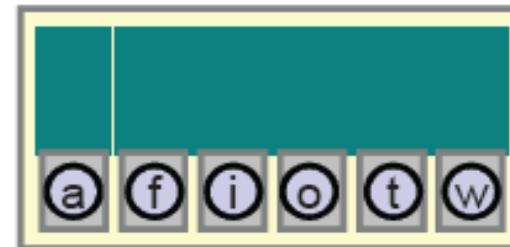
Imagine generala

Associative array

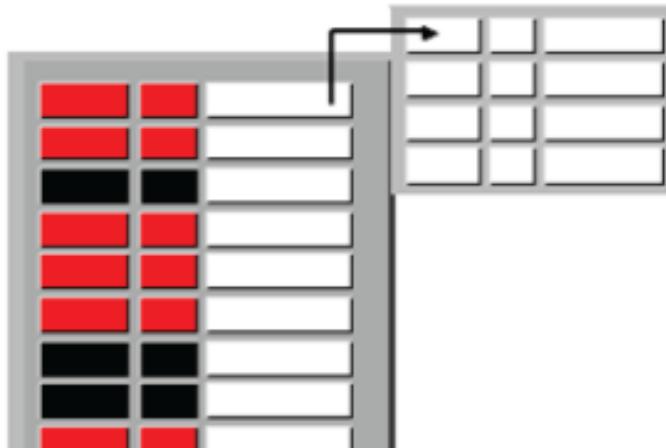
Index by
PLS_INTEGER



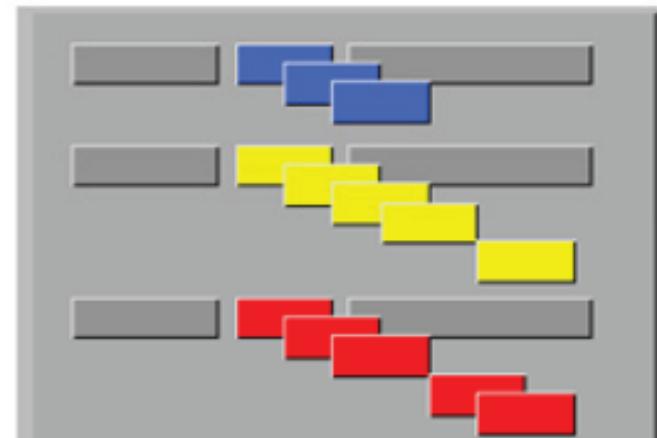
Index by
VARCHAR2



Nested table



Varray



Colectii pe mai multe niveluri

Se pot construi colecții pe mai multe niveluri (*multilevel collections*), prin urmare colecții ale căror elemente sunt, în mod direct sau indirect, colecții. În felul acesta pot fi definite structuri complexe: vectori de vectori, vectori de tablouri imbricate, tablou imbricat de vectori, tablou imbricat de tablouri imbricate, tablou imbricat sau vector de un tip definit de utilizator care are un atribut de tip tablou imbricat sau vector.

Aceste structuri complexe pot fi utilizate ca tipuri de date pentru definirea:

- coloanelor unui tabel relațional,
- atributelor unui obiect într-un tabel obiect,
- variabilelor *PL/SQL*.



Observații:

- Numărul nivelurilor de imbricare este limitat doar de capacitatea de stocare a sistemului.
- Pentru a accesa un element al colecției incluse sunt utilizate două seturi de paranteze.
- Obiectele de tipul colecție pe mai multe niveluri nu pot fi comparate.

Exercitii

1. Definiți un bloc anonim în cadrul căruia trebuie să definiți 3 tipuri de date (tablou, tablou imbricat, vector). Populați cu ajutorul clauzelor repetitive studiate 3 variabile de acest tip (minim 100 de numere) – clauza diferita pentru fiecare variabilă. Ulterior, pentru fiecare variabilă ștergeți componente care contin numere impare. Parcurgeți cele 3 variabile de la prima la ultima componentă și invers.
2. Definiți un tip de date care să permită să stocați o matrice pătratică de dimensiune n. Definiți și inițializați o astfel de variabilă cu valori care apar pe coloana *employees.employee_id*. Afipați conținutul acestei variabile parcurgând matricea în spirală.

Cursoare

Gestiunea cursoarelor in *PL/SQL*

Sistemul *Oracle* folosește, pentru a procesa o comandă *SQL*, o zona de memorie cunoscută sub numele de zona context (*context area*). Cand este procesată o instrucțiune *SQL*, server-ul *Oracle* deschide aceasta zona de memorie în care comanda este analizată sintactic și este executată.

Zona confine informații necesare procesării comenzii, cum ar fi:

- numarul de randuri procesate de instrucțiune;
- *unpointer* către reprezentarea internă a comenzii;
- în cazul unei cereri, mulțimea randurilor rezultate în urma execuției acestei comenzi (*active set*).

Un cursor este un *pointer* la aceasta zona context. Prin intermediul cursoarelor, un program *PL/SQL* poate controla zona context și transformările petrecute în urma procesării comenzii.

Există două tipuri de cursoare:

- *implicite*, generate de server-ul *Oracle* cand în partea executabilă a unui bloc *PL/SQL* apare o instrucțiune *SQL*;
- *explicite*, declarate și definite de către utilizator atunci cand o cerere (*SELECT*), care apare într-un bloc *PL/SQL*, întoarce mai multe linii ca rezultat.

Atât cursoarele *implicite* cat și cele *explicite* au o serie de atribute ale caror valori pot fi folosite în expresii. Lista atributelor este următoarea:

- *%ROWCOUNT*, care este de tip întreg și reprezintă numarul liniilor încărcate de cursor;
- *%FOUND*, care este de tip boolean și ia valoarea *TRUE* dacă ultima operație de încarcare (*FETCH*) dintr-un cursor a avut succes (în cazul cursoarelor explicite) sau dacă instrucțiunea *SQL* a întors cel puțin o linie (în cazul cursoarelor *implicite*);
- *%NOTFOUND*, care este de tip boolean și are semnificație opusă fata de cea a atributului *%FOUND*;
- *%ISOPEN*, care este de tip boolean și indică dacă un cursor este deschis (în cazul cursoarelor *implicite*, acest atribut are întotdeauna valoarea *FALSE*, deoarece un cursor implicit este închis de sistem imediat după execuțarea instrucțiunii *SQL* asociate).

Atributele pot fi referite prin expresia *SQL%nume_atribut*, în cazul cursoarelor *implicite*, sau prin *nume_cursor%nume_atribut*, în cazul unui cursor explicit. Ele pot să apară în comenzi *PL/SQL*, în funcții, în secțiunea de tratare a erorilor, dar nu pot fi utilizate în comenzi *SQL*.

Cursoare implice

Cand se proceseaza o comanda *LMD*, motorul *SQL* deschide un cursor implicit. Atributele scalare ale cursorului implicit (*SQL%ROWCOUNT*, *SQL%FOUND*, *SQL%NOTFOUND*, *SQL%ISOPEN*) furnizeaza informații referitoare la ultima comanda *INSERT*, *UPDATE*, *DELETE* sau *SELECT INTO* executata. înainte ca *Oracle* să deschida cursorul *SQL* implicit, attributele acestuia au valoarea *null*.

în *Oracle*, pentru cursoare implice a fost introdus atributul compus *%BULK_ROWCOUNT*, care este asociat comenzi *FORALL*. Atributul are semantica unui tablou indexat. Componenta *%BULK_ROWCOUNT(j)* confine numărul de linii procesate de a j-a execute a unei comenzi *INSERT*, *DELETE* sau *UPDATE*. Dacă a j-a execute nu afectează nici o linie, atunci atributul returnează valoarea 0. Comanda *FORALL* și atributul *%BULK_ROWCOUNT* au aceeași indici, deci folosesc același domeniu. Dacă *%BULK_ROWCOUNT(j)* este zero, atributul *%FOUND* este *FALSE*.

Exemplu:

în exemplul care urmează, comanda *FORALL* inserează un număr arbitrar de linii la fiecare iterare, iar după fiecare iterare atributul *%BULK_ROWCOUNT* returnează numărul acestor linii inserate.

```
SET SERVEROUTPUT ON
DECLARE
  TYPE alfa IS TABLE OF NUMBER;
  beta alfa;
BEGIN
  SELECT cod_artist BULK COLLECT INTO beta FROM artist;
  FORALL j IN 1..beta.COUNT INSERT INTO tab_art
    SELECT cod_artist,cod_opera FROM opera
    WHERE cod_artist = beta(j);
  FOR j IN 1..beta.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE ('Pentru artistul avand codul '
    || beta(j) || ' au fost inserate ' || SQL%BULK_ROWCOUNT(j)
    || ' înregistrări (opere de artă)');
  END LOOP;
  DBMS_OUTPUT.PUT_LINE ('Numarul total de înregistrări
  inserate este ' || SQL%ROWCOUNT);
END;
/
SET SERVEROUTPUT OFF
```

Cursoare explicite

Pentru gestiunea cursoarelor explicite sunt necesare urmatoarele etape:

- declararea cursorului (atribuirea unui nume și asocierea cu o comandă *SELECT*);
- deschiderea cursorului pentru cerere (executarea interogării asociate și determinarea multimii rezultat);
- recuperarea liniilor rezultatului în variabile *PL/SQL*;
- inchiderea cursorului (eliberarea resurselor relative la cursor).

Prin urmare, pentru a utiliza un cursor, el trebuie declarat în secțiunea declarativa a programului, trebuie deschis în partea executabilă, urmand să fie utilizat apoi pentru extragerea datelor. Dacă nu mai este necesar în restul programului, cursorul trebuie să fie închis.

DECLARE

declarare cursor BEGIN

deschidere cursor (OPEN)

WHILE raman linii de recuperat LOOP recuperare linie rezultat (FETCH)

END LOOP

inchidere cursor (CLOSE)

END;

Pentru a controla activitatea unui cursor sunt utilizate comenzi

DECLARE, OPEN,, FETCH și CLOSE.

Declararea unui cursor explicit

Prin declaratia *CURSOR* în cadrul comenzi *DECLARE* este definit un cursor explicit și este precizată structura cererii care va fi asociată acestuia.

Declaratia *CURSOR* are urmatoarea forma sintactica:

CURSOR nume cursor IS comanda select

Identifierul *nume cursor* este numele cursorului, iar *comanda select* este cererea *SELECT* care va fi procesata.

Observații:

- Comanda *SELECT* care apare în declararea cursorului, nu trebuie să includă clauza *INTO*.

- Daca se cere procesarea liniilor intr-o anumita ordine, atunci in cerere este utilizata clauza *ORDER BY*.
- Variabilele care sunt referite in comanda de selectare trebuie declarate inaintea comenzii *CURSOR*. Ele sunt considerate variabile de legatura.
- Daca in lista comenzii *SELECT* apare o expresie, atunci pentru expresia respectiva trebuie utilizat un *alias*, iar campul expresie se va referi prin acest *alias*.
- Numele cursorului este un identificator unic in cadrul blocului, care nu poate sa apara intr-o expresie si caruia nu i se poate atribui o valoare.

Deschiderea unui cursor explicit

Comanda *OPEN* executa cererea asociata cursorului, identifica multimea liniilor rezultat si pozitioneaza cursorul inaintea primei linii.

Deschiderea unui cursor se face prin comanda:

OPEN numecursor;

Identifierul *nume cursor* reprezinta numele cursorului ce va fi deschis.

La deschiderea unui cursor se realizeaza urmatoarele operatii:

- se evaluateaza cererea asociata (sunt examineate valorile variabilelor de legatura ce apar in declaratia cursorului);
- este determinata multimea rezultat (*active set*) prin executarea cererii *SELECT*, avand in vedere valorile de la pasul anterior;
- *pointer-ul* este pozitionat la prima linie din multimea activa.

Incarcarea datelor dintr-un cursor explicit

Comanda *FETCH* regaseste liniile rezultatului din multimea activa.

FETCH realizeaza urmatoarele operatii:

- anseaza *pointer-ul* la urmatoarea linie in multimea activa (*pointer-ul* poate avea doar un sens de deplasare de la prima spre ultima inregistrare);
- citeste datele liniei curente in variabile *PL/SQL*;
- daca *pointer-ul* este pozitionat la sfarsitul multimii active atunci seiese din bucla cursorului.

Comanda *FETCH* are urmatoarea sintaxa:

FETCH nume_cursor

INTO {nume}variabila [, nume variabila] ... | nume inregistrare);

Identifierul *nume cursor* reprezinta numele unui cursor declarat si deschis anterior. Variabila sau lista de variabile din clauza *INTO* trebuie sa fie

compatibila (ca ordine și tip) cu lista selectata din cererea asociata cursorului. La un moment dat, comanda *FETCH* regasește o singura linie. Totuși, în ultimele versiuni *Oracle* pot fi incarnate mai multe linii (la un moment dat) într-o colectie, utilizand clauza *BULK COLLECT*.

Exemplu:

în exemplul care urmează se încarcă date dintr-un cursor în două colectii.

```
DECLARE
  TYPE ccopera IS TABLE OF opera.cod_opera%TYPE;
  TYPE ctopera IS TABLE OF opera.titlu%TYPE;
  cod1 ccopera; titlu1 ctopera;
  CURSOR alfa IS SELECT cod_opera, titlu
    FROM opera
   WHERE stil = 'impressionism';
BEGIN
  OPEN alfa;
  FETCH alfa BULK COLLECT INTO cod1, titlu1;
  CLOSE alfa;
END;
```

Inchiderea unui cursor explicit

Dupa ce a fost procesata multimea activa, cursorul trebuie inchis. Prin aceasta se operează, *PL/SQL* este informat ca programul a terminat folosirea cursorului și resursele asociate acestuia pot fi eliberate. Aceste resurse includ spațiul utilizat pentru memorarea multimii active și spațiul temporar folosit pentru determinarea multimii active.

Cursorul va fi inchis prin comanda *CLOSE*, care are urmatoarea sintaxă:

CLOSE numecursor;

Identifierul *nume cursor* este numele unui cursor deschis anterior. Pentru a reutiliza cursorul este suficient ca acesta să fie redeschis. Dacă se încearcă să se încarcă date într-un cursor inchis, atunci apare excepția *INVALID_CURSOR*. Un bloc *PL/SQL* poate să se termine fără să se închidă cursorurile, dar acest lucru nu este indicat, deoarece este bine ca resursele să fie eliberate.

Exemplu:

Pentru toți artiștii care au opere de artă expuse în muzeu să se insereze în tabelul *temp* informații referitoare la numele acestora și anul nașterii.

```
DECLARE
  v_nume      artist.nume%TYPE;
  v_an_nas    artist.an_nastere%TYPE;
```

```

CURSOR info IS
    SELECT DISTINCT nume, an_nastere FROM artist;
BEGIN
    OPEN info;
    LOOP
        FETCH info INTO v_num, v_an_nas;
        EXIT WHEN info%NOTFOUND;
        INSERT INTO temp
            VALUES (v_num || TO_CHAR(v_an_nas));
    END LOOP;
    CLOSE info;
    COMMIT;
END;

```

Valorile atributelor unui cursor explicit sunt prezentate în urmatorul tabel:

		%FOUND	%ISOPEN	%NOTFOUN D	%ROWCOUNT
OPEN	înainte	Excepție	<i>False</i>	Excepție	Excepție
	Dupa	<i>Null</i>	<i>True</i>	<i>Null</i>	0
Prima	înainte	<i>Null</i>	<i>True</i>	<i>Null</i>	0
Incarcare	Dupa	<i>True</i>	<i>True</i>	<i>False</i>	1
Urmatoarea	înainte	<i>True</i>	<i>True</i>	<i>False</i>	1
incarcare	Dupa	<i>True</i>	<i>True</i>	<i>False</i>	Depinde de date
Ultima	înainte	<i>True</i>	<i>True</i>	<i>False</i>	Depinde de date
incarcare	Dupa	<i>False</i>	<i>True</i>	<i>True</i>	Depinde de date
CLOSE	înainte	<i>False</i>	<i>True</i>	<i>True</i>	Depinde de date
	Dupa	Excepție	<i>False</i>	Excepție	Excepție

După prima incarcare, dacă mulimea rezultat este vida, **%FOUND** va fi **FALSE**, **%NOTFOUND** va fi **TRUE**, iar **%ROWCOUNT** este 0.

intr-un pachet poate fi separată specificarea unui cursor de corpul acestuia. Cursorul va fi declarat în specificația pachetului prin comanda:

CURSOR nume cursor [(parametru [, parametru]...)]

RETURN tip_returnat;

în felul acesta va crea flexibilitatea programului, putând fi modificat doar corpul cursorului, fără a schimba specificația.

Exemplu:

```

CREATE PACKAGE exemplu AS
CURSOR alfa      (p_valoare_min      NUMBER) RETURN
opera%ROWTYPE;

-- declaratie specificatie cursor END exemplu;
CREATE PACKAGE BODY exemplu AS
CURSOR alfa (p_valoare_min NUMBER) RETURN opera%ROWTYPE IS
SELECT * FROM opera WHERE valoare > p_valoare_min;
-- definire corp cursor
END exemplu;

```

Procesarea liniilor unui cursor explicit

Pentru procesarea diferitelor linii ale unui cursor explicit se folosește operalia de ciclare (*LOOP*, *WHILE*, *FOR*), prin care la fiecare iterate se va încarca o nouă linie. Comanda *EXIT* poate fi utilizată pentru ieșirea din ciclu, iar valoarea atributului *%ROWCOUNT* pentru terminarea ciclului.

Procesarea liniilor unui cursor explicit se poate realiza și cu ajutorul unui ciclu *FOR* special, numit ciclu cursor. Pentru acest ciclu este necesara doar declararea cursorului, operabile de deschidere, încarcare și închidere ale acestuia fiind implicate.

Comanda are urmatoarea sintaxă:

FOR nume_inregistrare IN nume_cursor LOOP secvență_de_instrucțiuni;
END LOOP;

Variabila *nume_inregistrare* (care controlează ciclul) nu trebuie declarată. Domeniul ei este doar ciclul respectiv.

Pot fi utilizate cicluri cursor speciale care folosesc subcereri, iar în acest caz nu mai este necesara nici declararea cursorului. Exemplul care urmează este concludent în acest sens.

Exemplu:

Să se calculeze, utilizând un ciclu cursor cu subcereri, valoarea operelor de artă expuse într-o galerie al cărei cod este introdus de la tastatura. De asemenea, să se obțină media valorilor operelor de artă expuse în galeria respectivă.

```

SET SERVEROUTPUT ON
ACCEPT p_galerie PROMPT 'Dati codul galeriei:'
DECLARE
v_cod_galerie galerie.cod_galerie%TYPE:=&p_galerie;

```

```

val      NUMBER
media   ;
i       INTEGER
BEGIN    ;
val:=0;
i:=0;
FOR numar_opera IN
(SELECT cod_opera, valoare FROM opera
WHERE cod_galerie = v_cod_galerie) LOOP
val := val + numar_opera.valoare; i := i+1;
END LOOP;--inchidere implicita
DBMS_OUTPUT.PUT_LINE('Valoarea operelor de arta din
galeria cu numarul ' || TO_CHAR(v_cod_galerie) ||
' este ' || TO_CHAR(val));
IF i=0 THEN
DBMS_OUTPUT.PUT_LINE('Galeria nu are opere de arta'); ELSE
    media := val/i;
    DBMS_OUTPUT.PUT_LINE('Media valorilor operelor de
arta
din galeria cu numarul ' || TO_CHAR(v_cod_galerie)
|| ' este ' || TO_CHAR(media));
END IF;
END;
/
SET SERVEROUTPUT OFF

```

Cursoare parametrizate

Unei variabile de tip cursor ii corespunde o comanda *SELECT*, care nu poate fi schimbata pe parcursul programului. Pentru a putea lucra cu niște cursoare ale caror comenzi *SELECT* atașate depind de parametri ce pot fi modificati la momentul executiei, in *PL/SQL* s-a introdus notiunea de cursor parametrizat. Prin urmare, un cursor parametrizat este un cursor in care comanda *SELECT* atașata depinde de unul sau mai multi parametri.

Transmiterea de parametri unui cursor parametrizat se face in mod similar procedurilor stocate. Un astfel de cursor este mult mai ușor de interpretat și de intretinut, oferind și posibilitatea reutilizarii sale in blocul *PL/SQL*.

Declararea unui astfel de cursor se face respectand urmatoarea sintaxa:

CURSOR nume_cursor [(nume_parametru[, nume_parametru ...])] [RETURN tipreturnat]
IS comandaselect;

Identifierul *comanda select* este o instructivă *SELECT* fară clauza *INTO*, care returnă un tip înregistrare sau linie de tabel, iar *nume_parametru* are sintaxa:

nume_parametru [IN] *tip_parametru* [{:= | DEFAULT} *expresie*]

în aceasta declaratie, atributul *tip_parametru* reprezinta tipul parametrului, care este un tip scalar. Parametrii formali sunt de tip *IN* și, prin urmare, nu pot returna valori parametrilor actuali. Ei nu suportă constrângerea

NOT NULL.

Deschiderea unui astfel de cursor se face asemănător apelului unei funcții, specificând lista parametrilor actuali ai cursorului. În determinarea multimii active se vor folosi valorile actuale ale acestor parametri.

Sintaxa pentru deschiderea unui cursor parametrizat este:

OPEN numecursor [(*valoare_parametru* [, *valoare_parametru*] ...)];

Parametrii sunt specificați similar celor de la subprograme. Asocierea dintre parametrii formali și cei actuali se face prin:

- pozitie - parametrii formali și actuali sunt separați prin virgula;
- nume - parametrii actuali sunt aranjati într-o ordine arbitrară, dar cu o corespondență de forma *parametru formal => parametru actual*.

Dacă în definiția cursorului, toți parametrii au valori implicate (*DEFAULT*), cursorul poate fi deschis fără a specifica vreun parametru.

Exemplu:

Utilizând un cursor parametrizat să se obțină codurile operelor de artă din fiecare sală, identifierul sălii și al galeriei. Rezultatele să fie inserate în tabelul *mesaje*.

DECLARE

```
v_cod_sala    sala.cod_sala%TYPE;
v_cod_galerie galerie.cod_galerie%TYPE; v"car~
VARCHAR2(75);
CURSOR sala_cursor IS
SELECT    cod_sala,cod_galerie
FROM      sala;
CURSOR ope_cursor (v_id_sala NUMBER,v_id_galerie NUMBER) IS
SELECT cod_opera || cod_sala || cod_galerie FROM opera
WHERE    cod_sala = v_id_sala
AND      cod_galerie = v_id_galerie;
BEGIN      -
OPEN sala_cursor;
LOOP
FETCH sala_cursor INTO v_cod_sala,v_cod_galerie;
```

```

EXIT WHEN sala_cursor%NOTFOUND;
IF ope_cursor%ISOPEN THEN CLOSE ope_cursor;
END IF;
OPEN ope_cursor (v_cod_sala, v_cod_galerie);
LOOP
  FETCH ope_cursor INTO v_car;
  EXIT WHEN ope_cursor%NOTFOUND;
  INSERT INTO mesaje (rezultat)
    VALUES (v_car);
END LOOP;
CLOSE ope_cursor;
END LOOP;
CLOSE sala_cursor;
COMMIT;
END;

```

Cursoare *SELECT FOR UPDATE*

Uneori este necesara blocarea liniilor inainte ca acestea sa fie sterte sau reactualizate. Blocarea se poate realiza (atunci cand cursorul este deschis) cu ajutorul comenzii *SELECT* care confine clauza *FOR UPDATE*.

Declararea unui astfel de cursor se face conform sintaxei:

***CURSOR numecursor IS comandaselect
FOR UPDATE [OF listacampuri] [NOWAIT];***

Identifierul *lista campuri* este o lista ce include campurile tabelului care vor fi modificate. Atributul *NOWAIT* returneaza o eroare daca liniile sunt deja blocate de alta sesiune. Liniile unui tabel sunt blocate doar daca clauza *FOR UPDATE* se refera la coloane ale tabelului respectiv.

In momentul deschiderii unui astfel de cursor, liniile corespunzatoare mullimii active, determinate de clauza *SELECT*, sunt blocate pentru operalii de scriere (reactualizare sau stergere). In felul acesta este realizata consistent la citire a sistemului. De exemplu, aceasta situatie este utila cand se reactualizeaza o valoare a unei linii si trebuie avuta siguranla ca linia nu este schimbata de alt utilizator inaintea reactualizarii. Prin urmare, alte sesiuni nu pot schimba liniile din mullimea activa pana cand tranzactia nu este permanentizata sau anulata. Daca alta sesiune a blocat deja liniile din mullimea activa, atunci comanda *SELECT ... FOR UPDATE* va astepta (sau nu) ca aceste blocari sa fie eliberate. Pentru a trata aceasta situatie se utilizeaza clauza *WAIT*, respectiv *NOWAIT*.

In Oracle este utilizata sintaxa:

***SELECT ... FROM ... FOR UPDATE [OF lista campuri]
[{WAITn | NOWAIT}];***

Valoarea lui n reprezinta numarul de secunde de așteptare. Daca liniile nu sunt deblocate in n secunde, atunci se declanseaza eroarea *ORA-30006*, respectiv eroarea *ORA-00054*, dupa cum este specificata clauza *WAIT*, respectiv *NOWAIT*. Daca nu este specificata nici una din clauzele *WAIT* sau *NOWAIT*, sistemul așteapta pana ce linia este deblocata și atunci returneaza rezultatul comenzi *SELECT*.

Daca un cursor este declarat cu clauza *FOR UPDATE*, atunci comenziile *DELETE* și *UPDATE* corespunzatoare trebuie sa confina clauza *WHERE CURRENT OF numecursor*.

Aceasta clauza refera linia curenta care a fost gasita de cursor, permijand ca reactualizările și ștergerile sa se efectueze asupra acestei linii, fara referirea explicita a cheii primare sau pseudocoloanei *ROWID*. De subliniat ca instrucțiunile *UPDATE* și *DELETE* vor reactualiza numai coloanele listate in clauza *FOR UPDATE*.

Pseudocoloana *ROWID* poate fi utilizata daca tabelul referit in interogare nu are o cheie primara specificata. *ROWID-ul* fiecarei linii poate fi incarcat intro variabila *PL/SQL* (declarata de tipul *ROWID* sau *UROWID*), iar aceasta variabila poate fi utilizata in clauza *WHERE (WHERE ROWID = v rowid)*.

Dupa inchiderea cursorului este necesara comanda *COMMIT* pentru a realiza scrierea efectiva a modificarilor, deoarece cursorul lucreaza doar cu niște copii ale liniilor reale existente in tabele.

