ORACLE – PL-SQL

Blocs – Procédures stockées – Fonctions - Triggers

Bertrand LIAUDET

SOMMAIRE

Dernière édition : février 2018

PL-SQL – ORACLE

1. Programme et script

Structure d'un programme : le bloc

On peut déclarer des blocs de code dans SQL*PLUS

```
[ DECLARE
…]
BEGIN
…
[ EXCEPTION
…]
END ;
```

Au minimum : BEGIN et END ;

Programme en ligne

code

```
begin
   dbms_output.put_line('Bonjour');
end;
/

Procédure PL/SQL terminée avec succès.
```

dbms.output_line permet d'afficher du texte.

Commande « / »

Le « / » enregistre le bloc et demande l'exécution du bloc en cours.

Si on rappelle le /, ça exécute le bloc en cours.

Le « / » rappelle toujours la dernière commande passée.

Serveroutput à ON

L'exécution est correcte, mais rien ne s'affiche !

Il faut passer le paramètre « serveroutput » à ON.

```
SQL> show serveroutput
serveroutput OFF
SQL> set serveroutput ON
```

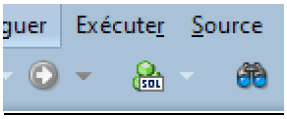
On peut relancer l'exécution avec « / »

```
SQL> /
```

```
Bonjour

Procédure PL/SQL terminée avec succès.
SQL>
```

SQL Developper



Aller sur l'icône Feuille de calcul SQL (ALT-F10).

Coller le code (sans le /)

Une fenêtre « sortie de script » s'ouvre. Il se peut que rien ne s'affiche.

Il faut ajouter la commande : set serveroutput ON.

Il est possible qu'il faille supprimer les tabulations pour éviter les caractères cachés.

Il est possible qu'il faille relancer plusieurs fois l'exécution pour que ça finisse par fonctionner !

Dans la fenêtre « sortie de script », on peut utiliser l'onglet « effacer » (gomme rouge) pour vider la fenêtre.

Il est possible qu'il faille aller dans affichage/sortie SGBD.

Autres paramètres : timing à ON

On peut aussi afficher la durée d'exécution des instructions :

```
SQL> show timing
timing OFF
SQL> set timing ON
```

Run

Le RUN affiche le bloc en plus de l'exécuter.

```
SQL> run
  1  begin
  2  dbms_output.put_line('Bonjour');
  3  end;
  4  /
Bonjour

Procédure PL/SQL terminée avec succès.
SQL>
```

clear buffer

On peut vider le buffer :

```
SQL> clear buffer
buffer effacé
SQL> /
SP2-0103: Rien à exécuter dans la mémoire tampon SQL.
```

Script

Les programmes peuvent être saisis directement sous SQLPLUS ou enregistrés dans des scripts.

```
-- test.sql
-- script qui affiche bonjour
```

```
begin
    dbms_output.put_line('Bonjour');
end;
/
```

Exécution du script

```
SQL> @test
Bonjour

Procédure PL/SQL terminée avec succès.

SQL>
```

Commentaires

```
    /*   script de définition d'une procédure
         procédure « bonjour » : affiche bonjour,
         usage des commentaires en style C
    */

    -- commentaires derrière deux tirets et un espace
```

Affichage : dbms_output.put_line

Un seul argument.

Type au choix : chaîne, number ou une date.

Version 1

```
begin
  dbms_output.put_line('Bonjour '||user||'. Nous sommes le
    '||to_char(sysdate, 'dd month yyyy'));
end;
/

Bonjour TPSQL. Nous sommes le 09 février   2018
Procédure PL/SQL terminée avec succès.
```

Version sans espace après le mois :

```
begin
   dbms_output.put_line('Bonjour '||user||'. Nous sommes le
    '||rtrim(to_char(sysdate, 'dd month'))||'
    '||to_char(sysdate,'yyyy'));
end;
/

Bonjour TPSQL. Nous sommes le 09 février 2018
```

Instructions SQL

Pas de SELECT classique en PL-SQL

Pas de DDL en PL-SQL : CREATE TABLE, DROP TABLE, ALTER TABLE

```
begin
  select * from emp;
end;
/

select * from emp;
*
ERREUR à la ligne 2 :
ORA-06550: Ligne 2, colonne 1 :
PLS-00428: une clause INTO est attendue dans cette instruction
   SELECT
```

L'erreur dit : « une clause INTO est attendue » : le INTO ça permettra de mettre le résultat d'un SELECT dans une variable.

Seules les instructions du DML sont autorisées : INSERT, UPDATE, DELETE

```
begin
  update emp set comm = comm + 0.1*comm
  where job='SALESMAN';
end;
/

Procédure PL/SQL terminée avec succès.
SQL>
```

On vérifie le bon fonctionnement du bloc en regardant les tuples avant et après la modification :

```
Select * from emp where job='SALESMAN';

begin
  update emp set comm = comm + 0.1*comm
  where job='SALESMAN';
end;
/

Select * from emp where job='SALESMAN';
```

Blocs imbriqués

On peut déclarer des blocs d'instructions à tout moment dans un bloc.

Quand on déclare un bloc d'instruction, on peut y associer de nouvelles déclarations de variables.

2. Variables et types scalaires

Variable globales - DEF - Variables de substitution

Principes : &&

Dans SQL*PLUS, Il existe des variables de niveau session.

On peut les créer avec la commande DEF.

On peut les utiliser dans un SELECT avec un &&.

La variable de substitution n'est visible qu'au niveau de SQL*PLUS : elle ne peut pas être utilisée par du code serveur (code serveur PHP ou JAVA par exemple). Autrement dit, elle n'est pas enregistrée dans la BD.

```
SELECT * FROM emp WHERE EMPNO=&&EMPNO;
```

L'interface SQL*PLUS demande :

```
Enter value for EMPNO: 7839
```

On saisit une valeur et on a le résultat pour la valeur saisie :

```
EMPNO    ENAME    JOB         MGR    HIREDATE    SAL    COMM      DEPTNO
----------------------------------------------------------------------
7839     KING     PRESIDENT          17/11/06    5000              10
```

La variable EMPNO est définie après la requête : c'est une variable globale.

Une variable définie avec un && garde sa valeur en dehors de la requête de définition.

```
SQL> DEF EMPNO

DEFINE EMPNO                = "7839" (CHAR)
```

C'est un (CHAR).

On a saisi 7839, il est interprété comme une chaine de caractères.

Si on saisi '7839', ça marche aussi

On peut aussi faire un SELECT FROM DUAL avec &EMPNO

```
SQL> SELECT &EMPNO FROM DUAL;
```

La variable est accessible avec un & (un seul). Si on met && ça marche aussi.

Visibilité limitée au SELECT : simple &

Si on écrit

```
UNDEF EMPNO;
SELECT * FROM dept WHERE DEPTNO=&DEPTNO;
DEF EMPNO;
```

La variable &DEPTNO n'est pas définie après le SELECT.

Une variable définie avec un & n'a sa valeur que dans la requête de définition.

Autres exemples :

- ***Chaine de caractères***

```
SELECT * FROM emp WHERE ename = &ename;
```

```
Entrez une valeur pour ename : 'KING'
```

Il faut saisir la chaine entre apostrophes. Sinon, on aura « ename = KING » ce qui ne marche pas.

- ***Avec utilisation du like***

```
SELECT * FROM emp WHERE ename like &ename ;
Entrez une valeur pour v_nomEmp : 'A%'
```

Ou encore

```
Entrez une valeur pour var_subst : '%' and JOB = 'MANAGER'
```

On aura ainsi contourné la requête initiale !

- ***Variable dans le FROM et entre apostrophes***

```
SELECT *
FROM &TAB
WHERE SAL > &SAL
AND JOB = '&JOB';
```

Dans ce cas, on saisit JOB sans apostrophes.

Création d'une variable

```
DEF[INE] maVar = texte
```

- ***remarques***

- La variable est accessible pendant toute la session.
- C'est un CHAR (chaîne de caractères). Toutefois, il pourra s'exploiter comme un NUM.

- ***exemples***

```
DEF[INE] maVar = 30
```

Ou

```
DEF[INE] maVar = DALLAS
```

A noter qu'on peut écrire indifféremment sans guillemets, avec guillemets ou avec apostrophes.

Suppression d'une variable

```
UNDEF[INE] maVar
```

Afficher la valeur d'une variable

```
DEF[INE] maVar
```

Afficher toutes les variables

```
DEF[INE]
```

Saisie d'une valeur

```
ACCEPT maVar NUM prompt 'entrez un nombre : '
ACCEPT maVar CHAR prompt 'entrez du texte : '
```

ACCEPT permet de définir le texte pour la saisie. Il définit aussi la variable si ça n'a pas déjà été fait.

Variable globale – bind variable - VAR

Principes

On peut définir des variables qui pourront circuler dans les blocs indépendamment de SQL*PLUS.

La variable de liaison (link) est visible au niveau de la BD : elle peut être utilisée par du code serveur (code serveur PHP ou JAVA par exemple).

Toutefois, on peut aussi utiliser ces variables et les déclarer dans SQL*PLUS.

On peut les créer avec la commande VAR dans SQL*PLUS.

Exemple 1

- ***Création de la variable dans SQL*PLUS : VAR [IABLE] [nom [type]]***

```
VAR maVar VARCHAR2(10)
```

- ***Utilisation dans un bloc : « :maVar »***

```
Begin
  :maVar := 'bonjour';
  dbms_output.put_line(:maVar);
end;
/
```

On accède à la variable avec un « : » devant : :maVar.

Exemple 2

- ***Création et initialisation de la variable dans SQL*PLUS***

```
VAR maVar2 VARCHAR2(10);
EXEC :maVar2 := 'bonjour2';
Print :maVar2 ;
```

EXEC permet d'exécuter une commande PLSQL, ici une affectation (:=)

Print permet d'afficher le contenu de la variable dans SQL*PLUS

- ***Utilisation dans un bloc :***

```
BEGIN
   dbms_output.put_line(:maVar2);
END;
/
```

Afficher toutes les variables

```
VAR[IABLE]
```

Variable locale : DECLARE : zone des variables et des types

```
DECLARE
    [TYPE nomType IS definitionDuType;]
    [...]
    NomVar [CONSTANT] nomType [NOT NULL] [ {DEFAULT | :=} VALEUR ];
    [...]
BEGIN
…
[ EXCEPTION
…]
END ;
```

Les CONSTANT sont associés à un DEFAULT.

Une variable déclarée NOT NULL doit avoir une valeur d'initialisation : elle est donc associée à un « := ».

Exemple

```
DECLARE

    f FLOAT;

    X NUMBER DEFAULT 123.456;

    maConstante CONSTANT VARCHAR2(20):='constante' ;

    v_deptno EMP.DEPTNO%TYPE;

    SUBTYPE CHIFFRE is NUMBER(1,0); -- sur 1 chiffre, 0 après la
    virgule
    n CHIFFRE;

BEGIN
    dbms_output.put_line('Ma constante : '|| maConstante);
    dbms_output.put_line('Number saisi : '|| x);
    n:=&chiffre;
    dbms_output.put_line('Chiffre saisi : '|| n);
    f:=&float;
    dbms_output.put_line('Flottant saisi : '|| f);

    SELECT deptno INTO v_deptno
    FROM dept WHERE deptno=10 ;
    dbms_output.put_line('v_deptno sorti du select : '|| v_deptno);

END;
/
```

Nommer les variables

Il faut faire attention à ne pas confondre le nom des variables PL-SQL avec celui des attributs des tables.

Pour éviter ça : on préfixe toutes les variables PL_SQL d'un « v_ ».

En cas d'ambiguïté, PL-SQL utilise toujours l'attribut de table.

%TYPE : Utiliser un type provenant d'un attribut d'une table ou d'une variable

Présentation

Partout où on utilise un nom de type (nomType), on peut utiliser un type provenant d'un attribut d'une table de la BD.

Exemple

Traité dans le premier exemple.

Syntaxe

```
NomVar [CONSTANT] {nomTable.nomAttribut | nomVariable}%TYPE [NOT
    NULL] [ {DEFAULT | :=} VALEUR ];
```

SUBTYPE : création d'un type avec restriction du domaine de valeur

Présentation

On peut créer des types qui restreignent le domaine des valeurs autorisées par le type original.

On peut aussi créer des types qui renomment des types existants.

Exemple

Traité dans le premier exemple.

