Université Catholique de l'Afrique de l'Ouest
Faculté des Sciences de Gestion
INSTITUT SUPERIEUR DE GESTION SAINT MICHEL
Science – Foi – Action

INFORMATIQUE DE GESTION

LPIG3

PL/SQL ORACLE

Par :

Edouard Ngor SARR

Enseignant-Chercheur à UCAO-Saint Michel

Avant-propos

Nous savons tous que le SQL est un langage non procédural c'est-à-dire pas de variables ni de procédures et de fonction. Le manque fait que certains traitements complexes sont difficiles à élaborer avec le simple SQL classique. Il est impossible alors de lier plusieurs requêtes SQL et de les utiliser dans d'autres structures.

Le PL/SQL (PROCEDURAL LANGUAGE / SQL) est une extension procédurale du SQL permettant d'effectuer des traitements complexes sur une base de données Oracle. C'est un langage procédural, qui intègre des ordres SQL c'est à dire des requêtes SQL qui cohabitent avec les structures de contrôle habituelles de la programmation structurée (blocs, alternatives, boucles).

Tous les SGBD ont un langage procédural qui lui est propre :

- PL/SQL pour Oracle
- TransacSQL pour SQLServer,
- PL/pgSQL pour Postgresql
- Procédures stockées pour MySQL depuis sa version 5.0

Dans la suite de ce document nous travaillerons avec la table EMPLOYES de l'utilisateur omar.
Voici à quoi doit ressembler notre table EMPLOYES :

IDEMP	PRENOM	NOM	DATE_EMBAUCHE	SALAIRE
1	Awa	Diop	12/10/00	239000
2	Fatou	Ka	06/10/15	560000
3	Modou	Diop	13/04/08	900000
4	Aliou	Sarr	19/06/00	670000
5	Serge	Sarr	23/02/14	230000
6	Jules	Fall	06/07/11	700000
7	Doudou	Diop	12/10/15	400000
8	Fred	Tine	23/03/12	400000
9	Abdou	Lo	09/11/10	300000
10	Amie	Sow	14/10/01	600000

Avec SYS, créer donc un compte utilisateur nommé OMAR avec comme mot de passe passeomar.

```
CREATE USER omar IDENTIFIED BY omar;
GRANT connect , resource TO omar;
```

Connectez vous avec omar et créer la table suivante en y insérant les données.

```
CREATE TABLE EMPLOYES
  ( IDEMP NUMBER PRIMARY KEY,
```

```
    PRENOM VARCHAR2(55),
    NOM VARCHAR2(56),
    DATE_EMBAUCHE DATE,
    SALAIRE NUMBER
  ) ;
```

```
INSERT   INTO  EMPLOYES VALUES (0001, 'Awa','Diop','12/10/2000',239000);
INSERT   INTO  EMPLOYES VALUES (0002, 'Fatou','Ka','06/10/2015',560000);
INSERT   INTO  EMPLOYES VALUES (0003, 'Modou','Diop','13/04/2008',900000);
INSERT   INTO  EMPLOYES VALUES (0004, 'Aliou','Sarr','19/06/2000',670000);
INSERT   INTO  EMPLOYES VALUES (0005, 'Serge','Sarr','23/02/2014',230000);
INSERT   INTO  EMPLOYES VALUES (0006, 'Jules','Fall','06/07/2011',700000);
INSERT   INTO  EMPLOYES VALUES (0007, 'Doudou','Diop','12/10/2015',400000);
INSERT   INTO  EMPLOYES VALUES (0008, 'Fred','Tine','23/03/2012',400000);
INSERT   INTO  EMPLOYES VALUES (0009, 'Abdou','Lo','09/11/2010',300000);
INSERT   INTO  EMPLOYES VALUES (0010, 'Amie','Sow','14/10/2001',600000);
```

Chapitre 1:

Les bases du PL/SQL

1. Définitions :

Le PL/SQL (PROCEDURAL LANGUAGE / SQL) est une extension procédurale du SQL permettant d'effectuer des traitements complexes sur une base de données Oracle. C'est un langage procédural, qui intègre des ordres SQL c'est à dire des requêtes SQL qui cohabitent avec les structures de contrôle habituelles de la programmation structurée (blocs, alternatives, boucles).

Le PL/SQL empreinte quelques notions au SQL :

- les **DML** (SELECT, INSERT, UPDATE, DELETE, MERGE)
- la gestion de transactions **CTL** (COMMIT, ROLLBACK, SAVEPOINT) sont acceptés dans le PLSQL
- Les fonctions implicites

En PL/SQL :

- Un programme est constitué de procédures et de fonctions
- Les variables permettent l'échange d'information entre les requêtes SQL et le reste du programme

Les structures de PL/SQL sont similaires à celles des langages évolués et fournissent une méthode souple pour manipuler l'information d'une base de données.

Le PL/SQL peut être utilisé

- pour l'écriture des procédures stockées et des triggers.
- pour écrire des fonctions explicites ou utilisateurs qui peuvent être utilisées dans les requêtes SQL (en plus des fonctions prédéfinies)
- avec les interfaces graphiques d'Oracle Ex : Forms et Report

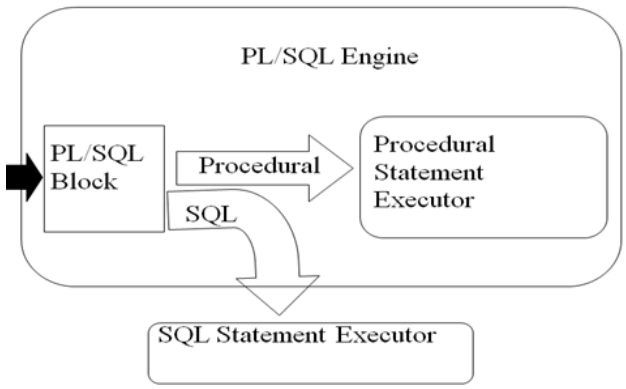
Le PL/SQL est donc une Langage à part entière comprenant

- Définition de variables, constantes, expressions, affectations
- Traitements conditionnels, répétitifs (IF, CASE, LOOP, FOR, WHILE)
- Traitement de Curseurs
- Traitement des erreurs et d'exceptions
- Etc…

Le PL/SQL offre plusieurs avantages :

- **La modularité** : Un bloc peut être nommé pour devenir une procédure ou une fonction catalogué, donc réutilisable. Une fonction ou procédure cataloguée peut être incluse dans un paquetage.
- **La portabilité :** Un programme PL/SQL est indépendant du système d'exploitation qui héberge le serveur Oracle. En changeant de système, les applicatifs n'ont pas à être modifiés.
- **L'intégration avec les données des tables** : On retrouvera avec PL/SQL tous les types de données et instructions disponibles sous SQL.

2. Architecture :



Un Code PL/SQL est un mélange entre des commandes. Une le code PL/SQL entré dans le moteur PL/SQL, il est alors segmenter vers deux entités :

- Les codes SQL sont redirigés et exécutés vers **SQL Statement Executor**
- Les structures et variables procédurales elles, sont orientées vers **Procedural Statement Executor.**

3. Structure d'un bloc PL/SQL

Un script écrit en PL/SQL sous SQLPLUS se termine obligatoirement par un slash / , sinon SQLPLUS ne l'interprète pas car le considérant comme du SQL classique. S'il contient des erreurs de compilation, il est possible d'afficher les messages d'erreur avec la commande SQL+ : *SHOW ERRORS*

Tout code écrit dans un langage procédural est formé de blocs. Chaque bloc comprend

- une section de déclaration de variables, de constantes, de cursors et d'exceptions avec DECLARE
- un ensemble d'instructions dans lequel les variables déclarées sont visibles. Avec BEGIN
- Gestion des erreurs avec EXCEPTIONS.

```
[DECLARE
                --déclarations de variables et de constantes
               -- exceptions et curseurs]
BEGIN [nom-bloc]
                --instructions SQL et extensions
[EXCEPTIONS
                 -- Traitement des exceptions]
END;  [ou END   nombloc ;]
/
```

Règle :

- *les sections DECLARE et EXCEPTION sont facultatives*
- *le BEGIN et le END sont obligatoires*
- *chaque instruction, de n'importe quelle section est terminée par un point virgule ;*
- *dans la section BEGIN*
  - *il y'a possibilité d'avoir de sous-blocs (imbrication de blocs)*
  - *possibilité de placer des commentaires*
    - *-- commentaire sur une ligne*
    - */* commentaire sur plusieurs lignes */*

- *pas de distinction entre la majuscule et la minuscule sauf dans le WHERE.*

- *Un programme PL/SQL peut être directement tapé sur une ligne de commande ou écrit dans un fichier puis chargé,*
- *aucune instruction du DDL n'est autorisée (CREATE, ALTER, DROP TABLE)*

4. LES VARIABLES

Une variable est un élément à contenu variable. Elle permet de transmettre des données entre un programme PL/SQL et la base de données. Nous pouvons les classer en deux catégories:

- LOCAL à PL/SQL
- Appartenant à l'environnement extérieur comme : SQL*FORM , PRO* ou SQL*Plus

NB : Une variable est toujours déclarée dans la parte DECLARE

Déclaration :

```
NomVARIABLE    TYPE ;
```

NB : la déclaration multiple interdite.

Exemple :

```
Var1, Var2  Type ;       Non accepté
```

Le PL/SQL utilise comme types de variables :

4.1. les types primitifs

Tous les types primitifs utilisés dans les tables du SQL

CHAR	chaine de caractères mais avec une réservation statique en mémoire. Exemple : Prenom CHAR(34) ;
VARCHAR et VARCHAR2	chaine de caractères mais avec une réservation dynamique en mémoire. Exemple : Nom Varchar2(66) ;
DATE et DATETIME	Type Date ou Date et heure Par défaut DD-MON-YY (18-DEC-02)
BOOLEEN	Type à valeur : TRUE, FALSE ou NULL
NUMBER (n,m)	Entier ou reel. Exemple : val1 number(10); Val2 Number(3,5);

4.2. % Type et %ROWTYPE

Il est possible de déclarer une variable ayant le même type :

- **qu'une autre variable avec %TYPE**

```
Variable1   TYPE ;
Variable2    Variable1%TYPE ;
```

La variable2 aura le meme type que la variable 1.

- **qu'une colonne d'une table de la base de données avec %TYPE**

```
Variable3   NomTable.NomColonne%TYPE ;
```

La variable Variable3 aura le meme type que celui de la colonne NomColonne de la table NomTable.

Exemple : déclarons une variable PRE ayant le même type que la colonne PRENOM de la table EMPLOYES.

Exemple : **PRE EMPLOYES.Prenom%TYPE ;**

Ceci nous permet de faire abstraction des problèmes d'incompatibilité de type rencontré le plus souvent dans la programmation.

- **qu'une ligne d'une table de la base (le résultat d'une requête SELECT) avec %ROWTYPE**

```
Variable4   NomTable%ROWTYPE ;
```

Exemple : mettons dans une seule variable LesClients tous les clients habitant Dakar.

Lemployes EMPLOYES%ROWTYPE ;

Règles :

- *Une variable doit être déclarée avant d'être utiliser:*
- *Un Identificateur (nom de la variable) est valide sous Oracle si et seulement si :*
  - *Il contient entre 0 et 30 caractères au plus*
  - *Il commence par une lettre*
- *Un Identificateur peut contenir des lettres, chiffres, _, $ et/ou #*
- *Une variable est seulement visible dans le bloc où elle a été déclarée ou dans les blocs imbriqués (si non redéfinie).*

Exemple :

```
DECLARE
        nom             char(15);
        salaire         number(7,2);
        embauche        DATE ;
        réponse boolean;
       prenom          Employes.Prenom%Type ;
       Ligne           Clients%ROWTYPE ;
BEGIN
END ;
/
```

5. La constante

La déclaration et l'utilisation d'une constante suit les mêmes règles que celles des variables sauf qu'avec les constantes l'initialisation est obligatoire.