Deoarece blocările implicate de clauza *FOR UPDATE* vor fi eliberate de comanda *COMMIT*, nu este recomandata utilizarea comenzi *COMMIT* in interiorul ciclului in care se fac incarcari de date. Orice *FETCH* executat dupa *COMMIT* va eșua. in cazul in care cursorul nu este definit prin *SELECT...FOR UPDATE*, nu sunt probleme in acest sens și, prin urmare, in interiorul ciclului unde se fac schimbari ale datelor poate fi utilizat un *COMMIT*.

Exemplu:

Sa se dubleze valoarea operelor de arta pictate pe panza care au fost achiziționate înainte de 1 ianuarie 1956.

```
DECLARE
  CURSOR calc IS SELECT *
    FROM opera
   WHERE material = 'panza'
     AND data_achizitie <= TO_DATE('01-JAN-56', 'DD-MON-YY')
    FOR UPDATE OF valoare NOWAIT;
BEGIN
  FOR x IN calc LOOP UPDATE opera
```

```

SET      valoare = valoare*2
WHERE CURRENT OF calc;
END LOOP;
-- se permanentizeaza actiunea si se elibereaza blocarea
COMMIT;
END;

```

Cursoare dinamice

Toate exemplele considerate anterior se referă la cursoare statice. Unui cursor static îl se asociază o comandă *SQL* care este cunoscută în momentul în care blocul este compilat.

În *PL/SQL* a fost introdusă variabila cursor, care este de tip referință. Variabilele cursor sunt similare tipului *pointer* din limbajele *C* sau *Pascal*. Prin urmare, un cursor este un obiect static, iar un cursor dinamic este un *pointer* la un cursor.

În momentul declarării, variabilele cursor nu solicită o comandă *SQL* asociată. În acest fel, diferite comenzi *SQL* pot fi asociate variabilelor cursor, la diferite momente de timp. Acest tip de variabilă trebuie declarată, deschisă, încarcată și inchisă în mod similar unui cursor static.

Variabilele cursor sunt dinamice deoarece îl se pot asocia diferite interogări atât timp cât coloanele returnate de fiecare interogare corespund declarării variabilei cursor.

Aceste variabile sunt utile în transmiterea seturilor de rezultate între subprograme *PL/SQL* stocate și diferenți clienți. De exemplu, un *client OCI*, o aplicație *Oracle Forms* și server-ul *Oracle* pot referi aceeași zona de lucru (care conține mulțimea rezultat). Pentru a reduce traficul în rețea, o variabilă cursor poate fi declarată pe stația *client*, deschisă și se poate încărca date din ea pe *server*, apoi poate continua încarcarea, dar de pe stația *client* etc.

Pentru a crea o variabilă cursor este necesară definirea unui tip *REF CURSOR*, urmând apoi declararea unei variabile de tipul respectiv. Dupa ce variabilă cursor a fost declarată, ea poate fi deschisă pentru orice cerere *SQL* care returnează date de tipul declarat.

Sintaxa pentru declararea variabilei cursor este următoarea:

```

TYPE tip ref cursor IS REF CURSOR [RETURN tip returnat];
var cursor tip ref cursor;

```

Identifierul *var cursor* este numele variabilei cursor, *tip ref cursor* este un nou tip de date ce poate fi utilizat în declarările următoare ale variabilelor cursor, iar *tip returnat* este un tip înregistrare sau tipul unei linii dintr-un tabel al bazei. Acest tip corespunde coloanelor returnate de către orice cursor asociat variabilelor cursor de tipul definit. Dacă lipsește clauza *RETURN*,

cursorul poate fi deschis pentru orice cerere *SELECT*.

Dacă variabila cursor apare ca parametru într-un subprogram, atunci trebuie specificat tipul parametrului (tipul *REF CURSOR*) și forma acestuia (*IN* sau *IN OUT*).

Există anumite restricții referitoare la utilizarea variabilelor cursor:

- nu pot fi declarate într-un pachet;
- nu poate fi asignată valoarea *null* unei variabile cursor;
- nu poate fi utilizat tipul *REF CURSOR* pentru a specifica tipul unei coloane în comanda *CREATE TABLE*;
- nu pot fi utilizate operatorii de comparare pentru a testa egalitatea, inegalitatea sau valoarea *null* a variabilelor cursor;
- nu poate fi utilizat tipul *REF CURSOR* pentru a specifica tipul elementelor unei colecții (*varray*, *nested table*);
- nu pot fi folosite cu *SQL* dinamic în *Pro *C/C++*.

În cazul variabilelor cursor, instrucțiunile de deschidere (*OPEN*), încarcare (*FETCH*), inchidere (*CLOSE*) vor avea o sintaxă similară celor comentate anterior.

Comanda *OPEN...FOR* asociază o variabilă cursor cu o cerere multilinie, execută cererea, identifică mulțimea rezultat și pozitionează cursorul la prima linie din mulțimea rezultat. Sintaxa comenzii este:

OPEN {variabilacursor | wariabilacursorhost}

FOR {cerere_select /

§ir_dinamic [USING argumentbind [, argument_bind ...]]};

Identifierul *variabilă cursor* specifică o variabilă cursor declarată anterior, dar fără opțiunea *RETURN tip*, *cerere select* este interogarea pentru care este deschisă variabilă cursor, iar *§ir_dinamic* este o secvență de caractere care reprezintă cererea multilinie.

O opțiunea *§ir_dinamic* este specifică prelucrării dinamice a comenzilor, iar posibilitățile oferite de *SQL* dinamic vor fi analizate într-un capitol separat. Identifierul *wariabilă cursor host* reprezintă o variabilă cursor declarată într-un mediu gazdă *PL/SQL* (de exemplu, un program *OCI*).

Comanda *OPEN - FOR* poate deschide același cursor pentru diferite cereri. Nu este necesară inchiderea variabilei cursor înainte de a o redeschide. Dacă se redeschide variabilă cursor pentru o nouă cerere, cererea anterioară este pierdută.

Exemplu:

```

CREATE OR REPLACE PACKAGE alfa AS
TYPE ope_tip IS REF CURSOR RETURN opera%ROWTYPE;
PROCEDURE deschis_ope (ope_var IN OUT ope_tip,
alege IN NUMBER);
END alfa;
CREATE OR REPLACE PACKAGE BODY alfa AS
PROCEDURE deschis_ope (ope_var IN OUT ope_tip,
alege IN NUMBER) IS
BEGIN
IF alege = 1 THEN
OPEN ope_var FOR SELECT * FROM opera;
ELSIF alege = 2 THEN
OPEN ope_var FOR SELECT * FROM opera WHERE valoare>2000;
ELSIF alege = 3 THEN
OPEN ope_var FOR SELECT * FROM opera WHERE valoare=7777;
END IF;
END deschis_ope;
END alfa;

```

Exemplu:

in urmatorul exemplu se declara o variabila cursor care se asociaza unei comenzi *SELECT* (*SQL* dinamic) ce returneaza anumite linii din tabelul *opera*.

```

DECLARE
TYPE operaref IS REF CURSOR; opera var operaref;
mm_val INTEGER := 100000;
BEGIN
OPEN opera_var FOR
'SELECT cod_opera,valoare FROM opera WHERE valoare> :vv'
USING mm_val;
END;

```

Comanda *FETCH* returneaza o linie din multimea rezultat a cererii multi-linie, atribuie valori componentelor din lista cererii prin clauza *INTO*, avanseaza cursorul la urmatoarea linie. Sintaxa comenzii este:

```

FETCH {variabila_cursor | :variabila_cursor_host}
INTO {variabila [, variabila]... / inregistrare}
[BULK COLLECT INTO {numecolecvie [, nume_colecjie]...} |
{nume_array_host [, nume_array_host]...}
[LIMIT expresie_numerica]];

```

Clauza *BULK COLLECT* permite incarcarea tuturor liniilor simultan in una sau mai multe colectii. Atributul *nume_colecpe* indica o colectie declarata anterior, in care sunt depuse valorile respective, iar *nume_array_host* identifica un vector declarat intr-un mediu gazda *PL/SQL* și trimis lui *PL/SQL* ca variabila de legatura. Prin clauza *LIMIT* se limiteaza numarul liniilor incarcate din baza de date.

Exemplu:

```
DECLARE
  TYPE alfa IS REF CURSOR RETURN opera%ROWTYPE;
  TYPE beta IS TABLE OF opera.titlu%TYPE;
  TYPE gama IS TABLE OF opera.valoare%TYPE; var1
  alfa; var2 beta; var3 gama;
BEGIN
  OPEN alfa FOR SELECT titlu, valoare FROM opera;
  FETCH var1 BULK COLLECT INTO var2, var3;
  CLOSE var1;
END;
```

Comanda *CLOSE* dezactiveaza variabila cursor precizata. Ea are sintaxa:

CLOSE {variabila_cursor | :variabila_cursor_host}

Cursoarele și variabilele cursor nu sunt interoperabile. Nu poate fi folosita una din ele, cand este așteptata cealalta. Urmatoarea secvența este incorecta.

```
DECLARE
  TYPE beta IS REF CURSOR RETURN opera%ROWTYPE; gama
  beta;
BEGIN
  FOR k IN gama LOOP --nu este corect!
END;
```

Expresie cursor

In *Oracle* a fost introdus conceptul de expresie cursor (*cursor expression*), care returneaza un cursor imbricat (*nested cursor*).

Expresia cursor are urmatoarea sintaxa:

CURSOR (subcerere)

Fiecare linie din mulțimea rezultat poate conține valori uzuale și cursoare generate de subcereri. *PL/SQL* acceptă cereri care au expresii cursor în cadrul unei declaratii cursor, declaratii *REF CURSOR* și a variabilelor cursor.

Prin urmare, expresia cursor poate să apară într-o comandă *SELECT* ce este utilizată pentru deschiderea unui cursor dinamic. De asemenea, expresiile cursor pot fi folosite în cereri *SQL* dinamice sau ca parametri actuali într-un subprogram.

Un cursor imbricat este încarcat automat atunci când liniile care îl conțin sunt încarcate din cursorul „parinte“. El este închis dacă:

- este închis explicit de către utilizator;
- cursorul „parinte“ este reexecutat, închis sau anulat;
- apare o eroare în timpul unei încarcări din cursorul „parinte“

Există câteva restricții asupra folosirii unei expresii cursor:

- nu poate fi utilizată cu un cursor implicit;
- poate să apară numai într-o comandă *SELECT* care nu este imbricată în alta cerere (exceptând cazul în care este o subcerere chiar a expresiei cursor) sau ca argument pentru funcții tabel, în clauza *FROM* a lui *SELECT*;
- nu poate să apară în interogarea ce definește o vizualizare;
- nu se pot efectua operații *BIND* sau *EXECUTE* cu aceste expresii.

Exemplu:

Să se definișească un cursor care furnizează codurile operelor expuse în cadrul unei expoziții având un cod specificat (*val_cod*) și care se desfășoară într-o localitate precizată (*val_oraș*). Să se afișeze data cand a avut loc vernisajul acestei expoziții.

în acest caz cursorul returnează două coloane, cea de-a doua coloană fiind **un cursor imbricat**.

```
CURSOR alfa (val_cod NUMBER, val_oras VARCHAR2(20)) IS
  SELECT l.datai,
    CURSOR (SELECT d.cod_expo,
      CURSOR (SELECT f.cod_opera FROM
        figureaza_in f WHERE
        f.cod_expo=d.cod_expo) AS
      xx
      FROM expozitie d
      WHERE l.cod_expo = d.cod_expo) AS yy FROM
    locped l
    WHERE cod_expo = val_cod AND nume_oras= val_oras;
```

Exemplu:

Să se listeze numele galeriilor din muzeu și pentru fiecare galerie să se afișeze numele salilor din galeria respectivă.

Sunt prezentate doua variante de rezolvare. Prima varianta reprezinta o implementare simpla utilizand programarea secven^iala clasica, iar a doua utilizeaza expresii cursor pentru rezolvarea acestei probleme.

Varianta 1:

```
BEGIN
  FOR gal IN (SELECT cod_galerie, nume_galerie FROM galerie)
  LOOP
    DBMS_OUTPUT.PUT_LINE (gal.nume_galerie);
    FOR sal IN (SELECT cod_sala, nume_sala FROM sala
                 WHERE cod_galerie = gal.cod_galerie)
    LOOP
      DBMS_OUTPUT.PUT_LINE (sal.nume_sala);
    END LOOP;
  END LOOP;
END;
```

Varianta 2:

```
DECLARE
  CURSOR c_gal IS
    SELECT nume_galerie,
           CURSOR (SELECT nume_sala FROM sala s
                    WHERE s.cod_galerie = g.cod_galerie) FROM galerie
g;
  v_nume_gal    galerie.nume_galerie%TYPE;
  v~sala~    SYS_REFCURSOR;
  TYPE sala_nume IS TABLE OF sala.nume_sala%TYPE -
INDEX BY BINARY_INTEGER;
  v nume_sala_sala_nume;
BEGIN
  OPEN c_gal;
  LOOP
    FETCH c_gal INTO v_nume_gal, v_sala;
    EXIT WHEN c_gal%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (v_nume_gal);
    FETCH v_sala BULK COLLECT INTO v_nume_sala;
    FOR ind IN v_nume_sala.FIRST..v_nume_sala.LAST
    LOOP
      DBMS_OUTPUT.PUT_LINE (v_nume_sala (ind));
    END LOOP;
  END LOOP;
  CLOSE c_gal;
END;
```

Subprogramme

Subprograme în *PL/SQL*

Noțiunea de subprogram (procedură sau funcție) a fost concepută cu scopul de a grupa o mulțime de comenzi *SQL* cu instrucțiuni procedurale pentru a construi o unitate logică de tratament.

Unitățile de program ce pot fi create în *PL/SQL* sunt:

- **subprograme locale** (definite în partea declarativă a unui bloc *PL/SQL* sau a unui alt subprogram);
- **subprograme independente** (stocate în baza de date și considerate drept obiecte ale acesteia);
- **subprograme împachetate** (definite într-un pachet care încapsulează proceduri și funcții).

Procedurile și funcțiile stocate sunt unități de program *PL/SQL* apelabile, care există ca obiecte în schema bazei de date *Oracle*. Recuperarea unui subprogram (în cazul unei corecții) nu cere recuperarea întregii aplicații. Subprogramul încărcat în memorie pentru a fi executat, poate fi partajat între obiectele (aplicații) care îl solicită.

Este important de făcut **distincție între procedurile stocate și procedurile locale** (declarate și folosite în blocuri anonte).

- Procedurile care sunt declarate și apelate în blocuri anonte sunt temporare. O procedură stocată (creată cu *CREATE PROCEDURE* sau conținută într-un pachet) este permanentă în sensul că ea poate fi invocată printr-un script *iSQL*Plus*, un subprogram *PL/SQL* sau un declanșator.
- Procedurile și funcțiile stocate, care sunt compilate și stocate în baza de date, nu mai trebuie să fie compilate a doua oară pentru a fi executate, în timp ce procedurile locale sunt compilate de fiecare dată când este executat blocul care conține procedurile și funcțiile respective.
- Procedurile și funcțiile stocate pot fi apelate din orice bloc de către utilizatorul care are privilegiul *EXECUTE* asupra subprogramului, în timp ce procedurile și funcțiile locale pot fi apelate numai din blocul care le conține.

Când este creat un subprogram stocat, utilizând comanda *CREATE OR REPLACE*, subprogramul este depus în dicționarul datelor. Este depus atât textul sursă, cât și forma compilată (*p-code*). Când subprogramul este apelat, *p-code* este citit de pe disc, este depus în *shared pool*, unde poate fi accesat de mai mulți utilizatori și este executat dacă este necesar. El va părăsi *shared pool* conform algoritmului *LRU* (*least recently used*).

Subprogramele se pot declara în blocuri *PL/SQL*, în alte subprograme sau în pachete, dar la sfârșitul secțiunii declarative. La fel ca blocurile *PL/SQL* anonime, subprogramele conțin o parte declarativă, o parte executabilă și optional, o parte de tratare a erorilor.

Crearea subprogramelor stocate

- 1) se editează subprogramul (*CREATE PROCEDURE* sau *CREATE FUNCTION*) și se salvează într-un *script file SQL*;
- 2) se încarcă și se execută acest *script file*, este compilat codul sursă, se obține *p-code* (subprogramul este creat);
- 3) se utilizează comanda *SHOW ERRORS* pentru vizualizarea eventualelor erori la compilare ale procedurii care a fost cel mai recent compilată sau *SHOW ERRORS PROCEDURE nume* pentru orice procedura compilată anterior (nu poate fi invocată o procedura care conține erori de compilare);
- 4) se execută subprogramul pentru a realiza acțiunea dorită (de exemplu, procedura poate fi executată de câte ori este necesar, utilizând comanda *EXECUTE* din *iSQL*Plus*) sau se invocă funcția dintr-un bloc *PL/SQL*.

Când este apelat subprogramul, motorul *PL/SQL* execută *p-code*.

```
CREATE PROCEDURE add_dept IS
v_dept_id dept.department_id%TYPE;
v_dept_name dept.department_name%TYPE;
BEGIN
v_dept_id:=280;
v_dept_name:='ST-Curriculum';
INSERT INTO dept(department_id,department_name)
VALUES(v_dept_id,v_dept_name);
DBMS_OUTPUT.PUT_LINE(' Inserted '|| SQL%ROWCOUNT
||' row ');
END;
...
BEGIN
    add_dept;
END;
```

Dacă există erori la compilare și se fac corecțiile corespunzătoare, atunci este necesară fie comanda *DROP PROCEDURE* (respectiv *DROP FUNCTION*), fie sintaxa *OR REPLACE* în cadrul comenzi *CREATE*.

Când este apelată o procedură *PL/SQL*, server-ul *Oracle* parcurge etapele:

- 1) Verifică dacă utilizatorul are privilegiul să execute procedura (fie pentru că el a creat procedura, fie pentru că i s-a dat acest privilegiu).
- 2) Verifică dacă procedura este prezentă în *shared pool*. Dacă este prezentă va fi executată, altfel va fi încărcată de pe disc în *database buffer cache*.
- 3) Verifică dacă starea procedurii este *validă* sau *invalidă*. Starea unei proceduri *PL/SQL* este *invalidă*, fie pentru că au fost detectate erori la compilarea procedurii, fie pentru că structura unui obiect s-a schimbat de când procedura a fost executată ultima oară. Dacă starea procedurii este *invalidă* atunci este recompilată automat. Dacă nici o eroare nu a fost detectată, atunci va fi executată noua versiune a procedurii.
- 4) Dacă procedura aparține unui pachet atunci toate procedurile și funcțiile pachetului sunt de asemenea încărcate în *database cache* (dacă ele nu erau deja acolo). Dacă pachetul este activat pentru prima oară într-o sesiune, atunci serverul va executa blocul de inițializare al pachetului.

```
CREATE [OR REPLACE] PROCEDURE / FUNCTION
procedure_name/function_name
[ (argument1 [mode1] datatype1,
argument2 [mode2] datatype2,
  . . . ) ]
IS | AS
subprogram_body;
```

Pentru a afișa codul unui subprogram, parametrii acestuia, precum și alte informații legate de subprogram poate fi utilizată comanda *DESCRIBE*.

Proceduri *PL/SQL*

Procedura *PL/SQL* este un program independent care se găsește compilat în schema bazei de date *Oracle*. Când procedura este compilată, identificatorul acesteia (stabilit prin comanda *CREATE PROCEDURE*) devine un nume obiect în dicționarul datelor. Tipul obiectului este *PROCEDURE*.

Sintaxa generală pentru crearea unei proceduri este următoarea:

```
[CREATE [OR REPLACE]] PROCEDURE nume_procedură
[(parametru[, parametru]...)] [AUTHID {DEFINER |
CURRENT_USER}] {IS | AS}
[PRAGMA AUTONOMOUS_TRANSACTION;]
[declarații locale]
BEGIN
partea executabilă
[EXCEPTION
partea de tratare a excepțiilor]
END [nume_procedură];
unde parametrii au următoarea formă sintactică:
nume_parametru [IN | OUT [NOCOPY] | IN OUT [NOCOPY]
tip_de_date{:= | DEFAULT} expresie]
```

Clauza *CREATE* permite ca procedura să fie stocată în baza de date. Când procedurile sunt create folosind clauza *CREATE OR REPLACE*, ele vor fi stocate în BD în formă compilată. Dacă procedura există, atunci clauza *OR REPLACE* va avea ca efect ștergerea procedurii și înlocuirea acesteia cu noua versiune. Dacă procedura există, iar *OR REPLACE* nu este prezent, atunci comanda *CREATE* va returna eroarea “*ORA-955: Name is already used by an existing object*”.

Clauza *AUTHID* specifică faptul că procedura stocată se execută cu drepturile proprietarului (implicit) sau ale utilizatorului curent. De asemenea, această clauză precizează dacă referințele la obiecte sunt rezolvate în schema proprietarului procedurii sau a utilizatorului curent.

Clauza *PRAGMA_AUTONOMOUS_TRANSACTION* anunță compilatorul *PL/SQL* că această procedură este autonomă (independentă). Tranzacțiile autonome permit suspendarea tranzacției principale, executarea unor instrucțiuni *SQL*, *commit-ul* sau *rollback-ul* acestor operații și continuarea tranzacției principale.

Parametrii formali (variabile declarate în lista parametrilor specificației subprogramului) pot să fie de tipul: `%TYPE`, `%ROWTYPE` sau un tip explicit fără specificarea dimensiunii.

Exemplu:

Să se creeze o procedură stocată care micșorează cu o cantitate dată (*cant*) valoarea polițelor de asigurare emise de firma ASIROM.

```
CREATE OR REPLACE PROCEDURE mic (cant IN NUMBER) AS
BEGIN
    UPDATE politaasig
    SET valoare = valoare - cant
    WHERE firma = 'ASIROM';
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR (-20010,'nu exista ASIROM');
END;
/
```

Dacă sunt operații de reactualizare în subprograme și există declanșatori relativ la aceste operații care nu trebuie considerați, atunci înainte de apelarea subprogramului declanșatorii trebuie dezactivați, urmând ca ei să fie reactivați după ce s-a terminat execuția subprogramului. De exemplu, în problema prezentată anterior ar trebui dezactivați declanșatorii referitori la tabelul *politaasig*, apelată procedura *mic* și în final reactivați acești declanșatori.

```
ALTER TABLE politaasig DISABLE ALL TRIGGERS;
EXECUTE mic(10000)
ALTER TABLE politaasig ENABLE ALL TRIGGERS;
```

Exemplu:

Să se creeze o procedură locală prin care se inserează informații în tabelul *editata_de*.

```
DECLARE
    PROCEDURE editare
        (v_cod_sursa    editata_de.cod_sursa%TYPE,
         v_cod_autor   editata_de.cod_autor%TYPE)
    IS
BEGIN
    INSERT INTO editata_de
    VALUES  (v_cod_sursa,v_cod_autor);
END;
BEGIN
...
editare(75643, 13579);      ...
END;
```

Procedurile stocate pot fi apelate:

- din corpul altrei proceduri sau a unui declanșator;
- interactiv de utilizator utilizând un instrument *Oracle*;
- explicit dintr-o aplicație (de exemplu, *SQL*Forms* sau utilizarea de precompilatoare).

Utilizarea (apelarea) unei proceduri se poate face:

- 1) prin comanda:

EXECUTE nume_procedură [(lista_parametri_actuali)];

- 2) în *PL/SQL* prin apariția numelui procedurii urmat de lista parametrilor actuali.

Funcții *PL/SQL*

Funcția *PL/SQL* este similară unei proceduri cu excepția că ea trebuie să întoarcă un rezultat. O funcție fără comanda *RETURN* va genera eroare la compilare.

Când funcția este compilată, identificatorul acesteia devine obiect în dicționarul datelor având tipul *FUNCTION*. Algoritmul din interiorul corpului subprogramului funcție trebuie să asigure că toate traectoriile sale conduc la comanda *RETURN*. Dacă o traectorie a algoritmului trimită în partea de tratare a erorilor, atunci *handler*-ul acesteia trebuie să includă o comandă *RETURN*. O funcție trebuie să aibă un *RETURN* în antet și cel puțin un *RETURN* în partea executabilă. Sintaxa simplificată pentru scrierea unei funcții este următoarea:

```

[CREATE [OR REPLACE]] FUNCTION nume_funcție
[(parametru[, parametru]...)]
RETURN tip_de_date
[AUTHID {DEFINER / CURRENT_USER}] [DETERMINISTIC]
{IS / AS}
[PRAGMA AUTONOMOUS_TRANSACTION;]
[declarații locale]
BEGIN
    partea executabilă
[EXCEPTION
    partea de mânuire a excepțiilor]
END [nume_funcție];

```

Opțiunea *tip_de_date* specifică tipul valorii returnate de funcție, tip care nu poate conține specificații de mărime. Dacă totuși sunt necesare aceste specificații se pot defini subtipuri, iar parametrii vor fi declarați de subtipul respectiv.

În interiorul funcției trebuie să apară *RETURN expresie*, unde *expresie* este valoarea rezultatului furnizat de funcție. Pot să fie mai multe comenzi *RETURN* într-o funcție, dar numai una din ele va fi executată, deoarece după ce valoarea este returnată, procesarea blocului incetează. Comanda *RETURN* (fără o expresie asociată) poate să apară și într-o procedură. În acest caz, ea va avea ca efect revenirea la comanda ce urmează instrucțiunii apelante.

Opțiunea *DETERMINISTIC* ajută optimizorul *Oracle* în cazul unor apeluri repetitive ale aceleiași funcții, având aceleiași argumente. Ea asigură folosirea unui rezultat obținut anterior.

În blocul *PL/SQL* al unei proceduri sau funcții stocate (defineste acțiunea efectuată de funcție) nu pot fi referite variabile *host* sau variabile *bind*.

O funcție poate accepta unul sau mai mulți parametri, dar trebuie să returneze o singură valoare. Ca și în cazul procedurilor, lista parametrilor este opțională. Dacă subprogramul nu are parametri, parantezele nu sunt necesare la declarare și la apelare.

Exemplu:

Să se creeze o funcție stocată care determină numărul operelor de artă realizate pe pânză, ce au fost achiziționate la o anumită dată.

```
CREATE OR REPLACE FUNCTION numar_opere
    (v_a IN opera.data_achizitie%TYPE)
RETURNS NUMBER AS
    alfa NUMBER;
BEGIN
    SELECT COUNT (ROWID)
    INTO   alfa
    FROM   opera
    WHERE  material='panza'
    AND    data_achizitie = v_a; RETURN alfa;
END numar_opere;
```

Dacă apare o eroare de compilare, utilizatorul o va corecta în fișierul editat și apoi va trimite fișierul modificat nucleului, cu opțiunea *OR REPLACE*.

Sintaxa pentru apelul unei funcții este:

```
[[schema.]nume_pachet] nume_funcție [@dblink]
[(lista_parametri_actuali)];
```

O funcție stocată poate fi apelată în mai multe moduri.

