Practică sisteme de gestiune pentru baze de date

Laborator

- **Instalare**
- **Ghid SQL*Plus**
- **SQL Developer**
- **RE-SQL**
- **Views**
- **Intro PLSQL**
- **Blocuri Anonime**
- **Variabile Operatori**
- **SELECT INTO**
- **Structuri de control**
- **Structuri iterative**
- **Cursoare**

Introducere în PL/SQL

Linkuri utile:

- **Tutorials Point - PL/SQL**

De ce PL/SQL?

Limbajul SQL este util pentru a face interogări asupra unei baze de date.

De obicei, o aplicație (PHP, Java, C# etc.) are nevoie să efectueze mai multe interogări ale bazei de date pentru a funcționa corect. Să presupunem că datele preluate de aplicația PHP necesită o anumită prelucrare **(fără intervenția utilizatorului)** și trebuie scrise înapoi în baza de date. Dacă interogările depind una de cealaltă, **timpul de transfer al datelor** între aplicația PHP și serverul SQL poate deveni prea mare. PL/SQL, fiind un limbaj de scripting executat direct pe serverul Oracle SQL, rezolvă această problemă. În plus, executarea scriptului direct pe server elimină transferul pe rețea, reducând timpul de răspuns și sporind **securitatea datelor**.

Un alt exemplu este preluarea datelor procesate într-un anumit mod din baza de date. Spre exemplu, am putea avea o interogare:

```
SELECT CMMDC(val1, val2) FROM cupluri_numere; -- CMMDC a două câmpuri val1 și val2 din tabelul cupluri_numere.
```

Funcția `CMMDC` nu este predefinită în SQL. Totuși, definind această funcție în **PL/SQL**, putem construi interogări de acest tip.

PL/SQL a fost dezvoltat de Oracle la sfârșitul anilor '80 pentru a **extinde capabilitățile SQL** fără a introduce funcții nenaturale acestui limbaj.

Ce poate defini o secvență de cod PL/SQL?

O secvență de cod PL/SQL poate defini:

- **un bloc anonim**,
- **o funcție**,
- **o procedură**,
- **un pachet**,
- **conținutul unui pachet (body)**,
- **un trigger**,
- **un tip (type)**,
- **corpul unui tip (type body)**.
- etc.

Practică sisteme de gestiune pentru baze de date

Laborator

Blocuri anonime

O **funcţie** are un nume pentru a putea fi apelată. Acestea sunt salvate în baza de date şi sunt utilizate ori de câte ori sunt apelate. **Blocurile anonime** sunt secvenţe de cod PL/SQL ce nu au un nume. Blocurile anonime pot fi executate din fişierele în care au fost scrise fără a fi reţinute de SGBD.

Un cod anonim este format din trei secţiuni:

- **Secţiunea declarativă** - în care sunt declarate variabilele împreună cu tipul lor de date şi eventual o iniţializare. Tot aici pot fi declarate constante, cursoare sau excepţii. Secţiunea declarativă începe prin cuvântul-cheie `DECLARE` .
- **Secţiunea executabilă** - conţine instrucţiuni SQL sau PL/SQL necesare pentru manipularea informaţiilor din baza de date. Secţiunea executabilă începe prin cuvântul `BEGIN` şi se termină cu `END;` .
- **Secţiunea de excepţii** - atunci când codul din secţiunea executabilă produce erori, în această secţiune pot fi scrise secvenţe de cod ce tratează aceste excepţii. Secţiunea de excepţii începe prin cuvântul-cheie `EXCEPTION` .

Dintre cele trei blocuri, singurul care este **obligatoriu** este cel din secţiunea executabilă.

Exemple

```
BEGIN
   DBMS_OUTPUT.PUT_LINE('Maine: ' || (SYSDATE+1));
END;
```

```
SET SERVEROUTPUT ON;
DECLARE
    v_mesaj VARCHAR2(50) := 'Salutare, lume!';
BEGIN
    DBMS_OUTPUT.PUT_LINE('Mesaj: ' || v_mesaj);
END;
```

La începutul blocului anonim de mai sus a fost pusă comanda `SET SERVEROUTPUT ON;` fără de care serverul SGBD va ignora comenzile de afişare.

Exemple suplimentare

```
SET SERVEROUTPUT ON;
DECLARE
    c_pi CONSTANT NUMBER := 3.1415;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Valoarea lui Pi: ' || c_pi);
END;
```

```
SET SERVEROUTPUT ON;
DECLARE
    v_index NUMBER := 1;
BEGIN
    WHILE v_index <= 5 LOOP
        DBMS_OUTPUT.PUT_LINE('Index: ' || v_index);
        v_index := v_index + 1;
    END LOOP;
END;
```

Comentarii

În toate limbajele de programare comentariile sunt binevenite: pentru a explica ce face o anumită secţiune de cod, pentru a vă aminti după un timp ce aţi vrut să faceţi într-o anumită zonă a programului sau pentru a explica altor programatori ce aţi vrut să faceţi.

Comentariile sunt ignorate de către interpretorul codului PL/SQL.

- Comentariile pe o singură linie încep cu `--` .
- Comentariile pe mai multe linii sunt delimitate de `/* ... */` .

```
-- Acesta este un comentariu pe o singură linie

/* Acesta este
   un comentariu
   pe mai multe linii */
```

Practică sisteme de gestiune pentru baze de date

Laborator

Declararea variabilelor

Sintaxa de declarare a variabilelor este:

```
Identificator [CONSTANT] tip_de_date [NOT NULL]
     [:= expresie | DEFAULT expresie]
```

Identificatorul - numele variabilei. Identificatorul trebuie să înceapă cu un caracter, după care pot urma alte litere, cifre, semnul dolar ($), diez (#) sau underscore (_). Nu trebuie să depășească 30 de caractere. Dacă numele variabilei este format din mai multe cuvinte, acestea vor fi despărțite prin underscore (_). Standardul precizează că variabilele vor începe cu **"v_"**, constantele cu **"c_"** și parametrii primiți într-o funcție sau procedură cu **"p_"**. Numele variabilelor nu este case-sensitive, dar nu se pot utiliza cuvinte cheie ale limbajului (ex.: ALL, CREATE, SELECT etc.).

Atenție: denumiți variabilele cât mai sugestiv, altfel nu vom puncta integral solutiile de la testari !

CONSTANT - cuvânt cheie utilizat pentru constante. Acestea sunt inițializate în blocul de declarare și nu pot fi modificate.

Tipul de date - poate fi scalar, compus, referință, obiect sau LOB (Large Object):

- **Scalare**: utilizate și în SQL (VARCHAR2, NUMBER etc.).
- **Compuse**: TABLE, RECORD, NESTED TABLE, VARRAY.
- **Referințe**: pointeri către zone de memorie.
- **Obiecte**: similare cu clasele din OOP.
- **LOB**: CLOB, BLOB, BFILE, NCLOB (date nestructurate până la 4GB).

Constrângerea **NOT NULL** impune ca variabila să fie inițializată și să nu aibă valoarea NULL.

:= - operatorul de atribuire. Valorile pot fi rezultate ale expresiilor. Dacă nu este specificată o valoare, variabila va avea valoarea NULL. Alternativ, se poate utiliza DEFAULT.

Exemple de declarații:

```
v_cartofi NUMBER := 2E4;
v_varsta INTEGER := &i_varsta;
v_nume_student VARCHAR2(20) := &i_nume;
v_nume_film VARCHAR2(30) := 'The Matrix';
c_pi CONSTANT DOUBLE PRECISION := 3.141592653;
v_salut VARCHAR2(40) DEFAULT 'Bine ați venit!';
v_data_de_nastere DATE;
v_numar_studenti NUMBER(3) NOT NULL := 1;
v_promovam_la_PSGBD BOOLEAN DEFAULT TRUE;
```

Recomandare: Declarați variabilele pe linii separate pentru claritate.

Domeniul de vizibilitate (Scope)

Variabilele declarate în secțiunea **DECLARE** sunt globale. Variabilele declarate în blocuri interne sunt locale și accesibile doar în acel bloc.

```
DECLARE
   v_nume VARCHAR2(20) := 'Cristi';
   v_varsta INTEGER := 21;
BEGIN
   DBMS_OUTPUT.PUT_LINE(v_nume);
   DECLARE
      v_nume NUMBER(3) := 5;
   BEGIN
      DBMS_OUTPUT.PUT_LINE(v_nume); -- 5
      DBMS_OUTPUT.PUT_LINE(v_varsta); -- 21
   END;
   DBMS_OUTPUT.PUT_LINE(v_nume); -- Cristi
END;
```

Pentru a accesa variabilele globale în blocuri interne, se utilizează eticheta:

```
<global>
DECLARE
   v_nume VARCHAR2(20) := 'Cristi';
BEGIN
   DECLARE
      v_nume NUMBER(3) := 5;
   BEGIN
      DBMS_OUTPUT.PUT_LINE(global.v_nume); -- Cristi
   END;
END;
```

Operatori

Operatorii disponibili sunt:

- **Aritmetici**: +, -, *, /, **
- **Relaționali**: !=, <>, ~=, <,>, <=, >=, =
- **Comparație**: LIKE, BETWEEN, IN, IS NULL
- **Logici**: AND, OR, NOT
- **Șiruri de caractere**

Exemplu operatori aritmetici:

```
DECLARE
   a NUMBER := 10;
   b NUMBER := 4;
BEGIN
   DBMS_OUTPUT.PUT_LINE(a + b);
   DBMS_OUTPUT.PUT_LINE(a ** b);
END;
```

Exemplu operatori relaționali:

```
DECLARE
   a NUMBER := 10;
   b NUMBER := 4;
BEGIN
   IF a > b THEN
      DBMS_OUTPUT.PUT_LINE('a > b');
   END IF;
END;
```

Exemplu operatori de comparație:

```
DECLARE
   a NUMBER := 50;
BEGIN
   IF a BETWEEN 20 AND 80 THEN
      DBMS_OUTPUT.PUT_LINE('a este între 20 și 80');
   END IF;
END;
```

Precedența operatorilor urmează regulile standard din alte limbaje de programare.

Funcții predefinite

Funcțiile predefinite pentru anumite tipuri de date le-ați învățat deja în timpul laboratorului de SQL.

Un mic review (care nu conține toate funcțiile):

Funcții pentru șiruri de caractere: **ASCII, LENGTH, RPAD, CHR, LOWER, RTRIM, CONCAT, LPAD, SUBSTR, INITCAP, LTRIM, TRIM, INSTR, REPLACE, UPPER**.

Funcții pentru numere: **ABS, EXP, ROUND, ACOS, LN, SIGN, ASIN, LOG, SIN, ATAN, MOD, TAN, COS, POWER, TRUNC**.

Funcții pentru variabile de tip **DATE**: **ADD_MONTHS, MONTHS_BETWEEN, CURRENT_DATE, ROUND, CURRENT_TIMESTAMP, SYSDATE, LAST_DAY, TRUNC**.

Conversii de date

Se face automat conversia între **varchar2** și toate celelalte tipuri de date, de la o variabilă de tip **DATE**, **NUMBER** sau **PLS_INTEGER** la **LONG** (dar nu și invers: **LONG** este mai mare, poate ține minte numere mai mari care s-ar pierde dacă ar fi convertite în **NUMBER**, spre exemplu). De la **NUMBER** la **PLS_INTEGER** și invers. **DATE** nu poate fi convertit automat în **NUMBER** sau **PLS_INTEGER**.

Atenție: conversiile implicite pot fi mai lente și pot depinde de sistemul de operare (de exemplu, **DATE** poate fi convertit în **varchar2** în feluri diferite dacă este aleasă conversia implicită). Codul ce utilizează conversiile implicite este mai greu de înțeles.

Conversiile explicite de date se realizează cu următoarele funcții: **TO_NUMBER(), ROWIDTONCHAR(), TO_CHAR(), HEXTORAW(), TO_CLOB(), RAWTOHEX(), CHARTOROWID(), RAWTONHEX(), ROWIDTOCHAR(), TO_DATE()**.

De exemplu:

```
BEGIN
     DBMS_OUTPUT.PUT_LINE(TO_CHAR(SYSDATE,'Month DD, YYYY'));
END;
```

Practică sisteme de gestiune pentru baze de date

Laborator

Preluarea unei singure valori dintr-un tabel

Pentru a prelua informații dintr-un tabel se folosește comanda **SELECT** (specifică limbajului SQL). Există câteva posibilități de răspuns (în afară de cazurile în care vă dă o eroare pentru că ați scris greșit comanda sau nu există tabelul).

- Sunt returnate mai multe rânduri: pentru a le prelua unul câte unul și pentru a le procesa, se vor folosi **cursoare** despre care veți învăța în laboratorul următor.
- Nu este returnată nici o înregistrare: caz acoperit de asemenea de laboratorul următor.
- Este returnat un singur rând - acest caz îl vom discuta în continuare.

Dacă este întors un singur rând, valorile acestuia pot fi stocate în **variabilele existente** în blocul anonim, suprascriindu-se valorile deja existente.

Pentru a face această operație, comanda **SELECT** trebuie modificată pentru a include și numele variabilelor în care se face stocarea. Se va utiliza așadar cuvântul cheie **INTO** pentru a face o asociere dintre valorile returnate și variabilele în care acestea vor fi stocate.

Exemplu:

```
SELECT nume INTO v_nume FROM studenti WHERE ROWNUM=1
```

În exemplul de mai sus s-a stocat numele primului student în variabila **v_nume**. Pentru a ne asigura că nu sunt returnate mai multe rânduri, s-a folosit clauza **ROWNUM=1**.

Dacă se folosește o funcție de agregare peste întreaga tabelă, puteți fi siguri că aceasta va returna doar o singură valoare. De exemplu, următoarea instrucțiune este corectă:

```
SELECT AVG(valoare) INTO v_medie FROM note
```

Dacă doriți să preluați simultan mai multe valori din rândul returnat, câmpurile vor fi delimitate cu virgulă și la fel vor fi delimitate și numele variabilelor în care vor fi introduse valorile:

```
SELECT nume, prenume INTO v_nume, v_prenume FROM studenti WHERE ROWNUM=1
                            --se preiau valorile a două câmpuri în două variabile diferite
```

Dacă răspunsul este format dintr-o singură înregistrare, folosiți mereu SELECT... INTO...

Puteți selecta și toate coloanele unui rând în acest mod, dar despre cum se face acest lucru vom discuta într-un laborator viitor.

Declararea variabilelor utilizând %TYPE

Așa cum am văzut, pentru a prelua o valoare dintr-un tabel, va fi utilizată o comandă de tip **SELECT** puțin modificată pentru a specifica și variabila în care se dorește preluarea acelei informații.

Spre exemplu, pentru a prelua nota maximă avută de un student în variabila **v_valoare_nota_maxima**, vom utiliza următorul **SELECT**:

```
SELECT MAX(valoare) INTO v_valoare_nota_maxima FROM note;
```

Problema pe care o avem totuși este tipul de date returnat de comanda **SELECT**. Este el de același tip cu cel al variabilei **v_valoare_nota_maxima**? (De exemplu, dacă variabila a fost declarată ca fiind una de tip întreg și valoarea returnată de interogare este una reală, atunci putem avea pierderi de informații). Pentru a ne asigura că tipul variabilei este același cu cel al câmpului valoare din tabela note, putem utiliza o construcție de tipul **%TYPE**. Într-o astfel de declarație, tipul variabilei nu este cunoscut a-priori, ci va avea același tip cu tipul declarat pentru coloana referită.

Declarația este așadar:

```
v_valoare_nota_maxima note.valoare%TYPE;
```

iar codul anonim scris în întregime:

```
DECLARE
   v_valoare_nota_maxima note.valoare%TYPE;
BEGIN
   SELECT MAX(valoare) INTO v_valoare_nota_maxima FROM note;
   DBMS_OUTPUT.PUT_LINE('Nota maximă: ' || v_valoare_nota_maxima);
END;
```

Utilizarea lui **%TYPE** mai are un avantaj: dacă la un moment dat modificați tipul datelor din tabelă (pentru că, spre exemplu, ați ajuns la un număr de 999 de utilizatori și inițial ați declarat câmpul user_id ca având maxim 3 cifre), atunci codurile PL/SQL vor fi în continuare funcționale. În cazul în care codul PL/SQL nu mai este valid după modificarea în tabelă (de exemplu, pentru că se aștepta ca toate ID-urile să fie mai mici decât 999), acesta va semnala o eroare.