Syntaxe :

```
nom_var   CONSTANT   type := valeur;
```

Exemple :

```
PIE   CONSTANT  Number :=3.14 ;
```

6. Affichage

Pour afficher à l'écran on utilise DBMS

```
DBMS_OUTPUT.PUT_LINE  ( Expression);


Dbms_output.put_line ('message .. ' ||nom_var||……);
```

Les chaines de caractères sont mises entre ' '.

Une expression peut etre:

- Une variable
- Une constante
- Une chaine de caractères
- Une ensemble de sous expression concaténées avec **||**.

Exemple :

```
DBMS_OUTPUT.PUT_LINE  ( VAR1);
DBMS_OUTPUT.PUT_LINE  ( 12);
DBMS_OUTPUT.PUT_LINE  ( 2*Val);
DBMS_OUTPUT.PUT_LINE  ('le resultat est :  ' || RES);
```

NB : L'affichage à l'écran par l'instruction DBMS_OUTPUT.PUT_LINE nécessite l'activation du 'serveur ' d'affichage par l'instruction :

```
SET  SERVEROUTPUT   ON ;
```

Permet d'activer l'affichage sur la console. Il est mis en premier ligne du code PL/SQL

Exemple 1 : écrire un code PL/SQL pour afficher BONJOUR JE SUIS M SARR

```
SET SERVEROUTPUT ON
DECLARE
      Chaine  VARCHAR2(39):= 'BONJOUR  JE SUIS M  SARR  ';
BEGIN
     DBMS_OUTPUT.PUT_LINE(Chaine);
END;
/
```

Exemple 2: Créer un programme pour calculer la racine carre de 12

```
SET SERVEROUTPUT ON
 DECLARE
        x  NUMBER:= 12;
BEGIN
       DBMS_OUTPUT.PUT_LINE(x*x);
END;
/
```

7. L'affectation

Il existe plusieurs façons pour donner une valeur (initialiser) à une variable:

- Par l'opérateur d'affectation **:=** par exemple : **n := Valeur ;**
- Par la directive **INTO** de la requête **SELECT**

7.1. Affectation avec :=

Exemple :

```
DECLARE
         nom      char(15);
         salaire  number(7,2);
BEGIN
       Nom := 'edouard' ;
       Salaire := 200000 ;
END ;
/
```

7.2. Affectation avec le INTO du SELECT

Il Permet de récupérer des valeurs dans la base de donnes et de les affecter à une ou plusieurs variables var1, var22, …

Syntaxe :

```
SELECT   expr1, expr2,…   INTO  var1, var2,…
```

Règle :

- *Le SELECT … INTO ne doit retourner qu'une seule ligne*
- *Avec PL/SQL Oracle il n'est pas possible d'inclure un SELECT sans « INTO ». Dans le cas contraire une erreur est générée.*
- *Autant de valeur que d'expression*
- *L'affectation se fait de manière respective (suivant la position) c'est-à-dire le contenu de expr1 ne pourra pas être mise dans var2.*

Exemple : Afficher les informations de l'employé de numéro 6:

```
SET SERVEROUTPUT ON
DECLARE
   PRENOM VARCHAR2(23);
   NOM  PRENOM%TYPE;
   DATE_EM VARCHAR2(23);
   SALAIRE VARCHAR2(23);
BEGIN
   SELECT   prenom, nom, Date_embauche,salaire INTO PRENOM, NOM, DATE_EM,SALAIRE
   FROM  EMPLOYES   WHERE  IDemp = 0006;
   DBMS_OUTPUT.PUT_LINE(PRENOM || '  ' || Nom );
   DBMS_OUTPUT.PUT_LINE('Embauche le ' || DATE_EM || ' avec un salaire de ' || SALAIRE );
END;
/
```

Ou bien :

```
SET SERVEROUTPUT ON
DECLARE
   PRENOM    Employes.prenom%TYPE;
   NOM       Employes.nom%TYPE;
   DATE_EM   Employes.Date_embauche%TYPE;
   SALAIRE   Employes.salaire%TYPE;
BEGIN
   SELECT   prenom, nom, Date_embauche,salaire INTO PRENOM, NOM, DATE_EM,SALAIRE
   FROM  EMPLOYES   WHERE  IDemp = 0006;
   DBMS_OUTPUT.PUT_LINE(PRENOM || '  ' || Nom );
   DBMS_OUTPUT.PUT_LINE('Embauche le ' || DATE_EM || ' avec un salaire de ' || SALAIRE );
END;
/
```

Resultat :

bloc anonyme terminé

Jules Fall

Embauche le 06/07/11 avec un salaire de 700000

8. Les traitements conditionnels if et case

8.1. l'instruction IF:

IF est le SI en Français ou en Algorithme. Il permet la gestion des conditions. Dans un IF, les instructions du ELSE ne sont exécutées que lorsqu'aucune autre condition antérieure n'est vraie. C'est le cas par defaut.

Syntaxe :

```
IF   condition   THEN
        /instructions du premier cas /
ELSIF  condition  THEN
        /instructions du deuxieme cas/
….
ELSE
        /instructions du sinon/
END IF
```

Exemple 1 : Vérifier si le salaire de l'employé 6 est supérieur à 200000

```
SET SERVEROUTPUT ON
DECLARE
  PRE  employes.prenom%TYPE;
  NM  employes.nom%TYPE;
  SAL  employes.salaire%TYPE;
BEGIN
    Select Prenom, nom, salaire   INTO   PRE, NM, SAL
    FROM   employes
    WHERE  idemp = 6;
    IF (SAL>200000) Then
      DBMS_OUTPUT.PUT_LINE('Salaire de '||pre || ' ' || nm || ' est  superieur à  200000 car egale a' || SAL );
    Else
       DBMS_OUTPUT.PUT_LINE('Salaire de '||pre || ' ' || nm || ' est  inferieur à  200000 car egale a' || SAL );
    END IF;
END;
/
```

Ou bien

```
SET SERVEROUTPUT ON
DECLARE
  LEMPLOYE  employes%ROWTYPE;
  PRE  employes.prenom%TYPE;
  NM  employes.nom%TYPE;
  SAL  employes.salaire%TYPE;
BEGIN
    Select * INTO  LEMPLOYE
    FROM   employes   WHERE  idemp = 6;
    SAL  := LEMPLOYE.Salaire;
    PRE := LEMPLOYE.Prenom;
```

```
    NM := LEMPLOYE.Nom;
    IF (SAL>200000) Then
        DBMS_OUTPUT.PUT_LINE('Salaire de '||pre || ' ' || nm || ' est  superieur à  200000 car egale a' || SAL
);
     Else
        DBMS_OUTPUT.PUT_LINE('Salaire de '||pre || ' ' || nm || ' est  inferieur à  200000 car egale a' || SAL );
   END IF;
END;
/
```

Resultat :

bloc anonyme terminé

Salaire de Jules Fall est superieur à 200000 car egale a700000

Exemple 2: Combien de temps a fait l'employé 6 dans l'entreprise. S'il à plus de 360 jours alors il est ancien dans le cas contraire c'est un nouveau.

NB : SYSDATE donne la date d'aujourd'hui. N'oubliez pas que la difference entre deux date donne un nombre de jour.

ROUND est une fonction implicite du SQL qui me permet d'arrondir une valeur numérique.

Syntaxe : ROUND (Val, i)

Val est la valeur à arrondir et i est la position.

```
SET SERVEROUTPUT  ON
DECLARE
    LEMPLOYE  employes%ROWTYPE;
    DUREE employes.DATE_EMBAUCHE%TYPE;
    NB Number;
 BEGIN
   Select * INTO LEMPLOYE
   from employes
   where idemp = 6;
   DUREE := LEMPLOYE.DATE_EMBAUCHE;
   NB :=SYSDATE - DUREE;
   NB :=ROUND(NB,0);
  If NB >360 Then
    DBMS_OUTPUT.PUT_LINE('l"employe 6 est un ANCIEN, il a fait  ' || NB || ' jours dans l entreprise' );
  Else
   DBMS_OUTPUT.PUT_LINE('l"employe 6 est un NOUVEAU, il a fait  ' || NB || ' jours dans l entreprise' );
   END IF;
END;
/
```

NB :

- Les IF imbriqués sont autorisés
- Les opérateurs utilisés dans les IF sont identiques à ceux de SQL :
  - =
  - <
  - >
  - !=
  - <=
  - >=
  - IS [NOT] NULL
  - BETWEEN
  - LIKE
  - AND
  - OR, …

Exemple 3: Modifier le prénom de l'employé 6 et nommer le Edouard Ngor avec un salaire de 500000.

```
SET SERVEROUTPUT ON
DECLARE
 LEMPLOYE employes%ROWTYPE;
 PRENOM employes.prenom%TYPE;
 NOM employes.nom%TYPE;
 SALAIRE employes.salaire%TYPE;
BEGIN
    UPDATE employes SET prenom= 'Edouard Ngor', salaire=500000
    WHERE idemp= 6;
    SELECT * INTO LEMPLOYE
    FROM employes
    WHERE idemp = 6;
    DBMS_OUTPUT.PUT_LINE('Vous etes : ' || LEMPLOYE.prenom || ' ' || LEMPLOYE.nom || ' votre
salaire est : ' || LEMPLOYE.salaire );
END;
/
```

Resultat:

bloc anonyme terminé

Vous etes : Edouard Ngor Fall votre salaire est : 500000

8.2. L'instruction CASE

C'est le selon en français. Il fait le même traitement que le IF. Nous avons deux types de syntaxe pour implémenter un CASE en PL/SQL.

Syntaxe1 :

```
CASE  Variable
    WHEN  Valeur1 THEN Instructions1
    WHEN  Valeur2 THEN Instructions2
     WHEN  Valeur3 THEN Instructions3
     ELSE InstructionsElse
END CASE;
```

Syntaxe2 :

```
Variable  :=  CASE  Valeur
      When  Valeur1  Then  Instructions1
      When  Valeur2    Then Instructions2
      When  Valeur2    Then Instructions2
       Else     InstructionsElse
END CASE;
```

Exemple 1:

```
SET SERVEROUTPUT ON
DECLARE
    Variable  Varchar2(66);
BEGIN
    CASE  ChoixPlat
         When  0   Then    Plat  :=  'Mafé'
        When   1   Then   Plat  :=  'Thiebou Djen
       Else                Plat  := 'Choix non valide'
    End;
END;
/
```

Exemple 2:

```
SET SERVEROUTPUT ON
DECLARE
    Variable  Varchar2(66);
BEGIN
     Plat  :=  CASE  ChoixPlat
              When  0  Then 'MAFE'
              When  1  Then 'Thiebou Djen
              Else     'Choix non valide'
    End;
END;
/
```

8.3. Différence entre IF et CASE.

Vu que le IF et le CASE font le même traitement alors la question est de savoir quand faut il utiliser un IF à la place d'un Case et Vis versa. Pour répondre à cette question nous allons dresser un tableau de comparaison.

	IF	CASE
Permet de gérer une condition	Oui	Oui
Plusieurs cas à gérer	Oui	Oui
Chaque cas nécessite un traitement qui lui est particulier	Pas forcement	Oui

Ainsi un CASE est utilisé à la place d'un IF si en plus d'avoir plusieurs cas à gerer, chacun de cas nécessite un traitement qui lui est particulier.

9. Les traitements répétitifs: LOOP, FOR et WHILE

Une traitement répétitif ou boucle un ensemble d'instruction qui se répète un certain nombre de fois suivant un certain nombre de conditions. En PL/SQL nous avons trois types de boucle :

- Le LOOP
- Le FOR
- Le WHILE

9.1. le traitement LOOP

Le traitement LOOP ... END LOOP permet d'implémenter une boucle.

Syntaxe :

```
LOOP
      Instructions exécutables ;
    EXIT [WHEN Condition] ;
  Instructions executables ;
END LOOP ;
```

NB : Obligation d'utiliser la commande EXIT pour éviter une boucle infinie, facultativement quand une condition est vraie.

