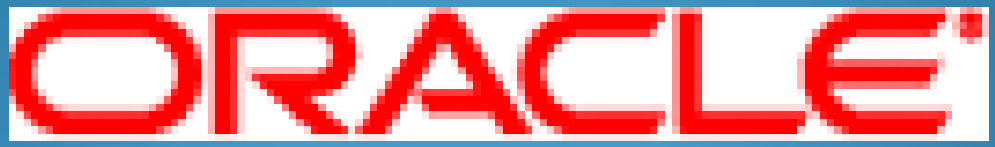
Université Hassan II - Casablanca
Faculté des Sciences et Techniques
Mohammedia

Procedural Language extensions to SQL

PL/SQL



Par : Prof. EL BEGGAR Omar

SOMMAIRE

- *Rappel SQL*
- *Introduction à PL/SQL*
- *Structure d'un bloc PL/SQL*
- *Variables*
- *Structures de contrôle*
- *Les curseurs*
- *Exceptions*
- *Les fonctions stockées*
- *Les procédures stockées*
- *Les triggers*
- *Packages*

Rappel SQL

- DOC SGBD I- SQL

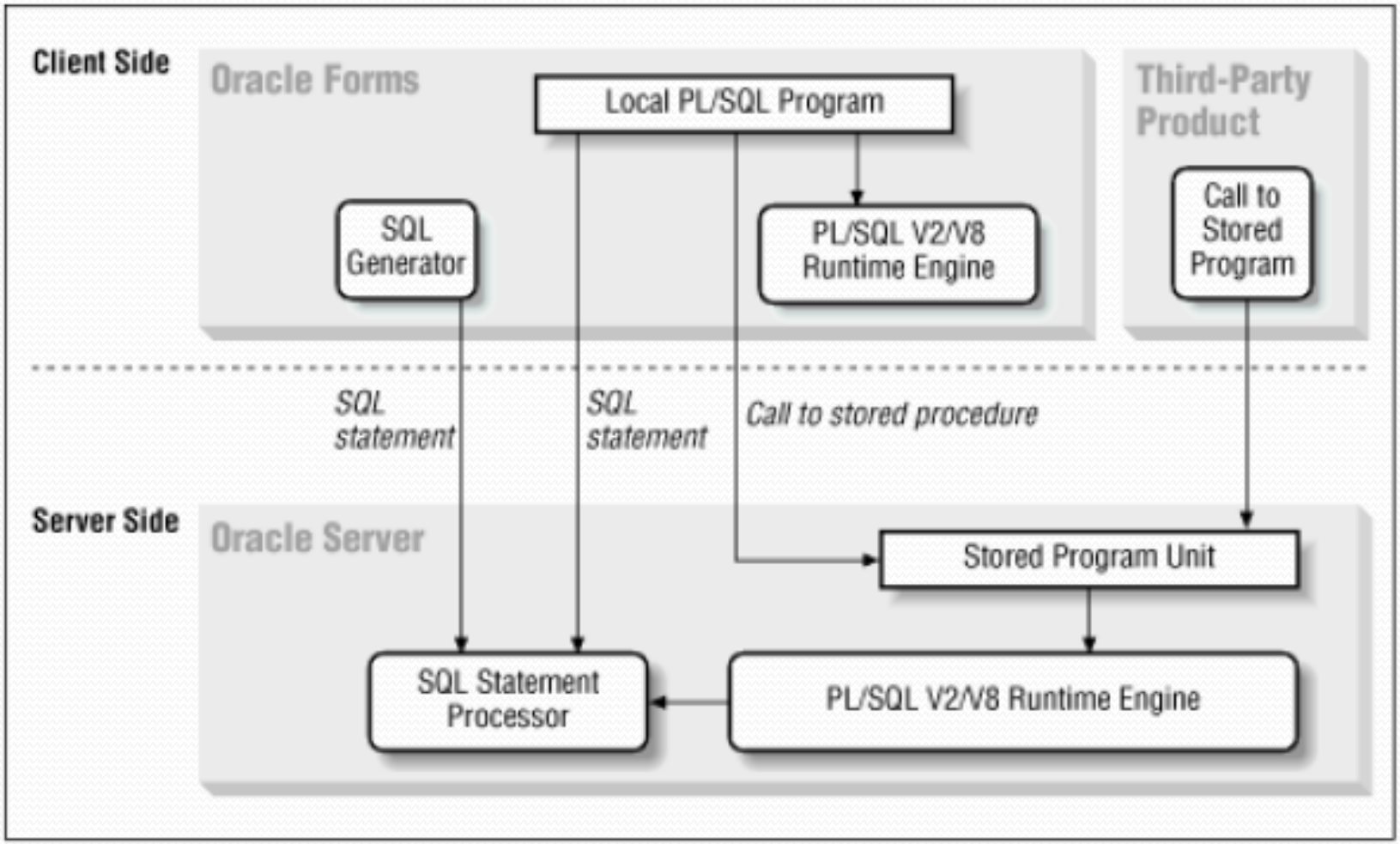
Introduction à PL/SQL

- SQL est un langage ensembliste, ie qu'il ne manipule qu'un ensemble de données satisfaisants des critères de recherches.
- PL/SQL (Prcedural Language extension to Structured Query Language) est un langage procédural, il permet de traiter de manière conditionnelle les données retournées par un ordre SQL.
- C'est une extension du SQL car il permet de cohabiter des structures de contrôle (IF…THEN, LOOP…) avec des instructions SQL(SELECT, UPDATE…)

Introduction à PL/SQL

- PL/SQL étend SQL en lui ajoutant des éléments tel que :
  - Les variables et les types,
  - Les structures de contrôles et les boucles.
  - Les procédures et fonctions.
  - Les types d'objets et les méthodes.
- Un bloc d'ordre SQL et PL/SQL est transmis au moteur de la BD oracle
- → traitements interne à la BD, réduction du temps des aller/retour entre le serveur et l'application

Architecture PL/SQL



Le bloc PL/SQL

- PL/SQL n'interprète pas une commande, mais un ensemble de commandes contenues dans le bloc PL/SQL.
- Ce bloc est compilé et exécuté par le moteur PL/SQL du produit ou de la base.
- Structure d'un bloc PL/SQL

```
DECLARE
(déclaration des variables, des constantes, des exceptions et des curseurs)

BEGIN
(instructions SQL, PL/SQL structures de contrôle)

EXCEPTION
(traitement des erreurs)

END [nom du bloc]
```

- Un bloc est composé de trois sections.
- Il est possible d'ajouter des commentaires à un bloc :
  - -- permet de mettre en commentaire ce qui suit sur la ligne
  - /*… …*/ permet de mettre en commentaire plusieurs lignes.
- Des étiquettes (label) permettent de marquer des parties de bloc, sous la forme <<nom_etiquette>>.
- La section DECLARE qui permet de déclarer les variables qui vont être utilisées dans le bloc PL/SQL, n'est nécessaire que si le bloc a besoin de définir des variables. De même, la section EXCEPTION ne sera présente que dans les blocs qui vont gérer les erreurs.

- Dans SQL*Plus, une simple instruction se termine par un point-virgule (;), une fois rencontré l'instruction en question est envoyé au serveur.
- La fin d'un bloc PL/SQL est déterminé par (/), raccourci de RUN.

