

## 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 anonime).

- Procedurile care sunt declarate și apelate în blocuri anonime 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 opțional, o parte de tratare a erorilor.

## Dezvoltare de subprograme utilizând *iSQL\*Plus*

*Oracle9i* dispune de o interfață *Internet* pentru *SQL\*Plus*, care este *iSQL\*Plus*. Se poate utiliza un *Web browser* pentru conectare la o bază de date *Oracle* și pentru a executa prin *iSQL\*Plus* orice acțiune fezabilă cu *SQL\*Plus*.

Algoritmul de lucru este următorul:

- 1) se utilizează un editor de texte pentru a crea un *script file SQL* (extensia este *.sql*);
- 2) se utilizează opțiunea *Browse* din meniu pentru a localiza acest fișier;
- 3) se utilizează opțiunea *Load Script* din meniul afișat pentru a încărca acest *script* în *buffer-ul iSQL\*Plus*;
- 4) se utilizează opțiunea *Execute* din meniu pentru executarea codului (implicit, rezultatul codului este afișat pe ecran).

## 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* (în *iSQL\*Plus* sau în *SQL\*Plus*) 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*.

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 comenzii *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 *server-ul* va executa blocul de inițializare al pachetului.

## Dezvoltare de subprograme utilizând *Oracle Procedure Builder*

*Oracle Procedure Builder* permite crearea și depanarea diferitelor unități de program (aplicație sau unități stocate). Subprogramele pot fi editate, create, compilate, salvate, depanate, apelate utilizând facilitățile oferite de acest instrument *Oracle*, prin intermediul unui editor grafic. *Procedure Builder* permite dezvoltarea de subprograme *PL/SQL* care pot fi folosite de aplicații *client* și aplicații *server*.

*Procedure Builder* conține cinci componente.

- *Object Navigator* furnizează o interfață *outline* pentru a vizualiza obiecte, relații între obiecte, pentru a edita proprietăți ale obiectelor. Cu ajutorul acestei componente poate fi listată o ierarhie a tuturor obiectelor ce pot fi accesate în timpul sesiunii utilizatorului.
- *PL/SQL Interpreter* depanează și evaluează codul *PL/SQL* în timp real.
- *Program Unit Editor* creează și editează codul *PL/SQL* sursă (*client-side*).
- *Stored Program Unit Editor* creează și editează construcții *PL/SQL server-side*, listează mesajele erorilor generate în timpul compilării
- *Database Trigger Editor* creează și editează declanșatori bază de date.

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 mînuire 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 altei proceduri sau a unui declanșator;
- interactiv de utilizator utilizând un instrument *Oracle* (de exemplu, *iSQL\*Plus*);
- explicit dintr-o aplicație (de exemplu, *SQL\*Forms* sau utilizarea de precompilatoare).

Utilizarea (apelarea) unei proceduri se poate face:

- 1) în *iSQL\*Plus* 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 traiectoriile sale conduc la comanda *RETURN*. Dacă o traiectorie a algoritmului trimite 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 mînuire 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 încetează. 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 repetate ale aceleiași funcții, având aceleași argumente. Ea asigură folosirea unui rezultat obținut anterior.

În blocul *PL/SQL* al unei proceduri sau funcții stocate (definește 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)
  RETURN 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 *iSQL\*Plus*:

```
VARIABLE val NUMBER
EXECUTE :val := numar_opere(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_opere(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_opere(v_data);
    DBMS_OUTPUT.PUT_LINE('numarul operelor de arta
    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_opera
    (v_tip opera.tip%TYPE) AS
FUNCTION nr_opere (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
    FROM    studiaza, opera
    WHERE    studiaza.cod_opera = opera.cod_opera
    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
    FROM    figureaza_in, opera
    WHERE    figureaza_in.cod_opera = opera.cod_opera
    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 oerelor 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 expozitii este ' ||
                          nr_expozitii(v_tip));
END;

```

## Instrucțiunea *CALL*

O instrucțiune specifică pentru *Oracle9i* 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 comenzii *EXECUTE IMMEDIATE*. Pentru executarea acestei comenzi, utilizatorul trebuie să aibă privilegiul *EXECUTE* asupra subprogramului. Comanda poate fi executată interactiv din *iSQL\*Plus*. 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

SQL> PRINT v_iesire
v_iesire
-----
Salut!

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

Ca și în cazul tabelelor, funcțiile și procedurile pot fi suprimate cu ajutorul comenzii *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 șters 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 aceleași.

***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 implicite chiar în cadrul comenzii 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 referință în cazul parametrilor *IN* și este prin valoare 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 referință.