- 1) Apelarea funcției și atribuirea valorii acesteia într-o variabilă de legătură:

```
VARIABLE val NUMBER
```

```
EXECUTE :val := numar_operе(SYSDATE) PRINT val
```

Când este utilizată declarația **VARIABLE**, pentru variabilele *host* de tip **NUMBER** nu trebuie specificată dimensiunea, iar pentru cele de tip **CHAR** sau **VARCHAR2** valoarea implicită este 1 sau poate fi specificată o altă valoare între paranteze. **PRINT** și **VARIABLE** sunt comenzi *iSQL*Plus*.

- 2) Apelarea funcției într-o instrucțiune *SQL*:

```
SELECT numar_operе(SYSDATE)
```

```
FROM dual;
```

- 3) Apariția numelui funcției într-o comandă din interiorul unui bloc *PL/SQL* (de exemplu, într-o instrucțiune de atribuire):

```
ACCEPT data PROMPT 'dati data achizitionare'
```

```
DECLARE
```

```
    num      NUMBER;
```

```
    v_data opera.data_achizitie%TYPE := '&data' ;
```

```
BEGIN
```

```
    num := numar_operе(v_data);
```

```
    DBMS_OUTPUT.PUT_LINE('numarul operelor de artă
        achizitionate la data' || TO_CHAR(v_data) || este'
        || TO_CHAR(num));
```

```
END;
```

```
/
```

Exemplu:

Să se creeze o procedură stocată care pentru un anumit tip de operă de artă (dat ca parametru) calculează numărul operelor din muzeu de tipul respectiv, numărul de specialiști care au expertizat sau au restaurat aceste opere, numărul de expoziții în care au fost expuse, precum și valoarea nominală totală a acestora.

```
CREATE OR REPLACE PROCEDURE date_tip_operа
    (v_tip      opera.tip%TYPE) AS
FUNCTION nr_operе (v_tip opera.tip%TYPE)
RETURN NUMBER IS
    v_numar      NUMBER(3);
BEGIN
    SELECT COUNT(*) INTO v_numar
```

```

    FROM opera
    WHERE tip = v_tip;
    RETURN v_numar;
END;
FUNCTION valoare_totala (v_tip opera.tip%TYPE)
RETURN NUMBER IS
    v_numar
        opera.valoare%TYPE;
BEGIN
    SELECT SUM(valoare) INTO v_numar
        FROM opera
    WHERE tip = v_tip;
    RETURN v_numar;
END;
FUNCTION nr_specialisti (v_tip opera.tip%TYPE)
RETURN NUMBER IS
    v_numar      NUMBER(3);
BEGIN
    SELECT COUNT(DISTINCT studiaza.cod_specialist)
    INTO v_numar studiaza, opera
    FROM studiaza.cod_opera = opera.cod_opera
    WHERE
        AND opera.tip = v_tip;
    RETURN v_numar;
END;
FUNCTION nr_expozitii (v_tip opera.tip%TYPE)
RETURN NUMBER IS
    v_numar      NUMBER(3);
BEGIN
    SELECT COUNT(DISTINCT figureaza_in.cod_expozitie)
    INTO v_numar figureaza_in, opera
    FROM figureaza_in.cod_opera = opera.cod_opera
    WHERE
        AND opera.tip = v_tip; RETURN v_numar;
END;
BEGIN
DBMS_OUTPUT.PUT_LINE('Numarul operelor de arta este
'||nr_opere(v_tip));
DBMS_OUTPUT.PUT_LINE('Valoarea operelor de arta este
'||valoare_totala(v_tip));
DBMS_OUTPUT.PUT_LINE('Numarul de specialisti este
'||nr_specialisti(v_tip));
DBMS_OUTPUT.PUT_LINE('Numarul de expoziții este
'||nr_expozitii(v_tip));
END;

```

Instrucțiunea ***CALL***

O instrucțiune specifică pentru *Oracle* este comanda ***CALL*** care permite apelarea subprogramelor *PL/SQL* stocate (independente sau incluse în pachete) și a subprogramelor *Java*.

CALL este o comandă *SQL* care nu este validă într-un bloc *PL/SQL*. Poate fi utilizată în *PL/SQL* doar dinamic, prin intermediul comenzi ***EXECUTE IMMEDIATE***. Pentru executarea acestei comenzi, utilizatorul trebuie să aibă privilegiul ***EXECUTE*** asupra subprogramului. Comanda poate fi executată interactiv din *SQL*. Ea are sintaxa următoare:

```
CALL [schema.]nume_subprogram ([lista_parametri actuali])
      [@dblink_nume] [INTO :variabila_host]
```

Nume_subprogram este numele unui subprogram sau numele unei metode. Clauza *INTO* este folosită numai pentru variabilele de ieșire ale unei funcții. Dacă clauza *@dblink_nume* lipsește, sistemul se referă la baza de date locală, iar într-un sistem distribuit clauza specifică numele bazei care conține subprogramul.

Exemplu:

Sunt prezentate două exemple prin care o funcție *PL/SQL* este apelată din *SQL*Plus*, respectiv o procedură externă *C* este apelată, folosind *SQL* dinamic, dintr-un bloc *PL/SQL*.

```
CREATE OR REPLACE FUNCTION apelfunctie(a IN VARCHAR2)
  RETURN VARCHAR2 AS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Apel functie cu ' || a); RETURN a;
END apelfunctie;
/

SQL> --apel valid
SQL> VARIABLE v_iesire VARCHAR2(20)
SQL> CALL apelfunctie('Salut!') INTO :v_iesire
Apel functie cu Salut!
Call completed

DECLARE
  a NUMBER(7);
  x VARCHAR2(10);
BEGIN
  EXECUTE IMMEDIATE 'CALL alfa_extern_procedura (:aa,
  :xx)' USING a, x;
END;
```

Modificarea și suprimarea subprogramelor *PL/SQL*

Pentru a lua în considerare modificarea unei proceduri sau funcții, recompilarea acestora se face prin comanda:

ALTER {FUNCTION / PROCEDURE} [schema.]nume COMPILE;

Comanda recompilează doar procedurile catalogate standard. Procedurile unui pachet se recompilează într-o altă manieră.

Ca și în cazul tabelelor, funcțiile și procedurile pot fi suprimate cu ajutorul comenzi *DROP*. Aceasta presupune eliminarea subprogramelor din dicționarul datelor. *DROP* este o comandă ce aparține limbajului de definire a datelor, astfel că se execută un *COMMIT implicit* atât înainte, cât și după comandă.

Când este şters un subprogram prin comanda *DROP*, automat sunt revocate toate privilegiile acordate referitor la acest subprogram. Dacă este utilizată sintaxa *CREATE OR REPLACE*, privilegiile acordate asupra acestui obiect (subprogram) rămân aceleiași.

DROP {FUNCTION / PROCEDURE} [schema.]nume;

Transferarea valorilor prin parametri

Lista parametrilor unui subprogram este compusă din parametri de intrare (*IN*), de ieșire (*OUT*), de intrare/ieșire (*IN OUT*), separați prin virgulă.

Dacă nu este specificat nimic, atunci implicit parametrul este considerat *IN*. Un parametru formal cu opțiunea *IN* poate primi valori implicate chiar în cadrul comenzi de declarare. Acest parametru este *read-only* și deci nu poate fi schimbat în corpul subprogramului. El acționează ca o constantă. Parametrul actual corespunzător poate fi literal, expresie, constantă sau variabilă inițializată.

Un parametru formal cu opțiunea *OUT* este neinițializat și prin urmare, are automat valoarea *NULL*. În interiorul subprogramului, parametrilor cu opțiunea *OUT* sau *IN OUT* trebuie să li se asigneze o valoare explicită. Dacă nu se atribuie nici o valoare, atunci parametrul actual corespunzător va fi *NULL*. Parametrul actual trebuie să fie o variabilă, nu poate fi o constantă sau o expresie.

Dacă în procedură apare o excepție, atunci valorile parametrilor formali cu opțiunile *IN OUT* sau *OUT* nu sunt copiate în valorile parametrilor actuali.

Implicit, transmiterea parametrilor este prin valoare în cazul parametrilor *IN* și este prin referință în cazul parametrilor *OUT* sau *IN OUT*. Dacă pentru realizarea unor performanțe se dorește transmiterea prin referință și în cazul parametrilor *IN OUT* sau *OUT* atunci se poate utiliza opțiunea *NOCOPY*. Dacă opțiunea *NOCOPY* este asociată unui parametru *IN*, atunci va genera o eroare la compilare deoarece acești parametri se transmit de fiecare dată prin valoare.

Când este apelată o procedură *PL/SQL*, sistemul *Oracle* furnizează două metode pentru definirea parametrilor actuali:

- specificarea explicită prin nume;
- specificarea prin poziție.

Exemplu:

```
CREATE PROCEDURE p1(a IN NUMBER, b IN VARCHAR2,
                    c IN DATE, d OUT NUMBER) AS...;
```

Sunt prezentate diferite moduri pentru apelarea acestei proceduri.

```
DECLARE
    var_a  NUMBER;
    var_b  VARCHAR2;
    var_c  DATE;
    var_d  NUMBER;
BEGIN
    --specificare prin poziție
    p1(var_a,var_b,var_c,var_d);
    --specificare prin nume
    p1(b=>var_b,c=>var_c,d=>var_d,a=>var_a);
    --specificare prin nume și poziție
    p1(var_a,var_b,d=>var_d,c=>var_c);
END;
```

Exemplu:

Fie *proces_data* o procedură care procesează în mod normal date zilei curente, dar care optional poate procesa și alte date. Dacă nu se specifică parametrul actual corespunzător parametrului formal *plan_data*, atunci acesta va lua automat valoarea dată implicit.

```
PROCEDURE proces_data(data_in IN NUMBER, plan_data
IN DATE:=SYSDATE) IS...
```

Următoarele comenzi reprezintă apele corecte ale procedurii *proces_data*:

```
proces_data(10);  proces_data(10,SYSDATE+1);
proces_data(plan_data=>SYSDATE+1,data_in=>10);
```

O declarație de subprogram (procedură sau funcție) fără parametri este specificată fără paranteze. De exemplu, dacă procedura *react_calc_dur* și funcția *obt_date* nu au parametri, atunci:

```
react_calc_dur;  apel corect
react_calc_dur(); apel incorrect
data_meia := obt_date; apel corect
```

Module *overload*

În anumite condiții, două sau mai multe module pot să aibă aceleași nume, dar să difere prin lista parametrilor. Aceste module sunt numite module *overload* (supraîncărcate). Funcția *TO_CHAR* este un exemplu de modul *overload*.

În cazul unui apel, compilatorul compară parametri actuali cu listele parametrilor formali pentru modulele *overload* și execută modulul corespunzător. **Toate programele *overload* trebuie să fie definite în același bloc *PL/SQL*** (bloc anonim, modul sau pachet). Nu poate fi definită o versiune într-un bloc, iar altă versiune într-un bloc diferit.

Modulele *overload* pot să apară în programele *PL/SQL* fie în secțiunea declarativă a unui bloc, fie în interiorul unui pachet. Supraîncărcarea funcțiilor sau procedurilor nu se poate face pentru funcții sau proceduri stocate, dar se poate face pentru subprograme locale, subprograme care apar în pachete sau pentru metode.

Observații:

- Două programe *overload* trebuie să difere, cel puțin, prin tipul unuia dintre parametri. Două programe nu pot fi *overload* dacă parametri lor formali diferă numai prin subtipurile lor și dacă aceste subtipuri se bazează pe același tip de date.
- Nu este suficient ca lista parametrilor programelor *overload* să difere numai prin numele parametrilor formali.
- Nu este suficient ca lista parametrilor programelor *overload* să difere numai prin tipul acestora (*IN*, *OUT*, *IN OUT*). *PL/SQL* nu poate face diferențe (la apelare) între tipurile *IN* sau *OUT*.
- Nu este suficient ca funcțiile *overload* să difere doar prin tipul datei returnate (tipul datei specificate în clauza *RETURN* a funcției).

Exemplu:

Următoarele subprograme nu pot fi *overload*.

- a) **FUNCTION alfa(par IN POSITIVE) ...;**
FUNCTION alfa(par IN BINARY_INTEGER) ...;
- b) **FUNCTION alfa(par IN NUMBER) ...;**
FUNCTION alfa(parar IN NUMBER) ...;
- c) **PROCEDURE beta(par IN VARCHAR2) IS...;**
PROCEDURE beta(par OUT VARCHAR2) IS...;

Exemplu:

Să se creeze două funcții (locale) cu același nume care să calculeze media valorilor operelor de artă de un anumit tip. Prima funcție va avea un argument reprezentând tipul operelor de artă, iar cea de a doua va avea două argumente, unul reprezentând tipul operelor de artă, iar celălalt reprezentând stilul operelor pentru care se calculează valoarea medie (adică funcția va calcula media valorilor operelor de artă de un anumit tip și care aparțin unui stil specificat).

```

DECLARE
    medie1 NUMBER(10,2);
    medie2 NUMBER(10,2);
FUNCTION valoare_medie (v_tip opera.tip%TYPE)
    RETURN NUMBER IS
    medie NUMBER(10,2);
BEGIN
    SELECT AVG(valoare)
        INTO medie
        FROM opera
        WHERE tip = v_tip;
    RETURN medie;
END;
FUNCTION valoare.medie (v_tip      opera.tip%TYPE,
                        v_stil     opera.stil%TYPE)
    RETURN NUMBER IS medie
    NUMBER(10,2);
BEGIN
    SELECT AVG(valoare)
        INTO medie
        FROM opera
        WHERE tip = v_tip AND stil = v_stil;
    RETURN medie;
END;
BEGIN
    medie1 := valoare_medie('pictura');
    DBMS_OUTPUT.PUT_LINE('Media valorilor picturilor din
                           muzeu este ' || medie1);
    medie2 := valoare_medie('pictura', 'impressionism');
    DBMS_OUTPUT.PUT_LINE('Media valorilor picturilor
                           impresioniste din muzeu este ' || medie2);
END;

```

Procedură *versus* funcție

Pot fi marcate câteva **deosebiri** esențiale între funcții și proceduri.

- Procedura se execută ca o comandă *PL/SQL*, iar funcția se invocă ca parte a unei expresii.
- Procedura poate returna (sau nu) una sau mai multe valori, iar funcția trebuie să returneze (cel puțin) o singură valoare.
- Procedura nu trebuie să conțină *RETURN tip_date*, iar funcția trebuie să conțină această opțiune.

De asemenea, pot fi marcate câteva elemente esențiale, comune atât funcțiilor cât și procedurilor. Ambele pot:

- accepta valori implicite;
- avea secțiuni declarative, executabile și de tratare a erorilor;
- utiliza specificarea prin nume sau poziție a parametrilor;

Recursivitate

Un subprogram recursiv presupune că acesta se apelează pe el însuși.

În *Oracle* o problemă delicată este legată de locul unde se plasează un apel recursiv. De exemplu, dacă apelul este în interiorul unui cursor *FOR* sau între comenzi *OPEN* și *CLOSE*, atunci la fiecare apel este deschis alt cursor. În felul acesta, programul poate depăși limita pentru *OPEN_CURSORS* setată în parametrul de inițializare *Oracle*.

Exemplu:

Să se calculeze recursiv al *m*-lea termen din sirul lui Fibonacci.

```
FUNCTION fibona(m POSITIVE) RETURN INTEGER IS
BEGIN
    IF (m = 1) OR (m = 2) THEN
        RETURN 1;
    ELSE
        RETURN fibona(m-1) + fibona(m-2);
    END IF;
END fibona;
```

Declarații *forward*

Subprogramele sunt reciproc recursive dacă ele se apelează unul pe altul direct sau indirect. Declarațiile *forward* permit definirea subprogramelor reciproc recursive.

În *PL/SQL*, un identificator trebuie declarat înainte de a-l folosi. De asemenea, un subprogram trebuie declarat înainte de a-l apela.

```
PROCEDURE alfa ( ... ) IS
BEGIN
    beta( ... );           -- apel incorect
    ...
END;
PROCEDURE beta ( ... ) IS
BEGIN
    ...
END;
```

Procedura *beta* nu poate fi apelată deoarece nu este încă declarată. Problema se poate rezolva simplu, inversând ordinea celor două proceduri. Această soluție nu este eficientă întotdeauna. *PL/SQL* permite un tip special de declarare a unui subprogram numit *forward*. El constă dintr-o specificare de subprogram terminată prin “;”.

```
PROCEDURE beta ( ... );      -- declaratie forward
...
PROCEDURE alfa ( ... ) IS
BEGIN
    beta( ... );
    ...
END;
PROCEDURE beta ( ... ) IS
BEGIN
    ...
END;
```

Se pot folosi declarații *forward* pentru a defini subprograme într-o anumită ordine logică, pentru a defini subprograme reciproc recursive, pentru a grupa subprograme într-un pachet.

Lista parametrilor formali din declarația *forward* trebuie să fie identică cu

cea corespunzătoare corpului subprogramului. Corpul subprogramului poate apărea oriunde după declarația sa *forward*, dar trebuie să rămână în aceeași unitate de program.

Utilizarea în expresii *SQL* a funcțiilor definite de utilizator

Începând cu *Release 7.1*, o funcție stocată poate fi referită într-o comandă *SQL* la fel ca orice funcție standard furnizată de sistem (*built-in function*), dar cu anumite restricții. Funcțiile *PL/SQL* definite de utilizator pot fi apelate din orice expresie *SQL* în care pot fi folosite funcții *SQL* standard.

Funcțiile *PL/SQL* pot să apară în:

- lista de selecție a comenzi *SELECT*;
- condiția clauzelor *WHERE* și *HAVING*;
- clauzele *CONNECT BY*, *START WITH*, *ORDER BY* și *GROUP BY*;
- clauza *VALUES* a comenzi *INSERT*;
- clauza *SET* a comenzi *UPDATE*.

Exemplu:

Să se afișeze operele de artă (titlu, valoare, stare) a căror valoare este mai mare decât valoarea medie a tuturor operelor de artă din muzeu.

```
CREATE OR REPLACE FUNCTION valoare_mediul RETURN NUMBER IS
v_val_mediul opera.valoare%TYPE;
BEGIN
SELECT AVG(valoare) INTO v_val_mediul FROM opera; RETURN
v_val_mediul;
END;
```

Referirea acestei funcții într-o comandă *SQL* se poate face prin:

```
SELECT titlu, valoare, stare
FROM opera
WHERE valoare >= valoare_mediul;
```

Există restricții referitoare la folosirea funcțiilor definite de utilizator într-o comandă *SQL*.

- funcția definită de utilizator trebuie să fie o funcție stocată (procedurile stocate nu pot fi apelate în expresii *SQL*), nu poate fi locală unui alt bloc;
- funcția apelată dintr-o comandă *SELECT*, sau din comenzi paralelizate *INSERT*, *UPDATE* și *DELETE* nu poate contine comenzi *LMD* care modifica tabelele bazei de date;
- funcția apelată dintr-o comandă *UPDATE* sau *DELETE* nu poate interoga sau modifica tabele ale bazei reactualizate chiar de aceste comenzi (*table mutating*);
- funcția apelată din comenziile *SELECT*, *INSERT*, *UPDATE* sau *DELETE* nu poate executa comenzi *LCD* (*COMMIT*), *ALTER SYSTEM*, *SET ROLE* sau comenzi *LDD* (*CREATE*);
- funcția nu poate să apară în clauza *CHECK* a unei comenzi *CREATE/ALTER TABLE*;

- funcția nu poate fi folosită pentru a specifica o valoare implicită pentru o coloană în cadrul unei comenzi *CREATE/ALTER TABLE*;
- funcția poate fi utilizată într-o comandă *SQL* numai de către proprietarul funcției sau de utilizatorul care are privilegiul *EXECUTE* asupra acesteia;
- funcția definită de utilizator, apelabilă dintr-o comandă *SQL*, trebuie să aibă doar parametri de tip *IN*, cei de tip *OUT* și *IN OUT* nefiind acceptați;
- parametrii unei funcții *PL/SQL* apelate dintr-o comandă *SQL* trebuie să fie specificați prin poziție (specificarea prin nume nefiind permisă);
- parametrii formali ai unui subprogram funcție trebuie să fie de tip specific bazei de date (*NUMBER*, *CHAR*, *VARCHAR2*, *ROWID*, *LONG*, *LONGROW*, *DATE*), nu tipuri *PL/SQL* (*BOOLEAN* sau *RECORD*);
- tipul returnat de un subprogram funcție trebuie să fie un tip intern pentru *server*, nu un tip *PL/SQL* (nu poate fi *TABLE*, *RECORD* sau *BOOLEAN*);
- funcția nu poate apela un subprogram care nu respectă restricțiile anterioare.

Exemplu:

```
CREATE OR REPLACE FUNCTION calcul (p_val NUMBER)
  RETURN NUMBER IS
BEGIN
  INSERT INTO opera(cod_opera, tip, data_achizitie,
valoare) ;
  VALUES (1358, 'gravura', SYSDATE, 700000);
  RETURN (p_val*7); END;
/
UPDATE opera
SET valoare = calcul (550000)
WHERE cod_opera = 7531;
```

Comanda *UPDATE* va returna o eroare deoarece tabelul *opera* este *mutating*. Reactualizarea este însă permisă asupra oricărui alt tabel diferit de *opera*.

Informații referitoare la subprograme

Informațiile referitoare la subprogramele *PL/SQL* și modul de acces la aceste informații sunt următoarele:

- codul sursă, utilizând vizualizarea *USER_SOURCE* din dicționarul datelor (*DD*);

- informații generale, utilizând vizualizarea *USER_OBJECTS* din dicționarul datelor;
- tipul parametrilor (*IN*, *OUT*, *IN OUT*), utilizând comanda *DESCRIBE* din *SQL*;
- *p-code* (nu este accesibil utilizatorilor);
- erorile la compilare, utilizând vizualizarea *USER_ERRORS* din dicționarul datelor sau comanda *SHOW ERRORS*;
- informații de depanare, utilizând pachetul *DBMS_OUTPUT*.

Vizualizarea *USER_OBJECTS* conține informații generale despre toate obiectele manipulate în BD, în particular și despre subprogramele stocate.

Vizualizarea *USER_OBJECTS* are următoarele câmpuri:

- *OBJECT_NAME* – numele obiectului;
- *OBJECT_TYPE* – tipul obiectului (*PROCEDURE*, *FUNCTION* etc.);
- *OBJECT_ID* – identificator intern al obiectului;
- *CREATED* – data când obiectul a fost creat;
- *LAST_DDL_TIME* – data ultimei modificări a obiectului;
- *TIMESTAMP* – data și momentul ultimei recompilări;
- *STATUS* – starea obiectului (*VALID* sau *INVALID*).

Pentru a verifica dacă recompilarea explicită (*ALTER*) sau implicită a avut succes se poate verifica starea subprogramelor utilizând *USER_OBJECTS*.

Orice obiect are o stare (*status*) sesizată în DD, care poate fi:

- *VALID* (obiectul a fost compilat și poate fi folosit când este referit);
- *INVALID* (obiectul trebuie compilat înainte de a fi folosit).

Exemplu:

Să se listeze procedurile și funcțiile deținute de utilizatorul curent, precum și starea acestora.

```
SELECT      OBJECT_NAME, OBJECT_TYPE, STATUS
FROM        USER_OBJECTS
WHERE       OBJECT_TYPE IN ('PROCEDURE', 'FUNCTION');
```

După ce subprogramul a fost creat, codul sursă al acestuia poate fi obținut consultând vizualizarea *USER_SOURCE* din DD, care are următoarele câmpuri:

- *NAME* – numele obiectului;
- *TYPE* – tipul obiectului;
- *LINE* – numărul liniei din codul sursă;
- *TEXT* – textul liniilor codului sursă.

Exemplu:

Să se afișeze codul complet pentru funcția *numar_opere*.

```
SELECT      TEXT  USER_SOURCE
FROM        NAME = 'numar_opere'
WHERE       LINE ;
ORDER BY
```

Exemplu:

Să se scrie o procedură care recompilează toate obiectele invalide din schema personală.

```
CREATE OR REPLACE PROCEDURE sterge IS
  CURSOR obj_curs IS
    SELECT OBJECT_TYPE, OBJECT_NAME
    FROM   USER_OBJECTS
   WHERE  STATUS = 'INVALID'
   AND    OBJECT_TYPE IN
          ('PROCEDURE', 'FUNCTION', 'PACKAGE',
           'PACKAGE BODY', 'VIEW');
BEGIN
  FOR obj_rec IN obj_curs LOOP
    DBMS_DDL.ALTER_COMPILE(obj_rec.OBJECT_TYPE,
                           'USER', obj_rec.OBJECT_NAME);
  END LOOP;
END sterge;
```

Dacă se recompilează un obiect *PL/SQL*, atunci *server-ul* va recompila orice obiect invalid de care depinde. Dacă recompilarea automată implicită a procedurilor locale dependente are probleme, atunci starea obiectului va rămâne *INVALID* și *server-ul Oracle* semnalează eroare. Prin urmare:

- este preferabil ca recompilarea să fie manuală (recompilare explicită utilizând comanda *ALTER (PROCEDURE, FUNCTION, TRIGGER, PACKAGE)* cu opțiunea *COMPILE*);
- este necesar ca recompilarea să se facă cât mai repede, după definirea unei schimbări referitoare la obiectele bazei.

Pentru a obține valori (de exemplu, valoarea contorului pentru un *LOOP*, valoarea unei variabile înainte și după o atribuire etc.) și mesaje (de exemplu, părăsirea unui subprogram, apariția unei operații etc.) dintr-un bloc *PL/SQL* pot fi utilizate procedurile pachetului *DBMS_OUTPUT*. Aceste informații se cumulează într-un *buffer* care poate fi consultat.

Dependența subprogramelor

Când este compilat un subprogram, toate obiectele *Oracle* care sunt referite vor fi înregistrate în dicționarul datelor. Subprogramul este dependent de aceste obiecte. Un subprogram care are erori la compilare este marcat ca "invalid" în dicționarul datelor. Un subprogram stocat poate deveni, de asemenea, invalid dacă o operație *LDD* este executată asupra unui obiect de care depinde.

Obiecte dependente:

*View, Table Procedure
Function
Package Specification
Package Body Database
Trigger
Obiect definit de utilizator
Tip colectie*

Obiecte referite

*Table,
Seventa View
Procedure Function
Synonym
Package Specification
Obiect definit de utilizator
Tip colectie*

Dacă se modifică definiția unui obiect referit, obiectul dependent poate (sau nu) să continue să funcționeze normal.