- **Exemple:**

```
SQL>BEGIN
1 INSERT INTO dept values(50, 'MAINTENANCE','CASA');
2 COMMIT;
3 END;
4 /
```

Types de Blocs

- Chaque unité de PL/SQL comprend un ou plusieurs blocs qui peuvent être distincts ou imbriqués les uns dans les autres.
- Les unités de base (procédures, fonctions : sous-programmes, et blocs anonymes) qui composent un programme PL/SQL sont des blocs logiques qui peuvent contenir un nombre quelconque de sous-blocs imbriqués.
- Ainsi, un bloc peut constituer une petite partie d'un autre bloc lui-même inclus dans l'unité de code. Les deux types de construction PL/SQL, les blocs anonymes et les sous-programmes.

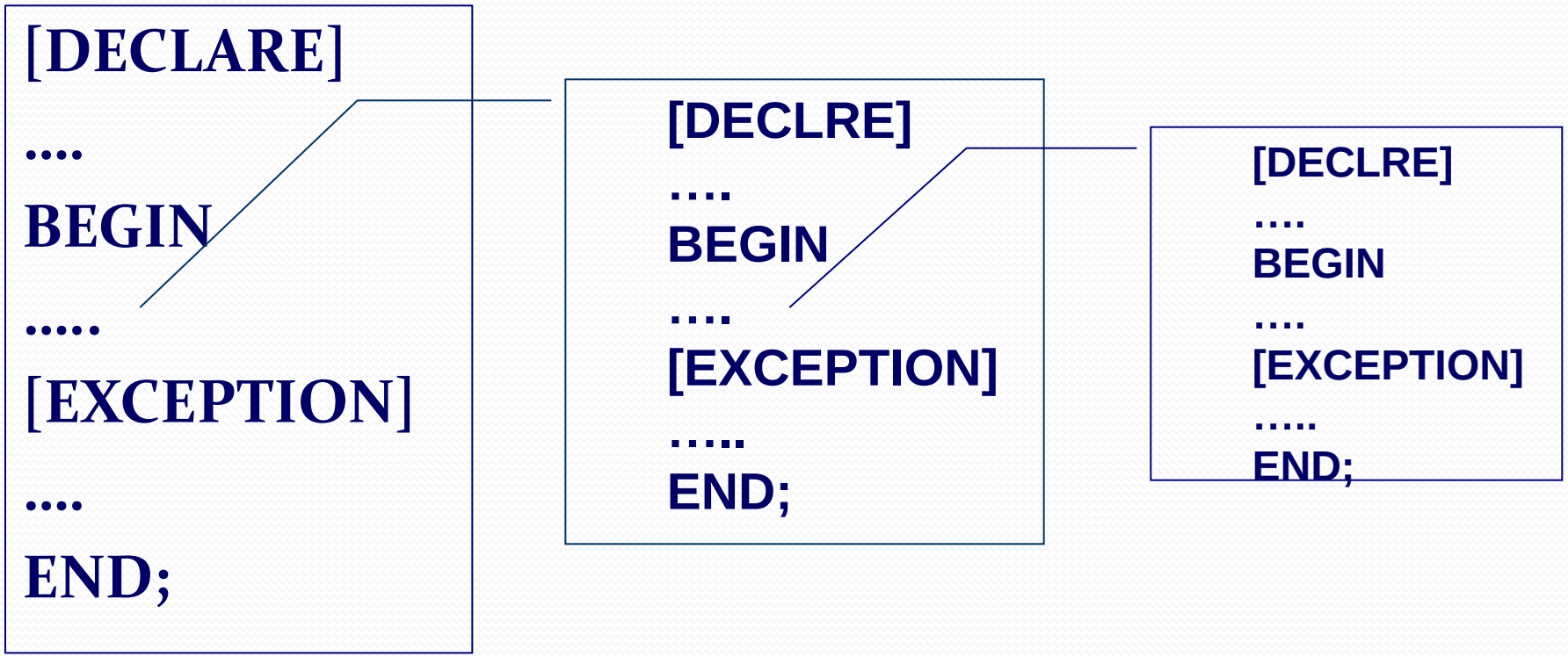
Blocs Anonymes

- Les blocs anonymes sont des blocs sans nom. Ils sont déclarés dans une application à l'endroit où ils doivent être exécutés et sont envoyés vers le moteur PL/SQL pour être exécutés en temps voulu.
- Vous pouvez inclure un bloc anonyme dans un programme précompiler, sous SQL*Plus ou Server Entreprise Manager.
- Les triggers d'Oracle Developer sont constitués de tels blocs.

Sous-Programmes

- Les sous-programmes sont des blocs PL/SQL nommés qui peuvent être paramétrés et appelés. Vous pouvez les déclarer comme procédures ou comme fonctions. Généralement, vous utiliserez une procédure pour réaliser une action et une fonction pour traiter une valeur.
- Vous pouvez stocker des sous-programmes au niveau du serveur Oracle ou de l'application.
- En utilisant les composants d'Oracle Developer (Forms, Reports et Graphics), vous pouvez déclarer les procédures et les fonctions comme des parties de l'application (du formulaire ou du rapport) et les appeler à partir d'autres procédures et fonctions ou à partir de triggers dans la même application à chaque fois que cela s'avère nécessaire

Bloc imbriqués



Mot clé PRAGMA

- PRAGMA (aussi appelé *pseudo-instruction*) est un mot-clé qui signifie que l'instruction est destinée au compilateur et qu'elle n'est donc pas traitée au moment de l'exécution du bloc PL/SQL.

- EXCEPTION_INIT : intercepte une exception d'oracle et affiche un autre message personnalisé par le programmeur.

- AUTONOMOUS_TRANSACTION : indique au compilateur d'exécuter le bloc de code PLSQL (anonyme ou sous programme)séparément dans une transaction autonome;
  (C'est une sorte d'isolation de la transaction qui se trouve dans le bloc PLSQL du reste des transactions écrites)

Exemple : PRAGMA Autonomous_transaction

```
SQL> DELETE FROM dep;
SQL>DECLARE
2 PRAGMA autonomous_transaction;
3 BEGIN
4 INSERT INTO dept values(60, 'STOCK','CASA');
5 COMMIT;
6 END;
7 /
SQL>ROLLBACK;
SQL>SELECT COUNT(*) FROM dep;
```

Exemple : PRAGMA Exception_INIT

Si on insère plus de caractères dans une variable que sa déclaration ne le permet, alors Oracle déclenche une erreur

-6502. Nous allons "nommer" cette erreur en exc_long et l'intercepter dans la section exception

```
SQL>Declare
1 chaine varchar2(3) ;
2 exc_long exception ;
3 pragma exception_init( exc_long, -6502 ) ;
4 Begin
5 chaine := 'Bonjour' ;
6 Exception
7  When exc_long then
8  dbms_output.put_line( 'Chaîne de caractères trop longue') ;
9 End ;
11 /
```

Identificateurs

- Le nom de tout objet PL/SQL (variables, curseurs, exceptions,…) doit commencer par une lettre suivie de symboles (lettres, chiffres,$, _, #).
- Un identificateur peut contenir un max de 30 caractères.
- Exemples d'identificateurs autorisés:
  - X
  - t2
  - télé#phone
  - Home_phone
  - My$num
- Exemples d'identificateurs non autorisés:
  - 1VAR: le nom de variable ne doit pas commencer par un chiffre
  - Credit-card : confusion avec – qui signifie soustraction;
  - On/off : confusion avec / qui signifie division;
  - First&name : confusion avec & qui signifie la valeur name.

