

Chapitre 4 : PL/SQL

(adapté d'un cours de A. Arsane)

L'objectif de ce chapitre est de découvrir comment implanter des traitements dans un SGBDR en utilisant le langage PL/SQL.

Table des matières

Introduction à PL/SQL.....	2
Présentation du langage.....	2
Un premier exemple.....	2
Programmer avec PL/SQL.....	3
La notion de bloc.....	3
Utilisation de variables.....	3
Déclaration.....	3
Types utilisables.....	3
Constantes.....	4
Opérateurs.....	4
Structures de contrôles.....	4
Structure SI.....	4
Structure SELON.....	5
Structure TANTQUE.....	5
Structure POUR.....	6
Procédures stockées.....	6
Exemple.....	6
Définition d'une procédure.....	7
Définition d'une fonction.....	8
Suppression d'une procédure stockée.....	9
Manipulation de données avec PL/SQL.....	9
Types référencés.....	9
Sélection de données dans une variable.....	9
Curseurs.....	10
Curseurs explicites.....	10
Déclaration d'un curseur.....	11
Ouverture et fermeture d'un curseur.....	11
Parcours d'un curseur.....	12
Mise à jour au travers d'un curseur.....	12
Curseurs implicites.....	13
Déclencheurs.....	13
Définition d'un déclencheur.....	14
Préfixes OLD et NEW.....	14
Suppression d'un déclencheur.....	15
Gestion des erreurs.....	15

Introduction à PL/SQL

Présentation du langage

PL/SQL (*Procedural Language/SQL*) est un langage **procédural** totalement intégré à Oracle et SQL. Il permet le développement d'applications dans l'environnement Oracle à travers l'écriture de **procédures** et de **fonctions**. Il permet de combiner des requêtes SQL et des instructions procédurales (boucles, conditions...), dans le but de créer des traitements complexes destinés à être stockés sur le serveur de base de données (objets serveur), comme par exemple des **procédures stockées** ou des **déclencheurs**. Les éléments définis en PL/SQL sont accessibles depuis un langage externe au SGBDR (Java, C#, ...) ainsi que depuis les environnement clients ORACLE (SQL*Plus, SQL Developer, etc).

A ce propos, pourquoi utiliser PL/SQL plutôt qu'un langage externe au SGBDR ? Essentiellement parce que l'écriture de code au niveau du SGBDR permet de rendre les traitements plus rapides (moins d'allers/retours réseau) et de travailler directement sur les données stockées. PL/SQL est un complément très utile aux langages externes pour réaliser des applications liées à ORACLE.

Un premier exemple

L'apprentissage d'un nouveau langage commence traditionnellement par l'affichage à l'écran d'un message "Hello world !". Voici sa traduction en français et en PL/SQL.

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Bonjour monde !');
END;
```

DBMS_OUTPUT.PUT_LINE est une **procédure** qui écrit un texte sur la sortie standard (l'écran le plus souvent).

Remarque : pour que cette instruction fonctionne sous (i)SQL*Plus, il est nécessaire d'activer au préalable l'option SERVEROUTPUT grâce à l'instruction suivante :

```
set serveroutput on
```

Programmer avec PL/SQL

La programmation sous PL/SQL fait appel à des concepts que vous connaissez déjà : variables, conditions, boucles, etc.

La notion de bloc

Tout programme PL/SQL se présente sous la forme de **blocs**.

La structure d'un bloc est la suivante (les sections **DECLARE** et **EXCEPTION** sont facultatives) :

```
[DECLARE]
    déclarations et initialisation
BEGIN
    instructions exécutables
[EXCEPTION]
    interception des erreurs
END;
```

Utilisation de variables

Déclaration

Comme tout langage, PL/SQL permet l'utilisation de variables. Elles sont déclarées dans la partie **DECLARE** d'un bloc, comme dans l'exemple ci-dessous.

