- tipurile de date care stochează valori numerice
  - tipul NUMBER cu subtipurile DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, INT, NUMERIC, REAL, SMALLINT;
  - tipul BINARY_INTEGER cu subtipurile NATURAL, NATURALN, POSITIVE, POSITIVEN, SIGNTYPE;
  - tipul PLS_INTEGER.
- tipurile de date care stochează caractere
  - tipul VARCHAR2 cu subtipurile STRING, VARCHAR;
  - tipul de date CHAR cu subtipul CHARACTER;
  - tipurile LONG, RAW, LONG RAW, ROWID.
- tipurile de date care stochează data calendaristică şi ora
  - tipurile DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND.
- tipurile de date globalizare ce stochează date unicode
  - tipurile NCHAR şi NVARCHAR2.
- tipul de date BOOLEAN stochează valori logice (true, false sau null).

```
[<<nume_bloc>>]
[DECLARE
variabile, cursoare]
BEGIN
instrucţiuni SQL şi PL/SQL
[EXCEPTION
tratarea erorilor]
END[nume_bloc]
```

```
SET SERVEROUTPUT ON;

<<principal>>
DECLARE
      v_client_id NUMBER(4):= 1600;
      v_client_nume VARCHAR2(50):= 'N1';
      v_nou_client_id NUMBER(3):= 500;
BEGIN
 <<secundar>>
 DECLARE
      v_client_id NUMBER(4) := 0;
      v_client_nume VARCHAR2(50) := 'N2';
      v_nou_client_id NUMBER(3) := 300;
      v_nou_client_nume VARCHAR2(50) := 'N3';
 BEGIN
      v_client_id:= v_nou_client_id;
      principal.v_client_nume:=
                              v_client_nume ||' '|| v_nou_client_nume;
    --poziţia 1
 END;
 v_client_id:= (v_client_id *12)/10;
 --poziţia 2
```

```
END;

-------------------------------------------------------------------------
-----------

VARIABLE g_mesaj VARCHAR2(50)
BEGIN
      :g_mesaj := 'Invat PL/SQL';
END;
/
PRINT g_mesaj

-------------------------------------------------------------------------
-----------

with zile_luna
  as
  (select trunc(sysdate,'month') + level-1 ziua
   from   dual
   connect by level<=extract (day from last_day(sysdate)))
select ziua, sum(case when to_char(ziua,'dd.mm.yyyy') =
to_char(book_date,'dd.mm.yyyy')
                      then 1
                      else 0
                 end) nr
from   zile_luna, rental
where  to_char(book_date,'mm-yyyy') = to_char(sysdate,'mm-yyyy')
group by ziua
order by 1;

-------------------------------------------------------------------------
-----------

IF v_salariu_anual>=20001
      THEN v_bonus:=2000;
ELSIF v_salariu_anual BETWEEN 10001 AND 20000
      THEN v_bonus:=1000;
ELSE v_bonus:=500;
END IF;

CASE WHEN v_salariu_anual>=20001
                  THEN v_bonus:=2000;
       WHEN v_salariu_anual BETWEEN 10001 AND 20000
                  THEN v_bonus:=1000;
       ELSE v_bonus:=500;
 END CASE;

-------------------------------------------------------------------------
-----------

 UPDATE ...
 SET ...
 WHERE ...
 IF SQL%ROWCOUNT = 0
```

```
      THEN "NU EXISTA"

-------------------------------------------------------------------------
-----------

LOOP
      ...
      EXIT WHEN condiţie;
END LOOP;

WHILE condiţie LOOP
      ...
END LOOP;

FOR contor_ciclu IN [REVERSE] lim_inf..lim_sup LOOP
      ...
END LOOP;

LOOP
      ...
      IF condiţie
                  THEN GOTO eticheta
END LOOP
<<eticheta>>

-------------------------------------------------------------------------
-----------
```

Tipul de date RECORD

- defineşte un grup de date stocate sub formă de câmpuri, fiecare cu tipul de date şi numele propriu;
- numărul de câmpuri nu este limitat;
- se pot defini valori iniţiale şi constrângeri NOT NULL asupra câmpurilor;
- câmpurile sunt iniţializate automat cu NULL;
- tipul RECORD poate fi folosit în secţiunea declarativă a unui bloc, subprogram sau pachet;
- se pot declara sau referi tipuri RECORD imbricate;
- sintaxa generală a definirii tipului RECORD este:

```
TYPE nume_tip IS RECORD
(nume_câmp1 {tip_câmp | variabilă%TYPE |
nume_tabel.coloană%TYPE | nume_tabel%ROWTYPE}
[ [NOT NULL] {:= | DEFAULT} expresie1][,
nume_câmp2 {tip_câmp | variabilă%TYPE |
nume_tabel.coloană%TYPE | nume_tabel%ROWTYPE}
[ [NOT NULL] {:= | DEFAULT} expresie2],…]);

DECLARE
      TYPE emp_record IS RECORD
                  (cod employees.employee_id%TYPE,
                  salariu employees.salary%TYPE,
                  job employees.job_id%TYPE);
      v_ang emp_record;
```

```
BEGIN
      v_ang.cod:=700;
      v_ang.salariu:= 9000;
      v_ang.job:='SA_MAN';
      ...
END
/

SELECT employee_id, salary, job_id
INTO v_ang
FROM employees
WHERE employee_id = 101;

DELETE FROM emp_***
WHERE employee_id=100
RETURNING employee_id, salary, job_id INTO v_ang;
```

