

ORACLE®



SGBD avancé (oracle PL/SQL)

**[Kamel BENRAIS
Expert nouvelle technologies**

Rappel SQL

Notion de base

SQL : Introduction

- SQL : Structured Query Language
- langage de gestion de bases de données relationnelles :
 - définir les données(LDD)
 - Interroger la base de données(Langage de requêtes) manipuler les données(LMD)
 - Contrôler l'accès aux données(LCD)

SQL Commande LDD

- Un **langage de définition de données (LDD** ; en anglais *data definition language*, DDL) est un langage de programmation et un sous-ensemble de SQL pour manipuler les structures de données d'une base de données, et non les données elles-mêmes, Il est donc destiné aux concepteurs.
- On distingue typiquement quatre types de commandes LDD
 - CREATE : création d'une structure de données ;
 - ALTER : modification d'une structure de données ;
 - DROP : suppression d'une structure de données ;
 - RENAME : renommage d'une structure de données.
- Ces commandes sont appliqués au objets suivantes:
TABLE ,INDEX ,VIEW,SEQUENCE,SYNONYM,USER.

Exemple LDD

- **CREATE TABLE** Employee(
CIN NUMBER NOT NULL,
nom VARCHAR2(20),
prenom VARCHAR2(20),
Gender CHAR(1),
salaire NUMBER(5) NOT NULL,
Dept NUMBER
);
- **ALTER TABLE** Employee **ADD** Age INTEGER;

SQL commande LMD

- Un **langage de manipulation de données (LMD** ; en anglais *data manipulation language*, DML) est un langage de programmation et un sous-ensemble de SQL pour manipuler les données d'une base de données, Il est donc destiné aux développeurs.
- On distingue typiquement quatre types de commandes SQL de manipulation de données :
 - SELECT : sélection de données dans une table ;
 - INSERT : insertion de données dans une table ;
 - DELETE : suppression de données d'une table ;
 - UPDATE : mise à jour de données d'une table.

Exemple LMD

- **Select nom,prenom from Employee;**
- **INSERT INTO** Employee(cin ,nom, prenom) **VALUES** (124578, 'sahbani', 'mouhamed');
- **DELETE FROM** Employee **WHERE** prenom = 'masmoudi' **and** nom = 'ali';
- **UPDATE** Employee **SET** prenom = 'oumar' **WHERE** nom = 'ayari';

SQL Commande LCD

- Un **langage de contrôle de données (LCD** ; en anglais *data control language*, DCL) est un langage de programmation et un sous-ensemble de SQL pour contrôler l'accès aux données d'une base de données, Il est donc destiné aux DBA.
- On distingue typiquement six types de commandes SQL de contrôle de données :
 - GRANT : autorisation d'un utilisateur à effectuer une action ;
 - DENY : interdiction à un utilisateur d'effectuer une action ;
 - REVOKE : annulation d'une commande de contrôle de données précédente ;
 - COMMIT : validation d'une transaction en cours ;
 - ROLLBACK : annulation d'une transaction en cours ;
 - LOCK : verrouillage sur une structure de données.

Example Commande LCD

- **GRANT UPDATE** (nom, prenom) **ON** Employee **TO** grh **WITH GRANT OPTION**;
- **DENY DELETE TO** grh
- **REVOKE UPDATE** (nom, prenom) **ON** Employee **FROM** grh
- **ROLLBACK TO** sauvegarde;
- **LOCK TABLE** Employee **IN EXCLUSIVE MODE**;

Oracle PL/SQL Introduction



Définition

- Acronyme

PL SQL = Procédural SQL

- Objectif

Proposer des fonctionnalités procédurales performantes (traitements proches des données) pour contrôler et manipuler les données (gestion de tableaux, boucles, condition ,etc.)

PL/SQL dans oracle

- PL/SQL est le langage procédural d'Oracle. Il est une extension du SQL qui est un langage ensembliste.
- PL/SQL permet de gérer des traitements qui utilisent les instructions SQL dans un langage procédural.

- Avantages de PL/SQL

Il est possible d'écrire des fonctions complexes de manipulation de données sans recourir à un langage externe.

Le code PL/SQL est très proche du moteur Oracle. De plus pour le code stocké, les requêtes qu'il manipule sont pré-compilées, et donc son exécution est optimisée.

Création de blocs PL/SQL

- PL/SQL est un langage structuré en blocs, constitués d'un ensemble d'instructions.
 - Un bloc PL/SQL peut être "externe", on dit alors qu'il est anonyme, ou alors stocké dans la base de données sous forme de procédure, fonction ou trigger.
 - un bloc PL/SQL est intégralement envoyé au moteur PL/SQL, qui traite chaque instruction PL/SQL et sous-traite les instructions purement SQL au moteur SQL, afin de réduire le trafic réseau.
 - Un bloc PL/SQL est terminé par un ; comme une instruction SQL.
- Par ailleurs, dans les environnements d'exécution Oracle (comme SQL Developer), il est nécessaire de séparer les blocs par un "/" (sur une nouvelle ligne).
- Une bonne habitude est donc de terminer les blocs PL/SQL par des "/".

Syntaxe du block simple

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

```
DECLARE  
    /*déclaration des variables*/  
BEGIN  
    /* execution des Instructions*/  
END;
```


Syntaxe de block complexe

- Un bloc PL/SQL est composé de trois parties
 - Une partie déclarative (Facultative)
 - Une partie exécutable (Obligatoire)
 - Une partie exception (Facultative)
- DECLARE
 - ? Déclarations de variables, constantes ,exceptions, curseurs
- BEGIN
 - Commandes SQL du langage de manipulation des données
 - Utilisation de structures de contrôles (conditionnels, itératifs)
 - Utilisation des curseurs
 - Appels de fonctions, procédures, packages
 - Utilisation de blocs PL/SQL imbriqués
- EXCEPTION
 - Traitement des exceptions (erreurs)
- END ;
- /

```
DECLARE
  l_message
  VARCHAR2 (100) := 'Hello';
BEGIN
  DECLARE
    l_message2  VARCHAR2 (100) :=
      l_message <strong>||</strong> ' World!';
  BEGIN
    DBMS_OUTPUT.put_line (l_message2);
  END;
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.put_line
      (DBMS_UTILITY.format_error_stack);
END;
```



Définition de variables et de types de données

Les variables

- Les variables PL/SQL doivent être déclarées dans la section de déclaration ou dans un package en tant que variable globale. Lorsque vous déclarez une variable, PL/SQL alloue de la mémoire pour la valeur de la variable et l'emplacement de stockage est identifié par le nom de la variable.
- **Syntaxe de déclaration**
 - `nom variable [CONSTANT] type [[NOT NULL] := expression] ;`
- **Exemples**
 - `sales number;`
 - `pi CONSTANT double precision := 3.1415;`
 - `name varchar2(25);`
 - `address varchar2(100);`

Les variables

- Lorsque vous fournissez une limite de taille, d'échelle ou de précision avec le type de données, cela s'appelle une déclaration contrainte. Les déclarations contraintes nécessitent moins de mémoire que les déclarations sans contraintes.
- Par exemple –
 - `sales number(10, 2);`
 - `name varchar2(25);`
 - `address varchar2(100);`

Initialisation Variables dans PL/SQL

- Chaque fois que vous déclarez une variable, PL/SQL lui attribue une valeur par défaut NULL.
- Si vous souhaitez initialiser une variable avec une valeur autre que la valeur NULL, vous pouvez le faire lors de la déclaration, en utilisant l'une des méthodes suivantes .
 - Le mot-clé DEFAULT
 - L'opérateur d'affectation(:=)
- Exemple
 - Compteur `binary_integer := 0;`
 - Message `varchar2(20) DEFAULT 'bonjour ';`

Initialisation Variables dans PL/SQL

- Vous pouvez spécifier qu'une variable ne doit pas avoir de valeur NULL à l'aide de la contrainte NOT NULL. Si vous utilisez la contrainte NOT NULL, vous devez explicitement affecter une valeur initiale à cette variable.

```
DECLARE  
V_gendre VARCHAR2( 25 ) NOT NULL := 'HOMME';  
BEGIN  
V_gendre := '';  
END;
```

- PL/SQL affiche l'erreur suivante :

```
ORA-06502: PL/SQL: numeric or value error
```

La portée des variables en PL/SQL

- PL/SQL permet l'imbrication de blocs, c'est-à-dire que chaque bloc de programme peut contenir un autre bloc interne. Si une variable est déclarée dans un bloc interne, elle n'est pas accessible au bloc externe. Cependant, si une variable est déclarée et accessible à un bloc externe, elle est également accessible à tous les blocs internes imbriqués. Il existe deux types de périmètre variable –
 - ✓ Variables locales – Variables déclarées dans un bloc interne et non accessibles aux blocs externes.
 - ✓ Variables globales - Variables déclarées dans le bloc le plus externe ou un package.

Exemple variable local et variable global

```
DECLARE
-- variables Globale
num1 number := 95;
num2 number := 85;
BEGIN
dbms_output.put_line('Variable externe num1: ' || num1);
dbms_output.put_line('Variable externe num2: ' || num2);
  DECLARE
    -- variables Locale
    num1 number := 195;
    num2 number := 185;
  BEGIN
    dbms_output.put_line('Variable interne num1: ' || num1);
    dbms_output.put_line('Variable interne num2: ' || num2);
  END;
END;
/
```

résultat

```
Variable externe num1: 95
Variable externe num2: 85
Variable interne num1: 195
Variable interne num2: 185

PL/SQL procedure successfully completed.
```


Affectation de résultats de requête SQL à des variables PL/SQL

- Vous pouvez utiliser l'instruction SELECT INTO de SQL pour affecter des valeurs aux variables PL/SQL. Pour chaque élément de la liste SELECT, il doit y avoir une variable de type compatible correspondante dans la liste INTO. L'exemple suivant illustre le concept. Créons une table nommée

CLIENTS–

```
CREATE TABLE CLIENTS(  
    ID INT NOT NULL,  
    NOM VARCHAR (20) NOT NULL,  
    AGE INT NOT NULL,  
    ADDRESSE CHAR (25),  
    SALAIRE DECIMAL (18, 2),  
    PRIMARY KEY (ID)  
);
```

Insert
données

```
INSERT INTO CLIENTS(ID,NOM,AGE,ADDRESSE,SALAIRE)  
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );  
INSERT INTO CLIENTS(ID,NAME,AGE,ADDRESSE,SALAIRE)  
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );
```

s
u
i
t
e

Le programme suivant affecte les valeurs du tableau ci-dessus aux variables PL/SQL à l'aide de la clause SELECT INTO de SQL –

- Le programme suivant affecte les valeurs du tableau ci-dessus aux variables PL/SQL à l'aide de la clause SELECT INTO de SQL –

```
DECLARE
  c_id clients.id%type := 1;
  c_nom clients.nom%type;
  c_addr clients.addresse%type;
  c_sal clients.salaire%type;
BEGIN
  SELECT nom, adresse, salaire INTO c_nom, c_addr, c_sal
  FROM clients
  WHERE id = c_id;
  dbms_output.put_line (clients' || c_nom || ' from ' || c_addr || ' earns ' || c_sal);
END;
/
```

Utilisation de %TYPE

- L' %TYPE attribut fournit le type de données d'une variable ou d'une colonne de base de données. Dans l'exemple suivant, %TYPE fournit le type de données d'une variable :
 - credit REAL(7,2);
 - debit credit%TYPE;
- L' %TYPE attribut est particulièrement utile lors de la déclaration de variables faisant référence à des colonnes de base de données. Vous pouvez référencer une table et une colonne, ou vous pouvez référencer un propriétaire, une table et une colonne, comme dans
- my_dname scott.dept.dname%TYPE ;

Utilisation de %ROWTYPE

- L' %ROWTYPE attribut fournit un type d'enregistrement qui représente une ligne dans une table (ou une vue). L'enregistrement peut stocker une ligne entière de données sélectionnées dans la table ou récupérées à partir d'un curseur ou d'une variable de curseur fortement typée. Dans l'exemple ci-dessous, vous déclarez deux enregistrements. Le premier enregistrement stocke une ligne sélectionnée dans la emptable. Le deuxième enregistrement stocke une ligne extraite du curseur c1.

— DECLARE

```
emp_rec emp%ROWTYPE;
```

```
CURSOR c1 IS SELECT deptno, dname, loc FROM dept;
```

```
dept_rec c1%ROWTYPE;
```

- Les colonnes d'une ligne et les champs correspondants d'un enregistrement ont les mêmes noms et types de données. Cependant, les champs d'un %ROWTYPE enregistrement n'héritent pas de la NOT NULL contrainte de colonne. Dans l'exemple suivant, vous sélectionnez des valeurs de colonne dans recordemp_rec :

— BEGIN

```
SELECT * INTO emp_rec FROM emp WHERE ...
```

Vous pouvez également affecter la valeur d'une expression à un champ spécifique, comme le montrent les exemples suivants :

```
emp_rec.ename := 'JOHNSON';
```

```
emp_rec.sal := emp_rec.sal * 1.15;
```

Reconnaissance des unités lexicales PL/SQL

- Une ligne de texte PL/SQL contient des groupes de caractères appelés *unités lexicales* , qui peuvent être classés comme suit :
 - délimiteurs (symboles simples et composés)
 - identifiants, qui incluent des mots réservés
 - Littéraux
 - Commentaires
- vous pouvez séparer les unités lexicales par des espaces
- Exemple: IF x>y THEN max:=x;ELSE max:=y;END IF;
- IF x > y THEN high := x; ENDIF; -- interdit (espace dans END IF obligatoire)
- count := count + 1;
- count : = count + 1; -- -- interdit (espace dans := interdit)

Affectation globale

- Une %ROWTYPE déclaration ne peut pas inclure de clause d'initialisation. Cependant, il existe deux façons d'attribuer des valeurs à tous les champs d'un enregistrement à la fois. Premièrement, PL/SQL permet une affectation agrégée entre des enregistrements entiers si leurs déclarations font référence à la même table ou au même curseur. Par exemple, l'affectation suivante est autorisée :

```
DECLARE
    dept_rec1 dept%ROWTYPE;
    dept_rec2 dept%ROWTYPE;
    CURSOR c1 IS SELECT deptno, dname, loc FROM dept;
    dept_rec3 c1%ROWTYPE;
BEGIN
    ... dept_rec1 := dept_rec2;
```