```
DECLARE
    msg VARCHAR2(20); -- déclaration d'une variable
BEGIN
    msg := 'Bonjour monde !';
    DBMS_OUTPUT.PUT_LINE(msg);
END;
```

Quel est le résultat de son exécution ?

Il affiche le même message "Bonjour monde !" que l'exemple précédent.

On peut tirer de cet exemple les conclusions suivantes :

- ✓ La déclaration d'une variable se fait sous la forme *nomVariable type* ;
- ✓ L'affectation d'une variable se fait grâce à l'expression *nomVariable := valeur* ;
- ✓ Les commentaires sont créés grâce aux caractères --
- ✓ Les valeurs littérales (chaînes de caractères, etc) sont placées entre ''

Types utilisables

La déclaration d'une variable permet de préciser son **nom** et son **type**. Les types utilisables sont ceux proposés par le SGBDR ORACLE : les types SQL standards comme VARCHAR, INTEGER, FLOAT ou encore DATE et des types spécifiques à ORACLE comme VARCHAR2.

Constantes

On peut déclarer comme **constante** une variable dont la valeur ne changera jamais après sa définition. L'exemple ci-dessous affiche le message "**PI vaut 3,14**".

```
DECLARE
  PI CONSTANT NUMBER(3,2) := 3.14;
BEGIN
  DBMS_OUTPUT.PUT_LINE('PI vaut ' || pi);
END;
```

On peut tirer de cet exemple les conclusions suivantes :

- ✓ La déclaration d'une constante se fait sous la forme :
nomConstante CONSTANT type := valeur;
- ✓ La concaténation de chaînes utilise l'opérateur **||**
- ✓ PL/SQL est **insensible** à la casse des variables. Cependant, garder une cohérence dans le nommage renforce la lisibilité du programme.
- ✓ NUMBER(e, d) permet de stocker un nombre réel avec e chiffres significatifs dont d décimales (chiffres après la virgule).

Opérateurs

Outre les opérateurs déjà étudiés comme := ou ||, PL/SQL permet d'employer différents types d'opérateurs :

- Arithmétiques : +, -, *, /
- Comparaisons : =, !=, <, <=, >, >=
- Logiques : AND, OR, NOT
- SQL : IS NULL, LIKE, BETWEEN, IN

Structures de contrôles

Structure SI

La traditionnelle structure SI ... SINON permet d'exprimer des alternatives basées sur des conditions (expressions ayant une valeur **booléenne**).

```
DECLARE
  panier INTEGER := 1;
BEGIN
  IF panier = 0 THEN
    DBMS_OUTPUT.PUT_LINE('votre panier est vide');
  ELSE
    DBMS_OUTPUT.PUT_LINE('votre panier n'est pas vide');
  END IF;
END;
```

L'exécution de ce code produit l'affichage : "**Votre panier n'est pas vide**".

Le SI ... SINON s'écrit de la manière suivante en PL/SQL :

IF (<condition>) THEN	ELSE
<actions>	<actions>

END IF;

On observe qu'on est très proche du SI ... SINON algorithmique.

Structure SELON

La structure SELON ... CAS permet d'effectuer un traitement parmi plusieurs possibles.

```
DECLARE
    meteo VARCHAR2(30) := 'neige';
BEGIN
    CASE meteo
        WHEN 'soleil' THEN DBMS_OUTPUT.PUT_LINE('Il fait beau');
        WHEN 'nuages' THEN DBMS_OUTPUT.PUT_LINE('Le temps se couvre');
        WHEN 'pluie' THEN DBMS_OUTPUT.PUT_LINE('Il pleut');
        ELSE DBMS_OUTPUT.PUT_LINE('Il fait nuit ?');
    END CASE;
END;
```

Le SELON ... CAS s'écrit de la manière suivante en PL/SQL :

```
CASE <variable>
    WHEN valeur1 THEN <action1> ;
    WHEN valeur2 THEN <action2> ;
    ...
    ELSE <action>;
END CASE;
```

Structure TANTQUE

La boucle TANTQUE permet d'effectuer des itérations tant qu'une condition est vraie.

```
DECLARE
    i INTEGER := 0;
BEGIN
    WHILE i < 3 LOOP
        i := i + 1;
        DBMS_OUTPUT.PUT_LINE(i);
    END LOOP;
END;
```

Le corps de la boucle ci-dessus sera exécuté **trois** fois.

Le TANTQUE s'écrit de la manière suivante en PL/SQL :

```
WHILE (<condition>) LOOP
    <actions>
END LOOP;
```

Structure POUR

La boucle POUR permet de connaître à l'avance le nombre d'itérations effectués.