Syntaxe

```
SUBTYPE nomSousType IS nomType [(contraintes)][NOT NULL];
```

TOUS LES TYPES

Les caractères

CHAR(n)	Chaîne fixe de n caractères.	n octets. 2000 caractères max
NCHAR(n)	Chaîne fixe de n caractères. Format Unicode	n octets. 2000 caractères max
VARCHAR2(n)	Chaîne variable de n caractères.	Taille variable. 4000 caractères max
NVARCHAR2(n)	Chaîne variable de n caractères. Format Unicode	Taille variable. 4000 caractères max
CLOB	Flot de caractères	Jusqu'à 4 Giga
NCLOB	Flot de caractères. Format Unicode	Jusqu'à 4 Giga
LONG	Flot de caractères	Jusqu'à 2 Giga. Obsolète

Unicode est une méthode de codage universelle utilisée par XML, Java, Javascript, LDAP et WML.

- ***Sous-types***

NOM	**Type d'origine**	**Caractéristiques**
CHARACTER	CHAR	Identique à CHAR

Les données numériques

NUMBER (N, D)	N chiffres en tout, dont D pour la partie décimale. Avec N+D <39.	21 octets max. De +/- 1*10 –130 à +/- 9.99*10 +125

- ***Sous-types***

NOM	**Type d'origine**	**Caractéristiques**
INTEGER	NUMBER (38, 0)	Taille variable. 4000 caractères max
PLS_INTEGER	NUMBER	Entiers signées. Moins coûteux qu'un NUMBER. Plus performant pour les opérations mathématiques due BINARY_INTEGER. Les valeurs réelles sont arrondies à l'entier le plus proche.
BINARY_INTEGER	NUMBER	Entiers signées. Moins coûteux qu'un NUMBER.
NATURAL, POSITIVE	BINARY_INTEGER	Entiers positifs.
NATURALN, POSITIVEN	BINARY_INTEGER	Entiers positifs et non nul.
SIGNTYPE	BINARY_INTEGER	-1, 0 ou 1
DEC, DECIMAL, NUMERIC	NUMBER	Décimaux, precision de 38 chiffres
DOUBLE PRECISON, FLOAT, REAL	NUMBER	Flottants
INTEGER, INT, SMALLINT	NUMBER	Entiers sur 38 chiffres.

Les dates

DATE	Date et heure jusqu'aux secondes	7 octets
TIMESTAMP	Date et heure jusqu'aux fractions de secondes	7 à 11 octets. La précision des fractions de seconde va de 0 à 9, 6 par défaut.
INTERVAL YEAR TO MONTH	Période en années et mois	5 octets
INTERVAL DAY TO SECOND	Période en jours, heures, seconde	11 octets
TIMESTAMP WITH TIME ZONE	Timestamp avec décalage horaire par rapport au serveur.	13 octets
TIMESTAMP WITH LOCAL TIME ZONE	Timestamp avec décalage horaire par rapport au client.	7 à 11 octets

Les données binaires

BLOB	Données binaires	Jusqu'à 4 Gigas
BFILE	Données binaries stockés dans un fichier externe.	Jusqu'à 4 Gigas
RAW (size)	Données binaires	Jusqu'à 2000 octets. Obsolète
LONG RAW	Données binaires	Jusqu'à 2 Gigas. Obsolète

Les booléens (type logique)

BOOLEAN		TRUE, FALSE, NULL
SIGNTYPE	BINARY_INTEGER	-1, 0 ou 1

Le PL-SQL définit un type BOOLEAN. Un BOOLEAN peut prendre 3 valeurs : TRUE, FALSE ou NULL.

Le type SIGNTYPE peut aussi jouer le rôle de Booléen : TRUE = 1, FALSE = 0, NULL = -1.

Attention, le SQL ORACLE ne définit pas de type BOOLEAN, ni le type SIGNINT. Un attribut de table ne peut pas ni être de type BOOLEAN, ni être de type SIGNTYPE.

```
Declare a Boolean:=&verite;
Begin
    If a then
         dbms_output.put_line('vrai');
    elsif not(a) then
         dbms_output.put_line('faux');
    else
         dbms_output.put_line('null');
    end if;
end;
/
```

3. Procédures

Bloc procedure

On peut créer une procédure dans la zone DECLARE de la création d'un bloc.

La procédure peut être appelée dans la zone BEGIN – END par une simple référence à son nom, avec des parenthèses.

```
DECLARE
   PROCEDURE affichage IS
      BEGIN
         dbms_output.put_line(
         'Bonjour ' || user ||
         '. Nous sommes le ' ||
         rtrim(to_char(sysdate, 'dd month'))|| ' '||
         to_char(sysdate,'yyyy')
      );
   END;

BEGIN
   affichage;
END;
/

Bonjour TPSQL. Nous sommes le 08 février 2018
Procédure PL/SQL terminée avec succès.
SQL>
```

Création d'une procedure

On peut créer une procédure qui est enregistrée dans la BD.

```
CREATE or REPLACE PROCEDURE affichage IS
BEGIN
   dbms_output.put_line(
      'Bonjour ' || user ||
      '. Nous sommes le ' ||
      rtrim(to_char(sysdate, 'dd month'))|| ' '||
      to_char(sysdate,'yyyy')
   );
END ;
/

Procédure créée.
SQL>
```

Appel d'une procédure dans SQL*PLUS : call ou execute ou exec

Pour appeler une procédure dans le PL-SQL, on fait un CALL (ou exec[ute])

```
SQL> call affichage();

Bonjour BERTRAND. Nous sommes le 05 juin 2008

Appel terminé.
```

Appel d'une procédure dans une procédure ou une fonction

Pour appeler une procédure dans une procédure, on fait une simple référence au nom de la procédure qu'on appelle, avec des parenthèses.

```
CREATE or REPLACE PROCEDURE affichage1 IS
Begin
  affichage();
end ;
/


call affichage1() ;
```

Variable locale : IS ou AS : Zone des variables des procédures et des fonctions

Dans les procédures et les fonctions les variables locales sont situées entre IS ou AS et BEGIN.

Il n'y a pas de DECLARE.

```
DECLARE PROCEDURE test
{IS | AS}
    [TYPE nomTypeLocal IS definitionDuType;]
    [...]
    [nomVarLocale TYPE;]
    [...]
BEGIN
```

Paramètres en entrée d'une procédure

```
CREATE or REPLACE PROCEDURE AffEmpDept(v_deptno EMP.DEPTNO%TYPE) IS
    v_nbemp number;

BEGIN
    SELECT count(*) INTO v_nbemp
       FROM emp WHERE deptno=v_deptno;
    dbms_output.put_line('Le département '||v_deptno||
       ' contient '||v_nbemp||' employe(s)') ;
    dbms_output.put_line('.');
    FOR tuple IN (
         SELECT empno, ename, sal, deptno FROM emp
         where deptno = v_deptno
    )
    LOOP
         dbms_output.put_line(
            '.     Employé n°' || tuple.EMPNO||
             ' : ' || tuple.ENAME||', salaire = '||tuple.SAL);
    END LOOP;
END;
/
    `


call affempdept(10);
```

A noter le **SELECT … INTO** qui permet de récupérer un retour de SELECT dans une variable.

Et le **FOR tuple IN (SELECT ..) LOOP … END LOOP**, qui permet de boucler sur chaque tuple d'un SELECT.

Structure d'une procédure

```
CREATE [OR REPLACE] PROCEDURE nomProcedure [(
    nomVar [{IN | OUT | IN OUT }] TYPE
    [, ...]
)]
{IS | AS}
    [nomVarLocale TYPE;]
    [...]
BEGIN
    instructions;
[EXCEPTION
    instructions d'exception]
END [nomProcedure];
```

IS ou AS sont équivalent

On reviendra sur les paramètres en sortie (OUT) et en entrée-sortie (IN OUT)

On reviendra sur les EXCEPTIONS

Rappel de la sémantique du métalangage

- Majuscule : les mots-clés
- Entre crochets : ce qui est facultatif
- Entre accolades : proposition de plusieurs choix possibles. Les choix sont séparés par des barres verticales.
- 3 petits points avant un crochet fermant : ce qui précède sur la même ligne, ou la ligne précédente, ou le bloc précédent, peut être répété. Le texte qui précède les 3 petits points et est après le crochet ouvrant est répété aussi. Cet aspect du métalangage n'est pas parfaitement formalisé et demande une interprétation intuitive.
- En italique (ou pas) et en minuscule : le nom d'une valeur, d'une variable, d'une expression, d'une instruction ou d'une série d'instructions.
- 3 petits points tout seul sur une ligne : tout ce qui est possible entre la ligne qui précède les 3 petits points et la ligne qui les suit.

RETURN

L'instruction RETURN entre le BEGIN et le END permet de quitter la procédure.

4. Fonctions

Bloc function

On peut créer une procédure dans la zone DECLARE de la création d'un bloc.

La fonction peut être appelée dans la zone BEGIN – END par une simple référence à son nom, possiblement sans parenthèses s'il n'y a pas de paramètres.

Ici on calcule la moyenne des salaires :

```
DECLARE
   FUNCTION moySal RETURN number IS
      v_moy EMP.SAL%TYPE;
   BEGIN
      SELECT avg(sal) INTO v_moy from emp;
      RETURN v_moy;
   END;

BEGIN
   dbms_output.put_line('Moyenne des salaires : '||moySal);
END;
/

Moyenne des salaires : 2043,5
Procédure PL/SQL terminée avec succès.
```

Création d'une fonction

On peut créer une procédure qui est enregistrée dans la BD.

```
CREATE or REPLACE FUNCTION moySal RETURN number IS
   v_moy EMP.SAL%TYPE;
BEGIN
   SELECT avg(sal) INTO v_moy from emp;
   RETURN v_moy;
END moySal;
/

Fonction créée.
```

Appel d'une fonction

Avec la pseudo-table dual

```
SQL> select moysal from dual;
```

Dans un select

```
SELECT empno, ename, sal, moysal FROM emp
WHERE sal > moysal order by sal;

     EMPNO ENAME              SAL     MOYSAL
---------- ---------- ---------- ----------
      7782 CLARK            2450    2132,69
      7698 BLAKE            2850    2132,69
      7566 JONES            2975    2132,69
      7788 SCOTT            3000    2132,69
```

```
      7902 FORD              3000    2132,69
      7839 KING              5000    2132,69
6 ligne(s) sélectionnée(s).
```

Dans un bloc

```
begin
   dbms_output.put_line('Moyenne des salaires : '||moySal);
end;
/


Moyenne des salaire : 2043,5
Procédure PL/SQL terminée avec succès.
SQL>
```

Variable locale : IS ou AS : Zone des variables des procédures et des fonctions

Dans les procédures et les fonctions les variables locales sont situées entre IS ou AS et BEGIN. Il n'y a pas de DECLARE.

```
DECLARE PROCEDURE test
{IS | AS}
    [TYPE nomTypeLocal IS definitionDuType;]
    [...]
    [nomVarLocale TYPE;]
    [...]
BEGIN
```

Paramètres en entrée d'une fonction

Ici on calcule la moyenne des salaires pour un département donné :

```
CREATE or REPLACE
FUNCTION moySalDept(v_deptno EMP.DEPTNO%TYPE) RETURN number IS
   moySalDept EMP.SAL%TYPE;
BEGIN
   SELECT avg(sal) INTO moySalDept FROM emp
      WHERE deptno=v_deptno;
   RETURN moySalDept;
END moySalDept;
/
```

Appel d'une fonction

```
CREATE or REPLACE
PROCEDURE AffEmpDept2(v_deptno EMP.DEPTNO%TYPE) IS
   v_nbemp number;
   v_nbemptot number;
   v_moysal EMP.SAL%TYPE;
Begin
   v_moysal := moySalDept(v_deptno);

   SELECT count(*) INTO v_nbemp FROM emp
      WHERE deptno=v_deptno and sal > v_moysal;

   SELECT count(*)INTO v_nbemptot FROM emp
      WHERE deptno=v_deptno;

   dbms_output.put_line('Le département '||v_deptno||
            ' contient '||v_nbemp||' employe(s) sur '||v_nbemptot||
           ' à salaire > ' || v_moysal || ':' ) ;

   FOR tuple IN (
      select empno, ename, sal, deptno from emp
      where deptno = v_deptno and sal > v_moysal
    )
    LOOP
       dbms_output.put_line('.    - Employé n°' || tuple.EMPNO||
           ' : ' || tuple.ENAME||',      salaire = ' || tuple.SAL);
    END LOOP;
END;
/

Procédure créée.
```

```
CALL affempdept2(10);
```

```
Le département 10 contient 2 employe(s) sur 4 à salaire > 2448,75
.
.     Employé n°7839 : KING,    salaire = 5000
.     Employé n°7782 : CLARK,   salaire = 2695

Appel terminé.
SQL>
```

Structure d'une fonction

```
CREATE [OR REPLACE] FUNCTION nomFonction [(
    nomVar [{IN | OUT | IN OUT }] type
    [, ...]
)]
RETURN type
{IS | AS}
    [nomVarLocale TYPE;]
    [...]
BEGIN
    instructions;
[EXCEPTION
    instructions d'exception]
END [nomFonction];
```

IS ou AS sont équivalent

On reviendra sur les paramètres en sortie (OUT) et en entrée-sortie (IN OUT)

On reviendra sur les EXCEPTIONS.