- Cependant, étant donné qu'elle dept_rec2 est basée sur une table et dept_rec3 sur un curseur, l'affectation suivante n'est pas autorisée :

```
dept_rec2 := dept_rec3; -- interdit
```

Reconnaissance des unités lexicales PL/SQL

- vous pouvez diviser les lignes à l'aide de retours chariot et les lignes d'indentation à l'aide d'espaces ou de tabulations(code plus lisible):
- IF x>y THEN max:=x;ELSE max:=y;END IF; | IF x > y THEN
| max := x;
| ELSE
| max := y;
| END IF;

Unités lexicales PL/SQL-Délimiteurs

- Un *délimiteur* est un symbole simple ou composé qui a une signification particulière pour PL/SQL:

symbole	Sens
+	opérateur d'addition
%	indicateur d'attribut
'	délimiteur de chaîne de caractères
.	sélecteur de composants
/	opérateur de division
(expression ou délimiteur de liste
)	expression ou délimiteur de liste
:	indicateur de variable hôte
,	séparateur d'articles
*	opérateur de multiplication
"	délimiteur d'identificateur entre guillemets
=	opérateur relationnel
<	opérateur relationnel
>	opérateur relationnel
@	indicateur d'accès à distance
;	terminateur d'instruction
-	opérateur de soustraction/négation

symbole	Sens
:=	opérateur d'assignation
=>	opérateur d'association
	opérateur de concaténation
**	opérateur d'exponentiation
<<	délimiteur d'étiquette (début)
>>	délimiteur d'étiquette (fin)
/*	délimiteur de commentaire multiligne (début)
*/	délimiteur de commentaire multiligne (fin)
..	opérateur de gamme
<>	opérateur relationnel
!=	opérateur relationnel
~=	opérateur relationnel
^=	opérateur relationnel
<=	opérateur relationnel
>=	opérateur relationnel
--	indicateur de commentaire sur une seule ligne

Unités lexicales PL/SQL-Identifiants

- Vous utilisez des identificateurs pour nommer des éléments et des unités de programme PL/SQL, qui incluent des constantes, des variables, des exceptions, des curseurs, des variables de curseur, des sous-programmes et des packages. Exp(X,t2,phone#,credit_limit,LastName,oracle\$number)
- Un identificateur se compose d'une lettre éventuellement suivie de plusieurs lettres, chiffres, signes dollar, traits de soulignement et signes dièse. Les autres caractères tels que les tirets, les barres obliques et les espaces ne sont pas autorisés
 - Exp : debit-amount -- non autorisé en raison du trait d'union
- Vous pouvez utiliser des majuscules, des minuscules ou des majuscules pour écrire les identifiants. PL/SQL n'est pas sensible à la casse

Définition des Sous-types PL/SQL par l'utilisateur

- PL/SQL vous permet de définir vos propres sous-types.
- Les sous-types peuvent :
 - Assurer la compatibilité avec les types de données ANSI/ISO
 - Montrer l'utilisation prévue des éléments de données de ce type
 - Détecter les valeurs hors plage
- Trois catégories de sous-type possible
 - Sous-types sans contrainte
 - Sous-types contraints
 - Sous-types avec types de base dans la même famille de types de données

Sous-types sans contrainte

- Un **sous-type non contraint** a le même ensemble de valeurs que son type de base, il ne s'agit donc que d'un autre nom pour le type de base.
- Pour définir un sous-type sans contrainte, utilisez cette syntaxe :
 - `SUBTYPE nom_sous -type IS type_base`
 - Un exemple de sous-type sans contrainte, que PL/SQL prédéfinit pour la compatibilité avec ANSI, est :
 - `SUBTYPE "DOUBLE PRECISION" IS FLOAT`
 - Les sous-types non contraints définis par l'utilisateur montrent l'utilisation prévue:

exemple, les sous-types sans contrainte

```
DECLARE
  SUBTYPE Balance IS NUMBER;
  checking_account Balance(6,2);
  savings_account Balance(8,2);
  certificate_of_deposit Balance(8,2);
  max_insured CONSTANT Balance(8,2) := 250000.00;
  SUBTYPE Counter IS NATURAL;
  accounts Counter := 1;
  deposits Counter := 0;
  withdrawals Counter := 0;
  overdrafts Counter := 0;
PROCEDURE deposit ( account IN OUT Balance, amount IN Balance ) IS
  BEGIN
    account := account + amount;
    deposits := deposits + 1;
  END;
BEGIN
  NULL;
END; /
```

Sous-types constraints

- Un **sous-type contraint** n'a qu'un sous-ensemble des valeurs de son type de base.
- La syntaxe de définition de sous-type est :

```
SUBTYPE subtype_name IS base_type { precision [, scale ] | RANGE low_value .. high_value } [ NOT NULL ]
```

exemple, le sous-type contraint Balance détecte les valeurs hors plage

```
DECLARE
    SUBTYPE Balance IS NUMBER(8,2);
    checking_account Balance;
    savings_account Balance;
BEGIN
    checking_account := 2000.00;
    savings_account := 1000000.00;
END; /
```

```
DECLARE
*
ERROR
at line 1:
ORA-06502: PL/SQL: numeric or value error: number precision too large
ORA-06512: at line 9
```

exemple, les trois sous-types contraints ont le même type de base. Les deux premiers sous-types peuvent être implicitement convertis en troisième sous-type, mais pas entre eux.

```
DECLARE
    SUBTYPE Digit IS PLS_INTEGER RANGE 0..9;
    SUBTYPE Double_digit IS PLS_INTEGER RANGE 10..99;
    SUBTYPE Under_100 IS PLS_INTEGER RANGE 0..99;

    d Digit := 4;
    dd Double_digit := 35;
    u Under_100;
BEGIN
    u := d; -- Réussit ; La plage Under_100 inclut la plage de chiffres
    u := dd; -- Réussit ; La plage Under_100 inclut la plage à deux chiffres
    dd := d; -- Génère une erreur ; La plage à deux chiffres n'inclut pas la plage de chiffres
END; /
```

Résultat

```
DECLARE
*
ERROR
at line 1: ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 12
```

Sous-types avec types de base dans la même famille de types de données

- Si deux sous-types ont des types de base différents dans la même famille de types de données, un sous-type peut être implicitement converti en l'autre uniquement si la valeur source ne viole pas une contrainte du sous-type cible.
- Si deux sous-types ont des types de base différents dans la même famille de types de données, un sous-type peut être implicitement converti en l'autre uniquement si la valeur source ne viole pas une contrainte du sous-type cible.

Conversion implicite entre sous-types avec des types de base dans la même famille

```
DECLARE
    SUBTYPE Word IS CHAR(6);
    SUBTYPE Text IS VARCHAR2(15);
    verb Word := 'run';
    sentence1 Text;
    sentence2 Text := 'Hurry!';
    sentence3 Text := 'See Tom run.';

BEGIN

    sentence1 := verb; -- Valeur à 3 caractères, limite à 15 caractères
    verb := sentence2; -- Valeur à 6 caractères, limite à 6 caractères
    verb := sentence3; -- Valeur de 12 caractères, limite de 6 caractères

END;
/
```

Résultat

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 13
```



instructions de contrôle PL/SQL

catégories PL/SQL des instructions de contrôle

- **Instructions de sélection conditionnelle** , qui exécutent différentes instructions pour différentes valeurs de données(IF et CASE).
- **Instructions de boucle** , qui exécutent les mêmes instructions avec une série de valeurs de données différentes(LOOP, FOR LOOP et WHILE LOOP).
- **Instructions de contrôle séquentielles** , qui ne sont pas cruciales pour la programmation PL/SQL(GOTO, qui va à une instruction spécifiée, et NULL, qui ne fait rien).

Instructions de sélection conditionnelle

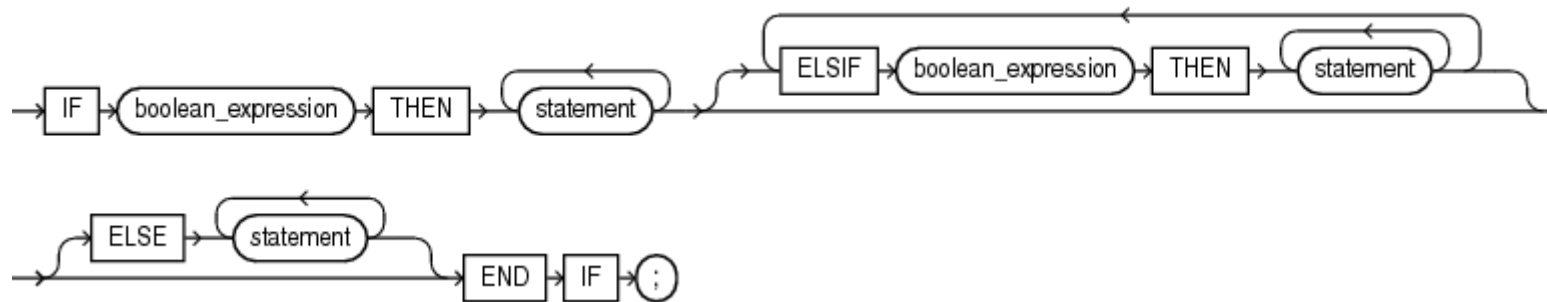
- Les **instructions de sélection conditionnelle** , IF et CASE, exécutent différentes instructions pour différentes valeurs de données.
- L' **IF** instruction exécute ou ignore une séquence d'une ou plusieurs instructions, selon une condition. La condition **IF** se déclare sous cette formes:
 - IF THEN
 - IF THEN ELSE
 - IF THEN ELSIF
- L'instruction **CASE** choisit parmi une séquence de conditions et exécute l'instruction correspondante. La condition **CASE** se déclare sous cette forme:
 - Simple, qui évalue une seule expression et la compare à plusieurs valeurs potentielles.
 - Searched, qui évalue plusieurs conditions et choisit la première qui est vraie.

Déclaration IF THEN

La déclaration de IF THEN a cette structure :

```
IF condition THEN  
    instructions  
END IF;
```

La déclaration complexe de IF THEN



Attention cas particulier

Évitez IF les déclarations maladroites telles que :

```
IF nouveau_solde < minimum_solde THEN
    a_decouvert := TRUE;
ELSE
    a_decouvert := FALSE;
END IF;
```

Une variable BOOLEAN est soit TRUE, FALSE ou NULL.
N'écris pas:

```
IF a_decouvert = TRUE THEN
    RAISE des_fonds_insuffisants ;
END IF;
```



Affectez plutôt la valeur de l'expression BOOLEAN
directement à une BOOLEAN variable :

```
à découvert := nouveau_solde < minimum_solde ;
```

Elles s'écrit plutôt :

```
IF a_decouvert THEN
    RAISE des_fonds_insuffisants ;
END IF;
```



Exemple Instruction IF THEN

```
DECLARE
  PROCEDURE p ( sales NUMBER, quota NUMBER, emp_id NUMBER ) IS
    bonus NUMBER := 0;
    updated VARCHAR2(3) := 'No';
  BEGIN
    IF sales > (quota + 200) THEN
      bonus := (sales - quota)/4;
      UPDATE employees SET salary = salary + bonus
        WHERE employee_id = emp_id;
      updated := 'Yes';
    END IF;
    DBMS_OUTPUT.PUT_LINE ( 'Table updated? ' || updated || ', ' ||
                          'bonus = ' || bonus || '.' );
  END p;
BEGIN
  p(10100, 10000, 120);
  p(10500, 10000, 121);
END;
```

résultat



Table updated? No, bonus = 0.
Table updated? Yes, bonus = 125.

Déclaration IF THEN ELSE

- La IF THEN ELSE a cette structure :


```
IF condition THEN  
    instructions  
ELSE  
    else_instructions  
END IF;
```

- Si la valeur de *condition* est vrai, le *instructions* s'exécute ; sinon, la *else_instructions* s'exécute .

Exemple Instruction IF THEN

```
DECLARE
  PROCEDURE p ( sales NUMBER, quota NUMBER, emp_id NUMBER ) IS
    bonus NUMBER := 0;
  BEGIN
    IF sales > (quota + 200) THEN
      bonus := (sales - quota)/4;
    ELSE
      bonus := 50;
    END IF;
    DBMS_OUTPUT.PUT_LINE ( 'bonus = ' || bonus );
    UPDATE employees
    SET salary = salary + bonus
    WHERE employee_id = emp_id;
  END p;
BEGIN
  p(10100, 10000, 120);
  p(10500, 10000, 121);
END;
```

résultat



```
bonus = 50
bonus = 125
```

Exemple Instructions IF THEN ELSE imbriquées

```
DECLARE
  PROCEDURE p ( sales NUMBER, quota NUMBER, emp_id NUMBER ) IS
    bonus NUMBER := 0;
  BEGIN
    IF sales > (quota + 200) THEN
      bonus := (sales - quota)/4;
    ELSE
      IF sales > quota THEN
        bonus := 50;
      ELSE
        bonus := 0;
      END IF;
    END IF;
    DBMS_OUTPUT.PUT_LINE ( 'bonus = ' || bonus );
    UPDATE employees SET salary = salary + bonus WHERE employee_id = emp_id;
  END p;
BEGIN
  p(10100, 10000, 120);
  p(10500, 10000, 121);
  p(9500, 10000, 122);
END;
```

résultat



```
bonus = 50
bonus = 125
bonus = 0
```

Déclaration IF THEN ELIF

La déclaration de IF THEN ELIF a cette structure :

```
IF condition_1 THEN
    statements_1
ELIF condition_2 THEN
    statements_2
[ ELIF condition_3 THEN
    statements_3 ]...
[ ELSE
    else_statements ]
END IF;
```

L' instruction IF THEN ELIF exécute la première *statements* pour laquelle *condition* est vrai. Les conditions restantes ne sont pas évaluées. Si non *condition* est vrai, la *else_statements* s'exécute , s'ils existent ; sinon, l'instruction IF THEN ELIF ne fait rien.

Simplification IF THEN ELSIF VS IF THEN ELSE

-- Instructions IF THEN ELSIF

```
IF condition_1 THEN statements_1;  
ELSIF condition_2 THEN statements_2;  
ELSIF condition_3 THEN statement_3;  
END IF;
```

-- Instructions IF THEN ELSE imbriquées logiquement équivalentes

```
IF condition_1 THEN  
    statements_1;  
ELSE  
    IF condition_2 THEN  
        statements_2;  
    ELSE  
        IF condition_3 THEN  
            statements_3;  
        END IF;  
    END IF;  
END IF;
```

Exemple Instructions IF THEN ELSIF

