# 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 facilite 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.
- 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 :

- `ENABLE` : active le paquetage en début de session.
- `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 la première instruction du bloc PL/SQL.

## 3. Structure d'un bloc PL/SQL

Un bloc PL/SQL est 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 VARCHAR2(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
   …
   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;
```

### 3.1.3 Variables de substitution

Il est possible de passer en paramètre d'entrée d'un bloc PL/SQL des variables saisies sous SQL*Plus. Ces variables sont dites de substitution. On accède aux valeurs d'une telle variable dans le code PL/SQL en faisant préfixer le nom de la variable du symbole « & » (avec ou sans guillemets simples suivant qu'il s'agit d'un nombre ou pas).

Exemple :

```
ACCEPT s_nomClient PROMPT 'Saisir le nom du client' ;
ACCEPT s_ageClient PROMPT 'Saisir l''age du client' ;
DECLARE
   v_nomClient VARCHAR2(30);
   v_ageClient NUMBER;
BEGIN
   v_nomClient := '&s_nomClient';
   v_ageClient := &s_ageClient;
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`).
- 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 un seul tuple.

Exemple :

```
SET SERVEROUTPUT ON;
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 || ' employés dans le camping');
END;
```

## 3.3 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`).

### 3.3.1 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:

```
SET SERVEROUTPUT ON;
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;
```

**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;
```

### 3.3.2 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;
```

## 3.4 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

```
SET SERVEROUTPUT ON;
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 de la ville de ' || 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;
```

# 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.

## *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 1 :

```
CREATE OR REPLACE FUNCTION nomDuCamping(p_idCamping IN
                                    Campings.idCamping%TYPE) RETURN VARCHAR IS
   v_nomCamping Campings.nomCamping%TYPE;
BEGIN
   SELECT nomCamping INTO v_nomCamping
   FROM Campings
   WHERE idCamping = p_idCamping;
   RETURN v_nomCamping;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RETURN 'n''existe pas';
END;
```

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

Exemple 2 :

```
-- cette fonction ne lèvera pas d'exception car la fonction SQL COUNT retourne toujours une valeur
CREATE OR REPLACE FUNCTION nbServicesBungalow(p_idBungalow IN
                                    Bungalows.idBungalow%TYPE) RETURN NUMBER IS
   v_nbServices NUMBER;
BEGIN
   SELECT COUNT(*) INTO v_nbServices
   FROM Proposer
   WHERE idBungalow = p_idBungalow;
   RETURN v_nbServices;
END;
```

Exemple 3 :

```
-- appel d'une fonction dans la requête SQL
CREATE OR REPLACE FUNCTION nbServicesCamping(p_idCamping IN
                                     Campings.idCamping%TYPE) RETURN NUMBER IS
   v_nbServices NUMBER;
BEGIN
   -- appel de la fonction nbServicesBungalow dans la requête
   SELECT SUM(nbServicesBungalow(idBungalow)) INTO v_nbServices
   FROM Bungalows
   WHERE idCamping = p_idCamping;
   RETURN v_nbServices;
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;
```

Passage des paramètres :

- IN : paramètre en entrée, non modifiable par la procédure (accessible en lecture seule)
- OUT : paramètre en sortie, peut être modifié par la procédure, 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

**Appel des procédures sous SQL*Plus :**

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

Exemple 1 :

```
CREATE OR REPLACE PROCEDURE affichageNomCamping(p_idCamping IN
                                                 Campings.idCamping%TYPE) IS
      v_nomCamping Campings.nomCamping%TYPE;
BEGIN
      -- appel de la fonction nomDuCamping
      v_nomCamping := nomDuCamping(p_idCamping);
      DBMS_OUTPUT.PUT_LINE('Le camping ' || v_nomCamping);
END;
```

```
-- appel de la procédure sous SQL*Plus
SET SERVEROUTPUT ON;
CALL affichageNomCamping('CAMP1');
```

Exemple 2 : cette procédure réalise la même chose que la fonction **nbServicesBungalow** écrite précédemment.

```
-- cette procédure écrit le résultat dans le paramètre p_resultat qui est accessible en écriture (OUT)
CREATE OR REPLACE PROCEDURE nbServicesBungalow(p_idBungalow IN
                        Bungalows.idBungalow%TYPE, p_resultat OUT NUMBER) IS
BEGIN
      SELECT COUNT(*) INTO p_resultat
      FROM Proposer
      WHERE idBungalow = p_idBungalow;
END;

-- appel de la procédure dans un bloc PL/SQL
SET SERVEROUTPUT ON;
DECLARE
      v_nbServices NUMBER;
BEGIN
      nbservicesbungalow('B2',v_nbServices);
      DBMS_OUTPUT.PUT_LINE(v_nbServices);
END;
```

**Remarque** : pour visualiser les erreurs lors de la compilation des sous-programmes, vous pouvez utiliser la commande `SHOW ERRORS` sous SQL*Plus.

# 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.

```
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;
```