Atunci când utilizați **%TYPE**, este bine ca variabila să fie prefixată cu numele tabelei și al coloanei (vă va fi mai ușor să o urmăriți în cod).

Puteți declara o variabilă pentru a avea același tip cu cel avut de altă variabilă.

De exemplu, pentru a prelua valoarea minimă a notei, se poate declara o variabilă **v_valoare_minima_nota** în felul următor:

```
v_valoare_nota_minima v_valoare_nota_maxima%TYPE;
```

Rescrieți codul pentru a fi afișate atât nota minimă, cât și cea maximă.

Utilizarea lui %TYPE este probabil cea mai importantă recomandare din acest laborator.... Declarați variabilele în acest mod ori de câte ori aveți de gând să selectați o valoare dintr-o coloană!

Exercițiu

Să se construiască un script la a cărui rulare utilizatorul să fie întrebat un nume (de familie). Dacă în tabela studenți nu există nici măcar un student cu acel nume, se va afișa un mesaj informativ asupra acestui fapt. În caz că în tabela studenți se găsește măcar un student având acel nume, atunci scriptul va afișa următoarele (pe câte un rând):

- Numărul de studenți având acel nume de familie;
- ID-ul și prenumele studentului care ar fi primul în ordine lexicografică (având același nume, veți ordona după prenume);
- Nota cea mai mică și cea mai mare a studentului de la punctul precedent;
- Numărul A la puterea B unde A este nota cea mai mare și B este nota cea mai mică a studentului.

Practică sisteme de gestiune pentru baze de date

Laborator

Structuri de control

Pentru ca un limbaj să fie considerat limbaj de programare, pe lângă operaţia de atribuire trebuie să permită evaluarea de condiţii (şi executarea unui cod diferit în funcţie de rezultat) şi crearea de bucle.

Structuri de control conditionale

În PL/SQL verificarea condiţiilor (pot fi utilizate şi funcţii care intorc o valoarea booleană - ca de exemplu between, like, etc.) este realizată de comanda **`IF - THEN - ELSE`** având următoarea structură:

```
IF condiție THEN
         secvență de instrucțiuni ce vor fi rulate în cazul în care condiția este îndeplinită;
   ELSE
         secvență de instrucțiuni ce vor fi executate în cazul în care condiția nu este îndeplinită;
END IF;
```

La fel ca şi în alte limbaje de programare, secţiunea **`ELSE`** împreună cu blocul de instrucţiuni arondat ei pot să lipsească.

Iata un exemplu al utilizării instrucţiunii **`IF-THEN-ELSE`** :

```
set serveroutput on;
accept i_numar prompt "Please enter your number: ";
DECLARE
   v_numar NUMBER(5);
   i_numar NUMBER(5);
BEGIN
   v_numar := &i_numar;
   IF (v_numar MOD 2 = 0)
      THEN
        DBMS_OUTPUT.PUT_LINE('Numarul '||v_numar||' este par.');
      ELSE
        DBMS_OUTPUT.PUT_LINE('Numarul '||v_numar||' este impar.');
   END IF;
END;
```

Dacă în secţiunea **`ELSE`** se doreşte testarea unei noi condiţii se poate utiliza comanda **`ELSIF`** (care la rândul ei va fi urmată de o secţiune **`THEN`** şi apoi de una **`ELSE/ELSIF`**). Atunci când sunt utilizate mai multe secţiuni de tipul **`ELSIF`** (atenţie, nu **`ELSEIF`**), în final se va pune o singură dată comanda ce identifică încheierea ultimului bloc (**`END IF`**). Spre exemplu, dacă dorim să testăm apartenenţa unui număr la un anumit interval de numere putem recurge la codul:

```
DECLARE
   v_numar NUMBER(5) := 50;
BEGIN
   IF (v_numar < 10)
       THEN
           DBMS_OUTPUT.PUT_LINE('Numarul este mai mic decat 10');
       ELSIF (v_numar > 80) THEN
           DBMS_OUTPUT.PUT_LINE('Numarul este mai mare decat 80');
       ELSE
           DBMS_OUTPUT.PUT_LINE('Numarul este in intervalul [10,80]');
   END IF;
END;
```

În cazul în care valoarea unei variabile este comparată cu mai multe valori posibile, se poate utiliza instrucţiunea CASE. Pe lângă compararea variabilei cu valorile predefinite, instrucţiunea de control **CASE** permite şi executarea unui cod default în cazul în care variabila nu are nici una dintre valorile aşteptate:

```
DECLARE
   numar NUMBER := 5;
BEGIN
   CASE (numar)
       WHEN 1 THEN DBMS_OUTPUT.PUT_LINE('Primul numar natural nenul.');
       WHEN 2 THEN DBMS_OUTPUT.PUT_LINE('Primul numar natural par.');
       ELSE
           DBMS_OUTPUT.PUT_LINE('Un numar mai mare sau egal cu 3');
   END CASE;
END;
```

Vă aminitiţi ce facea funcţia **DECODE** în SQL ? Această funcţie poate fi simulată cu o instrucţiune de tip **CASE** . Executaţi următoarea comandă **SELECT** :

```
SELECT nume, prenume,
CASE bursa
   WHEN 450 THEN 'Patru sute si cinci zeci'
   WHEN 350 THEN 'Trei sute si cinci zeci'
   WHEN 250 THEN 'Doua sute si cinci zeci'
   ELSE 'Fara bursa'
END
FROM studenti;
```

Comanda **CASE** poate să nu aibă un operand iniţial caz în care valorile ce urmează după **WHEN** trebuie să fie de tip boolean:

```
SELECT nume, prenume,
CASE
   WHEN bursa > 300 THEN 'Bogat'
   WHEN bursa > 500 THEN 'Foarte Bogat'
   ELSE 'Sarac'
END
FROM studenti;
```

Practică sisteme de gestiune pentru baze de date

Laborator

Structuri iterative în PLSQL

Buclele în PLSQL pot fi cu condiţie iniţială, cu condiţie finală şi care se vor repeta de un anumit număr de paşi (de tip for). Bucla cu condiţie iniţială este realizată prin utilizarea `WHILE(condiție) ... END LOOP` , cea cu condiţie finală poate fi obţinută prin `LOOP ... END LOOP` (de fapt va produce o buclă infinită din care se poate ieşi cu o instrucţiune te tip `EXIT` iar bucla ce se va repeta de un anumit număr de paşi se realizează cu instrucţiunea `FOR contor IN initial..final LOOP ... END LOOP;` . Iată câteva exemple:

Primul exemplu va fi pentru bucla de tip `WHILE` :

```
set serveroutput on;
DECLARE
   v_contor INTEGER := 0;
BEGIN
   WHILE (v_contor < 10) LOOP
       v_contor := v_contor + 1;
       DBMS_OUTPUT.PUT_LINE(v_contor);
   END LOOP;
END;
```

Exemplul adaptat pentru structura `LOOP` împreună cu comanda de ieşire din buclă (comanda `EXIT` poate fi aplicată pentru orice structură repetitivă şi are forma `EXIT WHEN` (condiţie) sau `EXIT etichetă WHEN` (condiţie) atunci când sunt utilizate mai multe cicluri impricate şi se doreşte ieşirea dintr-un anumit ciclu - de exemplu cel exterior.):

```
set serveroutput on;
DECLARE
   v_contor INTEGER := 0;
BEGIN
   LOOP
       v_contor := v_contor + 1;
       DBMS_OUTPUT.PUT_LINE(v_contor);
       EXIT WHEN v_contor = 10;
   END LOOP;
END;
```

Ultima metodă de a realiza o structură repetitivă în PLSQL este utilizând comanda `FOR` . Un exemplu în acest sens:

```
set serveroutput on;
DECLARE
   v_contor INTEGER := 0;
BEGIN
   FOR v_contor IN 1..10 LOOP
       DBMS_OUTPUT.PUT_LINE(v_contor);
   END LOOP;
END;
```

În cazul în care se doreşte parcurgerea în sens invers a valorilor din intervalul precizat, de va utiliza cuvântul **`REVERSE`** imediat după utilizarea lui **`IN`** (e.g. **`FOR v_contor IN REVERSE 1..10 LOOP`**).

Comanda **`EXIT`** poate fi invocată oricând în interiorul unei structuri repetitive pentru a forţa ieşirea din aceasta. În cazul în care doua structuri repetitive sunt imbricate şi se doreşte ieşirea din ambele structuri, prima structură repetitivă va fi etichetată:

```
set serveroutput on;
DECLARE
   v_contor1 INTEGER;
   v_contor2 INTEGER;
BEGIN
   <<eticheta>>
   FOR v_contor1 IN 1..5 LOOP
       FOR v_contor2 IN 10..20 LOOP
           DBMS_OUTPUT.PUT_LINE(v_contor1||'-'||v_contor2);
           EXIT eticheta WHEN ((v_contor1=3) AND (v_contor2=17));
       END LOOP;
   END LOOP;
END;
```

Comanda **`CONTINUE`** are rolul de a sări peste restul codului rămas în buclă şi de a începe o nouă iteraţie. De exemplu, în codul de mai sus, dacă dorim să nu fie afişată linia în care apare combinaţia 3-13, putem să introducem înaintea liniei de afişare următoarea linie:

```
CONTINUE WHEN ((v_contor1=3) AND (v_contor2=13));
```

Puteţi face în aşa fel încât să nu fie afişată ultima linie ? Cea cu combinaţia 3-17 ?

O ultimă comandă ce poate afecta o buclă este **`GOTO`** . Aceasta face saltul la o anumită etichetă. Se poate iesi forţat din buclă utilizând **`GOTO`** , urmatorul exemplu este doar un simplu salt

```
set serveroutput on;
BEGIN
   GOTO eticheta;
   DBMS_OUTPUT.PUT_LINE('Nu se va afisa.');
   <<eticheta>>
   DBMS_OUTPUT.PUT_LINE('Se va afisa.');
END;
```

Puteţi utiliza două instrucţiuni **`GOTO`** pentru a emula o structură repetitivă ?

Practică sisteme de gestiune pentru baze de date

Laborator

Utilizarea cursoarelor

Memoria în care este păstrată o interogare împreună cu datele pe care le utilizează se numeşte cursor. Cursorul poate fi implicit (utilizatorul nu are acces la el - de exemplu nu ştiţi ce se petrece în memoria serverului atunci când executaţi o comanda SQL) sau explicit (care este definit şi utilizat de către un programator PL/SQL special pentru a parcurge datele selectate una câte una sau pentru a le manipula).

Cursoare implicite

Deşi programatorul nu are acces direct la zona de memorie în care se află un cursor implicit, acesta poate obţine anumite informaţii despre execuţia interogării (relative la numărul de rânduri procesate) prin intermediul a trei atribute specifice. Cele trei atribute ale unui cursor implicit ce pot fi utilizate de programator sunt **SQL%FOUND**, **SQL%NOTFOUND**, **SQL%ROWCOUNT**.

Să considerăm următorul cod PL/SQL care modifică bursa tutror studenţilor cu o bursa având valoarea mai mare decât 300 (şi îi adaugă 10 Ron):

```
DECLARE
   v_randuri INTEGER;
BEGIN
   UPDATE studenti set bursa = bursa + 10 WHERE bursa > 300;
   IF(SQL%FOUND)
      THEN
         DBMS_OUTPUT.PUT_LINE('Am marit bursa la ' || SQL%ROWCOUNT || ' studenti.');
      ELSE
         DBMS_OUTPUT.PUT_LINE('Nimanui nu i s-a marit bursa.');
    END IF;
END;
```

Cursoare explicite

Cursoarele explicite sunt declarate şi utilizate în scripturile PL/SQL. Ele sunt utilizate atunci când interogările pe care le efectuăm asupra bazei de date vor returna mai mult de un singur rând (altfel valorile ar putea fi reţinute în variabile şi utilizate în acest mod) şi permit procesarea informaţiilor din rezultat linie cu linie.

Paşii pe care trebuie să îi urmăm când utilizăm un cursor sunt: declararea, deschiderea, preluarea de linii, închiderea cursorului.

Un cursor explicit se declară în zona de declaraţii a scriptului PL/SQL (deşi cursoare explicite pot fi utilizate şi direct în **FOR**). Pentru declarare se va utiliza următorul format:

```
DECLARE
   CURSOR   nume_cursor  IS  comanda_select;
```

În această construcţie, comanda_select este orice comandă de tip SELECT (poate conţine JOIN, GROUP BY, ORDER BY etc.). In cazul in care comanda select contine variabile PL/SQL acestea vor fi declarate înaintea utilizării lor în definiţia cursorului.

Deschiderea cursorului se realizează in secţiunea de cod aflată între comenzile **BEGIN** şi **END** . Acest lucru se face cu comanda **OPEN** urmată de numele cursorului şi are ca efect alocarea de memorie pentru datele ce vor fi selectate de comanda select, executarea comenzii select şi introducerea datelor în memorie, poziţionarea pointerului pe primul rând returnat.

În continuare, într-o secţiune **LOOP** (pentru că avem mai multe linii şi vrem ca fiecare să fie prelucrată individual), se apelează comanda **FETCH** urmată de numele cursorului, de cuvântul cheie **INTO** şi apoi de variabilele ce vor reţine valorile. Executarea comenzii **FETCH** va avea ca efect (pe lângă atribuirea variabilelor cu valorile din cursor) trecerea la următoarea linie returnată de comanda select. Din categoria "Bune practici",înainte de a intra în **LOOP** sau de a face primul fetch, aţi putea testa dacă selectul a returnat măcar o linie.

```
FECTH nume_cursor INTO v_var1, v_var2, v_var3;
```

Pentru a testa daca s-a ajuns la ultimul rând (util pentru a ieşi din buclă) se va utiliza atributul **NOTFOUND** specific cursoarelor explicite. Valoarea atributului **nume_cursor%NOTFOUND** este aşadar true atunci când operaţia **FETCH** nu a mai fost capabilă să returneze informaţii din cursor. Acesta este, probabil, momentul în care se doreşte părăsirea buclei:

În final, după ce s-a ieşit din buclă, cursorul va fi închis prin executarea comenzii **CLOSE** urmată de numele cursorului. Dacă se va utiliza din nou comanda **OPEN** , cursorul va fi redeschis, comanda select va fi re-executată şi va fi iarăşi poziţionat pe prima linie returnată de comanda select utilizată în declararea cursorului.

În afara atributului **%NOTFOUND** , în cazul cursoarelor explicite puteţi utiliza **%ISOPEN** , **%FOUND** , **%ROWCOUNT** . Vă lăsăm dvs. plăcerea de a descoperi rolul fiecăruia (tip: afişaţi **ROWCOUNT** în interiorul buclei).

Un exemplu în acest sens (care afişează lista studenţilor bursieri) este următorul:

```
DECLARE
   CURSOR lista_studenti_bursieri  IS
       SELECT nume, prenume FROM studenti WHERE bursa IS NOT NULL;
   v_nume studenti.nume%type;
   v_prenume studenti.prenume%type;
BEGIN
    OPEN lista_studenti_bursieri;
    LOOP
        FETCH lista_studenti_bursieri INTO v_nume, v_prenume;
        EXIT WHEN lista_studenti_bursieri%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_nume||' '|| v_prenume);
    END LOOP;
    CLOSE lista_studenti_bursieri;
END;
```

Evident că lista studenţilor bursieri putea fi afişată cu un simplu SELECT; daca doriti sa folositi datele in interiorul scriptului atunci trebuie sa folositi, totusi, un cursor.