Atributul %ROWTYPE

- Este utilizat pentru a declara o variabilă de tip înregistrare cu aceeași structură ca a altei variabile de tip înregistrare, a unui tabel sau cursor.

```
DECLARE
      v_ang1 employees%ROWTYPE;
      v_ang2 employees%ROWTYPE;
BEGIN
      ...
END
```

Metodele care se pot aplica colecțiilor PL/SQL sunt următoarele:

- COUNT întoarce numărul curent de elemente al unei colecții PL/SQL;
- DELETE(n) șterge elementul n dintr-o colecție PL/SQL; DELETE(m, n) șterge toate elementele având indecșii între m și n; DELETE șterge toate elementele unei colecții PL/SQL (nu este validă pentru tipul varrays);
- EXISTS(n) întoarce TRUE dacă există al n-lea element al unei colecții PL/SQL; altfel, întoarce FALSE;
- FIRST, LAST întorc indicele primului, respectiv ultimului element din colecție;
- NEXT(n), PRIOR(n) întorc indicele elementului următor, respectiv precedent celui de rang n din colecție, iar dacă nu există un astfel de element întorc valoarea null;
- EXTEND adaugă elemente la sfârșitul unei colecții: EXTEND adaugă un element null la sfârșitul colecției, EXTEND(n) adaugă n elemente null, EXTEND(n, i) adaugă n copii ale elementului de rang i (nu este validă pentru tipul index-by tables);

- LIMIT întoarce numărul maxim de elemente al unei colecţii (cel de la declarare) pentru tipul vector şi null pentru tablouri imbricate (nu este validă pentru tipul index-by tables);
- TRIM şterge elementele de la sfârşitul unei colecţii: TRIM şterge ultimul element, TRIM(n) şterge ultimele n elemente (nu este validă pentru tipul index-by tables). Similar metodei EXTEND, metoda TRIM operează asupra dimensiunii interne a tabloului imbricat.
- EXISTS este singura metodă care poate fi aplicată unei colecţii atomice null. Orice altă metodă declanşează excepţia COLLECTION_IS_NULL.
- COUNT, EXISTS, FIRST, LAST, NEXT, PRIOR şi LIMIT sunt funcţii, iar restul sunt proceduri PL/SQL.

Observaţie:
- Tipul tablou indexat poate fi utilizat numai în declaraţii PL/SQL. Tipurile vector şi tablou imbricat pot fi utilizate atât în declaraţii PL/SQL, cât şi în declaraţii la nivelul schemei (de exemplu, pentru definirea tipului unei coloane a unui tabel).
- Tablourile indexate pot avea indice negativ, domeniul permis pentru index fiind –2147483648..2147483647; pentru tablourile imbricate domeniul permis pentru index este 1..2147483647.
- Tablourile imbricate şi vectorii trebuie iniţializaţi şi/sau extinşi pentru a li se putea adăuga elemente noi.

Tablouri indexate (index-by table)
- Sunt mulţimi de perechi cheie-valoare, în care fiecare cheie este unică şi utilizată pentru a putea localiza valoarea asociată.
- Tablourile indexate pot creste în dimensiune în mod dinamic neavând specificat un număr maxim de elemente.
- Un tablou indexat nu poate fi iniţializat la declarare, este necesară o comandă explicită pentru a iniţializa fiecare element al său.
- Sintaxa generală pentru tabloul indexat este:

```
TYPE t_tablou_indexat IS TABLE OF {
{ tip_de_date | variabila%TYPE
| tabel.coloana%TYPE }[NOT NULL]}
| tabel%ROWTYPE }
INDEX BY PLS_INTEGER | BINARY_INTEGER | VARCHAR2(n);
v_tablou t_tablou_indexat;
```

```
DECLARE
      TYPE tablou_indexat IS TABLE OF emp_***%ROWTYPE INDEX BY
BINARY_INTEGER;
      t tablou_indexat;
BEGIN
      DELETE FROM emp_***
      WHERE ROWNUM<= 2
      RETURNING employee_id, first_name, last_name, email, phone_number,
hire_date, job_id, salary, commission_pct, manager_id, department_id
      BULK COLLECT INTO t;

      FOR i IN t.FIRST..t.LAST LOOP
                  IF t.EXISTS(i) THEN DBMS_OUTPUT.PUT(nvl(t(i), 0)|| ' ');
```

```
                END IF;
      END LOOP;

      t.delete;
      DBMS_OUTPUT.PUT_LINE('Tabloul are ' || t.COUNT ||' elemente.');
```

Tablouri imbricate (nested table)
Sintaxa generală pentru tabloul imbricat este:

```
[CREATE [OR REPLACE]] TYPE t_tablou_imbri IS TABLE OF
{ { tip_de_date | variabila%TYPE |
tabel.coloana%TYPE }[NOT NULL]} | tabel%ROWTYPE };
v_tablou_imbri t_tablou_imbri;
```

