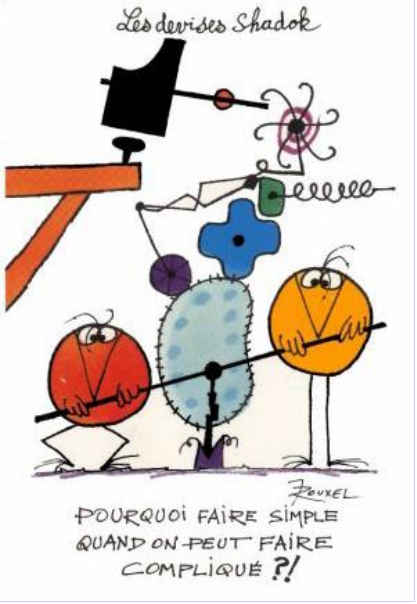
afpa
La formation professionnelle

Secteur Tertiaire Informatique
Filière étude - développement

FORMATION CONCEPTEUR DEVELOPPEUR INFORMATIQUE

ORACLE & LE LANGAGE PL/SQL



Description		
Réf. document :	CDI JEE - Langage PL-SQL.doc	
Projet :	Formation CDI JEE	
Emetteur :	Pascal DANGU	Tél : 05 61 17 20 48 Mail pascal.dangu@afpa.fr
Date d'émission :	07/2009	

Validation			
Nom	**Date**	**Validation (O/N)**	**Commentaires**

Historique des modifications			
Version	**Date**	**Etat**	**Description de la modification**
1.0	07/2009	Validé	Document initial rédigé par Pascal DANGU

Code barre

SOMMAIRE

1 OBJECTIFS DU DOCUMENT

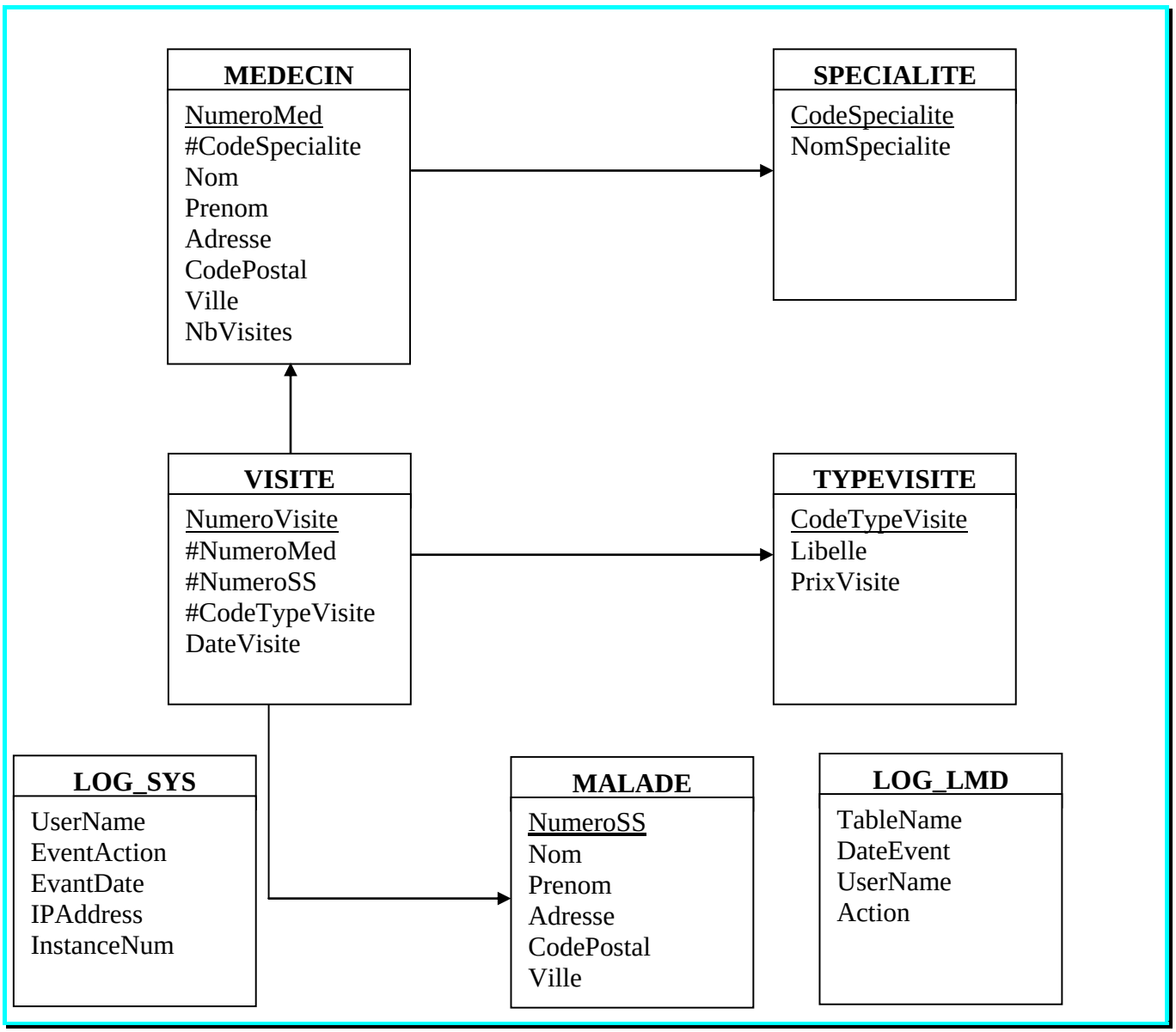
Ce document a pour objet de présenter le langage PL/SQL qui correspond au langage de programmation procédurale d'Oracle.

PL/SQL est un langage L4G (langage de quatrième génération), fournissant une interface procédurale au SGBD Oracle. Le langage PL/SQL intègre parfaitement le langage SQL en lui apportant une dimension procédurale.
En effet, le langage SQL est un langage déclaratif non procédural permettant d'exprimer des requêtes dans un langage relativement simple. En contrepartie il n'intègre aucune structure de contrôle permettant par exemple d'exécuter une boucle itérative.
Ainsi le langage PL/SQL permet de manipuler de façon complexe les données contenues dans une base Oracle en transmettant un bloc de programmation au SGBD au lieu d'envoyer une requête SQL. De cette façon les traitements sont directement réalisés par le système de gestion de bases de données. Cela a pour effet notamment de réduire le nombre d'échanges à travers le réseau et donc d'optimiser les performances des applications.

2 PREAMBULE

Certains exemples de ce document sont basés sur le Modèle Logique de Données Relationnel (au sens Merise) présenté ci-dessous.
Les clés primaires sont soulignées. Les clés étrangères sont précédées du caractère '#'.

Les tables LOG_SYS et LOG_LMD sont utilisées dans le cadre des déclencheurs.



Le script de création des tables et le script d'insertion des données sont présents dans le répertoire 01 - PRESENTATION\SCRIPTS PL-SQL :

01 - PRESENTATION\SCRIPTS PL-SQL\01 - Langage PL-SQL - Creation Tables.sql
01 - PRESENTATION\SCRIPTS PL-SQL\02 - Langage PL-SQL - Insertions.sql

Ci-dessous, le contenu des tables après les insertions d'enregistrements.

CODESPECIALITE	NOMSPECIALITE
CODESPECIALITE1	Libelle Specialite 1
CODESPECIALITE2	Libelle Specialite 2
CODESPECIALITE3	Libelle Specialite 3

NUMEROMED	NOM	PRENOM	ADRESSE	CODEPOSTAL	VILLE	CODESPECIALITE
NUMEROMED1	NOM1	PRENOM1	ADRESSE1	31100	VILLE1	CODESPECIALITE1
NUMEROMED2	NOM2	PRENOM2	ADRESSE2	31200	VILLE2	CODESPECIALITE2
NUMEROMED3	NOM3	PRENOM3	ADRESSE3	31200	VILLE3	CODESPECIALITE2
NUMEROMED4	NOM4	PRENOM4	ADRESSE4	31200	VILLE4	CODESPECIALITE3

NUMEROSS	NOM	PRENOM	ADRESSE	CODEPOSTAL	VILLE
164042561678250	NOM1	PRENOM1	ADRESSE1	31200	VILLE1
164042561678251	NOM2	PRENOM2	ADRESSE2	31200	VILLE2
164042561678252	NOM3	PRENOM3	ADRESSE3	31200	VILLE3
164042561678253	NOM4	PRENOM4	ADRESSE4	31200	VILLE4

CODETYPEVISITE	LIBELLE	PRIXVISITE
CODETYPEVISITE1	LIBELLE1	90
CODETYPEVISITE2	LIBELLE2	70
CODETYPEVISITE3	LIBELLE2	110
CODETYPEVISITE4	LIBELLE2	40

NUMEROVISITE	NUMEROMED	NUMEROSS	DATEVISITE	CODETYPEVISITE
NUMEROVISITE1	NUMEROMED1	164042561678250	31/05/98	CODETYPEVISITE1
NUMEROVISITE2	NUMEROMED2	164042561678250	31/07/98	CODETYPEVISITE2
NUMEROVISITE3	NUMEROMED3	164042561678250	31/07/98	CODETYPEVISITE2
NUMEROVISITE4	NUMEROMED3	164042561678251	31/07/98	CODETYPEVISITE2
NUMEROVISITE5	NUMEROMED3	164042561678252	31/07/98	CODETYPEVISITE2
NUMEROVISITE6	NUMEROMED3	164042561678253	31/07/98	CODETYPEVISITE2
NUMEROVISITE7	NUMEROMED4	164042561678253	31/07/98	CODETYPEVISITE4

3 PRESENTATION

PL/SQL, PROCEDURAL LANGUAGE with SQL, est un langage procédural d'Oracle corporation permettant d'étendre le langage SQL. Il permet donc de combiner les avantages d'un langage de programmation classique avec les possibilités de manipulation de données offertes par SQL.
Ce langage a fait sa première apparition en 1992, sous la forme d'une extension optionnelle d'Oracle 6. Depuis, PL/SQL est inclus par défaut dans toute nouvelle version d'Oracle.
Parmi les autres langages procéduraux, nous pouvons citer :

- **Transact SQL :** Sybase et Microsoft SQL Server,
- **Interactive SQL** : Interbase,
- **SQL/PSM (Persistent Stored Modules)** : MySQL.

Comme SQL, PL/SQL n'est pas sensible à la casse.

4 NOTION DE BLOC

La structure de base d'un programme PL/SQL est celle de **blocs**. Il existe 3 types de bloc :

- Anonyme
- Procédure
- Fonction

Ils ont les formes suivantes :

⊶ Syntaxe : bloc anonyme

DECLARE (facultatif)
/* **Section déclaration** */
Variables, curseurs, exceptions définies par l'utilisateur.
BEGIN (obligatoire)
/* **Section corps du bloc de programme** */
Instructions SQL
Instructions PL/SQL
EXCEPTION (facultatif)
/* **Section traitement des exceptions** */
Actions à effectuer lorsque des erreurs se produisent.
END; (obligatoire)
/ → nécessaire sous SQL*Plus pour lancer la compilation

⊶ Syntaxe : procédure

```
PROCEDURE  nom (obligatoire)
IS   (obligatoire)
    /* Section déclaration */
    Variables, curseurs, exceptions définies par l'utilisateur.
BEGIN (obligatoire)
    /* Section corps du bloc de programme */
    Instructions SQL
    Instructions PL/SQL
EXCEPTION (facultatif)
    /* Section traitement des exceptions */
    Actions à effectuer lorsque des erreurs se produisent.
END; (obligatoire)
/ → nécessaire sous SQL*Plus pour lancer la compilation
```

⊶ Syntaxe : fonction (retourne une valeur)

```
FUNCTION  nom (obligatoire)
RETURN typedonnée (obligatoire)
IS    (obligatoire)
    /* Section de déclaration */
    Variables, curseurs, exceptions définies par l'utilisateur.
BEGIN (obligatoire)
    /* Section corps du bloc de programme */
    Instructions SQL
    Instructions PL/SQL
    RETURN valeur ;   (obligatoire)
EXCEPTION (facultatif)
    /* Section traitement des exceptions */
    Actions à effectuer lorsque des erreurs se produisent.
END; (obligatoire)
/ → nécessaire sous SQL*Plus pour lancer la compilation
```

⚠ Pour les procédures et les fonctions, l'instruction DECLARE n'est plus présente.

Chaque bloc comporte au maximum trois sections :

- **Section déclaration** (facultative)
  - Elle contient la description des structures et des variables utilisées dans le bloc.
  - Elle commence par le mot clé DECLARE.
- **Section corps du bloc** (obligatoire)
  - Elle contient les instructions du programme et éventuellement, à la fin, la section de traitement des exceptions.
  - Elle commence par le mot clé BEGIN et se termine par le mot clé END.
- **Section traitement des exceptions** (facultative)
  - Elle commence par le mot clé EXCEPTION.