Există două tipuri de dependențe:

- dependență directă, în care obiectul dependent (de exemplu, *procedure* sau *function*) face referință direct la un *table*, *view*, *sequence*, *procedure*, *function*.
- dependență indirectă, în care obiectul dependent (*procedure* sau *function*) face referință indirect la un *table*, *view*, *sequence*, *procedure*, *function* prin intermediul unui *view*, *procedure* sau *function*.

În cazul dependențelor locale, când un obiect referit este modificat, obiectele dependente sunt invalidate. La următorul apel al obiectului invalidat, acesta va fi recompilat automat de către *server-ul Oracle*.

În cazul dependențelor la distanță, procedurile stocate local și toate obiectele dependente vor fi invalidate. Ele nu vor fi recompilate automat la următorul apel.

Exemplu:

Se presupune că procedura *filtru* va referi direct tabelul *opera* și că procedura *adaug* va reactualiza tabelul *opera* prin intermediul unei vizualizări *nou_opera*.

Pentru aflarea dependențelor directe se poate utiliza vizualizarea *USER_DEPENDENCIES* din dicționarul datelor.

```
SELECT NAME, TYPE, REFERENCED_NAME, REFERENCED_TYPE
FROM USER_DEPENDENCIES
WHERE REFERENCED_NAME IN ('opera', 'nou_opera');
```

NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE
filtru	Procedure	opera	Table
adaug	Procedure	nou_operă	View
nou_operă	View	opera	Table

Dependențele indirecte pot fi afișate utilizând vizualizările *DEPTREE* și *IDEPTREE*. Vizualizarea *DEPTREE* afișează o reprezentare a tuturor obiectelor dependente (direct sau indirect). Vizualizarea *IDEPTREE* afișează o reprezentare a aceleiași informații, sub forma unui arbore.

Pentru a utiliza aceste vizualizări furnizate de sistemul *Oracle* trebuie:

1. executat scriptul *UTLDTREE*;
2. executată procedura *DEPTREE_FILL* (are trei argumente: tipul obiectului referit, schema obiectului referit, numele obiectului referit).

Exemplu:

```
@UTLDTREE
EXECUTE DEPTREE_FILL ('TABLE', 'SCOTT', 'opera')
SELECT NESTED_LEVEL, TYPE, NAME
FROM DEPTREE
ORDER BY SEQ#;
    NESTED_LEVEL   TYPE          NAME
              0   Table        opera
              1   View         nou_operă
              2   Procedure    adaug
              1   Procedure    filtru

SELECT *
FROM IDEPTREE;

DEPENDENCIES
TABLE  nume_schema.opera
VIEW   nume_schema.nou_operă
PROCEDURE nume_schema.adaug
PROCEDURE nume_schema.filtru
```

Dependențele la distanță sunt manipulate prin una din modalitățile alese de utilizator: modelul *timestamp* (implicit) sau modelul *signature*.

Fiecare unitate *PL/SQL* are un *timestamp* care este setat când unitatea este modificată (creata sau recompilată) și care este depus în câmpul *LAST_DDL_TIME* din dicționarul datelor. Modelul *timestamp* realizează compararea momentelor ultimei modificări a celor două obiecte analizate. Dacă obiectul (referit) bazei are momentul ultimei modificări mai recent ca cel al obiectului dependent, atunci obiectul dependent va fi recompilat.

Modelul *signature* determină momentul la care obiectele bazei distante trebuie recompilate. Când este creată o procedură, o *signature* este depusă în dicționarul datelor, alături de *p-code*. Aceasta conține: numele construcției *PLSQL (PROCEDURE, FUNCTION, PACKAGE)*, tipurile parametrilor, ordinea parametrilor, numărul acestora și modul de transmitere (*IN, OUT, IN OUT*). Dacă parametrii se schimbă, atunci evident *signature* se schimbă. Dacă signatura nu se schimba, atunci execuția continuă.

Recompilarea procedurilor și funcțiilor dependente este fără succes dacă:

- obiectul referit este distrus (*DROP*) sau redenumit (*RENAME*);
- tipul coloanei referite este schimbat;
- coloana referita este stearsă;
- o vizualizare referită este înlocuită printr-o vizualizare având alte coloane;
- lista parametrilor unei proceduri referite este modificată.

Recompilarea procedurilor și funcțiilor dependente este cu succes dacă:

- tabelul referit are noi coloane;
- nici o coloană nou definită nu are restrictia NOT NULL;
- tipul coloanelor referite nu s-a schimbat;
- un tabel "private" este sters, dar există un tabel "public" având același nume și structură;
- toate comenziile *INSERT* contin efectiv lista coloanelor;
- corpul *PL/SQL* a unei proceduri referite a fost modificat și recompilat cu succes.

Cum pot fi minimizate erorile datorate dependențelor?

- utilizând comenzi *SELECT* cu opțiunea *;
- incluzând lista coloanelor în cadrul comenzi *INSERT*;
- declarând variabile cu atributul *%TYPE*;
- declarând înregistrări cu atributul *%ROWTYPE*.

În concluzie:

- Dacă procedura depinde de un obiect local, atunci se face recompilare automată la prima reexecuție.
- Dacă procedura depinde de o procedură distanță, atunci se face recompilare automată, dar la a doua reexecuție. Este preferabilă o recompilare manuală pentru prima reexecuție sau implementarea unei strategii de reinvoacare a ei (a două oară).
- Dacă procedura depinde de un obiect distant, dar care nu este procedură, atunci nu se face recompilare automată.

Rutine externe (optional)

PL/SQL a fost special conceput pentru *Oracle* și este specializat pentru procesarea tranzacțiilor *SQL*.

Totuși, într-o aplicație complexă pot să apară cerințe și funcționalități care sunt mai eficient de implementat în *C*, *Java* sau alt limbaj de programare. Dacă aplicația trebuie să efectueze anumite acțiuni care nu pot fi implementate optim

utilizând *PL/SQL*, atunci este preferabil să fie utilizate alte limbaje care realizează performant acțiunile respective. În acest caz este necesară comunicarea între diferite module ale aplicației care sunt scrise în limbaje diferite.

Până la versiunea *Oracle8*, singura modalitate de comunicare între *PL/SQL* și alte limbaje (de exemplu, limbajul *C*) a fost utilizarea pachetelor *DBMS_PIPE* și/sau *DBMS_ALERT*.

Începând cu *Oracle8*, comunicarea este simplificată prin utilizarea rutinelor externe. O rutină externă este o procedură sau o funcție scrisă într-un limbaj diferit de *PL/SQL*, dar apelabilă dintr-un program *PL/SQL*. *PL/SQL* extinde funcționalitatea *server-ului Oracle*, furnizând o interfață pentru apelarea rutinelor externe. Orice bloc *PL/SQL* executat pe *server* sau pe *client* poate apela o rutină externă. Singurul limbaj acceptat pentru rutine externe în *Oracle8* era limbajul *C*.

Pentru a marca apelarea unei rutine externe în programul *PL/SQL* este definit un punct de intrare (*wrapper*) care direcționează spre codul extern (program *PL/SQL* → *wrapper* → cod extern). O clauză specială (*AS EXTERNAL*) este utilizată (în cadrul comenzi *CREATE OR REPLACE PROCEDURE*) pentru crearea unui *wrapper*. De fapt, clauza conține informații referitoare la numele bibliotecii în care se găsește subprogramul extern (clauza *LIBRARY*), numele rutinei externe (clauza *NAME*) și corespondența (*C* <→ *PL/SQL*) între tipurile de date (clauza *PARAMETERS*). Ultimele versiuni renunță la clauza *AS EXTERNAL*.

Rutinele externe (scrise în *C*) sunt compilate, apoi depuse într-o bibliotecă dinamică (*DLL – dynamic link library*) și sunt încărcate doar când este necesar acest lucru. Dacă se invocă o rutină externă scrisă în *C*, trebuie setată conexiunea spre această rutină. Un proces numit *extproc* este declanșat automat de către *server*. La rândul său, procesul *extproc* va încărca biblioteca identificată prin clauza *LIBRARY* și va apela rutina respectivă.

Oracle9i permite utilizarea de rutine externe scrise în *Java*. De asemenea, prin utilizarea clauzei *AS LANGUAGE*, un *wrapper* poate include specificații de apelare. De fapt, aceste specificații permit apelarea rutinelor externe scrise în orice limbaj. De exemplu, o procedură scrisă într-un limbaj diferit de *C* sau *Java* poate fi utilizată în *SQL* sau *PL/SQL* dacă procedura respectivă este apelabilă din

C. În felul acesta, biblioteci standard scrise în alte limbaje de programare pot fi apelate din programe *PL/SQL*.

Procedura *PL/SQL* executată pe *server-ul Oracle* poate apela o rutină externă scrisă în *C* care este depusă într-o bibliotecă partajată.

Procedura *C* se execută într-un spațiu adresă diferit de cel al *server-ului Oracle*, în timp ce unitățile *PL/SQL* și metodele *Java* se execută în spațiul de adresă al *server-ului JVM (Java Virtual Machine)* de pe pe *server* va executa metoda *Java* direct, fără a fi necesar procesul *extproc*.

Maniera de a încărca depinde de limbajul în care este scrisă rutina (*C* sau *Java*).

- Pentru a apela rutine externe *C*, *server-ul* trebuie să cunoască poziționarea bibliotecii dinamice *DLL*. Acest lucru este furnizat de *alias-ul* bibliotecii din clauza *AS LANGUAGE*.
- Pentru apelarea unei rutine externe *Java* se va încărca clasa *Java* în baza de date. Este necesară doar crearea unui *wrapper* care direcționează spre codul extern. Spre deosebire de rutinele externe *C*, nu este necesară nici biblioteca și nici setarea conexiunii spre rutina externă.

Clauza *LANGUAGE* din cadrul comenzi de creare a unui subprogram, specifică limbajul în care este scrisă rutina (procedură externă *C* sau metodă *Java*) și are următoarea formă:

{IS / AS} LANGUAGE {C / JAVA}

Pentru o procedură *C* sunt date informații referitoare la numele acesteia (clauza *NAME*); *alias-ul* bibliotecii în care se găsește (clauza *LIBRARY*); opțiuni referitoare la tipul, poziția, lungimea, modul de transmitere (prin valoare sau prin referință) al parametrilor (clauza *PARAMETERS*); posibilitatea ca rutina externă să acceseze informații despre parametri, excepții, alocarea memoriei utilizator (clauza *WITH CONTEXT*).

**LIBRARY nume_biblioteca [NAME nume_proc_c] [WITH CONTEXT]
[PARAMETERS (parametru_extern [, parametru_extern ...])]**

Pentru o metodă *Java*, în cluză trebuie specificată doar signatura metodei (lista tipurilor parametrilor în ordinea apariției).

Exemplu:

```
CREATE OR REPLACE FUNCTION calc (x IN REAL) RETURN NUMBER AS
LANGUAGE C
LIBRARY biblioteca
NAME "c_calc"
PARAMETERS (x BY REFERENCES);
```

Scrierea "c_calc" este corecta, iar " " implica ca stocarea este *case sensitive*, altfel implicit se depune numele cu litere mari.

Procedura poate fi apelata dintr-un bloc *PL/SQL*:

```
DECLARE
    emp_id    NUMBER;
    procent   NUMBER;
BEGIN
...
calc(emp_id, procent);
... END;
```

Rutina externă nu este apelată direct, ci se apelează subprogramul *PL/SQL* care referă rutina externă.

Apelarea poate să apară în: blocuri anonte, subprograme independente sau aparținând unui pachet, metode ale unui tip obiect, declanșatori bază de date, comenzi *SQL* care apelează funcții (în acest caz trebuie utilizată pragma *RESTRICT_REFERENCES*).

De remarcat că o metodă *Java* poate fi apelată din orice bloc *PL/SQL*, subprogram sau pachet.

JDBC (Java Database Connectivity), care reprezintă interfața *Java* standard pentru conectare la baze de date relaționale și *SQLJ* permit apelarea de blocuri *PL/SQL* din programe *Java*. *SQLJ* face posibilă incorporarea operațiilor

SQL în codul *Java*. Standardul *SQLJ* acoperă doar operații *SQL* statice. *Oracle9i SQLJ* include extensii pentru a suporta direct *SQL* dinamic.

O altă modalitate de a încărca programe *Java* este folosirea interactivă în *iSQL*Plus* a comenzi: *CREATE JAVA instrucțiune*.

Funcții tabel

O funcție tabel (*table function*) returnează drept rezultat un set de linii (de obicei, sub forma unei colecții). Această funcție poate fi interogată direct printr-o comandă *SQL*, ca și cum ar fi un tabel al bazei de date. În felul acesta, funcția poate fi utilizată în clauza *FROM* a unei cereri.

O funcție tabel conductă (*pipelined table function*) este similară unei funcții tabel, dar returnează datele iterativ, pe măsură ce acestea sunt obținute, nu toate deodată. Aceste funcții sunt mai eficiente deoarece informația este returnată imediat cum este obținută.

Conceptul de funcție tabel conductă a fost introdus în versiunea *Oracle9i*. Utilizatorul poate să definească astfel de funcții. De asemenea, este posibilă execuția paralelă a funcțiilor tabel (evident și a celor clasice). În acest caz, funcția trebuie să conțină în declarație opțiunea *PARALLEL_ENABLE*.

Funcția tabel conductă acceptă orice argument pe care îl poate accepta o funcție obișnuită și trebuie să returneze o colecție (*nested table* sau *varray*). Un parametru input poate fi vector, tabel *PL/SQL*, *REF CURSOR*. Ea este declarată specificând cuvântul cheie *PIPELINED* în comanda *CREATE OR REPLACE FUNCTION*. Funcția tabel conductă trebuie să se termine printr-o comandă *RETURN* simplă, care nu întoarce nici o valoare.

Pentru a returna un element individual al colecției este folosită comanda *PIPE ROW*, care poate să apară numai în corpul unei funcții tabel conductă, în caz contrar generându-se o eroare. Comanda poate fi omisă dacă funcția tabel conductă nu returnează nici o linie.

După ce funcția a fost creată, ea poate fi apelată dintr-o cerere *SQL*

utilizând operatorul *TABLE*. Cererile referitoare la astfel de funcții pot să includă cursoare și referințe la cursoare, respectându-se semantica de la cursoarele clasice.

Funcția tabel conductă nu poate să apară în comenziile *INSERT*, *UPDATE*, *DELETE*. Totuși, pentru a realiza o reactualizare, poate fi creată o vizualizare relativă la funcția tabel și folosind un declanșator *INSTEAD OF*.

Exemplu:

```
CREATE FUNCTION ff(p SYS_REFCURSOR) RETURN cartype
PIPELINED IS
PRAGMA AUTONOMOUS_TRANSACTION; BEGIN ... END;
```

In timpul executiei paralele, fiecare instantă a funcției tabel va crea o tranzacție independentă.

Urmatoarele comenzi sunt incorecte.

```
UPDATE ff(CURSOR (SELECT * FROM tab))
SET col = valoare; INSERT INTO ff(...)
VALUES ('orice', 'vrei');
```

Exemplu:

Să se obțină o instanță a unui tabel ce conține informații referitoare la denumirea zilelor săptămânii.

Problema este rezolvată în două variante. Prima reprezintă o soluție clasică, iar a doua variantă implementează problema cu ajutorul unei funcții tabel conductă.

Varianta 1:

```
CREATE TYPE t_linie AS OBJECT (
    idl NUMBER, sir VARCHAR2(20));
CREATE TYPE t_tabel AS TABLE OF t_linie;
```

```

CREATE OR REPLACE FUNCTION calc1 RETURN t_tabel AS
  v_tabelt_tabel;  BEGIN
    v_tabel := t_tabel (t_linie (1, 'luni'));
    FOR j IN 2..7 LOOP
      v_tabel.EXTEND;  IF j = 2
        THEN v_tabel(j) := t_linie (2, 'marti');  ELSIF j
        = 3
          THEN v_tabel(j) := t_linie (3, 'miercuri');
        ELSIF j = 4
          THEN v_tabel(j) := t_linie (4, 'joi');
        ELSIF j = 5
          THEN v_tabel(j) := t_linie (5, 'vineri');
        ELSIF j = 6
          THEN v_tabel(j) := t_linie (6, 'sambata');
        ELSIF j = 7
          THEN v_tabel(j) := t_linie (7, 'duminica');
      END IF;
    END LOOP;
  RETURN v_tabel;  END calc1;

```

Funcția *calc1* poate fi invocată în clauza *FROM* a unei comenzi *SELECT*:

```

SELECT *
  FROM TABLE (CAST (calc1 AS t_tabel));

```

Varianta 2:

```

CREATE OR REPLACE FUNCTION calc2 RETURN t_tabel
PIPELINED AS
  v_liniel linie;  BEGIN
    FOR j IN 1..7 LOOP
      v_linie := CASE j
        WHEN 1 THEN t_linie (1, 'luni')
        WHEN 2 THEN t_linie (2, 'marti')
        WHEN 3 THEN t_linie (3, 'miercuri')
        WHEN 4 THEN t_linie (4, 'joi')
        WHEN 5 THEN t_linie (5, 'vineri')
        WHEN 6 THEN t_linie (6, 'sambata')
        WHEN 7 THEN t_linie (7, 'duminica')  END;
    PIPE ROW (v_linie);  END LOOP;
  RETURN;
END calc2;

```

Se observă că tabelul este implicat doar în tipul rezultatului. Pentru apelarea funcției *calc2* este folosită sintaxa următoare:

```
SELECT * FROM TABLE (calc2);
```

Funcțiile tabel sunt folosite frecvent pentru conversii de tipuri de date. *Oracle9i* introduce posibilitatea de a crea o funcție tabel care returnează un tip *PL/SQL* (definit într-un bloc). Funcția tabel care furnizează (la nivel de pachet) drept rezultat un tip de date trebuie să fie de tip conductă. Pentru apelare este utilizată sintaxa simplificată (fără *CAST*).

Procesarea tranzacțiilor autonome

Tranzacția este o unitate logică de lucru, adică o secvență de comenzi care trebuie să se execute ca un întreg pentru a menține consistența bazei. În mod uzual, o tranzacție poate să cuprindă mai multe blocuri, iar într-un bloc pot să fie mai multe tranzacții.

O tranzacție autonomă este o tranzacție independentă începută de altă tranzacție, numită tranzacție principală. Tranzacția autonomă permite suspendarea tranzacției principale, executarea de comenzi *SQL*, *commit*-ul și *rollback*-ul acestor operații.

Odată începută, tranzacția autonomă este independentă în sensul că nu împarte blocări, resurse sau dependențe cu tranzacția principală.

În felul acesta, o aplicație nu trebuie să cunoască operațiile autonome ale unei proceduri, iar procedura nu trebuie să cunoască nimic despre tranzacțiile aplicației.

Pentru definirea unei tranzacții autonome trebuie să se utilizeze pragma

AUTONOMOUS_TRANSACTION care informează compilatorul *PL/SQL* să marcheze o rutină ca fiind autonomă. Prin rutină se înțelege: bloc anonim de cel mai înalt nivel (nu încuibărit); procedură sau funcție locală, independentă sau împachetată; metodă a unui tip obiect; declanșator bază de date.

```
CREATE PACKAGE exemplu AS
  ...
  FUNCTION autono(x INTEGER) RETURN real;
END exemplu;
CREATE PACKAGE BODY exemplu AS
  ...
  FUNCTION autono(x INTEGER) RETURN real IS PRAGMA
    AUTONOMOUS_TRANSACTION;
    z real;
BEGIN
  ... END;
END exemplu;
```

Codul *PRAGMA AUTONOMOUS_TRANSACTION* poate marca numai rutine individuale ca fiind independente. Nu pot fi marcate toate subprogramele unui pachet sau toate metodele unui tip obiect ca autonome. Prin urmare, *pragma* nu poate să apară în partea de specificație a unui pachet. Codul *PRAGMA AUTONOMOUS_TRANSACTION* se specifică în partea declarativă a rutinei.

Observații:

- Declanșatorii autonomi, spre deosebire de cei clasici pot conține comenzi *LCD* (de exemplu, *COMMIT*, *ROLLBACK*).
- Excepțiile declanșate în tranzacții autonome generează un *rollback* la nivel de tranzacție, nu la nivel de instrucțiune.
- Când se intră în secțiunea executabilă a unei tranzacții autonome, tranzacția principală se suspendă.

Cu toate că o tranzacție autonomă este începută de altă tranzacție, ea **nu** este o tranzacție încuibărită deoarece:

- nu partajează resurse cu tranzacția principală;
- nu depinde de tranzacția principală (de exemplu, dacă tranzacția principală este *rollback*, atunci tranzacțiile încuibărite sunt de asemenea *rollback*, dar tranzacția autonomă nu este *rollback*);
- schimbările *commit* din tranzacții autonome sunt vizibile imediat altor tranzacții, pe când cele de la tranzacții încuibărite sunt vizibile doar după ce tranzacția principală este *commit*.

Pachete

Pachete in *PL/SQL*

Pachetul (*package*) permite incapsularea intr-o unitate logica in baza de date a procedurilor, functiilor, cursoarelor, tipurilor, constantelor, variabilelor si exceptiilor.

Pachetele sunt unitati de program care sunt compilate, depanate si testate, sunt obiecte ale bazei de date care grupeaza tipuri, obiecte si subprograme *PL/SQL* avand o legatura logica intre ele.

De ce sunt importante?

Atunci cand este referentiat un pachet (cand este apelata pentru prima data o constructie a pachetului), intregul pachet este incarcat in *SGA*, zona globala a sistemului, si este pregatit pentru executie. Plasarea pachetului in *SGA* (zona globala sistem) reprezinta avantajul vitezei de executie, deoarece *server-ul* nu mai trebuie sa aduca informatia despre pachet de pe disc, aceasta fiind deja in memorie. Prin urmare,apeluri ulterioare ale unor constructii din acelasi pachet, nu solicita operatii *I/O* de pe disc. De aceea, ori de cate ori apare cazul unor proceduri si functii inrudite care trebuie sa fie executate impreuna, este convenabil ca acestea sa fie grupate intr-un pachet stocat. Este de subliniat ca in memorie exista o singura copie a unui pachet, pentru toti utilizatorii.

Spre deosebire de subprograme, pachetele nu pot:

- fi apelate,
- transmite parametri,
- fi incuibarite.

Un pachet are doua parti, fiecare fiind stocata separat in dictionarul datelor.

- Specificarea pachetului (*package specification*) - partea „vizibila”, adica interfata cu aplicatii sau cu alte unitati program. Se declara tipuri, constante, variabile, exceptii, cursoare si subprograme folositoare utilizatorului.
- Corpul pachetului (*package body*) - partea „acunsa”, mascată de restul aplicatiei, adica realizarea specificatiei. Corpul defineste cursoare si subprograme, implementand specificatia. Obiectele continute in corpul pachetului sunt fie private, fie publice.

Prin urmare, specificatia defineste interfata utilizatorului cu pachetul, iar corpul pachetului contine codul care implementeaza operatiile definite in

specificație. Crearea unui pachet se face în două etape care presupun crearea specificației pachetului și crearea corpului pachetului.

Un pachet poate cuprinde, fie doar partea de specificație, fie specificația și corpul pachetului. Dacă conține doar specificația, atunci evident pachetul conține doar definiții de tipuri și declaratii de date.

Corpul pachetului poate fi schimbat fără schimbarea specificației pachetului. Dacă specificația este schimbata, aceasta invalidează automat corpul pachetului, deoarece corpul depinde de specificație.

Specificația și corpul pachetului sunt unități compilate separat. Corpul poate fi compilat doar după ce specificația a fost compilată cu succes.

Un pachet are urmatoarea formă generală:

```
CREATE PACKAGE nume_pachet {IS /AS} -- specificația
    /* interfața utilizator, care conține: declaratii de tipuri si obiecte
       publice, specificatii de subprograme */
END [nume_pachet];
CREATE PACKAGE BODY nume_pachet {IS /AS} -- corpul
    /* implementarea, care conține: declaratii de obiecte si tipuri private,
       corpuri de subprograme specificate in partea de interfață */
[BEGIN]
    /* instructiuni de initializare, executate o singura data cand pachetul
       este invocat prima oara de catre sesiunea utilizatorului */
END [nume_pachet];
```

Specificația unui pachet

Specificația unui pachet cuprinde declararea procedurilor, funcțiilor, constantelor, variabilelor și exceptiilor care pot fi accesibile utilizatorilor, adică declararea obiectelor de tip *PUBLIC* din pachet. Acestea pot fi utilizate în proceduri sau comenzi care nu aparțin pachetului, dar care au privilegiul *EXECUTE* asupra acestuia.

Variabilele declarate în specificația unui pachet sunt globale pachetului și sesiunii. Ele sunt initializate (implicit) prin valoarea *NULL*, evident dacă nu este specificată explicit o alta valoare.

```
CREATE [OR REPLACE] PACKAGE [schema.]nume_pachet
[AUTHID {CURRENTUSER /DEFINER}] {IS /AS}
specificație_PL/SQL;
```

Specificatie_PL/SQL poate include declaratii de tipuri, variabile, cursoare, exceptii, functii, proceduri, pragma etc. În secțiunea declarativa, un obiect trebuie declarat inainte de a fi referit.

Optiunea *OR REPLACE* este specificata daca exista deja corpul pachetului. Clauzele *IS* si *AS* sunt echivalente, dar daca se foloseste *PROCEDURE BUILDER* este necesara optiunea *IS*.

Clauza *AUTHID* specifica faptul ca subprogramele pachetului se executa cu drepturile proprietarului (implicit) sau ale utilizatorului curent. De asemenea, aceasta clauza precizeaza daca referintele la obiecte sunt rezolvate in schema proprietarului subprogramului sau a utilizatorului curent.

Corpus unui pachet

Corpus unui pachet contine codul *PL/SQL* pentru obiectele declarate in specificatia acestuia si obiectele private pachetului. De asemenea, corpus poate include o sectiune declarativa in care sunt specificate definitii locale de tipuri, variabile, constante, proceduri si functii locale. Obiectele private sunt vizibile numai in interiorul corpului pachetului si pot fi accesate numai de catre functiile si procedurile din pachetul respectiv. Corpus pachetului este optional si nu este necesar sa fie creat daca specificatia pachetului nu contine declaratii de proceduri sau functii.

Este importanta ordinea in care subprogramele sunt definite in interiorul corpului pachetului. O variabila trebuie declarata inainte ca sa fie referita de alta variabila sau subprogram, iar un subprogram privat trebuie declarat sau definit inainte de a fi apelat de alte subprograme.

CREATE [OR REPLACE] PACKAGE BODY [schema.]nume_pachet {IS / AS} corp_pachet;

Un pachet este instantiat cand este apelat prima data. Aceasta presupune ca pachetul este citit de pe disc in memorie si este executat codul compilat a subprogramului apelat. În acest moment, memoria este alocata tuturor variabilelor definite in pachet.