- Singura diferență sintactică între tablourile indexate și cele imbricate este absența clauzei INDEX BY. Mai exact, dacă această clauză lipsește tipul de date declarat este tablou imbricat.
- Numărul maxim de linii al unui tablou imbricat este dat de capacitatea maximă 2 GB.
- Tablourile imbricate:
  - folosesc drept indici numere consecutive;
  - sunt asemenea unor tabele cu o singură coloană;
  - nu au dimensiune limitată, ele cresc dinamic;
  - inițial, un tablou imbricat este dens (are elementele pe poziții consecutive), dar pot apărea spații goale prin ștergere;
  - metoda NEXT ne permite să ajungem la următorul element;
  - pentru a insera un element nou, tabloul trebuie extins cu metoda EXTEND.
- 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.
- Tablourile imbricate trebuie inițializate cu ajutorul constructorului.
  - PL/SQL apelează un constructor numai în mod explicit.
  - Tabelele indexate nu au constructori.
  - Constructorul primește ca argumente o listă de valori numerotate în ordine, de la 1 la numărul de valori date ca parametrii constructorului.
  - Dimensiunea inițială a colecției este egală cu numărul de argumente date în constructor, atunci când aceasta este inițializată.
  - Pentru vectori nu poate fi depășită dimensiunea maximă precizată la declarare.
  - Atunci când constructorul este apelat fără argumente se va crea o colecție fără niciun element (vidă), dar care este not null.

```
DECLARE
      TYPE tablou_imbricat IS TABLE OF NUMBER;
      t tablou_imbricat := tablou_imbricat();
      TYPE tablou_imbricat2 IS TABLE OF CHAR(1);
      t2 tablou_imbricat2 := tablou_imbricat2('m', 'i', 'n', 'i', 'm');
      i INTEGER;
BEGIN
```

```
      FOR i IN 1..10 LOOP
                  t.extend;
                  t(i):=i;
      END LOOP;
      t.DELETE(t.first);
      t.DELETE(5,7);
      t.DELETE(t.last);
      i := t.LAST;
      WHILE i >= t.FIRST LOOP
                  DBMS_OUTPUT.PUT(t(i));
                  i := t.PRIOR(i);
      END LOOP;
END
```

Vectori

- Sintaxa generală pentru declararea vectorilor:

```
[CREATE [OR REPLACE]] TYPE t_vector IS VARRAY(limita) OF
{ { tip_de_date | variabila%TYPE |
Tabel.coloana%TYPE }[NOT NULL]}
| tabel%ROWTYPE };
v_vector t_vector;
```

- Spre deosebire de tablourile imbricate, vectorii au o dimensiune maximă (constantă) stabilită la declarare. În special, se utilizează pentru modelarea relaţiilor one-to-many, atunci când numărul maxim de elemente din partea „many" este cunoscut şi ordinea elementelor este importantă.

metodele DELETE(n), DELETE(m,n) nu sunt valabile pentru vectori!!! -- din vectori nu se pot sterge elemente individuale!!!

```
DECLARE
      TYPE vector IS VARRAY(2) OF NUMBER;
      t vector:= vector();
BEGIN
      FOR i IN 1..10 LOOP
                  t.extend;
                  t(i):=i;
      END LOOP;
END
```

```
CREATE OR REPLACE TYPE subordonati_*** AS VARRAY(10) OF NUMBER(4);
CREATE TABLE manageri_*** (cod_mgr NUMBER(10), nume VARCHAR2(20), lista
subordonati_***);
```

```
DECLARE
      v_sub subordonati_***:= subordonati_***(100,200,300);
      v_lista manageri_***.lista%TYPE;
BEGIN
      INSERT INTO manageri_*** VALUES (1, 'Mgr 1', v_sub);
      INSERT INTO manageri_*** VALUES (2, 'Mgr 2', null);
      INSERT INTO manageri_*** VALUES (3, 'Mgr 3',
subordonati_***(400,500));
```

```
END

CREATE TABLE emp_test_*** AS SELECT employee_id, last_name FROM employees
WHERE ROWNUM <= 2;
CREATE OR REPLACE TYPE tip_telefon_*** IS TABLE OF VARCHAR(12);

ALTER TABLE emp_test_*** ADD (telefon tip_telefon_***) NESTED TABLE
telefon STORE AS tabel_telefon_***;
INSERT INTO emp_test_*** VALUES (500, 'XYZ',tip_telefon_***('074XXX',
'0213XXX', '037XXX'));
UPDATE emp_test_*** SET telefon = tip_telefon_***('073XXX', '0214XXX')
WHERE employee_id=100;
SELECT a.employee_id, b.* FROM emp_test_*** a, TABLE (a.telefon) b;
```

Obs. Comanda FORALL permite ca toate liniile unei colecţii să fie transferate simultan printr-o singură operaţie. Procedeul este numit bulk bind.

```
FORALL index IN lim_inf..lim_sup
comanda_sql;

DECLARE
      TYPE tip_cod IS VARRAY(20) OF NUMBER;
      coduri tip_cod := tip_cod(205,206);
BEGIN
      FORALL i IN coduri.FIRST..coduri.LAST
                  DELETE FROM emp_***
                  WHERE employee_id = coduri (i);
      END;
END
```

Serverul Oracle alocă o zonă de memorie, care se numeşte cursor, ori de câte ori este înaintată o cerere SQL.
Cursoarele pot fi de două feluri:

- implicite (create şi gestionate de PL/SQL în mod automat)

Cursoarele implicite sunt declarate de PL/SQL în mod implicit pentru toate comenzile LMD şi comanda SELECT, inclusiv comenzile care returnează o singura linie.