```
DECLARE
    i INTEGER;
```

```
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE(i);
  END LOOP;
END;
```

Le résultat de son exécution est identique à celui de l'exemple précédent.

Le POUR s'écrit de la manière suivante en PL/SQL :

```
FOR indice IN [REVERSE] debut..fin LOOP
  <actions>
END LOOP;
```

Procédures stockées

On désigne par ce terme un sous-programme (procédure ou fonction) enregistré dans la base de données. Il est possible de faire appel à une procédure stockée :

- depuis une interface cliente ORACLE (SQL*Plus, SQL Developer, etc),
- depuis un langage hôte (Java, C#, etc).

Exemple

Voici un exemple de définition d'une procédure PL/SQL.

```
CREATE OR REPLACE PROCEDURE Somme
(nb1 IN NUMBER, nb2 IN NUMBER, somme OUT NUMBER) IS
BEGIN
  somme := nb1 + nb2;
END;
```

Cette procédure affecte à la variable **somme**, passée en **écriture** (mot-clé **OUT**) la somme des valeurs des variables **nb1** et **nb2**, passées en **lecture** (mot-clé **IN**).

La création d'une procédure stockée depuis un client ORACLE déclenche la compilation de la procédure et son stockage au sein du SGBDR si aucune erreur de compilation n'est détectée. On peut ensuite faire appel à cette procédure depuis un autre bloc PL/SQL.

```
DECLARE
  resultat NUMBER;
BEGIN
  Somme(10, 20, resultat);
  DBMS_OUTPUT.PUT_LINE('Résultat : ' || resultat);
END;
```

Ici le résultat affiché vaudra **30**, puisque **nb1** (paramètre **formel**) reçoit la valeur **10** (paramètre **effectif**) et que **nb2** reçoit la valeur **20** au moment de l'appel de **Somme()**.

Remarque : il est également possible de faire appel à une procédure stockée en précisant explicitement la correspondance entre paramètres formels et effectifs, comme ci-dessous.

