Programmation sous Oracle : PL/SQL

1. Introduction

Le langage PL/SQL (Procedure Language / Structured Query Language) est le langage de prédilection d'Oracle depuis la version 6. Ce langage est une extension procédurale du langage SQL. Il permet de faire cohabiter des structures de contrôle usuelles des langages procéduraux (IF, WHILE, FOR) avec des instructions SQL (SELECT, INSERT, UPDATE...).

Avantages :

- le PL/SQL permet de regrouper dans un même bloc plusieurs instructions SQL qui seront exécutées en une seule fois sur le serveur. Cela permet donc d'alléger le trafic réseau entre le client et le serveur.
- un bloc PL/SQL peut faciliter l'atomicité des transactions (il suffit de réaliser un `COMMIT` si le bloc s'est déroulé correctement, et un `ROLLBACK` dans le cas contraire).
- un bloc PL/SQL peut être nommé pour devenir une fonction ou une procédure stockée, donc réutilisable. Il sera alors compilé qu'une seule fois même si la procédure est ensuite exécutée plusieurs fois, ce qui améliorera les performances.
- le PL/SQL offre des mécanismes pour parcourir les résultats des requêtes (curseurs) et pour traiter des erreurs (exceptions).

2. Paquetage DBMS_OUTPUT

Le paquetage DBMS_OUTPUT propose un ensemble de procédures qui assurent la gestion des entrées/sorties de blocs PL/SQL. Nous utiliserons ce paquetage pour afficher des données à l'écran sous SQL*PLUS.

Les principales procédures sont :

- `PUT(ligne)` : écriture de la ligne *`ligne`* dans le tampon.
- `NEW_LINE` : écriture du caractère de fin de ligne dans le tampon.
- `PUT_LINE(ligne)` : équivalent à `PUT` puis `NEW_LINE`.

Pour que le paquetage `DBMS_OUTPUT` fonctionne dans l'interface iSQL*Plus, il faut l'activer avec la commande `SET SERVEROUTPUT ON` avant de lancer le premier du bloc PL/SQL.

Exemple :

```
SET SERVEROUTPUT ON;
BEGIN
      DBMS_OUTPUT.PUT_LINE('Bonjour');
END;
```

3. Structure d'un bloc PL/SQL

Un bloc PL/SQL peut être composé de trois sections :

- `DECLARE` (optionnel) : déclaration des variables locales.
- `BEGIN .. END` : contient le code PL/SQL. Les instructions sont séparées par un ;
- `EXCEPTION` (optionnel) : permet de traiter les erreurs retournées par le SGBD.

3.1 Déclaration de variables

Un programme PL/SQL est capable de manipuler des variables et des constantes. Les variables et les constantes sont déclarées dans la section `DECLARE`.

Plusieurs types de variables sont manipulés par un programme PL/SQL :

- variables PL/SQL :
  - les types habituels correspondants aux types SQL2 ou Oracle : integer, varchar, number, etc...
  - les types composites adaptés à la récupération des attributs et tuples des tables SQL : %TYPE, %ROWTYPE.
- variables non PL/SQL définies sous SQL*Plus : les variables de substitution.

La déclaration d'une variable est de la forme suivante :

nomVariable [CONSTANT] *typeDeDonnée* [:= *expression*];

Exemple :

```
DECLARE
   C_pi CONSTANT NUMBER := 3.14159 ;
   v_dateNaissance DATE;
   v_nb NUMBER;
   v_nom VARCHAR(30);
   v_existe BOOLEAN := FALSE;
```

3.1.1 Variables %TYPE :

La directive %TYPE déclare une variable selon la définition d'une colonne d'une table existante.

Exemple :

```
DECLARE
   v_idBungalow Bungalows.idBungalow%TYPE ;
```

3.1.2 Variables %ROWTYPE :

La directive %ROWTYPE déclare une variable selon la définition d'un enregistrement d'une table existante.