5. Caractéristiques des paramètres de procédure et de fonction

Syntaxe

```
CREATE [OR REPLACE] {PROCEDURE | FUNCTION} nomProcOuFonc [(
    nomVar [ {IN | OUT [NOCOPY] | IN OUT [NOCOPY] } ]
    type [ { := | DEFAULT } valeurParDefaut ]
    [, ...]
)]
[ RETURN type]
{IS | AS}
```

IN : paramètre en entrée

Un argument IN est un argument dont la valeur d'entrée est utilisée par le bloc et n'est pas modifiée par le bloc (procédure ou fonction).

Il se comporte comme une constante dans le bloc.

OUT : paramètre en sortie

Un argument OUT est un argument dont la valeur d'entrée n'est pas utilisée par le bloc. L'argument est initialisé et modifié par le bloc (procédure ou fonction) pour produire une valeur en sortie du bloc.

IN OUT : paramètre en entrée-sortie

Ce mode est une combinaison du IN et du OUT.

Un argument IN OUT est un argument dont la valeur d'entrée est utilisée par le bloc (IN) et sera aussi modifiée par le bloc (OUT).

NOCOPY

De façon paradoxale, par défaut, les arguments IN sont passés par référence, les arguments OUT et IN OUT sont passés par valeur. Le but est de pouvoir annuler les modifications : en passant les paramètres OUT et INOUT par valeurs, on fera la modification réelle à la fin de la procédure et on peut ainsi annuler les modifications faites dans la procédure (ROLLBACK).

L'argument NOCOPY s'applique donc uniquement aux arguments OUT ou IN OUT pour qu'ils soient passés par référence.

L'avantage est que les traitements sont plus rapide.

Le défaut est que quand la modification est faite, on ne peut plus revenir en arrière.

Valeur par défaut

On peut fixer une valeur par défaut aux arguments.

Cette valeur sera utilisée si aucune autre n'est fournie à l'appel du bloc.

Passage des paramètres par position et par nom

Quand on utilise un bloc (procédure ou fonction), on peut passer les paramètres par position ou par nom.

Passage par position

Le premier argument d'appel prend la place du premier argument formel, etc.

Passage par nom

Les arguments d'appel sont reliés au nom de l'argument formel auxquels ils correspondent..
Ils peuvent alors est passés dans n'importe quel ordre.

```
BEGIN
    [instructions ]
    nomBloc( nomArgumentFormel => argumentDAppel [, ...] );
    [instructions ]
END;
```

Exemple : procedure equa1

Résolution d'une équation du premier degré

```
create or replace procedure equa1(a float, b float,  x out float,
    nbsol out integer) is
begin
    if a = 0 then
         if b = 0 then
           nbsol:=-1;
           x:=null;
         else
           nbsol:=0;
           x:=null;
         end if;
    else
         nbsol:=1;
         x:=-b/a;
    end if;
end ;
/

var x number;
var nbsol number;
-- equa1 : 2x-4=0
call equa1(2, -4, :x, :nbsol);
print :x;
print :nbsol;
```

6. Gestion des procédures et des fonctions

Lister les procédures et les fonctions

La vue USER_PROCEDURES permet de lister les procédures et les fonctions sans distinction entre les deux.

```
Select object_name from user_procedures ; // affiche les procedures
    et les functions.
```

La vue USER_SOURCE permet de lister les procédures et les fonctions en distinguant entre les deux :

```
Select distinct name, type from user_source;
```

Voir le code des procédures et les fonctions

La vue USER_SOURCE contient chaque ligne du code des procédures et des fonctions.

```
Set linesize 100
Col text format A80
Select line, text from user_source where name = 'NOMPROCouFONC';
```

Débogage

```
SQL> Show errors ;
```

« Show errors » utilise la vue USER_ERRORS.

```
SQL> select object_name, object_type, status from user_object
     where object_name='NOMPROCouFONC';
```

L'attribut STATUS de la vue USER_OBJECTS précise l'état des procédures et des fonctions.

Suppression d'une fonction

```
DROP FUNCTION nomFonction ;
```

Cette suppression met à jour les vues USER_SOURCE et USER_PROCEDURES.

Suppression d'une procédure

```
DROP PROCEDURE nomProcédure ;
```

Cette suppression met à jour les vues USER_SOURCE et USER_PROCEDURES.

7. Les structures de contrôle

Tests

Syntaxe

```
IF expression THEN
    instructions;
[ ELSIF expression THEN
    instructions; ...]
[ ELSE
    instructions;]
END IF;
```

- ***Rappel de la sémantique du métalangage***

Majuscule : les mots-clés

Entre crochets : ce qui est facultatif

Entre accolades : proposition de plusieurs choix possibles. Les choix sont séparés par des barres verticales.

3 petits points avant une accolade fermante : ce qui est entre accolades peut être répété.

En italitque : le nom d'une valeur, d'une variable, d'une expression, d'une instruction ou d'une série d'instructions.

Table de vérité du AND, OR et NOT en logique trivalente :

AND	**TRUE**	**FALSE**	**NULL**
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	**TRUE**	**FALSE**	**NULL**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	**TRUE**	**FALSE**	**NULL**
	FALSE	TRUE	NULL

Case simple: case sur valeur

Syntaxe

```
CASE expression
    WHEN valeur  THEN
         instructions
    [, … ]
```

```
    [ ELSE
        instructions ]
END CASE;
```

Case avec recherche : case sur condition

Syntaxe

```
CASE expression
    WHEN condition  THEN
        instructions
    [, … ]
    [ ELSE
        instructions ]
END CASE;
```

On sort du CASE dès qu'une condition est vérifiée.

Boucles sans fin : LOOP

Syntaxe

```
[<< nomBoucle >>]
LOOP
    instructions ;
END LOOP [ nomBoucle ] ;
```

On sort de la boucle sans fin avec un EXIT :

```
EXIT  [ nomBoucle ]   [ WHEN  condition ] ;
```

L'EXIT seul permet de sortir sans condition. En général un EXIT seul est dans un test.

L'EXIT WHEN inclut la condition de sortie. En général, il n'est pas dans un test.

Boucles WHILE : tant que

Syntaxe

```
[<< nomBoucle >>]
WHILE condition LOOP
    instructions ;
END LOOP [ nomBoucle ] ;
```

Boucle FOR : compteur

Syntaxe

```
[<< nomBoucle >>]
FOR variable IN [REVERSE] expression1 .. expression2 LOOP
    instructions ;
END LOOP [ nomBoucle ] ;
```

Boucle FORALL : compteur avec regroupement des instructions LMD

Syntaxe

```
[<< nomBoucle >>]
FORALL variable IN expression1 .. expression2 LOOP
    [instructions ;]
    instruction LMD ;
```

```
    [instructions ;]
  END LOOP [ nomBoucle ] ;
```

La boucle FORALL remplace le FOR et permet de regrouper les instructions LMD de la boucle en une seule série. Ca optimise le traitement.

Exit : sortie de boucle

Présentation

Exit permet de quitter la boucle.

Syntaxe

```
  EXIT [nomBoucle] [WHEN condition] ;
```

8. Types composés

Classification des types

Un type peut être simple ou composé :

- Simple C'est un des types de colonne des tables.
- Composé : comprend plus d'un élément.

Un type composé peut être un enregistrement ou un tableau

- Les enregistrements : RECORD et %ROWTYPE
- Les tableaux : TYPE, ARRAY et TABLE

Le type RECORD

Déclaration d'un type RECORD

```
TYPE nomType IS RECORD (
    NomChamp1 TYPE [NOT NULL] [:= expression]
    [,...]
);
```

Déclaration d'une variable de type RECORD

```
TYPE nomType IS RECORD (
    NomChamp1 TYPE [NOT NULL] [:= expression]
    [,...]
);
```

Usage d'une variable de type RECORD

Les variables de type RECORD peuvent correspondre à un tuple.

On fait un SELECT .. INTO dans le RECORD

```
undef empno; -- empno à saisir : 7839 par exemple

DECLARE
    TYPE type_employe IS RECORD (
         empno EMP.EMPNO%TYPE,
         ename EMP.ENAME%TYPE,
         sal EMP.SAL%TYPE
    );
    v_emp type_employe;

BEGIN
    SELECT empno, ename, sal INTO v_emp
         FROM emp
         WHERE empno=&empno;

    dbms_output.put_line
        (v_emp.ename||', employe n°' || v_emp.empno);
    dbms_output.put_line ('salaire mensuel :' || v_emp.sal) ;
    dbms_output.put_line ('salaire annuel :'  || 12*v_emp.sal) ;

END ;
/
```

%ROWTYPE : déclarer une variable dont le type correspond à la définition d'une table

Présentation

Partout où on utilise un nom de type (nomType), on peut utiliser un type RECORD provenant d'une table ou d'une variable.

Exemple

On reprend l'exemple précédent, mais avec un ROWTYPE.

```
undef empno; -- empno à saisir : 7839 par exemple

DECLARE
    v_emp EMP%ROWTYPE;
BEGIN
    SELECT * INTO v_emp
         FROM emp
         WHERE empno=&empno;

    dbms_output.put_line
        (v_emp.ename||', employe n°' || v_emp.empno);
    dbms_output.put_line ('salaire mensuel :' || v_emp.sal) ;
    dbms_output.put_line ('salaire annuel :'  || 12*v_emp.sal) ;

END ;
/
```

Remarque

Le ROWTYPE oblige à faire un SELECT * INTO : le * est nécessaire pour qu'on ait le même nombre d'attributs.

Subtilité syntaxique

Il ne faut pas créer des types synonymes ou des sous-types à partir de type ROWTYPE.

Le tableau de taille fixe : ARRAY

Déclaration d'un type tableau de taille fixe: ARRAY

```
TYPE nomType IS ARRAY(taille) OF nomTypeElt [NOT NULL];
```

Déclaration d'une variable de type tableau de taille fixe : ARRAY

```
TYPE nomType IS ARRAY(taille) OF nomTypeElt [NOT NULL];
v_tab nomType ;
```

Usage d'une variable de type ARRAY

```
DECLARE
    TYPE type_tabNumber IS ARRAY(5) OF NUMBER;
    v_tab type_tabNumber;
    i number;
    n number;
BEGIN
    v_tab:=type_tabNumber(1, 2, 3, 4, 5);
    FOR i IN 1 .. 5 LOOP
         dbms_output.put_line('v_tab['||I||']='||v_tab(i));
         v_tab(i):=v_tab(i)+10;
    END LOOP;

    FOR i IN 1 .. 5 LOOP
         dbms_output.put_line('v_tab['||I||']='||v_tab(i));
    END LOOP;
END;
/
```

Le tableau de taille variable : TABLE

Déclaration d'un type tableau de taille variable : TABLE

```
TYPE nomType IS TABLE OF nomTypeElt [NOT NULL]
INDEX BY {PLS_INTEGER | BINARY_INTEGER | VARCHAR2(taille)};
```

L'INDEX c'est l'indice d'accès aux éléments du table

PLS_INTEGER et BINARY_INTEGER sont équivalents. Ce sont des types entiers. Les réels sont arrondis pour donner un entier. L'entier correspond à l'indice d'accès aux éléments du tableau.

VARCHAR2(taille) permet de donner une chaîne de caractère comme indice d'accès à un élément du tableau.

