Notes de cours

1.1 Introduction au PL/SQL

1.1.1 PL/SQL

Le PL de PL/SQL signifie Procedural Language. Il s'agit d'une extension procédurale du SQL permettant d'effectuer des traitements complexes sur une base de données. Les possibilités offertes sont les mêmes qu'avec des langages impératifs (instructions en séquence) classiques.

Ecrivez-le dans un éditeur dont vous copierez le contenu dans `SQL+`. Un script écrit en `PL/SQL` se termine obligatoirement par un `/`, sinon `SQL+` ne l'interprète pas. S'il contient des erreurs de compilation, il est possible d'afficher les messages d'erreur avec la commande `SQL+ : SHOW ERRORS`.

1.1.2 Blocs

Tout code écrit dans un langage procédural est formé de blocs. Chaque bloc comprend une section de déclaration de variables, et un ensemble d'instructions dans lequel les variables déclarées sont visibles.

La syntaxe est

```
DECLARE
        /* declaration de variables */
BEGIN
        /* instructions a executer */
END;
```

1.1.3 Affichage

Pour afficher le contenu d'une variable, les procédures `DBMS_OUTPUT.PUT()` et `DBMS_OUTPUT.PUT_LINE()` prennent en argument une valeur à afficher ou une variable dont la valeur est à afficher. Par défaut, les fonctions d'affichage sont desactivées. Il convient, à moins que vous ne vouliez rien voir s'afficher, de les activer avec la commande `SQL+ SET SERVEROUTPUT ON`.

1.1.4 Variables

Une variable se déclare de la sorte :

```
nom type [:= initialisation] ;
```

L'initisation est optionnelle. Nous utiliserons les mêmes types primitifs que dans les tables. Par exemple :

```
SET SERVEROUTPUT ON
DECLARE
        c varchar2(15) := 'Hello World !';
BEGIN
        DBMS_OUTPUT.PUT_LINE(c);
END;
/
```

Les affectations se font avec la syntaxe `variable := valeur ;`

1.1.5 Traitements conditionnels

Le `IF` et le `CASE` fonctionnent de la même façon que dans les autres langages impératifs :

```
IF /* condition 1 */ THEN
        /* instructions 1 */
ELSE
        /* instructions 2 */
END IF;
```

voire

```
IF /* condition 1 */ THEN
        /* instructions 1 */
ELSIF /* condition 2 */
        /* instructions 2 */
ELSE
        /* instructions 3 */
END IF;
```

Les conditions sont les mêmes qu'en SQL. Le `switch` du langage C s'implémente en PL/SQL de la façon suivante :

```
CASE /* variable */
WHEN /* valeur 1 */ THEN
        /* instructions 1 */
WHEN /* valeur 2 */ THEN
        /* instructions 2 */
...
WHEN /* valeur n */ THEN
        /* instructions n */
ELSE
        /* instructions par défaut */
END CASE;
```

1.1.6 Traitements répétitifs

`LOOP ... END LOOP ;` permet d'implémenter les boucles

```
LOOP
        /* instructions */
END LOOP;
```

L'instruction `EXIT WHEN` permet de quitter une boucle.

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

La boucle `FOR` existe aussi en PL/SQL :

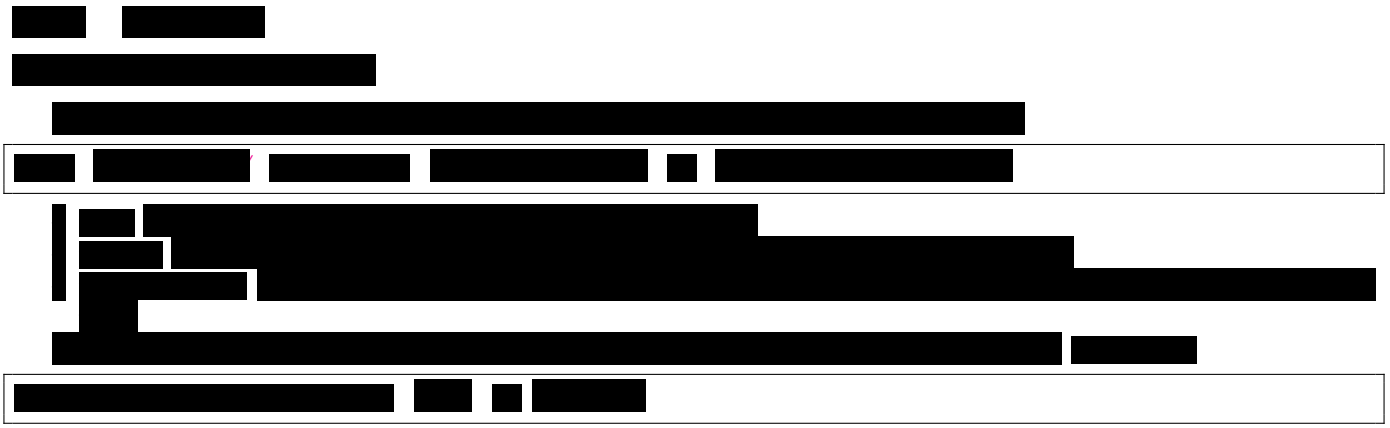
```
FOR /* variable */ IN /* inf */..*/ sup */ LOOP
        /* instructions */
END LOOP;
```