Exemple : Soit la table COMPTEUR

```
CREATE TABLE Compteur (val number(5)) ;
```

Ecrire un programme qui insere 10 lignes dans la table COMPTEUR.

```
DECLARE
      nbre number(2) := 1 ;
BEGIN
     LOOP
           Insert into Compteur values (nbre);
           nbre:= nbre +1;
           exit when nbre > 10 ;
      END LOOP;
END ;
/
```

9.2. le traitement FOR

C'est le POUR en algorithmique. Permet de repeter une ou plusieurs instruction autant de fois que la differece entre deux valeurs.

Syntaxe :

```
FOR Variable IN inferieur.. Superieur LOOP
  /Instructions
END LOOP ;
```

La variable de boucle prend successivement les valeurs inferieur, inferieur +1, inferieur +2, …, jusqu'à la valeur Superieur. On pourra également utiliser un curseur dans la clause IN

NB :

- En ajoutant un REVERSE à notre boucle FOR, nous faisons un décrémentation.
- Les .. sont obligatoires

Syntaxe :

```
FOR Variable  IN   REVERSE  inferieur.. Superieur LOOP
  /Instructions
END LOOP ;
```

La variable de boucle prend successivement les valeurs Superieur , Superieur - 1, Superieur -2, …, jusqu'à la valeur Inferieur.

Exemple 1: insérons de 1 à 10 dans notre table Compteur.

```
DECLARE
        nbre number(2) := 1 ;
BEGIN
       FOR  nbre  IN 1.. 10   LOOP
                Insert  into Compteur values (nbre);
       END LOOP;
END ;
/
```

Exemple 2: insérons de 10 à 1 dans notre table Compteur.

```
DECLARE
nbre number(2) := 1 ;
BEGIN
       FOR  nbre  IN  REVERSE 1.. 10   LOOP
            Insert  into Compteur values (nbre);
       END LOOP;
END ;
/
```

Exemple 3: Ecrire un programme pour calculer le factoriel de 5

```
SET SERVEROUTPUT ON
DECLARE
          Fact  number :=1 ;
          x number:=5;
BEGIN
          FOR i   IN 1..x   LOOP
                          Fact := Fact  * i;
         END LOOP;
        DBMS_OUTPUT.PUT_LINE('Le factoriel de ' ||x || ' est :  ' || Fact );
END;
/
```

Resultat:

bloc anonyme terminé

Le factoriel de 5 est : 120

9.3. le traitement WHILE

La boucle WHILE permet de répéter une ou plusieurs Instructions tant que la condition est vraie.

Syntaxe :

```
WHILE   condition   LOOP
        /i n s t r u c t i o n s/
END LOOP;
```

Exemple 3: Ecrire un programme pour calculer le factoriel de 5

```
SET  SERVEROUTPUT ON
DECLARE
          Fact  number :=1 ;
  i number:=1;
  X  number:=5;
BEGIN
      WHILE   (i<= x)  LOOP
         Fact := Fact  * i;
         i:=i+1;
      END LOOP;
      DBMS_OUTPUT.PUT_LINE('Le factoriel de ' ||x || ' est :  ' || Fact );
END;
/
```

Resultat*:*

bloc anonyme terminé

Le factoriel de 5 est : 120

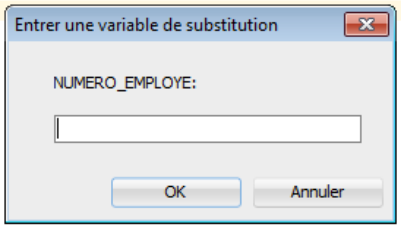
10. Les variables de substitution :

Il est possible de passer en paramètres d'entrée d'un bloc PL/SQL des variables définies sous SQL*PLUS. On accède aux valeurs d'une telle variable dans le code PL/SQL en faisant préfixer le nom de la variable par un « & ».

Exemple en SQL : Afficher les informations d'un employé

```
SELECT  *   FROM  employes
WHERE   idemp=&Numero_Employe ;
```

Le SGBD va nous demander de donner une valeur pour Numero_Employe.



Il est même possible de faire plusieurs variables de substitution.

Exemple :

```
SELECT   &col1, &Col2   FROM   &Table
where &col1= &val ;
```

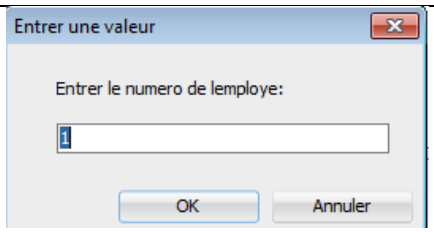
EN PL/SQL la directive **ACCEPT** permet de faire une lecture de la variable au clavier. Avant l'exécution du code PL/SQL, le système demande en premier lieu de donner un valeur pour la variable.

Syntaxe :

```
ACCEPT Nom_Variable   PROMPT  'Texte à afficher '
```

Exemple : Afficher le prénom et nom d'un employé en fonction du numéro donnée.

```
SET SERVEROUTPUT ON
ACCEPT   IdEMploye   PROMPT 'Entrer le numero de l'employe: '
DECLARE
        pre employes.prenom%TYPE;
       nm employes.nom%TYPE;
       sal employes.salaire%TYPE;
BEGIN
    SELECt prenom, nom, salaire INTO pre , nm, sal   FROM employes  WHERE  idemp = &IdEMploye;
    DBMS_OUTPUT.PUT_LINE('PRENOM : '||pre);
    DBMS_OUTPUT.PUT_LINE('NOM : '||nm);
    DBMS_OUTPUT.PUT_LINE('SALAIRE : '||sal);
END;
/
```



Résultat :

bloc anonyme terminé

PRENOM : Awa

NOM : Diop

SALAIRE : 239000

Chapitre 2:

Les curseurs

Les instructions de type SELECT ... INTO ... du PLSQL ne fonctionnent que sur des requêtes retournant une et une seule valeur. Elles sont Mono lignes. Alors, ne serait-il pas plus intéressant de pouvoir placer dans des variables le résultat d'une requête retournant plusieurs lignes : requête multi-ligne.

Un Curseur est une zone de mémoire de taille fixe, utilisée par le noyau d'Oracle pour analyser et interpréter tout ordre SQL. Les statuts d'exécution de l'ordre se trouvent dans le curseur.

Nous avons :

- Les curseurs implicites : créés et gérés par Oracle à chaque ordre SQL
- Les curseurs explicites :créés et gérés par l'utilisateur afin de pouvoir traiter un SELECT qui retourne plusieurs lignes

En PL/SQL, nous étudions les cursors explicites c'est-à-dire un objet contenant le résultat d'une requête avec aucune, une ou plusieurs lignes.

L'utilisation d'un curseur explicite nécessite 4 étapes :

- la déclaration du curseur
- l'ouverture du curseur
- le traitement des lignes
- la fermeture du curseur

1. Les attributs des CURSEURS

Tous les curseurs ont des attributs :

%ROWCOUNT	retourne le nombre de lignes traitées par le curseur
%FOUND	retourne vrai si au moins une ligne a été traitée par la requête ou la dernière ligne.
%NOTFOUND	vrai si aucune ligne n'a été traitée par la requête ou la dernière ligne.
%ISOPEN	vrai si le curseur est ouvert (utile seulement pour les curseurs explicites)

2. La déclaration

Un curseur se déclare dans une section DECLARE avec le mot CURSOR suivit d'un IS:

Syntaxe :

```
CURSOR NomCursor IS REQUETE_SELECT ;
```

Exemple, si on tient à récupérer tous les employés de la table EMPLOYES, on déclarera le curseur suivant.

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

3. L'ouverture

Lors de l'ouverture d'un curseur, la requête du curseur est exécutée, et le curseur contient toutes les données retournées par cette requête. Ainsi, toute manipulation se fera désormais via le cursors. On ouvre un curseur dans une section BEGIN.

Syntaxe :

```
OPEN   NomCursor ;
```

Par exemple :

```
OPEN emp_cur ;
```

4. Fermeture du curseur

Après le traitement des lignes, pour libérer la place mémoire.

Syntaxe :

```
CLOSE    nom_curseur ;
```

5. Le Traitement du cursor

Après l'exécution du SELECT, les lignes ramenées sont traitées une par une, la valeur de chaque colonne du SELECT doit être stockée dans une variable réceptrice.

a. On utilise FETCH pour ramener une seule ligne

La liste de variables peut être remplacée par une structure de type nom curseur%ROWTYPE que nous avons vue précédemment. Si la lecture de la ligne échoue, parce qu'il n'y a plus de ligne à lire, l'attribut %NOTFOUND prend la valeur vraie. Dans le cas contraire elle est à fausse. %NOTFOUND nous permettra donc de gérer le resultat null d'une requête.

Syntaxe :

```
FETCH    nom_curseur    INTO   liste_variables ;
```

b. pour traiter n lignes : LOOP et FOR

Pour récupérer n lignes du curseur, il faut prévoir une boucle LOOP, FOR ou WHILE

- **Avec LOOP.**

```
SET SERVEROUTPUT ON
DECLARE
    CURSOR emp_cur   IS    SELECT * FROM EMPLOYES ;
    ligne   emp_cur%ROWTYPE;
BEGIN
   OPEN emp_cur ;
    LOOP
        FETCH emp_cur INTO ligne ;
        EXIT WHEN emp_cur%NOTFOUND ;
       DBMS_OUTPUT . PUT_LINE ( ligne.idemp  ||' ' || ligne.prenom ||' '|| ligne.nom ||' '|| ligne.salaire   ) ;
   END LOOP ;
 CLOSE emp_cur;
END;
```

```
Résultat :
bloc anonyme terminé
1 Awa Diop 239000
2 Fatou Ka 560000
3 Modou Diop 900000
4 Aliou Sarr 670000
5 Serge Sarr 230000
6 Edouard  Ngor Fall 500000
7 Doudou Diop 400000
8 Fred Tine 400000
9 Abdou Lo 300000
10 Amie Sow 600000
```

Le EXIT WHEN emp_cur%NOTFOUND ; permet de sortir dés la première ligne vide rencontrée dans le cursor c'est-à-dire pas de contenue.

- **AVEC FOR**

L'utilisation de la boucle FOR permet en même temps de gérer l'ouverture, la lecture des lignes du curseur et la fermeture.

Syntaxe :

```
FOR    ligne   IN    emp_cur    LOOP
          Traitements
END LOOP ;
```

Par exemple,

```
SET SERVEROUTPUT ON
DECLARE
      CURSOR emp_cur IS   SELECT * FROM EMPLOYES ;
      ligne     emp_cur%rowtype ;
BEGIN
      FOR ligne IN emp_cur LOOP
          DBMS_OUTPUT . PUT_LINE ( ligne.idemp  ||' ' || ligne.prenom ||' '|| ligne.nom ||' '|| ligne.salaire   ) ;
      END LOOP ;
END;
/
```

NB: La requête d'un curseur ne peut pas contenir de variables de substitution c'est-à-dire dont les valeurs ne sont passées parce que les valeurs des ces variables sont susceptibles de changer entre la déclaration du curseur et son ouverture.

6. curseur paramétré.

Un curseur paramétré est un curseur dont la requête SELECT contient des variables de substitution. Pour déclarer un tel curseur on précise la liste des noms et des types des paramètres entre parenthèses après le nom du curseur :

```
CURSOR nom _curseur ( liste des paramètres ) IS Requête_SELECT ;
```

Exemple : Soit la table employés

IDEMP	PRENOM	NOM	DATE_EMBAUCHE	SALAIRE
1	Awa	Diop	12/10/00	239000
2	Fatou	Ka	06/10/15	560000
3	Modou	Diop	13/04/08	900000
4	Aliou	Sarr	19/06/00	670000
5	Serge	Sarr	23/02/14	230000
6	Edouard Ngor	Fall	06/07/11	500000
7	Doudou	Diop	12/10/15	400000
8	Fred	Tine	23/03/12	400000
9	Abdou	Lo	09/11/10	300000
10	Amie	Sow	14/10/01	600000

- Ecrire un programme qui liste les employés portant un nom donné.
  On affectera la valeur du nom en paramètre par exemple DIOP

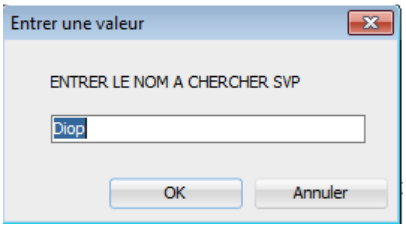
```
SET SERVEROUTPUT ON
DECLARE
    CURSOR Cur_emplNom ( LeNom  Varchar) IS SELECT  * FROM EMPLOYES WHERE nom =
LeNom  ;
BEGIN
    FOR ligne IN Cur_emplNom ( 'Diop') LOOP
        DBMS_OUTPUT . PUT_LINE ( ligne.idemp  ||' ' || ligne.prenom ||' '|| ligne.nom ||' '|| ligne.salaire   ) ;
    END LOOP ;
END;
/
```

```
bloc anonyme terminé
bloc anonyme terminé
1 Awa Diop 239000
3 Modou Diop 900000
7 Doudou Diop 400000
```

NB : Nous pouvons aller plus loin en utilisant les variables de substitution de telle sorte que le système nous demande de donner la valeur avant l'entame de l'exécution du programme. Pour ca on utilise ACCEPT et PROMP.