```
DECLARE
  PROCEDURE p (sales NUMBER) IS
    bonus NUMBER := 0;
  BEGIN
    IF sales > 50000 THEN
      bonus := 1500;
    ELSIF sales > 35000 THEN
      bonus := 500;
    ELSE
      bonus := 100;
    END IF;
    DBMS_OUTPUT.PUT_LINE ( 'Sales = ' || sales || ', bonus = ' || bonus || ' ');
  END p;
BEGIN
  p(55000);
  p(40000);
  p(30000);
END;
/
```

résultat



Sales = 55000, bonus = 1500.
Sales = 40000, bonus = 500.
Sales = 30000, bonus = 100.

Exemple Instructions IF THEN ELSIF simule une instruction CASE simple

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';
  IF grade = 'A' THEN
    DBMS_OUTPUT.PUT_LINE('Excellent');
  ELSIF grade = 'B' THEN
    DBMS_OUTPUT.PUT_LINE('Very Good');
  ELSIF grade = 'C' THEN
    DBMS_OUTPUT.PUT_LINE('Good');
  ELSIF grade = 'D' THEN
    DBMS_OUTPUT.PUT_LINE('Fair');
  ELSIF grade = 'F' THEN
    DBMS_OUTPUT.PUT_LINE('Poor');
  ELSE
    DBMS_OUTPUT.PUT_LINE('No such grade');
  END IF;
END; /
```

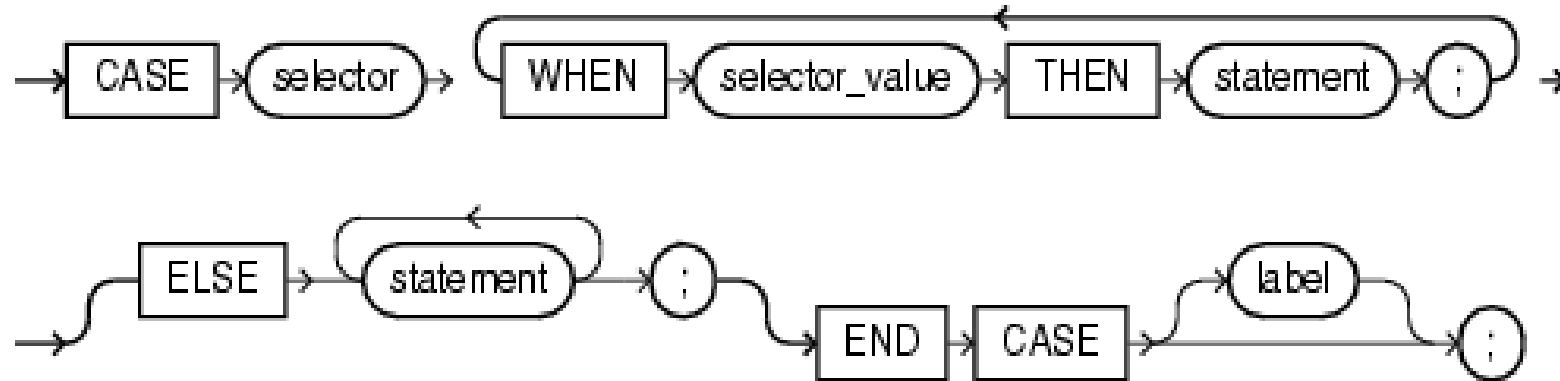
résultat

Very Good

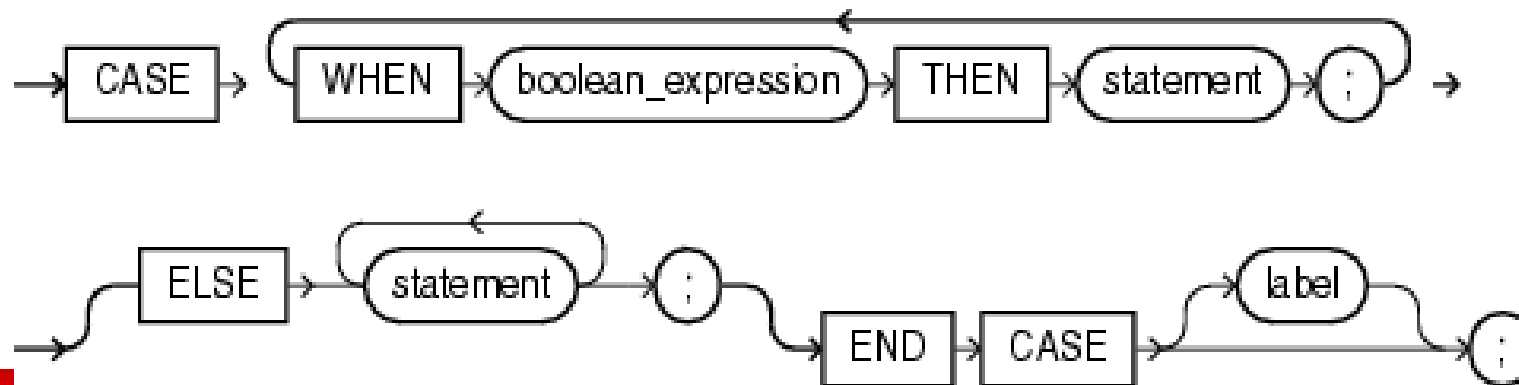


Déclaration de CASE Simple et recherche

-----*simple_case_statement*



-----*searched_case_statement*



Instruction CASE simple

- La déclaration simple CASE a cette structure :

```
CASE selector  
WHEN selector_value_1 THEN statements_1  
WHEN selector_value_2 THEN statements_2  
...  
WHEN selector_value_n THEN statements_n  
[ ELSE else_statements ]  
END CASE;
```

- *selectore* est une expression (généralement une seule variable) . Chacun *selector_value* peut être un littéral ou une expression.

Exemple Instruction CASE simple

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';
  CASE grade
    WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END; /
```



Résultat

Very Good

Déclaration CASE recherchée

- La déclaration CASE recherchée a cette structure :

```
CASE
  WHEN condition_1 THEN statements_1
  WHEN condition_2 THEN statements_2
  ...
  WHEN condition_n THEN statements_n
  [ ELSE
    else_statements ]
END CASE;
```

- L'instruction CASE recherchée exécute la première *statements* pour laquelle *condition* est vrai. Les conditions restantes ne sont pas évaluées. Si non *condition* est vrai, l'instruction CASE s'exécute *else_statements* si elles existent et déclenche l'exception prédéfinie dans le CASE_NOT_FOUND cas contraire.

Exemple Instruction CASE recherchée

```
DECLARE
    grade CHAR(1);
BEGIN
    grade := 'B';
    CASE
        WHEN grade = 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
        WHEN grade = 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
        WHEN grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
        WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
        WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
        ELSE
            DBMS_OUTPUT.PUT_LINE('No such grade');
        END CASE;
END;
/
```

Résultat



Very Good

EXCEPTION au lieu de la clause ELSE dans l'instruction CASE

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';
  CASE
    WHEN grade = 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN grade = 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
  END CASE;
EXCEPTION
  WHEN CASE_NOT_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No such grade');
END;
/
```

Résultat



Very Good

Instructions de boucle

- **Les instructions de boucle** exécutent les mêmes instructions avec une série de valeurs différentes. Les instructions de boucle sont :
 - LOOP de base
 - FOR LOOP
 - Le curseur FOR LOOP
 - WHILE LOOP
- Les instructions qui sortent d'une boucle sont :
 - EXIT
 - EXIT WHEN
- Les instructions qui sortent de l'itération courante d'une boucle sont :
 - CONTINUE
 - CONTINUE WHEN

Instruction LOOP de base

- La déclaration de base a cette structure LOOP

```
[étiquette ] LOOP  
    instructions  
END LOOP [étiquette ];
```

- À chaque itération de la boucle, l' *statement* s'exécute et le contrôle revient au sommet de la boucle. Pour empêcher une boucle infinie, une instruction ou une exception déclenchée doit quitter la boucle.

Déclaration de sortie

- L' instruction EXIT quitte l'itération en cours d'une boucle sans condition et transfère le contrôle à la fin de la boucle en cours ou d'une boucle étiquetée englobante.

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop: x = ' || TO_CHAR(x)); x := x + 1;
    IF x > 3 THEN
      EXIT;
    END IF;
  END LOOP;
  -- After EXIT, control resumes here
  DBMS_OUTPUT.PUT_LINE(' After loop: x = ' || TO_CHAR(x));
END;
/
```



Inside loop: x = 0
Inside loop: x = 1
Inside loop: x = 2
Inside loop: x = 3
After loop: x = 4

Instruction **EXIT WHEN**

- L' instruction EXIT WHEN quitte l'itération actuelle d'une boucle lorsque la condition de sa clause WHEN est vraie et transfère le contrôle à la fin de la boucle actuelle ou d'une boucle étiquetée englobante.

Instruction LOOP de base avec instruction EXIT WHEN

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Inside loop: x = ' || TO_CHAR(x));
    x := x + 1; -- prevents infinite loop
    EXIT WHEN x > 3;
  END LOOP;
  -- After EXIT statement, control resumes here
  DBMS_OUTPUT.PUT_LINE('After loop: x = ' || TO_CHAR(x));
END; /
```



```
Inside loop: x = 0
Inside loop: x = 1
Inside loop: x = 2
Inside loop: x = 3
After loop: x = 4
```

Instructions LOOP de base étiquetées imbriquées avec instructions EXIT WHEN

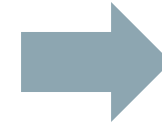
```
DECLARE
  s PLS_INTEGER := 0;
  i PLS_INTEGER := 0;
  j PLS_INTEGER;
BEGIN
  <<outer_loop>>
  LOOP
    i := i + 1;
    j := 0;
    <<inner_loop>>
    LOOP
      j := j + 1;
      s := s + i * j; -- Sum several products
      EXIT inner_loop WHEN (j > 5);
      EXIT outer_loop WHEN ((i * j) > 15);
    END LOOP inner_loop;
  END LOOP outer_loop;
  DBMS_OUTPUT.PUT_LINE ('The sum of products equals: ' || TO_CHAR(s));
END;
/
```



The sum of products equals: 166

Instruction CONTINUE dans l'instruction LOOP de base

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP -- After CONTINUE statement, control resumes here
    DBMS_OUTPUT.PUT_LINE ('Inside loop: x = ' || TO_CHAR(x));
    x := x + 1;
    CONTINUE WHEN x < 3;
    DBMS_OUTPUT.PUT_LINE
      ('Inside loop, after CONTINUE: x = ' || TO_CHAR(x));
    EXIT WHEN x = 5;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE (' After loop: x = ' || TO_CHAR(x));
END;
/
```



```
Inside loop: x = 0
Inside loop: x = 1
Inside loop: x = 2
Inside loop, after CONTINUE: x = 3
Inside loop: x = 3
Inside loop, after CONTINUE: x = 4
Inside loop: x = 4
Inside loop, after CONTINUE: x = 5
After loop: x = 5
```

Instruction FOR LOOP

L' `FOR LOOP` instruction exécute une ou plusieurs instructions pendant que l'index de boucle se trouve dans une plage spécifiée. La déclaration a cette structure :

```
[ label ] FOR index IN [ REVERSE ] lower_bound..upper_bound LOOP  
    statements  
END LOOP [ label ];
```

Sans `REVERSE`, la valeur de *index* commence à *lower_bound* et augmente de un à chaque itération de la boucle jusqu'à ce qu'elle atteigne *upper_bound*. Si *lower_bound* est supérieur à *upper_bound*, alors *statements* ne s'exécute jamais.

Avec `REVERSE`, la valeur de *index* commence à *upper_bound* et diminue de un à chaque itération de la boucle jusqu'à ce qu'elle atteigne *lower_bound*. Si *upper_bound* est inférieur à *lower_bound*, alors *statements* ne s'exécute jamais. Un `EXIT`, `EXIT WHEN`, `CONTINUE` ou `CONTINUE WHEN` dans le *statements* peut entraîner la fin prématurée de la boucle ou de l'itération en cours de la boucle.

Instructions FOR LOOP

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('lower_bound < upper_bound');

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

  DBMS_OUTPUT.PUT_LINE ('lower_bound = upper_bound');

  FOR i IN 2..2 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('lower_bound > upper_bound');

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



```
lower_bound < upper_bound
1
2
3
lower_bound = upper_bound
2
lower_bound > upper_bound
```

les instructions FOR LOOP Inverser

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('upper_bound > lower_bound');

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

  DBMS_OUTPUT.PUT_LINE ('upper_bound = lower_bound');

  FOR i IN REVERSE 2..2 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('upper_bound < lower_bound');

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



```
upper_bound > lower_bound
3
2
1
upper_bound = lower_bound
2
upper_bound < lower_bound
```

Simulation de la clause STEP dans l'instruction FOR LOOP

```
DECLARE
    step PLS_INTEGER := 5;
BEGIN
    FOR i IN 1..3 LOOP
        DBMS_OUTPUT.PUT_LINE (i*step);
    END LOOP;
END;
/
```



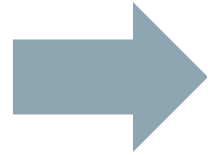
```
5
10
15
```

FOR LOOP Index

L'index de l'instruction FOR LOOP est implicitement déclaré comme une variable de type PLS_INTEGER local à la boucle. Les instructions de la boucle peuvent lire la valeur de l'index, mais ne peuvent pas la modifier. Les instructions en dehors de la boucle ne peuvent pas référencer l'index. Après l'exécution FOR LOOP instruction, l'index n'est pas défini. (Un index de boucle est parfois appelé compteur de boucle.)

L'instruction FOR LOOP tente de modifier la valeur de l'index

```
BEGIN
  FOR i IN 1..3 LOOP
    IF i < 3 THEN
      DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
    ELSE
      i := 2;
    END IF;
  END LOOP;
END;
/
```



```
      i := 2;
      *
```

ERROR at line 6:
ORA-06550: line 6, column 8:
PLS-00363: expression 'I' cannot be used as an assignment target
ORA-06550: line 6, column 8:
PL/SQL: Statement ignored

Références d'instructions extérieures *FOR LOOP* Index d'instructions

```
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop, i is ' || TO_CHAR(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
END;
/
```