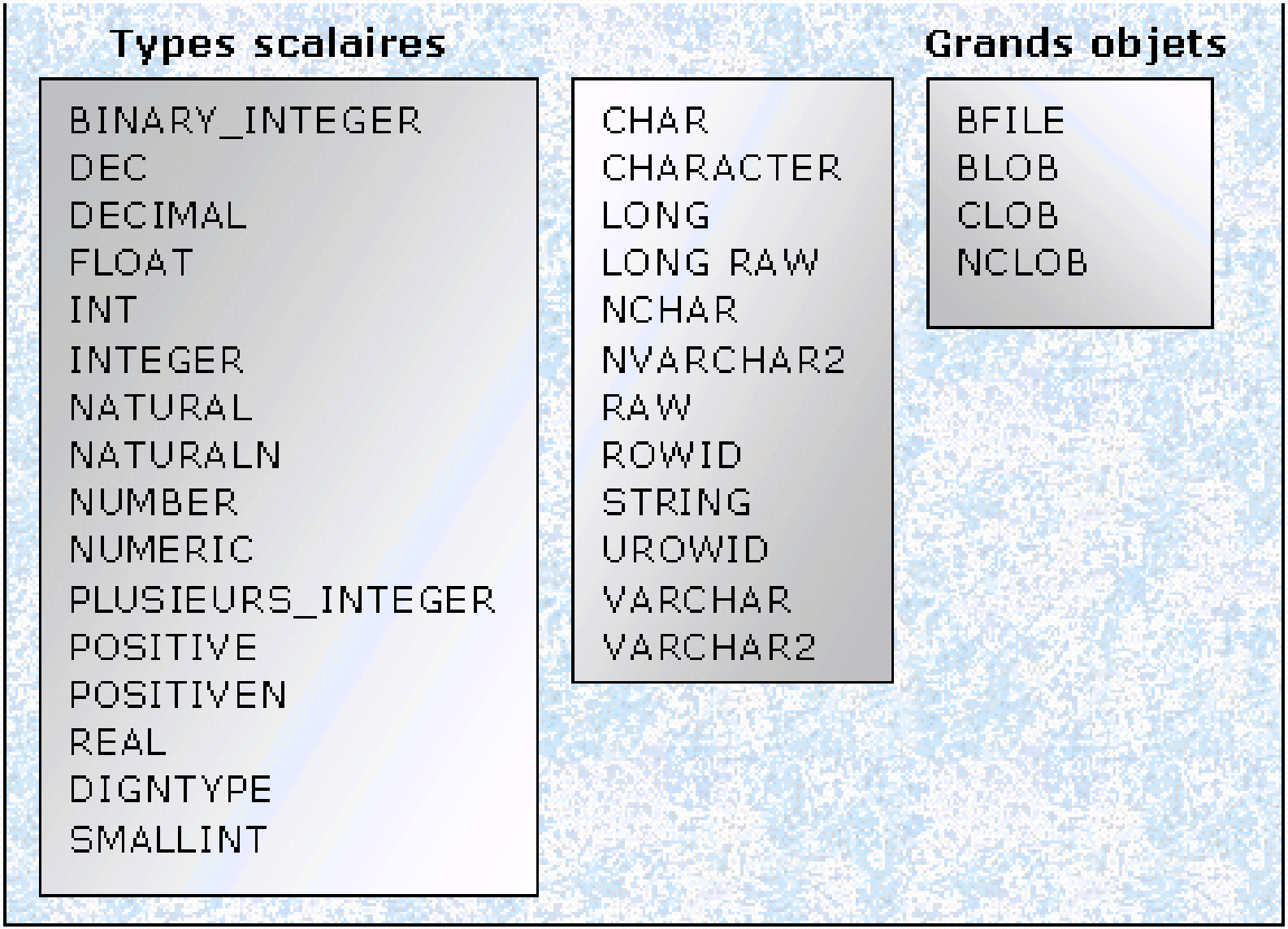
Variables

- Les variables sont des zones mémoires nommées permettant de stocker une valeur.
- En PL/SQL, elles permettent de stocker des valeurs issues de la base ou de calculs, afin de pouvoir effectuer des tests, des calculs ou des valorisations d'autres variables ou de données de la base.
- Les variables sont caractérisées par :
  - leur nom, composé de lettres, chiffres, $, _, ou #.
  - un maximum de 30 caractères est possible.
  - Ne doit pas être un nom réservé à Oracle.
  - Insensible à la casse
  - leur type, qui détermine le format de stockage et d'utilisation de la variable.

- Les variables doivent être obligatoirement déclarées avant leur utilisation.
- Comme SQL, PL/SQL n'est pas sensible à la casse. Les noms de variables peuvent donc être saisis indifféremment en minuscules ou en majuscules.
- Exemple :
  - DECLARE
  - Var1 number(2);
  - Var2 number(4):=Var1+Var3;//erreur, Var3 non définie
  - Var3 number(2);

Types de Variables

- variables PL/SQL
  - Les types scalaires reçoivent une seule valeur. Les principaux types utilisés sont ceux des colonnes du Serveur Oracle, le PL/SQL supporte aussi les variables booléennes.
  - Les types composés tels que les records, permettent de définir des groupes de champs et de les manipuler dans des blocs PL/SQL.
  - Les types références contiennent n valeurs, appelées aussi *pointeurs,* ils désignent d'autres éléments du programme.
  - Les types LOB contiennent des valeurs, appelés aussi *locators,* ils spécifient l'emplacement des Large Objets (Images).
- Les variables non-PL/SQL sont des variables du langage hôte, déclarées dans des programmes précompileurs, des champs d'écran dans les applications Forms ou des variables hôtes SQL*Plus.