⌨ Exemple de bloc anonyme (sous SQL*Plus)

```
SET SERVEROUTPUT ON → nécessaire pour avoir l'affichage
DECLARE
    x VARCHAR2(20);
BEGIN
    x := 'Formation AFPA';
    DBMS_OUTPUT.PUT_LINE(x);
END;
/ → nécessaire sous SQL*Plus pour lancer la compilation
```

5 COMMENTAIRES

Pour ajouter des commentaires, vous avez 2 possibilités :

- Commentaire mono ligne (double tiret)
  -- Ceci est un commentaire mono ligne

- Commentaire multi lignes encadrer avec les symboles **/*** et ***/**
  /* et ceci est un commentaire multi lignes ***/**

6 PROCEDURES ET FONCTIONS

Il est possible de créer, au sein de la base de données, des procédures et des fonctions comme dans n'importe quel langage de programmation classique.
A noter que contrairement à une procédure, une fonction retourne une valeur.
Les procédures et les fonctions sont des objets de la base de données et ils appartiennent à un schéma (au même titre qu'une table ou une vue, par exemple). Elles sont souvent appelées **Procédures stockées.**
Elles sont soumises aux mécanismes de sécurité et de confidentialité.
En résumé, une procédure (ou fonction) est un programme PL/SQL nommé, compilé et stocké dans la base.
Les procédures et fonctions supportent assez bien la surcharge (Coexistence de procédures de même nom ayant des listes de paramètres différentes). C'est le système qui, au moment de l'appel, décidera, en fonction du nombre d'arguments et de leurs types, quelle est la bonne procédure à appeler.

⊶ Syntaxe : création d'une procédure

```
CREATE OR REPLACE PROCEDURE nom_procedure [(liste de paramètres)]
AS
    /* Section de déclaration */
BEGIN
    /* Section corps du bloc de programme */
EXCEPTION
    /* Section traitement des exceptions */
END;
/  → nécessaire sous SQL*Plus pour lancer la compilation
```

⊶ Syntaxe : création d'une fonction

```
CREATE OR REPLACE FUNCTION nom_fonction [ (liste de paramètres) ]
RETURN type  -- permet de spécifier le type de donnée retourné
AS
```

```
    /* Section de déclaration */
BEGIN
    /* Section corps du bloc de programme */
RETURN valeur ;
EXCEPTION
    /* Section traitement des exceptions */
END;
/ → nécessaire sous SQL*Plus pour lancer la compilation
```

⌨ Exemple : création d'une procédure

```
CREATE or REPLACE PROCEDURE ma_proc
IS
--  Déclaration d'une variable
    v_texte VARCHAR2(20);
BEGIN
--  Affectation de la chaine à la variable
v_texte := 'Formation AFPA';
--  Affichage de la chaine
DBMS_OUTPUT.PUT_LINE(v_texte);
END;
```

Une procédure est sensible à la notion de droit. Son créateur peut décider par exemple d'en permettre l'utilisation à d'autres utilisateurs. Il donne le droit d'exécution sur la procédure qui manipule les données, sans donner des droits sur les données.

⊶ Syntaxe : accorder droits d'exécution sur la procédure à un autre utilisateur

```
GRANT EXECUTE ON nom_procedure TO utilisateur|role
```

Il est possible d'appeler une procédure depuis n'importe quel bloc PL/SQL. Elle s'exécute directement au sein d'Oracle.
Il est possible de passer à une procédure ou à une fonction une liste de paramètres.

⊶ Syntaxe : déclaration d'un paramètre

```
nom_param [mode] datatype [ := valeur_defaut ]
```

Il y a trois modes pour passer les paramètres dans une procédure (ou une fonction) :

- **IN (lecture seule).** Le mode IN est réservé aux paramètres qui ne doivent pas être modifiés par la procédure. C'est le mode par défaut, si l'on ne le précise pas.
- **OUT (écriture seule).** Le mode OUT pour les paramètres transmis en résultat.
- **IN OUT (lecture et écriture).** Le mode IN OUT pour les variables dont la valeur peut être modifiée en sortie et consultée par la procédure.

Ces 3 modes permettent de déterminer comment la variable est utilisée au sein de la procédure.

Le type d'un paramètre ne supporte pas la précision (longueur).

```
CREATE PROCEDURE ma_procedure (id IN NUMBER(10), taux IN NUMBER(8) )
IS BEGIN
…..
END;

CREATE PROCEDURE ma_procedure (id OUT NUMBER, taux IN NUMBER)
IS BEGIN
…..
END;
```

6.1 TYPES DE VARIABLES

Les types de variables peuvent être répertoriés de la façon suivante :

- Variables scalaires
- Types composés : Record, collection, type objet
- Type référence (REF)
- Type LOB (Large Object)

Une variable est utilisable dans le bloc où elle a été définie ainsi que dans les blocs imbriqués dans le bloc de définition, sauf si elle est redéfinie dans un bloc interne.

6.1.1 CONVENTIONS

Vous trouverez ci-dessous, une liste de conventions concernant le nommage des variables.

- Déclarer au plus une variable par ligne.
- Les noms de variables doivent commencer par une lettre et être composés d'au plus 30 caractères (lettres, nombres ou caractères spéciaux).
- Deux variables peuvent avoir le même nom dès lors qu'elles appartiennent à des blocs différents.
- Ne pas utiliser les noms de tables ni les noms d'attributs
- Utiliser les suffixes ou les préfixes suivants :
  - '_c' ou 'c_' pour les constantes
  - '_v' ou 'v_' pour les variables
  - '_g' ou 'g_' pour les variables globales

6.1.2 VARIABLES SCALAIRES

Parmi les variables scalaires, nous retrouvons :

- Types issus de SQL : CHAR, NUMBER, DATE, VARCHAR2
- Types PL/SQL : BOOLEAN, SMALLINT, BINARY_INTEGER, DECIMAL, FLOAT,INTEGER, REAL, ROWID

CHAR [(<taille_max>)] :
chaînes de caractères de longueur fixe (max 32767)

VARCHAR2(<taille_max>) :
chaînes de caractères de longueur variable (max 32767)

NUMBER [(<p>, <s>)] :
nombres réels, p chiffres en tout, s après la virgule.

⊶ Syntaxe : déclaration d'une variable

```
nom_variable [CONSTANT] nom-du-type [NOT NULL] [(:= | DEFAULT)
expression];
```

🖮 Exemples de déclaration de variables et constantes

```
v_numero1 NUMBER(10) ;

-- non nulle, donc forcément initialisée
v_numero2 NUMBER NOT NULL := 10.5;

v_date DATE;

bstock BOOLEAN := true;

-- une constante, donc forcément initialisée
limite CONSTANT REAL := 400.00;
```

La contrainte NOT NULL doit être suivie d'une clause d'initialisation (Cf. exemple variable v_numero2).
Les déclarations multiples ne sont pas permises. Une seule déclaration par ligne. Vous ne pouvez donc pas écrire : variable1, variable2 NUMBER ;

6.1.3 TYPES COMPOSES

Un type composé est :

- Un type **RECORD** : Quand tous les attributs sont d'un type SQL, une variable de type record peut représenter une ligne d'une table relationnelle.
- Un type **COLLECTION** : TABLE de type NESTED TABLE, TABLE de type INDEX-BY et VARRAY.
- Un type **OBJET** : modèle relationnel objet.

Type **RECORD**
Deux étapes :

- Déclaration du type RECORD
- Déclaration de la variable de type RECORD

⊶ Syntaxe : déclaration d'un type RECORD

```
TYPE nom_type IS RECORD (
nom_attribut1 type_attribut1,
nom_attribut2 type_attribut2, ...);
```

⊶ Syntaxe : déclaration de la variable de type RECORD

```
nom_variable nom_type;
```

🖮 Exemple

```
DECLARE
Type t_medecin is record(
nom VARCHAR2(15),
prenom VARCHAR2(20)
    );
v_medecin t_medecin;
BEGIN
SELECT NOM, PRENOM
INTO v_medecin.nom, v_medecin.prenom
FROM MEDECIN WHERE NUMEROMED = 'NUMEROMED1';
DBMS_OUTPUT.PUT_LINE ('NOM DU MEDECIN : ' || v_medecin.nom);
DBMS_OUTPUT.PUT_LINE ('PRENOM DU MEDECIN : ' ||
v_medecin.prenom);
END;
```

Type **COLLECTION TABLE**

Une collection est un ensemble ordonné d'éléments de même type. Elle est indexée par une valeur de type numérique ou alphanumérique. Elle ne peut avoir qu'une seule dimension. Cependant, en créant des collections de collections vous pouvez obtenir des tableaux à plusieurs dimensions.

On peut distinguer trois types différents de collections :

- Les **tables (INDEX-BY TABLES)** qui peuvent être indicées par des variables numériques ou alphanumériques.
- Les **tables imbriquées (NESTED TABLES)** qui sont indicées par des variables numériques et peuvent être lues et écrites directement depuis les colonnes d'une table.
- Les **tableaux de type VARRAY**, indicés par des variables numériques, dont le nombre d'éléments maximum est fixé dès leur déclaration et peuvent être lus et écrits directement depuis les colonnes d'une table.

Deux étapes :

- Déclaration du type TABLE (TABLE NESTED ou TABLE INDEX-BY).
- Déclaration de la variable de type TABLE.

⊶ Syntaxe : déclaration d'un type TABLE NESTED

```
TYPE nom_type IS TABLE OF type_argument;
```

⊶ Syntaxe : déclaration d'un type TABLE INDEX-BY

```
TYPE nom_type IS TABLE OF type_argument INDEX BY index_by_type;
```

index_by_type représente l'un des types suivants :

- BINARY_INTEGER
- PLS_INTEGER
- VARCHAR2(taille)
- LONG

⊶ Syntaxe : déclaration d'une collection de type VARRAY

```
TYPE nom type IS VARRAY (taille maximum) OF type élément [NOT NULL] ;
```

⊶ Syntaxe : déclaration de la variable de type TABLE

```
nom_variable nom_type;
```

Exemple de type TABLE NESTED et TABLE INDEX-BY

```
DECLARE
TYPE t_nommedecin IS TABLE OF VARCHAR2(30) INDEX BY
BINARY_INTEGER;
TYPE t_numero IS TABLE OF number;
nom t_nommedecin;
loto t_numero;
i integer;
BEGIN
    nom(5):='Bob';
    i:=2;
    nom(i):='Steve';
    DBMS_OUTPUT.PUT_LINE ('NOM DU MEDECIN : ' || nom(5));
    DBMS_OUTPUT.PUT_LINE ('PRENOM DU MEDECIN : ' || nom(i));
    loto := t_numero(10, 24, 12, 23, 33, 32);
    DBMS_OUTPUT.PUT_LINE ('2EME NUMERO : ' || loto(2));
END;
```

Exemple de type VARRAY

```
DECLARE
 -- Collection de type VARRAY
TYPE TYP_VAR_TABLE is VARRAY(30) of varchar2(100) ;
tab TYP_VAR_ TABLE := TYP_VAR_ TABLE ("","","","","","","","","");
BEGIN
   FOR i in 1..10 LOOP
     tab(i):= to_char(i) ;
   END LOOP ;
END;
```

6.1.4 TYPES IMPLICITES

Lors de la déclaration de variables, il est possible d'utiliser des types implicites. Il existe 2 attributs permettant de spécifier des types implicites :

- L'attribut **%TYPE** signifie du même type (attribut) que.
- L'attribut **%ROWTYPE** signifie du même type de ligne (record) que.

L'attribut **%TYPE** est employé pour déclarer une variable en réutilisant :

- La définition d'un attribut d'une table ou d'une vue.
- La définition d'une autre variable déclarée précédemment.

⊶ Syntaxe

```
v_nomvariable nom_table.nom_attribut%TYPE;
```

🖮 Exemple : attribute %TYPE

```
DECLARE
    v_nom medecin.nom%TYPE;
BEGIN
    SELECT nom INTO v_nom
    FROM MEDECIN WHERE NumeroMed = 'NUMEROMED1';
    DBMS_OUTPUT.PUT_LINE (‘NOM DU MEDECIN : ‘ || v_nom);
END;
```

L’attribut **%ROWTYPE** est employé pour définir une variable de type enregistrement dont les champs correspondent à tous les attributs d’une table.