Exemple :

```
DECLARE
   -- la structure rty_bungalow est composée de toutes les colonnes de la
   -- table bungalows
   rty_bungalow Bungalows%ROWTYPE ;
BEGIN
   …  // il faudra initialiser rty_bungalow ici
   DBMS_OUTPUT.PUT_LINE('L''identifiant du bungalow est ' || rty_bungalow.idBungalow);
   DBMS_OUTPUT.PUT_LINE('Le nom du bungalow est ' || rty_bungalow.nomBungalow) ;
END;
```

3.2 Affectation de variables

Il existe plusieurs possibilités pour affecter une valeur à une variable :

- l'affectation comme on la connaît dans les langages de programmation (`variable := expression`).

Exemple :

```
DECLARE
v_nom VARCHAR(25);
BEGIN
      v_nom := 'Palleja';
      DBMS_OUTPUT.PUT_LINE('Bonjour ' || v_nom);
END;
```

- par la directive `INTO` dans une requête (`SELECT … INTO variable FROM …`).

La clause `INTO` est obligatoire dans une requête SQL et permet de préciser les noms des variables PL/SQL contenant les valeurs renvoyées par la requête (une variable par attribut en respectant l'ordre).

Remarque : Une requête `SELECT … INTO` doit renvoyer une et une seule ligne (une exception sera levée si la requête retourne plusieurs lignes ou bien si elle ne retourne pas de ligne).

Exemple 1 :

```
DECLARE
      v_idCamping Campings.idCamping%TYPE;
      v_nbEmployes NUMBER;
BEGIN
      v_idCamping := 'CAMP1';
      SELECT COUNT(*) INTO v_nbEmployes
      FROM Employes
      WHERE idCamping = v_idCamping;
      DBMS_OUTPUT.PUT_LINE('Il y a ' || v_nbEmployes || ' dans le camping '
                                                      || v_idCamping);
END;
```

Exemple 2 :

```
DECLARE
      v_idCamping Campings.idCamping%TYPE;
      v_nomCamping Campings.nomCamping%TYPE;
      v_villeCamping Campings.villeCamping%TYPE;
BEGIN
      v_idCamping := 'CAMP1';

      -- si la requête retourne plusieurs attributs, il faut une variable par attribut en
      -- respectant l'ordre
      SELECT nomCamping, villeCamping INTO v_nomCamping, v_villeCamping
      FROM CAMPINGS
      WHERE idCamping = v_idCamping;

      DBMS_OUTPUT.PUT_LINE('Nom du camping ' || v_idCamping || ' : ' ||
                                                      v_nomCamping);
      DBMS_OUTPUT.PUT_LINE('Ville du camping ' || v_idCamping || ' : ' ||
                                                      v_villeCamping);
END;
```

Exemple 3 :

```
DECLARE
      v_idCamping Campings.idCamping%TYPE;
      rty_Camping Campings%ROWTYPE;
BEGIN
      v_idCamping := 'CAMP1';

      -- requête qui retourne une variable de type ROWTYPE
      SELECT * INTO rty_camping
      FROM Campings
      WHERE idCamping = v_idCamping;

      DBMS_OUTPUT.PUT_LINE('Nom du camping ' || v_idCamping || ' : ' ||
                                                rty_Camping.nomCamping);
      DBMS_OUTPUT.PUT_LINE('Ville du camping ' || v_idCamping || ' : ' ||
                                                rty_Camping.villeCamping);
      DBMS_OUTPUT.PUT_LINE('Nombre étoiles du camping ' || v_idCamping ||
                                  ' : ' || rty_Camping.nbEtoilesCamping);
END;
```

3.3 Exceptions

Les erreurs qui peuvent survenir dans les programmes PL/SQL déclenchent des exceptions. La partie `EXCEPTION` dans un bloc PL/SQL est facultative. Elle permet de capturer une erreur levée au cours de l'exécution d'une partie de programme (entre un `BEGIN` et un `END`). Une fois levée, l'exception termine le corps principal des instructions et renvoie au bloc `EXCEPTION` du programme.
La plupart des exceptions sont définies en standard mais il est possible de définir ses propres exceptions.