Types scalaires
BINARY_INTEGER
DEC
DECIMAL
FLOAT
INT
INTEGER
NATURAL
NATURALN
NUMBER
NUMERIC
PLUSIEURS_INTEGER
POSITIVE
POSITIVEN
REAL
DIGNTYPE
SMALLINT
CHAR
CHARACTER
LONG
LONG RAW
NCHAR
NVARCHAR2
RAW
ROWID
STRING
UROWID
VARCHAR
VARCHAR2
Grands objets
BFILE
BLOB
CLOB
NCLOB

Variables scalaires

- ***Nom_de_var [CONSTANT] typeDeDonnees [NOT NULL] [:= | DEFAULT expression];***
- CONSTANT : La valeur de la variable n'est pas modifiable dans le code de la section BEGIN.
- NOT NULL : Empêche l'affectation d'une valeur NULL à la variable, expression doit être fournies.
- expression : valeur initiale affectée à la variable lors de l'exécution du bloc.

```
SQL> connect pluser/123;
Connecté.
SQL> DECLARE
  2  mavar CONSTANT number :=300;
  3  BEGIN
  4  mavar:=400;
  5  END;
  6  /
mavar:=400;
*
ERREUR à la ligne 4 :
PLS-00363: expression 'MAVAR' ne peut être utilisée comme
   cible d'affectation
SQL>
```

```
SQL> DECLARE
  2  mavar  varchar(22) NOT NULL;
  3  BEGIN
  4  mavar:=400;
  5  END;
  6  /
mavar  varchar(22) NOT NULL;
       *
ERREUR à la ligne 2 :
ORA-06550: Ligne 2, colonne 8 :
PLS-00218: une variable déclarée NOT NULL doit avoir une
Affectation d'initialisation

SQL>
```

```
DECLARE
    v_date_naiss DATE;
    --⇔ v_date_naiss DATE :=NULL;
    v_total NUMBER(4) :=300;
    --⇔ v_total NUMBER(4) DEFAULT 300;
    v_tel CHAR(14) NOT NULL :='06-61-11-11-11';
    v_trouve BOOLEAN NOT NULL := TRUE;
    c_p CONSTANT NUMBER:=3.14159;
    v_curent_date DATE := SYSDATE;
BEGIN
    .....
END;
/
```

Affectation

- Trois possibilités:
  - variable:=expression;
  - la directive DEFAULT
  - la directive INTO d'une rêquete SQL : SELECT … INTO var FROM…..)
- Exemples:
  - v_today:=SYSDATE;
  - v_somme NUMBER DEFAULT 0;
  - SELECT count(*) INTO v_count FROM emp;

DATE ET HEURE ORACLE

```
declare
Heure date;
begin
heure:=sysdate;
dbms_output.put_line
(to_char(HEURE,'HH24:MI:SS'));
end;
/
```

11:41:15

```
declare
Auj date;
begin
auj:=sysdate;
dbms_output.put_line
(to_char(Auj,'DD/MM/YYYYHH24:MI:SS'));
end;
/
```

09/11/2009 11:41:15

Variables %TYPE

- Lors de la déclaration d'une variable PL/SQL devant contenir des valeurs de colonnes, vous devez vous assurez que la variable est de type et de précision corrects. Si ce n'est pas le cas, une erreur PL/SQL se produira lors de l'exécution.
- Plutôt que de coder en dur le type et la précision de la variable, vous pouvez utiliser l'attribut %TYPE pour déclarer une variable en relation avec une variable préalablement déclarée ou une colonne de la base.
- L'attribut %TYPE est utilisé la plupart du temps lorsque la valeur stockée dans la variable doit dériver d'une table de la base ou si la variable est destinée à être écrite dans la base.
- Pour utiliser l'attribut %TYPE à la place du type requis dans la déclaration de la variable, préfixez-le par le nom de la colonne et de la table. Si vous faites référence à une variable préalablement déclarée, préfixez le par le nom de la variable concernée.

- Une contrainte de colonne NOT NULL ne s'applique pas aux variables qui utilisent %TYPE. C'est pourquoi si vous déclarez une variable à l'aide de l'attribut %TYPE se référant à une colonne de la base définie comme NOT NULL, vous pouvez affecter la valeur NULL à cette variable.
- Exemples :
- v_ename emp.ename%TYPE;
- v_balance NUMBER(7,2);
- v_min_balance v_balance%TYPE := 10;

Variables %ROWTYPE

- Une variable peut contenir toutes les colonnes d'une ligne d'une table.
- On peut donc déclarer un enregistrement (record) qui est de même type que les enregistrements d'une table.
    - *Syntaxe: Nom_record nomtable%Rowtype;*
- Exemple : **emp_rec employees%ROWTYPE;**
- la variable emp_rec contiendra une ligne de la table employees.

Exemple d'utilisation

```
Declare
Emp-rec employees%ROWTYPE;
nom employees.last_name%TYPE;
Begin
select * INTO emp_rec
from employees
where employee_id= 900;
nom := emp_rec.nome;
emp_rec.dept := 20;
…
insert into employees values employe;
End;
/
```

Variables RECORD

- La directive %ROWTYPE permet de déclarer une structure composée de colonnes de table.
- Mais elle ne convient pas pour définir des structures personnalisées
- C'est l'équivalent des structures (***struct en langage C***)
- Syntaxe:

TYPE nomRecord IS RECORD
(nomChampe typeDonnees [[NOT NULL]{:= | DEFAULT} expression] [, nomChamps type de donnees ...]...);

Exemple de déclaration Record

```
TYPE emp2 IS RECORD (
    matr integer,
    nom varchar(30));
employe emp2;
employe.matr := 500;
```

Variables de substitution

- Il est possibles d'utiliser des variables définies sous SQL*Plus dite de substitution.
- On peut accéder à la valeur d'une variable en faisant préfixer le nom de la variable par le et commerciale « & ».

```
SQL> ACCEPT emp_no PROMPT 'entrer le numéro de l''employe : '
SQL> DECLARE
SQL> v_nom emp.ename%TYPE;
SQL> BEGIN
SQL> SELECT ename INTO v_nom from emp where
   empno=&emp_no;
SQL> DBMS_OUTPUT.PUT_LINE('le nom de l employe : ' || v_nom);
SQL> END;
SQL> /
```

Variables de session

- Il est possible de définir des variables de session (globales) sous SQL*Plus au niveau d'un bloc PL/SQL.
- La directive SQL*Plus permettant de définir une variable de session est : VARIABLE
- Pour utiliser une variables de session dans un bloc PL/SQL il faut préfixer son nom par deux points (:).
- L'affichage de cette variable de session sous SQL*Plus se fait à l'aide de la directive PRINT.

Exemple de variable session:

```
SQL> set serveroutput on;
SQL> VARIABLE g_compteur NUMBER
SQL> DECLARE
  2  V_compteur NUMBER(3) :=99;
  3  BEGIN
  4  :g_compteur:=1;
  5  v_compteur:= :g_compteur+200;
  6  DBMS_output.put_line(v_compteur);
  7  END;
  8  /
201
SQL> PRINT g_compteur;
G_COMPTEUR
----------
         1
```

Structures de contrôle

Structures de contrôle

- Les structures conditionnelles avec les instructions
  - IF..THEN…END IF
  - IF…THEN…ELSE…END IF
  - IF…THEN…ELSIF… END IF
- et les structures répétitives (LOOP).
  - WHILE
  - FOR

structures conditionnelles

```
IF condition THEN
  instructions;
[ELSIF condition THEN
  instructions;]
[ELSE
  instructions;]
END IF;
```

- ***condition*** C'est une variable ou une expression booléenne (TRUE, FALSE, ou NULL) (Elle est associée à une séquence d'instructions, exécutée seulement si l'expression vaut TRUE.)
- ELSIF: Si la première condition vaut FALSE ou NULL, alors le mot-clé ELSIF introduit d'autres conditions.
- ELSE: Le mot-clé est atteint lorsque toutes les expressions précédentes sont évaluées à FALSE, la séquence d'instructions qui suit est alors réalisée.