Atunci când puteţi să evitaţi cursoarele explicite faceţi acest lucru. De exemplu, nu este nevoie să utilizaţi un cursor dacă doriţi să preluaţi fiecare linie dintr-un tabel (studenti) şi apoi să introduceţi anumite informaţii în alt tabel. Următorul cod populează o tabelă denumită indivizi cu toate persoanele existente în facultate fără a utiliza cursoare sau variabile.

```
DROP TABLE indivizi;
CREATE TABLE indivizi(nume Varchar2(10), prenume varchar2(10));
BEGIN
   INSERT INTO indivizi SELECT upper(nume), prenume FROM studenti;
   INSERT INTO Indivizi SELECT upper(nume), prenume FROM profesori;
END;
```

Dacă selectul pe care l-am utilizat pentru a defini cursorul returnează foarte multe coloane, este anevoios să construim variabile pentru fiecare câmp ca apoi să le procesăm. În aceste cazuri se poate utiliza o variabilă în care se va încărca o întreagă linie. Variabila trebuie să fie declarată având ca tip **nume_cursor%ROWTYPE** şi pentru a accesa un anumit câmp returnat în această variabilă de va utiliza ".câmp". De exemplu, deşi în următorul cursor conţine toate informaţiile din tabela studenţi, putem să preluăm rând cu rând şi să afişăm doar numele studentului şi data sa de naştere:

```
DECLARE
   CURSOR lista_studenti  IS
       SELECT * FROM studenti;
   v_std_linie lista_studenti%ROWTYPE;
BEGIN
    OPEN lista_studenti;
    LOOP
        FETCH lista_studenti INTO v_std_linie;
        EXIT WHEN lista_studenti%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_std_linie.nume||' '|| v_std_linie.data_nastere);
    END LOOP;
    CLOSE lista_studenti;
END;
```

Cursoarele explicite sunt foarte uşor de utilizat în bucle de tip **FOR** . Acestea vor deschide automat cursorul şi la fiecare iteraţie vor sări la următoarea linie şi vor face şi operaţia de **FETCH** într-o variabilă ce apare în comanda **FOR** . Câmpurile din această linie pot fi utilizate în aceeaşi manieră. Iată un exemplu (codul anterior rescris cu **FOR**):

```
DECLARE
   CURSOR lista_studenti  IS
       SELECT * FROM studenti;
BEGIN
    FOR v_std_linie IN lista_studenti LOOP
        DBMS_OUTPUT.PUT_LINE(v_std_linie.nume||' '|| v_std_linie.data_nastere);
    END LOOP;
END;
```

Observaţi că: Nu mai avem nici una dintre operaţiile **OPEN** , **FETCH** , **CLOSE** . Nu am declarat (în zona de declaraţii) variabila **%ROWTYPE** **v_std_linie** ci aceasta este utilizată direct din structura **FOR** (declarare implicită). Nu a fost nevoie să scriem o instrucţiune de tip **EXIT** din buclă, ieşirea se face automat după ce au fost procesate toate rândurile.

Se poate chiar şi mai mult de atât: puteţi să nu declaraţi deloc cursorul ci să treceţi direct comanda de tip **SELECT** în **FOR** . Aceasta va fi pusă totuşi între paranteze rotunde:

```
BEGIN
    FOR v_std_linie IN  (SELECT * FROM studenti) LOOP
        DBMS_OUTPUT.PUT_LINE(v_std_linie.nume||' '|| v_std_linie.data_nastere);
    END LOOP;
END;
```

Un cursor explicit poate fi declarat utilizând parametri (de exemplu din cauză ca am dori ca să deschidem cursorul după ce am calculat o anumită valoare şi această valoare să fie utilizată de către comanda select). Pentru aceasta vom utiliza un Cursor explicit cu parametru. Diferenţa este că după numele cursorului vor fi trecuţi între paranteze rotunde parametri şi tipul lor despărţiţi prin caracterul virgulă, aceşti parametri vor fi utilizaţi efectiv în comanda **SELECT** iar la deschiderea cursorului se vor da tot între paranteze rotunde valorile ce vor fi asociate fiecărui parametru (în ordinea în care aceştia apar). Iată un exemplu care afişează studenţii cu o bursă mai mare de 300 şăi care sunt în anul 2:

```
DECLARE
   CURSOR lista_studenti_bursieri (p_bursa studenti.bursa%type, p_an studenti.an%type) IS
       SELECT nume, prenume FROM studenti WHERE bursa > p_bursa AND an > p_an;
   v_std_linie lista_studenti_bursieri%ROWTYPE;
BEGIN
    OPEN lista_studenti_bursieri (300,2); -- aceste valori pot fi calculate de codul PLSQL
    LOOP
        FETCH lista_studenti_bursieri INTO v_std_linie;
        EXIT WHEN lista_studenti_bursieri%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_std_linie.nume||' '|| v_std_linie.prenume);
    END LOOP;
    CLOSE lista_studenti_bursieri;
END;
```

Declararea cursoarelor explicite cu clauza FOR UPDATE

Dacă în timpul procesării cursorului dorim să modificăm o linie (sau chiar să o ştergem), după declararea cursorului se vor adăuga cuvintele " **FOR UPDATE** ". Acest lucru va bloca accesul altor utilizatori ce doresc să scrie în tabela (sau tabelele) selectată de cursor dar va permite totuşi citirea acestor informaţii. După cuvintele cheie **FOR UPDATE** poate fi specificat un anumit câmp ce va poate fi modificat prin adăugarea cuvnântului cheie OF urmat de numele coloanei ce va ar putea fi modificată (în acest caz doar acea coloană va fi blocată).

În cazul în care se doreşte modificarea prin intermediul unui cursor a bazei de date şi altcineva deja are un cursor pentru modificare deschis pe aceeaşi tabelă, se poate specifica (în definiţia cursorului) un număr de secunde ce vor fi aşteptate după care se va reîncerca deschiderea cursorului. Dacă şi a doua oară eşuează, va fi returnată o eroare. Pentru a specifica numărul de secunde se adaugă la sfârşit cuvântul cheie **WAIT** urmat de numărul de secunde ce poate fi aşteptat. În cazul în care nu se doreşte aşteptarea, acest lucru va fi specificat prin prezenţa cuvântului cheie **NOWAIT** (în loc de " **WAIT n** ").

Pentru a afecta rândul curent, după comanda ce indică modifcarea coloanei (sau ştergerea întregului rând) se vor adăuga cuvintele cheie " **WHERE CURRENT OF** " urmate de numele cursorului în care se face updateul.

Iată, în continuare, un script pe care toţi studenţii care au avut vreodată "probleme" la vreo materie şi-ar dori să îl execute pe serverul în care le sunt reţinute notele:

```
DECLARE
   CURSOR update_note IS
      SELECT * FROM note FOR UPDATE OF valoare NOWAIT;
BEGIN
   FOR v_linie IN update_note LOOP
       IF (v_linie.valoare < 5)
            THEN
                 UPDATE note SET valoare=5 WHERE CURRENT OF update_note;
       END IF;
   END LOOP;
END;
```

Observaţie: când facem updateul, acesta se realizează în tabela note şi nu în cursorul update_note.

Practică sisteme de gestiune pentru baze de date

Laborator

Subprograme în PLSQL

Stocarea subprogramelor

În primul rând (şi ca un failsafe - să nu ziceţi că nu vi s-a zis), subprogramele pe care le scrieţi în PLSQL, spre deosebire de codurile anonime, sunt stocate în baza de date - din acest motiv se mai numesc şi "proceduri stocate". Dacă lucraţi temele ce implică proceduri stocate pe calculatoare publice (cum sunt cele din laborator) nu uitaţi să le ştergeţi (şi din baza de date) după ce aţi terminat.

Spre exemplu, scrieţi în SQL developer următoarea procedură stocată:

```
CREATE OR REPLACE PROCEDURE afiseaza AS
   my_name varchar2(20):='Gigel';
BEGIN
   DBMS_OUTPUT.PUT_LINE('Ma cheama ' || my_name);
END afiseaza;
```

Selectaţi codul şi "rulaţi" procedura. De fapt aceasta va fi verificată sintactic după care codul ei va fi memorat în baza de date în cazul în care verificarea nu identifică nici o problemă (codul nu va mai fi recompilat şi atunci când se doreşte executarea procedurii). Tot în SQL Developer, în coloana din partea stângă, unde aveţi şi lista tabelelor, aveţi o secţiune denumită "Procedures". Procedura pe care tocmai aţi compilat-o va putea fi regăsită în acea secţiune (chiar şi codul sursă este vizibil).

Procedurile pot fi apelate numai din cadrul unui cod PL/SQL (funcţiile vor putea fi apelate şi dintr-o comandă de tip select spre exemplu):

```
set serveroutput on;
BEGIN
   afiseaza();
end;
```

Pentru a şterge procedura creată anterior, executaţi următoarea comandă:

```
DROP PROCEDURE afiseaza;
```

Codul sursă al procedurilor pe care le-aţi scris este introdus în tabela **USER_SOURCE**. Aflaţi structura acestei tabele şi încercaţi să vă găsiţi procedura pe care aţi compilat-o. Pentru a vedea procedura în SQL Plus puteţi executa comanda:

```
SELECT text FROM user_source WHERE LOWER(name) LIKE 'afiseaza';
```

Pe lângă tabela **USER_SOURCE**, alte două tabele interesante sunt **USER_OBJECTS** şi **USER_PROCEDURES**. Deocamdată vă lăsăm pe voi să descoperiţi ce informaţii sunt stocate în acestea (vom reveni ulterior asupra subiectului).

Deci: **NU UITATI SA STERGETI CE ATI LUCRAT DE PE CALCULATOARELE DIN LABORATOARE.**

Există două tipuri de subprograme pe care le puteţi construi: **proceduri** şi **funcţii**.

Proceduri

Blocurile pe care le-aţi executat până acum au fost denumite blocuri anonime - tocmai pentru că nu au un nume cu care să poată fi apelate. Blocurile anonime nu sunt stocate în tabela **USER_SOURCE** pentru că nu au un nume şi nimeni nu ar putea sa le refolosească.

În procedurile anonime exista o secţiune `DECLARE` în care erau precizate variabilele ce sunt utilizate în blocul anonim. Această secţiune era opţională și, în cazul în care nu era nevoie de variabile, şi cuvântul `DECLARE` putea fi omis (vezi exemplul de mai sus în care am apelat procedura "afiseaza").

Pentru a declara o procedură se va utiliza următoarea sintaxă:

```
CREATE [OR REPLACE] PROCEDURE nume_procedura [(parametru 1 [mod] tip_de data1, parametru 2 [mod] tip_de data2,...
parametru N [mod] tip_de dataN)] AS|IS
[var1 tip1;
etc.]
BEGIN
    ..... -- cod ce se va executa la apelul procedurii
END [nume_procedura];
```

După `CREATE` urmează, opţional cuvintele `OR REPLACE` . Aceste cuvinte au rolul de a înlocui procedura în cazul în care aceasta există deja în baza de date. Aţi putea face de fiecare data `DROP` la procedură după care să o construiţi din nou cu `CREATE` dar acest lucru ar fi mai anevoios.

Cuvântul `PROCEDURE` indică tipul de subprogram ce este construit (vom construi şi funcţii care vor avea cuvântul `FUNCTION` ca şi tip al subprogramului). Numele prin care va fi apelată procedura este dat în continuare în locul parametrului **nume_procedura** din sintaxa de mai sus.

O procedură poate avea mai mulţi parametri de intrare, mai multe valori de ieşire sau parametri care pot fi modificaţi în interiorul procedurii (care sunt în acelaşi timp şi de intrare şi de ieşire). Aceştia sunt daţi între paranteze rotunde, despărţiţi prin virgulă. Secţiunea opţională "mod" din definiţia sintactică de mai sus poate avea oricare din valorile `IN` , `OUT` , `IN OUT` (dacă modul lipseşte, varianta predefinită este `IN`). Atunci când se apelează procedura, trebuie ca pe poziţiile în care se află valori de ieşire (identificate prin `OUT`) să fie neapărat variabile de acelaşi tip cu cele declarate în definiţia procedurii. Aceste variabile vor primi după apel valorile variabilelor respective din procedură. Tipul variabilelor poate fi definit şi cu `%TYPE` . Recomandare: prefixaţi parametrii (de intrare sau de ieşire cu `p_`).

La sfârşitul declaraţiei procedurii se poate afla oricare dintre cuvintele `AS` sau `IS` . După acestea urmează variabilele (fără a mai fi nevoie de cuvântul `DECLARE`) şi, obligatoriu, codul executabil al procedurii (ce poate conţine şi o secţiune de tratare a excepţiilor).

O procedură poate fi apelată dintr-un bloc anonim, din altă procedură sau dintr-o aplicaţie ce poate interacţiona cu severul (de exemplu dintr-o aplicaţie PHP sau JAVA). O procedură nu poate fi apelată dintr-o comandă de tip `SELECT` .

Parametrii care sunt definiţi în antetul procedurii se numesc parametri formali (vor fi utilizaţi în subprogram şi de este recomandat să îi prefixaţi cu `p_`) iar parametrii cu care este apelată procedura (valori sau variabile) se numesc parametrii actuali. Când procedura este apelată, fiecarui parametru formal îi va fi atribuită valoarea parametrului actual din apel, în ordine: primului parametru formal îi va fi atribuită valoarea primului parametru actual etc.

În continuare iată un exemplu de procedură care incrementează valoarea primită ca parametru. Variabila care a fost trimisă ca şi parametru către procedură va fi incrementată, acest lucru putând fi vizibil în blocul anonim ce apelează procedura:

```
CREATE OR REPLACE PROCEDURE inc (p_val IN OUT NUMBER) AS
BEGIN
   p_val := p_val + 1;
END;
```

```
set serveroutput on;
DECLARE
   v_numar NUMBER := 7;
BEGIN
   inc(v_numar);
   DBMS_OUTPUT.PUT_LINE( v_numar );
END;
```

Încercaţi să rulaţi următorul bloc anonim şi explicaţi eroarea întâlnită:

```
BEGIN
   inc(7);
END;
```

Testaţi şi următorul cod... ce credeţi, va funcţiona ?

```
set serveroutput on;
DECLARE
   v_numar varchar2(10) := '7';
BEGIN
   inc(v_numar);
   DBMS_OUTPUT.PUT_LINE( v_numar );
END;
```

Încercaţi să construiţi un cod asemănător pentru o dată calendaristică.

Puteţi construi o procedură care să aibă ca parametri de intrare două valori numerice şi care să returneze suma acestora în o a treia variabilă ? Apelaţi această procedură punând direct valori numerice în locul parametrilor de intrare; o variabilă trebuie să fie furnizată pentru rezultat.

După cum aţi observat, puteţi trimite un şir de caractere către procedura ce incrementează valoarea şi ea va funcţiona corect. Acest lucru se întâmplă din cauză că PLSQL ştie să facă anumite conversii în mod automat. Din punct de vedere al eficienţei, este mai bine ca parametrii actuali (valorile trimise către procedură) să fie de acelaşi tip cu parametrii formali (cum au fost declaraţi în antetul procedurii). În cazul în care nu ştiţi cum au fost declaraţi parametrii (nu ştiţi tipul parametrilor formali), puteţi executa comanda **`DESCRIBE nume_procedură`** pentru a obţine aceste informaţii.