Exemple : Nous allons refaire le même exercice que précédemment.

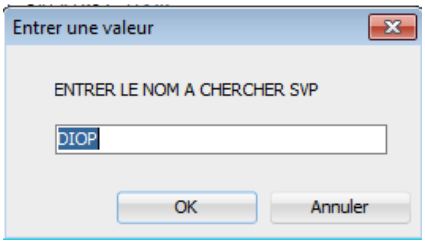
```
SET SERVEROUTPUT ON
ACCEPT  NOM PROMPT 'ENTRER LE NOM A CHERCHER SVP'
DECLARE
      CURSOR Cur_emplNom ( LeNom  Varchar) IS SELECT  * FROM EMPLOYES WHERE nom =
LeNom  ;
BEGIN
      FOR ligne IN Cur_emplNom ('&NOM') LOOP
         DBMS_OUTPUT . PUT_LINE ( ligne.idemp  ||' ' || ligne.prenom ||' '|| ligne.nom ||' '|| ligne.salaire   ) ;
      END LOOP ;
END;
/
```

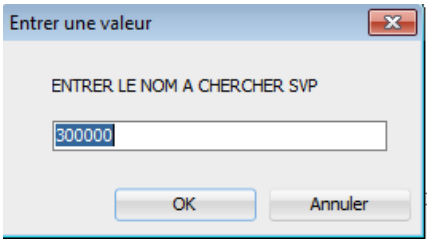



```
1 Awa Diop 239000
3 Modou Diop 900000
7 Doudou Diop 400000
```

Exemple : Ecrire un programme qui liste les employés ayant le même nom donné et ayant un salaire supérieur a une valeur donnée. Nous testerons avec le nom Diop et le salaire 300000

```
SET SERVEROUTPUT ON
ACCEPT  NOM PROMPT 'ENTRER LE NOM A CHERCHER SVP'
ACCEPT  SALAIRE PROMPT 'ENTRER LE NOM A CHERCHER SVP'
DECLARE
      CURSOR Cur_emplNom ( LeNom  Varchar,LeSal Number) IS SELECT  * FROM EMPLOYES
WHERE nom = LeNom AND salaire > LeSal  ;
BEGIN
      FOR ligne IN Cur_emplNom ('&NOM',&SALAIRE) LOOP
         DBMS_OUTPUT . PUT_LINE ( ligne.idemp  ||' ' || ligne.prenom ||' '|| ligne.nom ||' '|| ligne.salaire   ) ;
      END LOOP ;
END;
/
```





bloc anonyme terminé

3 Modou Diop 900000

7 Doudou Diop 400000

7. Exercice Pratique sur les cursors :

Exercice 1 :

Soit la table EMPLOYES_COPY avec la même structure que EMPLOYES

```
CREATE TABLE  EMPLOYES_COPY   AS   SELECT  *  FROM EMPLOYES WHERE 1=2;
```

Copier le contenu de la table EMPLOYES dans EMPLOYES_COPY et Afficher la nouvelle table

```
SET SERVEROUTPUT ON
DECLARE
      CURSOR Cur_empl IS SELECT  * FROM EMPLOYES;
      CURSOR Cur_emplCopy IS SELECT  * FROM EMPLOYES_COPY;
BEGIN
      FOR ligne IN Cur_empl LOOP
          INSERT  INTO EMPLOYES_COPY VALUES (ligne.idemp,ligne.prenom,
ligne.nom,ligne.date_embauche, ligne.salaire);
      END LOOP ;
      FOR lignecopy IN Cur_emplCopy LOOP
          DBMS_OUTPUT.PUT_LINE ( ligneCopy.idemp  ||' ' || ligneCopy.prenom ||' '|| ligneCopy.nom ||' '||
ligneCopy.salaire   ) ;
      END LOOP ;
END;
/
```

I...	PRENOM	NOM	DATE_EMBAUCHE	SALAIRE
1	Awa	Diop	12/10/00	239000
2	Fatou	Ka	06/10/15	560000
3	Modou	Diop	13/04/08	900000
4	Aliou	Sarr	19/06/00	670000
5	Serge	Sarr	23/02/14	230000
6	Edouard Ngor	Fall	06/07/11	500000
7	Doudou	Diop	12/10/15	400000
8	Fred	Tine	23/03/12	400000
9	Abdou	Lo	09/11/10	300000
10	Amie	Sow	14/10/01	600000

Exercice 2:

Dans un rapport demandé par le DRH, doit figurer la liste de tous les employés de la société, les deux employés les plus payés de l'entreprise, et les trois les moins payes , les chefs de département et les deux les plus anciens.

Dans la base nous avons :

I...	PRENOM	NOM	DATE_EMBAUCHE	SALAIRE
1	Awa	Diop	12/10/00	239000
2	Fatou	Ka	06/10/15	560000
3	Modou	Diop	13/04/08	900000
4	Aliou	Sarr	19/06/00	670000
5	Serge	Sarr	23/02/14	230000
6	Edouard Ngor	Fall	06/07/11	500000
7	Doudou	Diop	12/10/15	400000
8	Fred	Tine	23/03/12	400000
9	Abdou	Lo	09/11/10	300000
10	Amie	Sow	14/10/01	600000

```
SET SERVEROUTPUT ON
DECLARE
    CURSOR emp_cur   IS    SELECT * FROM EMPLOYES ;
    ligne   emp_cur%ROWTYPE;
    CURSOR CPLUSPAYE   IS select * from EMPLOYES  order by salaire DESC;
    lignePlus   CPLUSPAYE%ROWTYPE;
    CURSOR CMOINSPAYE   IS select * from EMPLOYES  order by salaire ASC;
    ligneMoins   CMOINSPAYE%ROWTYPE;
    x number;
    CURSOR CANCIENS   IS select * from EMPLOYES  order by date_embauche ASC;
    ligneAnciens   CANCIENS%ROWTYPE;
BEGIN
    DBMS_OUTPUT . PUT_LINE('--------------------------------------------------------------------------------');
    DBMS_OUTPUT . PUT_LINE('LISTE DE TOUS LES EMPLOYES');
    FOR ligne In emp_cur LOOP
        DBMS_OUTPUT . PUT_LINE ( ligne.idemp  ||' ' || ligne.prenom ||' '|| ligne.nom ||' '|| ligne.salaire   ) ;
   END LOOP ;
  DBMS_OUTPUT . PUT_LINE('--------------------------------------------------------------------------------');
  DBMS_OUTPUT . PUT_LINE('LISTE DES 2 EMPLOYES LES  PLYS PAYES ');
   x:=1;
    Open CPLUSPAYE;
    LOOP
      FETCH  CPLUSPAYE  INTO lignePlus;
      EXIT WHEN x>2;
      DBMS_OUTPUT . PUT_LINE ( lignePlus.idemp  ||' ' || lignePlus.prenom ||' '|| lignePlus.nom ||' '||
```

```
lignePlus.salaire   ) ;
        x:= x+1 ;
     END LOOP;
     close CPLUSPAYE;

        DBMS_OUTPUT . PUT_LINE('--------------------------------------------------------------------------------');
   DBMS_OUTPUT . PUT_LINE('LISTE DES 3 EMPLOYES LES MOYENS  PAYES');
    x:=1;
     Open CMOINSPAYE;
     LOOP
       FETCH  CMOINSPAYE  INTO lignemoins;
       EXIT WHEN x>3;
       DBMS_OUTPUT . PUT_LINE ( lignemoins.idemp  ||' ' || lignemoins.prenom ||' '|| lignemoins.nom ||' '||
lignemoins.salaire   ) ;
        x:= x+1 ;
     END LOOP;
     close CMOINSPAYE;
   DBMS_OUTPUT . PUT_LINE('--------------------------------------------------------------------------------');
   DBMS_OUTPUT . PUT_LINE('LES 2 EMPLOYES LES PLUS ANCIENS');
    x:=1;
     Open CANCIENS;
     LOOP
       FETCH  CANCIENS  INTO ligneAnciens;
       EXIT WHEN x>2;
       DBMS_OUTPUT . PUT_LINE ( ligneAnciens.idemp  ||' ' || ligneAnciens.prenom ||' '|| ligneAnciens.nom
||' '|| ligneAnciens.date_embauche  ) ;
        x:= x+1 ;
     END LOOP;
     close CANCIENS;
END;
/
```

```
bloc anonyme terminé
--------------------------------------------------------------------------------
LISTE DE TOUS LES EMPLOYES
1 Awa Diop 239000
2 Fatou Ka 560000
3 Modou Diop 900000
4 Aliou Sarr 670000
```

```
5 Serge Sarr 230000
6 Edouard  Ngor Fall 500000
7 Doudou Diop 400000
8 Fred Tine 400000
9 Abdou Lo 300000
10 Amie Sow 600000
---------------------------------------------------------------------
LISTE DES 2 EMPLOYES LES  PLYS PAYES
3 Modou Diop 900000
4 Aliou Sarr 670000
---------------------------------------------------------------------
LISTE DES 3 EMPLOYES LES MOYENS  PAYES
5 Serge Sarr 230000
1 Awa Diop 239000
9 Abdou Lo 300000
---------------------------------------------------------------------
LES 2 EMPLOYES LES PLUS ANCIENS
4 Aliou Sarr 19/06/00
    1    Awa Diop 12/10/00
```

8. Les tableaux

Un tableau est variable pouvant contenir une ou plusieurs valeurs de même type.

Déclaration

La déclaration d'ue variable se fait en deux etapes :

Déclaration du type de variable

Les tableaux associatifs doivent d'abord être déclarés comme un type particulier de variable dans la partie DECLARE.

- **TABLE** : est un tableau dynamique c'est-à-dire declaré sans taille fixe
- **VARRAY** : est un tableau dont on déclare à l'avance la longueur.

```
TYPE nomType  IS TABLE OF type_des_valeurs INDEX BY type_des_clés
```

Ou

```
TYPE nom IS VARRAY(longueur) OF type_des_valeurs
```

- **nomType** est un nom de variable donné. Il sera utilisé comme type du tableau à créer
- **type_des_valeurs** est un type de variable Oracle : NUMBER ou VARCHAR(longueur), par exemple. Ce sont les valeurs que l'on va récupérer de la base : on peut donc utiliser aussi la syntaxe colonne%TYPE.
- **type_des_clés** est BINARY_INTEGER ou VARCHAR(10) par exemple. Ce sont les clés du tableau associatif, elles doivent être uniques. Les types admis doivent être
  - BINARY_INTEGER
  - PLS_INTEGER
  - VARCHAR(longueur),
  - STRING,
  - LONG
  - VARCHAR2(longueur).
- **longueur** est le dernier indice du VARRAY, puisque ces tableaux sont indicés à partir de 1 (pas comme en perl !)Exemple : Déclarer un tableau de salaires déclarés comme NUMBER(10)

```
DECLARE
   TYPE  Typesalaire  IS TABLE  OF   NUMBER   INDEX   BY   BINARY_INTEGER;
   TYPE  Typeprenom  IS VARRAY (10) OF   VARCHAR2(66)   INDEX   BY   BINARY_INTEGER;
 TYPE  TypeAdresse  IS VARRAY (5) OF   employes.adresse%TYPE   INDEX  BY
BINARY_INTEGER;
```

Instanciation du tableau

Une fois le type déclaré, il faut encore déclarer une variable de ce type. C'est tout simple

```
Nom Type ;
```

- Nom est le nom de notre tableau.
- Type est le type qui vient d'être déclaré.