Déclaration d'une variable de type tableau de taille variable : TABLE

```
TYPE type_tabEmpnos IS TABLE of emp.empno%type;
v_tabEmpno type_tabEmpnos
```

Usage – 1 - d'une variable de type TABLE

```
DECLARE
    TYPE type_tabEmpnos IS TABLE of emp.empno%type
        INDEX BY PLS_INTEGER;
    v_tabEmpnos type_tabEmpnos;

BEGIN
    SELECT empno BULK COLLECT INTO v_tabEmpnos
       FROM emp
       WHERE ROWNUM<5;
    v_tabEmpnos(-5):=-5;
    v_tabEmpnos(10000):=1000;

    dbms_output.put_line(v_tabEmpnos(1));
    dbms_output.put_line(v_tabEmpnos(2));
    dbms_output.put_line(v_tabEmpnos(3));
    dbms_output.put_line(v_tabEmpnos(4));

    dbms_output.put_line(v_tabEmpnos(-5));
    dbms_output.put_line(v_tabEmpnos(10000));

END;
/
```

➢ ***A noter***

- v_tabEmpnos(-5):=10; Les indices d'un tableau de taille variable ne sont pas nécessairement contigus. C'est le principe du tableau associatif en PHP.
- Select BULK COLLECT INTO : équivalent à Select INTO pour une variable de type simple.

  Le BULK COLLECT INTO ne fonctionne qu'avec des index de type PLS_INTEGER ou BINARY_INTEGER.

Usage – 2 - d'une variable de type TABLE : parcours d'un SELECT

Récupération d'un Select

```
DECLARE
   TYPE type_tabEmployes IS TABLE of EMP%ROWTYPE
      INDEX BY PLS_INTEGER;
   v_tabEmp type_tabEmployes;
BEGIN
    SELECT * BULK COLLECT INTO v_tabEmp FROM emp;
    FOR i IN 1 ..  v_tabEmp.COUNT LOOP
         dbms_output.put_line(v_tabEmp(i).ename ||
         ', employe n°' ||v_tabEmp(i).empno ||
         ' : salaire mensuel =     '||v_tabEmp(i).sal) ;
    END LOOP;
END;
/
```

A noter qu'on n'est pas obligé de déclarer le « i » du for.

Usage – 3 - d'une variable de type TABLE : index de type char

```
DECLARE
    TYPE type_tabSalMoy IS TABLE of FLOAT INDEX BY varchar2(10);
    v_tabMoy type_tabSalMoy;
BEGIN
    select avg(sal) INTO v_tabMoy('CLERK') from emp where job =
    'CLERK';
    dbms_output.put_line('Moyenne des salaries des CLERK : '
    ||v_tabMoy('CLERK') ) ;
END;
/
```

➢ ***A noter***

- v_tabMoy('CLERK') : on retrouve le principe du tableau associatif du PHP.
- INDEX BY varchar2(10) : 10 fixe la limite du nom donné à l'indice du tableau.
- v_tabMoy('CLERK'). L'indice du tableau est une chaîne.

Méthodes associées aux tableaux de taille variable

Présentation

EXISTS(n) :	Revoie vrai si l'élément d'indice n existe.
COUNT :	Renvoie le nombre d'éléments du tableau.
FIRST / LAST	Renvoie le premier ou le dernier élément du tableau.
PRIOR(n) / NEXT(n)	Retourne l'élément précédent ou suivant l'élément d'indice n.
DELETE(n)	Supprime l'élément d'indice n.
DELETE	Supprime l'élément de l'indice en cours.
DELETE(a,b)	Supprime les éléments d'indice a et b

Principe d'utilisation

Ce sont des méthodes au sens de la programmation objet.

Exemple

```
DECLARE
```

```
    TYPE type_tabEntier IS TABLE of NUMBER(5,0)
    INDEX BY BINARY_INTEGER;
    v_tabEntier type_tabEntier;
BEGIN
    for i in 1..10 loop v_tabEntier(i):=i*100; end loop;

    if (v_tabEntier.exists(5)) then
         dbms_output.put_line('t(5) existe');
    else
         dbms_output.put_line('t(5) n''existe pas');
    end if;
    if (v_tabEntier.exists(11)) then
         dbms_output.put_line('t(11) existe');
    else
         dbms_output.put_line('t(11) n''existe pas');
    end if;
    dbms_output.put_line('first : '||v_tabEntier.first);
    dbms_output.put_line('last : '||v_tabEntier.last);
    dbms_output.put_line('count : '||v_tabEntier.count);
    v_tabEntier(20):=2000;
    dbms_output.put_line('count : '||v_tabEntier.count);
    dbms_output.put_line('last 20: '||v_tabEntier.last);
    dbms_output.put_line('prior(5) : '||v_tabEntier.prior(5));
    dbms_output.put_line('prior(20) : '||v_tabEntier.prior(20));
    dbms_output.put_line('next(5) : '||v_tabEntier.next(5));
    dbms_output.put_line('next(10) : '||v_tabEntier.next(10));
    dbms_output.put_line('next(20) : '||v_tabEntier.next(20));
    v_tabEntier.delete(3);
    dbms_output.put_line('count : '||v_tabEntier.count);
    dbms_output.put_line('next(2) : '||v_tabEntier.next(2));
    v_tabEntier.delete;
    dbms_output.put_line('count : '||v_tabEntier.count);
END;
/
t(5) existe
t(11) n'existe pas
first : 1
last : 10
count : 10
count : 11
last 20: 20
prior(5) : 4
prior(20) : 10
next(5) : 6
next(10) : 20
next(20) :
count : 10
next(2) : 4
count : 0
Procédure PL/SQL terminée avec succès.
```

9. SQL et PL-SQL

SELECT INTO : récupération d'une valeur ou d'un tuple de la BD

```
DECLARE
    v_nbEmp integer;
    v_emp emp%rowtype;
BEGIN
    SELECT count(*) into v_nbEmp from emp;
    dbms_output.put_line('nb_emp : '||v_nbEmp);

    SELECT * into v_emp from emp where empno=&numEmp;
    dbms_output.put_line('Employé n° '||v_emp.empno);

    dbms_output.put_line('Nom : '||v_emp.ename||' - salaire :
    '||v_emp.sal);
END;
/
```

Remarque: une requête qui ne renvoie aucun tuple génère une erreur.

SELECT BULK COLLECT INTO : récupération d'une liste de tuples de la BD

```
DECLARE
    TYPE type_tabEmp IS TABLE of emp%rowtype INDEX BY PLS_INTEGER;
    v_tabEmp type_tabEmp;
BEGIN
    select * BULK COLLECT INTO v_tabEmp from emp;
    for i in 1..v_tabEmp.count loop
         dbms_output.put_line('Employé n° '||v_tabEmp(i).empno||
         ', Nom : '||v_tabEmp(i).ename||
         ' - salaire : '||v_tabEmp(i).sal);
    end loop;
END;
/
```

INSERT INTO

C'est un INSERT INTO classique.

On peut faire : « VALUES(v_emp) » avec v_emp de type %ROWTYPE.

UPDATE

C'est un UPDATE classique.

On peut faire « SET ROW = v_emp » avec v_emp de type %ROWTYPE.

DELETE

C'est un DELETE classique.

RETOUR du DML

SQL%FOUND

Vrai (true) si un tuple a été créé ou modifié ou supprimé.

```
BEGIN
    Update emp
         Set sal=sal*1.1
    Where empno = &numEmp;
    If sql%found then
         dbms_output.put_line('Un employé a été augmenté');
    else
         dbms_output.put_line('Pas d''employé augmenté');
    end if;
END;
/
```

SQL%NOTFOUND

Vrai (true) si aucun tuple n'a été créé ou modifié ou supprimé.

C'est le contraire de SQL%FOUND.

SQL%ROWCOUNT

Renvoie le nombre de tuples qui ont été créés ou modifies ou supprimés.

UPDATE... RETURNING et RETURNING BULK COLLECT

Présentation

RETURNING permet de renvoyer le ou les tuples qui ont été modifiés dans une variable.

Exemple de RETURNING

```
DECLARE
    V_empno emp.empno%TYPE;
BEGIN
    Update emp
         Set sal=sal*1.1
    Where empno = &numEmp
    Returning empno into v_empno;
    If sql%found then
         dbms_output.put_line('L''employé n° '||v_empno||' a été
    augmenté');
    else
         dbms_output.put_line('Pas d''employé augmenté');
    end if;
END;
/
```

La clause RETURNING se place à la fin de l'instruction DML.

On peut aussi retourner le ROWID.

Syntaxe

```
RETURNING expression1 [,...] [BULK COLLECT] INTO variable1 [,...];
```

Exemple de RETURNING BULK COLLECT

```
DECLARE
    TYPE type_tabEmp IS TABLE of emp%rowtype INDEX BY
    BINARY_INTEGER;
    v_tabEmp type_tabEmp;
BEGIN
    Update emp
         Set sal=sal*1.1
    Where job = '&nomJob'
--  Returning *   NE MARCHE PAS !! Dommage !!
    Returning empno, ename, job, mgr, hiredate, sal, comm, deptno
    BULK COLLECT INTO v_tabEmp;
    If sql%found then
         dbms_output.put_line('Les employés suivants ont été
    augmentés : ');
         for i in 1..v_tabEmp.count loop
           dbms_output.put_line('Employé n° '||v_tabEmp(i).empno||
           ', Nom : '||v_tabEmp(i).ename||
           ' - salaire : '||v_tabEmp(i).sal);
         end loop;
    else
         dbms_output.put_line('Pas d''employé augmenté');
    end if;
END;
/
```

EXECUTE IMMEDIATE : SQL dynamique

Présentation

Le PL-SQL ne permet pas d'utiliser directement les commandes du DDL (CREATE TABLE, DROP TABLE, etc.) ni les commandes du DCL (GRANT, REVOKE, etc.).

La clause EXECUTE IMMEDIATE permet d'utiliser toutes ces commandes.

La clause EXECUTE IMMEDIATE remplace avantageusement le package DBMS_SQL (cf. chapitre sur les packages standards).

Exemple

```
ACCEPT nomAtt CHAR prompt 'Entrez l''attribut que vous voulez
    catégoriser : ';

DECLARE
    v_sqlDynamique varchar2(200);
    v_nomAtt varchar2(20);
    v_nomTable varchar2(20);
BEGIN
    v_nomAtt := '&nomAtt';
    v_nomTable :='cat_'||v_nomAtt ;
    dbms_output.put_line('nomAtt : '||v_nomAtt||' - nomTable :
    '||v_nomTable);

    v_sqlDynamique:= 'DROP TABLE '||v_nomTable;
    dbms_output.put_line('Instruction : '||v_sqlDynamique);
    EXECUTE IMMEDIATE v_sqlDynamique ;


    v_sqlDynamique:='CREATE TABLE '||v_nomTable||' as select ' ||
        v_nomAtt||', count(*) nb from emp group by '||v_nomAtt ;
    dbms_output.put_line('Instruction : '||v_sqlDynamique);
    EXECUTE IMMEDIATE v_sqlDynamique ;
END;
/
```

ATTENTION : la chaîne-instruction de la variable v_sqlDynamique ne doit pas se terminer par un « ; ».

Syntaxe

```
EXECUTE IMMEDIATE varInstruction
[ { INTO {nomVar1 [, ...]} | BULK COLLECT INTO varTable ]
[ RETURNING expression1 [,...] [BULK COLLECT] INTO nomVar1[,...] ]
[ USING [IN|OUT|IN OUT] nomArgument [, ...] ]
;
```

➢ ***Explications***

- INTO et BULK COLLECT INTO permettent la récupération d'un ou de plusieurs tuples résulats d'un SELECT.
- RETURNING permet de renvoyer le ou les tuples qui ont été modifiés dans une variable.
- USING permet de mettre des arguments dans l'instruction de l'EXECUTE IMMEDIATE.

  Dans la chaîne « varInstruction », les arguments sont précédés par « : ». Attention, ces arguments ne peuvent être passés que dans un SELECT ou une instruction du DML. On ne peut pas les utiliser pour une instruction du DDL ou du DCL.

Remarque

Quand on fait du DDL avec un EXECUTE IMMEDIATE, si on veut faire du DML sur les tables du DDL, il faut aussi utiliser un EXECUTE IMMEDIATE car le DML sans EXECUTE IMMEDIATE fait référence à l'état de la BD au moment de l'exécution du bloc.

10. Les curseurs

Présentation

Un curseur est une variable qui permet de parcourir les lignes d'un SELECT une par une.

Sa syntaxe est un peu compliquée.

Il existe un usage de base à syntaxe simplifiée pour parcourir un SELECT : le FOR IN, sans fetch et sans curseur.

L'usage de base du parcours d'un SELECT : FOR tuple IN SELECT (…) LOOP

Le FOR IN **sans fetch et sans curseur** permet de parcourir les lignes d'une table relationnelle de façon simple.

C'est l'usage à privilégier.