```
DECLARE
  nombre1 NUMBER;
```

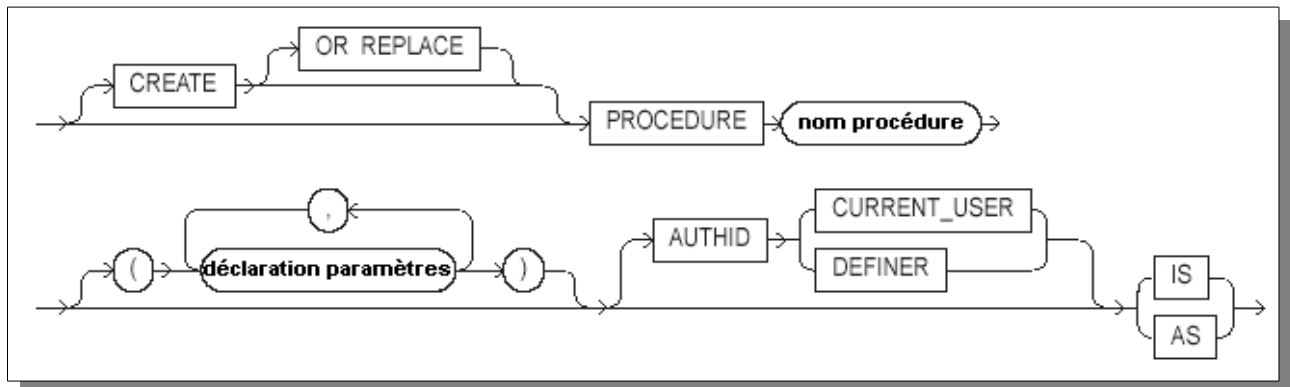
```

nombre2 NUMBER;
resultat NUMBER;
BEGIN
nombre1 := 10;
nombre2 := 20;
Somme(somme=>resultat, nb1=>nombre1, nb2=>nombre2);
DBMS_OUTPUT.PUT_LINE('Résultat : ' || resultat);
END;

```

Définition d'une procédure

Le schéma suivant présente la syntaxe complète de définition d'une procédure en PL/SQL



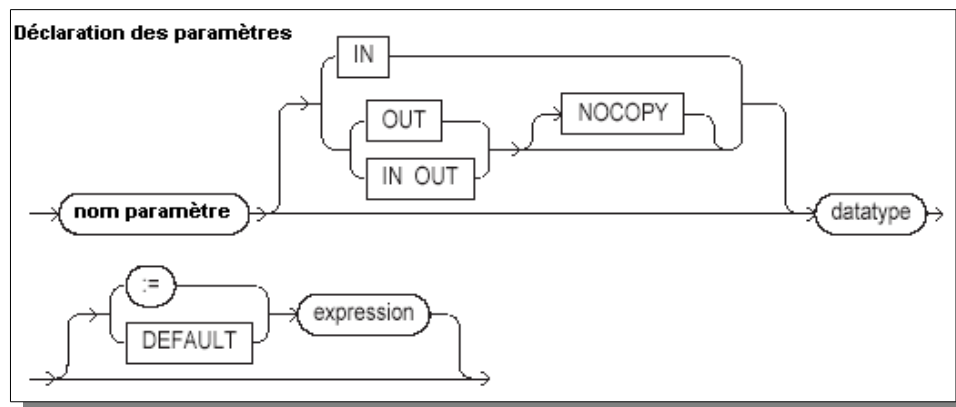
CREATE indique que l'on veut créer une procédure stockée dans la base.

La clause facultative **OR REPLACE** permet d'écraser une procédure existante de même nom.

Le mot-clé **IS** indique le début du code de la procédure.

(La clause facultative **AUTHID** indique sur quel schéma la procédure s'applique).

La définition des paramètres d'une procédure stockée se fait de la manière suivante.



IN (valeur par défaut) indique que le paramètre transmis par le programme appelant n'est pas modifiable par la procédure.

OUT indique que le paramètre est modifiable par la procédure.

datatype représente le type SQL ou PL/SQL du paramètre.

(**IN OUT** indique que le paramètre est transmis par l'appelant et renseigné par la procédure)

(**NOCOPY** indique que le paramètre est transmis par référence (pointeur) et non par copie)

(**:=** ou **DEFAULT** représente le symbole d'assignation d'une valeur par défaut)

Définition d'une fonction

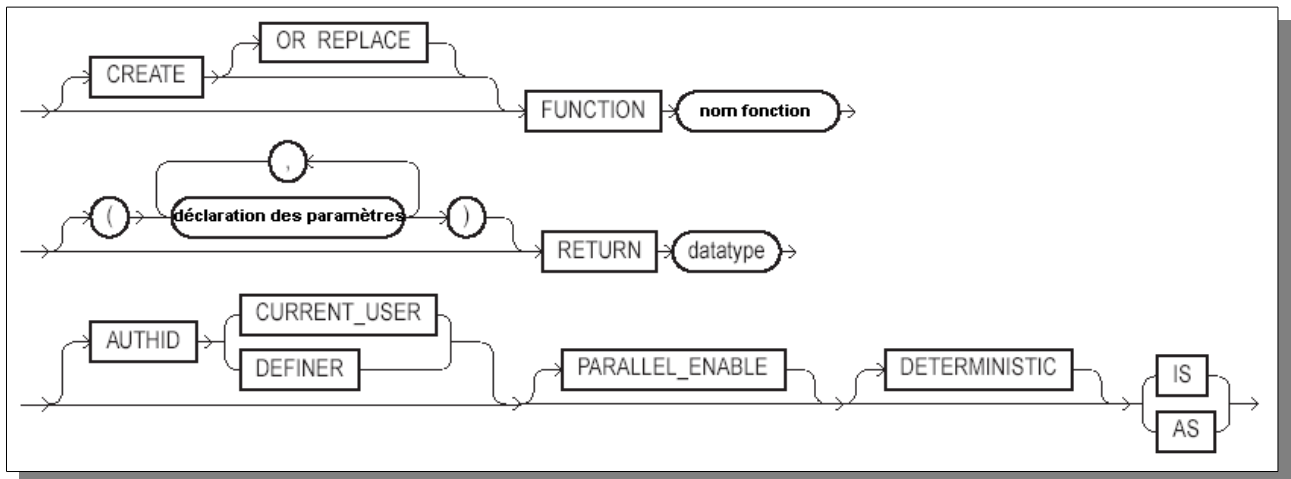
Une fonction est un sous-programme qui, contrairement à une procédure, renvoie une valeur de retour. Voici le sous-programme précédent, traduit sous forme de fonction.

```
CREATE OR REPLACE FUNCTION SommeBis
(nb1 IN NUMBER, nb2 IN NUMBER)
RETURN NUMBER IS
    somme NUMBER;
BEGIN
    somme := nb1 + nb2;
    RETURN(somme);
END;
```

Et voici un exemple d'appel de la fonction créée.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('Résultat : ' || SommeBis(10, 20));
END;
```

Le schéma suivant présente la syntaxe complète de définition d'une fonction en PL/SQL.