Unele limbaje de programare permit iniţializarea variabilelor formale în mod automat în cazul în care acestea nu primesc valori. Acest lucru este posibil şi în PLSQL:

```
CREATE OR REPLACE PROCEDURE pow (p_baza IN Integer := 3, p_exponent IN INTEGER DEFAULT 5) AS
   v_rezultat INTEGER;
BEGIN
    v_rezultat := p_baza ** p_exponent;
    DBMS_OUTPUT.PUT_LINE(v_rezultat);
END;
```

În acest cod parametrii de intrare sunt iniţializaţi. În cazul în care aceştia nu sunt transmişi din blocul anonim, valorile predefinite (din definiţia funcţiei) le sunt automat asociate. Pentru a preciza doar un anumit parametru (sau dacă doriţi să daţi parametrii în altă ordine decât cea din definiţia procedurii), puteţi să îi asociaţi în apel sub forma **`cheie=>valoare`** . De exemplu, următoarele trei scripturi vor avea acelaşi efect:

```
set serveroutput on;
BEGIN
   pow(2, 3);
END;
```

```
set serveroutput on;
BEGIN
   pow(p_baza=>2, p_exponent=>3);
END;
```

```
set serveroutput on;
BEGIN
   pow(p_exponent=>3, p_baza=>2);
END;
```

Puteţi să vă daţi seama înainte de a încerca efectiv următoarele scripturi care vor fi valorile afişate de procedura stocată (atenţie şi la valorile declarate în definiţia procedurii stocate) ?

```
set serveroutput on;
BEGIN
   pow(2, p_exponent=>3);
END;
```

```
set serveroutput on;
BEGIN
   pow(p_baza=>2);
END;
```

```
set serveroutput on;
BEGIN
   pow(p_exponent=>3);
END;
```

Iată un exemplu în care doar baza şi variabila în care trebuie să fie întors rezultatul sunt precizate (este nevoie de refacerea procedurii pentru a întoarce o valoare):

```
CREATE OR REPLACE PROCEDURE pow (p_baza IN Integer := 3, p_exponent IN INTEGER DEFAULT 5, p_out OUT Integer) AS
BEGIN
    p_out := p_baza ** p_exponent;
END;
```

```
set serveroutput on;
DECLARE
   v_out INTEGER;
BEGIN
   pow(p_baza=>3, p_out => v_out);
   DBMS_OUTPUT.PUT_LINE(v_out);
END;
```

Observaţie: o procedură poate conţine o comandă de tip **RETURN** pentru a forţa ieşirea din acea procedură.

Funcţii

Procedurile sunt utile atunci când codul PLSQL va intoarce mai mult decât o singură valoare. Atunci când este nevoie de o singură valoare returnată, se pot folosi funcţii.

Avantajul funcţiilor este că pot fi folosite şi în cadrul unei interogări (aşa cum aţi folosit de exemplu funcţia **UPPER** din SQL). Există câteva tipuri de date care previn utilizarea funcţiilor în comenzi de tip select (de exemplu boolean).

În funcţii încercaţi să evitaţi orice operaţie de tip DML (data manipulation language) sau DDL (data definition language). Adică, încercaţi să nu creaţi inseraţi/ştergeţi/modificaţi datele din tabele, să nu creaţi tabele, să le modificaţi structura sau să le ştergeţi. De asemenea, este bine să evitaţi **COMMIT** sau **ROLLBACK** sau modificarea variabilelor globale ale unui program PLSQL. În principiu funcţiile trebuie să calculeze ceva şi să returneze o informaţie. Puteţi spre exemplu să faceţi select într-o tabelă pentru că această operaţie nu modifică datele existente.

Sintaxa de creare a unei funcţii este asemănătoare cu cea a procedurii. În locul cuvântului **PROCEDURE** se va utiliza **FUNCTION** şi înainte de IS (sau AS) se adaugă tipul returnat de funcţie (**RETURN tip_de_date**). Obligatoriu trebuie ca în funcţie să se ajungă la o comandă de tip **RETURN expresie** , această valoare fiind de acelaşi tip cu cel declarat în antetul funcţiei. De obicei, toate modurile de transmitere a parametrilor vor fi setate pe **IN** (atunci când se utilizează **OUT** , funcţia nu mai poate fi folosită în **SELECT** -uri). Deoarece atunci când aceste moduri sunt omise valoarea lor predefinita este **IN** , puteţi să "le uitaţi".

În continuare este dat un exemplu de funcţie care are ca parametru de intrare un şir de caractere şi întoarce o prelucrare a şirului:

```
CREATE OR REPLACE FUNCTION make_waves(p_sir_caractere varchar2)
RETURN varchar2 AS
   v_index INTEGER;
   v_rezultat varchar2(1000) := '';
BEGIN
    FOR v_index IN 1..length(p_sir_caractere) LOOP
        IF(v_index MOD 2 = 1)
        THEN
           v_rezultat := v_rezultat || UPPER(SUBSTR(p_sir_caractere,v_index,1));
        ELSE
           v_rezultat := v_rezultat || LOWER(SUBSTR(p_sir_caractere,v_index,1));
        END IF;
    END LOOP;
    return v_rezultat;
END;
```

Funcţia poate fi apleată într-un select:

```
select make_waves('Facultatea de informatica') from dual;
```

Dar şi într-un bloc anonim:

```
DECLARE
   v_sir VARCHAR2(1000) :='Facultatea de informatica';
BEGIN
   v_sir := make_waves(v_sir);
   DBMS_OUTPUT.PUT_LINE(v_sir);
END;
```

Nu toate funcţiile trebuie să aibă parametri (e.g. **`SYSDATE`**).

Funcţiile apelate din **`SELECT`** nu au voie să conţină instrucţiuni **`DML`** .

Funcţiile apelate din **`UPDATE`** sau **`DELETE`** nu au voie să facă **`SELECT`** sau **`DML`** .

Funcţiile apelate din comenzi SQL nu au voie sa facă **`COMMIT`** sau **`ROLLBACK`** .

Funcţiile apelate din comenzi SQL nu pot contine instrucţiuni **`DDL`** (e.g. **`CREATE TABLE`**) sau **`DCL`** (e.g. **`ALTER SESSION`**) pentru ca ar face un **`COMMIT`** implicit.

Practică sisteme de gestiune pentru baze de date

Laborator

- **Instalare**
- **Ghid SQL*Plus**
- **SQL Developer**
- **RE-SQL**
- **Views**
- **Intro PLSQL**
- **Blocuri Anonime**
- **Variabile Operatori**
- **SELECT INTO**
- **Structuri de control**
- **Structuri iterative**
- **Cursoare**
- **Subprograme**
- **Colectii**
- **Pachete**
- **Obiecte**

Colecţii şi înregistrări

Linkuri utile:

- **Oracle Docs**
- **ORAcle Frequent Asked Questions**
- **Curs PUB**
- **Oracle Docs 2**

Pentru a permite lucrul cu şiruri de elemente, în SQL se vor folosi doua tipuri de date: `TABLE` şi `VARRAY` .

Colecţii - fiecare element al unei colecţii are un index ce îl identifică în mod unic în cadrul colecţiei. În PLSQL se pot face trei tipuri de colecţii:

- tablouri asociative (associative arrays)
- tabel/(ă) (nested tables)
- tablouri de dimensiune variabilă (varrays)

Pentru a lucra cu aceste tipuri de colecţii trebuie ca în prealabil să definiţi un tip după care să declaraţi variabile care să aibă tipul definit de voi. Declaraţia unui tip poate fi făcută, de exemplu, în secţiunea de declaraţii (fie a unui cod anonim, fie a unei funcţii/proceduri).

O colecţie poate fi trimisă ca parametru (pentru a transmite către o funcţie mai multe informaţii în acelaşi timp).

Tablourile asociative (associative arrays)

Tablourile asociative sunt mulţimi de cupluri de tipul cheie-valoare (cunoscute în alte limbaje de programare sub numele de tabele hash), fiecare cheie fiind unică (din cauză că pe baza ei se face accesul la valoare). Scopul acestui tip de date este de stocare temporară a informaţiilor, acestea fiind disponibile numai în cadrul aplicaţiei PL/SQL care le utilizează (deci tipurile nu sunt stocate pe disc în vreo bază de date pentru a fi regăsite ulterior, pentru aceasta trebuie să construiţi un cod special prin care să le salvaţi într-o bază de date).

Dacă se face o atribuire în tabloul asociativ pentru o cheie care nu există, aceasta este automat creată după care i se dă valoarea asignată de operaţia de atribuire.

Pentru a defini tipul ce va fi utilizat în declararea unui tablou asociativ se va utiliza comanda:

```
TYPE nume_tip IS TABLE OF tip_valori_indexate INDEX BY tip_cheie;
```

După definirea tipului trebuie să declaraţi o variabilă de acel tip. Valoarea "nume_tip" este un identificator pentru noul tip de dată (deci veţi putea declara o variabilă tablou de tipul nume_tip), tip_valori_indexate indică ce fel de elemente sunt stocate în fiecare poziţie a tabloului (se poate preciza ca această valoare trebuie să fie nenulă) iar în final este specificat tipul cheii (acesta poate fi un tip numeric sau şir de caractere).

Un exemplu de utilizare a unui tablou asociativ în care variabilele stocate sunt numere întregi iar cheile sunt şiruri de caractere:

```
DECLARE
    TYPE MyTab IS TABLE OF NUMBER INDEX BY VARCHAR2(10);
    varsta MyTab;
BEGIN
   varsta('Gigel') := 3;
   varsta('Ionel') := 4;
   DBMS_OUTPUT.PUT_LINE('Varsta lui Gigel este ' || varsta('Gigel'));
   DBMS_OUTPUT.PUT_LINE('Varsta lui Ionel este ' || varsta('Ionel'));
END;
```

În exemplul de mai sus se pot observa atât operaţia de asociere a unei chei cu o valoare cât şi modalitatea de extragere a unei valori atunci când cunoaştem cheia (în operaţia de afişare).

Dacă se doreşte adăugarea unui nou copil pe care îl cheamă (din întâmplare) tot Ionel şi care are vârsta de 7 ani, acest lucru nu va putea fi făcut deoarece cheia trebuie să fie unică (identificatorul Ionel în acest caz s-ar repeta). Încercarea de a face o atribuire de tipul varsta('Ionel') := 7; va duce la modificarea înregistrării deja existente în tablou.

Puteţi să construiţi un tablou asociativ în care cheile să nu mai fie de tip şir de caractere (ci un tip de date numeric) şi care să aibă ca şi valori posibile tipul liniei dintr-o tabelă:

```
DECLARE
    TYPE MyTab IS TABLE OF studenti%ROWTYPE INDEX BY PLS_INTEGER;
    linii MyTab;
BEGIN
    SELECT * INTO linii(0) FROM studenti WHERE ROWNUM = 1;
    DBMS_OUTPUT.PUT_LINE(linii(0).prenume);
END;
```

Dacă în poziţia cheii este utlizată o valoarea care nu este compatibilă (convertibilă în mod automat) de către SGBD se va semnala o eroare. În cazul în care conversia este posibilă, utilizarea valorii (de exemplu o data calendaristică ca şi cheie deşi cheile au fost declarate ca fiind de tip varchar2) nu va semnala nici o eroare şi codul va funcţiona corect:

```
DECLARE
    TYPE MyTab IS TABLE OF number INDEX BY varchar2(20);
    linii MyTab;
BEGIN
    linii(sysdate) := 123;
    DBMS_OUTPUT.PUT_LINE(linii(sysdate));
END;
```

Atenţie: în exemplul de mai sus, pentru valoarea de tip **DATE** ce a fost folosită ca şi cheie s-a aplicat funcţia **TO_CHAR** . În cazul în care un astfel de tabel asociativ ar fi salvat şi datele apoi mutate pe alt calculator, s-ar putea ca, din cauza modului în care sunt convertite datele (setările locale ale calculatorului), datele iniţiale să nu mai poată fi accesate.

Funcţiile ce pot fi utilizate pentru o colecţie de tipul tablou asociativ sunt exemplificate în următorul cod pe care vă invităm să îl rulaţi pentru a observa efectul fiecăreia. Explicaţii asupra acestor funcţii găsiţi în secţiunea "Funcţii ce pot fi utilizate pentru colecţii" din această pagină.

```
DECLARE
    TYPE MyTab IS TABLE OF NUMBER INDEX BY VARCHAR2(10);
    varsta MyTab;
BEGIN
   varsta('Gigel') := 3;
   varsta('Ionel') := 4;
   varsta('Maria') := 6;

   DBMS_OUTPUT.PUT_LINE('Numar de elemente in lista: ' || varsta.COUNT);

   DBMS_OUTPUT.PUT_LINE('Prima cheie din lista: ' || varsta.FIRST);
   DBMS_OUTPUT.PUT_LINE('Ultima cheie din lista: ' || varsta.LAST);

   DBMS_OUTPUT.PUT_LINE('Inaintea lui Ionel in lista: ' || varsta.PRIOR('Ionel'));
   DBMS_OUTPUT.PUT_LINE('Dupa Ionel in lista: ' || varsta.NEXT('Ionel'));

   varsta.DELETE('Maria');
   DBMS_OUTPUT.PUT_LINE('Dupa Ionel in lista: ' || varsta.NEXT('Ionel'));
END;
```

Tabel/(ă) (Nested tables)

Tipul tabel (sau tabelă - în www.dex.ro ambii termeni descriu acelaşi concept, indicând chiar unul spre celălalt) este un tip de date asemănător unei tabele dintr-o bază de date. Acest tip de date este utilizat, de obicei, pentru a stoca mai multe date asociate unei singure linii. La fel ca şi conceptul de tabelă din baza de date, informaţiile nu sunt organizate ci sunt mai degrabă văzute ca o mulţime: la un moment dat nu se ştie care este primul element, al doilea etc. Totuşi, la fel ca şi în baza de date, în momentul în care se doreşte iterarea elementelor, un număr provizoriu poate fi asociat fiecărei înregistrări (asemenei rownum).

La fel ca în cazul tablourilor asociative, se pot crea tabele a căror elemente să fie tot tabele (având aşadar o formă de matrice).

Pentru a accesa elementele dintr-un tabel se utilizează aceşi metodă ca şi la tablourile asociative. Există totuşi diferenţe în modul în care acestea sunt menţinute de către SGBD şi în uşurinţa cu care sunt transmise ca parametri unor funcţii sau proceduri stocate.

Spre deosebire de datele de tip tablou asociativ, datele de tip tabel pot fi stocate ca şi câmp în interiorul unui tabel din baza de date - deci se pot crea tabele (**`CREATE TABLE...`** care să aibă ca şi tip de data asociat unui câmp o variabilă de tip tabel).

Pentru a defini un nou tip ca şi tabel se va utiliza următoarea sintaxă:

În continuare se va declara o variabilă având ca şi tip "nume_tip" în care pot fi introduse elemente de având tipul "tip_element".

Iată un exemplu:

```
DECLARE
    TYPE prenume IS TABLE OF varchar2(10);
    student prenume;
BEGIN
    student := prenume('Gigel', 'Ionel');
    for i in student.first..student.last loop
       DBMS_OUTPUT.PUT_LINE(i||' - '||student(i));
    end loop;
END;
```

Să construim în continuare un tabel cu trei elemente, să îi adăugăm încă patru elemente copiind elementul din mijloc de încă două ori după care să ştergem elementul al doilea şi să afişăm toate poziţiile ocupate şi elementele de pe aceste poziţii:

```
DECLARE
    TYPE prenume IS TABLE OF varchar2(10);
    student prenume;
BEGIN
    student := prenume('Gigel', 'Ionel', 'Maria');
    student.EXTEND(4,2); -- copii elementul al doilea de 4 ori
    student.delete(2); -- sterg elementul al doilea
    for i in student.first..student.last loop
        if student.exists(i) then -- daca incerc sa afisez ceva ce nu exista se va produce o eroare
           DBMS_OUTPUT.PUT_LINE(i||' - '||student(i));  -- afisam pozitia si valoarea
        end if;
    end loop;
END;
```

Mai multe funcţii ce pot fi utilizate în cadrul tablourilor găsiţi în secţiunea "Funcţii ce pot fi utilizate pentru colecţii" din această pagină.

După cum se observă în exemplul anterior, ştergerea elementului al doilea din tabel nu a dus la deplasarea automată a informaţiilor de pe poziţiile 3..7 spre stânga. Atenţie: după ştergere, pe poziţia a doua nu a rămas nici un element. Încercarea de a afişa elementul de pe poziţia a doua va genera o eroare.