⊶ Syntaxe

```
v_nomvariable nom_table%ROWTYPE;
```

🖮 Exemple : attribute %ROWTYPE

```
DECLARE
    v_medecin medecin%ROWTYPE;
BEGIN
    SELECT * INTO v_medecin
    FROM MEDECIN WHERE NumeroMed = 'NUMEROMED1';
    DBMS_OUTPUT.PUT_LINE (‘NOM DU MEDECIN : ‘ || v_medecin.nom);
    DBMS_OUTPUT.PUT_LINE (‘PRENOM DU MEDECIN : ‘ ||
v_medecin.prenom);
    DBMS_OUTPUT.PUT_LINE (‘ADRESSE DU MEDECIN : ‘ ||
v_medecin.adresse || ‘ ‘ || v_medecin.codepostal || ‘ ‘ || v_medecin.ville);
END;
```

6.2 OPERATEURS

Pour une affectation simple, vous devez utiliser l’opérateur **:=**

🖮 Exemple : opérateur d'affectation

```
v_numero := 0;
v_numero := v_numero + 1;
```

Pour une affectation de valeurs issues d’une base de données, vous devez utiliser l’opérateur **INTO**.

⊶ Syntaxe : opérateur INTO

```
SELECT attribut, …
INTO liste de variables locales ou externes
FROM table
[WHERE condition]
```

Une telle requête SQL doit ramener obligatoirement une seule ligne. Si ce n’est pas le cas, les exceptions NO_DATA_FOUND ou TOO_MANY_ROWS sont levées.

Exemple : utilisation de l'opérateur INTO

```
DECLARE
    v_nom medecin.nom%TYPE;
    v_prenom medecin.prenom%TYPE;
BEGIN
    SELECT nom INTO v_nom
    FROM MEDECIN WHERE NumeroMed = 'NUMEROMED1';
    SELECT nom, prenom INTO v_nom, v_prenom
    FROM MEDECIN WHERE NumeroMed = 'NUMEROMED1';

    DBMS_OUTPUT.PUT_LINE ('NOM DU MEDECIN : ' || v_nom);
    DBMS_OUTPUT.PUT_LINE ('PRENOM DU MEDECIN : ' || v_prenom);
END;
```

Le tableau ci-dessous présente les différents types d'opérateur.

TYPE D'OPERATEUR	SYMBOLE	DESCRIPTION
Arithmétique	+	Addition
	-	Soustraction
	/	Division
	*	Multiplication
	**	Puissance
Concaténation	I I	Concaténation de 2 chaines de caractères.
Comparaison	=	Egal
	<	Strictement inférieur
	>	Strictement supérieur
	<=	Inférieur ou égal
	>=	Supérieur ou égal
	<>	Différent
	IS NULL	Retourne TRUE si l'opérande est NULL sinon FALSE.
	LIKE	Comparaison d'une variable et d'une chaine de caractères. Retourne TRUE si égalité sinon FALSE.
	BETWEEN	Test si une valeur se trouve dans un domaine de valeurs.
	IN	Test si une valeur se trouve dans une liste
Logique	AND	ET
	OR	OU
	NOT	Inverser la signification d'un opérateur

Exemple : opérateurs

```
DECLARE
v_nom VARCHAR(10);
v_nom2 VARCHAR(10);
v_numero integer;  -- par défaut, v_numero est NULL
BEGIN
```

```
v_nom := 'Bob';
v_nom2 := 'Steve';
v_numero := 10;
v_numero := v_numero + 10;
v_numero := v_numero - 1;
v_numero := v_numero * 4;
v_numero := v_numero / 2;

IF v_numero = 0 THEN
DBMS_OUTPUT.PUT_LINE ('v_numero est égal à 0');
END IF;

IF v_numero > 1 THEN
DBMS_OUTPUT.PUT_LINE ('v_numero est supérieur à 1');
END IF;

IF v_numero IS NULL THEN
DBMS_OUTPUT.PUT_LINE ('v_numero est NULL');
END IF;

IF v_numero BETWEEN 2 AND 44 THEN
DBMS_OUTPUT.PUT_LINE ('v_numero est entre 2 et 44 : ' || v_numero);
END IF;

IF v_nom LIKE 'Bob' THEN
DBMS_OUTPUT.PUT_LINE ('v_nom est Bob');
END IF;

IF v_nom2 NOT LIKE 'Bob' THEN
DBMS_OUTPUT.PUT_LINE ('v_nom2 n"est pas Bob');
END IF;

IF v_nom2 NOT LIKE 'Bob' AND v_nom LIKE 'Bob' THEN
DBMS_OUTPUT.PUT_LINE ('v_nom2 n »est pas Bob et v_nom est Bob');
END IF;

DBMS_OUTPUT.PUT_LINE ('NOM DU MEDECIN :  ' || v_nom);
END;
```

6.3 STRUCTURES DE CONTROLE

Concernant les structures de contrôle, nous pouvons identifier 2 catégories :

- Structures alternatives ou de branchement conditionnel
- Structures répétitives ou d'itération

6.4 STRUCTURES ALTERNATIVES

Ci-dessous, les différentes structures alternatives qui peuvent être créées.

⊶ Syntaxe : structure alternative

```
IF condition THEN
    instructions;
END IF;
IF condition THEN
    instructions;
ELSE
    instructions;
END IF;
IF condition THEN
    instructions;
ELSIF condition THEN
    instructions;
ELSE  -- else facultatif
    instructions;
END IF;
```

Exemple : structure alternative

```
DECLARE
v_numero integer;
BEGIN
v_numero := 10;
IF v_numero > 1 THEN
DBMS_OUTPUT.PUT_LINE ('v_numero est supérieur à 1');
END IF;
IF v_numero BETWEEN 2 AND 44 THEN
DBMS_OUTPUT.PUT_LINE ('v_numero est entre 2 et 44 ' );
ELSE
    DBMS_OUTPUT.PUT_LINE ('v_numero  n'est pas entre 2 et 44 ' );
END IF;
IF v_numero BETWEEN 0 AND 44 THEN
DBMS_OUTPUT.PUT_LINE ('v_numero est entre 0 et 44 ' );
ELSIF v_numero BETWEEN 45 AND 100 THEN
    DBMS_OUTPUT.PUT_LINE ('v_numero  est entre 45 et 100 ' );
    ELSE
    DBMS_OUTPUT.PUT_LINE ('v_numero  est supérieur à 100 ' );
END IF;
END;
```

6.5 STRUCTURES REPETITIVES

Ci-dessous, les différentes structures répétitives qui peuvent être créées.

Syntaxe : boucle POUR

```
FOR variable_indice IN [REVERSE] val_debut … val_fin LOOP
    instructions;
END LOOP;
```

variable_indice : variable locale à la boucle non déclarée.

val_debut et *val_fin* : variables locales déclarées et initialisées ou des constantes. Si REVERSE est présent, le pas est de -1, sinon il est égal à +1.

⊶ Syntaxe : boucle TANT QUE

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

⊶ Syntaxe : boucle REPETER JUSQU'A

```
LOOP
    instructions;
END LOOP;

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

LOOP
    instructions;
    IF condition THEN EXIT;
    END IF;
    instructions;
END LOOP;
```

⊶ Syntaxe : CAS QUAND ALORS

```
CASE
    instructions;
    WHEN value1 THEN expression1;
    WHEN value2 THEN expression2;
    ....
ELSE expressionpardefaut;          -- ELSE facultatif
END CASE;
```

Exemple

```
DECLARE
v_indice integer;
v_CodeTypeVisite typevisite.CodeTypeVisite%TYPE;
BEGIN
v_indice := 1;
CURSOR c1 IS SELECT CodeTypeVisite FROM typevisite;

    -- Boucle FOR
    FOR indice  IN 1..30 LOOP
        DBMS_OUTPUT.PUT_LINE ('BOUCLE FOR -> Valeur de l"indice : ' ||
         indice);
    END LOOP;
```

```
    -- Boucle WHILE
    WHILE v_indice < 31 LOOP
          DBMS_OUTPUT.PUT_LINE ('BOUCLE WHILE -> Valeur de l"indice : '
                || v_indice);
          v_indice := v_indice+1;
    END LOOP;

    -- Boucle WHILE
     v_indice := 1;
    LOOP
        EXIT WHEN v_indice = 31;
         DBMS_OUTPUT.PUT_LINE ('BOUCLE LOOP -> Valeur de l »indice : '
         || v_indice);
         v_indice := v_indice+1;
    END LOOP;

--  Boucle CASE… WHEN
OPEN c1;
LOOP
FETCH c1 INTO v_CodeTypeVisite;
EXIT WHEN c1%NOTFOUND;
CASE v_CodeTypeVisite
WHEN 'CODETYPEVISITE1' THEN DBMS_OUTPUT.PUT_LINE ('Type de
visite tres honereuse : ' || v_CodeTypeVisite);
WHEN 'CODETYPEVISITE2' THEN DBMS_OUTPUT.PUT_LINE ('Type de
visite honereuse : ' || v_CodeTypeVisite);
WHEN 'CODETYPEVISITE3' THEN DBMS_OUTPUT.PUT_LINE ('Type de
visite peu honereuse :' || v_CodeTypeVisite);
ELSE DBMS_OUTPUT.PUT_LINE ('Type de visite honereuse : ' ||
v_CodeTypeVisite);
    END CASE;
END LOOP;
CLOSE c1;
END;
```

6.6 LES EXCEPTIONS

En PL/SQL, la gestion des erreurs se fait grâce aux exceptions. Il existe un certain nombre d'exceptions prédéfinies mais vous pouvez définir vos propres exceptions.

6.6.1 TRAITEMENT D'UNE EXCEPTION

Le traitement de l'exception se fait dans la partie EXCEPTION du bloc PL/SQL.

⊶ Syntaxe : bloc EXCEPTION

```
BEGIN
--  Corps du bloc
EXCEPTION
WHEN exception1 [or exception2 ...] THEN
```

```
    instructions ;
WHEN exception3 [or exception4 ...] THEN
    instructions ;
...
[WHEN OTHERS THEN instructions ;]
END;
```

Si une exception se produit dans le corps du bloc, l'exécution de ce corps est interrompue et la partie EXCEPTION du bloc est exécutée. Dans cette partie, vous pouvez déclencher de nouveau l'exception que vous venez de traiter grâce à l'instruction RAISE.
Si l'exception existe dans une clause WHEN, alors les instructions de cette clause WHEN sont exécutées et le programme est terminé.
Si l'exception n'existe pas dans une clause WHEN :

- Soit il existe une clause WHEN OTHERS et dans ce cas les instructions de cette clause sont exécutées et le programme est terminé.
- Soit il n'y a pas de clause WHEN OTHERS et l'exception est propagée au bloc englobant ou au programme appelant.

Si aucun traitement d'exception n'est rencontré, la transaction qui a déclenché l'exception est annulée.

Exemple

```
DECLARE
v_codetypevisite VARCHAR2(20);
BEGIN
    SELECT codetypevisite INTO v_codetypevisite
    FROM visite
    WHERE NumeroMed = 'NUMEROMED3';
EXCEPTION
WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE ('PLUS D'UNE LIGNE SELECTIONNEE');
END;
```

6.6.2 DECLENCHEMENT D'UNE EXCEPTION

Le déclenchement d'une exception nomException se fait à l'aide de l'instruction suivante : **RAISE nomException;**
La déclaration d'une exception nomException s'effectue grâce à l'instruction suivante : **nomException EXCEPTION**;

Syntaxe : déclenchement d'une exception

```
DECLARE
  exception EXCEPTION ;
BEGIN
  RAISE exception;
END;
```

Exemple : déclenchement d'une exception

```
DECLARE
  exceptionval EXCEPTION ;
BEGIN
  IF val > 20 THEN
     RAISE exceptionval;
  END IF;
EXCEPTION
   WHEN exceptionval THEN
   DBMS_OUTPUT.PUT_LINE(‘ VAL > 20’);
END;
```

6.6.3 EXCEPTIONS PREDEFINIES

En PL/SQL, il existe des exceptions prédéfinies. Ces exceptions prédéfinies sont associées à des codes d'erreur Oracle.
Le tableau-ci-dessous présente quelques exceptions prédéfinies.

NOM	ERREUR ORACLE	VALEUR SQLCODE	DESCRIPTION
TOO_MANY_ROWS	ORA-01422	-1422	Instruction select ... into qui ramène plus d'une ligne.
INVALID_CURSOR	ORA-01001	-1001	Ouverture de curseur non valide.
VALUE_ERROR	ORA-06502	-6502	Erreur arithmétique (conversion, taille, ...) pour un NUMBER
ZERO_DIVIDE	ORA-01476	-1476	Division par 0
LOGIN_DENIED	ORA-01017	-1017	Connexion refusée
NO_DATA_FOUND	ORA-01403	+100	Instruction select ... into qui ne ramène aucune ligne.
CURSOR_ALREADY_OPEN	ORA-06511	-6511	Ouverture d'un curseur déjà ouvert.
STORAGE_ERROR	ORA-06500	-6500	Dépassement de capacité mémoire.