La principale différence avec une procédure vient du mot-clé **RETURN** qui indique le type de retour de la fonction.

Suppression d'une procédure stockée

L'instruction **DROP** permet de supprimer une procédure stockée. Exemples d'utilisation :

```
DROP PROCEDURE Somme;  
DROP FUNCTION SommeBis;
```

Manipulation de données avec PL/SQL

Jusqu'à présent, nous avons utilisé PL/SQL comme n'importe quel langage de programmation classique. Cependant, celui-ci est avant tout dédié à l'interaction avec les informations stockées dans la base de données et dispose d'outils spécifiques pour cet usage.

Types référencés

L'un des avantages de l'intégration de PL/SQL au SGBDR est qu'il est possible de déclarer des variables faisant référence au type précis d'une colonne.

```
DECLARE  
    nomUtil T_UTILISATEUR.USR_NOM%TYPE;  
BEGIN  
    ...  
END;
```

Ici, le type de la variable **nomUtil** sera exactement celui du champ **USR_NOM** de la table **T_UTILISATEUR**.

La syntaxe de la déclaration d'un type référencé est **nomVariable table.colonne%type ;**

Cette méthode de déclaration est à privilégier dans tous les cas où on manipule des variables faisant référence à des colonnes d'une table.

Sélection de données dans une variable

Sous certaines conditions, il est possible d'affecter la valeur renvoyée par un ordre SQL **SELECT** à une variable PL/SQL.

```
DECLARE  
    nbArticles INTEGER;  
BEGIN  
    SELECT COUNT(*) INTO nbArticles FROM ARTICLE;  
    DBMS_OUTPUT.PUT_LINE('Nombre d'articles : ' || nbArticles);  
END;
```

Ce programme mémorise dans la variable **nbArticles**, puis affiche le **nombre d'articles** de la table **T_ARTICLE**.

La syntaxe de l'instruction **SELECT ... INTO** est la suivante.

```
SELECT colonne1, colonne2, ...  
INTO variable1, variable2, ...  
FROM ...
```



Pour que cette instruction s'exécute sans erreur, la requête **SELECT** doit **obligatoirement** renvoyer une et une seule valeur.

Curseurs

Un curseur permet de manipuler ligne par ligne les données retournées par une requête. L'utilisation d'un curseur permet d'itérer sur un ensemble de lignes en effectuant des traitements sur chaque ligne.

Curseurs explicites

Un curseur est dit **explicite** quand il est explicitement déclaré au début d'un bloc PL/SQL, comme dans l'exemple ci-dessous.

```
DECLARE  
  CURSOR cde IS  
    SELECT nocde, etacde, nocli FROM commande;  
  vcde cde%ROWTYPE;  
  cpt_cde NUMBER;  
BEGIN  
  cpt_cde := 0;  
  OPEN cde;  
  FETCH cde INTO vcde;  
  WHILE cde%FOUND LOOP  
    IF (vcde.etacde = 'N') THEN  
      cpt_cde := cpt_cde + 1;  
    END IF;  
    FETCH cde INTO vcde;  
  END LOOP;  
  CLOSE cde;  
  DBMS_OUTPUT.PUT_LINE(cpt_cde);  
END;
```

Pouvez-vous deviner le résultat affiché par ce bloc de code ?

Le nombre de commandes dont l'état vaut "N".

Cet exemple est complexe. Nous allons l'étudier point par point.