Instruction IF simple :

- Si le nom de l'employé est Michel, lui associer le numéro de manager 22
- **IF v_ename = Michel' THEN**
- **v_mgr := 22;**
- **ENDIF;**
- Si le nom de l'employé est Miller, lui associer le métier de vendeur, le numéro de département 35, et une commission de 20% de son salaire actuel.
- Exemple
- **IF v_ename = 'MILLER' THEN**
- **v_job := 'SALESMAN';**
- **v_deptno := 35;**
- **v_new_comm := sal * 0.20;**
- **END IF;**

Exemple

- Pour une valeur donnée, on calcule un certain pourcentage de cette valeur initiale. Si cette valeur est supérieure à 100, alors la valeur finale vaut deux fois la valeur initiale. Si cette valeur est comprise entre 50 et 100, alors la valeur finale vaut 50% de la valeur initiale. Si cette valeur est inférieure à 50, alors la valeur finale vaut 10% de la valeur initiale.
- **IF v_start > 100 THEN**
- **v_start := 2 * v_start;**
- **ELSIF v_start >= 50 THEN**
- **v_start := .5 * v_start;**
- **ELSE**
- **v_start := .1 * v_start;**
- **END IF;**

Conditions Booléennes

- Vous pouvez écrire une condition booléenne complexe en combinant des conditions booléennes simples à l'aide des opérateurs logiques AND, OR, et NOT.
- AND renvoie TRUE seulement si ses deux conditions valent TRUE.(Multiplication False=0 and True=1)
- OR renvoie FALSE seulement si ses deux conditions valent FALSE.(Addition False=0, True=1)
- La négation de NULL (NOT NULL) retourne une valeur nulle car les valeurs nulles sont indéterminées.

Quelle est la valeur de V_FLAG dans chaque cas ?

- v_flag := v_reorder_flag AND v_available_flag;

V_REORDER_FLAG	V_AVAILABLE_FLAG	V_FLAG
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
NULL	TRUE	NULL
NULL	FALSE	FALSE

- v_flag := v_reorder_flag OR v_available_flag;

V_Recorder_flag	V_available_flag	V_flag
TRUE	TRUE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE
NULL	FALSE	NULL
NULL	TRUE	TRUE

Directive case

Case sans variable(when avec exp<>…)

```
<<etiquette>>
CASE
WHEN var <= 10THEN
        Instructions1;
WHEN between 10 and 50 THEN
        Instructions2;
....
WHEN expressionN-1 THEN
        InstructionsN-1;
[ELSE
        InstructionsN;]
END CASE [etiquette];
```

Case avec variable (When avec valeurs)

```
<<etiquette>>
CASE   VAR
WHEN valeur1 THEN
        Instructions1;
WHEN valeur2 THEN
        Instructions2;
....
WHEN valeurN-1 THEN
        InstructionsN-1;
[ELSE
        InstructionsN;]
END CASE [etiquette];
```

Structures répétitives

- **Boucle Basique**
- **Boucle FOR**
- **Boucle WHILE**

Boucle Basique

- Dite encore boucle infinie, contenant un jeu d'instructions entre les mots clés LOOP et END LOOP. Dès que le programme atteint le mot clé END LOOP, l'exécution est renvoyée à l'instruction LOOP qui précède.
- Une boucle de base permet l'exécution des instructions associées au moins une fois même si la condition de sortie est déjà vérifiée avant d'entrer dans la boucle.
- En l'absence du mot clé EXIT la boucle est infinie.
- Vous pouvez mettre fin à la boucle en utilisant la commande EXIT.

- La commande EXIT doit se trouver dans la boucle, elle peut dépendre soit d'une condition (IF) soit d'une instruction autonome.
- Dans le dernier cas, vous pouvez utiliser la commande WHEN, pour sortir de la boucle sous une certaine condition. Quand la commande EXIT est rencontrée, la condition WHEN est évaluée.
- Si la condition est vérifiée, la boucle se termine et l'exécution du programme reprend derrière le END LOOP. Une boucle de base peut contenir plusieurs instructions EXIT.

Syntaxe

LOOP **-- initialisation**

instruction1 **-- instructions**

. . .

EXIT [WHEN *condition*]; -- EXIT instruction

END LOOP; -- fermeture de la boucle

- **Où : *condition* est une variable booléenne ou une expression (TRUE, FALSE, ou NULL);**

Exemple

```
DECLARE
 v_qt        item.qte_stock%TYPE := 601;
 v_counter           NUMBER(2) := 1;
BEGIN
 LOOP
   INSERT INTO item(id_item,qte_stock)
    VALUES(v_counter,v_qt);
   v_counter := v_counter + 1;
   EXIT WHEN v_counter > 10;
 END LOOP;
END;
```

Boucle FOR

FOR *counter* in [REVERSE]
***borne_inférieure..borne_supérieure* LOOP**
***instruction1*;**
***instruction2*;**
. . .
END LOOP;

- ***compteur*** :: C'est un entier déclaré implicitement, dont la valeur décroît ou augmente automatiquement (elle décroît si le mot clé REVERSE est utilisé) de 1 à chaque itération de la boucle jusqu'à ce que la borne supérieure ou inférieure soit atteinte
- **REVERSE** :: Ce mot clé indique que le compteur décroît à chaque itération de la boucle depuis la borne supérieure jusqu'à la borne inférieure (Remarquez que la borne inférieure est toujours indiquée en premier).

Boucle WHILE

- Vous pouvez utiliser la boucle WHILE pour répéter un jeu d'instructions jusqu'à ce que la condition ne soit plus vérifiée.
- La condition est évaluée au début de chaque itération. La boucle se termine lorsque la condition n'est plus vérifiée.
- Si la condition n'est pas vérifiée dès le début de la boucle, cette boucle ne sera pas exécutée.

WHILE *condition* LOOP

***instruction1*;**

***instruction2*;**

. . .

END LOOP;

Exemple

- Dans l'exemple de la diapositive, des lignes de commandes sont ajoutés à la table ITEM pour une commande donnée. L'utilisateur se voit demander le numéro de commande (p_new_order) et le nombre de lignes pour cette commande (p_items).
- A chaque itération de la boucle WHILE le compteur (v_count) est incrémenté. Si le nombre d'itérations est inférieur ou égal au nombre de lignes de cette commande, les instructions de la boucle sont exécutées et une ligne est ajoutée à la table ITEM.
- Dès que le compteur dépasse le nombre de lignes pour cette commande, la condition régissant la boucle n'est plus vérifiée et la boucle se termine.

```
ACCEPT p_new_order PROMPT 'Entrez le numéro de commande: '
ACCEPT p_items  PROMPT 'Entrez le nombre de lignes pour la
    commande: '
DECLARE
v_count         NUMBER(2) := 1;
BEGIN
  WHILE v_count <= &p_items LOOP
    INSERT INTO item (ordid, itemid)
    VALUES (&p_new_order, v_count);
    v_count := v_count + 1;
  END LOOP;
  COMMIT;
END;
/
```

Boucles imbriquées et Etiquettes

- Vous pouvez imbriquer des boucles à plusieurs niveaux. Vous pouvez imbriquer des boucles basiques, FOR ou WHILE . La fin d'une boucle imbriquée ne signifie pas la fin de celle qui la contient sauf exception.
- Cependant la commande EXIT permet de quitter la boucle maître en faisant référence au label ou étiquette.
- Les noms des étiquettes suivent les mêmes règles que les autres identifiants.
- L'étiquette est placée avant l'instruction, soit sur la même ligne, soit sur une ligne distincte. Etiquetez les boucles en introduisant des labels avant le mot LOOP entre les (<<*label*>>).
- Si la boucle est étiquetée, le nom du label peut être inséré éventuellement après le mot clé END LOOP pour plus de clarté.