Par l'intermédiaire de **SQLCODE**, il est possible de récupérer le code numérique de l'exception qui vient d'être levée.
De même, par l'intermédiaire de **SQLERRM**, il est possible de récupérer le message correspondant à l'exception qui vient d'être levée.

Exemple : traitement de l'exception prédéfinie TOO_MANY_ROWS

```
DECLARE
    v_codetype VARCHAR2(25);
BEGIN
    DBMS_OUTPUT.enable;
    SELECT codetypevisite INTO v_codetype FROM visite WHERE
NumeroMed =
    ‘NUMEROMED3’;
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE (‘Code : ‘ || TO_CHAR(SQLCODE)) ;
    DBMS_OUTPUT.PUT_LINE (‘Message : ‘ || SQLERRM);
END;
```

6.6.4 LES EXCEPTIONS ET LES CODES D'ERREURS

Lorsqu'une exception n'est pas traitée dans le programme PL/SQL, le client qui a appelé ce programme reçoit le code d'erreur associé. Tous les codes d'erreur Oracle ne sont pas associés à un nom d'exception. Pour traiter une erreur non liée à une exception, vous pouvez utiliser la clause WHEN OTHERS mais vous ne savez pas exactement quelle erreur vous traitez.
Une autre solution consiste à lier un nom d'exception à un code d'erreur. Ceci est réalisé par l'intermédiaire de la directive suivante :
PRAGMA EXCEPTION_INIT (nomException, -codeErreurOracle) ;
PRAGMA est une directive de compilation qui est traitée au moment de la compilation.
Lier une exception à un code d'erreur peut aussi être utilisé pour définir des nouveaux codes d'erreur à partir de nouvelles exceptions.

Exemple : entre les tables MEDECIN et SPECIALITE, il existe une contrainte de clé étrangère (un médecin a une spécialité). Lorsqu'on veut supprimer une spécialité qui est rattachée à au moins un médecin, on obtient l'erreur : ORA-02292: violation de contrainte (HR.FK_MEDECIN_SPECIALITE) d'intégrité - enregistrement fils existant.

Exemple

```
CREATE or REPLACE PROCEDURE supprimerSpecialite
IS
BEGIN
DELETE FROM SPECIALITE WHERE CodeSpecialite =
'CODESPECIALITE1';
END;
EXECUTE supprimerSpecialite;
→ ORA-02292
```

La procédure suivante permet de gérer cette erreur en la liant à un nom d'exception. Vous remarquez que le code de l'exception est **-2292** et non **2292**.

Exemple

```
CREATE or REPLACE PROCEDURE supprimerSpecialiteEx
IS
    ex_resteMedecin EXCEPTION;
    PRAGMA EXCEPTION_INIT(ex_resteMedecin, -2292);
BEGIN
    DELETE FROM SPECIALITE WHERE CodeSpecialite =
'CODESPECIALITE1';
EXCEPTION
WHEN ex_resteMedecin THEN
DBMS_OUTPUT.PUT_LINE ('SUPPRESSION IMPOSSIBLE DU CODE
SPECIALITE');
END;
EXECUTE supprimerSpecialiteEx;
```

Dans la procédure ci-dessus, si aucune ligne de la table SPECIALITE n'est effacée, ceci n'entraine pas d'erreur.

La procédure ci-dessous permet de déclencher une exception si la spécialité passée en paramètre n'existe pas.

Exemple

```
CREATE or REPLACE PROCEDURE supprimerSpecialiteEx1(ma_specia
VARCHAR2)
IS
resteMedecin EXCEPTION;
PRAGMA EXCEPTION_INIT (resteMedecin, -2292);
specialiteInexistante EXCEPTION;  -- ce n'est pas nécessaire de la lier à un
code d'erreur

BEGIN
DELETE FROM SPECIALITE WHERE CodeSpecialite = ma_specia;
IF SQL%NOTFOUND THEN
RAISE specialiteInexistante;  -- déclenchement de l'exception
END IF;
EXCEPTION
    WHEN resteMedecin THEN
DBMS_OUTPUT.PUT_LINE ('SUPPRESSION IMPOSSIBLE DU CODE
SPECIALITE');
WHEN specialiteInexistante THEN
DBMS_OUTPUT.PUT_LINE ('CODE SPECIALITE N"EXISTE PAS');
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ('AUTRE ERREUR');
END;

--   Erreur car 'CODESPECIALITE1' est rattaché à un médecin
EXECUTE supprimerSpecialiteEx1('CODESPECIALITE1');
--   Erreur car 'CODESPECIALITE12' n'existe pas
EXECUTE supprimerSpecialiteEx1('CODESPECIALITE12');
```

6.6.5 DEFINITION DE SES PROPRES EXCEPTIONS

Il est aussi possible de définir vos propres exceptions de type ORA-. Pour ce faire, vous devez utiliser l'instruction RAISE_APPLICATION_ERROR.

Syntaxe

```
RAISE_APPLICATION_ERROR(CodeErreur, MessageErreur
[,{TRUE|FALSE}]);
```

où

- CodeErreur est un entier négatif compris entre -20999 et -20000. Ces valeurs sont réservées aux erreurs non prédéfinies.
- MessageErreur est le message d'erreur que vous désirez communiquer au client. Taille maximale est de 2048 bytes long.
- TRUE = erreur est placée dans la pile des erreurs. FALSE = erreur remplace toutes les erreurs précédentes (valeur par défaut).

Exemple : instruction RAISE_APPLICATION_ERROR

```
BEGIN
    DELETE FROM SPECIALITE WHERE CodeSpecialite =
'CODESPECIALITE12';
    IF SQL%NOTFOUND THEN
        RAISE_APPLICATION_ERROR  (-20101, 'Suppression impossible');
END IF;
END;
```

La procédure RAISE_APPLICATION_ERROR fait partie du package DBMS_STANDARD.

6.6.6 QUELQUES ASTUCES

ASTUCE 1 : Pour poursuivre le traitement après qu'une exception soit levée, vous pouvez définir un sous bloc.

Exemple

```
DECLARE
  BEGIN
    -- Sous bloc
    BEGIN
      DBMS_OUTPUT.PUT_LINE ('AVANT DELETE');
      DELETE FROM SPECIALITE WHERE CodeSpecialite =
'CODESPECIALITE1';
    EXCEPTION
      WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('TRAITEMENT EXCEPTION');
    -- Fin sous bloc
    END;
    DBMS_OUTPUT.PUT_LINE ('TRAITEMENT SUIVANT');
END;
```

ASTUCE 2 : Utilisation d'une variable pour localiser l'exception.
L'utilisation d'un traitement d'exceptions pour une séquence de requêtes, telles que INSERT, DELETE, UPDATE ou SELECT, peut masquer la requête qui est à l'origine de l'erreur.
Si vous avez besoin de connaître quelle requête échoue, vous pouvez utiliser une variable qui permet de localiser la requête à l'origine de l'exception.

Exemple

```
DECLARE
no_stmt INTEGER;
name VARCHAR2(100);
BEGIN
    no_stmt:= 1;  -- Désigne le 1er SELECT
SELECT table_name INTO name FROM user_tables WHERE table_name
LIKE 'MEDEC%';
    no_stmt:= 2;  -- Désigne le 2ème SELECT
```

```
SELECT table_name INTO name FROM user_tables WHERE table_name
LIKE ‘MEDES%’;
EXCEPTION
WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE (‘NOM DE TABLE NON TROUVEE DANS LA
REQUETE N° : ‘ || no_stmt);
END;
```

ASTUCE 3 : Essayer de nouveau une transaction.
Après qu’une exception soit levée, soit vous abandonnez la transaction, soit vous effectuez une nouvelle tentative. La technique est la suivante :

- Mettre la transaction dans un sous bloc
- Placer le sous bloc à l’intérieur d’une boucle que répète dans la transaction
- Avant de démarrer la transaction, placer un savepoint. Si la transaction réussie, effectuez un commit et quittez la boucle. Si la transaction échoue, le contrôle est transféré au handler d’exception, dans lequel vous effectuez un rollback à partir du savepoint, puis essayez de fixer le problème.

Dans l’exemple suivant, la commande INSERT peut lever une exception en raison d’une valeur dupliquée dans une colonne unique. Dans ce cas, vous changez la valeur qui doit être unique et continuez la boucle. Si la commande INSERT échoue, vous sortez immédiatement de la boucle. Avec cette technique, vous pouvez utiliser une bouche FOR ou WHILE pour limiter le nombre de tentatives.

Exemple

```
DECLARE
   numeromed VARCHAR2(12) := 'NUMEROMED4';
   nom VARCHAR2(25) :=  'NOM4';
   prenom VARCHAR2(25) := 'PRENOM4';
   adresse VARCHAR2(25) := 'ADRESSE4';
   codepostal NUMBER(5,0) := 31200;
   ville CHAR(25) := 'VILLE4';
   codespecialite CHAR(25) := 'CODESPECIALITE3';
   nbvisites NUMBER(5,0) := 0;
   suffix NUMBER := 1;
BEGIN
-- Essayer 10 fois
FOR i IN 1..10 LOOP
  -- Sous bloc BEGIN
  BEGIN
    -- Placer un savepoint
    SAVEPOINT start_transaction;
    -- Supprimer un enregistrement de la table TYPEVISITE
    DELETE FROM TYPEVISITE WHERE CODETYPEVISITE =
'CODETYPEVISITE5';
    -- Ajouter un medecin
```

```
    INSERT INTO MEDECIN (NUMEROMED, NOM, PRENOM, ADRESSE,
CODEPOSTAL, VILLE, CODESPECIALITE, NBVISITES)
VALUES (numeromed, nom, prenom, adresse, codepostal, ville,
codespecialite,nbvisites);
    --        raises DUP_VAL_ON_INDEX Si 2 medecins ont le meme numero
    -- Si ok alors valider les modifications
    COMMIT;
    -- Quitter la boucle
    EXIT;
    EXCEPTION
       WHEN DUP_VAL_ON_INDEX  THEN
            -- Annuler les modifications notamment le DELETE
            ROLLBACK TO start_transaction;
            -- Ajouter un suffixe pour essayer de résoudre le probleme
            suffix := suffix + 1;
             DBMS_OUTPUT.PUT_LINE ('Nom du médecin : ' || suffix);
            numeromed := numeromed || TO_CHAR(suffix);
            DBMS_OUTPUT.PUT_LINE ('Nom du médecin : ' || numeromed);
    -- Fin du sous bloc
    END;
END LOOP;
END;
```

6.7 COMPILATION D'UNE PROCEDURE/FONCTION

Si le schéma de la base évolue, par exemple par la suppression ou la modification de tables, il faut recompiler les procédures existantes pour qu'elles tiennent compte de ces modifications.

Compilation d'une procédure ou d'une fonction

```
ALTER {FUNCTION|PROCEDURE} [schema.]nom_procedure COMPILE ;
```

6.8 SUPPRESSION D'UNE PROCEDURE/FONCTION

Vous devez gérer la dépendance entre les procédures/fonctions. Il est important de savoir qu'une procédure appelant une procédure inexistante (détruite) devient invalide (décompilée).

⊶ Syntaxe : supprimer une procédure

```
DROP PROCEDURE nom_procedure;
```

⊶ Syntaxe : supprimer une fonction

```
DROP FUNCTION nom_fonction;
```

6.9 APPEL D'UNE PROCEDURE/FONCTION

Une procédure peut être appelée à partir :

- D'un bloc PL/SQL
- De SQL*Plus
- D'une requête

⌦ Appel dans un bloc SQL

⊶ Syntaxe : appel d'une procédure

```
nom_procedure ;                          -- sans paramètre
nom_procedure (liste_paramètres);  -- avec paramètres
```

⊶ Syntaxe : appel d'une fonction

```
valeur_retournee := nom_fonction(liste_paramètres);
```

⌦ Appel à partir de SQL*Plus ou dans un script SQL Developer

⊶ Syntaxe : appel d'une procédure

```
EXECUTE nom_procedure (liste_paramètres);
ou
BEGIN
nom_procedure (liste_paramètres);
END ;
```

⊶ Syntaxe : appel d'une fonction

```
BEGIN
valeur_retournee := nom_fonction(liste_paramètres);
END ;
```

⌦ Appel dans une requête

⊶ Syntaxe : appel d'une fonction

```
SELECT NumeroMed, Nom, ma_fonction (NumeroMed)
FROM medecin;
```

6.10 QUELQUES ASTUCES

Si vous utilisez SQL*Plus pour créer une procédure ou une fonction, il faut la faire suivre du caractère / sur la ligne suivante ; ce caractère correspond à la directive de compilation. Pour voir les erreurs de compilation, vous pouvez utiliser **SHOW ERRORS**.

Pour tester une procédure ou une fonction, vous pouvez l'exécuter sous SQL*Plus ou dans l'environnement SQL Developer.