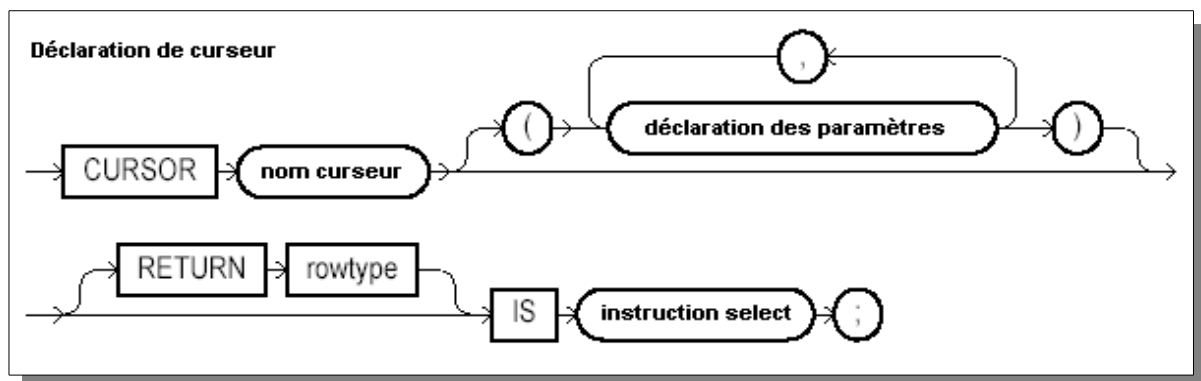
Déclaration d'un curseur

La déclaration d'un curseur se fait dans la partie **DECLARE** d'un bloc de code PL/SQL.

```
DECLARE
  CURSOR cde IS
    SELECT nocde, etacde, nocli FROM commande;
...
```

Ici, le curseur se nomme **cde** et permet de sélectionner certains champs de la table COMMANDE.

La syntaxe générale de la déclaration d'un curseur est donnée par le schéma ci-dessous.



Une fois le curseur déclaré, il est possible de définir des variables faisant référence à son type de ligne grâce au mot-clé **%ROWTYPE**, comme dans l'extrait ci-dessous. Cela permet de récupérer dans une variable chaque élément du curseur.

```
...
CURSOR cde IS
  SELECT nocde, etacde, nocli FROM commande;
  vcde cde%ROWTYPE;
...
```

Ici, la variable **vcde** correspond à un **enregistrement** (*record*) comportant trois champs : **nocde**, **etacde** et **nocli**.

Ouverture et fermeture d'un curseur

Pour pouvoir exploiter un curseur, il faut d'abord l'ouvrir, ce qui déclenche l'exécution de la requête associée. En fin d'utilisation, il est nécessaire de fermer un curseur ouvert.

```
...
OPEN cde;
...
CLOSE cde;
...
```

L'instruction **OPEN** ouvre un curseur précédemment déclaré.

L'instruction **CLOSE** ferme un curseur précédemment ouvert.

Il est possible d'utiliser des curseurs incluant des paramètres, comme dans cet exemple.

```
DECLARE
  CURSOR cde_no (numCommande IN commande.nocde%type) IS
    SELECT * FROM commande WHERE nocde = numCommande;
BEGIN
  OPEN cde_no(17);
  ...
  CLOSE cde_no;
END;
```

Parcours d'un curseur

Une fois le curseur ouvert, on peut itérer sur son contenu.

```
...
FETCH cde INTO vcde;
WHILE cde%FOUND LOOP
  ...
  FETCH cde INTO vcde;
END LOOP;
...
```

L'instruction **FETCH** déclenche la copie de la ligne courante du curseur dans une variable : elle fait avancer le curseur d'une ligne. L'attribut **%FOUND** permet de savoir si un **FETCH** a ramené des données ou non.

L'association d'un curseur et d'une boucle permet de parcourir ligne par ligne le résultat d'une requête SQL. La boucle peut être une boucle **WHILE** comme dans l'exemple ci-dessus, ou bien une boucle **FOR**.

-- Avec un WHILE

```
OPEN curseur
FETCH curseur INTO variable;
WHILE curseur%FOUND LOOP
  <actions>
  FETCH curseur INTO variable;
END LOOP;
CLOSE curseur
```

-- Avec un FOR

```
-- Pas d'ouverture ni de fermeture !
FOR variable IN curseur LOOP
  <actions>
END LOOP;
```

La boucle **FOR** permet de parcourir tout le curseur.

La boucle **WHILE** offre la possibilité de sortir en cours de parcours si sa condition devient fausse.

Mise à jour au travers d'un curseur

Lors de la déclaration d'un curseur, il est possible de spécifier que certains champs pourront être mis à jour en utilisant ce curseur.

```
DECLARE
  CURSOR cde IS
    SELECT nocde, etacde, nocli FROM commande
    FOR UPDATE OF etacde;
```

Le mot-clé **FOR UPDATE OF** indique les champs qui pourront être mis à jour.

Le mot-clé **FOR UPDATE** permettra de modifier l'ensemble des champs du curseur.