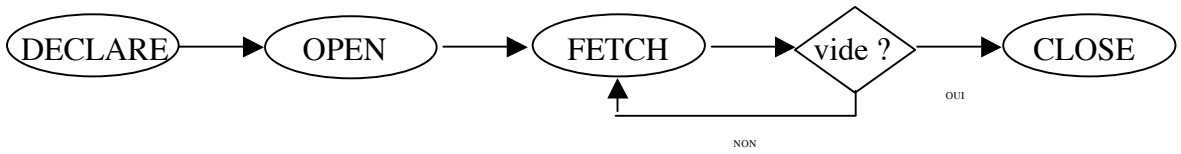
Version sans paramètre

```
DECLARE
    i number;
BEGIN
    i:=0;
    FOR tuple IN (
         SELECT empno, ename, job, sal+nvl(comm, 0) as saltot
         FROM emp order by saltot desc
    )
    LOOP
        dbms_output.put_line(tuple.empno||'-'|| tuple.ename
        ||'-'|| tuple.job||'-'|| tuple.saltot);
        i:=i+1;
    end loop;
    dbms_output.put_line('Nb lignes traitées : '|| i);
END;
/
```

Version avec paramètre

```
DECLARE
    i number;
BEGIN
    i:=0;
    FOR tuple IN (
         SELECT empno, ename, job, sal+nvl(comm, 0) as saltot
         FROM emp
         WHERE deptno='&v_deptno' order by saltot desc
    )
    LOOP
        dbms_output.put_line(tuple.empno||'-'|| tuple.ename
        ||'-'|| tuple.job||'-'|| tuple.saltot);
        i:=i+1;
    end loop;
    dbms_output.put_line('Nb lignes traitées : '|| i);
END;
/
```

Organigramme d'un curseur (diagramme d'activités)



L'algorithmique des curseurs ressemble à celle des fichiers :

1. Déclaration de la variable de type CURSOR
2. Ouverture du curseur : OPEN
3. Boucle pour lire une ligne et passer à la suivante : FETCH
4. Condition de sortie de la boucle : on est arrivé à la fin du tableau.
5. Fermeture du curseur.

Déclaration d'un curseur

Exemple

On déclare une variable de type CURSOR qui est un SELECT

Et une variable correspondant au ROWTYPE du curseur.

```
DECLARE
    CURSOR c_emp IS
       SELECT empno, ename, job, sal+nvl(comm, 0) as saltot
       FROM emp order by saltot desc;
    v_emp c_emp%ROWTYPE;
BEGIN
    ...
END;
```

Syntaxe

```
DECLARE
    CURSOR c_nomCurseur IS requiteSQL;
    v_nomVar c_nomCurseur%ROWTYPE;
BEGIN
    ...
END;
```

Open curseur

Pour commencer à travailler sur un curseur, on doit l'ouvrir :

```
    OPEN c_emp;
```

Close curseur

Quand on a fini d'utiliser un curseur, il vaut mieux le fermer : ça libère l'accès aux données pour d'autres utilisateurs.

Par défaut, PL-SQL ferme tous les curseurs ouverts à la fin de la procédure.

```
    CLOSE c_emp;
```

Deux types de FETCH

FETCH INTO : une seule ligne

Le FETCH INTO permet de lire une ligne dans une variable de type RECORD.

On peut utiliser un ROWTYPE correspondant au type du curseur.

On sort de la boucle du FETCH quand « c_emp%notfound »

```
DECLARE
    cursor c_emp is select empno, ename, job, sal+nvl(comm, 0) as
    saltot from emp order by saltot desc;
    v_emp c_emp%rowtype;
BEGIN
    OPEN c_emp;
    if c_emp%isopen then
         dbms_output.put_line('Le curseur c_emp est ouvert');
    end if ;
    LOOP
           FETCH c_emp INTO v_emp;
           exit when c_emp%NOTFOUND;
           dbms_output.put_line(v_emp.empno||'-'||v_emp.ename
           ||'-'||v_emp.job||'-'||v_emp.saltot);
    END LOOP;
    dbms_output.put_line('Nb lignes traitées : '||c_emp%rowcount);
    CLOSE c_emp;
END;
/
```

Etats d'un curseur

%FOUND	VRAI si le fetch renvoie une ligne ou un tableau.
%NOTFOUND	VRAI si le fetch n'a rien renvoyé.
%ISOPEN	VRAI si le curseur est ouvert.
%ROWOUNT	Nombre de lignes renvoyées par le fetch depuis l'ouverture du curseur.

FETCH BULK COLLECT INTO : toute la table

Le FETCH BULK COLLECT INTO permet de lire toute la table dans une variable de type tableau : TABLE.

On boucle de 1 à v_tabemp.COUNT.

```
DECLARE
    cursor c_emp is select empno, ename, job, sal+nvl(comm, 0) as
    saltot from emp order by saltot desc;
    TYPE type_tabEmp IS TABLE of c_emp%rowtype;
    v_tabEmp type_tabEmp ;
BEGIN
    OPEN c_emp;
    FETCH c_emp BULK COLLECT INTO v_tabEmp;
    for I in  1..v_tabemp.COUNT loop
           dbms_output.put_line(v_tabemp(i).empno||
           '-' || v_tabemp(i).ename||'-'||v_tabemp(i).job||
           '-'||v_tabemp(i).saltot);
    END LOOP;
    dbms_output.put_line('Nb lignes traitées : '||c_emp%rowcount);
    CLOSE c_emp;
END;
/
```

Curseur paramétré

Curseur avec des paramètres d'appel

```
DECLARE
    cursor c_emp (v_deptno emp.deptno%type)
    is
         select empno, ename, job, sal+nvl(comm, 0) as saltot
         from emp where deptno=v_deptno
         order by saltot desc;
    v_emp c_emp%rowtype;

BEGIN
    open c_emp(&v_deptno);
    if c_emp%isopen then
         dbms_output.put_line('Le curseur c_emp est ouvert');
    end if ;
    loop
           fetch c_emp into v_emp;
           exit when c_emp%notfound ;
           dbms_output.put_line(v_emp.empno||'-'||v_emp.ename
           ||'-'||v_emp.job||'-'||v_emp.saltot);
    end loop;
    dbms_output.put_line('Nb lignes traitées : '||c_emp%rowcount);
    close c_emp;
END;
/
```

Curseur avec des paramètres dans le select

```
DECLARE
    v_deptno emp.deptno%type;      -- utilise dans le curseur.
                                   -- déclaré avant le curseur.
    cursor c_emp
    is
         select empno, ename, job, sal+nvl(comm, 0) as saltot
         from emp where deptno=v_deptno
         order by saltot desc;
    v_emp c_emp%rowtype;

BEGIN
--  v_deptno := 20;
    v_deptno := &v_numDept;
    open c_emp;
    if c_emp%isopen then
         dbms_output.put_line('Le curseur c_emp est ouvert');
    end if ;
    loop
           fetch c_emp into v_emp;
           exit when c_emp%notfound ;
           dbms_output.put_line(v_emp.empno||'-'||v_emp.ename
           ||'-'||v_emp.job||'-'||v_emp.saltot);
    end loop;
    dbms_output.put_line('Nb lignes traitées : '||c_emp%rowcount);
    close c_emp;
END;
/
```

Curseur implicite : SQL

Quand on fait un UPDATE ou un DELETE, ça donne lieu à un curseur implicite.

Le nom du curseur implicite est : SQL.

On peut accéder à l'état du curseur implicite SQL.

```
DECLARE
BEGIN
    if sql%isopen = FALSE then
         dbms_output.put_line('Avant LMD : isopen vaut FALSE');
    end if;
    if sql%found is NULL then
         dbms_output.put_line('Avant LMD : found vaut NULL');
    end if ;
    if sql%rowcount is NULL then
         dbms_output.put_line('Avant LMD : rowcount vaut NULL');
    end if ;

    update emp set sal=1.1*sal where deptno=30;

    if sql%found = TRUE then
           dbms_output.put_line('Après LMD : found vaut TRUE  '
           ||sql%rowcount||' enregistrements modifiés');
    elsif sql%found = FALSE then
           dbms_output.put_line('Après LMD : found vaut FALSE  '
           ||sql%rowcount||' enregistrements modifiés');
    elsif sql%found is NULL then
           dbms_output.put_line('Après LMD : found vaut NULL  '
           ||sql%rowcount||' enregistrements modifiés');
    end if;
    if sql%isopen = FALSE then
         dbms_output.put_line('Après LMD : isopen vaut FALSE');
    end if ;

END;
/
```

A noter l'état du curseur implicite SQL:

ISOPEN est toujours à faux.

FOUND et REOWCOUNT vallent NULL avant le traitement.

FOUND vaut TRUE si un tuple au moins a été traité, FALSE sinon.

ROWCOUNT renvoie le nombre de tuples traités, 0 si aucun.

CURSEUR avec verrouillage : FOR UPDATE et WHERE CURRENT OF

On peut utiliser les tuples du select d'un curseur pour faire un update.

Syntaxe du FOR UPDATE

```
CURSOR nomCurseur
    [(nom_arg TYPE [:= valDef]  [, ...] ]
IS requete
FOR UPDATE
    [OF nomAttribut [, ...] ]
    [{NOWAIT |  WAIT nbSecondes}] ;
```

La clause FOR UPDATE peut porter sur tous les attributs du curseur (situation par défaut) ou seulement sur ceux qui sont précisés : OF.

La clause NOWAIT n'attend pas pour verrouiller les attributs. S'ils sont déjà verrouillés, la déclaration de curseur renverra une erreur.

La clause WAIT nbSecondes accorde nbSeconde pour réessayer de verrouiller les attributs avant de déclencher une erreur.

Where current of

Avec un curseur for update, on peut utiliser un « where current of » dans les update et les delete pour travailler sur le tuple courant du curseur.

```
DECLARE
    CURSOR c_nomCurseur …
    ...
    v_numCurseur c_nomCurseur;
BEGIN
    ...
    for v_nomCurseur in c_nomCurseur loop
         ...
         update ...
         set
         where CURRENT OF c_nomCurseur;
         ...
    end loop;
...
END;
```

Commit et Rollback

Pour libérer les attributs verrouillés il faut utiliser un COMMIT ou un ROLLBACK.

Après un COMMIT ou un ROLLBACK, on ne peut plus faire de fetch.

Variable curseur : REF CURSOR et OPEN … FOR

REF CURSOR avec RETURN et OPEN…FOR : déclaration et ouverture

```
DECLARE
    TYPE t_curseur_emp IS REF CURSOR RETURN emp%ROWTYPE;
    c_emp t_curseur_emp;
BEGIN
    OPEN c_emp FOR
         SELECT * FROM emp WHERE deptno=30;
    ...
END;
```

➢ ***Explications***

- On déclare d'abord un TYPE REF CURSOR en précisant le type de tuple retourné dans le RETURN.
- Ensuite on déclare une variable du type REF CURSOR prédéfini.
- Enfin, à l'ouverture, OPEN, on ajoute un FOR suivi d'une requête.
- Ainsi, on pourra utiliser le même curseur pour des requêtes différentes, à condition seulement que les attributs projetés soient les mêmes. Il faudra fermer le curseur (CLOSE), pour ensuite pouvoir le rouvrir : OPEN … FOR.

REF CURSOR sans RETURN

```
DECLARE
    TYPE t_curseur_emp IS REF CURSOR;
    c_emp t_curseur_emp;
BEGIN
    OPEN c_emp FOR
         SELECT emmno, ename, deptno FROM emp WHERE deptno=30;
    ...
END;
```

➢ ***Explications***

- On déclare d'abord un TYPE REF CURSOR sans préciser le type de tuple retourné.
- Ensuite on déclare une variable du type REF CURSOR prédéfini.
- Enfin, à l'ouverture, OPEN, on ajoute un FOR suivi d'une requête.
- Ainsi, on pourra utiliser le même curseur pour des requêtes différentes quels que soient les attributs projetés. Il faudra fermer le curseur (CLOSE), pour ensuite pouvoir le rouvrir : OPEN … FOR.

REF CURSOR avec SELECT paramétré

```
DECLARE
    TYPE t_curseur_emp IS REF CURSOR;
    c_emp t_curseur_emp;
    v_SQL varchar2(250)
         :='select * from emp where deptno=:vp_deptno';
    v_deptno emp.deptno%type;
    v_emp emp%rowtype;
BEGIN
    v_deptno := 20;
    OPEN c_emp FOR v_SQL USING v_deptno;
    loop
           fetch c_emp into v_emp;
```

```
            exit when c_emp%notfound ;
            dbms_output.put_line(v_emp.empno||'-'||v_emp.ename
            ||'-'||v_emp.job||'-'||v_emp.deptno);
    end loop;
    dbms_output.put_line('Nb lignes traitées : '||c_emp%rowcount);
    close c_emp;
END;
/
```

➢ ***Explications***

- On déclare d'abord un TYPE REF CURSOR sans préciser le type de tuple retourné.
- Ensuite on déclare une variable du type REF CURSOR prédéfini.
- On déclare une chaîne de caractères qui contient un SELECT avec un paramètre « :nomParam ».
- On ouvre le curseur, OPEN, avec un FOR de la chaîne de caractères et un USING proposant une valeur pour le paramètre de la chaîne.