Exceptions prédéfinies lors de l'exécution d'une requête `SELECT .. INTO` :

- `NO_DATA_FOUND` : requête ne retournant aucun résultat
- `TOO_MANY_ROWS` : requête qui retourne plusieurs lignes

Exemple 1 :

```
DECLARE
   v_villeCamping Campings.villeCamping%TYPE;
   v_nomCamping Campings.nomCamping%TYPE;
BEGIN
   v_villeCamping := 'Palavas';
   SELECT nomCamping INTO v_nomCamping
   FROM Campings
   WHERE villeCamping = v_villeCamping;
   DBMS_OUTPUT.PUT_LINE('Le camping qui se trouve à ' || v_villeCamping ||
' s''appelle ' || v_nomCamping);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Il n''y a pas de camping dans le ville de ' ||
                           v_villeCamping);
   WHEN TOO_MANY_ROWS THEN
      DBMS_OUTPUT.PUT_LINE('Il y a plusieurs campings dans la ville de ' ||
                           v_villeCamping);
END;
```

Exemple 2 :

Ici on utilise l'exception pour vérifier l'existence d'un tuple dans une table. Si le tuple n'existe pas, une exception est levée et le code qui suit n'est pas exécuté.

```
DECLARE
      v_idCamping Campings.idCamping%TYPE;
      v_idCampingExiste Campings.idCamping%TYPE;
      v_nbEmployes NUMBER;
BEGIN
      v_idCamping := 'CAMP1';
      -- on vérifie l''existence du camping dans la table Campings
      SELECT idCamping INTO v_idCampingExiste
      FROM CAMPINGS
      WHERE idCamping = v_idCamping;

      -- si le camping v_idcamping n''existe pas, une exception est levée et la requête
      -- suivante n'est pas exécutée
      SELECT COUNT(*) INTO v_nbEmployes
      FROM Employes
      WHERE idCamping = v_idCamping;
      DBMS_OUTPUT.PUT_LINE('Il y a ' || v_nbEmployes || ' employés dans le
                                                camping ' || v_idCamping);
      EXCEPTION
      WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Le camping ' || v_idCamping || ' n''existe pas');
END;
```

3.4 Structures de contrôles

En tant que langage procédural, PL/SQL offre la possibilité de programmer :

- les structures conditionnelles *si* et *cas* (`IF`… et `CASE`) ;
- les structures répétitives *tant que*, *répéter* et *pour* (`WHILE`, `LOOP`, `FOR`).

3.4.1 Structures conditionnelles :

Structure `IF` :

```
IF condition THEN
   instructions;
END IF;
```

```
IF condition THEN
   instructions;
ELSE
   instructions;
END IF;
```

```
IF condition1 THEN
   instructions;
ELSIF condition2 THEN
   instructions;
ELSE
   instructions;
END IF;
```

Exemple 1:

```
DECLARE
   v_nbEmployes NUMBER ;
BEGIN
   SELECT COUNT(*) INTO v_nbEmployes
   FROM Employes
   WHERE idCamping = 'CAMP1';
   IF v_nbEmployes > 100 THEN
      DBMS_OUTPUT.PUT_LINE('Il y a trop d''employés dans le camping');
   ELSIF v_nbEmployes > 50 THEN
      DBMS_OUTPUT.PUT_LINE('Il y a suffisamment d''employés dans le camping');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Il n''y a pas assez d''employés dans le camping');
   END IF ;
END;
```

Exemple 2 :