- explicite (create şi gestionate de utilizator).

Cursoarele explicite sunt create pentru cereri care returnează mai mult de o linie.
Mulţimea de linii procesate returnate de o cerere multiple-row se numeşte set activ.
Un cursor este o modalitate de a parcurge setul activ linie cu linie.

Etapele utilizării unui cursor:
a) Declarare.
Defineşte numele şi structura cursorului, împreună cu clauza SELECT care va popula cursorul cu date. Cererea este validată, dar nu executată.

În secţiunea declarativă prin intermediul cuvântului cheie CURSOR:
CURSOR c_nume_cursor [(parametru tip_de_date, ..)] IS
comanda SELECT;
b) Deschidere (comanda OPEN).
Este executată cererea şi se populează cursorul cu date.
OPEN c_nume_cursor [(parametru, …)];
c) Încărcare (comanda FETCH).
Încarcă un rând din cursor (acela indicat de pointerul cursorului) în variabile şi mută pointerul la rândul următor.
Numărul de variabile din clauza INTO trebuie să se potrivească cu lista SELECT returnată de cursor.
FETCH c_nume_cursor INTO variabila, …;
d) Verificare dacă nu am ajuns cumva la finalul setului activ folosind atributele definite si la cursoare implicite:
C_nume_cursor%NOTFOUND – TRUE, dacă nici un rând nu a fost procesat
C_nume_cursor%FOUND – TRUE, dacă cel puţin un rând a fost procesat
Dacă nu s-a ajuns la final mergi la c).
e) Închidere cursor (dacă rămâne deschis cursorul consumă din resursele serverului)
CLOSE c_nume_cursor;
Şterge datele din cursor şi îl închide. Acesta poate fi redeschis pentru o actualizare a datelor.
De asemenea, se pot folosi şi atributele %ISOPEN si %ROWCOUNT:
- %ROWCOUNT reprezintă numărul de rânduri procesate de cea mai recentă comandă SQL;
- %ISOPEN este TRUE în cazul în care cursorul este deschis şi FALSE, dacă nu a fost deschis sau a fost închis; se foloseşte numai cu cursoarele implicite.

```
DECLARE
      v_nr number(4);
      v_nume departments.department_name%TYPE;
      CURSOR c IS
                  SELECT department_name nume, COUNT(employee_id) nr
                  FROM departments d, employees e
                  WHERE d.department_id=e.department_id(+)
                  GROUP BY department_name;
BEGIN
      OPEN c;
      LOOP
                  FETCH c INTO v_nume,v_nr;
                  EXIT WHEN c%NOTFOUND;
                  IF v_nr=0 THEN
                              DBMS_OUTPUT.PUT_LINE('In departamentul '||
v_nume|| ' nu lucreaza angajati');
                  ELSIF v_nr=1 THEN
                              DBMS_OUTPUT.PUT_LINE('In departamentul '||
v_nume|| ' lucreaza un angajat');
                  ELSE
                              DBMS_OUTPUT.PUT_LINE('In departamentul '||
v_nume|| ' lucreaza '|| v_nr||' angajati');
                  END IF;
      END LOOP;
      CLOSE c;
```

```
END;

-----

DECLARE
      TYPE tab_nume IS TABLE OF departments.department_name%TYPE;
      TYPE tab_nr IS TABLE OF NUMBER(4);
      t_nr tab_nr;
      t_nume tab_nume;
      CURSOR c IS
                  SELECT department_name nume, COUNT(employee_id) nr
                  FROM departments d, employees e
                  WHERE d.department_id=e.department_id(+)
                  GROUP BY department_name;
BEGIN
      OPEN c;
      FETCH c BULK COLLECT INTO t_nume, t_nr;
      CLOSE c;
      FOR i IN t_nume.FIRST..t_nume.LAST LOOP
                  ... (t_nr(i) , t_nume(i))
      END LOOP;
END;

-----

DECLARE
      CURSOR c IS
                  SELECT department_name nume, COUNT(employee_id) nr
                  FROM departments d, employees e
                  WHERE d.department_id=e.department_id(+)
                  GROUP BY department_name;
BEGIN
      FOR i in c LOOP
                  ... (i.nr, i.nume)
      END LOOP

-----

BEGIN
      FOR i in (select ...) LOOP
                  ... (i.nr, i.nume)
      END LOOP
END

-----

DECLARE
      v_x number(4) := &p_x;
      ...
      CURSOR c (parametru NUMBER) IS
                  ...
                  having ... > parametru
BEGIN
      OPEN c (v_x);
```

```
      ...
END
```

Observaţie: Uneori este necesară blocarea liniilor înainte ca acestea să fie şterse sau reactualizate. Blocarea se poate realiza (atunci când cursorul este deschis) cu ajutorul comenzii SELECT care conţine clauza FOR UPDATE.
Comanda SELECT are următoarea extensie PL/SQL pentru blocarea explicita înregistrărilor ce urmează a fi prelucrate (modificate sau şterse):
SELECT … FROM … WHERE …GROUP BY ...ORDER BY …
FOR UPDATE [OF lista_coloane] [NOWAIT | WAIT n];
În cazul în care liniile selectate de cerere nu pot fi blocate din cauza altor blocări atunci:

- dacă se foloseşte NOWAIT apare imediat eroarea ORA-00054;
- dacă nu se foloseşte NOWAIT atunci se aşteaptă până când liniile sunt deblocate;
- dacă se foloseşte WAIT n atunci se aşteaptă un număr determinat de secunde pentru ca liniile ce trebuie selectate pentru modificare să fie deblocate.