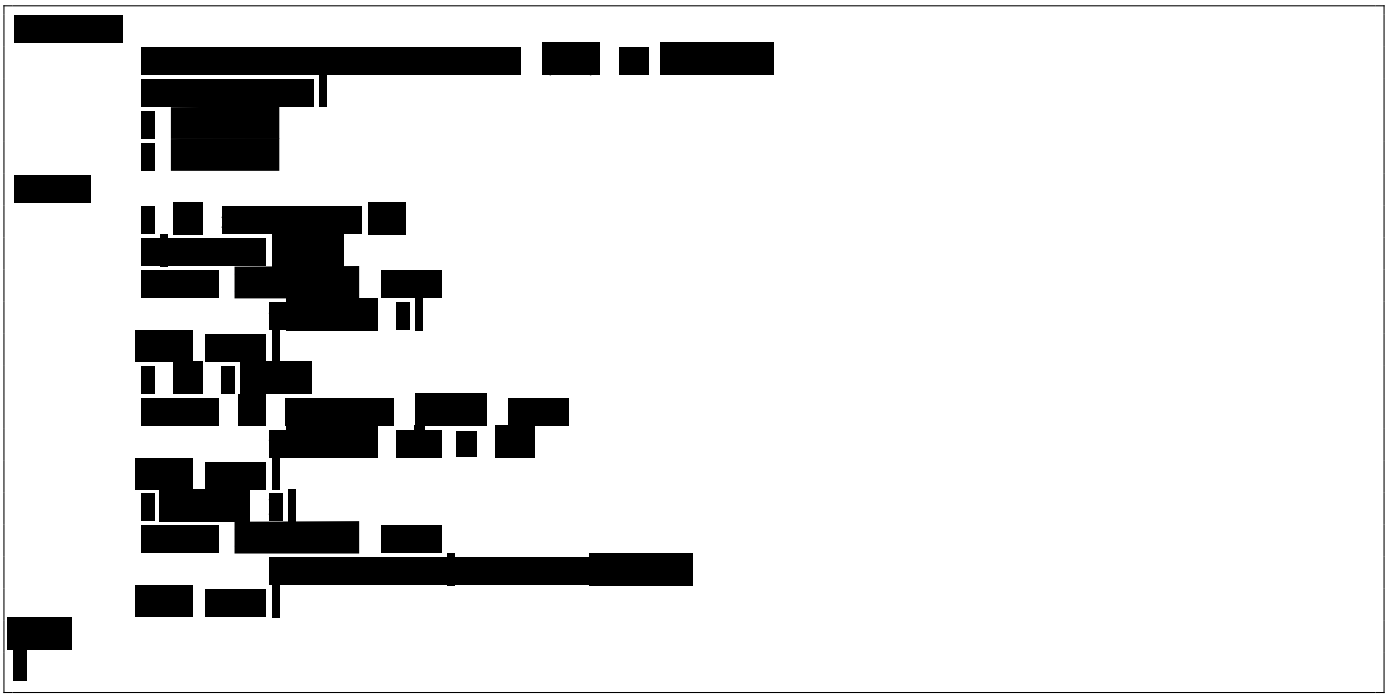
Ainsi que la boucle `WHILE` :

```
WHILE /* condition */ LOOP
        /* instructions */
END LOOP;
```

[redacted]

1.2 Tableaux et structures





1.2.2 Structures

Un structure est un type regroupant plusieurs types. Une variable de type structuré contient plusieurs variables, ces variables s'appellent aussi des champs.