Exemple

```
BEGIN
 <<Maitre_loop>>
 LOOP
  v_counter := v_counter+1;
 EXIT WHEN v_counter>10;
  <<Detail_loop>>
  LOOP
    ...
    EXIT Maitre_loop WHEN total_done = 'YES';
    -- Quitter les 2 boucles
    EXIT WHEN Detail_done = 'YES';
    -- Quitter la boucle détail seulement
    ...
  END LOOP Detail_loop;
  ...
 END LOOP Maitre_loop;
END;
```

Curseurs

Curseurs

- Le serveur Oracle utilise des zones de travail appelées *zones SQL privées* pour exécuter les instructions SQL et pour stocker les informations en cours de traitement. Vous pouvez utiliser les curseurs *PL/SQL* pour nommer une zone SQL privée et accéder aux données qu'elle contient.
- Un curseur est une zone mémoire qui permet de traiter individuellement chaque ligne renvoyé par un ordre SELECT.
- Chaque instruction SQL exécutée par le serveur Oracle a son propre curseur individuel qui lui est associé :
  - Curseurs implicites : déclarés pour toutes les instructions LMD et SELECT PL/SQL
  - Curseurs explicites : déclarés et nommés par le programmeur

Curseurs Implicites

- Le serveur Oracle ouvre implicitement un curseur pour traiter chaque instruction *SQL* non associée à un curseur explicitement déclaré.
- Le PL/SQL laisse au programmeur la possibilité de se référer au plus récent curseur implicite par le biais du *curseur SQL*.
- Vous ne pouvez pas utiliser les instructions OPEN, FETCH, et CLOSE pour contrôler le *curseur SQL*, mais vous pouvez utiliser les attributs du curseur pour obtenir des informations sur l'exécution de la dernière commande SQL.

Les Attributs d'un Curseur Implicite

- ***SQL%NOTFOUND*** Evalué à TRUE si le dernier fetch n'a pas retourné de ligne.
- ***SQL%FOUND*** Évalué à TRUE si le dernier fetch a retourné une ligne ; complément de %NOTFOUND
- ***SQL%ROWCOUNT*** Contient le nombre total de lignes retournées jusqu'ici.

Exemple avec attributs de curseur implicite

```
declare
rec item%ROWTYPE;
begin
select * into rec from item where id_item=24;
if SQL%FOUND THEN
dbms_output.put_line('le nombre d''enregistrement est :' ||
   SQL%ROWCOUNT);
end if;
end;
/
```

le nombre d'enregistrement est : 1

Procédure PL/SQL terminée avec succés.

Curseurs Explicites

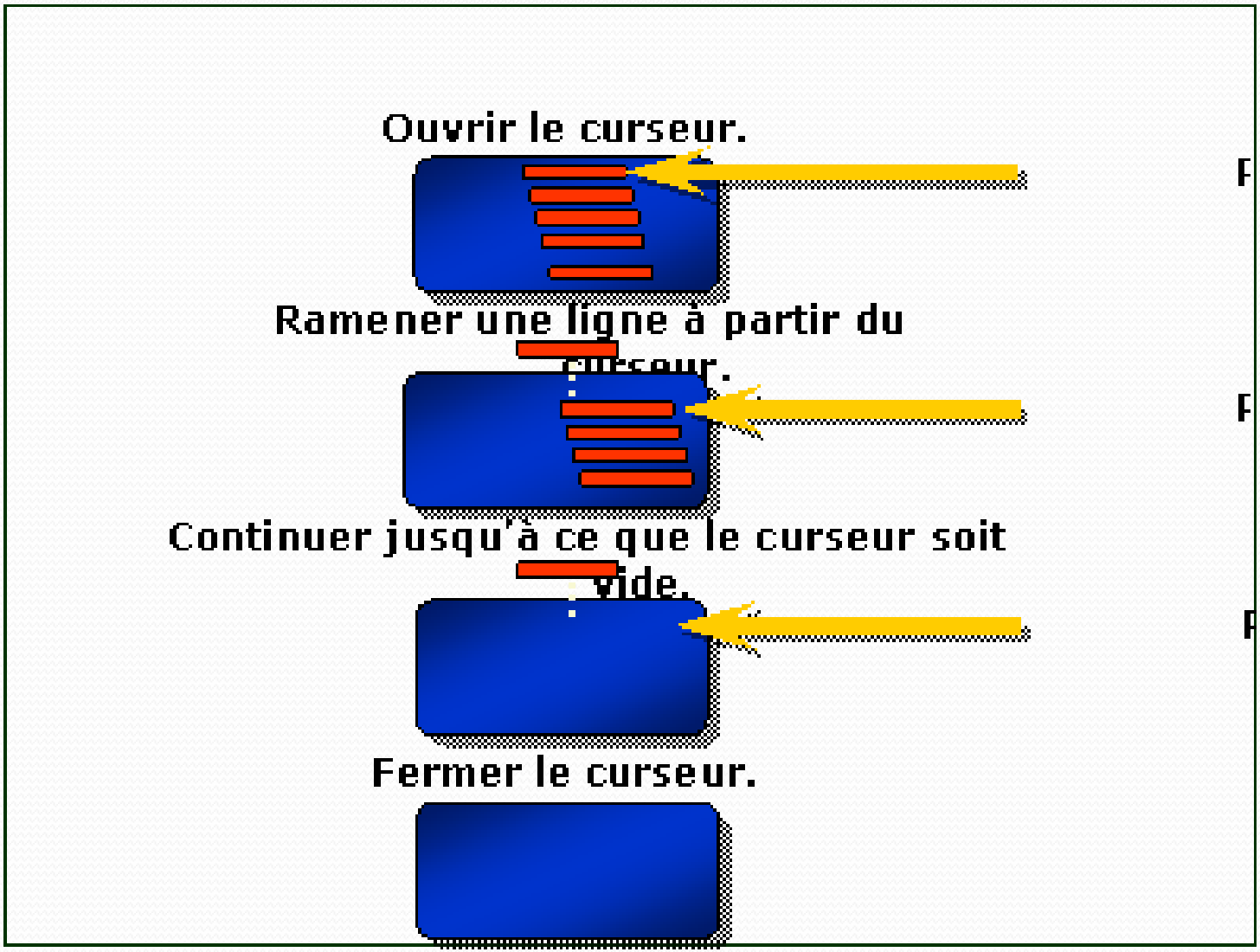
- Utilisez les curseurs explicites pour traiter individuellement chaque ligne retournée par une instruction SELECT.
- Cet ensemble de lignes retournées par une requête multi-lignes est appelé ***Ensemble Actif*** . Sa taille est égale au nombre de lignes répondant aux critères de recherche.
- Ceci permet à votre programme de traiter les enregistrements de la requête ligne par ligne.
- Un programme PL/SQL ouvre un curseur, traite les lignes retournées par la requête, et ensuite ferme le curseur. Le curseur marque la position courante dans l'Ensemble Actif.