Pentru a modifica o anumită linie returnată de un astfel de cursor se foloseşte clauza:
WHERE CURRENT OF nume_cursor

```
DECLARE
      CURSOR c IS
                  SELECT *
                  FROM emp_***
                  WHERE TO_CHAR(hire_date, 'YYYY') = 2000
                  FOR UPDATE OF salary NOWAIT;
BEGIN
      FOR i IN c LOOP
                  UPDATE emp_***
                  SET salary = salary+1000
                  WHERE CURRENT OF c;
      END LOOP;
END;
```

```
DECLARE
      TYPE refcursor IS REF CURSOR;
      CURSOR c_dept IS
                  SELECT ... , CURSOR (...)
                  ...
      v_cursor refcursor;
BEGIN
      OPEN c_dept;
      LOOP
                  FETCH c_dept INTO ..., v_cursor
                  ...
      END LOOP
      CLOSE c_dept;
```

```
END

-----

DECLARE
      TYPE refcursor IS REF CURSOR RETURN employees%ROWTYPE;
      v_emp refcursor;
BEGIN
      IF ... THEN
                  OPEN v_emp FOR SELECT ...
      ELSE
                  OPEN v_emp FOR SELECT ...
      END IF
END

-----

DECLARE
      TYPE v_emp IS REF CURSOR
      ...
      v_nr INTEGER := &n;
BEGIN
      OPEN v_emp FOR
                  'SELECT ...' ||
                  'FROM ...' ||
                  'WHERE ... > :bind_var'
                  USING v_nr;
      ...
END
```

Un subprogram este un bloc PL/SQL cu nume (spre deosebire de blocurile anonime) care poate primi parametri şi poate fi invocat dintr-un anumit mediu (de exemplu, SQL*Plus, Oracle Forms, Oracle Reports, Pro*C etc.) Subprogramele sunt bazate pe structura cunoscută de bloc PL/SQL. Similar, acestea conţin o parte declarativă facultativă, o parte executabilă obligatorie şi o parte de tratare de excepţii facultativă.
Exista 2 tipuri de subprograme:
- proceduri;
- funcţii (trebuie să conţină cel puţin o comandă RETURN).

Acestea pot fi locale (declarate într-un bloc PL/SQL) sau stocate (create cu comanda CREATE). Odată create, procedurile şi funcţiile sunt stocate în baza de date. De aceea ele se numesc subprograme stocate.

• Sintaxa simplificată pentru crearea unei proceduri este următoarea:

```
[CREATE [OR REPLACE] ]
PROCEDURE nume_procedură [ (lista_parameteri) ]
{IS | AS}
[declaraţii locale]
BEGIN
partea executabilă
[EXCEPTION
```

```
partea de tratare a excepţiilor]
END [nume_procedură];
```

• Sintaxa simplificată pentru scrierea unei funcţii este următoarea:

```
[CREATE [OR REPLACE] ]
FUNCTION nume_funcţie [ (lista_parameteri) ]
RETURN tip_de_date
{IS | AS}
[declaraţii locale]
BEGIN
partea executabilă
[EXCEPTION
partea de tratare a excepţiilor]
END [nume_funcţie];
```

• Lista de parametri conţine specificaţii de parametri separate prin virgulă:

```
nume_parametru mod_parametru;
Mod_parametru poate fi:
- de intrare (IN) – singurul care poate avea o valoare iniţială;
- de intrare / ieşire (IN OUT);
- de ieşire (OUT);
- are valoarea implicită IN.
```

• În cazul în care se modifică un obiect (vizualizare, tabel etc) de care depinde un subprogram, acesta este invalidat. Revalidarea se face ori prin recrearea subprogramului ori prin comanda:

```
ALTER PROCEDURE nume_proc COMPILE;
ALTER FUNCTION nume_functie COMPILE;
```

• Ştergerea unei funcţii sau proceduri se realizează prin comenzile:

```
DROP PROCEDURE nume_proc;
DROP FUNCTION nume_functie;
```

• Informaţii despre procedurile şi funcţiile deţinute de utilizatorul curent se pot obţine interogând vizualizarea USER_OBJECTS.

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

• Codul complet al unui subprogram poate fi vizualizat folosind următoarea sintaxă:

```
SELECT TEXT
FROM USER_SOURCE
WHERE NAME =UPPER('nume_subprogram');
```

Tipul unui subprogram se obţine prin comanda DESCRIBE.