```
DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
```

*

ERROR at line 6:

ORA-06550: line 6, column 58:

PLS-00201: identifier 'I' must be declared

ORA-06550: line 6, column 3:

PL/SQL: Statement ignored

Borne inférieure et borne supérieure

Les limites inférieure et supérieure d'une FOR LOOP instruction peuvent être des littéraux numériques, des variables numériques ou des expressions numériques. Si une borne n'a pas de valeur numérique, alors PL/SQL déclenche l'exception prédéfinie `VALUE_ERROR`.

Limites de l'instruction FOR LOOP

```
DECLARE
    first    INTEGER := 1;
    last     INTEGER := 10;
    high     INTEGER := 100;
    low      INTEGER := 12;
BEGIN
    -- Bounds are numeric literals:
    FOR j IN -5..5 LOOP
        NULL;
    END LOOP;

    -- Bounds are numeric variables:
    FOR k IN REVERSE first..last LOOP
        NULL;
    END LOOP;

    -- Lower bound is numeric literal,
    -- Upper bound is numeric expression:
    FOR step IN 0..(TRUNC(high/low) * 2) LOOP
        NULL;
    END LOOP;
END;
/
```

pécification des limites de l'instruction FOR LOOP au moment de l'exécution

```
DROP TABLE temp;
CREATE TABLE temp (
    emp_no          NUMBER,
    email_addr      VARCHAR2(50)
);

DECLARE
    emp_count  NUMBER;
BEGIN
    SELECT COUNT(employee_id) INTO emp_count
    FROM employees;

    FOR i IN 1..emp_count LOOP
        INSERT INTO temp (emp_no, email_addr)
        VALUES(i, 'to be added later');
    END LOOP;
END;
/
```

Instruction EXIT WHEN ou CONTINUE WHEN dans l'instruction FOR LOOP

Supposons que vous deviez quitter l'instruction FOR LOOP immédiatement si une certaine condition se présente. Vous pouvez placer la condition dans une instruction EXIT WHEN à l'intérieur de l' FOR LOOP instruction.

Instruction EXIT WHEN dans l'instruction FOR LOOP

```
DECLARE
    v_employees employees%ROWTYPE;
    CURSOR c1 is SELECT * FROM employees;
BEGIN
    OPEN c1;
    -- Fetch entire row into v_employees record:
    FOR i IN 1..10 LOOP
        FETCH c1 INTO v_employees;
        EXIT WHEN c1%NOTFOUND;
        -- Process data here
    END LOOP;
    CLOSE c1;
END;
/
```

Instruction EXIT WHEN dans l'instruction interne FOR LOOP

```
DECLARE
    v_employees employees%ROWTYPE;
    CURSOR c1 is SELECT * FROM employees;
BEGIN
    OPEN c1;

    -- Fetch entire row into v_employees record:
    <<outer_loop>>
    FOR i IN 1..10 LOOP
        -- Process data here
        FOR j IN 1..10 LOOP
            FETCH c1 INTO v_employees;
            EXIT outer_loop WHEN c1%NOTFOUND;
            -- Process data here
        END LOOP;
    END LOOP outer_loop;

    CLOSE c1;
END;
/
```


*Instruction **CONTINUE WHEN** dans l'instruction interne **FOR LOOP***

```
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 IS SELECT * FROM employees;
BEGIN
  OPEN c1;

  -- Fetch entire row into v_employees record:
  <<outer_loop>>
  FOR i IN 1..10 LOOP
    -- Process data here
    FOR j IN 1..10 LOOP
      FETCH c1 INTO v_employees;
      CONTINUE outer_loop WHEN c1%NOTFOUND;
      -- Process data here
    END LOOP;
  END LOOP outer_loop;

  CLOSE c1;
END;
/
```

Instruction WHILE LOOP

L' instruction WHILE LOOP exécute une ou plusieurs instructions tant qu'une condition est vraie. Il a cette structure :

```
[ étiquette ] WHILE condition LOOP  
    instructions  
END LOOP [ étiquette ];
```

Si le *condition* est vrai, l'exécution de l'instruction et le contrôle retournent au début de la boucle, où *condition* est à nouveau évalué. Si le *condition* n'est pas vrai, le contrôle est transféré à l'instruction après l'instruction WHILE LOOP. Pour éviter une boucle infinie, une instruction à l'intérieur de la boucle doit rendre la condition fausse ou nulle. Pour la syntaxe complète .

Un EXIT, EXIT WHEN, CONTINUE ou CONTINUE WHEN dans les instructions peut entraîner la fin prématurée de la boucle ou de l'itération en cours de la boucle.

Certains langages ont une structure LOOP UNTIL ou REPEAT UNTIL, qui teste une condition en bas de la boucle plutôt qu'en haut, de sorte que les instructions s'exécutent au moins une fois. Pour simuler cette structure en PL/SQL, utilisez une instruction LOOP de base avec une instruction EXIT WHEN :

Instructions *WHILE LOOP*

```
DECLARE
  done  BOOLEAN := FALSE;
BEGIN
  WHILE done LOOP
    DBMS_OUTPUT.PUT_LINE ('This line does not print. ');
    done := TRUE;  -- This assignment is not made.
  END LOOP;

  WHILE NOT done LOOP
    DBMS_OUTPUT.PUT_LINE ('Hello, world! ');
    done := TRUE;
  END LOOP;
END;
/
```



Hello, world!

Instructions de contrôle séquentiel

Contrairement aux instructions IF et LOOP, les **instructions de contrôle séquentiel** GOTO et NULL ne sont pas cruciales pour la programmation PL/SQL.

L'instruction GOTO, qui va à une instruction spécifiée, est rarement nécessaire. Parfois, il simplifie suffisamment la logique pour justifier son utilisation.

L'énoncé NULL, qui ne fait rien, peut améliorer la lisibilité en clarifiant le sens et l'action des énoncés conditionnels.

Les sujets

GOTO Déclaration

Instruction NULL

GOTO Déclaration

L'instruction GOTO transfère le contrôle à une étiquette sans condition. L'étiquette doit être unique dans sa portée et doit précéder une instruction exécutable ou un bloc PL/SQL. Lorsqu'elle est exécutée, l'instruction GOTO transfère le contrôle à l'instruction ou au bloc étiqueté. Pour GOTO les restrictions d'instruction .

Utilisez les instructions GOTO avec parcimonie, leur utilisation excessive aboutit à un code difficile à comprendre et à maintenir. N'utilisez pas l'instruction GOTO pour transférer le contrôle d'une structure profondément imbriquée vers un gestionnaire d'exceptions. Au lieu de cela, déclenchez une exception. Pour plus d'informations sur le mécanisme de gestion des exceptions PL/SQL

Instruction GOTO

```
DECLARE
  p  VARCHAR2(30);
  n  PLS_INTEGER := 37;
BEGIN
  FOR j in 2..ROUND(SQRT(n)) LOOP
    IF n MOD j = 0 THEN
      p := ' is not a prime number';
      GOTO print_now;
    END IF;
  END LOOP;

  p := ' is a prime number';

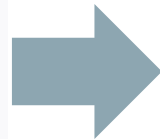
  <<print_now>>
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/
```



37 is a prime number

Placement incorrect de l'étiquette

```
DECLARE
  done  BOOLEAN;
BEGIN
  FOR i IN 1..50 LOOP
    IF done THEN
      GOTO end_loop;
    END IF;
    <<end_loop>>
  END LOOP;
END;
/
```



```
END LOOP;
```

```
*
```

ERROR at line 9:

ORA-06550: line 9, column 3:

PLS-00103: Encountered the symbol "END" when expecting one of the following:

(begin case declare exit for goto if loop mod null raise

return select update while with <an identifier>

<a double-quoted delimited-identifier> <a bind variable> <<

continue close current delete fetch lock insert open rollback

savepoint set sql run commit forall merge pipe purge

L'instruction GOTO passe à l'instruction NULL étiquetée

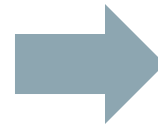
```
DECLARE
    done    BOOLEAN;
BEGIN
    FOR i IN 1..50 LOOP
        IF done THEN
            GOTO end_loop;
        END IF;
        <<end_loop>>
        NULL;
    END LOOP;
END;
```


L'instruction GOTO transfère le contrôle au bloc englobant

```
DECLARE
  v_last_name  VARCHAR2(25);
  v_emp_id     NUMBER(6) := 120;
BEGIN
  <<get_name>>
  SELECT last_name INTO v_last_name
  FROM employees
  WHERE employee_id = v_emp_id;

  BEGIN
    DBMS_OUTPUT.PUT_LINE (v_last_name);
    v_emp_id := v_emp_id + 5;

    IF v_emp_id < 120 THEN
      GOTO get_name;
    END IF;
  END;
END;
/
```



Weiss

L'instruction GOTO ne peut pas transférer le contrôle dans l'instruction IF

```
DECLARE
    valid BOOLEAN := TRUE;
BEGIN
    GOTO update_row;

    IF valid THEN
        <<update_row>>
        NULL;
    END IF;
END;
/
```



```
GOTO update_row;
*
ERROR at line 4:
ORA-06550: line 4, column 3:
PLS-00375: illegal GOTO statement; this GOTO cannot transfer control to label
'UPDATE_ROW'
ORA-06550: line 6, column 12:
PL/SQL: Statement ignored
```

Instruction NULL

L'instruction NULL ne passe le contrôle qu'à l'instruction suivante. Certaines langues se réfèrent à une telle instruction comme un no-op (pas d'opération).

Certaines utilisations de l'instruction NULL sont :

- Pour fournir une cible pour une instruction GOTO .

- Pour améliorer la lisibilité en clarifiant le sens et l'action des instructions conditionnelles.

- Pour créer des espaces réservés et des sous-programmes de remplacement

- Pour montrer que vous êtes conscient d'une possibilité, mais qu'aucune action n'est nécessaire

Instruction NULL ne montrant aucune action

```
DECLARE
    v_job_id  VARCHAR2(10);
    v_emp_id  NUMBER(6) := 110;
BEGIN
    SELECT job_id INTO v_job_id
    FROM employees
    WHERE employee_id = v_emp_id;

    IF v_job_id = 'SA_REP' THEN
        UPDATE employees
        SET commission_pct = commission_pct * 1.2;
    ELSE
        NULL; -- Employee is not a sales rep
    END IF;
END;
/
```

Instruction NULL comme espace réservé lors de la création d'un sous-programme

```
CREATE OR REPLACE PROCEDURE award_bonus (  
    emp_id NUMBER,  
    bonus NUMBER  
) AUTHID DEFINER AS  
BEGIN    -- Executable part starts here  
    NULL; -- Placeholder  
    -- (raises "unreachable code" if warnings enabled)  
END award_bonus;  
/
```

Instruction NULL dans la clause ELSE de l'instruction CASE simple

```
CREATE OR REPLACE PROCEDURE print_grade (  
    grade CHAR  
) AUTHID DEFINER AS  
BEGIN  
    CASE grade  
        WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');  
        WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');  
        WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');  
        WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');  
        WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');  
        ELSE NULL;  
    END CASE;  
END;  
/  
BEGIN  
    print_grade('A');  
    print_grade('S');  
END;  
/
```



Excellent

Procédures et fonctions

PL/SQL est aussi utilisé pour définir des procédures et fonctions stockées dans la BD.

Syntaxe: Procédure

CREATE [OR REPLACE] PROCEDURE

Nom_procédure [(*liste d'arguments*)] { **IS** | **AS** }

[variables locales]

Corps PL-SQL

Syntaxe: Fonction

CREATE [OR REPLACE] FUNCTION

Nom_fonction [(liste d'arguments)] **RETURN** *type* { **IS** | **AS** }

[variables locales]

Corps PL-SQL

Procédures et fonctions

L'option OR **REPLACE** permet de spécifier au système le remplacement de la procédure ou de la fonction si elle existe déjà dans la BD.

Liste d'arguments: nom_arg [**IN** | **OUT** | **IN OUT**] Type

- ❖ **IN**: la variable est passée en entrée
- ❖ **OUT**: la variable est renseignée par la procédure puis renvoyée à l'appelant
- ❖ **IN OUT**: passage par référence.

le mot **RETURN** permet de spécifier le type de la donnée de retour. Le corps PL/SQL doit commencer par le mot clé **BEGIN** et se termine par **END**. Il peut être composé d'une partie déclarative , d'un corps de la procédure et d'un gestionnaire d'erreurs.

PROCÉDURES

- Une procédure est un bloc nommé, éventuellement paramétré, qu'on peut exécuter à la demande.