Création d'un type structuré

On définit un type structuré de la sorte :

```
TYPE /* nomType */ IS RECORD
(
        /* liste des champs */
);
```

`nomType` est le nom du type structuré construit avec la syntaxe précédente. La liste suit la même syntaxe que la liste des colonnes d'une table dans un `CREATE TABLE`. Par exemple, construisons le type `point` (dans $\mathbb{R}^2$),

```
TYPE point IS RECORD
(
        abscisse NUMBER,
        ordonnee NUMBER
);
```

Notez bien que les types servant à définir un type structuré peuvent être quelconques : variables scalaires, tableaux, structures, etc.

Déclaration d'une variable de type structuré

`point` est maintenant un type, il devient donc possible de créer des variables de type point, la règle est toujours la même pour déclarer des variables en `PL/SQL`, par exemple

```
p point;
```

permet de déclarer une variable `p` de type `point`.

Utilisation d'une variable de type structuré

Pour accéder à un champ d'une variable de type structuré, en lecture ou en écriture, on utilise la notation pointée : `v.c` est le champ appelé `c` de la variable structuré appelée `v`. Par exemple,

```
DECLARE
        TYPE point IS RECORD
        (
        abscisse NUMBER,
        ordonnee NUMBER
        );
        p point;
BEGIN
        p.abscisse := 1;
        p.ordonnee := 3;
        DBMS_OUTPUT.PUT_LINE('p.abscisse = ' || p.abscisse ||
                ' and p.ordonnee =' ||p.ordonnee);
END;
/
```

Le script ci-dessous crée le type `point`, puis crée une variable `t` de type `point`, et enfin affecte aux champs `abscisse` et `ordonnee` du point `p` les valeurs 1 et 3.

1.3 Utilisation du PL/SQL

Ce cours est une introduction aux interactions possibles entre la base de données et les scripts `PL/SQL`.

1.3.1 Affectation

On place dans une variable le résultat d'une requête en utilisant le mot-clé `INTO`. Les instructions

```
SELECT champ_1, ..., champ_n INTO v_1, ..., v_n
FROM ...
```

affecte aux variables `v_1, ..., v_n` les valeurs retournées par la requête. Par exemple

```
DECLARE
        num NUMBER;
        nom VARCHAR2(30) := 'Poupée Batman';
BEGIN
        SELECT numprod INTO num
                FROM PRODUIT
                WHERE nomprod = nom;
        DBMS_OUTPUT.PUT_LINE('L''article ' ||
                nom || ' a pour numéro ' || num);
END;
/
```

Prêtez attention au fait que la requête doit retourner une et une une seule ligne, sinon, une erreur se produit à l'exécution.

1.3.2 Tables et structures

Si vous ne tenez pas à vous prendre la tête pour choisir le type de chaque variable, demandez-vous ce que vous allez mettre dedans ! Si vous tenez à y mettre une valeur qui se trouve dans une colonne d'une table, il est possible de vous référer directement au type de cette colonne avec le type `nomTable.nomColonne%type`. Par exemple,

```
DECLARE
        num PRODUIT.numprod%type;
        nom PRODUIT.nomprod%type := 'Poupée Batman';
BEGIN
        SELECT numprod INTO num
                FROM PRODUIT
                WHERE nomprod = nom;
        DBMS_OUTPUT.PUT_LINE('L''article ' ||
                nom || ' a pour numéro ' || num);
END;
/
```

Pour aller plus loin, il est même possible de déclarer une structure pour représenter une ligne d'une table, le type porte alors le nom suivant : `nomTable%rowtype`.

```
DECLARE
        nom PRODUIT.nomprod%type := 'Poupée Batman';
        ligne PRODUIT%rowtype;
BEGIN
        SELECT * INTO ligne
                FROM PRODUIT
                WHERE nomprod = nom;
        DBMS_OUTPUT.PUT_LINE('L''article ' ||
                ligne.nomprod || ' a pour numéro ' || ligne.numprod);
END;
/
```

1.3.3 Transactions

Un des mécanismes les plus puissants des SGBD récents réside dans le système des transactions. Une transaction est un ensemble d'opérations "atomiques", c'est-à-dire indivisible. Nous considérerons qu'un ensemble d'opérations est indivisible si une exécution partielle de ces instructions poserait des problèmes d'intégrité dans la base de données. Par exemple, dans le cas d'une base de données de gestion de comptes en banque, un virement d'un compte à un autre se fait en deux temps : créditer un compte d'une somme s, et débiter un autre de la même somme s. Si une erreur survient pendant la deuxième opération, et que la transaction est interrompue, le virement est incomplet et le patron va vous assassiner.