• Eroarea apărută la compilarea unui subprogram poate fi vizualizată folosind următoarea sintaxă:

```
SELECT LINE, POSITION, TEXT
FROM USER_ERRORS
WHERE NAME =UPPER('nume');
```

Erorile pot fi vizualizate şi prin intermediul comenzii SHOW ERRORS.

```
-----

DECLARE
      v_nume ...
      FUNCTION f1 RETURN NUMBER IS
                  salariu ...
      BEGIN
                  SELECT ... INTO salariu
                  ...
                  WHERE ... = v_nume;
                  RETURN salariu;
      EXCEPTION
                  WHEN NO DATA FOUND THEN
                              ...
                  WHEN TOO MANY ROWS THEN
                              ...
                  WHEN OTHERS THEN
                              ...
      END f1
BEGIN
      DBMS_OUTPUT.PUT_LINE('Salariul este '|| f1);
END

-----

CREATE OR REPLACE FUNCTION f2 (v_nume VARCHAR2(30) DEFAULT 'Bell')
RETURN NUMBER IS
      salariu ...
      BEGIN
                  ...
                  RETURN salariu;
      EXCEPTION
                  WHEN NO DATA FOUND THEN
                              RAISE_APPLICATION_ERROR(-20000, 'Nu exista
angajati cu numele dat');
                  WHEN ...
END f2

BEGIN
      DBMS_OUTPUT.PUT_LINE('Salariul este '|| f2('King'));
      DBMS_OUTPUT.PUT_LINE('Salariul este '|| f2);
END

SELECT *
from employees
where salary = f2('King');

SELECT f2 FROM DUAL;
SELECT f2('King') FROM DUAL;

VARIABLE nr NUMBER
EXECUTE :nr := f2('Bell');
PRINT nr
```

```
VARIABLE nr NUMBER
CALL f2('Bell') INTO :nr;
PRINT nr

-----

CREATE OR REPLACE PROCEDURE p1 (v_nume VARCHAR2(30) IS
      salariu ...
      BEGIN
                  SELECT ... INTO salariu
                  ...
                  DBMS_OUTPUT.PUT_LINE('Salariul este '|| salariu);
      EXCEPTION
                  ...
END p1;

BEGIN
      p1;
END;

EXECUTE p1('Bell');
EXECUTE p1(v_nume=>'Bell')

-----

CREATE OR REPLACE PROCEDURE p2 (v_nume IN employees.last_name%TYPE,
salariu OUT employees.last_name%TYPE) IS ...

DECLARE
      v_salariu ...
BEGIN
      p2('Bell',v_salariu)
      DBMS_OUTPUT.PUT_LINE('Salariul este '|| v_salariu);
END

VARIABLE v_sal NUMBER
EXECUTE p2('Bell',:v_sal)
PRINT val

-----

VARIABLE ang_man NUMBER
BEGIN
      :ang_man:=200;
END;

CREATE OR REPLACE PROCEDURE p3 (nr IN OUT NUMBER) IS
      BEGIN
                  SELECT manager_id INTO nr
                  FROM employees
                  WHERE employee_id=nr;
END p3;

EXECUTE p3(:ang_man)
```

```
PRINT ang_man

-----

CREATE OR REPLACE FUNCTION factorial(n NUMBER)
RETURN INTEGER IS
      BEGIN
                  IF (n=0) THEN
                              RETURN 1;
                  ELSE
                              RETURN n*factorial;
                  END IF;
END factorial;
```

- Pachetele sunt unităţi de program care pot cuprinde proceduri, funcţii, cursoare, tipuri de

date, constante, variabile şi excepţii.

- Pachetele nu pot fi apelate, nu pot transmite parametri şi nu pot fi încuibărite.
- Un pachet are două părţi, fiecare fiind stocată separat în dicţionarul datelor:

- specificaţia pachetului;

```
CREATE [OR REPLACE] PACKAGE [schema.]nume_pachet
      {IS | AS}
      declaraţii;
END [nume_pachet];
```

- corpul pachetului.

```
CREATE [OR REPLACE] PACKAGE BODY [schema.]nume_pachet
      {IS | AS}
      [BEGIN]
                  instrucţiuni;
END [nume_pachet];
```

- Recompilarea pachetului

```
      ALTER PACKAGE [schema.]nume_pachet
      COMPILE [ {PACKAGE | BODY} ];
```

- Eliminarea pachetului

```
      DROP PACKAGE [schema.]nume_pachet
      [ {PACKAGE | BODY} ];

-----

CREATE OR REPLACE PACKAGE pac1 AS
      FUNCTION f1 (...)
                  RETURN NUMBER;
      PROCEDURE p1 (...);
      CURSOR c1 (...) RETURN ...%ROWTYPE;
END pac1;
```

```
CREATE OR REPLACE PACKAGE BODY pac1 AS
      FUNCTION f1 (...)
                  RETURN NUMBER IS
                              ...
                  BEGIN
                              ...
                              RETURN ...;
      END;

      PROCEDURE p1 (...) IS
                  ...
                  BEGIN
                              ...
      END;

      CURSOR c1 (...) RETURN ...%ROWTYPE
                  IS ...
END pac1;

SELECT pac1.f1(...)
FROM DUAL;
SELECT pac1.p1(...)
FROM DUAL;
```

---

1. Pachetul DBMS_OUTPUT permite afişarea de informaţii. Procedurile pachetului sunt:

PUT – depune (scrie) în buffer informaţie;
PUT_LINE – depune în buffer informaţia, împreună cu un marcaj de sfârşit de linie;
NEW_LINE – depune în buffer un marcaj de sfârşit de linie;
GET_LINE – regăseşte o singură linie de informaţie;
GET_LINES – regăseşte mai multe linii de informaţie;
ENABLE/DISABLE – activează/dezactivează procedurile pachetului.