În multe cazuri este necesar sa se faca o initializare atunci cand pachetul este instantiat prima data intr-o sesiune. Aceasta se realizeaza prin adaugarea unei

secțiuni de inițializare (optională) în corpul pachetului secțiune încadrată între cuvintele cheie *BEGIN* și *END*. Secțiunea conține un cod de inițializare care este executat atunci când pachetul este invocat pentru prima dată.

Crearea pachetului face ca acesta să fie disponibil pentru utilizatorul care l-a creat sau orice cont de utilizator căruia i s-a acordat privilegiul *EXECUTE*.

Referința la o declarație sau la un obiect specificat în pachet se face prefixând numele obiectului cu numele pachetului. În corpul pachetului, obiectele din specificație pot fi referite fără a specifica numele pachetului.

Procesul de creare a specificei și corpului unui pachet urmează același algoritm ca cel întâlnit în crearea subprogramelor *PL/SQL* independente.

- sunt verificate erorile sintactice și semantice, iar modulul este depus în dicționarul datelor;
- sunt verificate instrucțiunile *SQL* individuale, adică dacă obiectele referite există și dacă utilizatorul le poate accesa;
- sunt comparate declarațiile de subprograme din specificația pachetului cu cele din corpul pachetului (dacă au același număr și tip de parametri). Orice eroare detectată la compilarea specificei sau a corpului pachetului este marcată în dicționarul datelor.

După ce specificația și corpul pachetului sunt compilate, ele devin obiecte în schema curentă. În vizualizarea *USER OBJECTS* din dicționarul datelor, vor fi două noi linii:

```
OBJECTTYPE OBJECTNAME
PACKAGE nume_pachet
PACKAGE BODY nume_pachet
```

Modificarea și suprimarea pachetelor

Modificarea unui pachet presupune de fapt recompilarea sa (pentru a putea modifica metoda de acces și planul de execuție) și se realizează prin comanda:

ALTER PACKAGE [schema.]nume_pachet COMPILE [PACKAGE / BODY]

Schimbarea corpului pachetului nu cere recompilarea construcțiilor dependente, în timp ce schimbările în specificația pachetului solicită recompilarea fiecarui subprogram stocat care referențiază pachetul.

Dacă se dorește modificarea sursei, utilizatorul poate recrea pachetul (cu opțiunea *REPLACE*) pentru a-l înlocui pe cel existent.

DROP PACKAGE [schema.]nume_pachet [PACKAGE / BODY]

Daca in cadrul comenzii apare optiunea *BODY* este distrus doar corpul pachetului, in caz contrar sunt distruse atat specificația, cat si corpul pachetului. Daca pachetul este distrus, toate obiectele dependente de acesta devin invalide. Daca este distrus numai corpul, toate obiectele dependente de acesta raman valide. În schimb, nu pot fi apelate subprogramele declarate in specificatia pachetului, pana cand nu este recreat corpul pachetului.

Pentru ca un utilizator sa poata distruge un pachet trebuie ca fie pachetul sa apartina schemei utilizatorului, fie utilizatorul sa posede privilegiul de sistem *DROP ANY PROCEDURE*.

Una din posibilitatile interesante oferite de pachetele *PL/SQL* este aceea de a crea proceduri/functii *overload*. Procesul implica definirea unui numar de proceduri cu acelasi nume, dar care difera prin numarul si tipul parametrilor pe care le folosesc in fiecare instanta a procedurii implementata separat in corpul pachetului. Acest tip de programare este folositor cand este necesara o singura functie care sa execute aceeasi operatie pe obiecte de tipuri diferite (diferite tipuri de parametri de intrare). Cand este apelata o procedura *overload* sistemul decide pe baza tipului si numarului de parametri care instanta a procedurii va fi executata. Numai subprogramele locale sau apartinand unui pachet pot fi *overload*. Subprogramele *stand-alone* nu pot fi *overload*.

Utilizarea unui pachet se realizeaza in functie de mediul (*SQL* sau *PL/SQL*) care solicita un obiect din pachetul respectiv.

- 1) În *PL/SQL* se face prin referirea:

nume_pachet.nume_componenta [(lista_de_argumete)];

- 2) În *SQL*Plus* se face prin comanda:

EXECUTE nume_pachet.nume_componenta [(lista_de_argumete)]

Exemplu:

Sa se creeze un pachet ce include o procedura prin care se verifica daca o combinatie specificata dintre atributele *cod_artist* si *stil* este o combinatie care exista in tabelul *opera*.

```

CREATE PACKAGE verif_pachet IS PROCEDURE verifica
(p_idartist IN opera.cod_artist%TYPE, p_stil      IN
 opera.stil%TYPE);
END verif_pachet;
/
CREATE OR REPLACE PACKAGE BODY verif_pachet IS
  NUMBER := 0;
  CURSOR opera_cu IS
    SELECT cod_artist, stil FROM opera;
  TYPE opera_table_tip IS TABLE OF opera_cu%ROWTYPE
    INDEX BY BINARY INTEGER; art_stil opera_table_tip;
  PROCEDURE verifica
  (p_idartist IN opera.cod_artist%TYPE, p_stil      IN
   opera.stil%TYPE);
  IS
  BEGIN
    FOR k IN art_stil.FIRST..art_stil.LAST LOOP IF
      p_idartist = art_stil(k).cod_artist AND p_stil =
      art_stil(k).stil THEN RETURN;
    END IF;
  END LOOP;
  RAISE_APPLICATION_ERROR (-20777, 'nu este buna
combinatia');
  END verifica;
  BEGIN
    FOR ope_in IN opera_cu LOOP art_stil(i) := ope_in; i
      := i+1;
    END LOOP;
  END verif_pachet;
/

```

Utilizarea in *PL/SQL* a unui obiect (*verifica*) din pachet se face prin:

```
verif_pachet.verifica (7935, 'impresionism');
```

Utilizarea in *SQL*Plus* a unui obiect (*verifica*) din pachet se face prin: **EXECUTE**

```
verif_pachet.verifica (7935, 'impresionism')
```

Observații:

- Un declansator nu poate apela o procedura sau o funcție ce conține comenziile *COMMIT*, *ROLLBACK*, *SAVEPOINT*. Prin urmare, pentru flexibilitatea apelului (de către declansatori) subprogramelor continute în pachete, trebuie verificat ca nici una din procedurile sau funcțiile pachetului nu contin aceste comenzi.
- Procedurile și funcțiile continute într-un pachet pot fi referite din fisiere *iSQL*Plus*, din subprograme stocate *PL/SQL*, din aplicații client (de exemplu, *Oracle Forms* sau *Power Builder*), din declansatori (baza de date), din programe aplicatie scrise în limbaje de generația a 3-a.
- Într-un pachet nu pot fi referite variabile gazda.
- Într-un pachet, mai exact în corpul acestuia, sunt permise declaratii *forward*.
- Funcțiile unui pachet pot fi utilizate (cu restricții) în comenzi *SQL*.

Dacă un subprogram dintr-un pachet este apelat de un subprogram *stand-alone* trebuie remarcat că:

- dacă corpul pachetului se schimbă, dar specificația pachetului nu se schimbă, atunci subprogramul care referă o construcție a pachetului ramane valid;
- dacă specificația pachetului se schimbă, atunci subprogramul care referă o construcție a pachetului, precum și corpul pachetului sunt invalidate.

Dacă un subprogram *stand-alone* referit de un pachet se schimbă, atunci întregul corp al pachetului este invalidat, dar specificația pachetului ramane validă.

Pachete predefinite

PL/SQL conține pachete predefinite utilizabile pentru dezvoltare de aplicații și care sunt deja compilate în baza de date. Aceste pachete adaugă noi funcționalități limbajului, protocoale de comunicare, acces la fisierele sistemului etc. Apelarea unor proceduri din aceste pachete solicită prefixarea numelui procedurii cu numele pachetului.

Dintre cele mai importante pachete predefinite se remarcă:

- *DBMSOUTPUT* (permite afisarea de informații);
- *DBMS DDL* (furnizează accesul la anumite comenzi *LDD* care pot fi folosite în programe *PL/SQL*);

- *UTL FILE* (permite citirea din fișierele sistemului de operare, respectiv scrierea în astfel de fișiere);
- *UTLHTTP* (folosește *HTTP* pentru accesarea din *PL/SQL* a datelor de pe *Internet*);
- *UTL TCP* (permite aplicațiilor *PL/SQL* să comunice cu *server-e* externe utilizând protocolul *TCP/IP*);
- *DBMSJOB* (permite planificarea programelor *PL/SQL* pentru execuție și execuția acestora);
- *DBMSSQL* (accedează baza de date folosind *SQL* dinamic);
- *DBMSPIPE* (permite operații de comunicare între două sau mai multe procese conectate la aceeași instanță *Oracle*);
- *DBMSLOCK* (permite folosirea exclusivă sau partajată a unei resurse),
- *DBMSSNAPSHOT* (permite exploatarea cliseelor);
- *DBMSUTILITY* (ofere utilități *DBA*, analizează obiectele unei scheme particulare, verifică dacă *server-ul* lucrează în mod paralel etc.);
- *DBMSLOB* (realizează accesul la date de tip *LOB*, permitând compararea datelor *LOB*, adăugarea de date la un *LOB*, copierea datelor dintr-un *LOB* în altul, stergerea unor porțiuni din date *LOB*, deschiderea, închiderea și regăsirea de informații din date *BFILE* etc.).

DBMSSTANDARD este un pachet predefinit fundamental prin care se declară tipurile, excepțiile, subprogramele care sunt utilizabile automat în programele *PL/SQL*. Continutul pachetului este vizibil tuturor aplicațiilor. Pentru referirea componentelor sale nu este necesară prefixarea cu numele pachetului. De exemplu, utilizatorul poate folosi ori de câte ori are nevoie în aplicația sa funcția *ABS* (*x*), apartinând pachetului *DBMS STANDARD*, care reprezintă valoarea absolută a numărului *x*, fără a prefixa numele funcției cu numele pachetului.

Pachetul *DBMS_OUTPUT*

DBMSOUTPUT permite afișarea de informații atunci când se executa un program *PL/SQL* (trimite mesajele din orice bloc *PL/SQL* într-un buffer în BD).

DBMS OUTPUT lucrează cu un *buffer* (continut în *SGA*) în care poate fi scrisă informație utilizând procedurile *PUT*, *PUTLINE* și *NEW LINE*. Aceasta informație poate fi regăsită folosind procedurile *GETLINE* și *GETLINES*. Procedura *DISABLE* dezactivează toate apelurile la pachetul *DBMS OUTPUT* (cu excepția procedurii *ENABLE*) și curată *buffer-ul* de orice informație.

Inserarea în *buffer* a unui sfârșit de linie se face prin procedura *NEW LINE*.

Procedura *PUT* depune (scrise) informatie in *buffer*, informatie care este de tipul *NUMBER*, *VARCHAR2* sau *DATE*. *PUTLINE* are același efect ca procedura *PUT*, dar inserează și un sfarsit de linie. Procedurile *PUT* și *PUT LINE* sunt *overload*, astfel încât informația poate fi scrisă în format nativ (*VARCHAR2*, *NUMBER* sau *DATE*).

Procedura *GETLINE* regăsește o singură linie de informație (de dimensiune maximă 255) din *buffer* (dar sub forma de sir de caractere). Procedura *GETLINES* regăsește mai multe linii (*nrlinii*) din *buffer* și le depune într-un tablou (*numetab*) *PL/SQL* având tipul sir de caractere. Valorile sunt plasate în tabel începând cu linia zero. Specificația este următoarea:

```
TYPE string255_table IS TABLE OF VARCHAR2(255)
INDEX BY BINARY_INTEGER;
PROCEDURE GET_LINES
(numetab OUT string255_table, nr_linii IN OUT
INTEGER);
```

Parametrul *nr linii* este și parametru de tip *OUT*, deoarece numărul liniilor solicitate poate să nu coincida cu numărul de liniî din *buffer*. De exemplu, pot fi solicitate 10 liniî, iar în *buffer* sunt doar 6 liniî. Atunci doar primele 6 liniî din tabel sunt definite.

Dezactivarea referirilor la pachet se poate realiza prin procedura *DISABLE*, iar activarea referirilor se face cu ajutorul procedurii *ENABLE*.

Exemplu:

Urmatorul exemplu plasează în *buffer* (apelând de trei ori procedura *PUT*) toate informațiile într-o singură linie.

```
DBMS_OUTPUT.PUT(:opera.valoare||:opera.cod_artist);
DBMS_OUTPUT.PUT(:opera.cod_opera);
DBMS_OUTPUT.PUT(:opera.cod_galerie);
```

Dacă aceste trei comenzi sunt urmate de comanda

```
DBMS_OUTPUT.NEW_LINE;
```

atunci informația respectivă va fi găsită printr-un singur apel *GET LINE*. Altfel, nu se va vedea nici un efect al acestor comenzi deoarece *PUT* plasează informația în *buffer*, dar nu adaugă sfârșit de linie.

Când este utilizat pachetul *DBMSOUTPUT* pot să apară erorile *buffer overflow* și *line length overflow*. Tratarea acestor erori se face apelând procedura

```
RAISE APPLICATION ERROR din pachetul standard DBMS STANDARD.
```

Pachetul *DBMS_SQL*

Pachetul *DBMSSQL* permite folosirea dinamica a comenziilor *SQL* în proceduri stocate sau în blocuri anonime și analiza gramaticală a comenziilor *LDD*. Aceste comenzi nu sunt incorporate în programul sursă, ci sunt depuse în siruri de caractere. O comandă *SQL* dinamică este o instrucțiune *SQL* care conține variabile ce se pot schimba în timpul executiei. De exemplu, pot fi utilizate instrucțiuni *SQL* dinamice pentru:

- a crea o procedură care operează asupra unui tabel al căruia nume nu este cunoscut decât în momentul executiei;
- a scrie și executa o comandă *LDD*;
- a scrie și executa o comandă *GRANT*, *ALTER SESSION* etc.

În *PL/SQL* aceste comenzi nu pot fi executate static. Pachetul *DBMS_SQL* permite, de exemplu, ca într-o procedură stocată să folosești comanda *DROP TABLE*. Evident, folosirea acestui pachet pentru a executa comenzi *LDD* poate genera interblocari. De exemplu, pachetul este utilizat pentru a sterge o procedură care însă este utilizată.

SQL dinamic suportă toate tipurile de date *SQL*, dar nu suportă cele specifice *PL/SQL*. Una excepție o constituie faptul că o înregistrare *PL/SQL* poate să apară în clauza *INTO* a comenzi *EXECUTE IMMEDIATE*.

Orice comandă *SQL* trebuie să treacă prin niște etape, cu observația că anumite etape pot fi evitate. Etapele presupun: analizarea gramaticală a comenzi, adică verificarea sintactică a comenzi, validarea acesteia, asigurarea că toate referințele la obiecte sunt corecte și asigurarea că există privilegiile referitoare la acele obiecte (*parse*); obținerea de valori pentru variabilele de legătură din comandă *SQL* (*binding variables*); executarea comenzi (*execute*); selectarea randurilor rezultatului și încarcarea acestor randuri (*fetch*).

Dintre subprogramele pachetului *DBMS_SQL*, care permit implementarea etapelor amintite anterior se remarcă:

- *OPEN CURSOR* (deschide un nou cursor, adică se stabilește o zonă de memorie în care este procesată comanda *SQL*);
- *PARSE* (stabilește validitatea comenzi *SQL*, adică se verifică sintaxa instrucțiunii și se asociază cursorului deschis);
- *BIND VARIABLE* (leagă valoarea datei de variabila corespunzătoare din comandă *SQL* analizată)
- *EXECUTE* (execută comanda *SQL* și returnează numărul de linii procesate);

- *FETCH ROWS* (regaseste o linie pentru un cursor specificat, iar pentru mai multe linii foloseste un *LOOP*);
- *CLOSECURSOR* (inchide cursorul specificat).

Sa se construiasca o procedura care foloseste *SQL* dinamic pentru a sterge liniile unui tabel specificat (*num tab*). Subprogramul furnizeaza ca rezultat numarul liniilor sterse (*nrlin*).

```
CREATE OR REPLACE PROCEDURE sterge_linii (num_tab IN
                                         VARCHAR2, nr_lin OUT NUMBER)
AS
  nume_cursor INTEGER;
BEGIN
  nume_cursor := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQLPARSE (nume_cursor, 'DELETE FROM' || num_tab, DBMS_SQL.V7);
  nr_lin := DBMS_SQLEXECUTE (nume_cursor);
  DBMS_SQLCLOSE_CURSOR (nume_cursor);
END;
```

Argumentul *DBMSSQL.V7* reprezinta modul (versiunea 7) in care *Oracle* trateaza comenzile *SQL*. Stergerea efectiva a liniilor tabelului *opera* se realizeaza:

```
VARIABLE linii_sterse NUMBER
EXECUTE sterge_linii ('opera', :linii_sterse)
PRINT linii_sterse
```

Pentru a executa o instructiune *SQL* dinamic poate fi utilizata si comanda *EXECUTE IMMEDIATE*. Comanda contine o clauza optionala *INTO* care este utilizabila pentru interogari ce returneaza o singura linie. Pentru o cerere care returneaza mai multe linii trebuie folosite comenzile *OPEN FOR*, *FETCH*, *CLOSE*.

```
CREATE OR REPLACE PROCEDURE sterge_linii (num_tab IN
                                         VARCHAR2, nr_lin OUT NUMBER)
IS
BEGIN
  EXECUTE IMMEDIATE 'DELETE FROM' || num_tab; nr_lin :=
    SQL%ROWCOUNT;
END;
```

Procedura se poate apela printr-o sevenita identica cu cea prezentata anterior.

Pachetul DBMS_DDL

Pachetul *DBMSDDL* furnizeaza accesul la anumite comenzi *LDD* care pot fi folosite in subprograme *PL/SQL* stocate. În felul acesta, pot fi accesate (in *PL/SQL*) comenzile *ALTER* sau *ANALYZE*.

Pachetul include procedura *ALTER_COMPILE* care permite recompilarea programului modificat (procedura, funcție, declanșator, pachet, corp pachet).

ALTER_COMPILE (tipobiect, numeschema, nume obiect);

Procedura este echivalenta cu instructiunea *SQL*:

ALTER PROCEDURE / FUNCTION / PACKAGE [schema.] nume COMPILE [BODY]

Cu ajutorul procedurii *ANALYZEOBJECT* poate fi analizat un obiect de tip *table*, *cluster* sau *index*. Procedura furnizeaza statistici referitoare la obiectele amintite. De exemplu, se pot obtine numarul liniilor unui tabel, numarul de blocuri ale unui tabel, lungimea medie a unei lini, numarul valorilor distincte ale unei col., numarul elementelor *null* dintr-o coloana, distributia datelor (histograma) etc.

ANALYZE_OBJECT (tip_object, nume_schema, nume_object, metoda, numar_liniu_estimate, procent, optiune_metoda, nume_partitie);

Metodele care pot fi utilizate sunt *COMPUTE*, *ESTIMATE* sau *DELETE*. Prin aceste metode se cuantifica distributia datelor si caracteristicile de stocare. *DELETE* determina stergerea statisticilor (depuse in *DD*) referitoare la obiectul analizat. *COMPUTE* calculeaza statistici referitoare la un obiect analizat si le depune in *DD*, iar *ESTIMATE* estimeaza statisticii. Statisticile calculate sau estimate sunt utilizate pentru optimizarea planului de executie a comenziilor *SQL* care acceseaza obiectele analizate.

Procedura este echivalenta cu instructiunea:

***ANALYZE TABLE | CLUSTER | INDEX [nume_schema] nume_object
[metoda] STATISTICS [SAMPLE n] [ROWS / PERCENT]]***

Daca *nume schema* este *null*, atunci se presupune ca este vorba de schema curenta. Daca *tip obiect* este diferit de *table*, *index* sau *cluster*, se declanseaza eroarea *ORA - 20001*. Parametrul *procent* reprezinta procentajul liniilor de estimat si este ignorat daca este specificat numarul liniilor de estimat (*numarliniiestimate*). Implicit, ultimele patru argumente ale procedurii iau valoarea *null*.

Argumentul *optiunemetoda* poate avea forma:

***[FOR TABLE] [FOR ALL INDEXES] [FOR ALL [INDEXED] COLUMNS]
[SIZE n]***

Pentru metoda *ESTIMATE* trebuie sa fie prezenta una dintre aceste optiuni.

Exemplu:

Utilizand pachetul *DBMS_DDL* si metoda *COMPUTE*, sa se creeze o procedura care analizeaza un obiect furnizat ca parametru acestei proceduri.

```

CREATE OR REPLACE PROCEDURE analiza (p_object_tip IN
    VARCHAR2, p_object_nume IN VARCHAR2);
IS
BEGIN
DBMS_DDL.ANALYZE_OBJECT(p_object_tip, USER,
    UPPER(p_object_nume), 'COMPUTE');
END;
/

```

Procedura se testeaza (relativ la tabelul *opera*) in felul urmator:

```

EXECUTE analiza ('TABLE', 'opera')
SELECT LAST_ANALYZED
FROM USER_TABLES
WHERE TABLE_NAME = 'opera';

```

Pachetul DBMS_JOB

Pachetul *DBMSJOB* este utilizat pentru planificarea programelor *PL/SQL* in vederea executiei. Cu ajutorul acestui pachet se pot executa programe *PL/SQL* la momente determinate de timp, se pot sterge sau suspenda programe din lista de planificari pentru executie, se pot rula programe de intretinere in timpul perioadelor de utilizare scazuta etc.

Dintre subprogramele acestui pachet se remarcă:

- *SUBMIT* - adauga un nou *job* in coada de asteptare a job-urilor;
- *REMOVE* - sterge un *job* specificat din coada de asteptare a job-urilor;
- *RUN* - executa imediat un *job* specificat;
- *BROKEN* - dezactiveaza executia unui *job* care este marcat ca *broken* (implicit, orice *job* este *not broken*, iar un *job* marcat *broken* nu se executa);
- *CHANGE* - modifica argumentele *WHAT*, *NEXTDATE*, *INTERVAL*;
- *WHAT* - furnizeaza descrierea unui *job* specificat;
- *NEXTDATE* - da momentul urmatoarei executii a unui *job*;
- *INTERVAL* - furnizeaza intervalul intre diferite executii ale unui *job*.

Fiecare dintre subprogramele pachetului are argumente specifice. De exemplu, procedura *DBMSJOB.SUBMIT* are ca argumente:

- *JOB* - de tip *OUT*, un identificator pentru *job* (*BINARYINTEGER*);

- *WHAT* - de tip *IN*, codul *PL/SQL* care va fi executat ca un *job* (*VARCHAR2*);
- *NEXT DATE* - de tip *IN*, data urmatoarei execuții a job-ului (implicit este *SYSDATE*);
- *INTERVAL* - de tip *IN*, functie care furnizeaza intervalul dintre executiile job-ului (*VARCHAR2*, implicit este *null*);
- *NO PARSE* - de tip *IN*, variabila logica care indica daca job-ul trebuie analizat gramatical (*BOOLEAN*, implicit este *FALSE*).

Daca unul dintre parametri *WHAT*, *INTERVAL* sau *NEXT DATE* are valoarea *null*, atunci este folosita ultima valoare asignata acestora.

Exemplu:

Sa se utilizeze pachetul *DBMS JOB* pentru a plasa pentru executie in coada de asteptare a job-urilor, procedura *verifica* din pachetul *verif_pachet*.

```

VARIABLE num_job NUMBER BEGIN
DBMS_JOB.SUBMIT(
  job => :num_job,
  what   =>
    'verif_pachet.verifica(8973, ''impressionism'')';
next_date => TRUNC(SYSDATE+1), interval =>
  'TRUNC(SYSDATE+1)');
  COMMIT;
END;
/
PRINT num_job

```

Vizualizarea *DBA JOBS* din dictionarul datelor furnizeaza informatii referitoare la starea *job*-urilor din coada de asteptare, iar vizualizarea *DBA JOBS RUNNING* contine informatii despre job-urile care sunt in curs de executie. Vizualizarile pot fi consultate doar de utilizatorii care au privilegiul *SYS.DBAJOBS*.

Exemplu:

```

SELECT JOB, LOG_USER, NEXT_DATE, BROKEN, WHAT
  FROM DBA JOBS;

```

Pachetul *UTL_FILE*

Pachetul *UTLFILE* permite programului *PL/SQL* citirea din fișierele sistemului de operare, respectiv scrierea în aceste fișiere. El este utilizat pentru exploatarea fisierelor text.

Folosind componente ale acestui pachet (functiile *FOPEN* și *ISOPEN*; procedurile *GETLINE*, *PUT*, *PUTLINE*, *PUTF*, *NEW LINE*, *FCLOSE*, *FCLOSEALL*, *FFLUSH*) se pot deschide fisiere, obține text din fisiere, scrie text în fisiere, inchide fisiere.

Pachetul procesează fisierele intr-o manieră clasica:

- verifică dacă fisierul este deschis (funcția *ISOPEN*);
- dacă fisierul nu este deschis, îl deschide și returnează un *handler* de fisier (de tip *UTLFILE.FILETYPE*) care va fi utilizat în urmatoarele operații I/O (funcția *FOPEN*);
- procesează fisierul (citire/scriere din/in fisier);
- inchide fisierul (procedura *FCLOSE* sau *FCLOSEALL*).

Funcția *IS OPEN* verifică dacă un fisier este deschis. Are antetul:

```
FUNCTION      IS_OPEN  (handler_fisier IN FILE_TYPE)
RETURN BOOLEAN;
```

Funcția *FOPEN* deschide un fisier și returnează un handler care va fi utilizat în urmatoarele operații I/O. Parametrul *openmode* este un string care specifică modul cum a fost deschis fisierul.

```
FUNCTION      FOPEN      (locatia      IN      VARCHAR2,
                        nume_fisier    IN      VARCHAR2,
                        open~mode     IN      VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;
```

Prin procedura *GET LINE*, pachetul *UTL FILE* citește o linie de text din fisierul deschis pentru citire și o plasează într-un *buffer* de tip sir de caractere, iar prin procedurile *PUT* și *PUT LINE* scrie un text din *buffer* în fisierul deschis pentru scriere sau adăugare.

Utilizarea componentelor acestui pachet pentru procesarea fisierelor sistemului de operare poate declansa exceptii, dintre care remarcăm:

- *INVALID PATH* - numele sau locația fisierului sunt invalide;
- *INVALID MODE* - parametrul *OPEN MODE* (prin care se specifică dacă fisierul este deschis pentru citire, scriere, adăugare) este invalid;
- *INVALID FILEHANDLE* - *handler-ul* de fisier obținut în urma deschiderii este invalid;

- *INVALIDOPERATION*- operație invalida asupra fișierului;
- *READERROR* - o eroare a sistemului de operare a aparut in timpul operației de citire;
- *WRITE ERROR* - o eroare a sistemului de operare a aparut in timpul operatiei de scriere;
- *INTERNALERROR* - o eroare nespecificata a aparut in *PL/SQL*.