On reprend l'exemple 2 de la partie 3.3 Exceptions afin de vérifier l'existence d'un tuple dans une table mais en utilisant une structure conditionnelle au lieu d'une exception. Pour cela, on compte le nombre de tuples recherchés dans la table à l'aide de la fonction `COUNT`. Cette fonctionne retourne toujours une ligne (si le tuple n'existe pas, la fonction retourne 0). On ne peut donc pas utiliser d'exception `NO_DATA_FOUND` car celle-ci ne sera jamais levée.

```
DECLARE
      v_idCamping Campings.idCamping%TYPE;
      v_nbCampings NUMBER;
      v_nbEmployes NUMBER;
BEGIN
      v_idCamping := 'CAMP1';
      SELECT COUNT(*) INTO v_nbCampings
      FROM CAMPINGS
      WHERE idCamping = v_idCamping;

      IF v_nbCampings > 0
      THEN
            SELECT COUNT(*) INTO v_nbEmployes
            FROM Employes
            WHERE idCamping = v_idCamping;
            DBMS_OUTPUT.PUT_LINE('Il y a ' || v_nbEmployes || ' employés
                                        dans le camping ' || v_idCamping);
      ELSE
            DBMS_OUTPUT.PUT_LINE('Le camping ' || v_idCamping || ' n''existe pas');
      END IF;
END ;
```

Structure CASE :

```
CASE
   WHEN condition1 THEN instructions1;
   WHEN condition2 THEN instructions2;
   [ELSE instructions;]
END CASE;
```

Exemple:

```
SET SERVEROUTPUT ON;
DECLARE
   v_nbEmployes NUMBER ;
BEGIN
   SELECT COUNT(*) INTO v_nbEmployes
   FROM Employes
   WHERE idCamping = 'CAMP1';
   CASE
      WHEN v_nbEmployes > 100 THEN
         DBMS_OUTPUT.PUT_LINE('Il y a trop d''employés dans le camping');
      WHEN v_nbEmployes > 50 THEN
         DBMS_OUTPUT.PUT_LINE('Il y a suffisamment d''employés dans le camping');
      ELSE
         DBMS_OUTPUT.PUT_LINE('Il n''y a pas assez d''employés dans le camping');
   END CASE;
END;
```

3.4.2 Structures répétitives :

Structure **FOR** :

```
FOR compteur IN [REVERSE] valeurInf..valeurSup LOOP
   Instructions;
END LOOP;
```

Exemple :

```
SET SERVEROUTPUT ON;
DECLARE
   v_somme NUMBER := 0;
BEGIN
   FOR v_compteur IN 1..100 LOOP
      v_somme := v_somme + v_compteur;
   END LOOP ;
   DBMS_OUTPUT.PUT_LINE('Somme = ' || v_somme) ;
END;
```

Structure **WHILE** :

```
WHILE condition LOOP
   instructions;
END LOOP;
```

Exemple :

```
SET SERVEROUTPUT ON;
DECLARE
   v_somme NUMBER := 0;
   v_compteur NUMBER := 1;
BEGIN
   WHILE v_compteur <= 100 LOOP
      v_somme := v_somme + v_compteur;
      v_compteur := v_compteur + 1 ;
   END LOOP ;
   DBMS_OUTPUT.PUT_LINE('Somme = ' || v_somme) ;
END;
```

Structure **LOOP** :

```
LOOP
   instructions;
EXIT WHEN condition
END LOOP;
```

Exemple :

```
SET SERVEROUTPUT ON;
DECLARE
   v_somme NUMBER := 0;
   v_compteur NUMBER := 1;
BEGIN
   LOOP
      v_somme := v_somme + v_compteur;
      v_compteur := v_compteur + 1 ;
   EXIT WHEN v_compteur > 100;
   END LOOP ;
   DBMS_OUTPUT.PUT_LINE('Somme = ' || v_somme) ;
END;
```

4. Fonctions et procédures stockées

Les fonctions et procédures stockées sont des blocs PL/SQL qui sont nommés et capables d'inclure des paramètres en entrée et sortie. Ces sous-programmes sont compilés et stockés dans la base de données.