```
create or replace procedure bonjour  
is  
begin dbms_output.put_line('Hello');  
end;
```

- On appelle une procédure PLSQL par son nom. On peut le faire directement depuis SQLPlus avec la commande execute

```
SQL> execute bonjour;  
Hello  
  
PL/SQL procedure successfully completed.
```

EXEMPLE PROCÉDURE

- Exemple : Créer une procédure qui permet d'augmenter le salaire d'un employé de la table employees

```
CREATE PROCEDURE augment_salaire(p_employe_id IN  NUMBER, Taux IN NUMBER ) IS
BEGIN
    if Taux <1      then
        UPDATE employees SET  salary= salary* ( 1 - Taux) WHERE id=
            p_employe_id ;
    end if;
END ;
```

Exemple FONCTION

Exemple : Créer une fonction qui retourne le nom d'un employee

```
CREATE FUNCTION NomEmployee( id IN Number) RETURN Varchar2(30) IS
    S employees.first_name%Type;
BEGIN
    Select first_name into S from employees Where
        employees.employee_id = id;
    Return (S);
EXCEPTION
    WHEN NO_DATA_FOUND THEN Return ('Aucun');
END ;
```

Modification d'une procédure (fonction)

- Si la base de données évolue, il faut recompiler les procédures existantes pour qu'elles tiennent compte de ces modifications.

La commande est la suivante:

ALTER { FUNCTION | PROCEDURE } nom COMPILE

Exemple:

```
ALTER PROCEDURE augment_salaire COMPILE;  
ALTER FUNCTION NomEmployee COMPILE;
```

Suppression d'une procédure (fonction)

Pour supprimer une procédure

DROP { FUNCTION | PROCEDURE } nom

Exemple:

```
DROP PROCEDURE augment_salaire;
```

Les collections et les enregistrements PL/SQL

- PL/SQL vous permet de définir deux types de types de données composites, la collecte et l'enregistrement.
 - Un **type de données composite** stocke des valeurs qui ont des composants internes.
 - Les composants internes peuvent être scalaires ou composites.
 - ❖ Dans une **collection** , les composants internes ont toujours le même type de données et sont appelés **éléments** ,Vous pouvez accéder à chaque élément d'une variable de collection par son index unique.
 - ❖ Dans un **enregistrement** , les composants internes peuvent avoir différents types de données et sont appelés **champs** , Vous pouvez accéder à chaque champ d'une variable d'enregistrement par son nom

NB : Vous pouvez créer une collection d'enregistrements et un enregistrement contenant des collections.

Tableau associatif indexé par chaîne

est un ensemble de paires clé-valeur. Chaque clé est un index unique, utilisé pour localiser la valeur associée à la syntaxe `.variable_name(index)`

```
DECLARE
  -- Associative array indexed by string:
  TYPE population IS TABLE OF NUMBER -- Associative array type
    INDEX BY VARCHAR2(64); -- indexed by string
  city_population population; -- Associative array variable
  i VARCHAR2(64); -- Scalar variable
BEGIN
  -- Add elements (key-value pairs) to associative array:
  city_population('Smallville') := 2000;
  city_population('Midland') := 750000;
  city_population('Megalopolis') := 1000000;
  -- Change value associated with key 'Smallville':
  city_population('Smallville') := 2001;
  -- Print associative array:
  i := city_population.FIRST; -- Get first element of array
  WHILE i IS NOT NULL LOOP
    DBMS_Output.PUT_LINE ('Population of ' || i || ' is ' || city_population(i));
    i := city_population.NEXT(i); -- Get next element of array
  END LOOP;
END;
```



Population of Megalopolis is 1000000
Population of Midland is 750000
Population of Smallville is 2001

La fonction renvoie un tableau associatif indexé par PLS_INTEGER

Cet exemple définit un type de tableau associatif indexé par PLS_INTEGER et une fonction qui renvoie un tableau associatif de ce type.

```
DECLARE
    TYPE sum_multiples IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER;
    n PLS_INTEGER := 5; -- number of multiples to sum for display
    sn PLS_INTEGER := 10; -- number of multiples to sum
    m PLS_INTEGER := 3; -- multiple
    FUNCTION get_sum_multiples ( multiple IN PLS_INTEGER, num IN PLS_INTEGER ) RETURN sum_multiples IS
        s sum_multiples;
    BEGIN
        FOR i IN 1..num LOOP
            s(i) := multiple * ((i * (i + 1)) / 2); -- sum of multiples
        END LOOP;
        RETURN s;
    END get_sum_multiples;
BEGIN
    DBMS_OUTPUT.PUT_LINE ( 'Sum of the first ' || TO_CHAR(n) || ' multiples of ' || TO_CHAR(m) || ' is ' ||
        TO_CHAR(get_sum_multiples (m, sn)(n)) );
END; /
```



Sum of the first 5 multiples of 3 is 45

Déclaration d'une constante de tableau associatif

Lors de la déclaration d'une constante de tableau associatif, vous devez créer une fonction qui remplit le tableau associatif avec sa valeur initiale, puis invoquer la fonction dans la déclaration de la constante.

```
CREATE OR REPLACE PACKAGE My_Types AUTHID CURRENT_USER IS
  TYPE My_AA IS TABLE OF VARCHAR2(20) INDEX BY PLS_INTEGER;
  FUNCTION Init_My_AA RETURN My_AA;
END My_Types;
/
CREATE OR REPLACE PACKAGE BODY My_Types IS
  FUNCTION Init_My_AA RETURN My_AA IS
    Ret My_AA;
  BEGIN
    Ret(-10) := '-ten'; Ret(0) := 'zero'; Ret(1) := 'one';
    Ret(2) := 'two'; Ret(3) := 'three'; Ret(4) := 'four'; Ret(9) := 'nine';
    RETURN Ret;
  END Init_My_AA;
END My_Types;
/
```

```
DECLARE
  v CONSTANT My_Types.My_AA := My_Types.Init_My_AA();
BEGIN
  DECLARE
    Idx PLS_INTEGER := v.FIRST();
  BEGIN
    WHILE Idx IS NOT NULL LOOP
      DBMS_OUTPUT.PUT_LINE(TO_CHAR(Idx, '999') || LPAD(v(Idx), 7));
      Idx := v.NEXT(Idx);
    END LOOP;
  END;
END;
```



-10 -ten 0 zero 1 one 2 two 3 three 4 four 9 nine

Collections multidimensionnelles

Bien qu'une collection n'ait qu'une seule dimension, vous pouvez modéliser une collection multidimensionnelle avec une collection dont les éléments sont des collections.

DECLARE

```
TYPE t1 IS VARRAY(10) OF INTEGER; -- varray of integer
```

```
va t1 := t1(2,3,5);
```

```
TYPE nt1 IS VARRAY(10) OF t1; -- varray of varray of integer
```

```
nva nt1 := nt1(va, t1(55,6,73), t1(2,4), va);
```

```
i INTEGER;
```

```
va1 t1;
```

BEGIN

```
i := nva(2)(3);
```

```
DBMS_OUTPUT.PUT_LINE('i = ' || i); nva.EXTEND;
```

```
nva(5) := t1(56, 32); -- replace inner varray elements
```

```
nva(4) := t1(45,43,67,43345); -- replace an inner integer element
```

```
nva(4)(4) := 1; -- replace 43345 with 1
```

```
nva(4).EXTEND; -- add element to 4th varray element
```

```
nva(4)(5) := 89; -- store integer 89 there
```

END;



Les Déclencheurs PL/SQL

Objectifs

A la fin de ce chapitre, vous pourrez :

- décrire différents types de déclencheur
- décrire les déclencheurs de base de données et leur utilisation
- créer des déclencheurs de base de données
- décrire les règles d'activation des déclencheurs de base de données
- supprimer des déclencheurs de base de données

Types de déclencheur

Un déclencheur :

- est une procédure ou un bloc PL/SQL associé à la base de données, à une table, à une vue ou à un schéma
- s'exécute de façon implicite lorsqu'un événement donné se produit
- il peut s'agir d'un :
 - déclencheur applicatif, qui s'exécute lorsqu'un événement se produit dans une application donnée
 - déclencheur de base de données, qui s'exécute lorsqu'un événement de type données (LMD) ou système (connexion ou arrêt) se produit dans un schéma ou une base de données

Règles relatives à la conception de déclencheurs

- Il est conseillé de concevoir des déclencheurs pour :
 - exécuter des actions associées
 - centraliser des opérations globales
- Leur conception est à proscrire :
 - lorsque la fonctionnalité est déjà intégrée au serveur Oracle
 - lorsqu'ils constituent des doublons d'autres déclencheurs
- Si le code PL/SQL est très long, créer des procédures stockées et les appeler dans un déclencheur
- L'utilisation excessive de déclencheurs peut entraîner des interdépendances complexes dont la gestion peut s'avérer difficile dans les applications volumineuses

Exemple de déclencheur de base de données

```
INSERT INTO EMPLOYEES  
. . . ;
```

Table EMPLOYEES

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000
102	De Haan	AD_VP	17000
103	Hunold	IT_PROG	9000
...

Déclencheur CHECK_SAL



Créer des déclencheurs LMD

Une instruction de déclenchement comporte les éléments suivants :

- moment du déclenchement
 - pour une table : `BEFORE`, `AFTER`
 - pour une vue : `INSTEAD OF`
- événement déclencheur : `INSERT`, `UPDATE` ou `DELETE`
- nom de la table : sur la table ou la vue
- type de déclencheur : ligne ou instruction
- clause `WHEN` : condition restrictive
- corps du déclencheur : bloc PL/SQL

Composants des déclencheurs LMD

Moment du déclenchement : à quel moment le déclencheur doit-il s'exécuter ?

- BEFORE : exécution du corps du déclencheur avant le déclenchement de l'événement LMD sur une table
- AFTER : exécution du corps du déclencheur après le déclenchement de l'événement LMD sur une table
- INSTEAD OF : exécution du corps du déclencheur au lieu de l'instruction de déclenchement. Ce déclencheur est utilisé pour les vues qui ne peuvent pas être modifiées autrement

Composants des déclencheurs LMD

Événement utilisateur déclencheur : quelle instruction LMD entraîne l'exécution du déclencheur ? Vous pouvez utiliser les instructions suivantes :

- INSERT
- UPDATE
- DELETE

Composants des déclencheurs LMD

Type de déclencheur : le corps du déclencheur doit-il s'exécuter une seule fois ou pour chaque ligne concernée par l'instruction ?

- Instruction : le corps du déclencheur s'exécute une seule fois pour l'événement déclencheur. Il s'agit du comportement par défaut. Un déclencheur sur instruction s'exécute une fois, même si aucune ligne n'est affectée
- Ligne : le corps du déclencheur s'exécute une fois pour chaque ligne concernée par l'événement déclencheur. Un déclencheur sur ligne ne s'exécute pas si l'événement déclencheur n'affecte aucune ligne

Composants des déclencheurs LMD

Corps du déclencheur : quelle action le déclencheur doit-il effectuer ?

Le corps du déclencheur est un bloc PL/SQL ou un appel de procédure

Séquence d'exécution

Lorsque la manipulation concerne une seule ligne, utilisez la séquence d'exécution suivante pour un déclencheur sur une table :

Instruction LMD

```
INSERT INTO departments (department_id,  
                        department_name, location_id)  
VALUES (400, 'CONSULTING', 2400);
```

1 row created.

Action de déclenchement

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
30	Purchasing	1700
...		
400	CONSULTING	2400

→ Déclencheur sur
instruction BEFORE

→ Déclencheur sur ligne BEFORE

→ Déclencheur sur ligne AFTER

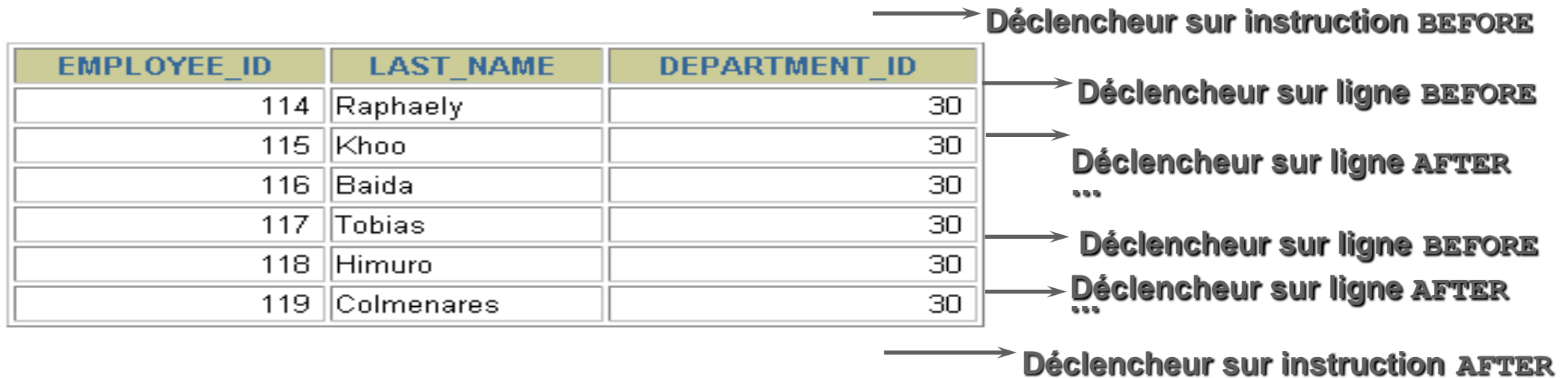
→ Déclencheur sur instruction AFTER

Séquence d'exécution

Lorsque la manipulation concerne plusieurs lignes, utilisez la séquence d'exécution suivante pour un déclencheur sur une table :

```
UPDATE employees
  SET salary = salary * 1.1
  WHERE department_id = 30;
```

6 rows updated.



Syntaxe pour la création de déclencheurs sur instruction LMD

Syntaxe :

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing
    event1 [OR event2 OR event3]
    ON table_name
    trigger_body
```

Remarque : Les noms des déclencheurs doivent être uniques au sein d'un même schéma

Créer des déclencheurs sur instruction LMD

Exemple :

```
CREATE OR REPLACE TRIGGER secure_emp
  BEFORE INSERT ON employees
  BEGIN
    IF (TO_CHAR(SYSDATE, 'DY') IN ('SAT', 'SUN')) OR
       (TO_CHAR(SYSDATE, 'HH24:MI')
        NOT BETWEEN '08:00' AND '18:00')
    THEN RAISE APPLICATION_ERROR (-20500, 'You may
insert into EMPLOYEES table only
business hours. ');
    END IF;
  END;
```

during

Trigger created.

Tester SECURE_EMP

```
INSERT INTO employees (employee_id, last_name,  
first_name, email, hire_date, job_id, salary,  
department_id)  
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE,  
'IT_PROG', 4500, 60);
```

```
INSERT INTO employees (employee_id, last_name, first_name, email,  
*  
ERROR at line 1:
```

ORA-20500: You may insert into EMPLOYEES table only during business hours.

ORA-06512: at "PLSQL.SECURE_EMP", line 4

ORA-04088: error during execution of trigger 'PLSQL.SECURE_EMP'

Utiliser des prédicats conditionnels

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT OR UPDATE OR DELETE ON employees
BEGIN
  IF (TO_CHAR (SYSDATE, 'DY') IN ('SAT', 'SUN')) OR
     (TO_CHAR (SYSDATE, 'HH24') NOT BETWEEN '08' AND '18')
  THEN
    IF DELETING THEN
      RAISE_APPLICATION_ERROR (-20502, 'You may delete from
EMPLOYEES table only during business hours. ');
    ELSIF INSERTING THEN
      RAISE_APPLICATION_ERROR (-20500, 'You may insert into
      EMPLOYEES table only during business hours. ');
    ELSIF UPDATING ('SALARY') THEN
      RAISE_APPLICATION_ERROR (-20503, 'You may update
      SALARY only during business hours. ');
    ELSE
      RAISE_APPLICATION_ERROR (-20504, 'You may update
      EMPLOYEES table only during normal hours. ');
    END IF;
  END IF;
END;
```

Créer un déclencheur sur ligne LMD

Syntaxe :

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing
    event1 [OR event2 OR event3]
    ON table_name
    [REFERENCING OLD AS old | NEW AS new]
FOR EACH ROW
    [WHEN (condition)]
    trigger_body
```

Créer des déclencheurs sur ligne LMD

```
CREATE OR REPLACE TRIGGER restrict_salary
  BEFORE INSERT OR UPDATE OF salary ON employees
  FOR EACH ROW
  BEGIN
    IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
      AND :NEW.salary > 15000
    THEN
      RAISE_APPLICATION_ERROR (-20202, 'Employee
        cannot earn this amount');
    END IF;
  END;
/
```

Trigger created.