Exemple :

```
DECLARE
   TYPE  Typesalaire  IS TABLE  OF   NUMBER   INDEX   BY   BINARY_INTEGER;
    Tsalaire  Typesalaire  ;
```

Cette deuxième ligne instancie un tableau associatif (encore vide) nommé **Tsalaire**, dont les valeurs et les clés seront des types **Typesalaire** définis à la ligne au-dessus.
Rien n'empêche d'instancier plusieurs tableaux du même type.

Affectation

Se fait avec := ou INTO du SELECT.

```
Tsalaire (1) := 200000;
```

Récupération

```
SAL : = Tsalaire (1);
```

Parcourir un tableau associatif

- **FIRST et LAST**

Un tableau associatif (comme toutes les autres collections en PL/SQL) a les propriétés FIRST et LAST auxquelles on accède en faisant nom.FIRST et nom.LAST. FIRST renvoie la première clé issue d'un tri numérique ou alphabétique. Pour parcourir le tableau, on peut donc faire une boucle comme ceci :

```
FOR i IN nom.FIRST..nom.LAST LOOP
…
END LOOP;
```

Exemple:

```
FOR i IN Tsalaire.FIRST .. Tsalaire.LAST LOOP
…..
END LOOP;
```

- **PRIOR et NEXT**

NEXT fait reference à la case suivante de notre tableau.
PRIOR à celle précédente.

```
i := nom.FIRST;
WHILE i IS NOT NULL LOOP
...
  i := nom.NEXT(i);
END LOOP;
```

Cette boucle s'arrte au dernier élément parce que NEXT(i) renvoie NULL au dernier i.

Exemple : remplir le salaire des employés dans un tableau.

```
SET SERVEROUTPUT ON

DECLARE

        TYPE TYPE_TABSALAIRE IS TABLE OF NUMBER INDEX    BY   BINARY_INTEGER ;

   TSALAIRE    TYPE_TABSALAIRE;

   CURSOR CUREMPL IS SELECT * FROM EMPLOYES;

   Ligne employes%ROWTYPE;

   i NUMBER;

   max NUMBER;

   min NUMBER;

BEGIN

      i:=0;

      FOR ligne IN CUREMPL LOOP

        TSALAIRE(i):=ligne.salaire;

        i:= i+1;

      END LOOP ;

      FOR i IN TSALAIRE.FIRST  .. TSALAIRE.LAST  LOOP

                    DBMS_OUTPUT.PUT_LINE(TSALAIRE(i));

      END LOOP;

END;
```

Chapitre 3:

Les exceptions

1. Définitions

Le PL/SQL permet de définir dans une zone particulière (de gestion d'exception), l'attitude que le programme doit avoir lorsque certaines erreurs définies ou prédéfinies se produisent. Une erreur survenue lors de l'exécution du code déclenche ce que l'on nomme une exception. Le code erreur associé est transmis à la section EXCEPTION, pour vous laisser la possibilité de la gérer et donc de ne pas mettre fin prématurément à l'application. Une exception est alors une erreur qui survient durant l'exécution d'une ligne de code d'un programme. Le traitement des exceptions PL/SQL est un mécanisme pour manipuler les erreurs rencontrées lors de l'exécution d'un programme.

Il y a 2 types d'exception :

- **Les exceptions prédéfinies par Oracle** : construites et gérées par la moteur Oracle.
- **Les exceptions utilisateurs :** implémentées par développeur.

La plus part des langages de programme implémente des procédures de gestion des erreurs. Le principe est simple et nous permettra :

- D'identifier l'erreur,
- De rattraper l'erreur en cours d'exécution,
- D'écrire le traitement pour l'erreur.

2. les exceptions prédéfinies

Lors de l'exécution d'un programme PLSQL ou d'une simple requête SQL, nous rencontrons souvent des messages d'erreur jusque là incompris. Il vous est arrivé de rencontrer une erreur du genre :

- DUP_VAL_ON_INDEX
  - Lorsqu'une instruction SQL tente de créer une valeur dupliquée dans une colonne sur laquelle un index unique a été défini
- INVALID_NUMBER
  - Lorsqu'une instruction SQL spécifie un nombre invalide
- NO_DATA_FOUND
  - Lorsqu'une instruction Select ne retourne aucune ligne
- TOO_MANY_ROWS
  - Une instruction Select ne peut pas renvoyer plus d'une ligne sans provoquer l'exception TOO_MANY_ROWS
- VALUE_ERROR
  - Provoquée dans des situations d'erreur résultant de valeurs tronquées ou converties

En PL/SQL, une exception ne provoque pas nécessairement l'arrêt du programme. Elle peut être saisie et gérer dans un bloc du code plus précisément dans la partie « EXCEPTIO. Par contre, une exception non saisie remonte peut bloquer l'exécution du code dans le quel il opère et peut même remonter dans la procédure appelante.

3. syntaxe générale

Voici la Syntaxe d'un code PL/SQL où la partie exception est implémentée.

```
DECLARE
        ….
BEGIN
        …
EXCEPTION
      WHEN Erreur1 THEN
            . . . instructions pour erreur 1
      WHEN Erreur2 THEN
             . . . instructions pour erreur 2
      WHEN Erreur3 THEN
             . . . instructions pour erreur 3
END;
```

NB : Une exception :

- ➔ est toujours **typée** c'est-à-dire à un type bien défini
- ➔ porte un nom comme une variable
- ➔ porte un numéro erreur oracle commençant par ORA-
- ➔ porte un identifiant nommé SQLCODE avec 5 chiffres

***Fonctionnement** : Lorsque dans les instructions, l'erreur se produit, le code du BEGIN s'interrompt et le code de la section EXCEPTION est lancé. On dit qu'une exception est levée (raised).*

Exemple 1: ZERO DIVIDE

- **Cas 1 : faisons la division entre 20 et 2 (20 /2)**

```
SET SERVEROUTPUT ON
DECLARE
    x NUMBER:=20;
    y NUMBER:=2;
    Res NUMBER(6,2);
BEGIN
    Res:=x/y;
    DBMS_OUTPUT.PUT_LINE ( 'la division est egale à ' || Res ) ;
END;
/
```

bloc anonyme terminé
la division est egale à 10

- **Cas 2 : faisons la division entre 20 et 0 (20 /0) non gérer**

```
SET SERVEROUTPUT ON
DECLARE
    x NUMBER:=20;
    y NUMBER:=0;
    Res NUMBER(6,2);
BEGIN
    Res:=x/y;
    DBMS_OUTPUT.PUT_LINE ( 'la division est egale à ' || Res ) ;
END;
/
```

Rapport d'erreur -
ORA-01476: le diviseur est égal à zéro
ORA-06512: à ligne 6
01476. 00000 - "divisor is equal to zero"

Une erreur de type …. Avec un code oracle ***ORA-06512*** *et un CODE* ***01476*** *est levée mais non gérée. Donc l'exécution est bloquée à partir de RES *=X/Y ;*

- **Cas 3 : faisons la division entre 20 et 0 (20 /0) avec gestion de l'erreur**

```
SET SERVEROUTPUT ON
DECLARE
    x NUMBER:=20;
    y NUMBER:=0;
    Res NUMBER(6,2);
BEGIN
     Res:=x/y;
     DBMS_OUTPUT.PUT_LINE ( 'la division est egale à ' || Res ) ;
     EXCEPTION
          WHEN   ZERO_DIVIDE  THEN
               DBMS_OUTPUT.PUT_LINE ( 'impossible de diviser un nombre par  0 ') ;
END;
/
```

bloc anonyme terminé
impossible de diviser un nombre par 0

Ici Nous avons géré l'erreur en lui affectant un comportement.

Exemple 2 : NA DATA_FOUND

Soit la table employés :

IDEMP	PRENOM	NOM	DATE_EMBAUCHE	SALAIRE
1	Awa	Diop	12/10/00	239000
2	Fatou	Ka	06/10/15	560000
3	Modou	Diop	13/04/08	900000
4	Aliou	Sarr	19/06/00	670000
5	Serge	Sarr	23/02/14	230000
6	Edouard Ngor	Fall	06/07/11	500000
7	Doudou	Diop	12/10/15	400000
8	Fred	Tine	23/03/12	400000
9	Abdou	Lo	09/11/10	300000
10	Amie	Sow	14/10/01	600000

Vous avez vu que nous n'avons pas un employé avec comme identifiant 20.

- Ecrire un script PL/SQL pour lister les information de l'employé 20 (Sans gestion de l'erreur NO DATA FOUND)

```
SET SERVEROUTPUT ON
DECLARE
  Lignes employes%ROWTYPE;
BEGIN
    SELECT * INTO Lignes
    FROM EMPLOYES
    WHERE idemp = 20;
    DBMS_OUTPUT.PUT_LINE ( lignes.prenom || ' ' || lignes.nom ) ;
END;
/
```

```
Rapport d'erreur -
ORA-01403: aucune donnée trouvée
ORA-06512: à ligne 4
01403. 00000 -  "no data found"
```

Une erreur nommée NO DATE FOUND avec comme SQLCODE 01403 et avec comme numero oracle ORA-06512 est levée. Cela signifie qu'aucune donnée n'a été trouvée.

- **Avec gestion de l'erreur**

```
SET SERVEROUTPUT ON
DECLARE
     Lignes employes%ROWTYPE;
BEGIN
    SELECT * INTO Lignes   FROM EMPLOYES   WHERE idemp = 20;
    DBMS_OUTPUT.PUT_LINE ( lignes.prenom || '  ' || lignes.nom ) ;
 EXCEPTION
```

```
        WHEN  NO_DATA_FOUND   THEN
        DBMS_OUTPUT.PUT_LINE ( ' il n y a pas de resultat pour cette requete dans la base ') ;
END;
/
```

```
bloc anonyme terminé
il n y a pas de resultat pour cette requete dans la base
```

4. Gestion multiple

Il est possible de faire une gestion multiple des exceptions. C'est-à-dire gerer plusieurs erreurs en dans le même EXCEPTION.

Exemple : Soit le code PL/SQL précédent où nous cherchions l'employé de matricule 20. Nous voulons maintenant gérer le cas où :

a. On ne trouve pas de valeur
b. On trouve plusieurs valeurs
c. Une erreur inconnue survient

```
DECLARE
  ligne employes%ROWTYPE;
BEGIN
  SELECT *  INTO ligne
  FROM EMPLOYES
  WHERE idemp = 20 ;
  DBMS_OUTPUT . PUT_LINE ( ligne.prenom  || '  ' || ligne.nom) ;
  EXCEPTION
      WHEN NO_DATA_FOUND THEN
          DBMS_OUTPUT . PUT_LINE ( 'Aucun resultat') ;
       WHEN TOO_MANY_ROWS THEN
          DBMS_OUTPUT . PUT_LINE ( ' des resultats trouvées ') ;
      WHEN OTHERS THEN
          DBMS_OUTPUT . PUT_LINE ( 'autres ') ;
END;
/
```

```
bloc anonyme terminé
Aucun resultat
```

5. Test d'exécution avec SQLCODE et SQLERRM

- **SQLCODE**

Fonction prédéfinie qui renvoie le statut d'erreur système de l'instruction qui vient d'être exécutée (si sans erreur, SQLCODE = 0). C'est un numéro généré par oracle pour chaque type d'erreur.

- **SQLERRM**

Fonction prédéfinie qui renvoie le message d'erreur associé à la valeur retournée par SQLCODE (si sans erreur, SQLERRM = 'ORA-0000').