Excepțiile trebuie prefixate cu numele pachetului. Evident, pot aparea și erorile *NO DATA FOUND* și *VALUEERROR*.

Pachetele *DBMS_PIPE* și *DBMS_ALERT*

Pachetul *DBMSPIPE* permite operatii de comunicare intre doua sau mai multe sesiuni conectate la aceeasi baza de date. De exemplu, pachetul poate fi utilizat pentru comunicarea dintre o procedura stocata si un program *Pro*C*. Comunicarea se face prin conducte (*pipe*). O conducta este o zona de memorie utilizata de un proces pentru a transmite informatie altui proces. Informatia trimisa prin conducta este depusa intr-un *buffer* din *SGA*. Toate informatiile din conducta sunt pierdute atunci cand instanta este inchisa.

Conductele sunt asincrone, ele operand independent de tranzactii. Daca un anumit mesaj a fost transmis, nu exista nici o posibilitate de oprire a acestuia, chiar daca sesiunea care a trimis mesajul este derulata inapoi (*rollback*).

Pachetul *DBMS PIPE* este utilizat pentru a trimite mesaje in conducta (*DBMSPIPE.SENDMESSAGE*), mesaje ce constau din date de tip *VARCHAR2*, *NUMBER*, *DATE*, *RAW* sau *ROWID*. Tipurile obiect definite de utilizator si colectiile nu sunt acceptate de acest pachet.

De asemenea, pachetul poate realiza primirea de mesaje din conducta in *buffer-ul* local (*DBMS_PIPE.RECEIVE_MESSAGE*), accesarea urmatorului articol din *buffer* (*DBMSPIPE.UNPACKMESSAGE*), crearea unei noi conducte (*DBMS PIPE.CREATEPIPE*) etc.

DBMSALERT este similar pachetului *DBMS PIPE*, fiind utilizat tot pentru comunicarea dintre sesiuni conectate la aceeasi baza de date. Exista totusi cateva deosebiri esentiale.

- *DBMS ALERT* asigura o comunicare sincrona.
- Un mesaj trimis prin *DBMS PIPE* este primit de un singur destinatar (cititor) chiar daca exista mai multi pe conducta, pe cand cel trimis prin *DBMS ALERT* poate fi primit de mai multi cititori simultan.
- Daca doua mesaje sunt trimise printr-o conducta (inainte ca ele sa fie citite), ambele vor fi primite de destinatar prin *DBMS PIPE*. În cazul pachetului *DBMS ALERT*, doar cel de al 2-lea mesaj va fi primit.

Pachete predefinite furnizate de Oracle

Oracle furnizează o varietate de pachete predefinite care simplifică administrarea bazei de date și oferă noi funcționalități legate de noile caracteristici ale sistemului. Dintre pachetele introduse se remarcă:

- *DBMS REDEFINITION* - permite reorganizarea *online* a tabelelor;
- *DBMS LIBCACHE* - permite extragerea de comenzi *SQL* și *PL/SQL* dintr-o instantă distanță într-o formă locală (vor fi compilate local, dar nu executate);
- *DBMSLOGMNRCDCPUBLISH* - realizează captarea schimbarilor din tabelele bazei de date (identifică datele adăugate, modificate sau stșe și editează aceste informații într-o formă utilizabilă în aplicații);
- *DBMSLOGMNRCDCSUBSCRIBE* - face posibilă vizualizarea și interogarea schimbarilor din datele care au fost captate cu pachetul *DBMSLOGMNRCDCPUBLISH*;
- *DBMS METADATA* - furnizează informații despre obiectele bazei de date;
- *DBMSRESUMABLE* - permite setarea limitelor de spațiu și timp pentru o operație specificată, operația fiind suspendată dacă sunt depășite aceste limite;
- *DBMSXMLQUERY*, *DBMSXMLSAVE*, *DBMS XMLGEN* - permit prelucrarea și conversia datelor *XML* (*XMLGEN* convertește rezultatul unei cereri *SQL* în format *XML*, *XMLQUERY* este similară lui *XMLGEN*, doar că este scrisă în C, iar *XMLSA VE* face conversia din format *XML* în date ale bazei);
- *UTLINADDR* - returnează numele unei gazde locale sau distante a cărei adresă *IP* este cunoscută și reciproc, returnează adresă *IP* a unei gazde careia își cunoaște numele (de exemplu, www.oracle.com);
- *DBMSAQELM* - furnizează proceduri și funcții pentru gestionarea configurației cozilor de mesaje asincrone prin *e-mail* și *HTTP*;
- *DBMS FGA* - asigură întreținerea unor funcții de securitate;
- *DBMS FLASHBACK* - permite trecerea la o versiune a bazei de date corespunzătoare unei unități de timp specificate sau unui *SCN* (*system change number*) dat, în felul acesta putând fi recuperate linii stșe sau mesaje *e-mail* distruse;
- *DBMS TRANSFORM* - furnizează subprograme ce permit transformarea unui obiect (expresie *SQL* sau funcție *PL/SQL*) de un anumit tip (sursă) într-un obiect având un tip (destinație) specificat;

Declansatori

Declansatori in *PL/SQL*

Un declansator (*trigger*) este un bloc *PL/SQL* sau apelul (*CALL*) unei proceduri *PL/SQL*, care se executa automat ori de cate ori are loc un anumit eveniment „declansator“ Evenimentul poate consta din:

- modificarea unui tabel sau a unei vizualizari,
- actiuni sistem
- anumite actiuni utilizator.

Blocul *PL/SQL* poate fi asociat unui tabel, unei vizualizari, unei scheme sau unei baze de date.

La fel ca si pachetele, declansatorii nu pot fi locali unui bloc sau unui pachet, ei trebuie depusi ca obiecte independente in baza de date.

Folosirea declansatorilor garanteaza faptul ca atunci cand o anumita operatie este efectuata, automat sunt executate niste actiuni asociate. Evident, nu trebuie introdusi declansatori care ar putea sa substituie functionalitati oferite deja de sistem. De exemplu, nu are sens sa fie definiti declansatori care sa implementeze regulile de integritate ce pot fi definite, mai simplu, prin constrangeri declarative.

Tipuri de declansatori

Declansatorii pot fi:

- la nivel de baza de date (*database triggers*);
- la nivel de aplicatie (*application triggers*).

Declansatorii baza de date se executa automat ori de cate ori are loc:

- o actiune (comanda *LMD*) asupra datelor unui tabel;
- o actiune (comanda *LMD*) asupra datelor unei vizualizari;
- o comanda *LDD* (*CREATE*, *ALTER*, *DROP*) referitoare la anumite obiecte ale schemei sau ale bazei;
- un eveniment sistem (*SHUTDOWN*, *STARTUP*);
- o actiune a utilizatorului (*LOGON*, *LOGOFF*);
- o eroare (*SERVERERROR*, *SUSPEND*).

Declansatorii baza de date sunt de trei tipuri:

- declansatori *LMD* - activati de comenzi *LMD* (*INSERT*, *UPDATE* sau *DELETE*) executate asupra unui tabel al bazei de date;

- declansatori *INSTEAD OF* - activati de comenzi *LMD* execute asupra unei vizualizari (relationale sau obiect);
- declansatori sistem - activati de un eveniment sistem (oprirea sau pornirea bazei), de comenzi *LDD* (*CREATE*, *ALTER*, *DROP*), de conectarea (deconectarea) unui utilizator. Ei sunt definiti la nivel de schema sau la nivel de baza de date.

Declansatorii asociati unui tabel (stocati in baza de date) vor actiona indiferent de aplicatia care a efectuat operatia *LMD*. Daca operatia *LMD* se refera la o vizualizare, declansatorul *INSTEAD OF* defineste actiunile care vor avea loc, iar daca aceste actiuni includ comenzi *LMD* referitoare la tabele, atunci declansatorii asociati acestor tabele sunt si ei, la randul lor, activati.

Daca declansatorii sunt asociati unei baze de date, ei se declanseaza pentru fiecare eveniment, pentru toti utilizatorii. Daca declansatorii sunt asociati unei scheme sau unui tabel, ei se declanseaza numai daca evenimentul declansator implica acea schema sau acel tabel. Un declansator se poate referi la un singur tabel sau la o singura vizualizare.

Declansatorii aplicatie se executa implicit ori de cate ori apare un eveniment particular intr-o aplicatie (de exemplu, o aplicatie dezvoltata cu *Developer Suite*). *Form Builder* utilizeaza frecvent acest tip de declansatori (*form builder triggers*). Ei pot fi declansati prin apasarea unui buton, prin navigarea pe un camp etc. In acest capitol se va face referinta doar la declansatorii baza de date.

Atunci cand un pachet sau un subprogram este depus in dictionarul datelor, alaturi de codul sursa este depus si *p-codul* compilat. In mod similar se intampla si pentru declansatori. Prin urmare, un declansator poate fi apelat fara recompilare. Declansatorii pot fi invalidati in aceeasi maniera ca pachetele si subprogramele. Daca declansatorul este invalidat, el va fi recompilat la urmatoarea activare.

Crearea declansatorilor *LMD*

Declansatorii *LMD* sunt creati folosind comanda *CREATE TRIGGER*.

Numele declansatorului trebuie sa fie unic printre numele declansatorilor din cadrul aceleasi scheme, dar poate sa coincida cu numele altor obiecte ale acesteia (de exemplu, tabele, vizualizari sau proceduri).

La crearea unui declansator este obligatorie una dintre optiunile *BEFORE* sau *AFTER*, prin care se precizeaza momentul in care este executat corpul declansatorului.

```

CREATE [OR REPLACE] TRIGGER [schema.]nume_declansator
{BEFORE | AFTER}
{DELETE | INSERT | UPDATE [OF coloana[, coloana ... ] ] }
[OR {DELETE | INSERT | UPDATE [OF coloana[, coloana ... ] ] } ... ]
ON [schema.]nume_tabel
[REFERENCING {OLD [AS] vechi NEW [AS] nou / NEW [AS] nou
OLD [AS] vechi } ]
[FOR EACH ROW]
[WHEN(conditie) ]
corpdeclansator (bloc PL/SQL sau apelul unei proceduri);

```

Declararea unui declansator trebuie sa cuprinda tipul comenzi *SQL care* duce la executarea declansatorului si tabelul asociat acestuia. In ceea ce priveste tipul comenzi *SQL* care va duce la executarea declasatorului, sunt incluse urmatoarele tipuri de optiuni: *DELETE*, *INSERT*, *UPDATE* sau o combinare a acestora cu operatorul logic *OR*. Cel putin una dintre optiuni este obligatorie.

In declararea declansatorului este specificat tabelul asupra caruia va fi executat declansatorul. Daca declansatorul este de tip *UPDATE*, atunci pot fi enumerate coloanele pentru care acesta se va executa.

In corpul fiecarui declansator pot fi cunoscute valorile coloanelor atat inainte de modificarea unei linii, cat si dupa modificarea acesteia. Valoarea unei coloane inainte de modificare este referita prin atributul *OLD*, iar dupa modificar, prin atributul *NEW*. Prin intermediul clauzei optionale *REFERENCING* din sintaxa comenzi de creare a declansatorilor, attributele *NEW* si *OLD* pot fi redenumite. In interiorul blocului PL/SQL, coloanele prefixate prin *OLD* sau *NEW* sunt considerate variabile externe, deci trebuie prefixate cu ":".

Un declansator poate activa alt declansator, iar acesta la randul sau poate activa alt declansator etc. Aceasta situatie (declansatori in cascada) poate avea insa efecte imprevizibile. Sistemul *Oracle* permite maximum 32 declansatori in cascada. Numarul acestora poate fi limitat (utilizand parametrul de initializare *OPENCURSORS*), deoarece pentru fiecare executie a unui declansator trebuie deschis un nou cursor.

Declansatorii la nivel de baze de date pot fi de doua feluri:

- la nivel de instructiune (*statement level trigger*);
- la nivel de linie (*row level trigger*).

Declansatori la nivel de instructiune

Declansatorii la nivel instructiune sunt executati o singura data pentru instructiunea declansatoare, indiferent de numarul de linii afectate (chiar daca nici o linie nu este afectata). Un declansator la nivel de instructiune este util daca actiunea declansatorului nu depinde de informatiile din liniile afectate.

Exemplu:

Programul de lucru la administratia muzeului este de luni pana vineri, in intervalul (8:00 a.m. - 10:00 p.m.). Sa se construiasca un declansator la nivel de instructiune care impiedica orice activitate asupra unui tabel al bazei de date, in afara acestui program.

```
CREATE OR REPLACE PROCEDURE verifica IS BEGIN
IF ((TO_CHAR(SYSDATE,'D') BETWEEN 2 AND 6)
AND
TO_DATE(TO_CHAR(SYSDATE,'hh24:mi'), 'hh24:mi')
NOT BETWEEN TO_DATE('08:00','hh24:mi')
                AND TO_DATE('22:00','hh24:mi'))
THEN
    RAISE_APPLICATION_ERROR (-27733, 'nu puteti reactualiza acest
tabel deoarece sunteți în afara programului');
END IF;
END verifica;
/
CREATE OR REPLACE TRIGGER BIUD_tabell
BEFORE INSERT OR UPDATE OR DELETE ON tabell BEGIN
verifica;
END;
```

Declansatori la nivel de linie

Declansatorii la nivel de linie sunt creati cu optiunea *FOR EACH ROW*. In acest caz, declansatorul este executat pentru fiecare linie din tabelul afectat, iar daca evenimentul declansator nu afecteaza nici o linie, atunci declansatorul nu este executat. Daca optiunea *FOR EACH ROW* nu este inclusa, declansatorul este considerat implicit la nivel de instructiune.

Declansatorii la nivel linie nu sunt performanti daca se fac frecvent reactualizari pe tabele foarte mari.

Restrictive declansatorilor pot fi incluse prin specificarea unei expresii booleene in clauza *WHEN*. Aceasta expresie este evaluata pentru fiecare linie afectata de catre declansator. Declansatorul este executat pentru o linie, doar daca expresia este adevarata pentru acea linie. Clauza *WHEN* este valida doar pentru declansatori la nivel de linie.

Exemplu:

Sa se implementeze cu ajutorul unui declansator constrangerea ca valorile operelor de arta nu pot fi reduse (trei variante).

Varianta 1:

```
CREATE OR REPLACE TRIGGER verifica_valoare BEFORE UPDATE OF
    valoare ON opera FOR EACH ROW
WHEN (NEW.valoare < OLD.valoare)
BEGIN
    RAISE_APPLICATION_ERROR (-20222, 'valoarea unei opere de
arta nu poate fi micsorata');
END;
```

Varianta 2:

```
CREATE OR REPLACE TRIGGER verifica_valoare BEFORE UPDATE OF
    valoare ON opera FOR EACH ROW BEGIN
    IF (:NEW.valoare < :OLD.valoare) THEN
        RAISE_APPLICATION_ERROR (-20222, 'valoarea unei opere de
arta nu poate fi micsorata');
    END IF;
END;
```

Varianta 3:

```
CREATE OR REPLACE TRIGGER verifica_valoare BEFORE UPDATE OF
    valoare ON opera FOR EACH ROW
WHEN (NEW.valoare < OLD.valoare)
CALL procedura -- care va face actiunea RAISE ...
/
```

Accesul la vechile si noile valori ale coloanelor liniei curente, afectata de evenimentul declansator, se face prin: *OLD.nume_coloana* (vechea valoare), respectiv prin *NEW.nume_coloana* (noua valoare). In cazul celor trei comenzi *LMD*, aceste valori devin:

INSERT : NEW.nume_coloana noua valoare
(: OLD.nume_coloana *NULL*);
UPDATE : NEW.nume_coloana noua valoare
: OLD.nume_coloana vechea valoare;
DELETE (: NEW.nume_coloana *NULL*)
: OLD.nume_coloana vechea valoare.

Exemplu:

Se presupune ca pentru fiecare galerie exista doua campuri (*min_valoare* si *max_valoare*) in care se retin limitele minime si maxime ale valorile operelor din galeria respectiva. Sa se implementeze cu ajutorul unui declansator constrangerea ca, daca aceste limite s-ar modifica, valoarea oricarei opere de arta trebuie sa ramana cuprinsa intre noile limite.

```
CREATE OR REPLACE TRIGGER verifica_limite
BEFORE UPDATE OF min_valoare, max_valoare ON galerie FOR
EACH ROW      - -
DECLARE
v_min_val opera.valoare%TYPE; v_max_val opera.valoare%TYPE;
e_invalid EXCEPTION;
BEGIN
SELECT MIN(valoare), MAX(valoare)
INTO v_min_val, v_max_val
FROM opera
WHERE cod_galerie = :NEW.cod_galerie;
IF (v_min_val < :NEW.min_valoare) OR (v_max_val >
:NEW.max_valoare) THEN RAISE e_invalid;
END IF;
EXCEPTION
WHEN e_invalid THEN
RAISE_APPLICATION_ERROR (-20567, 'Exista opere de arta ale
caror valori sunt in afara domeniului permis');
END verifica_limite;
/
```

Ordinea de executie a declansatorilor

PL/SQL permite definirea a mai multor tipuri de declansatori care sunt obtinuti prin combinarea proprietatii de moment (timp) al declansarii (*BEFORE*, *AFTER*), cu proprietatea nivelului la care actioneaza (nivel linie, nivel instructiune) si cu tipul operatiei atasate declansatorului (*INSERT*, *UPDATE*, *DELETE*).

De exemplu, *BEFORE INSERT* actioneaza o singura data, inaintea executarii unei instructiuni *INSERT*, iar *BEFORE INSERT FOR EACH ROW* actioneaza inainte de inserarea fiecarei noi inregistrari.

Declansatorii sunt activati cand este executata o comanda *LMD*. La aparitia unei astfel de comenzi se executa cateva actiuni care vor fi descrise in continuare.

1. Se executa declansatorii la nivel de instructiune *BEFORE*.
2. Pentru fiecare linie afectata de comanda *LMD*:

- 2.1. se executa declansatorii la nivel de linie *BEFORE*;
 - 2.2. se blocheaza si se modifica linia afectata (se executa comanda *LMD*), se verifica constrangerile de integritate (blocarea ramane valabila pana in momentul in care tranzactia este permanentizata);
 - 2.3. se executa declansatorii la nivel de linie *AFTER*.
3. Se executa declansatorii la nivel de instructiune *AFTER*.

Verificarea constrangerii referentiale este amanata dupa executarea declansatorului la nivel linie.

Observeatii:

- In expresia clauzei *WHEN* nu pot fi incluse functii definite de utilizator sau subcereri *SQL*.
- In clauza *ON* poate fi specificat un singur tabel sau o singura vizualizare.
- In interiorul blocului *PL/SQL*, coloanele tabelului prefixate cu *OLD* sau *NEW* sunt considerate variabile externe si deci, trebuie precedate de caracterul „:“.
- Conditia de la clauza *WHEN* poate contine coloane prefixate cu *OLD* sau *NEW*, dar in acest caz, acestea nu trebuie precedate de „:“.
- Declansatorii baza de date pot fi definiti numai pe tabele (exceptie, declansatorul *INSTEAD OF* care este definit pe o vizualizare). Totusi, daca o comanda *LMD* este aplicata unei vizualizari, pot fi activati declansatorii asociati tabelelor care definesc vizualizarea.
- Corpul unui declansator nu poate contine o interogare sau o reactualizare a unui tabel aflat in plin proces de modificare, pe timpul actiunii declansatorului (*mutating table*).
- Blocul *PL/SQL* care descrie actiunea declansatorului nu poate contine comenzi pentru gestiunea tranzactiilor (*COMMIT*, *ROLLBACK*, *SAVEPOINT*). Controlul tranzactiilor este permis, insa, in procedurile stocate. Daca un declansator apeleaza o procedura stocata care executa o comanda referitoare la controlul tranzactiilor, atunci va aparea o eroare la executie si tranzactia va fi anulata.
- Comenzile *DDL* nu pot sa apară decat in declansatorii sistem.
- Corpul declansatorului poate sa contina comenzi *LMD*.
- In corpul declansatorului pot fi referite si utilizate coloane *LOB*, dar nu pot fi modificate valorile acestora.
- Nu este indicata crearea declansatorilor recursivi.
- In corpul declansatorului se pot insera date in coloanele de tip *LONG* si *LONGRAW*, dar nu pot fi declarate variabile de acest tip.

- Daca un tabel este suprimat (se sterge din dictionarul datelor), automat sunt distrusii toti declansatorii asociati tabelului.
- Este necesara limitarea dimensiunii unui declansator. Este preferabil ca o parte din cod sa fie inclusa intr-o procedura stocata si aceasta sa fie apelata din corpul declansatorului.

Sunt doua diferente esentiale intre declansatori si procedurile stocate:

- declansatorii se invoca implicit, iar procedurile explicit;
- **instructiunile LCD (*COMMIT, ROLLBACK, SAVEPOINT*) nu sunt permise in corpul unui declansator.**

Predicate conditionale

In interiorul unui declansator care poate fi executat pentru diferite tipuri de instructiuni *LMD* se pot folosi trei functii booleene prin care se stabileste tipul operatiei execute. Aceste predicate conditionale sunt *INSERTING*, *UPDATING* si *DELETING*.

Functiile booleene nu solicita prefixarea cu numele pachetului si determina tipul operatiei (*INSERT*, *DELETE*, *UPDATE*). De exemplu, predicatul *INSERTING* ia valoarea *TRUE* daca instructiunea declansatoare este *INSERT*. Similar sunt definite predicatele *UPDATING* si *DELETING*. Utilizand aceste predicate, in corpul declansatorului se pot executa secvente de instructiuni diferite, in functie de tipul operatiei *LMD*.

In cazul in care corpul declansatorului este un bloc *PL/SQL* complet (nu o comanda *CALL*), pot fi utilizate atat predicatele *INSERTING*, *UPDATING*, *DELETING*, cat si identificatorii *:OLD*, *:NEW*.

Exemplu:

Se presupune ca in tabelul *galerie* se pastreaza (intr-o coloana numita *total_val*) valoarea totala a operelor de arta expuse in galeria respectiva.

```
UPDATE galerie SET      total_val =
  (SELECT SUM(valoare)
   FROM opera
   WHERE      opera.cod_galerie      =
             galerie.cod_galerie);
```

Reactualizarea acestui camp poate fi implementata cu ajutorul unui declansator in urmatoarea maniera:

```
CREATE OR REPLACE PROCEDURE creste
(v_cod_galerie    IN galerie.cod_galerie%TYPE,
v_val      IN galerie.total_val%TYPE) AS
BEGIN      -
      -
      UPDATE galerie
      SET      total_val = NVL (total_val, 0)      + v_val
```

```

WHERE      cod_galerie = v_cod_galerie;
END creste;
/
CREATE OR REPLACE TRIGGER calcul_val
AFTER INSERT OR DELETE OR UPDATE OF valoare ON opera FOR EACH
ROW BEGIN
IF DELETING THEN
creste (:OLD.cod_galerie, -1*:OLD.valoare);
ELSIF UPDATING THEN
creste (:NEW.cod_galerie,      :NEW.valoare      -
:OLD.valoare);
ELSE /* inserting */
creste (:NEW.cod_galerie, :NEW.valoare);
END IF;
END;
/

```

Declansatori *INSTEAD OF*

PL/SQL permite definirea unui nou tip de declansator, numit *INSTEAD OF*, care ofera o modalitate de actualizare a vizualizarilor obiect si a celor relationale.

Sintaxa acestui tip de declansator este similara celei pentru declansatori *LMD*, cu doua exceptii:

- clauza *{BEFORE | AFTER}* este inlocuita prin *INSTEAD OF*;
- clauza *ON [schema.]nume_tabel* este inlocuita printr-una din clauzele *ON [schema.]numeview sau ON NESTED TABLE (numecoloana) OF [schema.]nume_view*.

Declansatorul *INSTEAD OF* permite reactualizarea unei vizualizari prin comenzi *LMD*. O astfel de modificare nu poate fi realizata in alta maniera, din cauza regulilor stricte existente pentru reactualizarea vizualizarilor. Declansatorii de tip *INSTEAD OF* sunt necesari, deoarece vizualizarea pe care este definit declansatorul poate, de exemplu, sa se refere la *join-ul* unor tabele, si in acest caz, nu sunt actualizabile toate legaturile.

O vizualizare nu poate fi modificata prin comenzi *LMD* daca vizualizarea contine operatori pe multimi, functii grup, clauzele *GROUP BY, CONNECT BY, START WITH*, operatorul *DISTINCT* sau join-uri.

Declansatorul *INSTEAD OF* este utilizat pentru a executa operatii *LMD* direct pe tabelele de baza ale vizualizarii. De fapt, se scriu comenzi *LMD* relative la o vizualizare, iar declansatorul, in locul operatiei originale, va opera pe tabelele de baza.

De asemenea, acest tip de declansator poate fi definit asupra vizualizelor ce au drept campuri tablouri imbriicate, declansatorul furnizand o modalitate de reactualizare a elementelor tabloului imbricat.

In acest caz, el se declanseaza doar in cazul in care comenziile *LMD* opereaza asupra tabloului imbricat (numai cand elementele tabloului imbricat sunt modificate folosind clauzele *THE()* sau *TABLE()*) si nu atunci cand comanda *LMD* opereaza doar asupra vizualizarii. Declansatorul permite accesarea liniei „parinte“ ce contine tabloul imbricat modificat.

Observatii:

- Spre deosebire de declansatorii *BEFORE* sau *AFTER*, declansatorii *INSTEAD OF* se executa in locul instructiunii *LMD* (*INSERT*, *UPDATE*, *DELETE*) specificate.
 - Optiunea *UPDATE OF* nu este permisa pentru acest tip de declansator.
 - Declansatorii *INSTEAD OF* se definesc pentru o vizualizare, nu pentru un tabel.
 - Declansatorii *INSTEAD OF* actioneaza implicit la nivel de linie.
 - Daca declansatorul este definit pentru tablouri imbricate, atributele *:OLD* si *:NEW* se refera la liniile tabloului imbricat, iar pentru a referi linia curenta din tabloul „parinte“ s-a introdus atributul *:PARENT*.

Exemplu:

Se considera *nouopera*, respectiv *nouartist*, copii ale tabelelor *opera*, respectiv *artist* si *viopar* o vizualizare definita prin compunerea naturala a celor doua tabele. Se presupune ca pentru fiecare artist exista un camp (*sum val*) ce reprezinta valoarea totala a operelor de arta expuse de acesta in muzeu.