2. Pachetul DBMS_JOB este utilizat pentru planificarea execuţiei programelor PL/SQL

SUBMIT – adaugă un nou job în coada de aşteptare a job-urilor;
REMOVE – şterge un job din coada de aşteptare;
RUN – execută imediat un job specificat.

```
VARIABLE nr_job NUMBER

BEGIN
      DBMS_JOB.SUBMIT(
                  -- întoarce numărul jobului, printr-o variabilă de
legătură
                  JOB => :nr_job,
                  -- codul PL/SQL care trebuie executat
                  WHAT => 'f1(...)',
                  -- data de start a execuţiei (dupa 30 secunde)
```

```
                NEXT_DATE => SYSDATE+30/86400
                -- intervalul de timp la care se repetă execuţia
                INTERVAL => 'SYSDATE+1');

                COMMIT;
END;

-- informatii despre joburi
SELECT JOB, NEXT_DATE, WHAT
FROM USER_JOBS;
```

3. Pachetul UTL_FILE extinde operaţiile I/O la fişiere. Se apelează funcţia FOPEN pentru a deschide un fişier; acesta este folosit pentru operaţiile de citire sau scriere. După ce s-au încheiat operaţiile I/O se închide fişierul (FCLOSE).

```
CREATE OR REPLACE PROCEDURE scriu_fisier
      (director VARCHAR2,
      fisier VARCHAR2) IS
      v_file UTL_FILE.FILE_TYPE;
      CURSOR cursor_rez IS
                  SELECT department_id departament, SUM(salary) suma
                  FROM employees
                  GROUP BY department_id
                  ORDER BY SUM(salary);
      v_rez cursor_rez%ROWTYPE;
BEGIN
      v_file:=UTL_FILE.FOPEN(director, fisier, 'w');
      UTL_FILE.PUTF(v_file, 'Suma salariilor pe departamente \n Raport
generat pe data ');
      UTL_FILE.PUT(v_file, SYSDATE);
      UTL_FILE.NEW_LINE(v_file);
      OPEN cursor_rez;
      LOOP
                  FETCH cursor_rez INTO v_rez;
                  EXIT WHEN cursor_rez%NOTFOUND;
                  UTL_FILE.NEW_LINE(v_file);
                  UTL_FILE.PUT(v_file, v_rez.departament);
                  UTL_FILE.PUT(v_file, ' ');
                  UTL_FILE.PUT(v_file, v_rez.suma);
      END LOOP;
      CLOSE cursor_rez;
      UTL_FILE.FCLOSE(v_file);
END;
/
SQL> EXECUTE scriu_fisier('F:\','test.txt');
```

Un declanşator este un bloc PL/SQL care se execută automat ori de câte ori are loc un anumit eveniment “declanşator” (de exemplu, inserarea unei linii într-un tabel, ştergerea unor înregistrări etc.)

Tipuri de declanşatori:

- la nivel de bază de date – pot fi declanşaţi de o comandă LMD asupra datelor unui tabel; o comandă LMD asupra datelor unei vizualizări; o comandă LDD (CREATE, ALTER, DROP) referitoare la anumite obiecte ale schemei sau ale bazei de date; un eveniment sistem (SHUTDOWN, STARTUP); o acţiune a utilizatorului (LOGON, LOGOFF); o eroare (SERVERERROR, SUSPEND).
- la nivel de aplicaţie – se declanşează la apariţia unui eveniment într-o aplicaţie particulară.

• Sintaxa comenzii de creare a unui declanşator LMD este următoarea:

```
CREATE [OR REPLACE] TRIGGER [schema.]nume_declanşator
{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 (condiţie) ]
corp_declanşator;
```

• În cazul declanşatorilor LMD este important să stabilim:

- momentul când este executat declanşatorul: BEFORE, AFTER
- ce fel de acţiuni îl declanşează: INSERT, UPDATE, DELETE
- tipul declanşatorului: la nivel de instrucţiune sau la nivel de linie (FOR EACH ROW).

• Sintaxa comenzii de creare a unui declanşator INSTEAD OF este următoarea:

```
CREATE [OR REPLACE] TRIGGER [schema.]nume_trigger
--momentul când este declanşat
INSTEAD OF
--comanda/comenzile care îl declanşează
{ DELETE|INSERT|UPDATE [OF coloana[, coloana …] ] }
[OR {DELETE|INSERT|UPDATE [OF coloana[, coloana …] ] …}
ON [schema.]nume_vizualizare
[REFERENCING {OLD [AS] vechi NEW [AS] nou
| NEW [AS] nou OLD [AS] vechi } ]
FOR EACH ROW
[WHEN (condiţie) ]
corp_trigger (bloc anonim PL/SQL sau comanda CALL);
```


• Sintaxa comenzii de creare a unui declanşator sistem este următoarea:

```
CREATE [OR REPLACE] TRIGGER [schema.]nume_trigger
{BEFORE | AFTER}
{comenzi_LDD | evenimente_sistem}
ON {DATABASE | SCHEMA}
[WHEN (condiţie) ]
corp_trigger;
```

• Informaţii despre declanşatori se pot obţine interogând vizualizările

- USER_TRIGGERS, ALL_TRIGGERS, DBA_TRIGGERS
- USER_TRIGGER_COL

• Dezactivarea, respectiv activarea declanşatorilor se realizează prin următoarele comenzi:

```
ALTER TABLE nume_tabel
```

```
DISABLE ALL TRIGGERS;
ALTER TABLE nume_tabel
ENABLE ALL TRIGGERS;
ALTER TRIGGER nume_trig ENABLE;
ALTER TRIGGER nume_trig DISABLE;
```