Utiliser les qualificatifs OLD et NEW

```
CREATE OR REPLACE TRIGGER audit_emp_values
  AFTER DELETE OR INSERT OR UPDATE ON employees
  FOR EACH ROW
BEGIN
  INSERT INTO audit_emp_table (user_name, timestamp,
    id, old_last_name, new_last_name, old_title,
    new_title, old_salary, new_salary)
  VALUES (USER, SYSDATE, :OLD.employee_id,
    :OLD.last_name, :NEW.last_name, :OLD.job_id,
    :NEW.job_id, :OLD.salary, :NEW.salary );
END;
/
```

Trigger created.

Utiliser les qualificatifs OLD et NEW : exemple de la table Audit_Emp_Table

```
INSERT INTO employees
      (employee_id, last_name, job_id, salary, ...)
VALUES (999, 'Temp emp', 'SA_REP', 1000, ...);
```

```
UPDATE employees
      SET salary = 2000, last_name = 'Smith'
      WHERE employee_id = 999;
```

1 row created.
1 row updated.

```
SELECT user_name, timestamp, ... FROM audit_emp_table
```

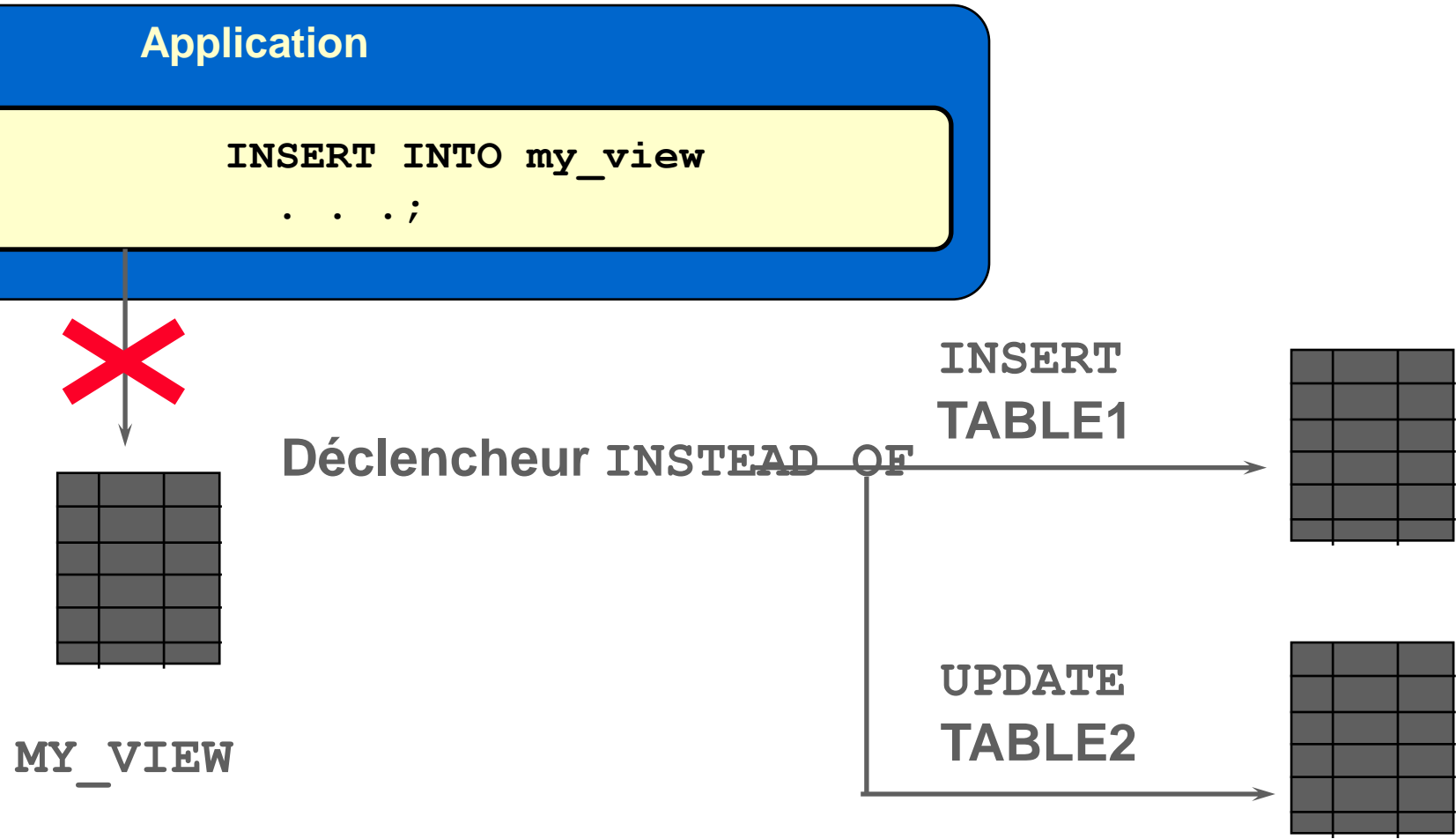
USER_NAME	TIMESTAMP	ID	OLD_LAST_N	NEW_LAST_N	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
PLSQL	28-SEP-01			Temp emp		SA_REP		1000
PLSQL	28-SEP-01	999	Temp emp	Smith	SA_REP	SA_REP	1000	2000

Restreindre l'action d'un déclencheur sur ligne

```
CREATE OR REPLACE TRIGGER derive_commission_pct
  BEFORE INSERT OR UPDATE OF salary ON employees
  FOR EACH ROW
  WHEN (NEW.job_id = 'SA_REP')
BEGIN
  IF INSERTING
    THEN :NEW.commission_pct := 0;
  ELSIF :OLD.commission_pct IS NULL
    THEN :NEW.commission_pct := 0;
  ELSE
    :NEW.commission_pct := :OLD.commission_pct + 0.05;
  END IF;
END;
/
```

Trigger created.

Déclencheurs `INSTEAD OF`



Créer un déclencheur `INSTEAD OF`

Syntaxe :

```
CREATE [OR REPLACE] TRIGGER trigger_name
  INSTEAD OF
    event1 [OR event2 OR event3]
    ON view_name
    [REFERENCING OLD AS old | NEW AS new]
  [FOR EACH ROW]
  trigger_body
```

Créer un déclencheur **INSTEAD OF**

Exécution d'une instruction **INSERT** dans la vue **EMP_DETAILS** basée sur les tables **EMPLOYEES** et **DEPARTMENTS**

1

```
INSERT INTO emp_details(employee_id, ... )  
VALUES (9001, 'ABBOTT', 3000, 10, 'abbott.mail.com', 'HR_MAN');
```

Opération **INSERT** d'un
déclencheur **INSTEAD OF** dans
EMP_DETAILS

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	EMAIL	JOB_
100	King	90	SKING	AD_PRE
101	Kochhar	90	NKOCHHAR	AD_VP
102	De Haan	90	LDEHAAN	AD_VP
...				

Créer un déclencheur **INSTEAD OF**

Exécution d'une instruction **INSERT** dans la vue **EMP_DETAILS**
basée sur les tables **EMPLOYEES** et **DEPARTMENTS**

1 `INSERT INTO emp_details(employee_id, ...)
VALUES (9001, 'ABBOTT', 3000, 10, 'abbott.mail.com', 'HR_MAN');`

Opération **INSERT** d'un
déclencheur **INSTEAD OF** dans
EMP_DETAILS

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	EMAIL	JOB_ID
100	King	90	SKING	AD_PRE
101	Kochhar	90	NKOCHHAR	AD_VP
102	De Haan	90	LDEHAAN	AD_VP
...				

2 **INSERT** dans
NEW_EMPS

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID	EMAIL
100	King	24000	90	SKING
101	Kochhar	17000	90	NKOCHHAR
102	De Haan	17000	90	LDEHAAN
...				
9001	ABBOTT	3000	10	abbott.m

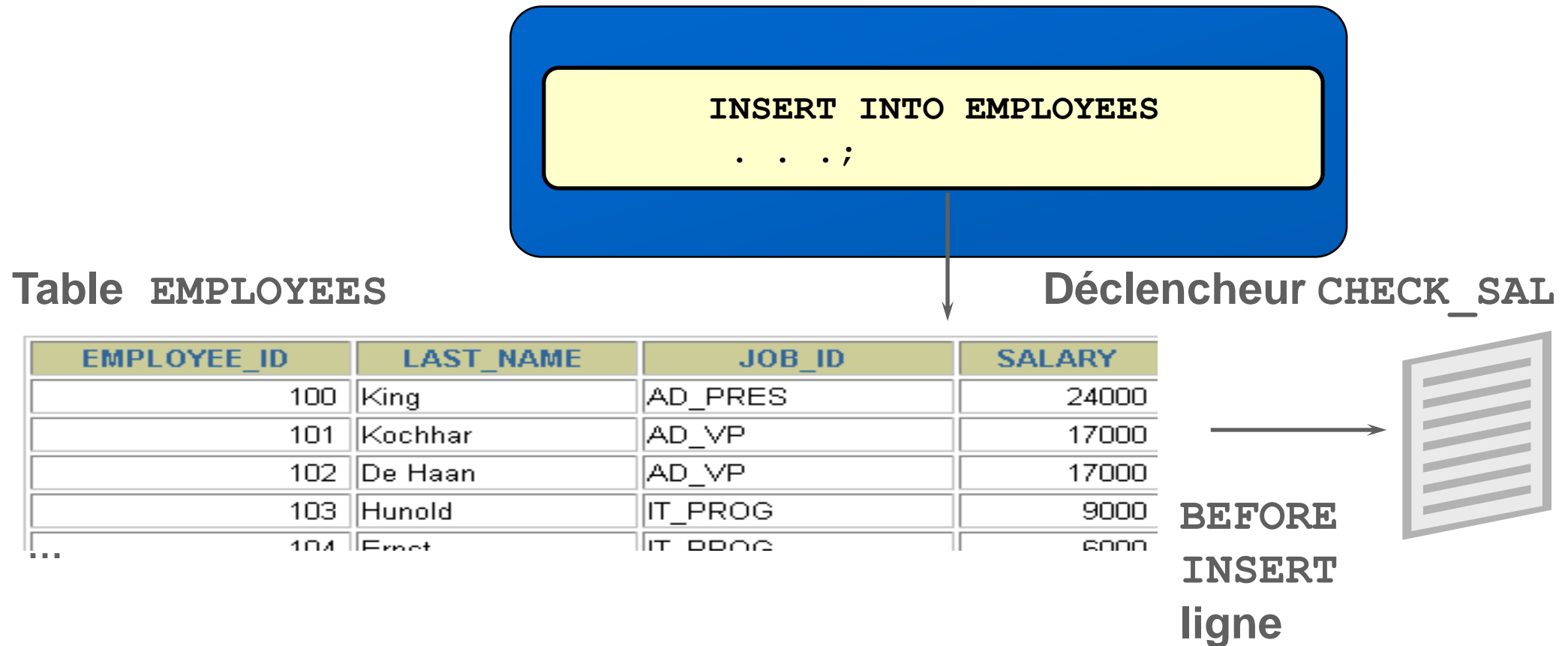
3 **UPDATE**
NEW_DEPTS

DEPARTMENT_ID	DEPARTMENT_NAME	TOT_DEPT_SA
10	Administration	9400
20	Marketing	19000
30	Purchasing	30125
40	Human Resources	6500
...		

Différences entre les déclencheurs de base de données et les procédures stockées

Déclencheurs	Procédures
<p>Définis via la commande <code>CREATE TRIGGER</code></p> <p>Le dictionnaire de données contient le code source dans <code>USER_TRIGGERS</code></p> <p>Appel implicite</p> <p>Les instructions <code>COMMIT</code>, <code>SAVEPOINT</code> et <code>ROLLBACK</code> ne sont pas autorisées</p>	<p>Définis via la commande <code>CREATE PROCEDURE</code></p> <p>Le dictionnaire de données contient le code source dans <code>USER_SOURCE</code></p> <p>Appel explicite</p> <p>Les instructions <code>COMMIT</code>, <code>SAVEPOINT</code> et <code>ROLLBACK</code> sont autorisées</p>

Différences entre les déclencheurs de base de données et les déclencheurs Form Builder



Gérer les déclencheurs

Désactiver ou réactiver un déclencheur de base de données :

```
ALTER TRIGGER trigger_name DISABLE | ENABLE
```

Désactiver ou réactiver tous les déclencheurs d'une table :

```
ALTER TABLE table_name DISABLE | ENABLE ALL TRIGGERS
```

Recompiler un déclencheur pour une table :

```
ALTER TRIGGER trigger_name COMPILE
```

Syntaxe DROP TRIGGER

Pour supprimer un déclencheur de la base de données, utiliser la syntaxe DROP TRIGGER :

```
DROP TRIGGER trigger_name;
```

Exemple:

```
DROP TRIGGER secure_emp;
```

Trigger dropped.

Remarque : Lorsqu'une table est supprimée, tous ses déclencheurs sont également supprimés

Tests des déclencheurs

- Tester toutes les opérations sur les données qui provoquent un déclenchement, ainsi que celles n'en produisent pas
- Tester chaque cas de la clause `WHEN`
- Provoquer une exécution directe du déclencheur via une opération de base sur les données, et une exécution indirecte via une procédure
- Tester l'impact du déclencheur sur les autres déclencheurs
- Tester l'impact des autres déclencheurs sur le déclencheur

Modèle d'exécution des déclencheurs et vérification des contraintes

1. Exécuter tous les déclencheurs `BEFORE STATEMENT`.
2. Effectuer une boucle pour toutes les lignes affectées :
 - a. exécuter tous les déclencheurs `BEFORE ROW`
 - b. exécuter tous les déclencheurs `AFTER ROW`
3. Exécuter l'instruction `LMD` et vérifier les contraintes d'intégrité.
4. Exécuter tous les déclencheurs `AFTER STATEMENT`.