Il convient donc de disposer d'un mécanisme permettant de se protéger de ce genre de désagrément. Plutôt que se casser la tête à tester les erreurs à chaque étape et à balancer des instructions permettant de "revenir en arrière", nous allons utiliser les instructions `COMMIT` et `ROLLBACK`.

Voici le squelette d'un exemple :

```
/* instructions */
IF /* erreur */ THEN
        ROLLBACK;
ELSE
        COMMIT;
END;
```

Le `ROLLBACK` annule toutes les modifications faites depuis le début de la transaction (donc depuis le précédent `COMMIT`), `COMMIT` les enregistre définitivement dans la base de données.

La variable d'environnement `AUTOCOMMIT`, qui peut être positionnée à `ON` ou à `OFF` permet d'activer la gestion des transactions. Si elle est positionnée à `ON`, chaque instruction a des répercussions immédiates dans la base, sinon, les modifications ne sont effectives qu'une fois qu'un `COMMIT` a été exécuté.

1.4 Exceptions

Le mécanisme des `exceptions` est implémenté dans la plupart des langages récent, notament orientés objet. Cette façon de programmer a quelques avantages immédiats :

- **obliger les programmeurs à traiter les erreurs :** combien de fois votre prof de C a hurlé en vous suppliant de vérifier les valeurs retournées par un `malloc`, ou un `fopen` ? La plupart des compilateurs des langages à `exceptions` (notamment java) ne compilent que si pour chaque erreur potentielle, vous avez préparé un bloc de code (éventuellement vide...) pour la traiter. Le but est de vous assurer que vous n'avez pas oublié d'erreur.
- **Rattraper les erreurs en cours d'exécution :** Si vous programmez un système de sécurité de centrale nucléaire ou un pilote automatique pour l'aviation civile, une erreur de mémoire qui vous afficherait l'écran bleu de windows, ou le message "Envoyer le rapport d'erreur ?", ou plus simplement le fameux "Segmentation fault" produirait un effet des plus mauvais. Certaines erreurs d'éxecution sont rattrapables, autrement dit, il est possible de résoudre le problème sans interrompre le programme.
- **Ecrire le traitement des erreurs à part :** Pour des raisons fiabilité, de lisibilité, il a été considéré que mélanger le code "normal" et le traitement des erreurs était un style de programmation perfectible... Dans les langages à exception, les erreurs sont traitées à part.

1.4.1 Rattraper une exception

Je vous ai menti dans le premier cours, un bloc en `PL/SQL` a la forme suivante :

```
DECLARE
        /* declarations */
BEGIN
        /* instructions */
EXCEPTION
        /* traitement des erreurs */
END;
```

Une exception est une "erreur type", elle porte un nom, au même titre qu'une variable a une identificateur, par exemple `GLUBARF`. Lorsque dans les instructions, l'erreur `GLUBARF` se produit, le code du `BEGIN` s'interrompt et le code de la section `EXCEPTION` est lancé. On dit aussi que quand une exception est **levée** (raised) (on dit aussi **jetée** (thrown)), on la **rattrape** (catch) dans le bloc `EXCEPTION`. La section `EXCEPTION` a la forme suivante :

```
EXCEPTION
        WHEN E1 THEN
                /* traitement */
        WHEN E2 THEN
                /* traitement */
        WHEN E3 THEN
                /* traitement */
        WHEN OTHERS THEN
                /* traitement */
END;
```

On énumère les erreurs les plus pertinentes en utilisant leur nom et en consacrant à chacune d'elle un traitement particulier pour rattraper (ou propager) l'erreur. Quand un bloc est traité, les `WHEN` suivants ne sont pas évalués. `OTHERS` est l'exception par défaut, `OTHERS` est toujours vérifié, sauf si un cas précédent a été vérifié. Dans l'exemple suivant :

```
DECLARE
        /* declarations */
BEGIN
        /* instructions */
        COMMIT;
EXCEPTION
        WHEN GLUBARF THEN
                ROLLBACK;
                DBMS_OUTPUT.PUT_LINE('GLUBARF exception raised!');
        WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
                DBMS_OUTPUT.PUT_LINE('SQLERRM = ' || SQLERRM);
```

```
END;
```

Les deux variables globales `SQLCODE` et `SQLERRM` contiennent respectivement le code d'erreur Oracle et un message d'erreur correspondant à la dernière exception levée. Chaque exception a donc, en plus d'un nom, un code et un message.