Sa se defineasca un declansator prin care reactualizarile executate asupra vizualizarii *vi op ar* se vor transmite automat tabelelor *nou opera* si *nouartist*.

```

CREATE TABLE nou_opera AS
SELECT cod_opera, cod_artist, valoare, tip, stil FROM
opera;
CREATE TABLE nou_artist AS
SELECT cod_artist, nume, sum_val FROM artist;
CREATE VIEW vi_op_ar AS
SELECT cod_opera,o.cod_artist,valoare,tip,nume, sum
val
FROM     opera o, artist a
WHERE o.cod artist = a.cod artist
  
```

```

CREATE OR REPLACE TRIGGER react
INSTEAD OF INSERT OR DELETE OR UPDATE ON vi_op_ar
FOR EACH ROW
BEGIN
IF INSERTING THEN
INSERT INTO nou_opera
VALUES (:NEW.cod_opera, :NEW.cod_artist, :NEW.valoare,
:NEW.tip);
UPDATE nou_artist
SET sum_val = sum_val + :NEW.valoare
WHERE cod_artist = :NEW.cod_artist;
ELSIF DELETING THEN
DELETE FROM nou_opera
WHERE cod_opera = :OLD.cod_opera;
UPDATE nou_artist
SET sum_val = sum_val - :OLD.valoare
WHERE cod_artist = :OLD.cod_artist;
ELSIF UPDATING ('valoare') THEN UPDATE nou_opera SET
valoare = :NEW.valoare WHERE cod_opera = :OLD.cod_opera;
UPDATE nou_artist
SET sum_val = sum_val + (:NEW.valoare -
:OLD.valoare)
WHERE cod_artist = :OLD.cod_artist;
ELSIF UPDATING ('cod_artist') THEN
    UPDATE nou_opera
    SET cod_artist = :NEW.cod_artist
    WHERE cod_opera = :OLD.cod_opera;
    UPDATE nou_artist
    SET sum_val = sum_val - :OLD.valoare
    WHERE cod_artist = :OLD.cod_artist;
    UPDATE nou_artist
    SET sum_val = sum_val + :NEW.valoare
    WHERE cod_artist = :NEW.cod_artist;
END IF;
END;
/

```

Declansatori sistem

Declansatorii sistem sunt activati de comenzi *LDD* (*CREATE*, *DROP*, *ALTER*) si de anumite evenimente sistem (*STARTUP*, *SHUTDOWN*, *LOGON*, *LOGOFF*, *SERVERERROR*, *SUSPEND*). Un declansator sistem poate fi definit la nivelul bazei de date sau la nivelul schemei.

Sintaxa pentru crearea unui astfel de declansator este urmatoarea:

```
CREATE [OR REPLACE] TRIGGER [schema.]numeddeclansator {BEFORE | AFTER}
{listaevenimenteLDD | lista evenimente baza}
ON {DATABASE / SCHEMA}
[WHEN(conditie) ] corpdeclansator;
```

Cuvintele cheie *DATABASE* sau *SCHEMA* specifica nivelul declansatorului. Exista restrictii asupra expresiilor din conditia clauzei *WHEN*. De exemplu, declansatorii *LOGON* si *LOGOFF* pot verifica doar identificatorul (*userid*) si numele utilizatorului (*username*), iar declansatorii *LDD* pot verifica tipul si numele obiectelor definite, identificatorul si numele utilizatorului.

Evenimentele amintite anterior pot fi asociate clauzelor *BEFORE* sau *AFTER*. De exemplu, un declansator *LOGON* (*AFTER*) se activeaza dupa ce un utilizator s-a conectat la baza de date, un declansator *CREATE* (*BEFORE* sau *AFTER*) se activeaza inainte sau dupa ce a fost creat un obiect al bazei, un declansator *SERVERERROR* (*AFTER*) se activeaza ori de cate ori apare o eroare (cu exceptia erorilor: *ORA-01403*, *ORA-01422*, *ORA-01423*, *ORA-01034*, *ORA-04030*).

Pentru declansatorii sistem se pot utiliza functii speciale care permit obtinerea de informatii referitoare la evenimentul declansator. Ele sunt functii *PL/SQL* stocate care trebuie prefixate de numele proprietarului (*SYS*).

Printre cele mai importante functii care furnizeaza informatii referitoare la evenimentul declansator, se remarcă:

- *SYSEVENT* - returneaza evenimentul sistem care a activat declansatorul (este de tip *VARCHAR2(20)* si este aplicabila oricarui eveniment);
- *DATABASE NAME* - returneaza numele bazei de date curente (este de tip *VARCHAR2(50)* si este aplicabila oricarui eveniment);
- *LOGIN USER* - returneaza identificatorul utilizatorului care activeaza declansatorul (este de tip *VARCHAR2(30)* si este aplicabila oricarui eveniment);

Exemplu:

```
CREATE OR REPLACE TRIGGER logutiliz AFTER CREATE ON
SCHEMA BEGIN
INSERT INTO ldd_tab(user_id, object_name, creation_date)
VALUES      (USER, SYS.DICTIONARY_OBJ_NAME, SYSDATE);
END logutiliz;
```

Evenimentul *SERVERRERROR* poate fi utilizat pentru a urmari erorile care apar in baza de date. Codul erorii este furnizat, prin intermediul declaratorului, de functia *SERVER ERROR*, iar mesajul asociat erorii poate fi obtinut cu procedura *DBMS_UTILITY.FORMA_T_ERROR_STACK*.

Exemplu:

```
CREATE TABLE erori ( moment      DATE,
utilizator VARCHAR2(30), nume_baza    VARCHAR2(50),
stiva_erori VARCHAR2(2000) );
/
CREATE OR REPLACE TRIGGER logerori AFTER SERVERRERROR ON
DATABASE
BEGIN
INSERT INTO erori
VALUES (SYSDATE, SYS.LOGIN_USER, SYS.DATABASE_NAME,
DBMS_UTILITY.FORMAT_ERROR_STACK);
END logerori;
/
```

Modificarea si suprimarea declansatorilor

Optiunea *OR REPLACE* din cadrul comenzii *CREATE TRIGGER* recreaza declansatorul, daca acesta exista. Clauza permite schimbarea definitiei unui declansator existent fara suprimarea acestuia.

Similar procedurilor si pachetelor, un declansator poate fi suprimat prin:

DROP TRIGGER [schema.]nume_declansator;

Uneori actiunea de suprimare a unui declansator este prea drastica si este preferabila doar dezactivarea sa temporara. In acest caz, declansatorul va continua sa existe in dictionarul datelor.

Modificarea unui declansator poate consta din recompilarea (*COMPILE*),

redenumirea (*RENAME*), activarea (*ENABLE*) sau dezactivarea (*DISABLE*) acestuia si se realizeaza prin comanda:

```
ALTER TRIGGER [schema.]nume declansator {ENABLE | DISABLE /  
COMPILE | RENAME TO nume_nou}  
{ALL TRIGGERS}
```

Daca un declansator este activat, atunci sistemul *Oracle* il executa ori de cate ori au loc operatiile precizate in declansator asupra tabelului asociat si cand conditia de restrictie este indeplinita. Daca declansatorul este dezactivat, atunci sistemul *Oracle* nu il va mai executa. Dupa cum s-a mai subliniat, dezactivarea unui declansator nu implica stergerea acestuia din dictionarul datelor.

Totii declansatorii asociati unui tabel pot fi activati sau dezactivati utilizand optiunea *ALL TRIGGERS* (*ENABLE ALL TRIGGERS*, respectiv *DISABLE ALL TRIGGERS*). Declansatorii sunt activati in mod implicit atunci cand sunt creati.

Pentru activarea (*enable*) unui declansator, server-ul *Oracle*:

- verifica integritatea constrangerilor,
- garanteaza ca declansatorii nu pot compromite constrangerile de integritate,
- garanteaza consistenta la citire a vizualizarilor,
- gestioneaza dependentele.

Activarea si dezactivarea declansatorilor asociati unui tabel se poate realiza si cu ajutorul comenzii *ALTER TABLE*.

Un declansator este compilat in mod automat la creare. Daca un *site* este neutilizabil atunci cand declansatorul trebuie compilat, sistemul *Oracle* nu poate valida comanda de accesare a bazei distante si compilarea esueaza.

Informatii despre declansatori

In DD exista vizualizari ce contin informatii despre declansatori si despre starea acestora (*USER_TRIGGER*, *USER_TRIGGER_COL*, *ALL_TRIGGER*, *DBATRIGGER* etc.). Aceste vizualizari sunt actualizate ori de cate ori un declansator este creat sau suprimat.

Atunci cand declansatorul este creat, codul sau sursa este stocat in vizualizarea *USERTRIGGERS*. Vizualizarea *ALLTRIGGERS* contine informatii despre toti declansatorii din baza de date. Pentru a detecta dependentele declansatorilor poate fi consultata vizualizarea *USER_DEPENDENCIES*, iar *ALLDEPENDENCIES* contine informatii despre dependentele tuturor obiectelor din baza de date. Erorile rezultate din compilarea declansatorilor pot fi analizate din vizualizarea *USERERRORS*, iar prin comanda *SHOW ERRORS* se vor afisa erorile corespunzatoare ultimului declansator compilat.

In operatiile de gestiune a bazei de date este necesara uneori reconstruirea instructiunilor *CREATE TRIGGER*, atunci cand codul sursa original nu mai este disponibil. Aceasta se poate realiza utilizand vizualizarea *USERTRIGGERS*.

Vizualizarea include numele declansatorului (*TRIGGERNAME*), tipul acestuia (*TRIGGERTYPE*), evenimentul declansator (*TRIGGERINGEVENT*), numele proprietarului tabelului (*TABLEOWNER*), numele tabelului pe care este definit declansatorul (*TABLENAME*), clauza *WHEN* (*WHENCLAUSE*), corpul declansatorului (*TRIGGERBODY*), antetul (*DESCRIPTION*), starea acestuia (*STATUS*) care poate sa fie *ENABLED* sau *DISABLED* si numele utilizate pentru a referi parametrii *OLD* si *NEW* (*REFERENCINGNAMES*). Daca obiectul de baza nu este un tabel sau o vizualizare, atunci *TABLENAME* este *null*.

Exemplu:

Presupunand ca nu este disponibil codul sursa pentru declansatorul *alfa*, sa se reconstruiasca instructiunea *CREATE TRIGGER* corespunzatoare acestuia.

```
SELECT 'CREATE OR REPLACE TRIGGER ' || DESCRIPTION ||  
TRIGGER_BODY FROM USER_TRIGGERS  
WHERE TRIGGER_NAME = 'ALFA';
```

Cu aceasta interogare se pot reconstrui numai declansatorii care apartin contului utilizator curent. O interogare a vizualizarilor *ALL TRIGGERS* sau *DBATRIGGERS* permite reconstruirea tuturor declansatorilor din sistem, daca se dispune de privilegii *DBA*.

Exemplu:

```
SELECT USERNAME FROM USER_USERS;
```

Aceasta cerere furnizeaza numele "proprietarului" (creatorului) declansatorului si nu numele utilizatorului care a reactualizat tabelul.

Privilegii sistem

Sistemul furnizeaza privilegii sistem pentru gestiunea declansatorilor:

- *CREATE TRIGGER* (permite crearea declansatorilor in schema personala);
- *CREATE ANY TRIGGER* (permite crearea declansatorilor in orice schema cu exceptia celei corespunzatoare lui *SYS*);
- *ALTER ANY TRIGGER* (permite activarea, dezactivarea sau compilarea declansatorilor in orice schema cu exceptia lui *SYS*);
- *DROP ANY TRIGGER* (permite suprimarea declansatorilor la nivel de baza de date in orice schema cu exceptia celei corespunzatoare lui *SYS*);
- *ADMINISTER DATABASE TRIGGER* (permite crearea sau modificarea unui declansator sistem referitor la baza de date);
- *EXECUTE* (permite referirea, in corpul declansatorului, a procedurilor, functiilor sau pachetelor din alte scheme).

Tabele *mutating*

Asupra tabelelor si coloanelor care pot fi accesate de corpul declansatorului exista anumite restrictii. Pentru a analiza aceste restrictii este necesara definirea tabelelor in schimbaare (*mutating*) si constranse (*constraining*).

Un tabel *constraining* este un tabel pe care evenimentul declansator trebuie sa-l consulte fie direct, printr-o instructiune *SQL*, fie indirect, printr-o constrangere de integritate referentiala declarata. Tabelele nu sunt considerate *constraining* in cazul declansatorilor la nivel de instructiune. Comenzile *SQL* din corpul unui declansator nu pot modifica valorile coloanelor care sunt declarate chei primare, externe sau unice (*PRIMARY KEY*, *FOREIGN KEY*, *UNIQUE KEY*) intr-un tabel *constraining*.

Un tabel *mutating* este tabelul modificat de instructiunea *UPDATE*, *DELETE* sau *INSERT*, sau un tabel care va fi actualizat prin efectele actiunii integritatii referentiale *ON DELETE CASCADE*. Chiar tabelul pe care este definit declansatorul este un tabel *mutating*, ca si orice tabel referit printr-o constrangere *FOREING KEY*.

Tabelele nu sunt considerate *mutating* pentru declansatorii la nivel de instructiune, cu exceptia celor declansati ca efect al optiunii *ON DELETE CASCADE*. Vizualizarile nu sunt considerate *mutating* in declansatorii *INSTEAD OF*.

Regula care trebuie respectata la utilizarea declansatorilor este: **comenzile *SQL* din corpul unui declansator nu pot consulta sau modifica date dintr-un tabel *mutating*.**

Exceptia! Daca o comanda *INSERT* afecteaza numai o inregistrare, declansatorii la nivel de linie (*BEFORE* sau *AFTER*) pentru inregistrarea respectiva nu trateaza tabelul ca fiind *mutating*. Acesta este unicul caz in care un declansator la nivel de linie poate citi sau modifica tabelul. Comanda *INSERT INTO tabel SELECT ...* considera tabelul *mutating* chiar daca cererea returneaza o singura linie.

Exemplu:

17

```

CREATE OR REPLACE TRIGGER cascada
AFTER UPDATE OF cod_artist ON artist FOR EACH ROW
BEGIN
    UPDATE opera
    SET     opera.cod_artist= :NEW.cod_artist
    WHERE   opera.cod_artist= :OLD.cod_artist
END;

UPDATE artist
SET     cod_artist = 71
WHERE   cod_artist = 23;

```

La executia acestei sevante este semnalata o eroare. Tabelul *artist* referentiaza tabelul *opera* printr-o constrangere de cheie externa. Prin urmare, tabelul *opera* este *constraining*, iar declansatorul *cascada* incerca sa schimbe date in tabelul *constraining*, ceea ce nu este permis. Exemplul va functiona corect daca nu este definita sau activata constrangerea referentiala intre cele doua tabele.

Exemplu:

Sa se implementeze cu ajutorul unui declansator restrictia ca intr-o sala pot sa fie expuse maximum 10 opere de arta.

```

CREATE OR REPLACE TRIGGER TrLimitaopere
BEFORE INSERT OR UPDATE OF cod_sala ON opera FOR EACH
ROW DECLARE
    v_Max_opere CONSTANT NUMBER := 10;
    v_opere_curente    NUMBER;
BEGIN
    SELECT COUNT(*) INTO v_opere_curente FROM opera
    WHERE cod_sala = :NEW.cod_sala;
    IF v_opere_curente + 1 > v_Max_opere THEN
        RAISE_APPLICATION_ERROR(-20000,'Prea multe opere de
arta in sala avand codul ' || :NEW.cod_sala);
    END IF;
END TrLimitaopere;

```

Cu toate ca declansatorul pare sa produca lucrul dorit, totusi dupa o reactualizare a tabelului *opera* in urmatoarea maniera:

```

INSERT INTO opera (cod_opera, cod_sala)
VALUES (756893, 10);

```

se obtine urmatorul mesaj de eroare:

```

ORA-04091: tabel opera is mutating,
trigger/function may not see it
ORA-04088: error during execution of trigger

```

Eroarea *ORA-04091* apare deoarece declansatorul *TrLimitaopere* consulta chiar tabelul (*opera*) la care este asociat declansatorul (*mutating*).

Tabelul *opera* este *mutating* doar pentru un declansator la nivel de linie. Aceasta inseamna ca tabelul poate fi consultat in interiorul unui declansator la nivel de instructiune. Totusi, limitarea numarului operelor de arta nu poate fi facuta in interiorul unui declansator la nivel de instructiune, din moment ce este necesara valoarea *:NEW.cod_sala* in corpul declansatorului.

```

CREATE OR REPLACE PACKAGE PSalaDate AS
TYPE t_cod_sala IS TABLE OF opera.cod_sala%TYPE -- INDEX
BY BINARY_INTEGER;-
TYPE t_cod_opera IS TABLE OF opera.cod_opera%TYPE -- INDEX
BY BINARY_INTEGER;-
v_cod_sala      t_cod_sala;
v_cod_opera     t_cod_opera;
v_NrIntrari BINARY_INTEGER := 0;
END PSalaDate;

CREATE OR REPLACE TRIGGER TrLLimitaSala BEFORE INSERT OR
UPDATE OF cod_sala ON opera FOR EACH ROW BEGIN
PSalaDate.v_NrIntrari := PSalaDate.v_NrIntrari + 1;
PSalaDate.v_cod_sala(PSalaDate.v_NrIntrari) :=
:NEW.cod_sala;
PSalaDate.v_cod_opera(PSalaDate.v_NrIntrari) :=
:NEW.cod_opera;
END TrLLimitasala;

CREATE OR REPLACE TRIGGER TrILimitaopere
AFTER INSERT OR UPDATE OF cod_sala ON opera
DECLARE
    v_Max_opere v CONSTANT NUMBER := 10;
    opere_curente v NUMBER;
    cod_operax    opera.cod_opera%TYPE;
    v_cod_salax BEGIN opera.cod_sala%TYPE;
        FOR v_LoopIndex IN 1..PsalaDate.v_NrIntrari LOOP
            v_cod_operax := PsalaDate.v_cod_opera(v_LoopIndex);
            v_cod_salax := PsalaDate.v_cod_sala(v_LoopIndex);
            SELECT COUNT(*) - - -
            INTO v_opere_curente
            FROM opera
            WHERE cod_sala = v_cod_salax;
            IF v_opere_curente > v_Max_opere THEN
                RAISE_APPLICATION_ERROR(-20000, 'Prea multe opere de
                arta in sala' || v_cod_salax || 'din cauza inserarii
                operei avand codul' || v_cod_operax)
        END LOOP;
    END;

```

```

    END IF;
    END LOOP;
    /* Reseteaza contorul deoarece urmatoarea executie va
folosi date noi */
    PSalaDate.v_NrIntrari := 0;
END TrILimitaopere;

```

O solutie pentru acesta problema este crearea a doi declansatori, unul la nivel de linie si altul la nivel de instructiune. In declansatorul la nivel de linie se inregistreaza valoarea lui :*NEW.cod_opera*, dar nu va fi interrogat tabelul *opera*.

Interrogarea va fi facuta in declansatorul la nivel de instructiune si va folosi valoarea inregistrata in declansatorul la nivel de linie.

O modalitate pentru a inregistra valoarea lui :*NEW.cod_opera* este utilizarea unui tablou indexat in interiorul unui pachet.

Exemplu:

Sa se creeze un declansator care:

- a) daca este eliminata o sala, va sterge toate operele expuse in sala respectiva;
- b) daca se schimba codul unei sali, va modifica aceasta valoare pentru fiecare opera de arta expusa in sala respectiva.

```

CREATE OR REPLACE TRIGGER sala_cascada
BEFORE DELETE OR UPDATE OF cod_sala ON sala FOR EACH
ROW BEGIN
    IF DELETING THEN DELETE FROM opera WHERE      cod_sala =
:OLD.cod_sala;
    END IF;
    IF UPDATING AND :OLD.cod_sala != :NEW.cod_sala THEN
UPDATE opera
    SET      cod_sala = :NEW.cod_sala
    WHERE      cod_sala = :OLD.cod_sala;
    END IF;
END sala_cascada;

```

Declansatorul anterior realizeaza constrangerea de integritate *UPDATE* sau *ON DELETE CASCADE*, adica stergerea sau modificarea cheii primare a unui tabel „parinte“ se va reflecta si asupra inregistrarilor corespunzatoare din tabelul „copil“.

Executarea acestuia, pe tabelul *sala* (tabelul „parinte“), va duce la efectuarea a doua tipuri de operatii pe tabelul *opera* (tabelul „copil“).

La eliminarea unei sali din tabelul *sala*, se vor sterge toate operele de arta corespunzatoare acestei sali.

```
DELETE FROM sala WHERE cod_sala = 773;
```

La modificarea codului unei sali din tabelul *sala*, se va actualiza codul salii atat in tabelul *sala*, cat si in tabelul *opera*.

```
UPDATE sala
SET cod_sala = 777
WHERE cod_sala = 333;
```

Se presupune ca asupra tabelului *opera* exista o constrangere de integritate:

```
FOREIGN KEY (cod_sala) REFERENCES sala(cod_sala)
```

In acest caz sistemul *Oracle* va afisa un mesaj de eroare prin care se precizeaza ca tabelul *sala* este *mutating*, iar constrangerea definita mai sus nu poate fi verificata.

*ORA-04091: table MASTER.SALA is mutating,
trigger/function may not see it*

Pachetele pot fi folosite pentru incapsularea detaliilor logice legate de declansatori. Exemplul urmator arata un mod simplu de implementare a acestei posibilitati. Este permisa apelarea unei proceduri sau functii stocate din blocul *PL/SQL* care reprezinta corpul declansatorului.

Exemplu:

```
CREATE OR REPLACE PACKAGE pachet IS
PROCEDURE procesare_trigger(pvaloare IN NUMBER,
- pstare IN VARCHAR2);
END pachet;
CREATE OR REPLACE PACKAGE BODY pachet IS
PROCEDURE procesare_trigger(pvaloare IN NUMBER,
- pstare IN VARCHAR2) IS
BEGIN
END procesare_trigger;
END pachet;
CREATE OR REPLACE TRIGGER gama AFTER INSERT ON opera FOR
EACH ROW BEGIN
pachet.procesare_trigger(:NEW.valoare,:NEW.stare)
END;
```

Tratarea erorilor

Tratarea erorilor

Mecanismul de gestiune a erorilor permite utilizatorului sa defineasca si sa controleze comportamentul programului atunci cand acesta genereaza o eroare. In acest fel, aplicatia nu este oprita, revenind intr-un regim normal de executie.

Intr-un program **PL/SQL** pot sa apara erori la compilare sau erori la executie.

Erorile care apar in timpul compilarii sunt detectate de motorul **PL/SQL** si sunt comunicate programatorului care va face corectia acestora. Programul nu poate trata aceste erori deoarece nu a fost inca executat.

Erorile care apar in timpul executiei nu mai sunt tratate interactiv. In program trebuie prevazuta aparitia unei astfel de erori si specificat modul concret de tratare a acesteia. Atunci cand apare eroarea este declansata o exceptie, iar controlul trece la o sectiune separata a programului, unde va avea loc tratarea erorii.

Gestiunea erorilor in **PL/SQL** face referire la conceptul de exceptie. Exceptia este un eveniment particular (eroare sau avertisment) generat de **serverul Oracle** sau de aplicatie, care necesita o tratare speciala. In **PL/SQL** mecanismul de tratare a exceptiilor permite programului sa isi continue executia si in prezenta anumitor erori.

Exceptiile pot fi definite, activate, tratate la nivelul fiecarui bloc din program (program principal, functii si proceduri, blocuri interioare acestora). Executia unui bloc se termina intotdeauna atunci cand apare o exceptie, dar se pot executa actiuni ulterioare aparitiei acesteia, intr-o sectiune speciala de tratare a exceptiilor.

Posibilitatea de a da nume fiecarei exceptii, de a izola tratarea erorilor intr-o sectiune particulara, de a declansa automat erori (in cazul exceptiilor interne) imbunatasteste lizibilitatea si fiabilitatea programului. Prin utilizarea exceptiilor si rutinelor de tratare a exceptiilor, un program **PL/SQL** devine robust si capabil sa trateze atat erorile asteptate, cat si cele neasteptate ce pot aparea in timpul executiei.

Sectiunea de tratare a erorilor

Pentru a gestiona exceptiile, utilizatorul trebuie sa scrie cateva comenzi care preiau controlul derularii blocului **PL/SQL**. Aceste comenzi sunt situate in sectiunea de tratare a erorilor dintr-un bloc **PL/SQL** si sunt cuprinse intre

cuvintele cheie *EXCEPTION* si *END*, conform urmatoarei sintaxe generale:

EXCEPTION

WHEN nume_exceptie1 [OR nume_exceptie2 ...] THEN secentadeinstructiunil;
[WHEN nume_exceptie3 [OR nume_exceptie4 ...] THEN
secenta_de_instructiuni_2;]

[WHEN OTHERS THEN

secentadeinstructiunin;]

END;

De remarcat ca *WHEN OTHERS* trebuie sa fie ultima clauza si trebuie sa fie unica. Toate exceptiile care nu au fost analizate vor fi tratate prin aceasta clauza. Evident, in practica nu se utilizeaza forma *WHEN OTHERS THEN NULL*.

In *PL/SQL* exista doua tipuri de exceptii:

- exceptii interne, care se produc atunci cand un bloc *PL/SQL* nu respecta o regula *Oracle* sau depaseste o limita a sistemului de operare;
- exceptii externe definite de utilizator (*user-defined error*), care sunt declarate in sectiunea declarativa a unui bloc, subprogram sau pachet si care sunt activate explicit in partea executabila a blocului *PL/SQL*.

Exceptiile interne *PL/SQL* sunt de doua tipuri:

- exceptii interne predefinite (*predefined Oracle Server error*);
- exceptii interne nepredefinite (*non-predefined Oracle Server error*).

Functii pentru identificarea exceptiilor

Indiferent de tipul exceptiei, aceasta are asociate doua elemente:

- un cod care o identifica;
- un mesaj cu ajutorul caruia se poate interpreta exceptia respectiva.

Cu ajutorul functiilor *SQLCODE* si *SQLERRM* se pot obtine codul si mesajul asociate exceptiei declansate. Lungimea maxima a mesajului este de 512 caractere.

De exemplu, pentru eroarea predefinita *ZERODIVIDE*, codul *SQLCODE* asociat este -1476, iar mesajul corespunzator erorii, furnizat de *SQLERRM*, este „divide by zero error“.

Codul erorii este:

- un numar negativ, in cazul unei erori sistem;
- numarul +100, in cazul exceptiei *NODATAFOUND*;
- numarul 0, in cazul unei executii normale (fara exceptii);

- numarul 1, in cazul unei exceptii definite de utilizator.