Contrôler les Curseurs Explicites

- ***Déclarer le curseur*** en le nommant et en définissant la requête à exécuter (***DECLARE***).
- ***Ouvrir le curseur***. L'instruction ***OPEN*** exécute la requête avec toutes les Binds variables. Les lignes identifiées par celle-ci constitues l'*Ensemble Actif* et sont maintenant disponibles pour être traitées.
- ***Ramener les données*** du curseur. L'instruction ***FETCH*** charge la ligne courante du curseur dans des variables.
- Fermer le curseur avec la directive Close.

- Chaque fetch fait avancer le pointeur du curseur vers la ligne suivante dans *l'Ensemble Actif*. C'est pourquoi chaque fetch donne accès à une ligne différente retournée par la requête.
- après chaque fetch il faut tester si le curseur contient encore des lignes. Si des lignes sont trouvées, le fetch charge la ligne courante dans des variables, sinon il faut fermer le curseur.

- Fermer le curseur. L'instruction CLOSE libère *l'Ensemble Actif* de lignes considéré jusqu'à présent. Il est possible de ré-Ouvrir le curseur pour ré-Activer l'Ensemble de lignes.


Ouvrir le curseur.
Ramener une ligne à partir du
curseur.
Continuer jusqu'à ce que le curseur soit
vide.
Fermer le curseur.

Déclarer un Curseur

- Utilisez l'instruction CURSOR pour déclarer un curseur explicite. Vous pouvez référencer des variables dans la requête mais vous devez les déclarer avant l'instruction CURSOR.
- **CURSOR *cursor_name* IS**
- ***select_statement;***
- Dans la syntaxe:
- *cursor_name* est l'identifiant PL/SQL du curseur
- *select_statement* est une instruction SELECT
- N'incluez pas de clause INTO dans la déclaration de curseur parce qu'elle apparaît plus tard dans l'instruction FETCH.

Exemple

```
DECLARE
  CURSOR emp_cursor IS
    SELECT empno, ename
          FROM   emp;
   CURSOR dept_cursor IS
    SELECT *
    FROM   dept
    WHERE  deptno = 10;
BEGIN
....
```

Ouvrir un Curseur

- L'instruction OPEN
- Ouvrez le curseur pour exécuter la requête et identifier *l'Ensemble Actif*, constitué de toutes les lignes répondant aux critères de recherche. Le curseur pointe à présent sur l'avant première ligne de *l'Ensemble Actif*.
- **OPEN cursor_name;**
- OPEN est une instruction qui accomplit les opérations suivantes :
  - Effectue des allocations dynamiques de mémoire pour stocker les informations cruciales du curseur.
  - Analyse de l'ordre SELECT
  - Relie les variables d'entrée—c'est-à-dire, fixe la valeur des variables d'entrée en obtenant leur adresse mémoire.
  - Identifie *l'Ensemble Actif*—c'est-à-dire, l'ensemble des lignes qui satisfont les critères de recherche.
  - Positionne le pointeur juste avant la première ligne dans *l'Ensemble Actif* .

L'instruction FETCH

- L'instruction FETCH ramène les lignes de *l'Ensemble Actif* une par une. Après chaque fetch, le pointeur du curseur se place devant la ligne suivante de *l'Ensemble Actif* .
- **FETCH *cursor_name* INTO [*variable1, variable2, …*]**
- **| *record_name*];**
- *cursor_name :* est le nom du curseur précédemment déclaré
- *variable :* est une variable de sortie pour stocker les résultats
- *record_name :* est le nom du record dans lequel est placé la donnée ramenée

- L'instruction FETCH accomplit les opérations suivantes :
  - Avance le pointeur jusqu'à la nouvelle ligne dans *l'Ensemble Actif* .
  - Ramène les données de la ligne courante dans les variables de sortie PL/SQL.
  - Sors le curseur de la boucle FOR si le pointeur est positionné à la fin de *l'Ensemble Actif*.
- *Chargez les valeurs de la ligne courante dans des variables de sortie:*
  - *Prévoir le même nombre de variables.*
  - *Veiller à avoir le même type de données entre des variables de sortie et les données récupérées avec SQL statement.*
  - *Ajuster la position des variables par rapport aux colonnes.*
  - *Testez si le curseur contient des lignes.*

Exemple

```
DECLARE
CURSOR cursor_name IS …
BEGIN
OPEN defined_cursor;
LOOP
  FETCH defined_cursor INTO defined_variables
  EXIT WHEN …;
  …
    -- Traiter les données ramenées
  …
CLOSE defined cursor
END;
```

Fermer un Curseur

- L'instruction CLOSE désactive le curseur, et *l'Ensemble Actif* n'est plus défini. Fermer le curseur après avoir achevé le traitement de l'instruction SELECT.
- Cette étape permet au curseur d'être ré-ouvert, si nécessaire. C'est pourquoi vous pouvez établir *un Ensemble Actif* plusieurs fois.
- **CLOSE *cursor_name ;***
- N'essayez pas de ramener des données à partir d'un curseur fermé, sans quoi l'exception ***INVALID_CURSOR*** apparaîtra.
- L'instruction CLOSE annule le contexte mémoire.

Les Attributs d'un Curseur Explicite

- ***%ISOPEN*** Evalué à TRUE si le curseur est ouvert.
- ***%NOTFOUND*** Evalué à TRUE si le dernier fetch n'a pas retourné de ligne.
- ***%FOUND*** Évalué à TRUE si le dernier fetch a retourné une ligne ; complément de %NOTFOUND
- ***%ROWCOUNT*** Contient le nombre total de lignes retournées jusqu'ici.

- Vous pouvez ramener des lignes seulement si le curseur est ouvert. Utilisez l'attribut %ISOPEN pour savoir si le curseur est ouvert.
- Ramenez les lignes dans une boucle. Utilisez les attributs de curseur pour déterminer quand sortir de la boucle.
- Utilisez l'attribut de curseur %ROWCOUNT pour ramener un nombre exact de lignes. Traiter les lignes dans une boucle FOR, ou dans une boucle simple en utilisant un test sur la valeur du %ROWCOUNT pour sortir de la boucle.