Valoarea lui tip_element poate fi şi altceva decât varchar2(10). Puteţi încerca să puneţi acelaşi tip cu al unei coloane existente într-un tabel (**`studenti.prenume%type`**), a unui rând dintr-o tabelă (**`studenti%ROWTYPE`**) sau a unui rând dintr-un cursor (**`cursor%ROWTYPE`**).

Nu puteți extinde șiruri care sunt nule (sau neinițializate):

```
DECLARE
    TYPE prenume IS TABLE OF varchar2(10);
    student prenume := prenume();
    student_err prenume;
BEGIN
    student.EXTEND; -- merge ok
    student_err.EXTEND; -- da eroare pentru ca este null (sau nu a fost initializat)
END;
```

Iată un exemplu care încarcă într-o variabilă de tip tabel toate elementului tabelului studenti:

```
DECLARE
    CURSOR curs IS SELECT nume, prenume FROM studenti;
   -- cursorul este utilizat doar in linia urmatoare, pentru a defini tipul valorilor
   -- din nested table. Se poate folosi si un record in care definiti doar nume, prenume.
   TYPE linie_student IS TABLE OF curs%ROWTYPE;
   lista_studenti linie_student;
BEGIN
   SELECT nume, prenume BULK COLLECT INTO lista_studenti FROM studenti;
    for i in lista_studenti.first..lista_studenti.last loop
        if lista_studenti.exists(i) then -- daca incerc sa afisez ceva ce nu exista se va produce o eroare
           DBMS_OUTPUT.PUT_LINE(i||' - '||lista_studenti(i).nume);  -- afisam pozitia si valoarea
        end if;
    end loop;
    DBMS_OUTPUT.PUT_LINE('Numar studenti: '||lista_studenti.COUNT);
END;
```

Exemplul anterior poate fi refăcut utilizând funcţia **`NEXT`** pentru a obţine cheia următorului element. În acest caz nu mai este nevoie de testarea existenţei elementului (se poate elimina condiţia if).

Puteţi crea tabele (cu **`CREATE TABLE`**) care să aibă un anumit câmp de tipul tabel. Iată un exemplu în acest sens:

```
GRANT CREATE TYPE TO STUDENT; -- aceasta linie se executa din "SYS as SYSDBA"

CREATE OR REPLACE TYPE lista_prenume AS TABLE OF VARCHAR2(10);
/
CREATE TABLE persoane (nume varchar2(10),
       prenume lista_prenume)
       NESTED TABLE prenume STORE AS lista;
/

INSERT INTO persoane VALUES('Popescu', lista_prenume('Ionut', 'Razvan'));
INSERT INTO persoane VALUES('Ionescu', lista_prenume('Elena', 'Madalina'));
INSERT INTO persoane VALUES('Rizea', lista_prenume('Mircea', 'Catalin'));
/
SELECT * FROM persoane;

-- mai multe operatii direct cu tabelele interne dintr-un tabel gasiti in linkurile de la inceputul acestei pagini.
```

Şi în continuare o funcţie PL/SQL care să insereze în această nouă tabelă o persoană:

```
DECLARE
    sir_prenume persoane.prenume%type;
BEGIN
    sir_prenume := lista_prenume('Cristi', 'Tudor', 'Virgil');
    INSERT INTO persoane VALUES ('Gurau', sir_prenume);
    DBMS_OUTPUT.PUT_LINE('Gata');
END;
```

Nu uitaţi să testaţi funcţiile specifice tabelelor (date într-o secţiune următoare a acestei pagini).

Tablouri cu dimensiune variabilă (Varrays)

Al doilea tip de colecţie precizat este cel al tablourilor cu dimensiune variabilă. Pentru a declara un varray se foloseşte următoarea sintaxă

```
TYPE nume_tip IS VARRAY( numar_initial_elemente ) OF tip_de_baza;
```

Iată un exemplu de utilizare a unui tablou cu dimensiune variabilă (din care se şterge un element după care se adaugă încă două elemente):

```
DECLARE
  TYPE varr IS VARRAY(5) OF varchar2(10);
  orase varr;
BEGIN
  orase := varr('Iasi', 'Bacau', 'Suceava', 'Botosani');
  DBMS_OUTPUT.PUT_LINE('Numar orase: '||orase.COUNT);
  orase.TRIM;
  FOR i IN orase.FIRST..orase.LAST LOOP
     DBMS_OUTPUT.PUT_LINE(orase(i));
  END LOOP;

  orase.EXTEND(2);
  orase(4):='Sibiu';
  orase(5):='Brasov';
  DBMS_OUTPUT.PUT_LINE('Dupa adaugare:');
  FOR i IN orase.FIRST..orase.LAST LOOP
     DBMS_OUTPUT.PUT_LINE(orase(i));
  END LOOP;
END;
```

Nu uitaţi să testaţi şi celelalte funcţii care pot fi aplicate variabilelor de tip varray (din lista cu funcţii aplicabile colecţiilor din această pagină).

Funcţii ce pot fi utilizate pentru colecţii

- **FIRST** - returnează valoarea cheii (sau indicele) primului element;

- **`LAST`** - returnează valoarea cheii (sau indicele) ultimului element;
- **`PRIOR(cheie)`** - returnează cheia elementului dinaintea celui dat ca parametru (cheie poate fi şi o valoare numerică dacă este vorba despre altceva decât tablouri asociative);
- **`NEXT(cheie)`** - returnează cheia elementului următor celui dat ca parametru (cheie poate fi şi o valoare numerică dacă este vorba despre altceva decât tablouri asociative);
- **`EXISTS(cheie)`** - returnează valoarea true dacă există o valoarea atribuită cheii (cheia poate fi şi poziţia într-o colecţie în cazul în care nu este vorba de tablouri asociative);
- **`COUNT`** - returnează numărul de elemente din colecţie;
- **`varray.LIMIT`** - câte elemente pot fi adăugatre în variabila de tip varray;
- **`EXTEND [(n[,i])]`** - pentru tipul tabel şi varray: pentru a adăuga n poziţii în structură (eventual toate având valoarea elementului de pe poziţia i). În cazul în care nu există nici un parametru, se extinde cu un singur element. Nu se aplică tabelelor asociative;
- **`TRIM [(n)]`** - şterge n elemente de la sfârştiul unei variabile de tip tabel sau dintr-un varray (nu este şi pentru tablouri asociative). În cazul în care n nu este dat, se şterge ultimul element;
- **`DELETE [(n,[m])]`** - şterge fie toate elementele (când nu are parametru), fie elementul de pe poziţia n, fie elementele de pe poziţiile n, n+1, ... m. Nu se aplică variabilelor de tip varray.

Tipul înregistrare (record)

Tipul înregistrare permite cumularea mai multor valori (nu neapărat de acelaşi tip) într-un singur tip - tipul înregistrării. Fiecare câmp al unei înregistrări are un nume (prin care poate fi identificat) şi un tip.

V-aţi mai întâlnit deja cu variabile de tip **RECORD**, atunci când le-aţi declarat ca şi **`ROWTYPE`** (în cadrul unui cursor sau pentru a prelua o linie din tabel). Să vedem care este sintaxa pentru definirea unei înregistrări în PL/SQL printr-un exemplu:

```
CREATE TABLE minions (culoare varchar2(20), numar_ochi number(3), nume varchar2(20));
 /
DECLARE
   TYPE minion IS RECORD(
      culoare varchar2(20) := 'Galben',
      numar_ochi number(3),
      nume varchar2(20)
      );
   v_minion minion;
BEGIN
   v_minion.culoare:='Galben';
   v_minion.numar_ochi := 2;
   v_minion.nume:='Kevin';
   INSERT INTO MINIONS VALUES V_MINION;
   DBMS_OUTPUT.PUT_LINE(v_minion.culoare);
END;
```

Practică sisteme de gestiune pentru baze de date

Laborator

Pachete

La fel ca şi bibliotecile din C / C++ (şi probabil şi alte limbaje de programare), pachetele sunt utilizate pentru a organiza mai multe proceduri, funcţii (sau clase) care fie sunt interdependente fie sunt utilizate în acelaşi scop (de exemplu pentru a face managementul studenţilor dintr-o facultate) la un loc. Pachetele din PLSQL sunt containere care au scopul de a grupa subprograme(proceduri sau funcţii), variabile, cursoare şi excepţii.

Un pachet este construit din două entităţi:

- **specificaţia sau antetul** - oferă o interfaţă pe care programatorul poate sa o utilizeze. Aceasta trebuie să fie declarată la început (nu puteţi compila cea de-a doua entitate fără a o avea pe prima compilată). Aici sunt specificate toate elementele vizibile din exterior: proceduri, funcţii, variabile etc.
- **body** - conţine efectiv codul procedurilor declarate în specificaţie. După compilare, body-ul nu va mai fi vizibil, interfaţa descrisă în antet fiind singura la care programatorul are acces. Secţiunea de body poate fi recompilată fără a fi necesară recompilarea antetului.

Antetul

Pentru a crea un antetul unui pachet se utilizează sintaxa:

```
CREATE [OR REPLACE] PACKAGE nume_pachet IS|AS
        tipuri publice
        declarații de variabile publice (vor fi inițializate cu NULL daca nu se specifica altceva)
        specificații publice pentru subprograme
END [nume_pachet]
```

Tot ce apare este vizibil utilizatorilor care au drept de EXECUTE pe acel pachet.

Cuvântul public indică faptul că elementele descrise în antet sunt accesibile din exterior (şi nu sunt private - remember Java encapsulation).

Exemplu:

```
CREATE OR REPLACE PACKAGE manager_facultate IS
      g_today_date   DATE:= SYSDATE;
      CURSOR lista_studenti IS SELECT nr_matricol, nume, prenume, grupa, an FROM studenti ORDER BY nume;
     PROCEDURE adauga_student (nume studenti.nume%type, prenume studenti.prenume%type);
     PROCEDURE sterge_student (nr_matr studenti.nr_matricol%type);
END manager_facultate;
```

Body

Pentru a construi şi body-ul, se utilizează următoarea sintaxă:

```
CREATE [OR REPLACE] PACKAGE BODY nume_pachet IS|AS
      variabile și tipuri de date private
      conținutul subprogramelor (fie publice fie private);
[BEGIN  zona de initializare]
END nume_pachet;
```

Trebuie să declaraţi o procedură înaintea utilizării ei în altă procedură. Fiecare procedură sau funcţie declarată în antet trebuie neapărat să aibă şi un cod în secţiunea de body. Iată un exemplu care ar putea constitui un body pentru pachetul de mai sus:

```
CREATE OR REPLACE PACKAGE BODY manager_facultate IS
    nume_facultate VARCHAR2(100) := 'Facultatea de Informatica din IASI';

    FUNCTION calculeaza_varsta (data_nastere DATE) RETURN INT AS
    BEGIN
       RETURN FLOOR((g_today_date - data_nastere)/365);
    END calculeaza_varsta;

    PROCEDURE adauga_student (nume studenti.nume%type, prenume studenti.prenume%type)
       IS BEGIN
           DBMS_OUTPUT.PUT_LINE('Exemplu apel functie privata: '||
calculeaza_varsta(to_date('01/01/1990','DD/MM/YYYY')));
           DBMS_OUTPUT.PUT_LINE('Aici ar trebui sa scrieti cod pentru adaugarea unui student');
    END adauga_student;

    PROCEDURE sterge_student (nr_matr studenti.nr_matricol%type) IS
    BEGIN
       null; -- nu face nimic
    END sterge_student;

END manager_facultate;
```

-- BODY-UL PACHETULUI VA DA EROARE LA COMPILARE DACA NU ATI COMPILAT IN PREALABIL ANTETUL (vezi secțiunea anterioara) SAU DACA NU AVETI IN BODY FUNCTIILE DECLARATE SI IN ANTET (trebuie să aveți în body funcții având aceeași signatură ca cele declarate în antet).

În acest exemplu, funcţia calculeaza_varsta nu a fost expusă publicului şi din acest motiv putem să o considerăm privată. Dacă veţi încerca să compilaţi secţiunea de body fără a introduce şi procedura de ştergere a unui student, va fi semnalată o eroare conform căreia toate metodele declarate în partea de antet a pachetului trebuie să fie implementate.

În cadrul pachetului puteţi utiliza variabilele sau cursoarele care au fost declarate public în antetul pachetului dar puteţi declara şi variabile care să nu fie expuse.

Dacă aţi uitat ce proceduri aveţi expuse într-un pachet, puteţi executa comanda **`DESCRIBE`** urmată de numele pachetului (e.g. **`DESCRIBE manager_facultate;`**).

Pentru a apela un subprogram din cadrul unui aceluiaşi pachet este suficient să scrieţi numele subprogramului (procedură sau funcţie) - în exemplul de mai sus, funcţia privată (care nu a fost expusă în antet) a fost apelată în procedura de adăugare a studentului (doar pentru exemplificare). Puteţi să daţi calea completă utilizând formatul nume_pachet.nume_metodă pentru a apela metoda dinafara pachetului (deşi este posibil un astfel de apel şi din cadrul pachetului). Iată un exemplu de apel a unei metode expuse (publice) dinafara pachetului:

```
set serveroutput on;
BEGIN
   manager_facultate.adauga_student('Becali', 'Gigi');
END;
```

Pentru a şterge un întreg pachet puteţi executa comanda:

Pentru a şterge numai BODY-ul pachetului:

```
DROP PACKAGE BODY nume_pachet
```

Nu se poate şterge doar antetul (specificaţia) fără a şterge şi body.

Observație: puteți scrie antetul și body-ul în același fișier dar trebuie ca între ele să faceți un commit (scrieti pe un rând nou caracterul '/').

Practică sisteme de gestiune pentru baze de date

Laborator

Obiecte

Linkuri utile:

- **http://www.tutorialspoint.com/plsql/plsql_object_oriented.htm**
- **https://docs.oracle.com/cd/B28359_01/appdev.111/b28371/adobjplsql.htm**
- **https://docs.oracle.com/cd/B13789_01/appdev.101/b10807/10_objs.htm#i16312**

În majoritatea limbajelor de programare moderne programarea este orientată-obiect. Deşi scripturile care sunt necesare în cadrul bazelor de date sunt de obicei destul de simple şi fac lucruri specifice asupra unei baze de date, şi în cadrul PL/SQL este permis lucrul cu obiecte.

Câteva dintre conceptele pe care le cunoaşteţi de la OOP (C++ sau Java) sunt implementate şi aici. Puteţi să compuneţi tipuri, să derivaţi obiecte noi din altele deja existente, etc. Ca şi în programarea OOP, obiectele au o mulţime de proprietăţi (sau atribute) şi o mulţime de metode. Atributele sunt utilizate pentru a reprezenta starea obiectului în timp ce metodele sunt utilizate pentru a simula diverse comportamente ale obiectelor.

Construirea unei clase in PL/SQL

"Clasa" din care este făcut un obiect este de fapt un tip pe care îl puteţi declara după sintaxa:

În secţiunea dintre paranteze sunt declarate atributele obiectului şi tipurile lor împreună cu procedurile şi funcţiile membre. Un exemplu în acest sens (vom construi un tip student prin intermediul căruia vom putea "instanţia" studenţi) este următorul:

```
CREATE OR REPLACE TYPE student AS OBJECT
(nume varchar2(10),
 prenume varchar2(10),
 grupa varchar2(4),
 an number(1),
 data_nastere date,
 member procedure afiseaza_foaie_matricola
);
```

În cadrul declaraţiei unui obiect nu puteţi să utilizaţi `%TYPE` . Acest lucru nu este valid deoarece dacă cineva se hotărăşte să modifice tabela pe baza căreia obiectul a fost construit atunci şi obiectul ar avea de suferit (şi probabil şi alte subprograme ce folosesc acest obiect).

Împreună cu declararea unui obiect trebuie declarat şi un **BODY** pentru acel obiect. În acest body pot fi scrise implementările pentru metodele declarate în obiect (de exemplu pentru procedura afiseaza_foaia_matricola).