Exemple :

```
Declare
 lignes          employes%ROWTYPE;
Begin
        dbms_output.enable;
        select * into lignes   from  employes;
Exception
        when OTHERS then
                dbms_output.put_line('SQLCODE:'|| to_char(SQLCODE));
                dbms_output.put_line(SQLERRM);
End;
/
```

SQLCODE: -1422

ORA-01422: l'extraction exacte ramène plus que le nombre de lignes demandé

Procédure PL/SQL terminée avec succès.

6. Gestion avec le SQLCODE

Certaines exceptions n'ont pas de nom. Elles ont seulement un code d'erreur appelé SQLCODE.

Voici un tableau avec quelque SQLCODE

Exception prédéfinie	Erreur Oracle	Valeur de SQLCODE
ACCESS_INTO_NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Ces genres d'exception on les traite de la façon suivante :

```
EXCEPTION
        WHEN OTHERS THEN
            IF    SQLCODE = CODE1 THEN
                        Traitement1
            ELSIF  SQLCODE = CODE2 THEN
                        Traitement2
            ELSE
                    DBMS_OUTPUT . PUT_LINE ( ' erreur inconnue. ' ) ;
END;
```

Exemple : pour NO DATA FOUND le SQLCODE est de 100

Faisons maintenant la gestion de cette erreur :

```
DECLARE
  ligne employes%ROWTYPE;
BEGIN
  SELECT *  INTO ligne
  FROM EMPLOYES
  WHERE idemp = 20 ;
  DBMS_OUTPUT . PUT_LINE ( ligne.prenom  || '  ' || ligne.nom) ;
EXCEPTION
     WHEN OTHERS THEN
            IF    SQLCODE = 100 THEN
                  DBMS_OUTPUT.PUT_LINE ( ' Erreur : aucun employé porte ce ID ' );
            ELSE
                DBMS_OUTPUT.PUT_LINE ('erreur inconnue') ;
            END IF;
END;
/
```

bloc anonyme terminé

Erreur : aucun employé porte ce ID

7. Exceptions utilisateur :

Ce sont des exceptions implémentées par le développeur. Elles doivent être déclarées avec le type **EXCEPTION** et sont levées avec l'instruction **RAISE**

Exemple : tout salaire < 300000 est considéré comme salaire trop bas. Vérifions pour l'employé de ID 10.

```
SET SERVEROUTPUT ON
DECLARE
```

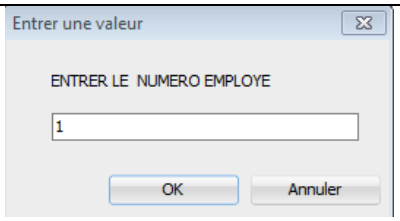
```
    salaire   numeric(8,2);
    salaire_trop_bas   EXCEPTION;
    CUR
BEGIN
    SELECT  salaire INTO  salaire
    FROM employes
    where idemp = 1;
    dbms_output.put_line('Salaire est de : ' || salaire);
    if salaire < 300000 then
         RAISE salaire_trop_bas;
     end if;
EXCEPTION
     WHEN salaire_trop_bas
           THEN dbms_output.put_line('le Salaire est trop bas car < 200000');
     WHEN OTHERS THEN
            dbms_output.put_line(SQLERRM);
END;
/
```

bloc anonyme terminé
Salaire est de : 239000
le Salaire est trop bas car < 200000

Allons plus loin en essayant de paramétrer la tache. Ecrivons un script qui compare le salaire d'un employé avec le salaire moyen.

```
SET SERVEROUTPUT ON
   ACCEPT  ID PROMPT 'ENTRER LE  NUMERO EMPLOYE'
DECLARE
   salaire_trop_bas   EXCEPTION;
   LIGNES  employes%ROWTYPE;
   MOY    NUMBER;
   SAL employes.salaire%TYPE;
BEGIN
      SELECT   ROUND (AVG(SALAIRE),0) INTO  MOY
      FROM EMPLOYES;
      dbms_output.put_line('le salaire moyen est ' || MOY);
      SELECT * INTO LIGNES
      FROM  EMPLOYES   WHERE  Idemp=&ID;
      dbms_output.put_line('le salaire de ' || Lignes.idemp || ' '||lignes.prenom || ' ' || lignes.nom || ' est ' ||
lignes.salaire);
```

```
    SAL := lignes.salaire;
    IF SAL < MOY then
       RAISE salaire_trop_bas;
    end if;
    EXCEPTION
     WHEN salaire_trop_bas
          THEN dbms_output.put_line('le Salaire est trop bas car inferieur à la moyenne');
     WHEN OTHERS THEN
           dbms_output.put_line(SQLERRM);
END;
/
```




```
bloc anonyme terminé
le salaire moyen est 479900
le salaire de 1 Awa Diop est 239000
le Salaire est trop bas car inferieur à la moyenne
```

Chapitre 4:

Les procédures et fonctions

Sous Oracle, un programme peut être conçu et exécuté de deux façons :

- on peut exécuter directement un bloc PL/SQL anonyme,
- on peut crée une procédure ou une fonction stockée sous forme d'objet afin de pouvoir la réutiliser.

Ainsi, Il est donc possible de créer des procédures et des fonctions comme dans n'importe quel langage de programmation classique. Toute fonction ou procédure créée sous Oracle devient ainsi un objet à part entière de la base comme une table ou une vue. Elle est souvent appelée "procédure stockée".

Elle est donc, entre autres, sensible à la notion de privilège et de droit : son créateur peut décider par exemple d'en permettre l'utilisation à d'autres utilisateurs du système ou d'en restreindre l'accès.

Etant un objet stocké et utilisable, elle est alors appelable depuis n'importe quel autre bloc PL/SQL.

On peut à cette effet créer des bibliothèques de fonctions et de procédures appelées "packages".

L'utilisation de procédures ou de fonctions permettra d'enregistrer des programmes réutilisables dans le noyau d'Oracle

Une procédure est une méthode sans valeur de retour. Une fonction comporte une instruction obligatoirement dans le bloc une instruction return qui renvoie la variable résultat de la fonction : **return** (nom_var_résultat);

Il y a trois façons de passer les paramètres dans une procédure : IN (lecture seule), OUT (écriture seule), INOUT (lecture et écriture) ou NOCOPY.

- **Le mode IN** : paramètre en entrée non modifiables par la procédure. Ce qui le désigne comme une valeur passée au sous-programme. Le fait de définir un paramètre IN l'empêche de recevoir une valeur dans le sous programme. Elle devient alors une variable STATIC: EN LECTURE SEULE
- **Le mode OUT :** paramètre en sortie modifiable par la procédure. Ils sont définis comme des valeurs OUT rendues au programme l'appelant. Ces paramètres sont non initialisés à l'entrée du sous programme et peuvent recevoir une valeur à l'intérieur du sous programme. Le fait de désigner un paramètre OUT interdit de l'utiliser dans une expression de sous-programme tel que les opérations, l'affichage … EN ECRITURE SEULE
- **Le mode IN OUT** : paramètre en entrée et en sortie, peut être modifiée en sortie et consultée par la procédure. Ces paramètres sont des variables initialisées qui sont disponibles pour des réaffectations dans le sous-programme.
- **Le mode NOCOPY** permet de passer des références (passage par référence) et non des valeurs

NB : par défaut une variable est en mode IN.

C'est un programme écrit en PL/SQL , qui peut être appelé :

- en mode interactif ou toute seule
- dans une application
- dans d'autres procédures ou dans des déclencheurs

1. Les procédures

Syntaxe :

```
CREATE OR REPLACE PROCEDURE     nom_PROCEDURE  ( Param1 Mode1 TYPE, Param2 TYPE
DEFAULT  Valeur , Para…)
  IS
Variable1  Type;
Variable2  Type;
…..
BEGIN
    i n s t r u c t i o n s
END  nom_PROCEDURE  ;
```

Règles :

- *Mode : c'est soit IN, OUT ou IN OUT.*
- *Variable1 et variable2 sont des variables locales donc visibles qu'a l'intérieur de la procédure.*
- *Des valeurs par défaut peuvent être spécifiées dans la définition de la procédure. De cette manière, la procédure peut être appelée avec tous, un ou aucun des paramètres. (Param2 TYPE DEFAULT Valeur)*

Exemple : créer une procédure PAddition qui fait la somme de deux valeurs.

```
CREATE OR REPLACE PROCEDURE Paddition  ( X IN NUMBER , Y IN NUMBER  )
AS
res NUMBER;
BEGIN
  RES := x+y;
  dbms_output.put_line('la somme est :' || RES);
END Paddition;
```

```
Elément PROCEDURE PADDITION compilé
```

Exécution :

En PL/SQL, une procédure s'invoque tout simplement avec son nom. On peut utiliser le mot-clé CALL ou EXECUTE.

Exemple :

```
EXECUTE   Paddition(10,20);
```

```
bloc anonyme terminé
la somme est :30
```

```
CALL  Paddition(10,20);
```

```
bloc anonyme terminé
la somme est :30
```

NB : pour lister les parametres d'un fonction ou d'une procedure on utilise la commande DESC.

Exemple :

```
DESC  Paddition ;
```

```
Nom d'argument    Type        In/Out    Valeur par défaut
--------------- ------ ------ ----------------------------------------
X                 NUMBER        IN          unknown
Y                 NUMBER        IN          unknown
```

Exercice 1 : Créer une procédure qui liste les informations d'un employé.

```
CREATE OR REPLACE PROCEDURE PINFOEMPLOYE  ( idE IN  NUMBER)
AS
    infos employes%ROWTYPE;
BEGIN
    SELECT * INTO infos
    FROM  employes
    WHERE idemp=idE;
    dbms_output.put_line('PRENOM :' || infos.prenom);
    dbms_output.put_line('NOM :' || infos.nom);
    dbms_output.put_line('DATE  EMBAUCHE :' || infos.date_embauche);
    dbms_output.put_line('SALAIRE :' || infos.salaire);
END PINFOEMPLOYE;
```

```
Elément PROCEDURE PINFOEMPLOYE compilé
```

```
EXECUTE  PINFOEMPLOYE (1);
```

```
bloc anonyme terminé
PRENOM :Awa
NOM :Diop
DATE  EMBAUCHE :12/10/00
SALAIRE :239000
```

```
EXECUTE  PINFOEMPLOYE (4);
```

```
bloc anonyme terminé
PRENOM :Aliou
NOM :Sarr
DATE  EMBAUCHE :19/06/00
SALAIRE :670000
```

NB : Au lieu de AS on peut aussi utiliser IS

2. les fonctions

Une Fonction est un sous programmes (méthode) qui retourne une valeur.