Une fonction qui ne comporte pas d'instruction modifiant la base de données peut s'exécuter avec une pseudo requête.

Permet de tester la fonction ma_fonc1 avec une pseudo-requête

```
-- Possible car pas de modification de la base
SELECT ma_fonc1 FROM dual;
```

Sous SQL*Plus, pour que l'affichage soit effectif (dbms_output.put_line), il faut positionner la variable d'environnement **SERVEROUTPUT** de la façon suivante :

```
SET SERVEROUTPUT ON
```

Ceci n'est à faire qu'une seule fois dans la session.

Sous SQL*Plus, vous pouvez utiliser des variables globales.

Exemple : création de la procédure

```
CREATE or REPLACE PROCEDURE ObtenirNomMedecin(p_code
VARCHAR2, p_nom OUT VARCHAR2)
IS
    medecinInexistant EXCEPTION;
BEGIN
SELECT nom INTO p_nom FROM MEDECIN WHERE NumeroMed = p_code;
IF SQL%NOTFOUND THEN
RAISE medecinInexistant;
END IF;
EXCEPTION
WHEN medecinInexistant THEN
DBMS_OUTPUT.PUT_LINE ('CODE MEDECIN N"EXISTE PAS');
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ('AUTRE ERREUR');
END;
```

Exemple d'utilisation d'une variable globale

```
SQL*Plus > VARIABLE le_nom VARCHAR2(20);
SQL*Plus >EXECUTE ObtenirNomMedecin ('NUMEROMED1', :le_nom);
SQL*Plus >PRINT le_nom ;
```

Ces variables globales sont propres à SQL*Plus, elles sont utilisables en PL/SQL en les préfixant du caractère ' :'
Les instructions EXECUTE et PRINT sont aussi réservées à SQL*Plus.
Il est possible de passer en paramètres d'entrée d'un bloc PL/SQL des variables définies sous SQL*PLUS. Vous accédez aux valeurs d'une telle variable dans le code PL/SQL en faisant préfixer le nom de la variable par un caractère « & ».
La directive ACCEPT permet de faire une lecture de la variable au clavier.

Exemple : directive ACCEPT

```
SQL*Plus > ACCEPT s_codemed PROMPT 'Entrer code medecin : '
SQL*Plus > DECLARE
SQL*Plus > v_nom medecin.nom%TYPE;
SQL*Plus > BEGIN
SQL*Plus > SELECT nom  INTO v_nom, FROM medecin WHERE numeromed
= '&s_codemed ';
SQL*Plus > DBMS_OUTPUT.PUT_LINE('Nom du medecin : '  v_nom);
SQL*Plus > END;
SQL*Plus > /
```

7 CURSEURS

Oracle crée des zones de travail pour exécuter les ordres SQL, stocker leurs résultats et les utiliser. Ces zones sont appelées des CURSEURS.
Il existe 2 types de curseurs :

- **Implicite** : il est créé automatiquement par Oracle pour ses propres traitements. Il est cependant possible de manipuler certaines informations liées à son exécution grâce aux attributs de curseur.
- **Explicite** : il est créé par l'utilisateur pour pouvoir traiter le résultat d'une requête SELECT retournant plusieurs lignes. Les informations liées à son exécution sont disponibles grâce aux attributs de curseur.

Les curseurs explicites sont donc utilisés quand une requête retourne plus d'une ligne. Dans ce cas, le résultat doit être placé dans un curseur.
Pour utiliser un curseur explicite, il faut respecter les étapes suivantes :

- Déclaration (DECLARE)
- Ouverture (OPEN)
- Utilisation (FETCH ou FOR)
- Fermeture (CLOSE)

7.1 DECLARATION D'UN CURSEUR

⊶ Syntaxe

```
CURSOR nom_curseur IS ordre_select;
```

🖮 Exemple : déclaration d'un curseur c1 qui permet de récupérer tous les enregistrements de la table MEDECIN

```
CURSOR c1 IS SELECT * FROM MEDECIN;
```

La déclaration du curseur doit se faire dans la section de déclaration (DECLARE) d'un bloc PL/SQL.
Il est cependant possible de ne pas déclarer le curseur dans cette section mais de le définir directement dans l'instruction FOR.

🖮 Exemple : déclaration du curseur dans un FOR

```
FOR variable IN (SELECT * FROM MEDECIN)
```

7.2 OUVERTURE D'UN CURSEUR

Cette commande déclenche :

- L'allocation mémoire du curseur.
- L'analyse syntaxique et sémantique de l'ordre SELECT.
- Le positionnement de verrous éventuels (si SELECT … FOR UPDATE).
- L'exécution de la requête associée.

L'ouverture du curseur doit se faire dans le corps du bloc PL/SQL

⊶ Syntaxe

```
OPEN nom_curseur;
```

🖮 Exemple : création d'un curseur nommé **c1**

```
DECLARE
CURSOR c1 IS SELECT CodeTypeVisite FROM type_visite WHERE
PrixVisite < 30;
BEGIN
OPEN c1;
```

```
...
END;
```

7.3 UTILISATION D'UN CURSEUR : FETCH

La commande **FETCH** permet de parcourir une à une les lignes retournées par l'ouverture du curseur.
Cette commande ne ramène qu'une seule ligne, il faut donc la mettre dans une boucle pour récupérer l'ensemble des enregistrements.

⊶ Syntaxe

```
FETCH nom_curseur INTO liste_variable;
```

🖮 Exemple : récupérer une ligne et la mémoriser dans la variable v_medecin

```
FETCH  c1 INTO v_medecin;
```

7.4 UTILISATION D'UN CURSEUR : FOR

Oracle propose une variante de la boucle **FOR** qui:

- déclare implicitement la variable de parcours du curseur,
- ouvre le curseur,
- réalise les FETCH successifs,
- ferme le curseur.

⊶ Syntaxe : boucle FOR

```
FOR variable IN cursor LOOP
…..
END LOOP;
```

🖮 Exemple : boucle FOR avec déclaration du curseur

```
DECLARE
v_medecin medecin%ROWTYPE;
CURSOR c1 IS SELECT * FROM medecin;
BEGIN
FOR v_medecin IN c1 LOOP
DBMS_OUTPUT.PUT_LINE ('Nom du médecin : ' || v_medecin.prenom || ' '||
v_medecin.nom);
END LOOP;
END;
```

🖮 Exemple : boucle FOR sans déclaration du curseur

```
DECLARE
v_medecin medecin%ROWTYPE;
BEGIN
FOR v_medecin IN  (SELECT * FROM medecin) LOOP
DBMS_OUTPUT.PUT_LINE ('Nom du médecin : ' || v_medecin.prenom || ' '||
v_medecin.nom);
END LOOP;
END;
```

7.5 FERMETURE D'UN CURSEUR

La fermeture du curseur permet de libérer l'espace mémoire.

⊶ Syntaxe

```
CLOSE nom_curseur;
```

🖮 Exemple : fermeture du curseur c1

```
CLOSE c1;
```

🖮 Exemple : utilisation d'un curseur

```
DECLARE
v_medecin medecin.nom%TYPE;
CURSOR c1 IS SELECT nom FROM medecin;
BEGIN
    OPEN c1;
    LOOP
    FETCH c1 INTO  v_medecin;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE ('Nom du médecin : ' || v_medecin );
    END LOOP;
    CLOSE c1;
END;
```

A l'aide de l'attribut %ISOPEN, vous pouvez tester si un curseur est ouvert avant de le fermer.

⊶ Syntaxe

```
IF nom_curseur % ISOPEN THEN
    CLOSE nom_curseur;
ENDIF
```

7.6 MODIFICATION DES VALEURS D'UN CURSEUR

Lorsqu'une ligne stockée dans un curseur est lue par un FETCH, il est possible de mettre à jour directement cette ligne dans la table. Pour cela, il faut utiliser **FOR UPDATE OF** et **CURRENT OF**.

La clause **CURRENT OF** permet d'accéder directement en modification ou en suppression à la ligne que vient de ramener l'ordre FETCH.

Il faut au préalable réserver les lignes lors de la déclaration du curseur par un verrou d'intention (**FOR UPDATE** OF nom_colonne). Avec ce verrou, les données ne peuvent pas être modifiées par un autre utilisateur, entre le moment où les données sont lues et le moment où les données sont modifiées.

🖮 Exemple

```
SELECT * FROM typevisite;
DECLARE
    CURSOR c1 IS SELECT prixvisite FROM typevisite FOR UPDATE OF
prixvisite;
nb_update NUMBER :=0;
```

```
BEGIN
FOR v_typevisite IN c1
LOOP
IF v_typevisite.prixvisite < 80 THEN
UPDATE typevisite SET prixvisite = prixvisite * 1.10
WHERE CURRENT OF c1;
nb_update := nb_update + 1;
END IF;
END LOOP;
DBMS_OUTPUT.PUT_LINE ('Nombre de prix modifiés : ' || nb_update);
END;
```

7.7 ATTRIBUTS D'UN CURSEUR

Les attributs d'un curseur (implicite ou explicite) permettent d'évaluer son état et d'obtenir des informations relatives aux données retournées.
Le tableau-ci-dessous présente les différents attributs d'un curseur implicite ou explicite.

⊶ Syntaxe

```
-- Curseur explicite
nomcurseur%attribut

-- Curseur implicite
SQL%attribut
```

ATTRIBUT	CURSEUR IMPLICITE	VALEUR
%FOUND	SQL%FOUND	TRUE si la dernière instruction INSERT, UPDATE ou DELETE a traité au moins 1 ligne TRUE si le dernier SELECT … INTO a ramené une et une seule ligne.
%NOTFOUND	SQL%NOTFOUND	TRUE si la dernière instruction INSERT, UPDATE ou DELETE n'a traité aucune ligne. TRUE si le dernier SELECT … INTO n'a pas ramené de ligne.
%ISOPEN	SQL%ISOPEN	Toujours à FALSE car ORACLE ferme les curseurs après utilisation.
%ROWCOUNT	SQL%ROWCOUNT	Nombre de lignes traitées par le dernier ordre SQL (INSERT, UPDATE ou DELETE). 0 si le dernier SELECT… INTO n'a ramené aucune ligne. 1 si le dernier SELECT… INTO a ramené exactement 1 ligne. 2 si le dernier SELECT… INTO a ramené plus d'1 ligne.

ATTRIBUT	CURSEUR EXPLICITE	VALEUR
%FOUND	NomCurseur%FOUND	TRUE si le dernier FETCH a ramené une ligne
%NOTFOUND	NomCurseur%NOTFOUND	TRUE si le dernier FETCH n'a pas ramené une ligne
%ISOPEN	NomCurseur%ISOPEN	TRUE si le curseur est ouvert
%ROWCOUNT	NomCurseur%ROWCOUNT	Nombre de lignes traitées par l'ordre SQL. Evolution de l'attribut à chaque ligne traitée par un **FETCH** (zéro au départ).

🖮 Exemple : curseur implicite

```
INSERT INTO MEDECIN (NUMEROMED, NOM, PRENOM, ADRESSE,
CODEPOSTAL, VILLE, CODESPECIALITE)  VALUES ('CURSMED1', 'NOM1',
'PRENOM1', 'ADRESSE1', 31100, 'VILLE1', 'CODESPECIALITE1');
BEGIN
DELETE FROM MEDECIN
WHERE numeromed like 'CURS%';

DBMS_OUTPUT.PUT_LINE ('NOMBRE DE MEDECINS SUPPRIMES : ' ||
SQL%ROWCOUNT);
IF SQL%FOUND THEN
DBMS_OUTPUT.PUT_LINE ('MEDECINS SUPPRIMES' ||
SQL%ROWCOUNT);
END IF;
IF SQL%NOTFOUND THEN
DBMS_OUTPUT.PUT_LINE ('AUCUN MEDECIN SUPPRIME' );
END IF;
--   Toujours à FALSE pour un curseur implicite
IF SQL%ISOPEN THEN
DBMS_OUTPUT.PUT_LINE ('CURSEUR OUVERT');
END IF;
END;
```

⌨ Exemple : curseur explicite

```
DECLARE
v_medecin medecin.nom%TYPE;
CURSOR c1 IS SELECT nom FROM medecin;
BEGIN
    OPEN c1;
LOOP
FETCH c1 INTO v_medecin;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE ('Nom du medecin : ' || v_medecin );
    END LOOP;
    DBMS_OUTPUT.PUT_LINE ('Nombre de medecin : ' || c1%ROWCOUNT);
    IF c1%ISOPEN THEN
    CLOSE c1;
    END IF;
END;
```

8 TRIGGERS (DECLENCHEURS)

Un trigger est une action stockée dans la base de données et qui s'exécute suite à un événement. La suite d'instructions que constitue cette action est exécutée automatiquement par le système de gestion de base de données, chaque fois que l'événement auquel elle est associée se produit.
L'exécution d'un déclencheur est un succès ou un échec. En cas d'échec, l'exécution du traitement est stoppée, mais la transaction qui l'a appelé peut soit continuer soit être annulée.
Dans Oracle, un trigger peut être de plusieurs types :

- Trigger LMD (Langage de Manipulation des Données) sur une table.