Conclusion

L'intérêt des variables curseur c'est qu'elles pourront être passées en paramètre à des procédures ou des fonctions.

11. Les exceptions

Présentation

Quand une erreur système ou applicative se produit, une exception est générée.

Les traitements en cours dans la section d'exécution (section du BEGIN) sont stoppés.

Les traitements de la section d'exception (section du EXCEPTION) commence.

Syntaxe

```
DECLARE
    déclaration de variables
BEGIN
    instructions
EXCEPTION
    WHEN nom d'exceptions THEN
        instructions
END;
```

Les 4 types d'exception

Type d'exception	Nommée	Anonyme (numérotée)
Système	1 : Exception système nommée	2 : Exception système anonyme
Utilisateur	3 : Exception utilisateur nommée	4 : Exception utilisateur anonyme

SQLCODE et SQLERRM

SQLCODE : pour afficher le numéro de l'erreur.

SQLERRM : pour afficher le numéro et le message d'erreur.

```
dbms_output.put_line('SQLCODE = '||SQLCODE);
dbms_output.put_line('SQLERRM = '||SQLERRM);
```

Exceptions système nommées et exceptions système anonymes (numérotées)

Les exceptions prédéfinies par ORACLE ne sont pas toutes nommées (il y en a trop). Par contre, elles ont toutes un numéro.

Les exceptions prédéfinies non nommées sont dites anonymes.

Exceptions prédéfinies nommées

DUP_VAL_ON_INDEX : violation de l'unicité lors d'un insert ou d'un update (-1).

NO_DATA_FOUND : un select into ne ramène aucune ligne (-1403 ; 100). A noter qu'avec une fonction du groupe, le select ramène toujours au moins une valeur, même si il n'y a pas de ligne (dans ce cas le retour vaut NULL).

TOO_MANY_ROWS : un select into ramène plusieurs lignes (-1422).

VALUE_ERROR : erreur de onversion de type (-6502).

ZERO_DIVIDE : division par 0 (-1476).

Etc.

Exceptions prédéfinies anonymes

Dans le code suivant, on affiche le message correspondant aux exceptions anonymes numérotés de –2299 à –2290.

```
BEGIN
    For i in –2299 .. –2290 loop
         dbms_output.put_line(sqlerrm(i));
    end loop ;
END ;
/
ORA-02299: impossible de valider (.) - clés en double trouvées
ORA-02298: impossible de valider (.) - clés parents introuvables
ORA-02297: impossible de désactiver la contrainte (.) - des
    dépendances existent
ORA-02296: impossible d'activer la contrainte (.) - valeurs null
    trouvées
ORA-02295: plusieurs clauses ENABLE/DISABLE trouvées pour la
    contrainte
ORA-02294: impossible d'activer (.) - contrainte modifiée pendant
    la validation
ORA-02293: impossible de valider (.) - violation d'une contrainte
    de contrôle
ORA-02292: violation de contrainte (.) d'intégrité - enregistrement
    fils existant
ORA-02291: violation de contrainte d'intégrité (.) - clé parent
    introuvable
ORA-02290: violation de contraintes (.) de vérification

Procédure PL/SQL terminée avec succès.
```

WHEN exceptionPrédéfinie : exception système nommée (1)

Principe

En utilisant les exceptions prédéfinies par le système, on va pouvoir préciser les messages d'erreur.

Exemple

```
DECLARE
    v_emp emp%rowtype;
    v_empno emp.empno%type;
BEGIN
    v_empno:=&numEmp;
    SELECT * into v_emp from emp where empno=v_empno;
    dbms_output.put_line('Employé n° '||v_emp.empno);
    dbms_output.put_line('Nom : '||v_emp.ename||' - salaire :
    '||v_emp.sal);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
         dbms_output.put_line('L''employé n° '||v_empno||' n''existe
    pas');
         dbms_output.put_line('SQLCODE = '||SQLCODE);
         dbms_output.put_line('SQLERRM = '||SQLERRM);
END;
/
Entrez une valeur pour numemp : 1000
ancien   5 :    v_empno:=&numEmp;
nouveau   5 :   v_empno:=1000;
L'employé n° 1000 n'existe pas
SQLCODE = 100
SQLERRM = ORA-01403: aucune donnée trouvée
```

```
Procédure PL/SQL terminée avec succès.
```

WHEN OTHERS : exception système anonyme (2)

Principe

En déclarant une zone d'exception avec le cas « OTHERS », on prend en compte toutes les erreurs système.

Exemple

```
DECLARE
    v_emp emp%rowtype;
    v_empno emp.empno%type;
BEGIN
    v_empno:=&numEmp;
    SELECT * into v_emp from emp where empno=v_empno;
    dbms_output.put_line('Employé n° '||v_emp.empno);
    dbms_output.put_line('Nom : '||v_emp.ename||' - salaire :
    '||v_emp.sal);
EXCEPTION
    WHEN OTHERS THEN
         dbms_output.put_line('SQLCODE = '||SQLCODE);
         dbms_output.put_line('SQLERRM = '||SQLERRM);
END;
/
Entrez une valeur pour numemp : 2000
ancien   5 :    v_empno:=&numEmp;
nouveau   5 :   v_empno:=2000;
SQLCODE = 100
SQLERRM = ORA-01403: aucune donnée trouvée

Procédure PL/SQL terminée avec succès.
```

➢ ***Précisions***

La fonction SQLCODE renvoie le code de l'erreur.

La fonction SQLERRM renvoie le message de l'erreur.

EXCEPTION_INIT : exception système anonyme nommée par l'utilisateur (2 bis)

On peut associer un nom à une exception prédéfinie anonyme.

L'opération se fait en trois temps : 2 déclarations dans le bloc DECLARE et un WHEN dans le bloc EXCEPTION.

```
DECLARE
    NOM_EXCEPTION EXCEPTION;
    PRAGMA EXCEPTION_INIT(NOM_EXCEPTION, NUMERO_EXCEPTION);
    ...
BEGIN
    ...
EXCEPTION
    WHEN NOM_EXCEPTION THEN
         Instructions ;
    ...
END;
```

NOM_EXCEPTION est un nom choisi par le programmeur.

NUMERO_EXCEPTION est le numéro d'erreur prédéfinie par ORACLE. Par exemple : -2291 correspond à un problème d'intégrité référentielle.

A noter que les NUMERO_EXCEPTION (ou CODE_ERREUR) sont le plus souvent négatifs.

RAISE nomException : exception utilisateur nommée (3)

Principe

L'utilisateur peut gérer ses propres erreurs, c'est-à-dire des contraintes d'intégrité spécifiques non prises en compte par le SQL.

En déclarant une exception et en utilisant la fonction RAISE nomException, l'utilisateur renvoie le programme vers la zone nomException en cas d'erreur. L'exception est nommée.

Exemple

On considère qu'un nouveau salaire doit toujours être supérieur à l'ancienne valeur.

```
DECLARE
    UPDATE_SAL_EMP EXCEPTION;
    v_emp emp%rowtype;
    v_empno emp.empno%type;
    v_sal emp.sal%type;
BEGIN
    v_empno:=&numEmp;
    v_sal:=&salaire;
    SELECT * into v_emp from emp where empno=v_empno;
    dbms_output.put_line('Employé n° '||v_emp.empno);
    dbms_output.put_line('Nom : '||v_emp.ename||' - salaire :
    '||v_emp.sal);
    if v_sal < v_emp.sal then
         dbms_output.put_line('Le nouveau salaire : '||v_sal||
         ' ne peut pas être inférieur à l''ancien : '||v_emp.sal) ;
         RAISE UPDATE_SAL_EMP ;
    end if;
EXCEPTION
    WHEN UPDATE_SAL_EMP THEN
         dbms_output.put_line('Problème de mise à jour des
    données');
    WHEN NO_DATA_FOUND THEN
         dbms_output.put_line('L''employé n° '||v_empno||' n''existe
    pas');
         dbms_output.put_line('SQLCODE = '||SQLCODE);
         dbms_output.put_line('SQLERRM = '||SQLERRM);
END;
/
Entrez une valeur pour numemp : 7839
ancien   7 :    v_empno:=&numEmp;
nouveau   7 :   v_empno:=7839;
Entrez une valeur pour salaire : 4000
ancien   8 :    v_sal:=&salaire;
nouveau   8 :   v_sal:=4000;
Employé n° 7839
Nom : KING - salaire : 5000
Le nouveau salaire : 4000 ne peut pas être inférieur à l'ancien :
    5000
Problème de mise à jour des données

Procédure PL/SQL terminée avec succès.
```

RAISE_APPLICATION_ERROR : exception utilisateur anonyme (4)

Principe

L'utilisateur peut gérer ses propres erreurs, c'est-à-dire des contraintes d'intégrité spécifiques non prises en compte par le SQL. En utilisant la fonction RAISE_APLICATION_ERREUR, l'utilisateur renvoie le programme vers la zone OTHERS en cas d'erreur. L'exception n'est pas nommée.

Exemple

On considère qu'un nouveau salaire doit toujours être supérieur à l'ancienne valeur.

```
DECLARE
    v_emp emp%rowtype;
    v_empno emp.empno%type;
    v_sal emp.sal%type;
BEGIN
    v_empno:=&numEmp;
    v_sal:=&salaire;
    SELECT * into v_emp from emp where empno=v_empno;
    dbms_output.put_line('Employé n° '||v_emp.empno);
    dbms_output.put_line('Nom : '||v_emp.ename||' - salaire :
    '||v_emp.sal);
    if v_sal < v_emp.sal then
         dbms_output.put_line('Le nouveau salaire : '||v_sal||
         ' ne peut pas être inférieur à l''ancien : '||v_emp.sal) ;
         RAISE_APPLICATION_ERROR(-20000,
         'Problème d''intégrité des données') ;
    end if;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
         dbms_output.put_line('L''employé n° '||v_empno||' n''existe
    pas');
         dbms_output.put_line('SQLCODE = '||SQLCODE);
         dbms_output.put_line('SQLERRM = '||SQLERRM);
    WHEN OTHERS THEN
         dbms_output.put_line('SQLCODE = '||SQLCODE);
         dbms_output.put_line('SQLERRM = '||SQLERRM);
END;
/
Entrez une valeur pour numemp : 7839
ancien   6 :    v_empno:=&numEmp;
nouveau   6 :   v_empno:=7839;
Entrez une valeur pour salaire : 4000
ancien   7 :    v_sal:=&salaire;
nouveau   7 :   v_sal:=4000;
Employé n° 7839
Nom : KING - salaire : 5000
Le nouveau salaire : 4000 ne peut pas être inférieur à l'ancien :
    5000
SQLCODE = -20000
SQLERRM = ORA-20000: Problème d'intégrité des données

Procédure PL/SQL terminée avec succès.
```

- ***Précisions***

La fonction RAISE_APPLICATION_ERROR a deux paramètres :

- Un numéro d'erreur, qu'on choisit entre –20 000 et –20 999 pour éviter les conflits avec les erreurs ORACLE.

- Un message d'erreur.

Ces deux paramètres sont ensuite utilisés par les fonctions SQLCODE et SQLERRM dans la zone d'exception WHEN OTHERS.

Propagation des exceptions : exception et blocs imbriqués.

1) Dans un bloc, en cas d'erreur, on passe à la zone d'exception du bloc.

2) Si l'exception est prise en compte (par son nom ou par le OTHERS), les instructions correspondantes sont exécutées et on quitte le bloc : on revient donc dans le bloc appelant.