Une fois le curseur ouvert, il est possible de modifier la valeur des champs indiqués.

```
...  
UPDATE commande SET etacde = 'O' WHERE CURRENT OF cde;  
...
```

Le mot-clé **WHERE CURRENT OF** déclenche la mise à jour sur la position courante du curseur.

Curseurs implicites

PL/SQL offre la possibilité d'itérer sur les résultats d'une requête sans déclarer explicitement de curseur.

```
DECLARE  
  vClient client%rowtype;  
BEGIN  
  FOR vClient IN (SELECT * FROM client) LOOP  
    DBMS_OUTPUT.PUT_LINE(vClient.nocli || ' : ' || vClient.rscli);  
  END LOOP;  
END;
```

Que fait l'exemple ci-dessus ?

Il affiche la concaténation des numéros et raison sociale des clients.

Déclencheurs

La mise en place d'un **déclencheur** (*trigger*) permet de réaliser un traitement en réaction à un évènement de mise à jour des données d'une table (ajout, modification, suppression).

On peut choisir de programmer le déclencheur pour qu'il s'exécute :

- Avant la réalisation de l'évènement (déclencheur BEFORE).
- Après la réalisation de l'évènement (déclencheur AFTER).

On peut également choisir d'activer le déclencheur :

- Par ordre : il ne s'exécute qu'une seule fois.
- Par ligne (le plus fréquent) : il s'exécute pour chaque ligne concernée.

Définition d'un déclencheur

L'exemple ci-dessous permet de créer un déclencheur nommé tai_lignecde ("tai" est l'abréviation de "Trigger After Insert").

```
CREATE OR REPLACE TRIGGER tai_lignecde AFTER INSERT  
ON lignecde FOR EACH ROW  
BEGIN  
  UPDATE article SET stothe = stothe - :new.qtecde
```

```
WHERE article.noart = :new.noart;  
END;
```

Quel est le rôle de ce déclencheur ?

Mettre à jour le stock théorique de l'article commandé dans ARTICLE.

Les mots-clés **AFTER INSERT** signifient que le déclencheur sera exécuté après une insertion dans la table LIGNECDE.

Les mots-clés **FOR EACH ROW** signifient que le déclencheur s'exécute par **ligne**.



Un déclencheur par lignes ne peut pas agir sur la table sur laquelle il est défini !

Voici un exemple de définition d'un déclencheur après mise à jour.

```
CREATE OR REPLACE TRIGGER tau_lignecde AFTER UPDATE OF qtecde  
ON lignecde FOR EACH ROW  
BEGIN  
  UPDATE article SET stothe = stothe ...  
END;
```

Préfixes OLD et NEW

Ces préfixes permettent d'accéder aux données telles qu'elles seraient avant (OLD) et après (NEW) l'évènement qui active le déclencheur. Ils correspondent à la structure d'une ligne de la table.

On peut utiliser OLD et NEW dans le corps du déclencheur.

```
CREATE OR REPLACE TRIGGER tad_lignecde AFTER DELETE  
ON lignecde FOR EACH ROW  
BEGIN  
  UPDATE article SET stothe = stothe + :old.qtecde  
  WHERE article.noart = :old.noart;  
END;
```

Le tableau ci-dessous résume le comportement de OLD et NEW en fonction du contexte.

	NEW	OLD
INSERT	pointe la valeur à insérer	inutilisable
UPDATE	valeur après mise à jour	valeur avant mise à jour
DELETE	inutilisable	valeur avant suppression

Suppression d'un déclencheur

La commande DROP TRIGGER permet de supprimer un déclencheur existant.

```
DROP TRIGGER tai_lignecde;
```

Gestion des erreurs

Dans un bloc PL/SQL, la section **EXCEPTION** permet d'intercepter et de gérer les éventuelles erreurs qui se produisent pendant l'exécution. A l'intérieur de cette section, des instructions **WHEN** permettent de traiter les différents cas d'erreur.

```
...  
EXCEPTION  
  WHEN NO_DATA_FOUND THEN  
    ...  
  WHEN OTHERS THEN  
    IF SQLCODE = ... THEN  
      ...  
    ELSE  
      ...  
    END IF;  
END;
```

Ici, le premier **WHEN** permet de traiter l'erreur ORACLE NO_DATA_FOUND.
Le second **WHEN** gère toutes les autres erreurs (mot-clé **OTHERS**).