Avantages :

- performance : ils sont déjà compilés. De plus, ils sont nommés ce qui permet de les invoquer soit directement à partir du client Oracle, soit dans des requêtes ou des blocs PL/SQL, soit dans d'autres procédures ou fonctions soit à partir d'autres programmes clients (application en JAVA par exemple).
- sécurité : il est possible de donner des droits d'accès sur ces sous-programmes.

Il existe deux vues systèmes qui permettent de connaitre les informations sur les procédures et fonctions stockées sous Oracle :

- **user_objects** : permet de voir la liste des procédures et fonctions stockées
- **user_source** : permet de voir le code des fonctions et procédures stockées

Remarque : pour visualiser les erreurs lors de la compilation des sous-programmes, on utilisera la commande `SHOW ERRORS` sous SQL*Plus qui affiche les erreurs du dernier sous-programme compilé.

4.1 Fonctions stockées :

Syntaxe:

```
CREATE [OR REPLACE] FUNCTION nomfonction
[(parametre1 [ IN | OUT | IN OUT ] typeSQL), parametre2…]
RETURN type_de_la_variable_retournée IS
      -- zone de déclaration des variables locales
BEGIN
      -- instructions
      -- clause RETURN
EXCEPTION
      -- traitement des exceptions
      -- clause RETURN
END;
```

Appel des fonctions sous SQL*Plus :

Le moyen le plus simple pour exécuter une fonction sous SQL*Plus est de faire une requête qui utilise la pseudo-table `DUAL`.

```
SELECT nomFonction(paramètre1, paramètre2)
FROM DUAL ;
```

Exemple :

```
CREATE OR REPLACE FUNCTION nbEmployesCamping(p_idCamping Campings.idCamping%TYPE)
                                                          RETURN NUMBER IS
v_nbEmployes NUMBER;
v_idCamping Campings.idCamping%TYPE;
BEGIN
      SELECT idCamping INTO v_idCamping
      FROM CAMPINGS
      WHERE idCamping = p_idCamping;

      SELECT COUNT(*) INTO v_nbEmployes
      FROM Employes
      WHERE idCamping = v_idCamping;
      RETURN v_nbEmployes;

      EXCEPTION
      WHEN NO_DATA_FOUND THEN RETURN NULL;
END;
/
SHOW ERRORS;
```

```
-- appel de la fonction sous SQL*Plus
SELECT nomDuCamping('CAMP1')
FROM DUAL;

-- on peut appeler la fonction sur une table
-- cette requête affiche le nombre d'employés pour chaque camping
SELECT nomCamping, nbEmployesCamping(idCamping)
FROM Campings;

-- on peut appeler une fonction stockée à l'intérieur d'une fonction SQL
-- cette requête affiche le nombre d'employés de tous les campings
SELECT SUM(nbEmployesCamping(idCamping))
FROM Campings;

-- une function peut aussi être appelée dans un bloc PL/SQL, une procedure ou une autre fonction
DECLARE
      v_nbEmployes NUMBER;
BEGIN
      v_nbEmployes := nbEmployesCamping('CAMP1');
      DBMS_OUTPUT.PUT_LINE('Le nombre d''employes du CAMP1 est ' || v_nbEmployes);
END;
```

4.2 Procédures stockées :

Syntaxe :

```
CREATE [OR REPLACE] PROCEDURE nomProcédure
[(parametre1 [ IN | OUT | IN OUT ] typeSQL), parametre2 …] IS
      -- zone de déclaration des variables locales
BEGIN
      -- instructions
EXCEPTION
      -- traitement des exceptions
END;
```

Appel des procédures sous SQL*Plus :

```
CALL nomProcedure(paramètre1,paramètre2) ;
```