1.4.2 Exceptions prédéfinies

Bon nombre d'exceptions sont prédéfinies par Oracle, par exemple

- `NO_DATA_FOUND` est levée quand la requête d'une instruction de la forme `SELECT ... INTO ...` ne retourne aucune ligne
- `TOO_MANY_ROWS` est levée quand la requête d'une instruction de la forme `SELECT ... INTO ...` retourne plusieurs lignes
- `DUP_VAL_ON_INDEX` est levée si une insertion (ou une modification) est refusée à cause d'une contrainte d'unicité.

On peut enrichir notre exemple de la sorte :

```
DECLARE
        num NUMBER;
        nom VARCHAR2(30) := 'Poupée Batman' ;
BEGIN
        SELECT numprod INTO num
                FROM PRODUIT
                WHERE nomprod = nom;
        DBMS_OUTPUT.PUT_LINE('L''article ' ||
                nom || ' a pour numéro ' || num);
EXCEPTION
        WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE('Aucun article ne porte le nom '
                || nom);
        WHEN TOO_MANY_ROWS THEN
                DBMS_OUTPUT.PUT_LINE('Plusieurs articles portent le nom '
                || nom);
        WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE('Il y a un gros problème...');
END;
/
```

`SELECT numprod INTO num...` lève une exception si la requête renvoie un nombre de lignes différent de 1.

1.4.3 Codes d'erreur

Je vous encore menti, certaines exceptions n'ont pas de nom. Elle ont seulement un code d'erreur, il est conseillé de se reporter à la documentation pour les obtenir. On les traite de la façon suivante

```
EXCEPTION
        WHEN OTHERS THEN
                IF SQLCODE = CODE1 THEN
                        /* traitement */
                ELSIF SQLCODE = CODE2 THEN
                        /* traitement */
                ELSE
                        DBMS_OUTPUT.PUT_LINE('J''vois pas c''que ca
                        peut etre...');
END;
```

C'est souvent le cas lors de violation de contraintes.

1.4.4 Déclarer et lancer ses propres exceptions

Exception est un type, on déclare donc les exceptions dans une section DECLARE. Une exception se lance avec l'instruction `RAISE`. Par exemple,

```
DECLARE
        GLUBARF EXCEPTION;
BEGIN
        RAISE GLUBARF;
EXCEPTION
        WHEN GLUBARF THEN
                DBMS_OUTPUT.PUT_LINE('glubarf raised.');
END;
/
```

1.5 Sous-programmes

1.5.1 Procédures

Syntaxe

On définit une procédure de la sorte

```
CREATE OR REPLACE PROCEDURE /* nom */ (/* parametres */) IS
        /* declaration des variables locales */
BEGIN
        /* instructions */
END;
```

les paramètres sont une simple liste de couples `nom type`. Par exemple, la procedure suivante affiche un compte à rebours.

```
CREATE OR REPLACE PROCEDURE compteARebours (n NUMBER) IS
BEGIN
        IF n >= 0 THEN
                DBMS_OUTPUT.PUT_LINE(n);
                compteARebours(n - 1);
        END IF;
END;
```

Invocation

En `PL/SQL`, une procédure s'invoque tout simplement avec son nom. Mais sous `SQL+`, on doit utiliser le mot-clé `CALL`. Par exemple, on invoque le compte à rebours sous `SQL+` avec la commande `CALL compteARebours(20)`.