- Eliminarea unui declanșator se face prin

```
DROP TRIGGER nume_trig;

-----

CREATE OR REPLACE TRIGGER trig1_***
BEFORE INSERT OR UPDATE OR DELETE ON emp_***
 BEGIN
      IF (TO_CHAR(SYSDATE,'D') = 1) OR (TO_CHAR(SYSDATE,'HH24') NOT
BETWEEN 8 AND 20)
                  THEN
                              RAISE_APPLICATION_ERROR(-20001,'tabelul nu
poate fi actualizat');
      END IF;
END;
/
DROP TRIGGER trig1_***;

-----

CREATE OR REPLACE TRIGGER trig21_***
BEFORE UPDATE OF salary ON emp_***
FOR EACH ROW
      BEGIN
                  IF (:NEW.salary < :OLD.salary)
                              THEN
                                    RAISE_APPLICATION_ERROR(-
20002,'salariul nu poate fi micsorat');
                  END IF;
END; /

-----

CREATE OR REPLACE TRIGGER trig22_***
BEFORE UPDATE OF salary ON emp_***
FOR EACH ROW
WHEN (NEW.salary < OLD.salary)
      BEGIN
                  RAISE_APPLICATION_ERROR(-20002,'salariul nu poate fi
micsorat');
END;

-----

CREATE OR REPLACE VIEW v_info_*** AS
      SELECT e.id, e.nume, e.prenume, e.salariu, e.id_dept, d.nume_dept,
d.plati
      FROM info_emp_*** e, info_dept_*** d
      WHERE e.id_dept = d.id;
```

```
SELECT *
FROM user_updatable_columns
WHERE table_name = UPPER('v_info_***');

CREATE OR REPLACE TRIGGER trig5_***
INSTEAD OF INSERT OR DELETE OR UPDATE ON v_info_***
FOR EACH ROW
 BEGIN
      IF INSERTING THEN
                  -- inserarea in vizualizare determina inserarea
                  -- in info_emp_*** si reactualizarea in info_dept_***
                  -- se presupune ca departamentul exista
                  INSERT INTO info_emp_***
                  VALUES (:NEW.id, :NEW.nume, :NEW.prenume, :NEW.salariu,
:NEW.id_dept);
                  UPDATE info_dept_***
                  SET plati = plati + :NEW.salariu WHERE id = :NEW.id_dept;

      ELSIF DELETING THEN
                  -- stergerea unui salariat din vizualizare determina
                  -- stergerea din info_emp_*** si reactualizarea in
info_dept_***
                  DELETE FROM info_emp_***
                  WHERE id = :OLD.id;
                  UPDATE info_dept_***
                  SET plati = plati - :OLD.salariu
                  WHERE id = :OLD.id_dept;

      ELSIF UPDATING ('salariu') THEN
                  /* modificarea unui salariu din vizualizare determina
modificarea salariului in info_emp_*** si reactualizarea in info_dept_***
*/ UPDATE info_emp_***
                  SET salariu = :NEW.salariu
                  WHERE id = :OLD.id;
                  UPDATE info_dept_***
                  SET plati = plati - :OLD.salariu + :NEW.salariu
                  WHERE id = :OLD.id_dept;

      ELSIF UPDATING ('id_dept') THEN
                  /* modificarea unui cod de departament din vizualizare
determina modificarea codului in info_emp_*** si reactualizarea in
info_dept_*** */
                  UPDATE info_emp_***
                  SET id_dept = :NEW.id_dept
                  WHERE id = :OLD.id;
                  UPDATE info_dept_***
                  SET plati = plati - :OLD.salariu
                  WHERE id = :OLD.id_dept;
                  UPDATE info_dept_***
                  SET plati = plati + :NEW.salariu
                  WHERE id = :NEW.id_dept;
      END IF;
```

```
--------------------------------------------------------------------------
-----------

EXCEPTION
WHEN nume_excepţie1 [OR nume_excepţie2 …] THEN
secvenţa_de_instrucţiuni_1;
[WHEN nume_excepţie3 [OR nume_excepţie4 …] THEN
secvenţa_de_instrucţiuni_2;]
…
[WHEN OTHERS THEN
secvenţa_de_instrucţiuni_n;]
END;

EXCEPTION
      WHEN NO_DATA_FOUND THEN
                  DBMS_OUTPUT.PUT_LINE (' no data found: ' ||SQLCODE || ' -
' || SQLERRM);
      WHEN TOO_MANY_ROWS THEN
                  DBMS_OUTPUT.PUT_LINE (' too many rows: ' ||SQLCODE || ' -
' || SQLERRM);
      WHEN INVALID_NUMBER THEN
                  DBMS_OUTPUT.PUT_LINE (' invalid number: ' ||SQLCODE || '
- ' || SQLERRM);
      WHEN CURSOR_ALREADY_OPEN THEN
                  DBMS_OUTPUT.PUT_LINE (' cursor already open: ' ||SQLCODE
|| ' - ' || SQLERRM);
      WHEN OTHERS THEN
                  DBMS_OUTPUT.PUT_LINE (SQLCODE || ' - ' || SQLERRM);
END;

SELECT LINE, POSITION, TEXT
FROM USER_ERRORS
WHERE NAME = UPPER('nume');
```