- Trigger LDD (Langage de Définition des Données) sur un schéma ou une base de données.
- Trigger INSTEAD OF sur une vue.
- Trigger événements systèmes sur la base de données.

Un trigger peut provoquer le déclenchement d'un autre trigger (triggers en cascade). A priori, il est possible d'avoir jusqu'à 32 triggers en cascade.
Un trigger peut être activé ou désactivé. S'il est désactivé, ORACLE le stocke mais l'ignore. Par défaut, un trigger est activé dès sa création.

⊶ Syntaxe : désactiver un trigger

```
ALTER TRIGGER nomtrigger DISABLE;
```

⊶ Syntaxe : désactiver tous les triggers associés à une table

```
ALTER TABLE nomtable DISABLE ALL TRIGGERS;
```

⊶ Syntaxe : réactiver un trigger

```
ALTER TRIGGER nomtrigger ENABLE;
```

⊶ Syntaxe : réactiver tous les triggers associés à une table

```
ALTER TABLE nomtable ENABLE ALL TRIGGERS;
```

Pour modifier un trigger, il faut exécuter une instruction CREATE TRIGGER suivie de 'OR REPLACE' ou bien supprimer le trigger (DROP TRIGGER) et le créer à nouveau.

⊶ Syntaxe : supprimer un trigger

```
DROP TRIGGER nom_trigger;
```

Pour obtenir des informations sur les triggers, il faut interroger les vues suivantes : USER_TRIGGERS, ALL_TRIGGERS et DBA_TRIGGERS.
Pour accéder à la vue DBA_TRIGGERS, il faut être connecté en tant que DBA.

⊶ Syntaxe : informations sur un trigger

```
SELECT * FROM user_triggers WHERE trigger_name ='nom_trigger';
```

8.1 TRIGGERS LMD

Rappel : Le langage de manipulation de données (LMD) permet de sélectionner, insérer, modifier ou supprimer des données dans une table d'une base de données relationnelle. Les instructions du LMD sont : INSERT, UPDATE, DELETE, SELECT, EXPLAIN PLAN, LOCK TABLE, MERGE, CALL.

Ce type de trigger se déclenche donc sur un événement correspondant à une instruction du LMD (INSERT, UPDATE, DELETE). Ce type de trigger se définit uniquement selon une table. Son nom doit être unique dans un même schéma.

Nous pouvons identifier 12 types de déclencheurs :

- 3 événements :

  - INSERT
  - UPDATE
  - DELETE
- 2 modes :
  - Table (option FOR EACH ROW n'est pas spécifiée)
  - Ligne (option FOR EACH ROW est spécifiée)
- 2 moments :
  - BEFORE : avant l'événement
  - AFTER : après l'événement

Concernant l'ordre d'exécution, il est possible d'associer un et un seul déclencheur de chaque type à chaque table. Lorsque plusieurs déclencheurs sont associés à la même table, l'ordre d'exécution est le suivant :

1. Déclencheur par ordre BEFORE
2. Pour chaque ligne (FOR EACH ROW)
   1. Déclencheur par ligne BEFORE
   2. Déclencheur par ligne AFTER
3. Déclencheur par ordre AFTER

Note : depuis Oracle 11G, vous pouvez définir l'ordre à l'aide de la clause FOLLOWS.

⊶ Syntaxe : création d'un trigger

```
CREATE  [OR REPLACE] TRIGGER [schéma.]trigger
{BEFORE | AFTER | INSTEAD OF | FOR }
{DELETE | INSERT | UPDATE [OF colonne [,colonne]...]}
[OR {DELETE | INSERT | UPDATE [OF colonne [,colonne]...]}...
ON [schema.]table
[[REFERENCING {OLD [AS] ancien NEW [AS] nouveau }
| NEW [AS] nouveau OLD [AS] ancien ]]]
[FOR EACH ROW]
[FOLLOWS nom_autre_trigger]                    -- depuis Oracle 11G
[COMPOUND TRIGGER]                             -- depuis Oracle 11G
[ENABLE|DISABLE]
[WHEN (condition)]
BEGIN
Bloc PL/SQL
END;
```

BEFORE|AFTER	Précise le moment d'exécution du trigger. BEFORE : le bloc est exécuté avant la vérification des contraintes de tables et la mise à jour des données dans la table. AFTER : Le bloc est exécuté après la mise à jour des données dans la table. A noter que les triggers AFTER row sont plus efficaces que les BEFORE row parce qu'ils ne nécessitent pas une double lecture des données.
INSERT | UPDATE [OF…] |	Type d'instruction qui déclenche le trigger. On peut en mettre une, deux ou les trois.

DELETE	Pour UPDATE, on peut spécifier une liste de colonnes. Dans ce cas, le trigger ne se déclenchera que si l'instruction UPDATE porte sur l'une au moins des colonnes précisée dans la liste. Si l'événement déclencheur est **l'insertion** de lignes, il peut être détecté grâce à la variable booléenne **INSERTING**. Si l'événement déclencheur est la **suppression** de lignes, il peut être détecté grâce à la variable booléenne **DELETING**. Si l'événement déclencheur est la **modification** de lignes, il peut être détecté grâce à la variable booléenne **UPDATING**. L'opération peut être restreinte à la modification d'une ou plusieurs **colonnes**.
[[REFERENCING {OLD [AS] *ancien* NEW [AS] *nouveau* } | NEW [AS] *nouveau* OLD [AS] *ancien*]]}	Vous pouvez spécifier un nom différent pour :New et :Old avec la clause REFERENCING. Vous pouvez ainsi, dans le corps du trigger, utiliser les nouveaux noms à la place des :New et :Old.
FOR EACH ROW	Cette option définit si le trigger est un trigger de ligne ou de table. S'il est défini comme trigger de ligne, le corps du trigger sera exécuté pour chaque ligne "touchée" par l'instruction qui a déclenché le trigger. Si le trigger est un trigger de table, il sera déclenché une et une seule fois, indépendamment du nombre de lignes manipulées par l'instruction. La plupart des triggers sont spécifiés avec l'option FOR EACH ROW dans la pratique.
FOLLOWS	Oracle permet de définir plusieurs triggers sur une même table et surtout pour un même événement. Avant Oracle 11G, vous ne pouviez pas déterminer l'ordre d'exécution de ces triggers. Avec la clause FOLLOWS, vous pouvez demander à Oracle d'exécuter ce triggers après les triggers cités avec la clause FOLLOWS.
COMPOUND TRIGGER	Un trigger composé peut comporter jusqu'à 4 sections correspondant chacune à un instant de déclenchement : -- Avant instruction (BEFORE STATEMENT) -- Avant chaque ligne (BEFORE EACH ROW) -- Après instruction (AFTER STATEMENT) -- Après chaque ligne (AFTER EACH ROW)

	Par rapport à l'utilisation de plusieurs triggers séparés, les différentes sections peuvent partager des déclarations communes (variables, types, curseurs,....).
WHEN (condition)	La condition indiquée doit être vérifiée (clause retourne TRUE) pour que le code s'exécute. Si cette clause est absente, le trigger est toujours exécuté. La clause WHEN est réévaluée lors de chaque déclenchement du trigger. Si celle-ci retourne un FALSE ou un état inconnu (NULL) alors le trigger n'est pas exécuté.

8.1.1 MOMENT D'EXECUTION : AFTER ou BEFORE ?

Si le trigger doit déterminer si l'instruction SQL est autorisée, vous devez utiliser BEFORE.
Si le trigger doit "fabriquer" la valeur d'une colonne pour pouvoir ensuite la mettre dans la table, vous devez utiliser BEFORE (exemple : trigger ligne qui fabrique la valeur de clef primaire à partir d'une séquence).
Si vous avez besoin que l'instruction SQL soit terminée pour exécuter le corps du trigger, vous devez utiliser AFTER.

Note : pour des triggers lignes (option FOR EACH ROW est spécifiée), il se peut que l'utilisation de BEFORE ou AFTER n'ait aucune importance.

8.1.2 DEUX MODES

ORACLE propose deux modes :

- Les **triggers lignes** qui se déclenchent individuellement pour chaque ligne de la table affectée par le trigger (option FOR EACH ROW est spécifiée).
- Les **triggers globaux (ou de table)** qui ne se déclenchent qu'une seule fois (option FOR EACH ROW n'est pas spécifiée).

Les **triggers de table** sont exécutés une seule fois lorsque des modifications surviennent sur une table (même si ces modifications concernent plusieurs lignes de la table). Ils sont utiles si des opérations de groupe doivent être réalisées (comme le calcul d'une moyenne, d'une somme totale, d'un compteur, …).

Les **triggers lignes** sont exécutés "séparément" pour chaque ligne modifiée dans la table. Ils sont très utiles s'il faut mesurer une évolution pour certaines valeurs, effectuer des opérations pour chaque ligne en question.

Pour des raisons de performance, il est préférable, si c'est possible, d'utiliser les triggers de table plutôt que les triggers lignes.

Les données de la table à laquelle est associé le trigger sont inaccessibles depuis les instructions du bloc PL/SQL.
Les triggers lignes (FOR EACH ROW) permettent de manipuler les attributs (colonnes) de la ligne en cours de traitement, à l'aide de deux variables de type RECORD appelées :

- **:NEW** (:NEW.nom_du_champ).
- **:OLD** (:OLD.nom_du_champ).

Trigger d'insertion (INSERT)
NEW = nouvelle ligne à insérer

Trigger de suppression (DELETE)
OLD = ligne en cours de suppression

Trigger de modification (UPDATE)
NEW = ligne après modification
OLD = ligne avant modification

Séquencement sur un événement INSERT

- Une instruction INSERT est exécutée sur une table comportant un déclencheur INSERT.
- L'instruction INSERT est journalisée dans le journal des transactions.
- Le record NEW reçoit la copie de la ligne ajoutée à la table.
- Le déclencheur est lancé et ses instructions s'exécutent.

Séquencement sur un événement DELETE

- Une instruction DELETE est exécutée sur une table comportant un déclencheur DELETE.
- L'instruction DELETE est journalisée dans le journal des transactions.
- Le record OLD reçoit la copie de la ligne supprimée de la table.
- Le déclencheur est lancé et ses instructions s'exécutent.

Séquencement sur un événement UPDATE

- Une instruction UPDATE est exécutée sur une table comportant un déclencheur UPDATE
- L'instruction UPDATE est journalisée dans le journal des transactions sous la forme INSERT et DELETE
- Le record OLD reçoit la copie de la ligne de la table représentant l'image avant la modification.
- Le record NEW reçoit la copie de la ligne de la table représentant l'image après la modification.
- Le déclencheur est lancé et ses instructions s'exécutent.

Dans un trigger ligne, il est possible de modifier les éléments de NEW.

Dans la documentation Oracle, il est indiqué que : « *Old and new values are available in both BEFORE and AFTER row triggers. A new column value can be assigned in a BEFORE row trigger, but not in an AFTER row trigger (because the triggering statement takes effect before an AFTER row trigger is fired). If a BEFORE row trigger changes the value of new.column, then an AFTER row trigger fired by the same statement sees the change assigned by the BEFORE row trigger.* »

Par conséquent, il est autorisé de modifier les valeurs des colonnes au travers de NEW (et donc écrire dans le code PL/SQL :new.val:=2 par exemple), uniquement dans des triggers lignes BEFORE.

Si dans un trigger ligne vous spécifiez que son déclenchement dépend d'une colonne précise d'une table (ex : NOM de la table MEDECIN), il serait logique que l'élément OLD et NEW ne porte que sur cette colonne (NOM). Comme c'est cet élément précis qui intervient dans votre trigger, il serait très étrange d'utiliser OLD ou NEW pour atteindre une autre colonne de la table (CLIENT dans notre exemple) même si le SGBD vous l'autorise.

Les variables NEW et OLD peuvent être utilisées dans la clause WHEN ou dans le bloc d'instructions. Dans ce dernier cas, elles sont préfixées du caractère ' **:**'

Exemple d'utilisation de NEW et OLD avec WHEN

```
--  Afficher message si prix visite à positionner est inférieur au prix courant
CREATE or REPLACE TRIGGER tr_update_prixvisite
    AFTER UPDATE OF prixvisite
    ON TYPEVISITE
    FOR EACH ROW
WHEN (OLD.prixvisite > NEW.prixvisite)
BEGIN
DBMS_OUTPUT.PUT_LINE ('PRIX VISITE ' || To_char(:OLD.prixvisite) || '
SUPERIEUR AU SEUIL ' || To_char(:NEW.prixvisite) );
END;

-- Update pour déclencher le trigger
UPDATE TYPEVISITE SET prixvisite = 9;
```

Autre exemple d'utilisation de NEW et OLD avec WHEN.