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 data zilei curente, dar care opțional 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 apeluri 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 incorect
data_mea := 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*.

- FUNCTION** alfa(par **IN** **POSITIVE**) ...;  
**FUNCTION** alfa(par **IN** **BINARY\_INTEGER**) ...;
- FUNCTION** alfa(par **IN** **NUMBER**) ...;  
**FUNCTION** alfa(parar **IN** **NUMBER**) ...;
- 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', 'impresionism');
    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;
- pot accepta parametri *NOCOPY*.

## 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 comenzile *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 șirul 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 ( ... );    -- declarație 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 comenzii *SELECT*;
- condiția clauzelor *WHERE* și *HAVING*;
- clauzele *CONNECT BY*, *START WITH*, *ORDER BY* și *GROUP BY*;
- clauza *VALUES* a comenzii *INSERT*;
- clauza *SET* a comenzii *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_medie
RETURN NUMBER IS
  v_val_mediu opera.valoare%TYPE;
BEGIN
  SELECT AVG(valoare)
  INTO    v_val_mediu
  FROM    opera;
  RETURN v_val_mediu;
END;
```

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

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

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 conține 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 comenzile *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 oricarui 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 *iSQL\*Plus*;
- *p-code* (nu este accesibil utilizatorilor);
- erorile la compilare, utilizând vizualizarea *USER\_ERRORS* din dicționarul datelor sau comanda *SHOW ERRORS* din *iSQL\*Plus*;
- 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
FROM        USER_SOURCE
WHERE       NAME = 'numar_opere'
ORDER BY    LINE;
```

**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.

<u>Obiecte dependente:</u>	<u>Obiecte referite</u>
<i>View, Table</i>	<i>Table, Secventa</i>
<i>Procedure</i>	<i>View</i>
<i>Function</i>	<i>Procedure</i>
<i>Package Specification</i>	<i>Function</i>
<i>Package Body</i>	<i>Synonym</i>
<i>Database Trigger</i>	<i>Package Specification</i>
<i>Obiect definit de utilizator</i>	<i>Obiect definit de utilizator</i>
<i>Tip colectie</i>	<i>Tip colectie</i>

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, REFENCED_NAME, REFENCED_TYPE
FROM USER_DEPENDENCIES
WHERE REFENCED_NAME IN ('opera', 'nou_opera');
```

<u>NAME</u>	<u>TYPE</u>	<u>REFERENCED_NAME</u>	<u>REFERENCED_TYPE</u>
filtru	Procedure	opera	Table
adaug	Procedure	nou_opera	View
nou_opera	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 aceleași 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#;
```

<u>NESTED_LEVEL</u>	<u>TYPE</u>	<u>NAME</u>
0	Table	opera
1	View	nou_opera
2	Procedure	adaug
1	Procedure	filtru

```
SELECT *
FROM IDEPTREE;
```

```
DEPENDENCIES
TABLE nume_schema.opera
VIEW nume_schema.nou_opera
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 recompilata) 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 schimbă, atunci executia continua.

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 stearsa;
- 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 coloana nou definita nu are restrictia NOT NULL;
- tipul coloanelor referite nu s-a schimbat;
- un tabel "private" este sters, dar exista un tabel "public" avand acelasi nume si structura;
- toate comenzile *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 in cadrul comenzii *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 distantă, atunci se face recompilare automată, dar la a doua reexecuție. Este preferabilă o recompilare manuală pentru prima reexecuție sau implementarea unei strategii de reinvocare a ei (a doua oară).
- Dacă procedura depinde de un obiect distant, dar care nu este procedură, atunci nu se face recompilare automată.

## Rutine externe

*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 eficiente 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 comenzii *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 ( $C \leftrightarrow PL/SQL$ ) 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.

*Oracle8i* 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 *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 comenzii 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 clauză trebuie specificată doar semnatura 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 corectă, iar " " implică ca stocarea este *case sensitive*, altfel implicit se depune numele cu litere mari.

Procedura poate fi apelată 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 anonime, 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 comenzii: *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 comenzile *INSERT*, *UPDATE*, *DELETE*. Totuși, pentru a realiza o reactualizare, poate fi creată o vizualizare relativă la funcția tabel și folosit un declanșator *INSTEAD OF*.

### **Exemplu:**

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

În timpul execuției paralele, fiecare instanță 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 (
    id1 NUMBER, sir VARCHAR2(20));
CREATE TYPE t_tabel AS TABLE OF t_linie;
```

```

CREATE OR REPLACE FUNCTION calc1 RETURN t_tabel
AS
    v_tabel t_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_linie t_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 specifica în partea declarativă a rutinei.

**Observații:**

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

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

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