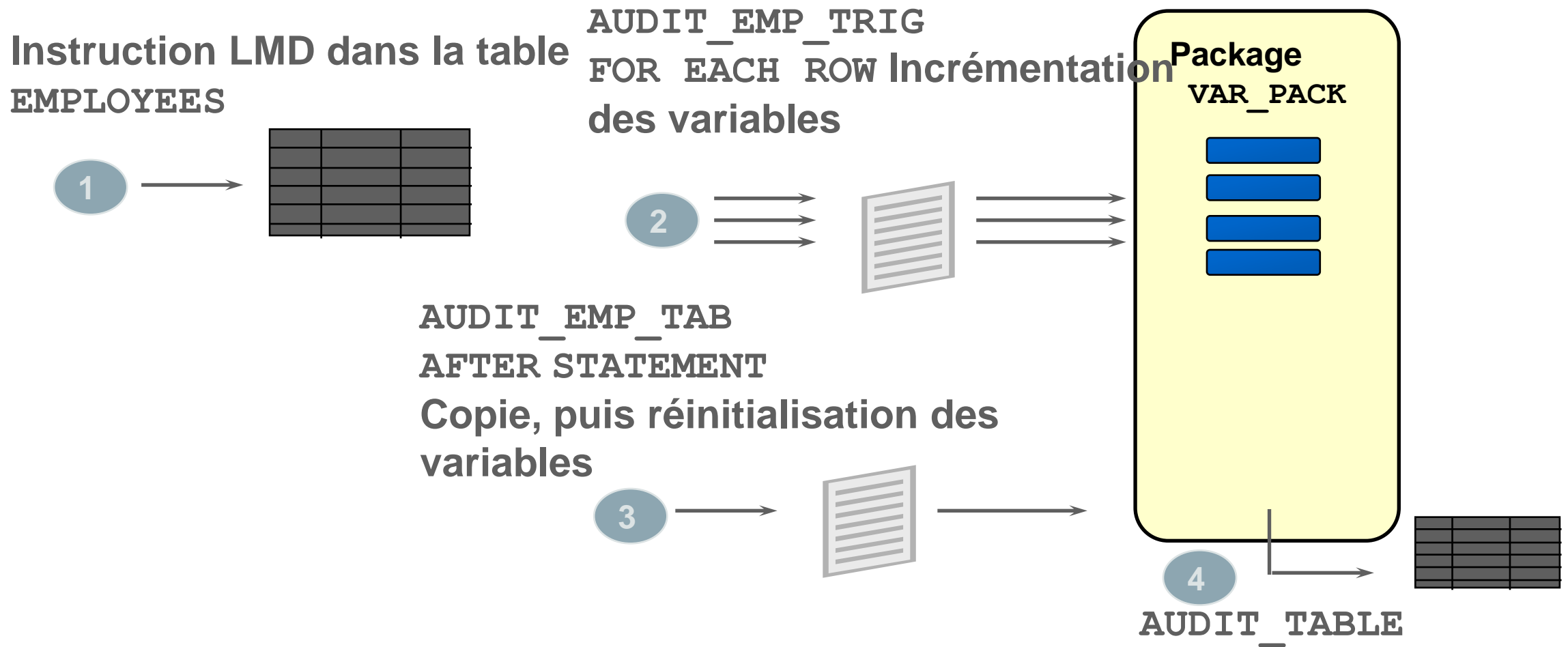
Exemple de modèle d'exécution des déclencheurs et de vérification des contraintes

```
UPDATE employees SET department_id = 999
WHERE employee_id = 170;
-- Integrity constraint violation error
```

```
CREATE OR REPLACE TRIGGER constr_emp_trig
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO departments
        VALUES (999, 'dept999', 140, 2400);
END;
/
```

```
UPDATE employees SET department_id = 999
WHERE employee_id = 170;
-- Successful after trigger is fired
```

Démonstration type de déclencheurs utilisant des structures de package



Déclencheurs AFTER sur ligne et sur instruction

```
CREATE OR REPLACE TRIGGER audit_emp_trig
AFTER      UPDATE or INSERT or DELETE on EMPLOYEES
FOR EACH ROW
BEGIN
    IF      DELETING      THEN  var_pack.set_g_del(1);
    ELSIF   INSERTING     THEN  var_pack.set_g_ins(1);
    ELSIF   UPDATING ('SALARY')
                                THEN  var_pack.set_g_up_sal(1);
    ELSE    var_pack.set_g_upd(1);
    END IF;
END audit_emp_trig;
/
```

```
CREATE OR REPLACE TRIGGER audit_emp_tab
AFTER      UPDATE or INSERT or DELETE on employees
BEGIN
    audit_emp;
END audit_emp_tab;
/
```

Démonstration de spécification du package VAR_PACK

var_pack.sql

```
CREATE OR REPLACE PACKAGE var_pack
IS
-- these functions are used to return the
-- values of package variables
    FUNCTION g_del RETURN NUMBER;
    FUNCTION g_ins RETURN NUMBER;
    FUNCTION g_upd RETURN NUMBER;
    FUNCTION g_up_sal RETURN NUMBER;
-- these procedures are used to modify the
-- values of the package variables
    PROCEDURE set_g_del      (p_val IN NUMBER);
    PROCEDURE set_g_ins      (p_val IN NUMBER);
    PROCEDURE set_g_upd      (p_val IN NUMBER);
    PROCEDURE set_g_up_sal   (p_val IN NUMBER);
END var_pack;
/
```

Démonstration d'utilisation de la procédure AUDIT_EMP

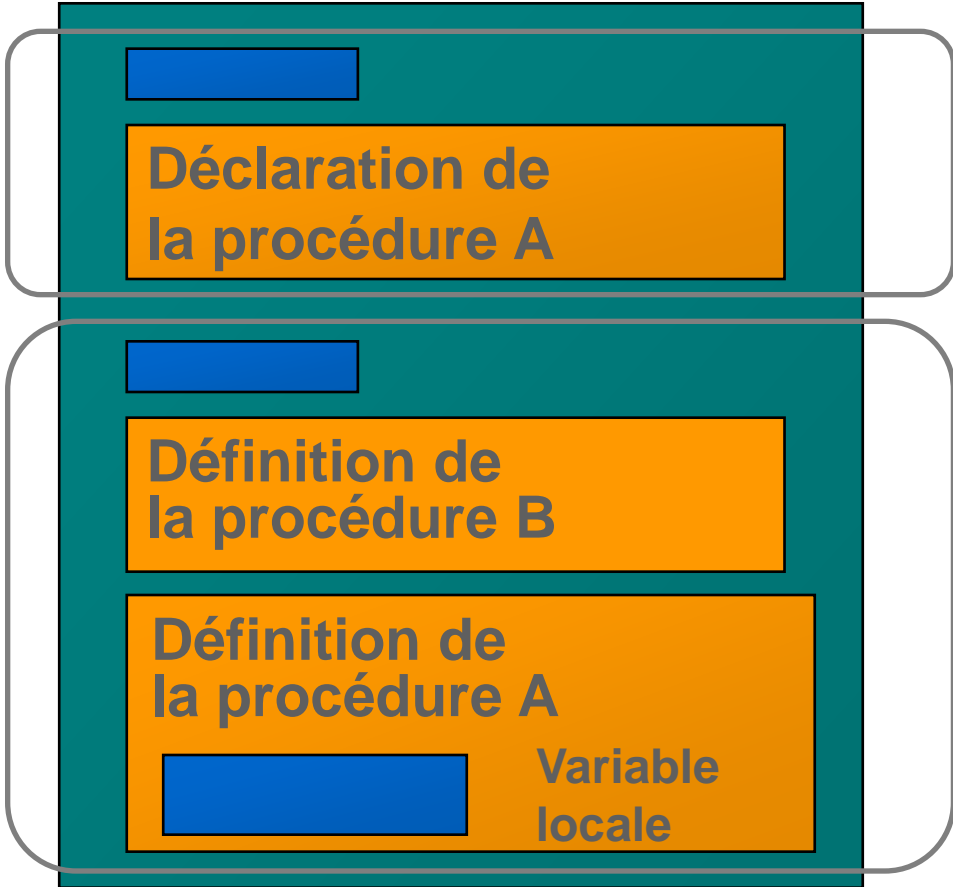
```
CREATE OR REPLACE PROCEDURE audit_emp IS
  v_del      NUMBER      := var_pack.g_del;
  v_ins      NUMBER      := var_pack.g_ins;
  v_upd      NUMBER      := var_pack.g_upd;
  v_up_sal   NUMBER      := var_pack.g_up_sal;
BEGIN
  IF v_del + v_ins + v_upd != 0 THEN
    UPDATE audit_table SET
      del = del + v_del, ins = ins + v_ins,
      upd = upd + v_upd
    WHERE user_name=USER AND tablename='EMPLOYEES'
    AND   column_name IS NULL;
  END IF;
  IF v_up_sal != 0 THEN
    UPDATE audit_table SET upd = upd + v_up_sal
    WHERE user_name=USER AND tablename='EMPLOYEES'
    AND   column_name = 'SALARY';
  END IF;
  -- resetting global variables in package VAR_PACK
  var_pack.set_g_del (0); var_pack.set_g_ins (0);
  var_pack.set_g_upd (0); var_pack.set_g_up_sal (0);
END audit_emp;
```

Synthèse

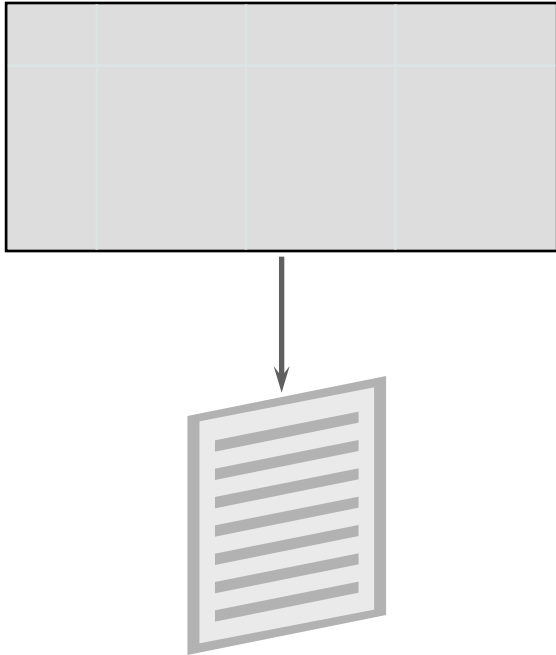
Procédure

```
XXXXXXXXXXXXXXXXXXXXX
VVVVVVVVVVVVVVVVVVVV
XXXXXXXXXXXXXXXXXXXXX
VVVVVVVVVVVVVVVVVVVV
XXXXXXXXXXXXXXXXXXXXX
VVVVVVVVVVVVVVVVVVVV
XXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXX
VVVVVVVVVVVVVVVVVVVV
XXXXXXXXXXXXXXXXXXXXX
VVVVVVVVVVVVVVVVVVVV
XXXXXXXXXXXXXXXXXXXXX
VVVVVVVVVVVVVVVVVVVV
XXXXXXXXXXXXXXXXXXXXX
```

Package



Déclencheur



Exercice

Dans cet exercice, vous allez :

- créer des déclencheurs sur instruction et sur ligne
- créer des déclencheurs avancés afin d'accroître les fonctionnalités de la base de données Oracle



Créer des packages

Objectifs

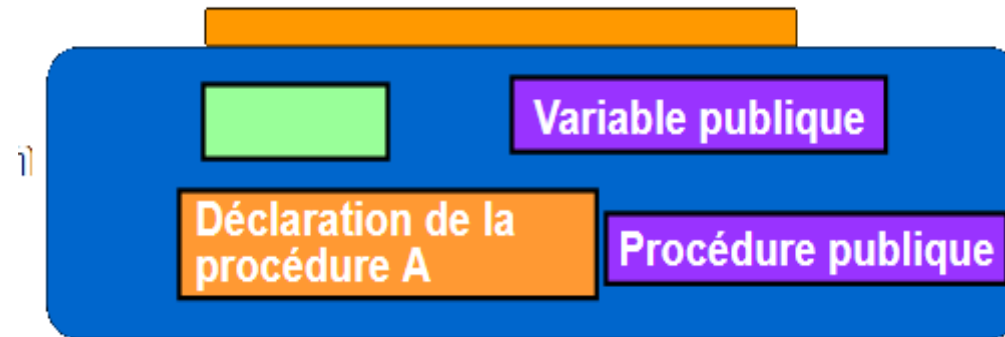
- **A la fin de ce chapitre, vous pourrez :**
 - décrire des packages et répertorier leurs éventuels composants
 - créer un package regroupant des variables, constantes, exceptions, procédures, fonctions et curseurs associés
 - désigner une structure de package comme publique ou privée
 - appeler une structure de package
 - décrire l'utilisation d'un package sans corps

Présentation des packages

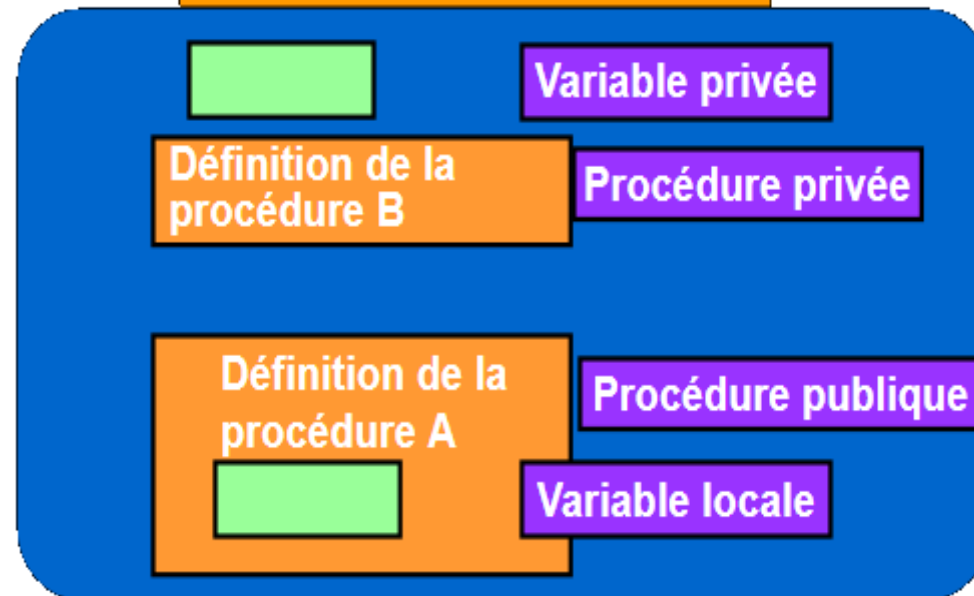
- Les packages :
 - regroupent des types PL/SQL, des éléments et des sous-programmes présentant une relation logique
 - sont constitués de deux éléments :
 - spécification
 - corps
 - ne peuvent pas être appelés, paramétrés ou imbriqués
 - permettent au serveur Oracle de lire simultanément plusieurs objets en mémoire