Exemple :

```
CREATE OR REPLACE PROCEDURE affichageNbEmployes(p_idCamping IN
                                                Campings.idCamping%TYPE) IS
v_nbEmployes NUMBER;
BEGIN
      -- appel de la fonction précédente
      v_nbEmployes := nbEmployesCamping(p_idCamping);
      IF v_nbEmployes IS NOT NULL
      THEN
            DBMS_OUTPUT.PUT_LINE('Il y a ' || v_nbEmployes || ' employés
                                        dans le camping ' || p_idCamping);
      ELSE
            DBMS_OUTPUT.PUT_LINE('Le camping ' || p_idCamping || ' n''existe pas');
      END IF;
END;
/
CALL affichageNbEmployes('CAMP1');


-- une procédure peut être appelée directement dans un bloc PL/SQL, une procedure ou une autre
-- function (sans utiliser le mot clé CALL).
BEGIN
      DBMS_OUTPUT.PUT_LINE('Bonjour');
      affichageNbEmployes('CAMP1');
END
```

4.3 Paramètres d'une procedure ou d'une fonction :

Un sous-programme peut inclure des paramètres qui peuvent être en entrée ou en sortie.

- IN : paramètre en entrée, non modifiable par le sous-programme (accessible en lecture seule)
- OUT : paramètre en sortie, peut être modifié par le sous-programme et transmis au programme appelant (accessible en écriture)
- IN OUT : à la fois en entrée et en sortie (accessible en lecture et écriture)
- par défaut : IN

Lors de la déclaration du sous-programme on peut préciser si le paramètre est en entrée ou sortie. Si rien n'est indiqué, le paramètre sera par défaut en entrée.

5. Les curseurs

On utilise un curseur PL/SQL pour parcourir le résultat d'une requête SQL renvoyant plusieurs lignes.
Un curseur est un pointeur vers une zone mémoire SQL privée allouée pour le traitement d'une instruction SQL. Le curseur permet de parcourir et traiter séquentiellement les tuples ramenés par une requête SQL.

5.1 Fonctionnement

- déclaration du curseur dans la partie `DECLARE`
- ouverture du curseur avec la directive `OPEN` (chargement du résultat de la requête SQL)
- parcours du curseur en récupérant les lignes une par une dans une variable locale
- fermeture du curseur

Syntaxe :

```
DECLARE
-- Déclaration du curseur (un curseur peut contenir des paramètres)
CURSOR nomCurseur [(déclaration des paramètres)] IS SELECT …;

BEGIN
      -- Ouverture du curseur (chargement des lignes).
      OPEN nomCurseur [(valeur des paramètres)];

      -- chargement de l'enregistrement courant dans une ou plusieurs
      variables ou dans une variable de type ROWTYPE et positionnement sur
      la ligne suivante
      FETCH nomCurseur INTO listeVariables | nomVariable;
      …
      -- Ferme le curseur. L'exception INVALID_CURSOR se déclenche si des
      accès au curseur sont opérés après sa fermeture.
     CLOSE nomCurseur;
END ;
```

5.2 Attributs du curseur

`nomCurseur%ISOPEN`	retourne TRUE si le curseur est ouvert, FALSE sinon
`nomCurseur%FOUND`	retourne TRUE si le dernier FETCH a renvoyé une ligne, FALSE sinon
`nomCurseur%NOTFOUND`	retourne TRUE si le dernier FETCH n'a pas renvoyé de ligne, FALSE sinon
`nomCurseur%ROWCOUNT`	retourne le nombre total de lignes traitées jusqu'à présent (0 si le curseur est ouvert mais n'a pas encore été lu, puis la valeur s'incrémente de 1 après chaque FETCH)

5.3 Parcours d'un curseur

Suivant le traitement à parcourir sur un curseur, on peut choisir d'utiliser une structure répétitive `tant que`, `répéter` ou `pour`.