3) Si l'exception n'est pas prise en compte, on passe à la zone d'exception du bloc appelant si il existe et on revient à l'étape 2. S'il n'y a pas de bloc appelant, le programme va s'arrêter (planter !) sur cette erreur.

```
DECLARE
    MON_EXCEPTION EXCEPTION;
BEGIN
    BEGIN
         BEGIN
           RAISE MON_EXCEPTION;

         EXCEPTION
           WHEN NO_DATA_FOUND THEN
                 dbms_output.put_line('No date found – bloc 3');
         END;
    dbms_output.put_line('bloc 2');

    EXCEPTION
         WHEN NO_DATA_FOUND THEN
           dbms_output.put_line('No date found – bloc 2');
         WHEN MON_EXCEPTION THEN
           dbms_output.put_line('Mon exception – bloc 2');
         WHEN OTHERS THEN
           dbms_output.put_line('Others – bloc 2');
    END;
    dbms_output.put_line('bloc 1');

EXCEPTION
    WHEN NO_DATA_FOUND THEN
         dbms_output.put_line('No date found – bloc 1');
    WHEN MON_EXCEPTION THEN
         dbms_output.put_line('Mon exception – bloc 1');
    WHEN OTHERS THEN
         dbms_output.put_line('Others – bloc 1');
END;
/
Mon exception - bloc 2
bloc 1

Procédure PL/SQL terminée avec succès.
```

Si on supprime les WHEN MON_EXCEPTION, c'est le WHEN OTHERS qui sera pris en compte.

```
/
Others - bloc 2
bloc 1

Procédure PL/SQL terminée avec succès.
```

12. Les triggers (déclencheurs)

Présentation

Principe

Un trigger est un bloc qui est déclenché automatiquement à chaque occurrence d'un événement déclenchant. Ils n'ont jamais d'arguments. Ils ne peuvent pas être déclenchés « manuellement ».

Usages

Les triggers servent à :

- Maintenir des contraintes d'intégrité non gérées par le DDL.
- Consigner les changements apportés à une table et mettre à jour des statistiques.
- Mettre à jour des données calculées.

3 types de triggers

Les triggers DML

Ils sont déclenchés avant (BEFORE) ou après (AFTER) une instruction du DML : INSERT, UPDATE ou DELETE.

Ce sont les plus utilisés.

Les triggers INSTEAD OF

L'exécution du trigger remplace celle de l'instruction déclenchante.

Les triggers Système

Ils sont déclenchés à l'ouverture ou la fermeture d'une BD, ou lors d'une opération du DDL (create table, drop table, etc.). Pas abordé dans ce poly.

Caractéristiques des triggers DML

syntaxe

```
CREATE [OR REPLACE] TRIGGER [nomSchema.]nomTrigger
    {BEFORE | AFTER | INSTEAD OF} instructionDeclencheur
    [REFERENCING reference]
    [FOR EACH ROW]
    [WHEN condition]
[DECLARE ...]
BEGIN
...
[EXCEPTION ...]
END [nomTrigger];
```

3 caractéristiques principales

- **L'instruction déclenchante :** INSERT et/ou UPDATE et/ou DELETE avec ses prédicats associés : INSERTING, UPDATING, DELETING.
- **Le moment de l'exécution :** AFTER ou BEFORE.
- **Le niveau de l'exécution :** niveau tuple (FOR EACH ROW) ou niveau table. Le niveau tuple permet l'accès au tuple en cours de modification avant modification : OLD, et au tuple en cours de modification après modification : NEW.

2 caractéristiques secondaires liées au niveau de l'exécution

- La condition de déclenchement : WHEN
- Renommer NEW et OLD : REFERENCING

Instruction déclenchante : INSERT, UPDATE, DELETE

syntaxe

```
{BEFORE | AFTER } {INSERT | UPDATE | DELETE }
[OR {INSERT | UPDATE | DELETE }]
[OR {INSERT | UPDATE | DELETE }]
[OF nomAttribut [,nomAttribut ...] ]
ON nomTable
[FOR EACH ROW]
```

Le trigger peut être déclenché sur un INSERT, un UPDATE, un DELETE ou n'importe quelle combinaison des trois.

Il concerne nécessairement une table et éventuellement un ou plusieurs attributs de cette table.

Prédicats associés : INSERTING, UPDATING, DELETING

Dans le corps du trigger, on peut utiliser les prédicats INSERTING, UPDATING, DELETING.

```
IF INSERTING THEN ...
```

Ce sont des booléens. Ils permettent de spécifier une partie du corps du trigger en fonction de la nature de l'instruction déclenchante : INSERT, UPDATE ou DELETE.

Moment d'exécution : BEFORE et AFTER

Un trigger BEFORE est un trigger qui se déclenche avant l'action déclenchante du DML (INSERT, UPDATE ou DELETE) et peut empêcher son exécution.

Il sert à vérifier l'intégrité de la BD. Si l'intégrité des données est altérée, le trigger renverra un message d'erreur et l'action déclenchante du DML ne sera pas effectuée.

Un trigger AFTER est un trigger qui se déclenche après l'action déclenchante. Il ne peut pas empêcher son exécution.

Il sert à mettre à jour la BD du fait de la modification qu'on vient d'y apporter : attributs calculés, statistiques, audit.

Niveau d'exécution : niveau tuple FOR EACH ROW et niveau table

La notion de niveau d'exécution concerne les triggers UPDATE et DELETE.

En effet, l'UPDATE et le DELETE peuvent concerner plusieurs tuples de la table.

Par contre, un INSERT ne concerne qu'un seul tuple.

Lors d'un UPDATE ou d'un DELETE, un trigger peut être déclenché plusieurs fois, à chaque modification de tuple (FOR EACH ROW), ou une seule fois, après toutes les modifications de l'UPDATE ou du DELETE.

On distingue donc entre un NIVEAU TUPLE et un NIVEAU TABLE.

Trigger niveau tuple : « :NEW » et « :OLD »

Pour les triggers de niveau tuple, le tuple avant modification est nommé « :OLD », le tuple après modification est nommé « :NEW ».

:OLD n'est pas défini pour un insert.

:NEW n'est pas défini pour un delete.

On ne peut pas modifier le NEW dans un trigger AFTER.

Trigger niveau table

NEW et OLD ne peuvent pas être utilisés avec un trigger de niveau table.

WHEN : conditionner le déclenchement

Le WHEN permet de conditionner le déclenchement en fonction des valeurs de NEW et de OLD.

```
WHEN (NEW.att1 > NEW.att2)
```

```
WHEN (OLD.att in (1, 2, 3))
```

A noter qu'il n'y a pas de « : » avant le OLD et le NEW.

REFERENCING : renommer NEW et OLD

Le REFERENCING permet de renommer les pseudo-tuples NEW et OLD :

```
{BEFORE | AFTER | INSTEAD OF} instructionDeclencheur
REFERENCING NEW as nouveauNew OlD as nouveauOld
[FOR EACH ROW]
[WHEN condition]
```

A noter qu'il n'y a pas de « : » avant le OLD et le NEW.

Algorithme ORACLE de l'exécution d'une instruction DML

1 : Exécuter les triggers BEFORE de niveau TABLE ;
2 : Pour chaque tuple modifié par l'instruction
    2.1 : Exécuter les triggers BEFORE de niveau TUPLE ;
    2.2 : Exécuter l'instruction
    2.3 : Exécuter les triggers AFTER de niveau TUPLE ;
Fin pour
3 : Exécuter les trigger AFTER de niveau TABLE ;

Exemples de TRIGGER

TRIGGER BEFORE

- ***Vérification de contraintes d'intégrité***

```
Create or replace trigger tbiu_emp
    -- tbiu: trigger before insert or update on emp
    before insert or update of sal on emp
    for each row
declare
    v_empno emp.empno%type;
begin
    dbms_output.put_line('entrée dans tbiu_emp : '||:old.sal||' -
    '||:new.sal);
    if :new.sal < 0 then
         raise_application_error(-20000,'Erreur: salaire négatif');
    elsif :new.sal < :old.sal then
         raise_application_error(-20000,'Erreur: salaire réduit');
    end if ;
end ;
/
```

- ***Mise à jour d'attribut calculé dans la table***

```
alter table emp add saltot number(7,2);
update emp set saltot=sal+nvl(comm,0);

Create or replace trigger tbiu_emp
    before insert or update on emp
    for each row
begin
    :new.saltot:=:new.sal+nvl(:new.comm,0);
end ;
/
```

Remarques

On ne peut pas modifier un NEW dans un trigger after.

On ne peut pas utiliser NEW et OLD dans un trigger de niveau table.

- ***Gestion d'un auto-incrément avec une séquence***

On part de la table « emp » et de la sequence « empSeqEmpno » qui gère le numéro d'employé, « empno » qui est aussi la clé primaire.

```
Create or replace trigger tbi_emp
    -- tbi: trigger before insert on emp
    before insert on emp
    for each row
declare
    v_empno emp.empno%type;
begin
    select empSeqEmpno.nextval into v_empno from dual ;
    :new.empno:= v_empno;
end ;
/
```

TRIGGER AFTER

- ***Mise à jour d'attribut calculé dans la table***

```
Alter table livres add dispo number(1,0) check (dispo in (0,1));
Update livres set dispo=1;
Update livres set dispo=0 where NL in (select NL from emprunter
    where dateret is null);

Create or replace trigger tau_emprunter
after update on emprunter
for each row
begin
    dbms_output.put_line('entrée dans tai_emprunter');
    if :new.dateret is not null then
         update livres set dispo=1 where nl = :new.nl;
         dbms_output.put_line('Le livre n°'|| :new.nl||' est de
    nouveau disponible') ;
    end if;
end;
/

update emprunter set dateret=current_date where dateret is null and
    nl=29;
```

Dans la BD biblio, on ajoute le champ « dispo » dans la table LIVRES : c'est un champ calculé qui dit si le livre est disponible ou pas. On le met à jour. Ensuite on crée un trigger pour que l'attribut calculé soit mis à jour automatiquement.

A noter qu'on pourrait remplacer l'after par un before : on obtiendrait le même résultat.

Les transactions autonomes

Les instructions d'un trigger peuvent toujours être annulées par un ROLLBACK.

Si on veut les valider, il suffit de faire un COMMIT dans le trigger.

Mais ce COMMIT s'appliquera à toutes les instructions actuellement non validées.

Pour limiter l'effet d'un COMMIT dans un trigger aux instructions du trigger, on ajoute dans la déclaration des variables un PRAGMA AUTONOMOUS_TRANSACTION.

```
CREATE [OR REPLACE] TRIGGER
...
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    ...
    COMMIT;
[EXCEPTION ...]
END [nomTrigger];
```

Remarque: ce principe vaut pour tous les blocs ORACLE (Procédures, fonctions, etc.).

Les triggers INSTEAD OF

Les triggers INSTEAD OF permettent de remplacer une instruction du DML par le code du trigger.

Ce type de trigger permet de gérer les évolutions d'un système.

Par exemple: on peut remplacer un DELETE par un UPDATE d'un attribut "supprimé" qui serait de type date et qu'on mettrait à la date du jour.

13. Gestion des triggers

Lister les triggers

La vue USER_TRIGGERS permet de lister les triggers

```
Select trigger_name from user_triggers ;
```

La vue USER_SOURCE permet de lister les triggers, les procédures et les fonctions en distinguant entre les trois :

```
Select distinct name, type from user_source;
```

Voir le code des triggers

La vue USER_SOURCE contient chaque ligne du code des triggers.

```
Set linesize 100
Col text format A80
Select line, text from user_source where name = 'TBIU_EMP';
```

Suppression d'un trigger

```
DROP TRIGGER nomTrigger ;
```

Cette suppression met à jour les vues USER_SOURCE et USER_TRIGGERS.

Débogage

Outils système

```
Show errors ;
```

« Show errors » utilise la vue USER_ERRORS.

```
Set linesize 100
Col object_name format A20
Col object_type format A20
select object_name, object_type, status from user_objects
where object_name='TBIU_EMP';
```

L'attribut STATUS de la vue USER_OBJECTS précise l'état du trigger.

Outils classique

On peut afficher des commentaires dans un trigger via un dbms_output.put_line.

Particulièrement, pour tracer l'entrée dans un trigger.

Nommer des triggers

On ne peut avoir plusieurs triggers BEFORE par action de DML pour une table donnée. Toutefois, comme l'ordre d'exécution des ces triggers n'est pas garanti, il faut mieux n'avoir qu'un trigger BEFORE par action du DML.

Idem pour les triggers AFTER.

14. Packages utilisateur

Principe

Un package permet de regrouper des fonctions et des procédures qui vont ensemble.

On peut y ajouter des variables, des types et des curseurs.

Spécifications et corps du package

Le package fonctionne avec la même logique qu'une classe.

On distingue deux parties dans un package : les spécifications et le corps .

Les spécifications : partie publique

Les spécifications du package décrivent les en-têtes des procédures et des fonctions.

On y trouve aussi les variables globales accessibles au niveau du package, ainsi que les types et les curseurs.