Syntaxe :

```
CREATE OR REPLACE FUNCTION     nom_FONCTION  ( Param1 Mode1 TYPE, Param2 TYPE
DEFAULT  Valeur , Para…)    RETURN   Type_Retourné
  IS
Variable1  Type;
Variable2  Type;
…..
BEGIN
    i n s t r u c t i o n s;
   RETURN   valeur;
END  nom_ FONCTION  ;
```

Règles :

- *L'instruction RETURN sert à retourner une valeur. Elle est obligatoire et est toujours la dernière instruction avant de sortir de la fonction.*
- ***TYPE_retourné** permet de definir le type de valeur de retour VALEUR_RETURN*
- *Dans une fonction **MODE** est toujours IN.*

NB : Au lieu de AS on peut aussi utiliserIS

Exemple : Ecrire un Fonction qui retourne une le salaire d'un employé

```
CREATE OR REPLACE FUNCTION Psaliare  ( idE IN  NUMBER) RETURN NUMBER
AS
    infos employes%ROWTYPE;
BEGIN
     SELECT * INTO infos
     FROM  employes
     WHERE idemp=idE;
      return infos.salaire;
END Psaliare;
```

Elément FUNCTION FONCCOMPARE compilé

Tout comme les procédures, l'invocation des fonctions ne pose aucun problème en PL/SQL, par contre, sous SQL+, c'est quelque peu particulier. On passe par une pseudo-table nommée DUAL de la façon suivante :

```
SELECT Psaliare (2) FROM DUAL ;
```

	PSALIARE(2)
1	560000

Les procédures et fonctions peuvent être utilisées dans d'autres procédures ou fonctions ou dans des blocs PL/SQL anonymes. Les fonctions peuvent aussi être utilisées dans les requêtes SQL

Exemple : Lister le prénom, nom et salaire des employés en utilisant notre fonction précédente

```
SELECT  prenom,nom, Psaliare(idemp) from employes;
```

PRENOM	NOM	PSALIARE(IDEMP)
Awa	Diop	239000
Fatou	Ka	560000
Modou	Diop	900000
Aliou	Sarr	670000
Serge	Sarr	230000
Edouard Ngor	Fall	500000
Doudou	Diop	400000
Fred	Tine	400000
Abdou	Lo	300000
Amie	Sow	600000

Exercice 1 : Ecrire un programme qui compte le nombre d'enregistrement dans une table.

```
create or replace function  nombre_employes     return number
IS
BEGIN
   DECLARE
      i number;
      CURSOR get_nb_employes IS select count(*) from employes;
  BEGIN
    open     get_nb_employes;
     fetch get_nb_employes INTO i;
    return i;
   END;
END;
/
```

Elément FUNCTION NOMBRE_EMPLOYES compilé

SELECT nombre_employes () FROM DUAL ;

NOMBRE_EMPLOYES()
10

Pour visualiser toutes les procédures et fonctions

```
select * from user_source ;
ou bien :
select Distinct(name),type from user_source
```

NAME	TYPE
PADDITION	PROCEDURE
FCOMPTERLIGNES	FUNCTION
NOMBRE_EMPLOYES	FUNCTION
PSALIARE	FUNCTION
PINFOEMPLOYE	PROCEDURE

Suppression de procédures ou fonctions

```
DROP PROCEDURE nom_procedure ;
DROP FUNCTION nom_fonction ;
```

Rappel : il existe d'autres types de fonction nommées Fonctions implicites c'est-à-dire des fonctions interne à Oracle donc non définies par le développeur. Ce sont :

- Fonctions de chaines
- Fonctions numériques
- Fonctions de date
- Fonctions de conversion
- Fonctions diverses

Chapitre 6:

Les packages

Le PL/SQL permet de grouper tous les programmes dans un objet de la base de données appelée un package. C'est un ensemble complet de programmes PL/SQL exécutant une certaine tâche est appelée un package. De même que dans les autres langages de programmation, un package est un ensemble de sous-programmes (procédures et Fonctions). Un package satisfait les points suivants :

- **L'encapsulation :** certains traitements sont masques, seule l'interface du package est visible.
- **modularité :** il est possible de développer séparément les diverses parties du package et après les assembler.

Un programme devient ainsi devient ainsi un assemblage de package.

Un package contient deux parties :

- La **spécification** dans la quelle on déclare l'ensemble des variable et des méthodes
- Le **cops** ou toutes les spécifications (procédures et fonctions) sont implémentées.

1. Création d'un package

La syntaxe de création d'un package pour l'écriture d'un script facilite la maintenance en cas de changements.

La création d'un package se fait en deux étapes :

a. la spécification

Syntaxes de la spécification:

```
CREATE OR REPLACE PACKAGE nompackage IS
   Variable type;
….
   procedure NomP( Param1  TYPE,Param2 Type,..)   ;
….
   function Nomf( Param1  TYPE,Param2 Type,..)   return TYPE ;
….
END nomPackage ;
/
```

b. le corp

Syntaxes du corps:

```
CREATE OR REPLACE PACKAGE  BODY  nompackage IS
     Variable type;
…..
      PROCEDURE  NomP ( Param1  TYPE,Param2 Type,..)  Is ….;
…
      FUNCTION  Nomf( Param1  TYPE,Param2 Type,..)    return TYPE   Is …;
…
END nomPackage ;
/
```

NB : Au lieu de IS on peut aussi utiliser AS

Ca donne :

```
CREATE OR REPLACE PACKAGE  BODY  nompackage AS
    Variable type;
…..
     PROCEDURE  NomP ( Param1  TYPE,Param2 Type,..)  Is ….;
…
     FUNCTION  Nomf( Param1  TYPE,Param2 Type,..)    return TYPE   Is …;
…
END nomPackage ;
/
```

Exemple: créer un package contenant deux méthodes:

- une fonction pour afficher le salaire d'un employé
- une procédure qui liste les informations d'un employé.

- La 53specification:

```
CREATE OR REPLACE PACKAGE PackEmployes IS
  procedure PlisteUnEmploye  (idE IN  NUMBER) ;
  Function FSalaireEmp( idE IN  NUMBER) RETURN NUMBER;
END PackEmployes ;
```

```
Elément PACKAGE PackEmployes compilé
```

- Le corps:

```
CREATE OR REPLACE PACKAGE BODY PackEmployes IS

 PROCEDURE PlisteUnEmploye  ( idE IN  NUMBER)
 IS
   infos employes%ROWTYPE;
 BEGIN
   SELECT * INTO infos
   FROM  employes
   WHERE idemp=idE;
   dbms_output.put_line('PRENOM :' || infos.prenom);
   dbms_output.put_line('NOM :' || infos.nom);
   dbms_output.put_line('DATE  EMBAUCHE :' || infos.date_embauche);
   dbms_output.put_line('SALAIRE :' || infos.salaire);
 END PlisteUnEmploye;
```

```
 FUNCTION FSalaireEmp  ( idE IN  NUMBER) RETURN NUMBER
 IS
    infos employes%ROWTYPE;
 BEGIN
     SELECT * INTO infos
     FROM  employes
     WHERE idemp=idE;
      return infos.salaire;
 END FSalaireEmp;


END PackEmployes ;
```

```
Elément PACKAGE BODY PACKEMPLOYES compilé
```

2. Utilisation :

On peut utiliser un package depuis n'importe quel script PL/SQL.

Syntaxe :

```
NomPackage. NomMethode;
```

Exemple pour la procédure: Listons les informations de l'employé 4 en utilisant la procédure PlisteUnEmploye se trouvant dans le package PACKEMPLOYES.

```
EXECUTE   PACKEMPLOYES.PlisteUnEmploye(4);
```

```
bloc anonyme terminé
PRENOM :Aliou
NOM :Sarr
DATE  EMBAUCHE :19/06/00
SALAIRE :670000
```

Exemple pour la fonction: Listons le salaire de l'employé 4 en utilisant la fonction FSalaireEmp se trouvant dans le package PACKEMPLOYES.

```
SELECT   PACKEMPLOYES. FSalaireEmp(4) FROM  DUAL ;
```

PACKEMPLOYES.FSALAIREEMP(4)
670000

NB:

- SET ECHO ON, affiche un listing du package à l'écran comme il a été compilé.
- La commande **ECHO SPOOL Chemin/nomFic** crée un fichier de listing (order_total.lst) à des fins de debugging. Ce fichier contiendra la compilation de la procédure, y compris les erreurs, avec les numéros de ligne.

TP : créer un package nommé PACKAGE_EMPLOYES pour gérer les employés avec les méthodes suivantes :

3. Méthodes pour lister les employés,
4. Méthodes pour lister les employés en fonction ayant le même salaire,
5. Méthodes pour ajouter un employé,
6. Méthodes pour modifier un employé,
7. Méthodes pour supprimer un employé,
8. Méthodes pour rechercher un employé,

Chapitre 7:

Les triggers

Les déclencheurs ou ”triggers” sont des séquences d’actions définis par le programmeur qui se déclenchent, non pas sur un appel, mais directement quand un événement particulier (spécifié lors de la définition du trigger) sur une ou plusieurs tables se produit. Un trigger sera un objet stocké (comme une table ou une procédure)

Ainsi, un trigger est une procédure stockée qui se lance automatiquement lorsqu'un événement se produit. Par événement, on entend toute modification des données se trouvant dans les tables de ma base de données.
On s'en sert pour contrôler ou appliquer des contraintes qu'il est impossible de formuler de façon déclarative.

Lors de la création d'un trigger, il convient de préciser quel est le type d'événement qui le déclenche
Au moment de l’exécution du trigger, on précise aussi si le trigger doit être exécuté

- avant (BEFORE) l'événement.
- après (AFTER) l'événement.

Nous avons deux types de triggers :

- Triggers pour événement objet
- Triggers pour événement system

1. Triggers pour événement objet

a. création

Elle s’applique sur les modifications de données (DML). Donc ces triggers permettent d’automatiser des actions lors de certains événements du type :

- INSERT
- DELETE
- UPDATE

Lorsque l'on fait un DELETE ..., il y a une seule instruction, mais plusieurs lignes peuvent être affectées.
Dans ce cas de figure, le trigger doit-il être exécuté soit :

- pour chaque ligne affectée (**FOR EACH ROW**)
- seulement une fois pour toute l'instruction (**STATEMENT**)

Ainsi :

- un **FOR EACH ROW** est exécuté à chaque fois qu'une ligne est affectée. L’option FOR EACH ROW [WHEN (<condition>)]] fait s’exécuter le trigger à chaque modification d’une ligne de la table spécifiée (on dit que le trigger est de ”niveau ligne”). On dit que le trigger est de niveau ligne
- un **STATEMENT** est exécuté à chaque fois qu'une instruction est lancée qu’importe le nombre de lignes affecté. On dit que le trigger est de niveau table

Les préfixes NEW et OLD (en cas de UPDATE ou de DELETE) vont permettre de faire référence aux valeurs des colonnes après et avant les modifications dans la table.

Exemple : new.prenom

NB : l’instruction **RAISE_APPLICATION_ERROR (-Code , 'message') ;** permet de générer une erreur personnalisé sans nom avec le numero **code et le texte message**. Ce numéro erreur doit être compris entre -20000 et -20999

Syntaxe de base:

```
CREATE   OR   REPLACE  TRIGGER   nomtrigger
 [ BEFORE / AFTER ] [INSERT / DELETE /  UPDATE] ON nomtable
   [ FOR EACH ROW / STATEMENT]   WHEN    Condition
DECLARE
    /d e c l a r a t i o n s /
BEGIN
   /i n s t r u c t i o n s  à faire /
END;
```

Exemple 1: Créons un trigger pour bloquer le DELETE sur EMPLOYES.

```
CREATE OR REPLACE TRIGGER   pasDeDeleteDansEmployes
BEFORE   DELETE  ON  Employes
BEGIN
   RAISE_APPLICATION_ERROR (-20555 , 'Vous ne pouvez pas effectuer une suppression sur la table employes' ) ;
END;
/
```

```
Elément TRIGGER PASDEDELETEDANSEMPLOYES compilé
```

Vérifions le nombre de lignes avant de faire notre DELETE

```
select Count(*) from employes;
```

COUNT(*)
10

Faisons maintenant un DELETE sur EMPLOYES pour vider cette table :

```
Delete from employes;
```

```
Erreur commençant à la ligne: 1 de la commande -
Delete from employes
Rapport d'erreur -
Erreur SQL : ORA-20555: Vous ne pouvez pas effectuer une suppression sur la table employes
ORA-06512: à "OMAR.PASDEDELETEDANSEMPLOYES", ligne 2
ORA-04088: erreur lors d'exécution du déclencheur 'OMAR.PASDEDELETEDANSEMPLOYES'
```

Vérifions le nombre de lignes avant de faire notre DELETE

```
select Count(*) from employes;
```

COUNT(*)
10

Toujours 10 employes. Alors que c'est il passé ? Vous remarquez que comme l'erreur à été levée **avant la suppression**, les données sont toujours présentes dans la table EMPLOYES.

Si à la place de BEFORE nous avions un AFTER alors dans ce cas les lignes serons supprimées et ensuite l'événement lancé.