Trigger nommé "inverse" : en cas d'insertion d'un enregistrement dans T1 dont la première coordonnée est inférieure à 3, l'enregistrement inverse est inséré dans T2. Les préfixes NEW et OLD (en cas d'UPDATE ou de DELETE) vont permettre de faire référence aux valeurs des colonnes après et avant les modifications dans la table. Ils sont utilisés sous la forme **NEW.num1** dans la condition du trigger et sous la forme **:NEW.num1** dans le corps.

Exemple : utilisation de NEW et OLD avec WHEN

```
CREATE TABLE T1 (num1 INTEGER, num2 INTEGER);
CREATE TABLE T2 (num3 INTEGER, num4 INTEGER);

CREATE TRIGGER inverse
AFTER INSERT ON T1
FOR EACH ROW
WHEN (NEW.num1 <=3)
BEGIN
    INSERT INTO T2 VALUES(:NEW.num2, :NEW.num1);
END inverse;
```

Vous pouvez spécifier un nom différent pour :NEW et :OLD avec la clause **REFERENCING**.
Cette clause est placée juste avant la clause FOR EACH ROW si elle existe.

⌨ Syntaxe : REFERENCING

```
REFERENCING OLD AS nouveau_nom1 NEW AS nouveau_nom2
```

⌨ Exemple d'utilisation de NEW et OLD avec REFERENCING

```
-- Afficher message si prix visite à positionner est inférieur au prix courant
CREATE or REPLACE TRIGGER tr_update_prixvisite
    AFTER UPDATE OF prixvisite
    ON TYPEVISITE
    REFERENCING OLD AS ancien NEW AS nouveau
    FOR EACH ROW
WHEN (ancien.prixvisite > nouveau.prixvisite)
BEGIN
DBMS_OUTPUT.PUT_LINE ('PRIX VISITE ' || To_char(:ancien.prixvisite) || '
SUPERIEUR AU SEUIL ' || To_char(:nouveau.prixvisite) );
    END;
-- Update pour déclencher le trigger
UPDATE TYPEVISITE SET prixvisite = 9;
```

Le corps du trigger est un bloc PL/SQL. Il peut contenir du code SQL et du code PL/SQL. Il est exécuté si l'instruction de déclenchement se produit et si la clause de restriction WHEN, le cas échéant, est évaluée à vrai. Il est différent pour un trigger ligne et pour un trigger global.
Si une erreur se produit pendant l'exécution d'un trigger, toutes les mises à jour produites par le trigger ainsi que par l'instruction qui l'a déclenché sont annulées.

8.1.3 TRIGGER ET EXCEPTIONS

Dans la documentation Oracle, il est précisé le point suivant :
"DDL statements are not allowed in the body of a trigger. Also, no transaction control statements are allowed in a trigger. ROLLBACK, COMMIT, and SAVEPOINT cannot be used. (…)"

Comme il n'est pas possible de définir des transactions ou de lancer un ROLLBACK pour terminer l'exécution normale, d'autres moyens doivent être mis en œuvre, notamment l'utilisation des exceptions.

Dans la documentation Oracle, il est précisé le point suivant :
"*Oracle Database allows user-defined errors in PL/SQL code to be handled so that user-specified error numbers and messages are returned to the client application. After received, the client application can handle the error based on the user-specified error number and message returned by Oracle Database. User-specified error messages are returned using the RAISE_APPLICATION_ERROR procedure. (…) This procedure stops procedure execution, rolls back any effects of the procedure, and returns a user-specified error number and message (unless the error is trapped by an exception handler). The error number must be in the range of -20000 to -20999.*

In database PL/SQL program units, an unhandled user-error condition or internal error condition that is not trapped by an appropriate exception handler causes the implicit rollback of the program unit.".

Par conséquent, dans un trigger BEFORE, vous pouvez empêcher la modification (INSERT, UPDATE, DELETE) en lançant une erreur via la procédure RAISE_APPLICATION_ERROR.

8.2 RESTRICTIONS

L'exécution d'un déclencheur dont le bloc PL/SQL inclut des ordres INSERT, DELETE ou UPDATE peut entraîner la mise en œuvre d'un autre déclencheur associé à la table modifiée par ces actions.

Dans ce cas, lors de l'exécution d'un déclencheur de type ligne :

- Aucun ordre SQL ne doit consulter ou modifier une table déjà utilisée en mode modification par un autre utilisateur.
- Un déclencheur ne peut modifier la valeur d'un attribut déclaré avec l'une des contraintes PRIMARY KEY, UNIQUE ou FOREIGN KEY
- Un déclencheur ne peut pas consulter les données d'une table en mutation : une table en mutation est une table directement ou indirectement concernée par l'événement qui a provoqué la mise en œuvre du déclencheur

8.3 TRIGGERS INSTEAD OF SUR UNE VUE

L'option INSTEAD OF (littéralement : au lieu de) peut être spécifiée uniquement sur les triggers placés sur une vue.
Cette option offre une possibilité transparente et efficace pour permettre la mise à jour de données à travers des vues non modifiables (non UPDATABLE).
L'action du trigger est bien effectuée en lieu et place de l'instruction qui le déclenche. Les triggers INSTEAD OF ne disposent pas des options BEFORE et AFTER.
Vous pouvez lire ensemble les valeurs :OLD et :NEW, mais vous ne pouvez modifier ni l'une ni l'autre.

⊶ Syntaxe : trigger INSTEAD OF

```
CREATE TRIGGER nom_trigger INSTEAD OF
{INSERT¦UPDATE¦DELETE}
ON nom_de_vue
BEGIN
Bloc PL/SQL
END;
```

8.4 DECLENCHEURS SUR LES EVENEMENTS SYSTEME

Les triggers sur événements SYSTEME sont déclenchés sur les événements STARTUP, SHUTDOWN, SERVERERROR, LOGON ou LOGOFF. Ils sont généralement utilisés par les administrateurs pour suivre les changements d'état du système.
Il est possible d'utiliser les triggers pour suivre les changements d'état du système, tels que :

- L'arrêt ou le démarrage de l'instance d'une base de données.

- La connexion ou la déconnexion d'un utilisateur à la base de données.
- L'apparition d'une erreur au niveau d'Oracle.

⊶ Syntaxe : trigger sur événements SYSTEME

```
CREATE OR REPLACE TRIGGER nom_trigger
{BEFORE | AFTER } evenement_systeme
ON [DATABASE | SCHEMA]
BEGIN
Bloc PL/SQL
END;
```

Le tableau ci-dessous liste les événements en fonction du moment d'exécution (AFTER ou BEFORE).

EVENEMENT	AFTER	BEFORE
STARTUP	X	
SHUTDOWN		X (uniquement shutdown normal ou shutdown immediate)
SERVERERROR	X	
LOGON	X	
LOGOFF		X
SUSPEND	X	X
DB_ROLE_CHANGE	X	

Pour ce type d'événement, vous pouvez créer un déclencheur sur la base de données (DATABASE) ou sur le schéma (SCHEMA).
Le tableau ci-dessous présente les différents événements systèmes sur lesquels il est possible de créer un déclencheur.

EVENEMENT	DESCRIPTION	PORTEE
AFTER STARTUP	Evénement déclenché lors de l'ouverture de l'instance	DATABASE
BEFORE SHUTDOWN	Evénement déclenché avant le processus d'arrêt de l'instance (non déclenché en cas d'arrêt brutal du serveur)	DATABASE
AFTER SERVERERROR	Evénement déclenché lors d'une erreur Oracle (sauf les erreurs suivantes : ORA-1034, ORA-1403, ORA-1422, ORA-1423 et ORA-4030)	DATABASE ou SCHEMA
AFTER LOGON	Evénement déclenché lorsqu'un utilisateur se connecte à la base de données.	DATABASE ou SCHEMA
BEFORE LOGOFF	Evénement déclenché lorsqu'un utilisateur se déconnecte de la base de données.	DATABASE ou SCHEMA
SUSPEND	Evénement déclenché lorsqu'une erreur survient sur le serveur et entraine la suspension d'une	DATABASE ou SCHEMA

EVENEMENT	DESCRIPTION	PORTEE
	transaction.	
AFTER DB_ROLE_CHANGE	Evénement déclenché dans une configuration “Data Guard”, lorsqu’un rôle change de standby à primary ou de primary à standby	DATABASE

Les déclencheurs présentés ci-dessous (tr_logon et tr_startup) enregistrent dans la table table_logon les attributs ora_login_user, ora_sysevent, sysdate, ora_client_ip_address et ora_instance_num.
Ces attributs permettent d’obtenir des informations concernant l’origine de l’événement. La liste des attributs disponibles est présentée au paragraphe LES ATTRIBUTS concernant les déclencheurs sur événements LDD.
La table suivante doit être créée afin d’enregistrer les événements et de permettre ainsi la vérification de l’exécution des déclencheurs.

Exemple : création de la table

```
CREATE TABLE table_log (
    UserName VARCHAR2(32),
    EventAction VARCHAR2(20),
    EventDate Date,
    IP_Address VARCHAR2(20),
    InstanceNum VARCHAR2(25)
);
```

Exemple : déclencheur sur un événement de type AFTER LOGON

```
CREATE OR REPLACE TRIGGER tr_logon AFTER LOGON
ON DATABASE
BEGIN
INSERT INTO table_log (UserName, EventAction, EventDate, IP_Address,
InstanceNum)
VALUES (ora_login_user, ora_sysevent, sysdate, ora_client_ip_address,
ora_instance_num);
END;
```

Exemple : déclencheur sur un événement de type AFTER STARTUP

```
CREATE OR REPLACE TRIGGER tr_startup
AFTER STARTUP
ON DATABASE
BEGIN
INSERT INTO table_log (UserName, EventAction, EventDate, IP_Address,
InstanceNum)
VALUES (ora_login_user, ora_sysevent, sysdate, ora_client_ip_address,
ora_instance_num);
END;
```

Exemple : vérification du déclenchement

```
SELECT * FROM table_log;
```

Pour créer ces déclencheurs, il faut posséder le privilège ADMINISTER

DATABASE TRIGGER.

8.5 DECLENCHEURS SUR LES EVENEMENTS LDD

RAPPEL : Le Langage de Définition des Données – LDD (en anglais DDL - Data Definition Language) regroupe l'ensemble des possibilités permettant de modifier la structure de la base de données.
Les principales instructions du LDD sont : CREATE, ALTER, DROP, RENAME, TRUNCATE, COMMENT, GRANT, DENY, REVOKE et UPDATE STATISTICS.
Ces déclencheurs s'exécutent donc en réponse aux instructions LDD. Il est possible d'utiliser les triggers pour suivre les changements de la structure de la base de données, tels que :

- Création, modification ou suppression d'un objet au sein de la base de données ou d'un schéma.
- Mise en place ou la suppression d'un audit sur un objet de la base de données ou d'un schéma.
- Modification des privilèges et des rôles.

Exemples d'utilisation des déclencheurs LDD :

- Vous voulez empêcher certaines modifications sur votre schéma de base de données.
- Vous voulez qu'un événement se produise dans la base de données en réponse à une modification du schéma.
- Vous voulez enregistrer des modifications ou des événements dans le schéma de la base de données.

Par exemple, un déclencheur LDD créé pour être activé en réponse à un événement ALTER TABLE se déclenchera à chaque fois qu'un événement ALTER TABLE se produira dans la base de données.
Les déclencheurs LDD, tout comme les déclencheurs LMD, exécutent des instructions PL/SQL en réponse à un événement.

⊶ Syntaxe : trigger sur événement LDD

```
CREATE [OR REPLACE] TRIGGER nom_trigger {AFTER|BEFORE}
evenement_ldd
ON {DATABASE|SCHEMA}
BEGIN
    Bloc PL/SQL
END;
```

Les déclencheurs LDD ne peuvent pas être utilisés comme déclencheurs INSTEAD OF.

Le tableau ci-dessous présente les différents événements LDD sur lesquels il est possible de créer un déclencheur.