Passage de paramètres

Oracle permet le passage de paramètres par référence. Il existe trois types de passage de paramètres :
- `IN` : passage par valeur
- `OUT` : aucune valeur passée, sert de valeur de retour
- `IN OUT` : passage de paramètre par référence

Par défaut, le passage de paramètre se fait de type `IN`.

```
CREATE OR REPLACE PROCEDURE incr (val IN OUT NUMBER) IS
BEGIN
        val := val + 1;
END;
```

1.5.2 Fonctions

Syntaxe

On crée une nouvelle fonction de la façon suivante :

```
CREATE OR REPLACE FUNCTION /* nom */ (/* parametres */) RETURN /* type
*/ IS
        /* declaration des variables locales */
BEGIN
        /* instructions */
END;
```

L'instruction `RETURN` sert à retourner une valeur. Par exemple,

```
CREATE OR REPLACE FUNCTION module (a NUMBER, b NUMBER) RETURN NUMBER IS
BEGIN
        IF a < b THEN
                RETURN a;
        ELSE
```

```
                RETURN module(a - b, b);
        END IF;
END;
```

[redacted]

[redacted]

[redacted]

1.6 Curseurs

1.6.1 Introduction

Les instructions de type `SELECT ... INTO ...` manquent de souplesse, elles ne fontionnent que sur des requêtes retourant une et une seule valeur. Ne serait-il pas intéressant de pouvoir placer dans des variables le résultat d'une requête retournant plusieurs lignes ? A méditer...

1.6.2 Les curseurs

Un curseur est un objet contenant le résultat d'une requête (0, 1 ou plusieurs lignes).

déclaration

Un curseur se déclare dans une section `DECLARE` :

```
CURSOR /* nomcurseur */ IS /* requête */;
```

Par exemple, si on tient à récupérer tous les employés de la table `EMP`, on déclare le curseur suivant.

```
CURSOR emp_cur IS
        SELECT * FROM EMP;
```

Ouverture

Lors de l'ouverture d'un curseur, la requête du curseur est évaluée, et le curseur contient toutes les données retournées par la requête. On ouvre un curseur dans une section `BEGIN` :

```
OPEN /* nomcurseur */;
```

Par exemmple,

```
DECLARE
        CURSOR emp_cur IS
                SELECT * FROM EMP;
BEGIN
        OPEN emp_cur;
        /* Utilisation du curseur */
END;
```

Lecture d'une ligne

Une fois ouvert, le curseur contient toutes les lignes du résultat de la requête On les récupère une par une en utilisant le mot-clé `FETCH` :

```
FETCH /* nom_curseur */ INTO /* liste_variables */;
```

La liste de variables peut être remplacée par une structure de type `nom_curseur%ROWTYPE`. Si la lecture de la ligne échoue, parce qu'il n'y a plus de ligne à lire, l'attribut `%NOTFOUND` prend la valeur vrai.

```
DECLARE
        CURSOR emp_cur IS
                SELECT * FROM EMP;
        ligne emp_cur%rowtype
BEGIN
        OPEN emp_cur;
        LOOP
                FETCH emp_cur INTO ligne;
                EXIT WHEN emp_cur%NOTFOUND;
                DBMS_OUTPUT.PUT_LINE(ligne.ename);
        END LOOP;
        /* ... */
END;
```

Fermeture

Après utilisation, il convient de fermer le curseur.

```
CLOSE /* nomcurseur */;
```

Complétons notre exemple,

```
DECLARE
        CURSOR emp_cur IS
                SELECT * FROM EMP;
        ligne emp_cur%rowtype;
BEGIN
        OPEN emp_cur;
        LOOP
                FETCH emp_cur INTO ligne;
                EXIT WHEN emp_cur%NOTFOUND;
                DBMS_OUTPUT.PUT_LINE(ligne.ename);
        END LOOP;
        CLOSE emp_cur;
END;
/
```

Le programme ci-dessus peut aussi s'écrire

```
DECLARE
        CURSOR emp_cur IS
                SELECT * FROM EMP;
        ligne emp_cur%rowtype;
BEGIN
        OPEN emp_cur;
        FETCH emp_cur INTO ligne;
        WHILE emp_cur%FOUND LOOP
                DBMS_OUTPUT.PUT_LINE(ligne.ename);
                FETCH emp_cur INTO ligne;
        END LOOP;
        CLOSE emp_cur;
END;
```

Boucle `FOR`

Il existe une boucle FOR se chargeant de l'ouverture, de la lecture des lignes du curseur et de sa fermeture,

```
FOR ligne IN emp_cur LOOP
        /* Traitement */
END LOOP;
```

Par exemple,

```
DECLARE
        CURSOR emp_cur IS
                SELECT * FROM EMP;
        ligne emp_cur%rowtype;
BEGIN
        FOR ligne IN emp_cur LOOP
                DBMS_OUTPUT.PUT_LINE(ligne.ename);
        END LOOP;
END;
/
```