NB : Pour visualiser les triggers il faut consulter le dictionnaire de données via la vue **USER_TRIGGERS**

```
SELECT  TRIGGER_NAME,TRIGGER_TYPE,TRIGGERING_EVENT,STATUS  from  user_triggers;
```

	TRIGGER_NAME	TRIGGER_TYPE	TRIGGERING_EVENT	STATUS
1	PASDEDELETEDANSEMPLOYES	BEFORE STATEMENT	DELETE	ENABLED

b. Activer / Désactiver un trigger:

Activation

```
ALTER TRIGGER nom_decl DISABLE ;
```

Desactivation

```
ALTER TRIGGER nom_trigger  ENABLE ;
```

c. combinaisons d'événements

Il est possible, en séparant les types d'événement par le mot-clé OR, de définir un trigger déclenche par plusieurs événements. Les variables booléennes INSERTING, UPDATING et DELETING permettent d'identité l'événement qui a déclenche le trigger.

Syntaxe :

```
CREATE TRIGGER
BEFORE INSERT OR UPDATE OR DELETE
ON…..
BEGIN
    IF INSERTING THEN ….. END IF;
    IF DELETING THEN ….. END IF;
     IF UPDATING THEN ….. END IF;
END ;
```

Exemple 2: bloquons les INSERT, UPDATE et DELETE sur employes

```
CREATE  OR REPLACE TRIGGER  BloquerDML
BEFORE  INSERT  OR  UPDATE  OR  DELETE  ON  employes
FOR EACH ROW
BEGIN
   IF INSERTING THEN
          RAISE_APPLICATION_ERROR (-20000 , 'Vous ne pouvez pas inserer  sur la table employes' ) ;
   ELSIF UPDATING THEN
          RAISE_APPLICATION_ERROR (-20001 , 'Vous ne pouvez pas modifier  sur la table employes' )
;
```

```
    ELSE
            RAISE_APPLICATION_ERROR (-20002 , 'Vous ne pouvez pas supprimer  sur la table employes'
) ;
END IF ;
END;
```

Elément TRIGGER BloquerDML compilé

- Faisons un UPDATE sur EMPLOYES

```
UPDATE EMPLOYES set prenom='jean' where  idemp = 2;
```

Rapport d'erreur -
Erreur SQL : ORA-20001: Vous ne pouvez pas modifier sur la table employes
ORA-06512: à "OMAR.AFFICHEEVENEMENT", ligne 1
ORA-04088: erreur lors d'exécution du déclencheur OMAR.AFFICHEEVENEMENT'

- Faisons un DELETE

```
Delete from employes;
```

Rapport d'erreur -
Erreur SQL : ORA-20555: Vous ne pouvez pas effectuer une suppression sur la table employes
ORA-06512: à "OMAR.PASDEDELETEDANSEMPLOYES", ligne 1
ORA-04088: erreur lors d'exécution du déclencheur 'OMAR.PASDEDELETEDANSEMPLOYES'

- Faisons un INSERT

```
INSERT INTO EMPLOYES VALUES
(12,'Edouard','sarr','17/06/1997',240000);
```

Rapport d'erreur -
Erreur SQL : ORA-20000: Vous ne pouvez pas inserer sur la table employes
ORA-06512: à "OMAR.AFFICHEEVENEMENT", ligne 1
ORA-04088: erreur lors d'exécution du déclencheur 'OMAR.AFFICHEEVENEMENT'
20000. 00000 - "%s"
*Cause: The stored procedure 'raise_application_error'
        was called which causes this error to be generated.

Exercice 3 : Archiver le nom de l'utilisateur, la date et l'action effectuée dans une nouvelle table nommée LOG_EMPLOYES lors de l'ajout, la suppression et la modification d'un employé de la table EMPLOYES.

- Créer la table LOG_EMPLOYES avec 3 colonnes USERNAME, DATEMODIF, TYPEMODIF

```
CREATE TABLE   LOG_EMPLOYES (
  USERNAME  VARCHAR2(20),
 DATEMODIF  VARCHAR2(56),
```

```
 TYPE_MODIF  VARCHAR2(20),
 idemp  NUMBER,
prenom  VARCHAR2(20),
nom  VARCHAR2(20),
DateEmbauche  DATE,
salaire  NUMBER ) ;
```

table LOG_EMPLOYES créé(e).

- Créons un trigger

```
CREATE OR REPLACE TRIGGER ArchiveEmployes
AFTER INSERT OR UPDATE OR DELETE ON employes
FOR EACH ROW
BEGIN
   IF INSERTING THEN
        insert into log_employes values ( USER,SYSDATE,'INSERT',:new.idemp,:new.prenom,
:new.nom,:new.date_embauche,:new.salaire);
    ELSIF UPDATING THEN
        insert into log_employes values ( USER,SYSDATE,'UPDATE',:old.idemp,:old.prenom,
:old.nom,:old.date_embauche,:old.salaire);
    ELSE
        insert into log_employes values ( USER,SYSDATE,'DELETE',:old.idemp,:old.prenom,
:old.nom,:old.date_embauche,:old.salaire);
   END IF;
END;
/
```

Elément TRIGGER ArchiveEmployes compilé

- Nous allons modifier le nom de l'employé 4 et nommons le BASSE

```
UPDATE employes SET nom='BASSE'  Where idemp=4;
```

1 lignes mis à jour.

IDEMP	PRENOM	NOM	DATE_EMBAUCHE	SALAIRE
1	Awa	Diop	12/10/00	239000
2	Fatou	Ka	06/10/15	560000
3	Modou	Diop	13/04/08	900000
4	Aliou	BASSE	19/06/00	670000
5	Serge	Sarr	23/02/14	230000
6	Edouard Ngor	Fall	06/07/11	500000
7	Doudou	Diop	12/10/15	400000
8	Fred	Tine	23/03/12	400000
9	Abdou	Lo	09/11/10	300000
10	Amie	Sow	14/10/01	600000

- Consultons maintenant notre table LOG_EMPLOYES

```
Select * from LOG_EMPLOYES ;
```

USERNAME	DATEMODIF	TYPE_MODIF	IDEMP	PRENOM	NOM	DATEEMBAUCHE	SALAIRE
OMAR	23/10/15	UPDATE	4	Aliou	BASSE	19/06/00	670000

2. Les déclencheurs sur événements système ou utilisateur

Depuis la version Oracle8i, il est désormais possible d'utiliser des déclencheurs pour suivre les changements d'état du système ainsi que les connexions/déconnexions utilisateur et la surveillance des ordres DDL et DML Lors de l'écriture de ces déclencheurs, il est possible d'utiliser des attributs pour identifier précisément l'origine des évènements et adapter les traitements en conséquence

a. Les attributs :

ORA_CLIENT_IP_ADRESS	Adresse IP du poste client qui se connecte
ORA_DATABASE_NAME	Nom de la base de données
ORA_DES_ENCRYPTED_PASSWORD	Description codée du mot de passe de l'utilisateur créé ou modifé
ORA_DICT_OBJ_NAME	Nom de l'objet visé par l'opération DDL
ORA_DICT_OBJ_NAME_LIST	Liste de tous les noms d'objets modifiés
ORA_DICT_OBJ_OWNER	Propriétaire de l'objet visé par l'opération DDL
ORA_DICT_OBJ_OWNER_LIST	Liste de tous les propriétaires d'objets modifiés
ORA_DICT_OBJ_TYPE	Type de l'objet visé par l'opération DDL
ORA_GRANTEE	Liste des utilisateurs disposant du privilège
ORA_INSTANCE_NUM	Numéro de l'instance
ORA_IS_ALTER_COLUMN	Vrai si la colonne en paramètre a été modifiée
ORA_IS_CREATING_NESTED_TABLE	Création ou non d'une table de fusion
ORA_IS_DROP_COLUMN	Modification ou non de la colonne en paramètre
ORA_IS_SERVERERROR	Vrai si le numéro erreur passé en paramètre se trouve dans la pile des erreurs
ORA_LOGIN_USER	Nom de la connexion
ORA_PRIVILEGES	Liste des privilèges accordés ou retirés par un utilisateur
ORA_REVOKEE	Liste des utilisateurs à qui le privilège a été retiré
ORA_SERVER_ERROR	Numéro d'erreur dans la pile dont la position est passée en paramètre
ORA_SYSEVENT	Nom de l'évènement système qui a activé le déclencheur
ORA_WITH_GRANT_OPTION	Vrai si le privilège a été accordé avec option d'administration

b. Les évènements système

Syntaxe :

```
CREATE TRIGGER nom_déclencheur {BEFORE|AFTER} évènement_système
ON{DATABASE|SCHEMA}
BEGIN
…
END ;
```

L'événement est soit :

- **STARTUP :** Evènement déclenché lors de l'ouverture de l'instance (AFTER seulement)
- **SHUTDOWN :** Evènement déclenché avant le processus d'arrêt de l'instance (non déclenché en cas d'arrêt brutal du serveur) (BEFORE seulement)
- **SERVERERROR :** Evènement déclenché lors d'une erreur Oracle (sauf ORA-1034, ORA-1403, ORA-1422, ORA-1423 et ORA-4030) (AFTER seulement)

c. Les évènements utilisateur

Syntaxe :

```
CREATE TRIGGER nom_déclencheur {BEFORE|AFTER} évènement_utilisateur ON
{DATABASE|SCHEMA}
BEGIN
…
END ;
```

L'événement est soit :

- **LOGON :** Après une connexion (AFTER seulement)
- **LOGOFF :** Avant une déconnexion (BEFORE seulement)
- **CREATE :** Lors de la création d'un objet
- **ALTER :** Lors de la modification d'un objet
- **DROP :** Lors de la suppression d'un objet
- **ANALYZE :** Lors de l'analise d'un objet
- **ASSOCIATE STATISTICS :** Lors de l'association d'une statistique
- **AUDIT :** Lors de la mise en place d'un audit
- **NOAUDIT :** Lors de l'annulation d'un audit
- **COMMENT :** Lors de l'insertion d'un commentaire
- **DDL :** Lors de l'exécution des ordres DDL (sauf ALTER DATABASE, CREATE CONTROLFILE et CREATE DATABASE)
- **DISSOCIATE STATISTICS :** Lors de la dissociation d'une statistique
- **GRANT :** Lors de l'exécution d'une commande GRANT

- **RENAME :** Lors de l'exécution d'une commande RENAME
- **REVOKE :** Lors de l'exécution d'une commande REVOKE
- **TRUNCATE :** Lors d'une troncature de table

NB : Un trigger peut être activé ou désactivé par les instructions : **ALTER TRIGGER <nom> {ENABLE|DISABLE};** et détruit par : **DROP TRIGGER <nom>.**

Exemple : Surveillons les connexions à la base de données. Faire le travail ous SYS c'est lui qui dispose des privilèges

- Créons la table AUDITCONNECT

```
CREATE TABLE AUDITCONNECT (
DateCon DATE,
login Varchar2(77)
);
```

- Créons le trigger

```
CREATE OR REPLACE TRIGGER SURVEILLECONNECT
AFTER LOGON ON DATABASE
BEGIN
  INSERT  INTO AUDITCONNECT  Values (SYSDATE,ORA_LOGIN_USER);
END;
```

- Maintenant on va se connecter successivement avec Omar, anissa, livre et helene
- Listons maintenant notre table AUDITCONNECT

```
SELECT  *  FROM AUDITCONNECT;
```

DATECON	LOGIN
23/10/15	SYS
23/10/15	SYS
23/10/15	OMAR
23/10/15	AGNES
23/10/15	HELENE
23/10/15	SYS

NB : Les informations sur les déclencheurs sont visibles à travers les vues du dictionnaire de données **USER_TRIGGERS** pour les déclencheurs appartenant au schéma **ALL_TRIGGERS** pour les déclencheurs appartenant aux schémas accessibles **DBA_TRIGGERS** pour les déclencheurs appartenant à tous les schémas.

Référence :

- Site Oracle et PL/SQL : http://www.techonthenet.com/oracle/
- Référence PL/SQL Oracle : http://download.oracle.com/docs/cd/B19306 01/appdev.102/b14261/toc.htm
- Cours 5 : PL/SQL http://deptinfo.unice.fr/~grin/messupports/plsql.pdf http://sheikyerbouti.developpez.com/pl_sql/
- Anne Vilnat , Cours 4 : PL/SQL : ou comment faire plus avec ORACLE 2_eme partie
- Alexandre Mesle Introduction au PL/SQL Oracle, 2011