Parcours avec WHILE :

```
DECLARE
   CURSOR curs_employes IS SELECT nomEmploye, prenomEmploye
                           FROM Employes
                           WHERE idCamping = 'CAMP1'
                           ORDER BY salaireEmploye;
   v_ligne curs_employes%ROWTYPE;
BEGIN
   OPEN curs_employes;
   FETCH curs_employes INTO v_ligne;
   WHILE (curs_employes%FOUND) LOOP
      DBMS_OUTPUT.PUT_LINE(v_ligne.nomEmploye || ' ' || v_ligne.prenomEmploye);
      FETCH curs_employes INTO v_ligne;
   END LOOP;
   CLOSE curs_employes;
END;
```

Parcours avec LOOP :

```
DECLARE
   CURSOR curs_employes IS SELECT nomEmploye, prenomEmploye
                           FROM Employes
                           WHERE idCamping = 'CAMP1'
                           ORDER BY salaireEmploye;
   v_ligne curs_employes%ROWTYPE;
BEGIN
   OPEN curs_employes;
   LOOP
        FETCH curs_employes INTO v_ligne;
        EXIT WHEN curs_employes%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_ligne.nomEmploye || ' ' || v_ligne.prenomEmploye);
   END LOOP;
   CLOSE curs_employes;
END;
```

Parcours avec FOR :

L'utilisation d'une boucle FOR facilite la programmation car elle évite les directives `OPEN`, `FETCH` et `CLOSE`.

```
DECLARE
   CURSOR curs_employes IS SELECT nomEmploye, prenomEmploye
                           FROM Employes
                           WHERE idCamping = 'CAMP1'
                           ORDER BY salaireEmploye;
BEGIN
   FOR v_ligne IN curs_employes LOOP
      DBMS_OUTPUT.PUT_LINE(v_ligne.nomEmploye || ' ' || v_ligne.prenomEmploye);
   END LOOP;
END;
```

On peut aussi déclarer le curseur à l'intérieur de l'instruction FOR mais dans ce cas il ne sera pas possible de réutiliser le curseur puisqu'il n'a d'existence que dans la boucle.

```
BEGIN
   FOR v_ligne IN (SELECT nomEmploye, prenomEmploye
                   FROM Employes
                   WHERE idCamping = 'CAMP1'
                   ORDER BY salaireEmploye) LOOP
      DBMS_OUTPUT.PUT_LINE(v_ligne.nomEmploye || ' ' || v_ligne.prenomEmploye);
   END LOOP;
END;
```

5.4 Curseur paramétré

Un curseur paramétré permet de réutiliser le même curseur aves des valeurs différentes dans un même bloc PL/SQL.

Exemple 1 :

```
DECLARE
   CURSOR curs_employes (v_idCamping Campings.idCamping%TYPE) IS
                           SELECT nomEmploye, prenomEmploye
                           FROM Employes
                           WHERE idCamping = v_idCamping
                           ORDER BY salaireEmploye;
BEGIN
   FOR v_ligne IN curs_employes('CAMP1') LOOP
      DBMS_OUTPUT.PUT_LINE(v_ligne.nomEmploye || ' ' || v_ligne.prenomEmploye);
   END LOOP;

   FOR v_ligne IN curs_employes('CAMP2') LOOP
      DBMS_OUTPUT.PUT_LINE(v_ligne.nomEmploye || ' ' || v_ligne.prenomEmploye);
   END LOOP;
END;
```

Exemple 2 :

```
DECLARE
   CURSOR curs_campings IS SELECT idCamping, nomCamping
                           FROM Campings
                           ORDER BY nomCamping;
   CURSOR curs_employes (v_idCamping Campings.idCamping%TYPE) IS
                           SELECT nomEmploye, prenomEmploye, salaireEmploye
                           FROM Employes
                           WHERE idCamping = v_idCamping
                           ORDER BY salaireEmploye;
BEGIN
   FOR v_camping IN curs_campings LOOP
      DBMS_OUTPUT.PUT_LINE(v_camping.nomCamping);
      FOR v_employe IN curs_employes(v_camping.idCamping) LOOP
         DBMS_OUTPUT.PUT_LINE(v_employe.nomEmploye || ' ' ||
               v_employe.prenomEmploye || ' ' || v_employe.salaireEmploye);
      END LOOP;
   END LOOP;
END;
```