Pentru a crea secţiunea de body se va utiliza următoarea sintaxă:

```
CREATE OR REPLACE TYPE BODY student AS
        ..................
ENDl
```

Între antetul BODYului şi cuvântul END sunt scrise toate implementările metodelor declarate în obiect. Ca şi exemplu, pentru obiectul student de mai sus vom putea declara următorul body:

```
CREATE OR REPLACE TYPE BODY student AS
   MEMBER PROCEDURE afiseaza_foaie_matricola IS
   BEGIN
       DBMS_OUTPUT.PUT_LINE('Aceasta procedura calculeaza si afiseaza foaia matricola');
   END afiseaza_foaie_matricola;
END;
```

Putem să creem tabele în care să introducem ca şi coloană valori-instanţă ale tipului nou creat (student). De exemplu, dacă executăm comanda următoare vom crea un tabel de tip cheie-valoare în care ca şi cheie vom trece numărul matricol al studentului iar ca şi valoare vom introduce efectiv obiectul:

```
CREATE TABLE studenti_oop (nr_matricol VARCHAR2(4), obiect STUDENT);
```

Un exemplu de declarare şi utilizare a unui obiect de tip student precum şi o comandă de inserare în tabela nou creată este dat în continuare:

```
set serveroutput on;
DECLARE
   v_student1 STUDENT;
   v_student2 STUDENT;
BEGIN
   v_student1 := student('Popescu', 'Ionut', 'A2', 3, TO_DATE('11/04/1994', 'dd/mm/yyyy'));
   v_student2 := student('Vasilescu', 'George', 'A4', 3, TO_DATE('22/03/1995', 'dd/mm/yyyy'));
   v_student1.afiseaza_foaie_matricola();
   dbms_output.put_line(v_student1.nume);
   insert into studenti_oop values ('100', v_student1);
   insert into studenti_oop values ('101', v_student2);
END;
```

Acum puteți să construiți o cerere de tip select care să vă returneze doar anumite câmpuri din obiectul stocat în baza de date în câmpul "obiect":

```
SELECT TREAT(obiect AS student).nume FROM studenti_oop;
```

În cadrul obiectului puteţi să declaraţi şi funcţii (în mod asemănător cum a fost definită şi procedura dar cu diferenţele specifice funcţiilor). Antetul funcţiei din body va fi trecut în declaraţia obiectului. Nu uitaţi de cuvântul cheie "Member".

După cum se poate observa în exemplul pe care l-am dat, un obiect are un constructor (care este apelat prin numele tipului obiectului) care preia ca parametri valorile atributelor din acel obiect (în ordinea declarării lor) şi le atribuie obiectului. Dacă doriţi să aveţi un constructor propriu (şi nu cel implicit), iată cum se face acest lucru în cazul exemplului nostru:

În cadrul declaraţiei obiectului adăugaţi ca şi metodă constructorul:

```
CONSTRUCTOR FUNCTION student(nume varchar2, prenume varchar2)
    RETURN SELF AS RESULT
```

Apoi, în cadrul body-ului, specificaţi ce doriţi să faceţi în această nouă metodă-constructor:

```
CONSTRUCTOR FUNCTION student(nume varchar2, prenume varchar2)
    RETURN SELF AS RESULT AS
BEGIN
    SELF.nume := nume;
    SELF.prenume := prenume;
    SELF.data_nastere := sysdate;
    SELF.an := 1;
    SELF.grupa := 'A1';
    RETURN;
END;
```

Declaraţi o nouă variabilă de tipul student şi instanţiaţi-o folosind noul constructor ce tocmai a fost definit.

Înafara constructorului, o altă metodă "mai specială" este cea care permite compararea a două de acelaşi tip. Această funcţie este necesară pentru că, în mod normal, SGBDul nu ştie care dintre cele două variabile pe care le-am declarat mai sus (v_student1 şi v_student2) este mai mare. În cazul în care am dori să sortăm tabela studenti_oop (declarată mai sus) după coloana obiect, ce ordine ar fi adoptată (cum ar fi comparate două elemente ale acestei coloane? după nume, după prenume sau poate după vârsta studentului ?).

Sortarea şi compararea

În continuare vă invităm să construiţi în cadrul obiectului student o funcţie (cu denumirea varsta_in_zile) ce returnează vârsta în zile a studentului (scădeţi din sysdate valoarea data_nastere din obiect). Vom folosi această funcţie pentru a compara doi studenţi.

Există două metode de a compara două obiecte: Printr-o metodă `MAP` sau printr-o metodă `Order` . Metoda de tip `MAP` va returna o valoare (spre exemplu de tip number). Dacă primul obiect returnează un număr mai mic decât cel de-al doilea obiect atunci el este considerat mai mic.

Adăugând cuvântul `MAP` în faţa funcţiei varsta_in_zile (pe care aţi creat-o puţin mai înainte), veţi putea compara doua obiecte de tip student în funcţie de numărul de zile de când s-au născut.

Adăugaţi în blocul anonim următoarea secţiune de cod:

```
    if (v_student1 < v_student2)
       THEN DBMS_OUTPUT.PUT_LINE('Studentul '|| v_student1.nume || ' este mai tanar.');
       ELSE DBMS_OUTPUT.PUT_LINE('Studentul '|| v_student2.nume || ' este mai tanar.');
    END IF;
```

În cele mai multe limbaje de programare funcţia de comparare nu este bazată pe reprezentarea obiectului ca şi o valoare ce ar putea fi comparată cu cea obţinută din reprezentarea similară a unui alt obiect ci mai degrabă prin întoarcerea uneia dintre valorile -1, 0, 1 (în funcţie dacă obiectul curent SELF în PLSQL sau this în Java spre exemplu este -1: mai mic, 0: egal, 1: mai mare decât obiectul dat ca parametru).

Diferenţa dintre cele două funcţii: `MAP` şi `ORDER` este că în timp ce funcţia de tip `MAP` nu are nevoie de nici un parametru pentru că întoarce reprezentarea obiectului curent ca un tip de dată comparabilă, funcţia de tip `ORDER` are obligatoriu ca parametru un obiect de acelaşi tip cu obiectul cu care se face comparaţia. Nu se pot folosi ambele metode în acelaşi timp, pentru comparaţia ce va fi făcută cu metoda `ORDER` , codul dat mai sus pentru `MAP` este în continuare corect. Încercaţi să rescrieţi funcţia `MAP` astfel încât să fie de top `ORDER` (de fapt când se compară (`v_student1 < v_student2`) puteţi să vă închipuiţi că se apelează ceva de genul "v_student1.comparaCu(v_student2)" şi de fapt funcţia comparaCu este funcţia ce trebuie declarata cu `ORDER`).

Moştenirea în PLSQL

În PLSQL, la fel ca şi în alte limbaje orientate obiect, se pot crea obiecte plecând de la obiectele existente. Pentru a putea crea un obiect-copil, clasa parinte (superclasă) trebuie să fie declarată ca `NOT FINAL` (valoarea predefinită este `FINAL`). Refaceţi tipul student adăugând cuvintele `NOT FINAL` (la final). Pentru a permite suprascrierea unei metode din cadrul obiectului, în faţa declaraţiei sale trebuie să fie scrise cuvintele `NOT FINAL` .

În exemplul nostru am putea construi declaraţia studentului în felul următor:

```
CREATE OR REPLACE TYPE student AS OBJECT
(nume varchar2(10),
 prenume varchar2(10),
 grupa varchar2(4),
 an number(1),
 data_nastere date,
 NOT FINAL member procedure afiseaza_foaie_matricola,
 map member FUNCTION varsta_in_zile RETURN NUMBER,
 CONSTRUCTOR FUNCTION student(nume varchar2, prenume varchar2)
    RETURN SELF AS RESULT
) NOT FINAL;
```

În acest caz, se poate crea un nou tip (clasă) ca şi subclasă pentru student şi, mai mult, în această nouă subclasă putem să suprascriem procedura **afiseaza_foaia_matricola** .

Iată în continuare codul care creează o subclasă a clasei student care are în plus o proprietate "bursa" şi care suprascrie metoda de afişare a foii matricole. Totodată este dat şi blocul anonim care ar putea fi utilizat pentru a declara şi utiliza acest nou tip de obiect.

```
drop type student_bazat;
CREATE OR REPLACE TYPE student_bazat UNDER student
(
   bursa NUMBER(6,2),
   OVERRIDING member procedure afiseaza_foaie_matricola
)
/


CREATE OR REPLACE TYPE BODY student_bazat AS
    OVERRIDING MEMBER PROCEDURE afiseaza_foaie_matricola IS
    BEGIN
       dbms_output.put_line('bursier');
    END afiseaza_foaie_matricola;
END;
/

DECLARE
   v_student_bazat student_bazat;
BEGIN
    v_student_bazat := student_bazat('Mihalcea', 'Mircea', 'A1', 2, TO_DATE('18/09/1996', 'dd/mm/yyyy'), 1000);
    dbms_output.put_line(v_student_bazat.nume);
    v_student_bazat.afiseaza_foaie_matricola();
END;
```

Construiţi două instanţe ale studentului bazat şi comparaţi-le. Se păstrează metoda de comparare pe care aţi utilizat-o în compararea obiectelor de tip student declarat iniţial ?

Pentru a crea o clasă abstractă utilizaţi **NOT INSTANTIABLE** după declaraţia clasei (înainte de **NOT FINAL**). Are rost să folosiţi **NOT INSTANTIABLE** fără a fi urmată de **NOT FINAL** ? Executaţi codul:

```
DECLARE
  s student;
BEGIN
  SELECT obiect INTO s FROM studenti_oop WHERE nr_matricol='100';
  dbms_output.put_line(s.nume);
END;
```

Practică sisteme de gestiune pentru baze de date

Laborator

Exceptii

Mai multe detalii despre excepții găsiți în documentația oficială Oracle:

http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/errors.htm

De ce folosim excepții

Multi programatori aleg sa gândească superficial și să scrie cod care să respecte specificațiile inițiale și atât. Ei speră că userii vor folosi codul doar conform acelor specificații și că nu vor aparea buguri. Oh, well... Pe lângă ei mai sunt programatorii buni care conștientizează că oricând userii sau chiar colegii lor programatori care lucrează la alte module ale aplicației pot face ceva în afara acelor specificații și își scriu codul în consecință. De exemplu userul poate apela o procedură cu niște parametri invalizi iar colegul programator poate sa omită să scrie o validare împotriva acelor valori. În final pentru noi totul se reduce la o alegere: faci suficient și nu-ți pasă ce se întâmplă în viitor sau proiectezi și scrii aplicația în așa fel încât să protejezi userii și în cazul nostru baza de date de posibile erori. Aș vrea să cred că noi facem parte din a doua categorie.

Din fericire, limbajul PL/SQL ne oferă un mod suficient de puternic și de flexibil ca să tratăm excepțiile și să oferim userilor mesaje cât mai utile în diverse scenarii. Și cât timp folosim în mod corect excepțiile nici nu este nevoie de foarte mult cod în plus ca să prevedem și să tratăm posibilele erori. În secțiunile următoare vom vedea ce este o excepție în PL/SQL, cum se definește ea și mai ales cum se prinde și se trimite mai departe la apelant.

Când folosim excepții

Vom porni de la un exemplu și anume o funcție care primește ca și argument un ID al unui student și trebuie să returneze cea mai recentă notă a acelui student.

Iată o primă variantă de cod pentru această procedură:

```
CREATE OR REPLACE FUNCTION nota_recenta_student(
    pi_id_student IN studenti.id%type)
  RETURN VARCHAR2
AS
  nota_recenta INTEGER;
  mesaj        VARCHAR2(32767);
BEGIN
  SELECT valoare
  INTO nota_recenta
  FROM
    (SELECT valoare
    FROM note
    WHERE id_student = pi_id_student
    ORDER BY data_notare DESC
    )
  WHERE rownum <= 1;
  mesaj        := 'Cea mai recenta nota a studentului cu matricolul ' || pi_id_student || ' este ' || nota_recenta ||
'.';
  RETURN mesaj;
END nota_recenta_student;
```

Acesta este un cod din prima categorie de care vorbeam anterior. Respectă specificațiile inițiale și anume să returneze cea mai recentă notă a studentului, puteți să îl prezentați la un laborator și eventual un profesor îngăduitor vă va da punctajul maxim indiferent că poate va spune că mai puteați adăuga ceva.

Acum să vedem ce s-ar putea adăuga la acest cod și pentru asta pornim de la posibile erori sau posibile valori irelevante întoarse de această funcție. Vom rula următoarele două selecturi și iată rezultatele lor:

```
 select nota_recenta_student(1) from dual;
```

Pentru a trata într-un mod mai util pentru user aceste cazuri vom face următoarele modificări în procedură:

```
  CREATE OR REPLACE FUNCTION nota_recenta_student(
    pi_id_student IN studenti.id%type)
  RETURN VARCHAR2
AS
  nota_recenta INTEGER;
  mesaj        VARCHAR2(32767);
  counter      INTEGER;
BEGIN
  SELECT valoare
  INTO nota_recenta
  FROM
    (SELECT valoare
    FROM note
    WHERE id_student = pi_id_student
    ORDER BY data_notare DESC
    )
  WHERE rownum <= 1;
  mesaj        := 'Cea mai recenta nota a studentului cu matricolul ' || pi_id_student || ' este ' || nota_recenta ||
'.';
  RETURN mesaj;
EXCEPTION
WHEN no_data_found THEN
  SELECT COUNT(*) INTO counter FROM studenti WHERE id = pi_id_student ;
  IF counter = 0 THEN
    mesaj   := 'Studentul cu ID-ul ' || pi_id_student || ' nu exista in baza de date.';
  ELSE
    SELECT COUNT(*) INTO counter FROM note WHERE id_student = pi_id_student ;
    IF counter = 0 THEN
      mesaj   := 'Studentul cu ID-ul ' || pi_id_student || ' nu are nici o nota.';
    END IF;
  END IF;
RETURN mesaj;
END nota_recenta_student;
```

Așa cum ați văzut, în funcție/procedură/bloc anonim a apărut o nouă secțiune, și anume exception:

```
begin
exception
```

Să rulăm din nou cele două apeluri de funcție:

```
select nota_recenta_student(1) from dual;
```

Care variantă vi se pare mai ok pentru utilizatorul aplicației? Ce mesaj preferați să îi

afisați? "No data found" sau "Studentul cu matricolul -1 nu există în baza de date."?

Tipuri de excepții

Sunt trei tipuri de excepții (tehnic vorbind două, plus o a treia atât de generică încât merită o categorie separata):

- excepții definite de catre user - declarate ca și obiect de tip exception și inițializate cu ajutorul funcției PRAGMA EXCEPTION_INIT cu un cod de eroare între -20001 și -20999.

declare

student_inexistent exception;

PRAGMA EXCEPTION_INIT(student_inexistent, -20001);

begin

if...then

raise student_inexistent;

end if;

exception

when student_inexistent then

[do something]

end;

In unele cazuri, desi o eroare este definita la nivel de SGBD, aceasta nu este in mod clar asociata unui identificator de tip string (cum a fost folosit "no_data_found" in cazul precedent). Spre exemplu, atunci cand o data calendaristica nu este formatata corect, aceasta eroare poate fi prinsa cu "others", dar daca vreau sa o prind exact pe aceasta (si numai pe aceasta), ar trebui sa prind eroarea asociata codului de eroare ORA-01830 - care nu are asociata o prezentare alfanumerica. Ca sa o putem prinde, ar trebui sa asociem acest cod unei erori definite de utilizator:

```
   e_bad_date_format   EXCEPTION;
   --PRAGMA EXCEPTION_INIT (e_bad_date_format, -1830);
BEGIN
   DBMS_OUTPUT.put_line (TO_DATE ('10-04-2010', 'DD-YYYY'));
EXCEPTION
   WHEN e_bad_date_format
   THEN
      DBMS_OUTPUT.put_line ('Bad date format');
END;
```

- excepții predefinite existente in baza de date - țineți minte că excepțiile predefinite au coduri de eroare intre -1 (unique constraint violated) și -20000. Atunci când ne definim excepțiile noastre este recomandat să folosim coduri de eroare începând cu -20001 pentru a nu ne intersecta cu acele coduri deja folosite sau rezervate de Oracle. Capătul intervalului este -20999, deci putem defini maxim o mie de excepții custom.