Composants d'un package

Spécification du package

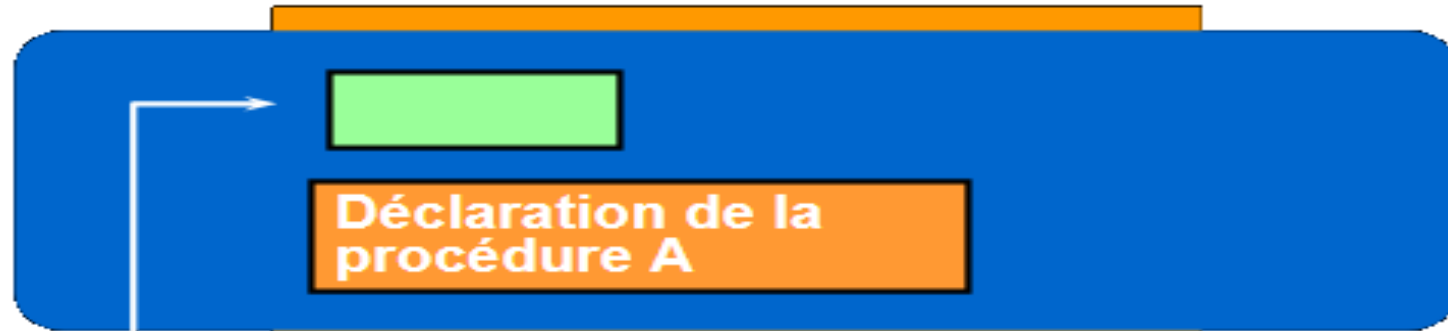


Corps du package

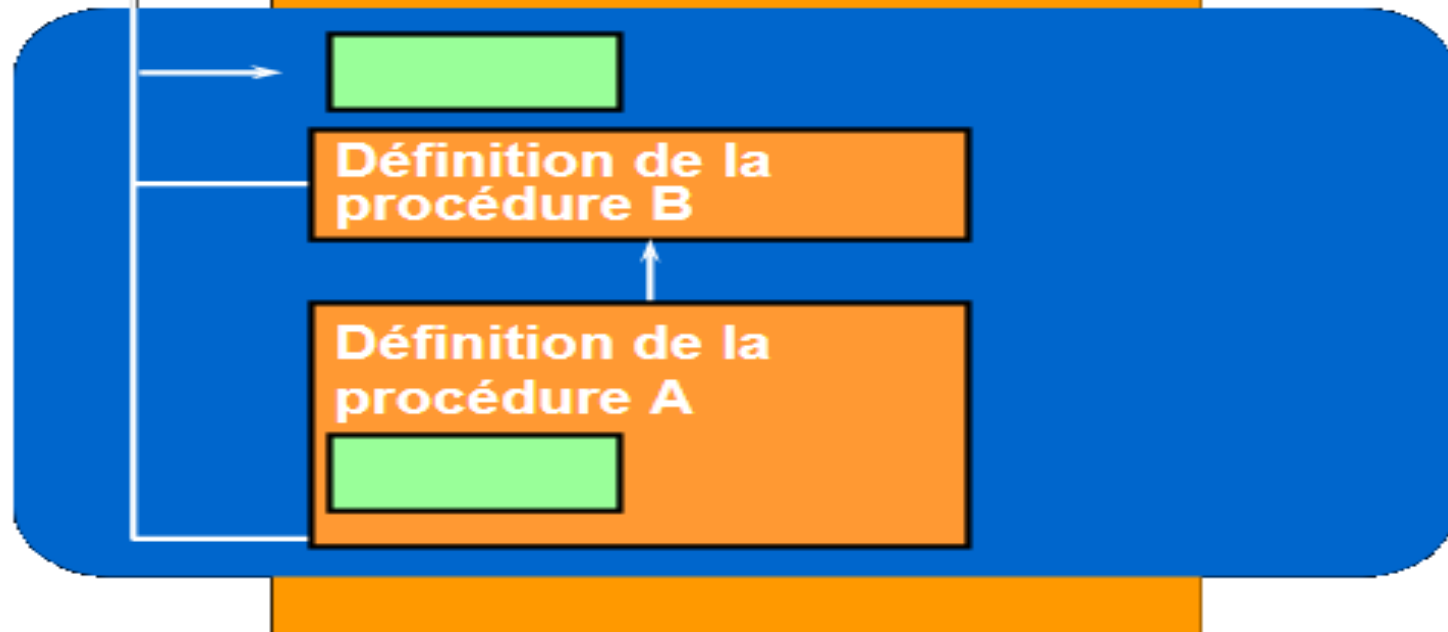


Référencer des objets de package

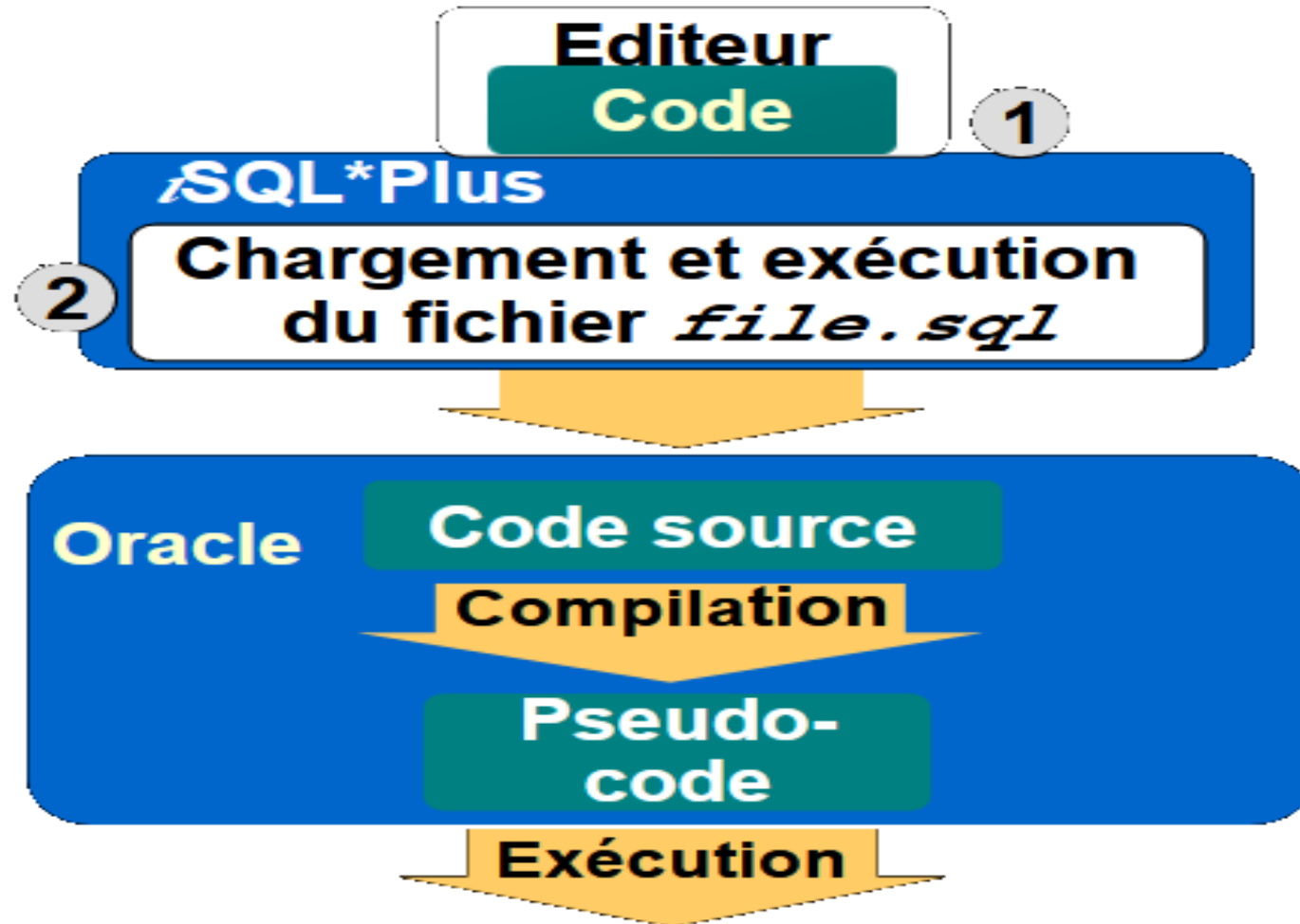
Spécification du package



Corps du package



Chargement et exécution du fichier file.sql



Développer un package

- L'enregistrement du texte de l'instruction CREATE PACKAGE dans deux fichiers SQL distincts facilite les modifications ultérieures du package
- Une spécification de package peut exister sans corps de package, mais l'inverse n'est pas vrai

Créer la spécification du package

- Syntaxe:

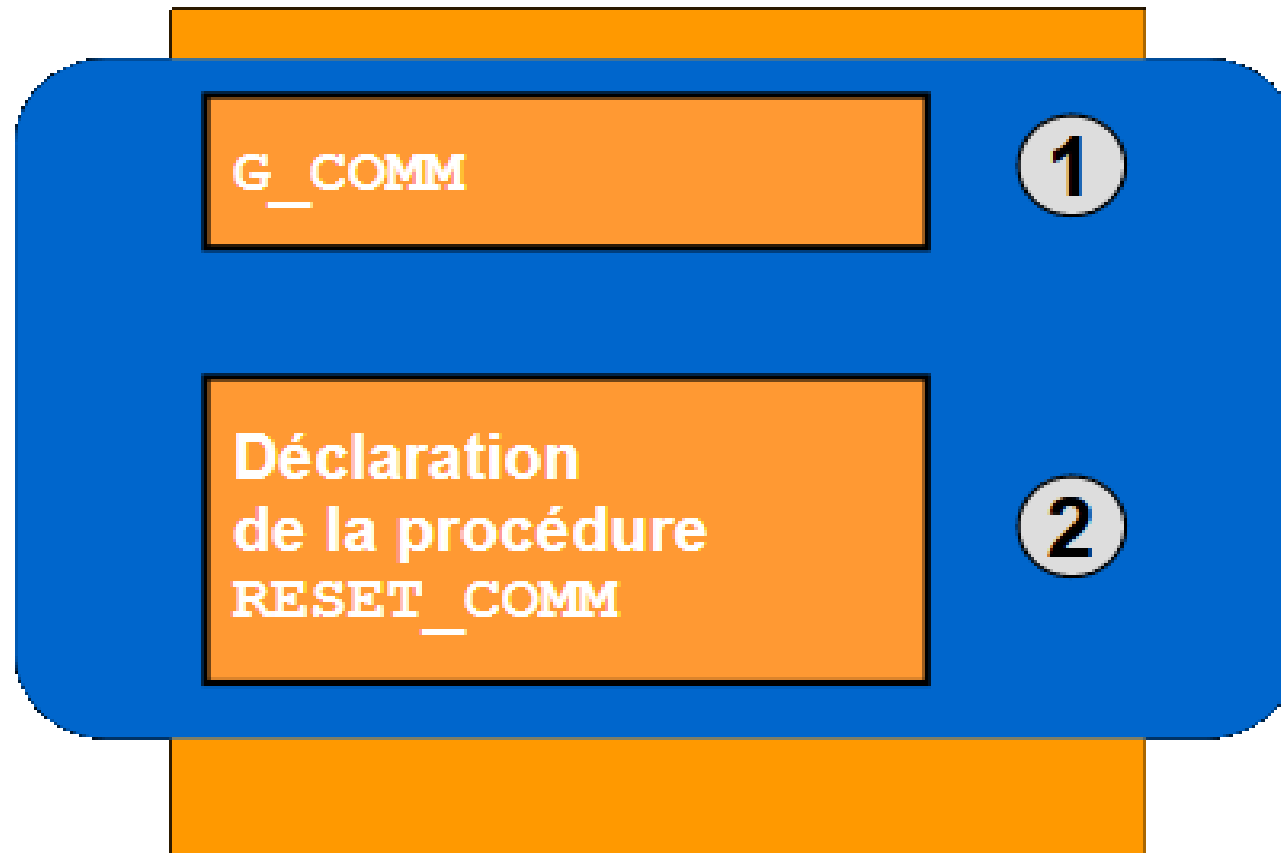
```
CREATE [OR REPLACE] PACKAGE package_name  
IS/AS  
    public type and item declarations  
    subprogram specifications  
END package_name;
```

- L'option REPLACE supprime et recrée la spécification du package
- Par défaut, la valeur NULL est affectée aux variables déclarées dans la spécification du package
- Toutes les structures déclarées dans une spécification de package peuvent être visibles par les utilisateurs disposant de privilèges sur le package

9 Déclarer des structures publiques

Package COMM_PACKAGE G_COMM

Spécification du package



Exemple de création de spécification de package

```
CREATE OR REPLACE PACKAGE comm_package IS  
    g_comm NUMBER := 0.10; --initialized to 0.10  
    PROCEDURE reset_comm (p_comm IN NUMBER);  
END comm_package;  
/
```

- G_COMM est une variable globale dont la valeur d'initialisation est 0,10.
- RESET_COMM est une procédure publique implémentée dans le corps du package.

Créer le corps du package

Syntaxe:

```
CREATE [OR REPLACE] PACKAGE BODY package_name  
IS/AS  
    private type and item declarations  
    subprogram bodies  
END package_name;
```

- L'option REPLACE supprime et recrée le corps du package
- Les identificateurs définis exclusivement dans le corps du package sont des structures privées. Ils ne sont pas visibles à l'extérieur du corps du package
Toutes les structures privées doivent être déclarées avant d'être utilisées dans les structures publiques

ORACLE®