EVENEMENT	DESCRIPTION	PORTEE
CREATE	Evénement déclenché lors de la création d'un objet	DATABASE ou SCHEMA
ALTER	Evénement déclenché lors de la modification d'un objet	DATABASE ou SCHEMA

DROP	Evénement déclenché lors de la suppression d'un objet	DATABASE ou SCHEMA
ANALYZE	Evénement déclenché lors de l'analyse d'un objet	DATABASE ou SCHEMA
ASSOCIATE STATISTICS	Evénement déclenché lors de l'association d'une statistique	DATABASE ou SCHEMA
DISSOCIATE STATISTICS	Evénement déclenché lors de la dissociation d'une statistique	DATABASE ou SCHEMA
AUDIT	Evénement déclenché lors de la mise en place d'un audit	DATABASE ou SCHEMA
NOAUDIT	Evénement déclenché lors de l'annulation d'un audit	DATABASE ou SCHEMA
COMMENT	Evénement déclenché lors de l'insertion d'un commentaire	DATABASE ou SCHEMA
DDL	Evénement déclenché lors de l'exécution des ordres LDD (sauf ALTER DATABASE, CREATE CONTROLFILE et CREATE DATABASE)	DATABASE ou SCHEMA
GRANT	Evénement déclenché lors de l'exécution d'une commande GRANT	DATABASE ou SCHEMA
RENAME	Evénement déclenché lors de l'exécution d'une commande RENAME	DATABASE ou SCHEMA
REVOKE	Evénement déclenché lors de l'exécution d'une commande REVOKE	DATABASE ou SCHEMA
TRUNCATE	Evénement déclenché lors d'une troncature de table (suppression uniquement des données de la table).	DATABASE ou SCHEMA

⌨ Exemple : trigger sur événement COMMENT

```
CREATE OR REPLACE TRIGGER tr_commentaire
AFTER COMMENT
ON SCHEMA
BEGIN
DBMS_OUTPUT.PUT_LINE('Ajout d"un commentaire');
END;
```

⌨ Exemple : vérification du déclenchement

```
COMMENT ON TABLE "EMPLOYEES" IS 'Commentaire sur la table'
```

⌨ Exemple : trigger sur événement RENAME

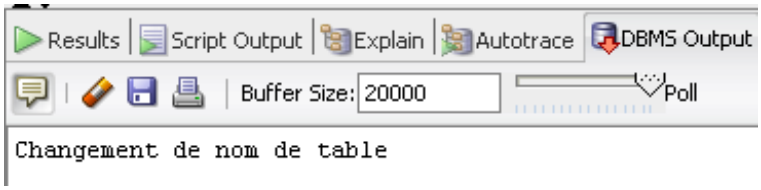
```
CREATE OR REPLACE TRIGGER tr_rename
BEFORE RENAME
ON DATABASE
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('Changement de nom de table');
END;
```

Exemple : vérification du déclenchement

```
RENAME EMPLOYEES TO SALARIES
RENAME SALARIES TO EMPLOYEES
```

Sortie DBMS sous SQL Developer.



8.5.1 LES ATTRIBUTS

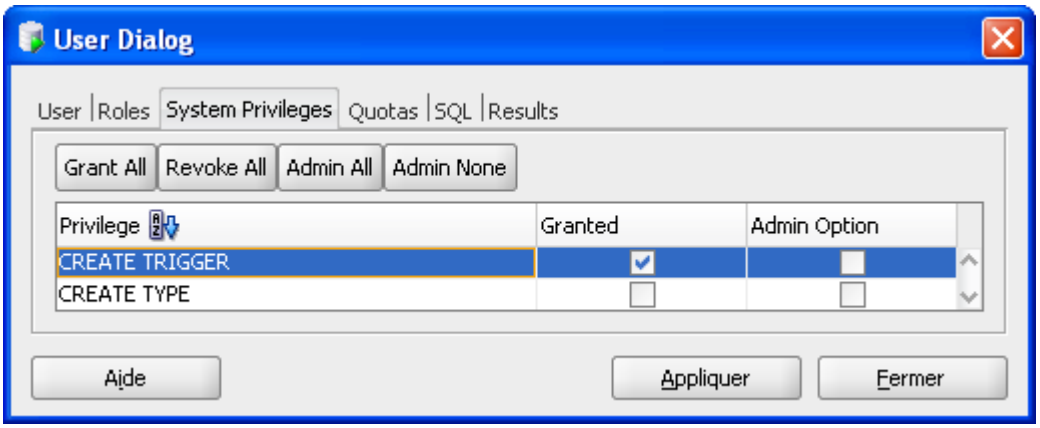
Lors de l'écriture de ces déclencheurs, il est possible d'utiliser des attributs pour identifier précisément l'origine de l'événement et adapter les traitements en conséquence.
Le tableau ci-dessous présente les différents attributs.

ATTRIBUT	DESCRIPTION
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 modifié
ora_dict_obj_name	Nom de l'objet visé par l'opération LDD
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 LDD
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 LDD
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'événement système qui a activé le déclencheur
ora_with_grant_option	Vrai si le privilège a été accordé avec option

ATTRIBUT	DESCRIPTION
	d'administration

8.6 PRIVILEGES

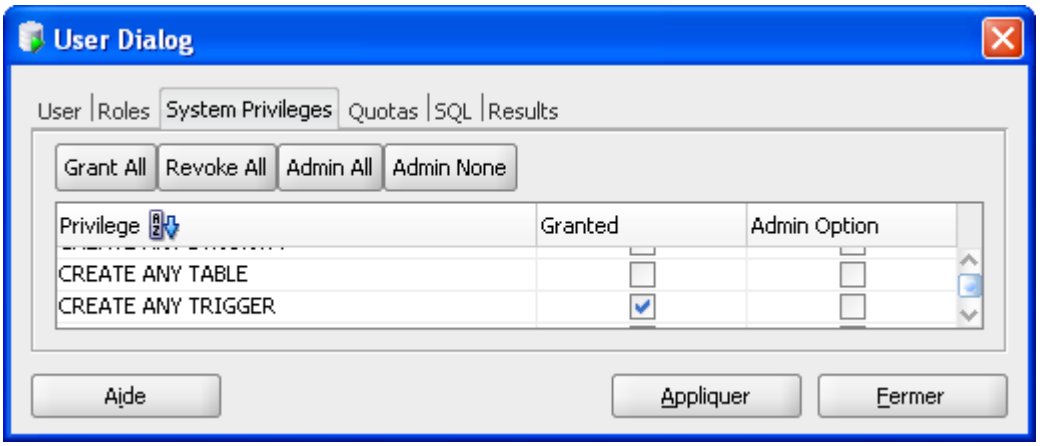
La création d'un trigger nécessite d'avoir certains privilèges.
Pour créer un trigger dans votre propre schéma sur une table de votre schéma ou sur votre schéma lui-même, vous devez avoir le privilège **CREATE TRIGGER**.



⊶ Syntaxe : privilège pour creation trigger sur schema de *utilisateur*

```
SQL> GRANT CREATE TRIGGER TO utilisateur;
```

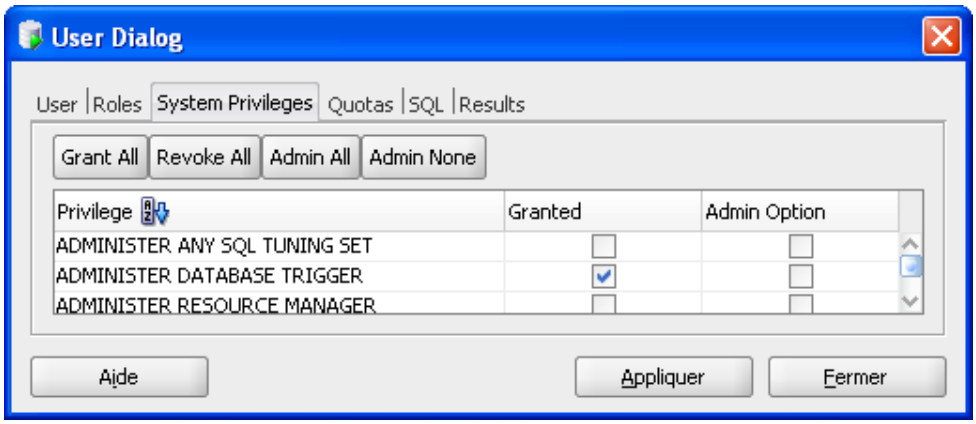
Pour créer un trigger sur une table de n'importe quel autre schéma ou sur un autre schéma que le vôtre (schema.SCHEMA), vous devez avoir le privilège **CREATE ANY TRIGGER**.



⊶ Syntaxe : privilège pour creation trigger sur tous les schémas

```
SQL> GRANT CREATE ANY TRIGGER TO utilisateur ;
```

En plus de ces privilèges, pour créer un trigger sur la base de données, vous devez avoir le privilège **ADMINISTER DATABASE TRIGGER**.



⊶ Syntaxe : privilège pour creation trigger sur la base de données

```
SQL> ALTER ADMINISTER DATABASE TRIGGER FROM utilisateur;
```

8.7 UTILISATION DES DECLENCHEURS

Les déclencheurs servent à garantir l'exécution d'une tâche et à centraliser les opérations sans s'occuper de qui les déclenche ni où elles se déclenchent puisqu'elles se trouvent dans la base de données.
Les déclencheurs sont généralement utilisés pour :

- Mise à jour de données. Lors de l'effacement de données, on peut stocker automatiquement les valeurs effacées dans une table journal. L'effacement devient alors logique. Une autre utilisation classique est la gestion de champs calculés.
- Duplication des données. La duplication de données, pour des raisons de performance sur une base distribuée peut être faite avec des déclencheurs. On peut facilement gérer une réplication synchrone asymétrique.
- Intégrité étendue des données. La gestion de l'intégrité des données peut être faite avec des déclencheurs, particulièrement dans les cas où elle ne peut être assurée par des contraintes déclaratives. Toutefois, les déclencheurs ne doivent pas se substituer au système d'intégrité natif du SGBD pour des raisons de portabilité. Il est important de noter que si les déclencheurs peuvent gérer l'ensemble de l'intégrité des données de la base, les contrôles s'appliquent au niveau de la seule transaction. Ils ne contrôlent donc pas les données existantes.
- Sécurité étendue des données. On peut également gérer les droits d'accès aux données avec des triggers, particulièrement si ceux-ci ne peuvent être traités par les commandes classiques de gestion des droits.
- Ou encore…
  - Renforcement de règles de gestion complexes.
  - Mise en place d'événements de connexion transparents.
  - Génération automatique de colonnes dérivées.
  - Mise en place de vues modifiables complexes.
  - Surveillance d'événements systèmes.

9 PACKAGES

Un package est un ensemble nommé d'objets associés (TYPES, VARIABLES, FONCTIONS, PROCEDURES) stocké dans la base.

Un package comporte deux parties :

- Une partie SPECIFICATION visible depuis les applications externes. Elle permet de décrire le contenu du package, de connaître le nom et les paramètres d'appel des fonctions et des procédures.
- Une partie CORPS invisible depuis les applications externes. Elle contient l'implémentation des procédures et fonctions exposées dans la partie SPECIFICATION.

⊶ Syntaxe : création d'un package

```
CREATE [OR REPLACE] PACKAGE [Schéma.]Nom_package IS
Spécification_pl/sql
```

⊶ Syntaxe : création du corps d'un package

```
CREATE [OR REPLACE] PACKAGE BODY [Schéma.]Nom_package IS
pl/sql_package_body
```

⊶ Syntaxe : exécution d'une procédure d'un package

```
EXECUTE nom_package.nom_procédure(liste_paramètres effectifs)
```

9.1 PAQUETAGE DBMS_OUTPUT

Les procédures de ce package permettent d'écrire des lignes dans un buffer depuis un bloc PL/SQL. Le contenu de ce buffer est affiché à l'écran à la fin d'exécution du bloc.

9.1.1 DESCRIPTION DES METHODES

Pour initialiser le buffer et autoriser les ordres de lecture et d'écriture dans ce buffer.

```
DBMS_OUTPUT.ENABLE (taille_buffer IN INTEGER DEFAULT 20000)
```

taille_buffer : Taille allouée au buffer. Taille maximum du buffer : 1.000.000 d'octets (par défaut 2000). Taille maximum d'une ligne : 255 octets.

Pour purger le buffer et empêcher les ordres de lecture et d'écriture.

```
DBMS_OUTPUT.DISABLE
```

Procédure PUT : permet d'ajouter des données dans la ligne en cours du buffer.

```
DBMS_OUTPUT.PUT (élément IN NUMBER | VARCHAR2 | DATE )
```

Procédure PUT_LINE : permet d'ajouter une ligne entière dans le buffer.

```
DBMS_OUTPUT.PUT_LINE (élément IN NUMBER | VARCHAR2 | DATE)
```

Procédure NEW_LINE : permet d'ajouter un caractère de fin de ligne.

```
DBMS_OUTPUT.NEW_LINE
```

Procédure GET_LINE : lit une ligne et la place dans la variable ligne.

```
DBMS_OUTPUT.GET_LINE (ligne OUT VARCHAR2, état OUT INTEGER)
```

Procédure GET_LINES : lit nb_lignes lignes et les place dans un tableau de chaines de caractères lignes.

```
DBMS_OUTPUT.GET_LINES (lignes OUT tab_char, nb_lignes OUT
INTEGER)
```