Cateva exemple de excepții predefinite:

DUP_VAL_ON_INDEX - A program attempts to store duplicate values in a column that is constrained by a unique index.

NO_DATA_FOUND - A SELECT INTO statement returns no rows, or your program references a deleted element in a nested table or an uninitialized element in an index-by table.

ZERO_DIVIDE - A program attempts to divide a number by zero.

```
[code]
exception
  when NO_DATA_FOUND then
    [do something]
```

- excepția predefinită OTHERS pe care baza de date o aruncă în cazul oricărei excepții din afara celor predefinite și definite de către user.

begin

[code]

exception

when NO_DATA_FOUND then

[do something]

when OTHERS then

[do something]

end;

Putem folosi WHEN OTHERS pentru a afișa mesaje generice gen "Eroare neașteptată. Vă rugăm contactați administratorul aplicației".

Un alt mod de a folosi WHEN OTHERS este sintaxa EXCEPTION WHEN OTHERS THEN NULL care ignoră orice excepție aruncată de codul nostru. Sper că este inutil să precizez să nu folosiți niciodată într-o aplicație această construcție :) Singura folosire acceptabilă a ei este îintr-un cod de test, de exemplu dacă vrem să afișăm ceva și ignorăm deliberat orice excepție. Repet, doar pentru teste.

Cum folosim o excepție

Sunt două moduri în care putem folosi o exceptie:

- aruncăm excepția într-un anumit scenariu (varianta mai simpla). În acest caz nu definim explicit exceptia și doar trimitem apelantului un cod de eroare si un mesaj:

```
CREATE OR REPLACE FUNCTION nota_recenta_student(
    pi_id_student IN studenti.id%type)
  RETURN VARCHAR2
AS
  nota_recenta INTEGER;
  mesaj        VARCHAR2(32767);
  counter      INTEGER;
BEGIN
  SELECT valoare
  INTO nota_recenta
  FROM
    (SELECT valoare
    FROM note
    WHERE id_student = pi_id_student
    ORDER BY data_notare DESC
    )
  WHERE rownum <= 1;
  mesaj        := 'Cea mai recenta nota a studentului cu ID-ul ' || pi_id_student || ' este ' || nota_recenta || '.';
  RETURN mesaj;
EXCEPTION
WHEN no_data_found THEN
  SELECT COUNT(*) INTO counter FROM studenti WHERE id = pi_id_student ;
  IF counter = 0 THEN
    raise_application_error (-20001,'Studentul cu ID-ul ' || pi_id_student || ' nu exista in baza de date.');
  ELSE
    SELECT COUNT(*) INTO counter FROM note WHERE id_student = pi_id_student;
    IF counter = 0 THEN
      raise_application_error (-20002,'Studentul cu ID-ul ' || pi_id_student|| ' nu are nici o nota.');
    END IF;
  END IF;
END nota_recenta_student;
```

În caz că studentul nu există în baza de date returnăm apelantului codul de eroare -20001 și un mesaj personalizat iar în caz că studentul există dar nu are note returnăm codul de eroare -20002 si alt mesaj.

- definim excepția, o apelăm și apoi o aruncăm în exterior (varianta mai complicată dar cu avantajul că putem refolosi excepțiile declarate).

```
CREATE OR REPLACE FUNCTION nota_recenta_student(
    pi_id_student IN studenti.id%type)
  RETURN VARCHAR2
AS
  nota_recenta       INTEGER;
  mesaj              VARCHAR2(32767);
  counter            INTEGER;
  student_inexistent EXCEPTION;
  PRAGMA EXCEPTION_INIT(student_inexistent, -20001);
  student_fara_note EXCEPTION;
  PRAGMA EXCEPTION_INIT(student_fara_note, -20002);
BEGIN
  SELECT COUNT(*) INTO counter FROM studenti WHERE id = pi_id_student ;
  IF counter = 0 THEN
    raise student_inexistent;
  ELSE
    SELECT COUNT(*) INTO counter FROM note WHERE id_student = pi_id_student ;
    IF counter = 0 THEN
      raise student_fara_note;
    END IF;
  END IF;
SELECT valoare
INTO nota_recenta
FROM
  (SELECT valoare
  FROM note
  WHERE id_student = pi_id_student
  ORDER BY data_notare DESC
  )
WHERE rownum <= 1;
mesaj        := 'Cea mai recenta nota a studentului cu ID-ul ' || pi_id_student || ' este ' || nota_recenta || '.';
RETURN mesaj;
EXCEPTION
WHEN student_inexistent THEN
  raise_application_error (-20001,'Studentul cu ID-ul ' || pi_id_student || ' nu exista in baza de date.');
WHEN student_fara_note THEN
  raise_application_error (-20002,'Studentul cu ID-ul ' || pi_id_student || ' nu are nici o nota.');
END nota_recenta_student;
```

Rezultatul este același, doar că am definit noi excepția. Acest stil are avantajul că am putea, într-un pachet separat de excepții (să-i spunem chiar excepții), să definim o listă de excepții, fiecare având codul și mesajul său. Este mai elegant să definim într-un singur loc excepțiile și doar să le apelăm în diverse locuri. În acest fel evităm duplicarea codului și hardcodările.

```
when excepții.student_inexistent then
[cod]
```

Pentru a vedea ce anume se executa si ce anume nu se mai executa cand apare o exceptie, va lasam sa rulati urmatorul exemplu:

```
declare
  testex exception;
  PRAGMA EXCEPTION_INIT(testex, -20001);
begin
  begin
    raise testex;
    dbms_output.put_line('Cod ce nu se va mai executa... se va sari la tratarea exceptiei.');
    exception
      when testex then
        dbms_output.put_line('Exceptia.');
    dbms_output.put_line('Cod ce se va executa... pentru ca exceptia a fost deja tratata.');
  end;
  dbms_output.put_line('Cod ce se va executa... pentru ca nu este in acelasi bloc cu cel in care s-a produs exceptia.');
end;
```

NU PRINDETI EXCEPTIA IN ACELASI BLOC IN CARE ATI ARUNCAT-O !!!

Scopul unei exceptii este de a informa apelantul unei functii sau a unei proceduri de aparitia unei situatii neprevazute in timpul rularii acelui bloc (functie sau procedura). Daca prindeti exceptia in acelasi bloc (cum am facut noi in acest laborator pentru a exemplifica majoritatea exceptiilor), nu este corect deoarece apelantul va ramane cu impresia ca totul a decurs ok si ca rezultatul este cel pe care il astepta.

Incercati sa explicati ce s-ar intampla daca spre exemplu nu ar fi aruncata exceptia ZERO_DIVIDE, sau ce s-ar intampla daca selectul ce ar trebui sa arunce exceptia NO_DATA_FOUND ar trata singur aceasta exceptie si ar returna niste valori la intamplare in variabilele in care se facea selectia... v-ar placea un astfel de comportament ?

Din acest motiv (ca nu v-ar placea - sper ca asta era si raspunsul vostru), ar trebui ca sa aveti grija ca in functiile voastre doar sa aruncati o exceptie si sa lasati pe cel care a apelat functia sa o trateze (ca sa isi dea si el seama ca a gresit trimitand acei parametri functiei voastre).

Sa consideram in continuare o functie care incrementeaza valoarea trimisa ca parametru, doar daca aceasta nu este zero. In caz ca este transmis zero, ea va arunca o exceptie si isi va incheia rularea (evident nemaintorcand nici o valoare).

```
create or replace function test_ex (p_var IN INT)
RETURN INT as
    testex exception;
    PRAGMA EXCEPTION_INIT(testex, -20001);
begin
    if (p_var = 0) then raise testex;
    else return p_var+1;
    end if ;
end;
```

In caz ca functia este apelata avand ca parametru valoarae 0, exceptia custom definita cu codul de eroare 20001 va fi aruncata in speranta ca acel cineva care a apelat functia o va prinde (in cel mai rau caz cu "others" - pentru ca nu stie despre ce e vorba, in cel mai bun caz, chiar prin intermediul codului de eroare 20001 care poate fi asociat unei exceptii definite in acea locatie in care s-a apelat functia:

```
DECLARE
    ex_1 exception;
    PRAGMA EXCEPTION_INIT(ex_1, -20001);
BEGIN
    dbms_output.put_line(test_ex(0));
EXCEPTION
    WHEN ex_1 then --poate fi inlocuit si cu WHEN OTHERS, caz in care nu mai trebuie sectiunea de declare, dar prinde
tot...
        dbms_output.put_line('L-am prins pe 0');
END;
```

Concluzia ce trebuie retinuta este: NICIODATA NU PRINDETI EXCEPTIA PE CARE ATI ARUNCAT-O IN ACELASI BLOC IN CARE ATI ARUNCAT-O (si nu ma refer numai la PLSQL ci si la Java, PHP, JavaScript sau orice alt limbaj capabil de a transmite mesaje intre blocurile de cod prin intermediul exceptiilor).

= Exercitii (5pt) =

Dupa cum puteti observa din scriptul de creare, toti studentii au note la materia logica.

Asta inseamna ca o noua nota nu ar trebui sa fie posibil sa fie inserata pentru un student si

pentru aceasta materie (nu poti avea doua note la aceeasi materie).

Construiti o [https://profs.info.uaic.ro/~bd/wiki/index.php/Laboratorul_11#ALTER_TABLE constrangere] care sa arunce o exceptie cand regula de mai sus este incalcata (poate fi unicitate pe campurile id_student+id_curs, index unique peste aceleasi doua campuri sau cheie primara peste cele doua).

Prin intermediul unui script PLSQL incercati de 1 milion de ori sa inserati o nota la materia logica.

Pentru aceasta aveti doua metode:

- sa vedeti daca exista nota (cu count, cum deja ati mai facut) pentru studentul X la logica si sa inserati doar daca nu exista.

- sa incercati sa inserati si sa prindeti exceptia in caz ca aceasta este aruncata.

Implementati ambele metode si observati timpii de executie pentru fiecare dintre ele. (3pct)

Construiti o functie PLSQL care sa primeasca ca parametri numele si prenumele unui student si care sa returneze media si, in caz ca nu exista acel student (dat prin nume si prenume) sa arunce o exceptie definita de voi. Dintr-un bloc anonim care contine intr-o structura de tip colectie mai multe nume si prenume (trei studenti existenti si trei care nu sunt in baza de date), apelati functia cu diverse valori. Prindeti exceptia si afisati un mesaj corespunzator atunci cand studentul nu exista sau afisati valoarea returnata de functie daca studentul exista. (2pct)

= Tema (2pt) =

Demonstrati prinderea unei exceptii aruncate de catre aplicatia PLSQL intr-un limbaj de programare la algere (aruncati din PLSQL si prindeti in JAVA / PHP / ce vreti voi).

Practică sisteme de gestiune pentru baze de date

Laborator

- **Instalare**
- **Ghid SQL*Plus**
- **SQL Developer**
- **RE-SQL**
- **Views**
- **Intro PLSQL**
- **Blocuri Anonime**
- **Variabile Operatori**
- **SELECT INTO**
- **Structuri de control**
- **Structuri iterative**
- **Cursoare**
- **Subprograme**
- **Colectii**
- **Pachete**
- **Obiecte**
- **Exceptii**
- **Triggere**
- **Proiecte**

Triggere (declansatoare)

Triggerele sunt blocuri de cod care se executa automat, de obicei cand o operatie de tip DML este executata in baza de date. Acestea au un nume (identificator) si pot fi activate sau dezactivate utilizand acest nume.

Atunci cand un trigger este creat se specifica si cand acesta va fi executat in mod automat. Aceasta executie este asociata cu o anumita operatie care se efectueaza asupra unei tabele, asupra unui view (triggerele peste tabele/view sunt de tip DML) a unei scheme de baze de date (de tip DDL) sau chiar asupra intregii bazei de date (de tip system). Impreuna cu tipul operatiei ce va lansa in executie trigerul, se specifica si cand se doreste executarea acestuia: de exemplu, daca vrem sa facem o operatie de stergere dintr-o tabela si dorim ca valoarea ce este stearsa sa fie copiata intr-o alta tabela de bkup, este firesc ca sa dorim executarea triggerului inainte ca stergerea efectiva sa fie efectuata - in acest fel aveam acces la valoarea ce va fi stearsa si putem sa o copiem in tabela de bkup.

lata care ar fi motivele pentru care am dori sa utilizam un trigger:

- generarea automata de valori intr-o coloana
- realizarea de LOG-uri
- realizarea de statistici
- modificarea datelor dintr-o tabela atunci cand este executata o operatie intr-un view
- asigurarea integritatii dintre chei primare/straine atunci cand tabelele nu sunt in acelasi tablespace (de exemplu tabela studenti este pe un calculator si tabela note este pe alt calculator si vrem ca atunci cand inseram o nota sa verificam daca exista cheia in tabela stocata pe celalalt calculator)
- publicarea de evenimente cand sunt facute anumite opreatii in baza de date (de exemplu afisarea automata in consola ca cineva incearca sa stearga anumite date dintr-o tabela).
- interzicearea operatiilor de tip DML intr-un anumit interval orar (de exemplu pentru a nu se putea pune note decat in ziua examenului)
- interzicerea tranzactiilor incorecte sau restrictionarea pe baza unor reguli complexe ce nu pot fi obtinute doar prin chei primare/straine, unicitate sau alte constrangeri la nivel de tabela (de exemplu am putea sa permitem sa avem mai multe inregistrari cu un acelasi identificator dar care sa aiba suma unui anumit camp mai mica decat o anumita valoare).

Triggere de tip DML (cu BEFORE / AFTER)

Pot fi construite pentru operatii de tip delete, insert, update. Tinand cont de momentul in care este executat triggerul relativ la momentul in care se face operatia asupra tabelei, triggerele pot fi BEFORE sau AFTER.

Cand zicem BEFORE ne putem referi la momentul executiei comenzii de tip DML sau putem sa specificam o granularitate mai mare si sa executam triggereul inainte de modificarea fiecarui rand ce va fi transformat de operatia DML. De asemenea, un trigger care se executa pentru un anumit rand poate avea acces la informtaiile care existau in acel rand (sau care vor exista).

Sa vedem un trigger care se executa inainte ca o operatie de tip insert, delete sau update sa fie executata:

```
set serveroutput on;

CREATE OR REPLACE TRIGGER dml_stud
   BEFORE INSERT OR UPDATE OR DELETE ON studenti
BEGIN
  dbms_output.put_line('Operatie DML in tabela studenti !');
  -- puteti sa vedeti cine a declansat triggerul:
  CASE
     WHEN INSERTING THEN DBMS_OUTPUT.PUT_LINE('INSERT');
     WHEN DELETING THEN DBMS_OUTPUT.PUT_LINE('DELETE');
     WHEN UPDATING THEN DBMS_OUTPUT.PUT_LINE('UPDATE');
     -- WHEN UPDATING('NUME') THEN .... // vedeti mai jos trigere ce se executa doar la modificarea unui camp
  END CASE;
END;
/


delete from studenti where id=10000;
```

Puteti sa vedeti efectul BEFORE/AFTER compiland urmatoarele doua triggere:

```
set serveroutput on;

CREATE OR REPLACE TRIGGER dml_stud1
   BEFORE INSERT OR UPDATE OR DELETE ON studenti
declare
   v_nume studenti.nume%type;
BEGIN
  select nume into v_nume from studenti where id=200;
  dbms_output.put_line('Before DML TRIGGER: ' || v_nume);
END;
/

CREATE OR REPLACE TRIGGER dml_stud2
   AFTER INSERT OR UPDATE OR DELETE ON studenti
declare
   v_nume studenti.nume%type;
BEGIN
  select nume into v_nume from studenti where id=200;
  dbms_output.put_line('After DML TRIGGER: ' || v_nume);
END;
/
```

si executand apoi:

```
update studenti set nume='NumeNou' where id=200;
```

Aceste triggere se vor executa totusi chiar si atunci cand se face update la un alt camp sau la un alt ID (si va afisa mereu informatiile privitoare la studentul cu IDul 200).