Functiile *SQLCODE* si *SQLERRM* nu se pot utiliza direct ca parte a unei instructiuni *SQL*. Valorile acestora trebuie atribuite unor variabile locale.

Rezultatul functiei *SQLCODE* poate fi asignat unei variabile de tip numeric, iar cel al functiei *SQLERRM* unei variabile de tip caracter. Variabilele locale astfel definite pot fi utilizate in comenzi *SQL*.

Exemplu:

Sa se scrie un bloc *PL/SQL* prin care sa se exemplifice situatia comentata.

```
DECLARE
    eroare_cod      NUMBER;
    eroare_mesaj    VARCHAR2(100);
BEGIN
    EXCEPTION
        WHEN OTHERS THEN
            eroare_cod := SQLCODE;
            eroare_mesaj := SUBSTR(SQLERRM,1,100);
            INSERT INTO erori
            VALUES (eroare_cod, eroare_mesaj);
END;
```

Mesajul asociat exceptiei declansate poate fi furnizat si de functia *DBMS_UTILITY.FORMA_TERRORSTACK*.

Exceptii interne

Exceptiile interne se produc atunci cand un bloc *PL/SQL* nu respecta o regula *Oracle* sau depaseste o limita a sistemului de exploatare.

Aceste exceptii pot fi independente de structura bazei de date sau pot sa apară datorita nerespectarii constrangerilor statice implementate in structura (*PRIMARY KEY*, *FOREIGN KEY*, *NOT NULL*, *UNIQUE*, *CHECK*).

Atunci cand apare o eroare *Oracle*, exceptia asociata ei se declanseaza implicit. De exemplu, daca apare eroarea *ORA-01403* (deoarece o comanda *SELECT* nu returneaza nici o linie), atunci implicit *PL/SQL* activeaza exceptia *NODATAFOUND*. Cu toate ca fiecare astfel de exceptie are asociat un cod specific, ele trebuie referite prin nume.

Exceptii interne predefinite

Exceptiile interne predefinite nu trebuie declarate in sectiunea declarativa si sunt tratate implicit de catre *server-ul Oracle*. Ele sunt referite prin nume

(**CURSORALREADY_OPEN**, **DUP_VAL_ON_INDEX**, **NO_DATAFOUND** etc.). **PL/SQL** declara aceste exceptii in pachetul **DBMSSTANDARD**.

Nume exceptie	Cod eroare	Descriere
ACCESINTONULL	ORA-06530	Asignare de valori atributelor unui obiect neinitializat.
CASENOTFOUND	ORA-06592	Nu este selectata nici una din clauzele <i>WHEN</i> ale lui <i>CASE</i> si nu exista nici clauza <i>ELSE</i> (exceptie specifica lui <i>Oracle9i</i>).
COLLECTIONISNULL	ORA-06531	Aplicarea unei metode (diferite de <i>EXISTS</i>) unui tabel imbricat sau unui vector neinitializat.
CURSORALREADYOPEN	ORA-06511	Deschiderea unui cursor care este deja deschis.
DUPVALONINDEX	ORA-00001	Detectarea unei dubluri intr-o coloana unde acestea sunt interzise.
INVALID_CURSOR	ORA-01001	Operatie ilegală asupra unui cursor.
INVALIDNUMBER	ORA-01722	Conversie nepermisa de la tipul sir de caractere la numar.
LOGIN_DENIED	ORA-01017	Nume sau parola incorecte.
NODATAFOUND	ORA-01403	Comanda <i>SELECT</i> nu returneaza nici o inregistrare.
NOTLOGGEDON	ORA-01012	Programul <i>PL/SQL</i> apeleaza baza fara sa fie conectat la <i>Oracle</i> .
SELFISNULL	ORA-30625	Apelul unei metode cand instanta este <i>NULL</i> .
PROGRAM_ERROR	ORA-06501	<i>PL/SQL</i> are o problema interna.
ROWTYPEISMATCH	ORA-06504	Incompatibilitate intre parametrii actuali si formali, la deschiderea unui cursor parametrizat.
STORAGEERROR	ORA-06500	<i>PL/SQL</i> are probleme cu spatiul de memorie.
SUBSCRIPTBEYONDCONSTANT	ORA-06533	Referire la o componenta a unui <i>nested table</i> sau <i>varray</i> , folosind un index mai mare decat numarul elementelor colectiei respective.
SUBSCRIPTOUTSIDELIMIT	ORA-06532	Referire la o componenta a unui tabel imbricat sau vector, folosind un index care este in afara domeniului (de exemplu, -1).
SYSINVALIDROWID	ORA-01410	Conversia unui sir de caractere intr-un <i>ROWID</i> nu se poate face deoarece sirul nu reprezinta un <i>ROWID</i> valid.
TIMEOUTONRESOURCE	ORA-00051	Expirarea timpului de asteptare pentru eliberarea unei resurse.
TRANSACTIONBACKEDOUT	ORA-00061	Tranzactia a fost anulata datorita unei interblocari.
TOOMANYROWS	ORA-01422	<i>SELECT INTO</i> intoarce mai multe linii.

VALUEERROR	ORA-06502	Aparitia unor erori in conversii, constrangeri sau erori aritmetice.
ZERO DIVIDE	ORA-01476	Sesizarea unei impartiri la zero.

Exemplu:

Sa se scrie un bloc **PL/SQL** prin care sa se afiseze numele artistilor de o anumita nationalitate care au opere de arta expuse in muzeu.

- 1) Daca rezultatul interogarii returneaza mai mult decat o linie, atunci sa se trateze exceptia si sa se insereze in tabelul **mesaje** textul „mai multi creatori“.
- 2) Daca rezultatul interogarii nu returneaza nici o linie, atunci sa se trateze exceptia si sa se insereze in tabelul **mesaje** textul „nici un creator“.
- 3) Daca rezultatul interogarii este o singura linie, atunci sa se insereze in tabelul **mesaje** numele artistului si pseudonimul acestuia.
- 4) Sa se trateze orice alta eroare, inserand in tabelul **mesaje** textul „alte erori au aparut“.

```

SET VERIFY OFF
ACCEPT national      PROMPT 'Introduceti nationalitatea:'
DECLARE
    v_nume_artist      artist.nume%TYPE;
    v_pseudonim        artist.pseudonim%TYPE;
    v_national         artist.national%TYPE:='&national';
BEGIN
    SELECT nume,pseudonim
        INTO v_nume_artist,v_pseudonim
    FROM artist
    WHERE national=v_national;
    INSERT INTO mesaje(rezultate)
    VALUES (v_nume_artist||'-'||v_pseudonim);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        INSERT INTO mesaje      (rezultate)
        VALUES ('nici un creator');
    WHEN TOO_MANY_ROWS THEN
        INSERT INTO mesaje      (rezultate)
        VALUES ('mai multi creatori');
    WHEN OTHERS THEN
        INSERT INTO mesaje      (rezultate)
        VALUES ('alte erori au aparut');
END;
/
SET VERIFY ON

```

Aceeași exceptie poate să apară în diferite circumstanțe. De exemplu, exceptia **NODATAFOUND** poate fi generată fie pentru că o interogare nu întoarce un rezultat, fie pentru că se referă un element al unui tablou **PL/SQL**.

care nu a fost definit (nu are atribuita o valoare). Daca intr-un bloc *PL/SQL* apar ambele situatii, este greu de stabilit care dintre ele a generat eroarea si este necesara restructurarea blocului, astfel incat acesta sa poata diferentia cele doua situatii.

Exceptii interne nepredefinite

Exceptiile interne nepredefinite sunt declarate in sectiunea declarativa si sunt tratate implicit de catre *server-ul Oracle*. Ele pot fi gestionate prin clauza *OTHERS*, in sectiunea *EXCEPTION*.

Diferentierea acestor erori este posibila doar cu ajutorul codului. Dupa cum s-a mai specificat, codul unei exceptii interne este un numar negativ, in afara de exceptia *NODATAFOUND*, care are codul +100.

O alta metoda pentru tratarea unei erori interne nepredefinite (diferita de folosirea clauzei *OTHERS* drept detector universal de exceptii) este utilizarea directivei de compilare (pseudo-instructiune) *PRAGMA EXCEPTIONINIT*. Aceasta directiva permite asocierea numelui unei exceptii cu un cod de eroare intern. In felul acesta, orice exceptie interna poate fi referita printr-un nume si se pot scrie rutine speciale pentru tratarea acesteia. Directiva este procesata in momentul compilarii, si nu la executie.

Directiva trebuie sa apara in partea declarativa a unui bloc, pachet sau subprogram, dupa definirea numelui exceptiei. *PRAGMA EXCEPTION INIT* poate sa apara de mai multe ori intr-un program. De asemenea, pot fi asignate mai multe nume pentru acelasi cod de eroare.

In acest caz, tratarea erorii se face in urmatoarea maniera:

- 1) se declara numele exceptiei in partea declarativa sub forma:

numeexceptie EXCEPTION;

- 2) se associaza numele exceptiei cu un cod eroare standard *Oracle*, utilizand comanda:

PRAGMA EXCEPTION_INIT (nume exceptie, coderoare);

- 3) se refera exceptia in sectiunea de gestiune a erorilor (exceptia este tratata automat, fara a fi necesara comanda *RAISE*).

Exemplu:

Daca exista opere de arta create de un anumit artist, sa se tipareasca un mesaj prin care utilizatorul este anuntat ca artistul respectiv nu poate fi sters din baza de date (violarea constrangerii de integritate avand codul eroare *Oracle -2292*).

```
SET VERIFY OFF DEFINE p_nume = Monet
```

```

DECLARE
  opera_exista EXCEPTION;
  PRAGMA EXCEPTION_INIT(opera_exista,-2292);
BEGIN
  DELETE FROM artist WHERE nume = '&p_nume';
  COMMIT;
EXCEPTION
  WHEN opera_exista THEN
    DBMS_OUTPUT.PUT_LINE ('nu puteti sterge artistul cu numele
      ' || '&p_nume' || ' deoarece exista in
      muzeu opere de arta create de acesta');
END;
/
SET VERIFY ON

```

Exceptii externe

PL/SQL permite utilizatorului sa defineasca propriile sale exceptii. Aceste exceptii pot sa apară in toate sectiunile unui bloc, subprogram sau pachet. Exceptiile externe sunt definite in partea declarativa a blocului, deci posibilitatea de referire la ele este asigurata. In mod implicit, toate exceptiile externe au asociat acelasi cod (+1) si acelasi mesaj (*USER DEFINED EXCEPTION*).

Tratarea unei astfel de erori se face intr-o maniera similara modului de tratare descris anterior. Activarea exceptiei externe este facuta explicit, folosind comanda *RAISE* insotita de numele exceptiei. Comanda opreste executia normala a blocului *PL/SQL* si transfera controlul „admnistratorului“ exceptiilor.

Declararea si prelucrarea exceptiilor externe respecta urmatoarea sintaxa:

DECLARE

numeexceptie EXCEPTION; -- declarare exceptie **BEGIN**

RAISE nume exceptie; --declansare exceptie -- codul care urmeaza nu mai este executat

EXCEPTION

WHEN nume exceptie THEN -- definire mod de tratare a erorii

END;

Exceptiile trebuie private ca niste variabile, in sensul ca ele sunt active in sectiunea in care sunt declarate. Ele nu pot sa apară in instructiuni de atribuire sau in comenzi *SQL*.

Este recomandat ca fiecare subprogram sa aiba definita o zona de tratare a exceptiilor. Daca pe parcursul executiei programului intervine o eroare, atunci acesta genereaza o exceptie si controlul se transfera blocului de tratare a erorilor.

Exemplu:

Sa se scrie un bloc ***PL/SQL*** care afiseaza numarul creatorilor operelor de arta din muzeu care au valoarea mai mare sau mai mica cu 100000\$ decat o valoare specificata. Sa se tipareasca un mesaj adevarat, daca nu exista nici un artist care indeplineste aceasta conditie.

```
VARIABLE g_mesaj VARCHAR2(100)
SET VERIFY OFF
ACCEPT p_val PROMPT 'va rog specificati valoarea:'
DECLARE
    v_val          opera.valoare%TYPE := &p_val;
    v_inf          opera.valoare%TYPE := v_val - 100000
    v_sup          opera.valoare%TYPE := v_val      ;
    numar          NUMBER(7);                  +
                    100000
    e_nimeni      EXCEPTION;
    mai_mult      EXCEPTION;
BEGIN
    SELECT COUNT(DISTINCT cod_autor)
    INTO   v_numar
    FROM   opera
    WHERE  valoare BETWEEN v_inf AND     v_sup;
    IF v_numar = 0 THEN
        RAISE e_nimeni;
    ELSIF v_numar > 0 THEN RAISE e_mai_mult;
    END IF;
EXCEPTION
    WHEN e_nimeni THEN
        :g_mesaj:='nu exista nici un artist cu valoarea
                   operelor cuprinsa intre'||v_inf|| si
                   ||v_sup; WHEN e_mai_mult THEN      -
                   -
                   :g_mesaj:='există'||v_numar|| artisti cu valoarea
                   operelor cuprinsa intre'||v_inf|| si'||v_sup;
    WHEN OTHERS THEN
        :g_mesaj:='au aparut alte erori';
END;
/
SET VERIFY ON
PRINT g_mesaj
```

Activarea unei exceptii exteme poate fi facuta si cu ajutorul procedurii ***RAISE APPLICATION ERROR***, furnizata de pachetul ***DBMS STANDARD***.

RAISEAPPLICATIONERROR poate fi folosita pentru a returna un mesaj de eroare unitatii care o apeleaza, mesaj mai descriptiv (non standard) decat identificatorul erorii. Unitatea apelanta poate fi ***SQL*Plus***, un subprogram ***PL/SQL*** sau o aplicatie *client*.

Procedura are urmatorul antet:

RAISE APPLICATION ERROR (*numareroare IN NUMBER*,
mesajeroare INVARCHAR2, [*{TRUE / FALSE}*]);

Atributul *numar eroare* este un numar cuprins intre -20000 si -20999, specificat de utilizator pentru exceptia respectiva, iar *mesaj eroare* este un text asociat erorii, care poate avea maximum 2048 octeti.

Parametrul boolean este optional. Daca acest parametru este *TRUE*, atunci noua eroare se va adauga listei erorilor existente, iar daca este *FALSE* (valoare implicita) atunci noua eroare va inlocui lista curenta a erorilor (se retine ultimul mesaj de eroare).

O aplicatie poate apela ***RAISEAPPLICATIONERROR*** numai dintr-un subprogram stocat (sau metoda). Daca ***RAISE APPLICATION ERROR*** este apelata, atunci subprogramul se termina si sunt returnate codul si mesajul asociate erorii respective.

Procedura ***RAISE APPLICATION ERROR*** poate fi folosita in sectiunea executabila, in sectiunea de tratare a erorilor si chiar simultan in ambele sectiuni.

- In sectiunea executabila:

```
DELETE FROM opera WHERE material = 'carton';
IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20201,'info incorecta');
END IF;
```

- In sectiunea de tratare a erorilor:

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20202,'info invalida');
END;
```

- In ambele sectiuni:

```
DECLARE
    e_material EXCEPTION;
    PRAGMA EXCEPTION_INIT (e_material, -20777);
BEGIN
```

```

DELETE FROM opera WHERE valoare < 100001;
IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20777,
        'nu exista opera cu aceasta valoare');
END IF;
EXCEPTION
    WHEN e_material THEN
        -- trateaza eroarea aceasta
END;

```

RAISE APPLICATION ERROR faciliteaza comunicatia dintre *client* si *server*, transmitand aplicatiei *client* erori specifice aplicatiei de pe *server* (de obicei, un declansator). Prin urmare, procedura este doar un mecanism folosit pentru comunicatia *server* ^ *client* a unei erori definite de utilizator, care permite ca procesul *client* sa trateze exceptia.

Exemplu:

Sa se implementeze un declansator care nu permite acceptarea in muzeu a operelor de arta avand valoarea mai mica de 100000\$.

```

CREATE OR REPLACE TRIGGER minim_valoare BEFORE INSERT
ON opera FOR EACH ROW BEGIN
    IF :NEW.valoare < 100000 THEN RAISE_APPLICATION_ERROR
        (-20005,'operele de arta trebuie sa aiba valoare mai
        mare de 100000$');
    END IF;
END;

```

Pe statia *client* poate fi scris un program care detecteaza si trateaza eroarea.

```

DECLARE
/* declarare exceptie */ nu_accepta EXCEPTION;
/* asociaza nume,codului eroare folosit in trigger */
PRAGMA EXCEPTION_INIT(nu_accepta,-20005);
BEGIN
/* incerc sa inserez */
INSERT INTO opera ...;
EXCEPTION
/* tratare exceptie */
WHEN nu_accepta THEN
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
    /* SQLERRM va returna mesaj din RAISE_APPLICATION_ERROR
*/ END;

```

Cazuri speciale in tratarea exceptiilor

Daca se declanseaza o exceptie intr-un bloc simplu, atunci se face saltul la partea de tratare (**handler**) a acesteia, iar dupa ce este terminata tratarea erorii seiese din bloc (instructiunea **END**).

Prin urmare, daca exceptia se propaga spre blocul care include blocul curent, restul actiunilor executabile din subbloc sunt „pierdute“. Daca dupa o eroare se doreste totusi continuarea prelucrarii datelor, este suficient ca instructiunea care a declansat exceptia sa fie inclusa intr-un subbloc.

Dupa ce subblocul a fost terminat, se continua secventa de instructiuni din blocul principal.

Exemplu:

```
BEGIN
DELETE ...
SELECT ...--poate declansa exceptia A
--nu poate fi efectuat INSERT care urmeaza INSERT INTO ...
EXCEPTION
WHEN A THEN ...
END;
```

Deficiența anterioara se poate rezolva incluzand intr-un subbloc comanda **SELECT** care a declansat exceptia.

```
BEGIN
DELETE ...
BEGIN
SELECT ...

EXCEPTION WHEN A THEN .
/* dupa ce se trateaza exceptia A, controlul este
transferat blocului de nivel superior, de fapt comenzii
INSERT */
END;
INSERT INTO .

EXCEPTION

END;
```

Uneori este dificil de aflat care comanda **SQL** a determinat o anumita eroare, deoarece există o singura secțiune pentru tratarea erorilor unui bloc. Sunt sugerate două soluții pentru rezolvarea acestei probleme.

1) Introducerea unui contor care sa identifice instructiunea *SQL*.

12

```
DECLARE
  v_sel_cont NUMBER(2):=1;
BEGIN
  SELECT ... v_sel_cont:=2;
  SELECT ... v_sel_cont:=3;
  SELECT ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    INSERT INTO log_table(info)
    VALUES ('comanda SELECT ' || TO_CHAR(v_sel_cont)
    || ' nu gaseste date');
END;
```

2) Introducerea fiecarei instructiuni *SQL* intr-un subbloc.

```
BEGIN
  BEGIN
    SELECT .
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      INSERT INTO log_table(info)
      VALUES ('SELECT 1 nu gaseste date');
  END;
  BEGIN
    SELECT .
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      INSERT INTO log_table(info)
      VALUES ('SELECT 2 nu gaseste date');
  END;
END;
```

Activarea exceptiilor

Pentru activarea unei exceptii exista doua metode:

- activarea explicita a exceptiei (definite de utilizator sau predefinite) in interiorul blocului, cu ajutorul comenzi **RAISE**;
- activarea automata a exceptiei asociate unei erori **Oracle**.

Exceptiile pot fi sesizate in sectiunea executabila, declarativa sau in cea de tratare a exceptiilor. La aceste niveluri ale programului, o exceptie poate fi gestionata in moduri diferite.

Pentru a reinvoca o exceptie, dupa ce a fost tratata in blocul curent, se foloseste instructiunea **RAISE**, dar fara a fi insotita de numele exceptiei. In acest fel, dupa executarea instructiunilor corespunzatoare tratarii exceptiei, aceasta se

transmite si blocului „parinte“. Pentru a fi recunoscuta ca atare de catre blocul „parinte“, exceptia trebuie sa nu fie definita in blocul curent, ci in blocul „parinte“ (sau chiar mai sus in ierarhie), in caz contrar ea putand fi captata de catre blocul „parinte“ doar la categoria *OTHERS*.

Pentru a executa acelasi set de actiuni in cazul mai multor exceptii nominalizate explicit, in sectiunea de prelucrare a exceptiilor se poate utiliza operatorul *OR*.

Pentru a evita tratarea fiecarei erori in parte, se foloseste sectiunea *WHEN OTHERS* care va cuprinde actiuni pentru fiecare exceptie care nu a fost tratata, adica pentru captarea exceptiilor neprevazute sau necunoscute. Aceasta sectiune trebuie utilizata cu atentie deoarece poate masca erori critice sau poate impiedica aplicatia sa raspunda in mod corespunzator.

Propagarea exceptiilor

Daca este declansata o eroare in sectiunea executabila si blocul curent are un *handler* pentru tratarea ei, atunci blocul se termina cu succes, iar controlul este dat blocului imediat exterior.

Daca se produce o exceptie care nu este tratata in blocul curent, atunci exceptia se propaga spre blocul „parinte“, iar blocul *PL/SQL* curent se termina fara succes. Procesul se repeta pana cand fie se gasesc intr-un bloc modalitatea de tratare a erorii, fie se opreste executia si se semnaleaza situatia aparuta (*unhandled exception error*).

Daca este declansata o eroare in partea declarativa a blocului, aceasta este propagata catre blocul imediat exterior, chiar daca exista un *handler* al acesteia in blocul corespunzator sectiunii declarative.

La fel se intampla daca o eroare este declansata in sectiunea de tratare a erorilor. La un moment dat, intr-o sectiune *EXCEPTION*, poate fi activa numai o singura exceptie.

Instructiunea *GOTO* nu permite:

- saltul la sectiunea de tratare a unei exceptii;
- saltul de la sectiunea de tratare a unei exceptii, in blocul curent.

Comanda *GOTO* permite totusi saltul de la sectiunea de tratare a unei exceptii la un bloc care include blocul curent.

Exemplu:

Exemplul urmator marcheaza un salt ilegal in blocul curent.

```
DECLARE
  v_var NUMBER(10,3);
BEGIN
  SELECT dim2/NVL(valoare,0) INTO v_var
  FROM opera WHERE diml > 100;
  <<eticheta>>
  INSERT INTO politaasig(cod_polita, valoare)
  VALUES (7531, v_var);      -
EXCEPTION
  WHEN ZERO_DIVIDE THEN v_var:=0;
  GOTO <<eticheta>>; --salt ilegal in blocul curent
END;
```

In continuare, vor fi analizate modalitatile de propagare a exceptiilor in cele trei cazuri comentate: exceptii sesizate in sectiunea declarativa, in sectiunea executabila si in sectiunea de tratare a erorilor.

Exceptie sesizata in sectiunea executabila

Exceptia este sesizata si tratata in subbloc. Dupa aceea, controlul revine blocului exterior.

```
DECLARE
  A EXCEPTION;
BEGIN
  BEGIN
    RAISE A; -- exceptia A sesizata in subbloc EXCEPTION
  WHEN A THEN ... -- exceptia tratata in subbloc END;
  -- aici este reluat controlul END;
```

Exceptia este sesizata in subbloc, dar nu este tratata in acesta si atunci se propaga spre blocul exterior. Regula poate fi aplicata de mai multe ori.

```
DECLARE
  A EXCEPTION;
  B EXCEPTION;
BEGIN
  BEGIN
    RAISE B; --exceptia B sesizata in subbloc EXCEPTION
  WHEN A THEN ...
  --exceptia B nu este tratata in subbloc END;
  EXCEPTION
    WHEN B THEN ...
    /* exceptia B s-a propagat spre blocul exterior unde a fost
       tratata, apoi controlul trece in exteriorul blocului */
  END;
```

Exceptie sesizata in sectiunea declarativa

Daca in sectiunea declarativa este generata o exceptie, atunci aceasta se propaga catre blocul exterior, unde are loc tratarea acestiei. Chiar daca exista un *handler* pentru exceptie in blocul curent, acesta nu este executat.

Exemplu:

Sa se realizeze un program prin care sa se exemplifice propagarea erorilor aparute in sectiunea declarativa a unui bloc *PL/SQL*.

Programul calculeaza numarul creatorilor de opere de arta care au lucrari expuse in muzeu.

```
BEGIN
DECLARE
nr_artisti      NUMBER(3) := 'XYZ';
BEGIN
SELECT COUNT (DISTINCT cod_autor)
INTO   nr_artisti
FROM   opera;
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('Eroare bloc intern:' || SQLERRM); END;
EXCEPTION
WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Eroare bloc extern:' || SQLERRM
);
END;
```

Deoarece la initializarea variabilei *nr artisti* apare o neconcordanta intre tipul declarat si cel asignat, este generata eroarea interna *VALUE ERROR*. Cum eroarea a aparut in partea declarativa a blocului intern, desi acesta contine un *handler OTHERS* care ar fi putut capta eroarea, *handler-ul* nu este executat, eroarea fiind propagata catre blocul extern unde este tratata in *handler-ul OTHERS* asociat. Aceasta se poate remarka deoarece la executie se obtine mesajul: „*Eroare bloc extern: ORA-06502: PL/SQL: numeric or value error*“

Exceptie sesizata in sectiunea *EXCEPTION*

Daca exceptia este sesizata in sectiunea *EXCEPTION* ea se propaga imediat spre blocul exterior.

```
BEGIN
DECLARE
  A EXCEPTION;
  B EXCEPTION;
BEGIN
  RAISE A; --sesizare exceptie A EXCEPTION
  WHEN A THEN
    RAISE B; --sesizare exceptie B WHEN B THEN ...
    /* exceptia este propagata spre blocul exterior cu
       toate ca exista aici un handler pentru ea */
  END;
  EXCEPTION
  WHEN B THEN .
  --exceptia B este tratata in blocul exterior END;
```

Informatii despre erori

Pentru a obtine textul corespunzator erorilor la compilare, poate fi utilizata vizualizarea *USERERRORS* din dictionarul datelor. Pentru informatii aditionale referitoare la erori pot fi consultate vizualizarile *ALL ERRORS* sau *DBAERRORS*.

Vizualizarea *USER ERRORS* are campurile:

NAME (numele obiectului),

TYPE (tipul obiectului),

SEQUENCE (numarul secventei),

LINE (numarul liniei din codul sursa in care a aparut eroarea),

POSITION (pozitia in linie unde a aparut eroarea),

TEXT (mesajul asociat erorii).

Exemplu:

Sa se afiseze erorile de compilare din procedura *alfa*.

```
SELECT LINE, POSITION, TEXT FROM      USER_ERRORS
WHERE NAME = 'ALFA';
```

LINE specifica numarul liniei in care apare eroarea, dar acesta nu corespunde liniei efective din fisierul text (se refera la codul sursa depus in *USERSOURCE*).