Le corps : partie privée

Le corps du package décrit le corps des procédures et des fonctions publiques, mais aussi des procédures et des fonctions totalement privées dont les en-têtes apparaissent à ce niveau.

Syntaxe

Création d'un package : spécifications

```
Create [or replace] package nom_package
{is | as}
    [Déclaration des variables]
    [Déclaration des variables]
    [Déclaration des en-têtes des procedures et des fonctions]
End [nom_package]
```

Accès aux éléments d'un package :

```
nom_package.nom_element
```

Création du corps d'un package

```
Create [or replace] package body nom_package
{is | as}
    [Déclaration des variables]
    [Déclaration des variables]
    [Déclaration des en-têtes des procédures et des fonctions]
Begin
    Instructions
[Exceptions
    Instructions d'exceptions
]
End [nom_package]
```

A noter qu'il peut exister des packages sans corps : ils ne concerneront donc que des variables et des types. En général, ils contiennent des exceptions.

Suppression d'un package

```
Drop package body nom_package;
```

```
Drop package nom_package;
```

Modification d'un package

On peut faire des « Alter package » mais mieux vaut faire un « create or replace » à partir d'un script sauvegardé.

15. Packages ORACLE

Les packages ORACLE

http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14258/toc.htm

Il existe plus de 100 packages ORACLE.

Les packages ORACLE offrent un ensemble de fonctionnalités supplémentaires.

Il faut les exploiter pour éviter de réécrire des procédures et des fonctions qui existent déjà.

On connaît déjà le dbms_output.put_line. Le package dbms_ouput offre la fonction put_line qui permet de faire de l'affichage dans la calculette SQL.

Exemples de packages :

- DBMS_OUTPUT : pour afficher des messages
- DBMS_LOB : pour travailler avec des données binaires
- DBMS_JOB : pour soumettre des travaux à une fréquence déterminée (scheduler).
- UTL_FILE : pour manipuler des fichiers
- UTL_MAIL : pour envoyer des mails
- DBMS_SQL : pour faire du DDL avec du PLSQL, est avantageusement remplacé par EXECUTE IMMEDIATE
- DBMS_RANDOM : pour générer des nombres aléatoires
- Etc.

DBMS_OUTPUT : pour afficher des messages

http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14258/d_output.htm

La principale fonction de ce package est :

dbms_output.put_line

UTL_FILE : manipuler des fichiers avec PLSQL

http://download.oracle.com/dcs/cd/B19306_01/appdev.102/b14258/u_file.htm

On peut manipuler des fichiers en PLSQL

La syntaxe est proche de celle du langage C :

Type : UTL_FILE.**FILE_TYPE**

FILE_TYPE **FOPEN** (nom_répertoire Chaîne, nom_fichier Chaîne, mode_ouverture caractère, [taille_ligne Entier])

FCLOSE (fichier FILE_TYPE)

FCLOSE_ALL ()

Booléen **IS_OPEN** (fichier FILE_TYPE)

PUT (fichier FILE_TYPE, chaîne) : pas de passage à la ligne

PUT_LINE (fichier FILE_TYPE, chaîne) : avec passage à la ligne

GET_LINE (fichier FILE_TYPE, chaîne)

NEW_LINE (fichier FILE_TYPE, [nb_passages_à_la_ligne Entier])

FFLUSH() : force l'écriture dans le fichier (vide le tampon).

On peut manipuler des répertoires en PL-SQL

```
SQL> CREATE DIRECTORY log_dir AS '/appl/gl/log';
SQL> GRANT READ ON DIRECTORY log_dir TO DBA;
```

Pour plus d'explications, il faut se rendre sur la page de documentation officielle.

DBMS_LOB : pour travailler avec des données binaires

http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14258/d_lob.htm#BABEAJAD

Les LOB sont des gros objets (jusqu'à 4 gigas).

Le package DBMS_LOB permet de lire et de modifier des objets BLOB, CLOB et NLOB qui sont stockés dans la base ORACLE. Il permet aussi la lecture des BFILE stockés en dehors de la base ORACLE.

DBMS_LOB permet uniquement la manipulation d'objets existants déjà.

Variable et type utilisés

Les procédures et les fonctions utilisent des « locators » qui peuvent être vus comme des pointeurs sur le LOB.

Un « locateur » est de type LOB. Il est initialisé par un « select into » qui renvoie une colonne de type LOB.

Le locateur est ensuite utilisé dans les procédures et les fonctions.

Procédures et fonctions LOB

APPEND : remplacer le contenu d'un LOB (un « locator ») par un autre.

COPY : copie un LOB dans un autre

ERASE : efface un LOB

LOADFROMFILE : copie un BFILE dans un LOB

WRITE : force l'écriture

INSTR : recherche dans un LOB

READ : lecture complète d'un LOB

SUBSTR : lecture partielle d'un LOB

Procédures et fonctions BFILE

FILEOPEN : ouvre le fichier BFILE

FILECLOSE : ferme le fichier BFILE

DBMS_SCHEDULER : pour exécuter des tâches à intervalles réguliers

Présentation

Le scheduler ORACLE permet d'exécuter des tâches SQL ou OS à intervalles réguliers.

Il permet de :

- définir la tâche à exécuter : CREATE_PROGRAM
- définir la durée et la fréquence de l'exécution : CREATE_SCHEDULE
- lancer la tâche à exécuter : CREATE_JOB

Les procédures utilisées appartiennent au package : DBMS_SCHEDULER.

Processus dédié : CJQNxx

Le processus CJQNxx a pour rôle d'extraire de la table des JOBS les travaux prêts à s'exécuter.

Codage

Programmation PLSQL via SQLPLUS

Programmation graphique assistée via OEM (Oracle Entreprise Manager).

Création en 3 étapes

- ***CREATE_PROGRAM***

```
BEGIN
DBMS_SCHEDULER.CREATE_PROGRAM (
    program_name => 'nomDuProgramme'
    program_action => 'codeDuProgramme'
    program_type => 'typeDuPrograme'
    comments => 'commentaires'
);
END;
/
```

- Program_name : le nom du programme sera utilisé dans la création du JOB
- Program_action : du code PL_SQL (commence par un BEGIN et finit par un END ;), ou bien un appel à une procédure stockée (le nom de la procédure suffit), un appel à un exé ou un batch OS (le nom du batch avec le chemin et l'extension suffit).
- Program_type : 'PLSQL_BLOCK' pour un bloc PLSQL, 'stored_procedure' pour une procédure stockée, 'exécutable' pour un exécutable OS.
- Comments : des commentaires.

- ***CREATE_SCHEDULE***

```
BEGIN
DBMS_SCHEDULER.CREATE_SCHEDULE (
    schedule_name => 'nomDuScheduler'
    start_date => dateDeDébut
    end_date => dateDeFin
    repeat_intervel => 'fréquenceDExecution'
    comments => 'commentaires'
```

```
);
END;
/
```

- schedule_name : le nom du scheduler sera utilisé dans la création du JOB
- start_date : date de début. SYSTIMETAMP ou SYSDATE par exemple.
- date_end : date de fin. SYSTIMETAMP + INTERVAL '2' DAY par exemple
- repeat_interval : 'FREQ=HOURLY; INTERVAL=1'. MINUTELY, DAYLY, etc.

➢ ***CREATE_JOB***

```
BEGIN
DBMS_SCHEDULER.CREATE_JOB (
    job_name => 'nomDuJob'
    program_name => 'nomDuProgramme'
    schedule_name => 'nomDuScheduler'
);
END;
/
```

- Job_name : le nom du job
- Program_name : le nom du programme créé avec CREATE_PROGRAM
- schedule_name : le nom du scheduler créé avec CREATE_SCHEDULER

Création en 1 étape

Le CREATE_JOB peut reprendre tous les attributs utiles des CREATE_PROGRAM et CREATE_SCHEDULER

➢ ***CREATE_JOB***

```
BEGIN
DBMS_SCHEDULER.CREATE_JOB (
    job_name => 'nomDuJob'
    job_type => 'typeDuPrograme'
    job_action => 'codeDuProgramme'
    start_date => dateDeDébut
    end_date => dateDeFin
    repeat_intervel => 'fréquenceDExecution'
);
END;
/
```

- Job_name : le nom du job
- job_type : équivalent de « program_type »
- job_action : équivalent de « program_action »

➢ ***ENABLE***

ENABLE permet de rendre actif un JOB.

```
BEGIN
DBMS_SCHEDULER.ENABLE ('nomDuJob');
END;
/
```

Suppression d'un JOB

```
BEGIN
```

```
DBMS_SCHEDULER.DROP_JOB (
    job_name => 'nomDuJob'
);
END;
/
```

Consultation: DBA_SCHEDULER_RUNNING_JOB

La vue DBA_SCHEDULER_RUNNING_JOB permet de savoir si un JOB est en cours d'exécution.

➢ ***Attributs :***

- JOB_NAME
- ENABLED : actif ou pas
- STATE : en cours ou pas (entre le start_date et le end_date)
- RUN_COUNT : nombre d'exécutions ou date de début
- MAX_RUNS
- NEXT_RUN_DATE

TP

Soit une table des employés qui contient l'attribut Salaire et l'attribut « numéro de département ». Créer un JOB qui, chaque minute, augmente de 5 tous les employés du département 10.

Vérifier le bon fonctionnement de ce JOB en consultant la table des employés et la vue des JOBS.

Créer ensuite un job qui exécutera toutes les minutes une commande UNIX. Par exemple un « df » pour connaître la taille du disque. Le résultat sera enregistré dans un fichier de log.

UTL_MAIL : pour gérer des mails

http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14258/u_mail.htm#i1001258

Le package UTL_MAIL permet d'envoyer des mails en précisant les pièces jointe, les adresses en copies, si on veut un A.R., etc., autrement dit toutes les possibilités classiques de paramétrage d'un mail.

Les trois fonctions principales sont :

SEND : pour envoyer un mail

SEND_ATTACH_RAW : pour ajouter des pièces jointes

SEND_ATTACH_VARCHAR2 : pour ajouter des pièces jointes de type texte.

DBMS_SQL : pour faire du DDL avec du PLSQL

http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14258/d_sql.htm

Le package DBMS_SQL permet de faire du DDL avec du PL_SQL

Ce package est avantageusement remplacer par la clause EXECUTE IMMEDIATE (cf. paragraphe EXECUTE IMMEDIATE).

16. Classe et objet

Type structuré

Création du type

```
CREATE OR REPLACE TYPE type_adresse
IS OBJECT (
    RUE     VARCHAR(100),
    CP      VARCHAR(10),
    VILLE   VARCHAR(20)
)
/
```

Consultation du type

```
DESC type_adresse ;
```

Création de la table

```
CREATE TABLE ELEVES (
    NE      NUMBER,
    NOM     VARCHAR(50),
    ADRESSETYPE_ADRESSE
) ;
```

Insertion dans la table

```
INSERT INTO ELEVES VALUES(
    1, 'TOTO', TYPE_ADRESSE('30 rue Tartempion', '75010', 'PARIS')
) ;
```

Consultation dans la table

```
SQL> column nom format a20
SQL> column adresse format a60
SQL> select * from eleves;

        NE NOM                  ADRESSE(RUE, CP, VILLE)
---------- -------------------- ----------------------------------
    -------------------------
         1 TOTO                 TYPE_ADRESSE('30 rue Tartempion',
    '75010', 'PARIS')
```

Consultation d'un champ du type structuré : encapsulation

On ne peut pas accéder directement à un champs d'un type structuré car ce sont en réalité des types « classes », auxquels on ajoutera des méthodes pour accéder aux champs.

La programmation des méthodes relèvent du PL SQL.

Type objet

Les types structurés sont en fait des objets.

L'accès aux champs n'est possible qu'à travers des méthodes

```
CREATE OR REPLACE TYPE type_adresse
AS OBJECT (
    RUE     VARCHAR2(100),
    CP      VARCHAR2(10),
```

```
    VILLE  VARCHAR2(20),
Member function cp return varchar2
)
/
```

```
Create or replace type body type_adresse as
Member function cp return varchar2(10) is
Begin
Return self.cp
End ;
End ;
/
```

ORACLE permet de créer des « classes » au sens de la programmation objet : un type structuré auquel sont associées des méthodes : procédures, fonctions et constructeurs surchargeables.

Le constructeur permet d'affecter les champs du tuples en fonction des paramètres passés au type.

Les procédures et fonctions permettent de localiser dans un type les procédures ou les fonctions classiques. Par exemple, dans le type « type_employé », on pourrait ajouter une fonction « ancienneté » qui calcule l'ancienneté de l'employé.