Pentru a face referinta la valorile din randul curent va face un trigger care se executa la operatia pe un rand. Iata un astfel de exemplu care va afisa notele vechi si pe cele noi de fiecare data cand se va face un update (chiar si pe mai multe randuri):

```
set serveroutput on;
CREATE OR REPLACE TRIGGER marire_nota
  before UPDATE OF valoare ON note   -- aici se executa numai cand modificam valoarea !
  FOR EACH ROW
BEGIN
  dbms_output.put_line('ID nota: ' || :OLD.id); -- observati ca aveti acces si la alte campuri, nu numai la cele
modificate...
  dbms_output.put_line('Vechea nota: ' || :OLD.valoare);
  dbms_output.put_line('Noua nota: ' || :NEW.valoare);

  -- totusi nu permitem sa facem update daca valoarea este mai mica (conform regulamentului universitatii):
  IF (:OLD.valoare>:NEW.valoare) THEN :NEW.valoare := :OLD.valoare;
  end if;
END;
/
```

update note set valoare =8 where id in (1,2,3,4);
In cazul in care modificarea se face intr-un nested table dintr-unul din campuri, puteti avea acces la randul ce contine acel nested table prin :PARENT. Atunci cand faceti o operatie de tip insert valoarea :OLD este NULL (pentru ca nu exista) si la fel pentru :NEW intr-o operatie de tip delete.

Dupa cum ati vazut, ca sa evitam sa punem o nota mai mica peste o nota mai mare, am putut modifica valoarea din :NEW cu altceva in locul valorii adevarate. Nu puteti face acest lucru pentru valoarea :OLD (ea fiind deja existenta in tabela). Evident, daca operatia este delete, nu este posibil sa modificati nici valoarea :NEW (deoarece ea trebuie sa fie NULL).

De asemenea, puteti vedea ca triggerul este construit cu BEFORE. Nu puteti modifica valoarea :NEW intr-un trigger de tip AFTER (pentru ca deja valoarea a fost scrisa in tabela cand ajunge sa fie executat triggerul).

Daca o modificare lanseaza in executie doua triggere, unul de tip BEFORE si unul de tip AFTER si cel de tip BEFORE modifica valoarea din :NEW, triggerul after va vedea valoarea modificata (tot in :NEW). (va vedea si valoarea :OLD - aceeasi care este si in triggerul before).

Aparitia erorilor de tip Mutating Table

Atunci cand facem o operatie DML si din cauza acesteia se executa un trigger, acest trigger nu are voie sa citeasca datele din tabela care este in curs de modificare. Incercarea de a face un select asupra acestei tabele va genera o eroare de tipul Mutating table. Iata un exemplu:

```
create or replace trigger mutate_example
after delete on note for each row
declare
   v_ramase int;
begin
   dbms_output.put_line('Stergere nota cu ID: '|| :OLD.id);
   select count(*) into v_ramase from note;
   dbms_output.put_line('Au ramas '|| v_ramase || ' note.');
end;
/
delete from note where id between 101 and 110;
/
```

Evident, in momentul in care o exceptie este aruncata, nici modificarea din DML nu va avea loc (puteti verifica dupa aceea ca de fapt nu s-a modificat nimic in tabela si nota cu id-ul 101 este inca acolo).

Pentru a evita erorile de tip mutating table exista doua metode:

- Utilizarea triggerilor compusi
- Utilizarea unei tabele temporare

Un trigger compus este de fapt o compunere a celor 4 tipuri de trigere disponibile: BEFORE, BEFORE EACH ROW, AFTER EACH ROW, AFTER (aceasta este si ordinea in care se exacuta triggerele in cazul in care sunt mai multe). Solutia pentru problema anterioara este sa construim un trigger care sa afiseze doar la sfarsit numarul de linii ramase (numai triggerele de tipul "for each row" vor crea mutating table). Iata cum arata un astfel de trigger compus:

```
set serveroutput on;

CREATE OR REPLACE TRIGGER stergere_note
FOR DELETE ON NOTE
COMPOUND TRIGGER
  v_ramase INT;

  AFTER EACH ROW IS
  BEGIN
     dbms_output.put_line('Stergere nota cu ID: '|| :OLD.id);
  END AFTER EACH ROW;

  AFTER STATEMENT IS BEGIN
     select count(*) into v_ramase from note;
     dbms_output.put_line('Au ramas '|| v_ramase || ' note.');
  END AFTER STATEMENT ;
END stergere_note;
/
delete from note where id between 241 and 250;
```

Daca aveti doua triggere care au acelasi tip de declansare (de exemplu ambele sunt de tip for each row), puteti sa controlati ordinea in care acestea se eecuta cu FOLLOWS respectiv PRECEDES.

Pentru a activa/dezactiva un trigger puteti sa il modificati astfel:

```
ALTER TRIGGER STERGERE_NOTE DISABLE; -- SAU ENABLE
```

Triggere de tipul instead of

Acest tip de triggere nu poate fi construit pentru operatii DML ce sunt efectuate asupra tabelelor ci doar daca aceste operatii afecteaza un view. Sa construim asadar un view:

```
create view std as select * from studenti;
```

Este destul de complicat sa stergem din tabela studenti... nu si daca avem trigere. Acest triger poate fi facut si direct peste tabel (cu before sau after) dar de aceasta data il vom realiza peste view-ul creat anterior. Triggerul are rolul de a sterge toate informatiile din tabelele aditionale atunci cand dorim sa stergem un student:

```
CREATE OR REPLACE TRIGGER delete_student
  INSTEAD OF delete ON std
BEGIN
  dbms_output.put_line('Stergem pe:' || :OLD.nume);
  delete from note where id_student=:OLD.id;
  delete from prieteni where id_student1=:OLD.id;
  delete from prieteni where id_student2=:OLD.id;
  delete from studenti where id=:OLD.id;
END;
```

Si sa testam triggerul prin stergerea unui student:

```
delete from std where id=75;
```

Triggere DDL

Din punctul de vedere al momentului cand sunt executate, triggerele de sistem pot fi de tipul before, after sau instead.

Din punctul de vedere al evenimentului ce poate fi tratat, pot fi triggere ce au loc cand este modificata schema, de baze de date sau de tipul instead of create.

In urmatorul exemplu, se va interzice utilizarea oricarei comenzi de tipul drop. Aceasta poate fi o problema dat fiind faptul ca pentru a elimina triggerul drop_trigger trebuie sa initiem chiar o comanda de tipul drop. Ati putea, ca inaintea emiterii erorii sa interogati valoarea variabilei ora_dict_obj_name pentru a testa daca drop-ul pe care vreti sa il interziceti este pentru un singur (acel) obiect pentru care nu vreti sa se faca drop.

```
CREATE OR REPLACE TRIGGER drop_trigger
  BEFORE DROP ON student.SCHEMA
  BEGIN
    RAISE_APPLICATION_ERROR (
      num => -20000,
      msg => 'can''t touch this');
  END;
/

DROP TABLE NOTE;
```

In Oracle se pot face triggere care sa fie activate in momentul in care se executa o operatie de create:

```
CREATE OR REPLACE TRIGGER t
  INSTEAD OF CREATE ON SCHEMA
  BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE T (n NUMBER, m NUMBER)';
  END;
/
create table a(x number); -- de fapt va crea tabelul T.
```

Lista tuturor trigerelor de tip DDL poate fi consultata aici:
https://docs.oracle.com/cd/E11882_01/appdev.112/e25519/triggers.htm#CHDGIJDB

Triggere Sistem

Inafara triggerelor pentru operatii de tip DDL exista doua tipuri de triggere prin intermediul carora puteti obtine diverse informatii atunci cand anumite evenimente se intampla in sistem. De exemplu ati putea sa creati o tabela de log in care sa scrieti adresa IP a clientului care s-a conectat la baza de date sau numele acestuia. Sa cream un astfel de tabel in care sa stocam lista utilizatorilor ce se autentifica in sistem si orele la care aceste autentificari au fost facute (evident, acestea trebuie sa fie facute din contul de sys.... nimeni altcineva nu ar trebui sa aiba controlul utilizatorilor care se autentifica in sistemul de gestuine de baze de date).

```
create table autentificari(nume varchar2(30), ora timestamp);
/
CREATE OR REPLACE TRIGGER check_user
  AFTER LOGON ON DATABASE
DECLARE
  v_nume VARCHAR2(30);
BEGIN
  v_nume := ora_login_user;
  INSERT INTO autentificari VALUES(v_nume, CURRENT_TIMESTAMP);
END;
/
```

dupa care incercati sa va autentificati ca student, inapoi ca sys as sysdba si apoi vedeti continutul tabelei autentificari.

Puteti vedea mai multe informatii intr-un trigger de tip system. O lista completa a variabilelor ce le puteti interoga o veti gasi in tabelul de la pagina aceasta: https://docs.oracle.com/cd/E11882_01/appdev.112/e25519/triggers.htm#CHDCFDJG

Atunci cand nu mai aveti nevoie de triggere, puteti sa le eliminati cu drop trigger urmata de numele triggerului pe care doriti sa il eliminati.

Exercitii

Problema 1.

Creati doua tabele la alegere care sa fie legate printr-o cheie primara/straina si un

view care sa fie peste full outer joinul dintre cele doua tabele. Construiti triggere care sa permita

realizarea de operatii de tipul INSERT, UPDATE, DELETE pe view. Cand veti prezenta problema

veti avea in view macar 10 randuri (ca sa avem ce sterge de exemplu).

Problema 2.

Creati un trigger care sa scrie intr-o tabela de tip LOG cate note au fost modificate

de o comanda de tip update. Nu se va permite decat modificarea in plus a notelor.

Problema 3.

Creati un trigger prin intermediul caruia sa preveniti orice modificare destructiva asupra bazei de date:

- Eliminarea unei coloane dintr-o tabela
- Stergerea unui tabel
- Truncherea tabelului

Se va scrie intr-o tabela aditionala timpul la care s-a incercat actiunea si numele utilizatorului ce a incercat sa o faca (http://www.java2s.com/Code/Oracle/User-Previliege/Getcurrentusername.htm).

PS. veti avea nevoie de doi utilizatori care sa aiba acces in aceeasi schema de baze de date (check this: https://profs.info.uaic.ro/~bd/wiki/index.php/Doi_utilizatori_cu_aceeasi_schema).

PS2. Daca doar unul din utilizatori declanseaza triggerul pe care l-ati facut, veti pierde doua puncte.

Problema 4.

- Creati triggere care sa modifice o tabela de LOGuri atunci cand se face o operatie (insert/update/delete) in tabelul de note, asupra valorii. In acest tabel de LOGuri vor fi stocate ID-ul notei, vechea valoare, noua valoare, tipul operatiei si momentul in care s-a executat operatia si de catre cine (userul curent autentificat - SELECT USER FROM DUAL).
- Campul updated_at din tabelul note va fi modificat in conformitate (cu noul timp).
- Aceste operatii de modificare se vor efectua doar daca minutul sistemului este unul impar.
- Construiti triggere care sa nu permita modificarea, stergerea informatiilor din tabelul log si nici stergerea in intregime a acestui tabel.

Problema 5.

Sa se construiasca un view ca fiind joinul dintre tabelul studenti, note si cursuri, cu rolul de catalog: va contine numele si prenumele studentului, materia si nota pe care studentul a luat-o la acea materie.

Dupa cum va puteti da seama, operatii de genul INSERT nu sunt permise pe acest view din cauza ca ar trebui sa inserati datele in toate tabelele. Totusi, cu ajutorul unui trigger, puteti sa verificati existenta studentului (si sa il creati daca nu exista), existenta materiei (si sa o creati daca nu exista) sau a notei.

Construiti triggere pentru realizarea de operatii de tipul INSERT, UPDATE si DELETE pe viewul creat, care sa genereze date random atunci cand sunt adaugate informatii inexistente (de exemplu daca faceti insert cu un student inexistent, va genera un nr matricol, o bursa, grupa, an etc pentru acel student sau un numar de credite si un an, semestru pentru un curs, etc.)

Cazuri posibile:

- Stergerea unui student si totodata a dependentelor sale (pentru tabelele definite exact ca in scriptul de creare);
- Inserarea unei note la un curs pentru un student inexistent cu adaugarea studentului;
- Inserarea unei note la un curs pentru un curs inexistent - cu adaugarea cursului;
- Inserarea unei note cand nu exista nici studentul si nici cursul.
- Update la valoarea notei pentru un student - se va modifica valoarea campului updated_at. De asemenea, valoarea nu poate fi modificata cu una mai mica (la mariri se considera nota mai mare).

ex: INSERT INTO CATALOG VALUES ('Popescu', 'Mircea', 10, 'Yoga');

Problema 6.

Creati triggere care sa adauge intr-o tabela de LOG-uri operatiile DML (INSERT / UPDATE / DELETE) efectuate asupra tabelului note. In acest tabel de LOG-uri vor fi stocate ID-ul notei, vechea valoare, noua valoare, tipul operatiei, momentul in care s-a executat operatia si de catre cine (userul curent autentificat - SELECT USER FROM DUAL) - aveti nevoie de macar 2 utilizatori care sa aiba acces la aceiasi schema (check this: https://profs.info.uaic.ro/~bd/wiki/index.php/Doi_utilizatori_cu_aceeasi_schema respectiv http://www.java2s.com/Code/Oracle/User-Previliege/Getcurrentusername.htm).

Problema 7.

- Creati triggere care sa modifice o tabela de LOGuri atunci cand se face o operatie (insert/update/delete) in tabelul de note, asupra valorii. In acest tabel de LOGuri vor fi stocate ID-ul notei, vechea valoare, noua valoare, tipul operatiei si momentul in care s-a executat operatia si de catre cine (userul curent autentificat - SELECT USER FROM DUAL) - evti aevea nevoied e macar 2 utilizatori care sa aiba acces la aceiasi schema.
- Campul updated_at din tabelul note va fi modificat in conformitate (cu noul timp). (cu exceptia operatiei delete cand se va sterge in totalitate)
- Aceste operatii de modificare se vor efectua doar daca minutul sistemului este unul impar.

- Construiti o functie care sa permita o operatie de "ROLLBACK" a tabelului note relativ la un anumit moment de timp. De exemplu daca vreau sa refac tabelul de note asa cum era acum 90 de minute, sa pot executa "functie_rollback(90);". Aceasta functie, va prelua din tabelul de LOGuri modificarile din acea perioada (eg. ultimele 90 de minute) si va realiza operatiile inverse (daca s-a sters ceva, se va insera | daca s-a inserat ceva, se va sterge | daca s-a modificat o nota, se va modifica cu valoarae initiala). Operatiile trebuie efectuate in ordine inversa (prima executata va fi cea care 'reapara' ultima operatie scrisa in LOG).

PS. veti avea nevoie de doi utilizatori care sa aiba acces in aceeasi schema de baze de date (check this: https://profs.info.uaic.ro/~bd/wiki/index.php/Doi_utilizatori_cu_aceeasi_schema).

Problema 8.

Construiti un trigger care sa verifice ca nota cu care se incearca modificarea notei curente este mai mare decat aceasta si sa nu permita inlocuirea notei cu una mai mica. Daca modificarea s-a efectuat atunci triggerul va afisa un mesaj de forma ". Nota lui la materia era si acum este ." Unde nr este numarul modificarii, restul sunt valori din tabelele utilizate la laborator. In final se va afisa si cate note au fost cu adevarat modificate (pentru ca s-ar putea ca unele sa nu respecte conditia de a fi mai mare decat valoarea existenta.

Problema 9.

Din PHP, java asu alt limbaj la alegere, accesati o tabela din baza de adte studenti. Construiti doua astfel de scripturi: unul care sa permita SQL injection si altul care sa utilizeze variabile transmise dinamic (prin acest mod impiedicandu-se SQL injection). Dati exemplu de injectie ce functioneaza in primul caz, dar nu si in al doilea.