Exemple

```
DECLARE
  CURSOR emp_cursor IS SELECT empno, ename
        FROM   emp;
BEGIN
IF NOT emp_cursor%ISOPEN THEN
     OPEN emp_cursor;
END IF;
LOOP
  FETCH emp_cursor...
  EXIT WHEN emp_cursor%NOTFOUND;
.......
```

Curseurs et Records

- Vous pouvez aussi définir un enregistrement basé sur une liste choisie de colonnes dans un curseur explicite. C'est pratique pour traiter les lignes de *l'Ensemble Actif*, parce que vous pouvez simplement faire le fetch dans le Record.
- C'est pourquoi les valeurs de la ligne sont chargées directement dans les champs correspondant du record.
- **DECLARE**
- **CURSOR emp_cursor IS**
- **SELECT empno, ename**
- **FROM emp;**
- **emp_record emp_cursor%ROWTYPE;**
- **BEGIN**
- **OPEN emp_cursor;**
- **LOOP**
- **FETCH emp_cursor INTO emp_record;**
- **…**

Curseur dans une boucle FOR

- Un curseur dans une boucle FOR traite les lignes dans un curseur implicite.
- C'est une Simplification d'écriture parce que : l'ouverture, le fetch des lignes et la fermeture du curseur, une fois toutes les lignes traitées, sont implicitement gérés.
- La boucle elle-même est terminée automatiquement à la fin du traitement de la dernière ligne.
- **FOR *record_name* IN *cursor_name* LOOP**
- ***statement1*;**
- ***statement2*;**
- **. . .**
- **END LOOP;**

Règles

- Ne déclarez pas le record qui contrôle la boucle. Sa portée est seulement dans la boucle.
- Testez les attributs du curseur pendant la boucle, si nécessaire.
- Prévoir un curseur paramétré, si nécessaire, en passant les paramètres entre parenthèses derrière le nom du curseur dans la boucle FOR.
- N'utilisez pas un curseur dans une boucle FOR lorsque les opérations du curseur doivent être gérées manuellement.

Exemple

```
DECLARE
  CURSOR emp_cursor IS
    SELECT ename, deptno
    FROM   emp;
BEGIN
  FOR emp_record IN emp_cursor LOOP
        -- un open et un fetch implicites ont lieu
    IF emp_record.deptno = 30 THEN
      ...
  END LOOP; -- un close implicite a lieu
END;
```

Curseurs paramétrés

- Les paramètres permettent de passer des valeurs à un curseur lors de son ouverture et de les utiliser dans la requête lors de l'exécution.
- Cela signifie que vous pouvez ouvrir et fermer plusieurs fois un curseur explicite dans un bloc, en passant à chaque fois un paramétrage différent.
- Chaque paramètre formel dans la déclaration du curseur doit avoir une valeur correspondant au moment de l'OPEN.
- Les types de donnée des paramètres sont les mêmes que ceux des variables scalaires, mais sans Taille. Les noms des paramètres servent de référence dans la requête.
- **CURSOR nom_de_curseur**
- **[(nom_de_parametre *type_de_donnée*, ...)]**
- **IS *instruction_SELECT*;**

Exemple 1 (Open)

```
DECLARE
  CURSOR emp_cursor
  (v_deptno NUMBER, v_job VARCHAR2) IS
    SELECT       empno, ename
    FROM  emp
    WHERE       deptno = v_deptno
     AND   job = v_job;
BEGIN
  OPEN emp_cursor(10, 'SALEMAN');
...
```

Exemple 2 (FOR)

```
DECLARE
  CURSOR emp_cursor
  (v_deptno NUMBER, v_job VARCHAR2) IS
    SELECT       empno, ename
    FROM  emp
    WHERE       deptno = v_deptno
     AND   job = v_job;
BEGIN
  FOR rec IN emp_cursor(10, 'CLERK') LOOP;
...
```

La Clause FOR UPDATE

- Vous pouvez vouloir verrouiller les lignes avant de les mettre à jour ou de les supprimer. Ajoutez la clause FOR UPDATE dans la requête du curseur afin de verrouiller les enregistrements lus lorsque le curseur est ouvert.
- Comme le Serveur Oracle libère les verrous à la fin de la transaction, il vaudrait mieux éviter de faire des commits intermédiaires si la clause FOR UPDATE est utilisée.
- **SELECT ...**
- **FROM ...**
- **FOR UPDATE [OF *column_reference*][NOWAIT|WAIT nb_scd]**
- ***column_reference*** est une colonne de le table sur laquelle la requête porte (une liste de colonnes peut aussi être utilisée.)
- **NOWAIT** retourne une erreur Oracle si les enregistrements sont verrouillés par une autre session.

Exemple

```
DECLARE
  CURSOR emp_cursor IS
    SELECT empno, ename, sal
    FROM   emp
    WHERE  deptno = 30
    FOR UPDATE NOWAIT;
```

La Clause WHERE CURRENT OF

- Pour référencer l'enregistrement courant depuis un curseur explicite, utilisez la clause WHERE CURRENT OF.
- Cela vous permet d'appliquer des mises à jour ou des suppressions sur l'enregistrement courant, sans avoir à le référencer explicitement par ROWID.
- Vous devez inclure la clause FOR UPDATE dans la requête du curseur afin que les enregistrements soient verrouillés sur OPEN.

Exemple

```
DECLARE
  CURSOR sal_cursor IS
    SELECT       sal
    FROM  emp
    WHERE       deptno = 30
    FOR UPDATE NOWAIT;
BEGIN
  FOR emp_record IN sal_cursor LOOP
    UPDATE      emp
    SET     sal = emp_record.sal * 1.10
    WHERE CURRENT OF sal_cursor;
  END LOOP;
  COMMIT;
END;
```

Curseurs avec des Sous-Requêtes

- Dans cet exemple, la sous-requête crée une source de données constituée du numéro de département et du nombre d'employés dans chaque département (sous l'alias STAFF).
- Une table, sous l'alias t2, renvoie à cette source temporaire de données dans la clause FROM. Quand ce curseur est ouvert, *l'Ensemble Actif* contient le numéro, le nom et le nombre d'employés par département ayant au moins 5 employés.
- Les sous-requêtes et sous-requêtes corrélées peuvent être utilisées.

Exemple

```
DECLARE
  CURSOR my_cursor IS
    SELECT t1.deptno, dname, STAFF
    FROM   dept t1, (SELECT deptno,
                     count(*) STAFF
                    FROM   emp
                    GROUP BY deptno) t2
   WHERE  t1.deptno = t2.deptno
   AND    STAFF >= 5;
```

Curseur dynamique

```
Begin
For v_emp in (SELECT ename, job, hiredate
                    FROM emp
                    WHERE ename like 'S') LOOP
Dbms_output.put_line(v_emp.ename ||'—
  '||v_emp.job||'—'||v_emp.hiredate);
End loop;
End;
/
```

Exceptions

Traitement des Exceptions

- Une exception est un identifiant PL/SQL de type erreur généré au cours de l'exécution d'un bloc qui termine le corps principal des instructions. Un bloc s'arrête quand une exception PL/SQL est déclenchée, cependant vous pouvez spécifier un traitement dans les exceptions afin de réaliser des instructions finales.
- Deux méthodes pour déclencher une Exception
  - Lorsqu'une erreur Oracle se produit, l'exception associée est émise automatiquement. Par exemple, si l'erreur ORA-01403 survient car un SELECT n'a ramenée aucune ligne, alors le PL/SQL émet l'exception NO_DATA_FOUND.
  - Dans le programme. (RAISE)

Types d'exceptions (1/2)

- Les exceptions systèmes nommées, sont des exceptions auxquelles PL/SQL a attribué des noms et qui sont déclenchées suite d'une erreur de traitement de PL/SQL ou oracle.
- Les exceptions utilisateurs nommées, sont des exceptions déclenchées à la suite des erreurs dans le code applicatif. Elle sont nommées lors de leurs déclaration dans la section déclaration. On les déclenche explicitement dans l'exécution du programme.

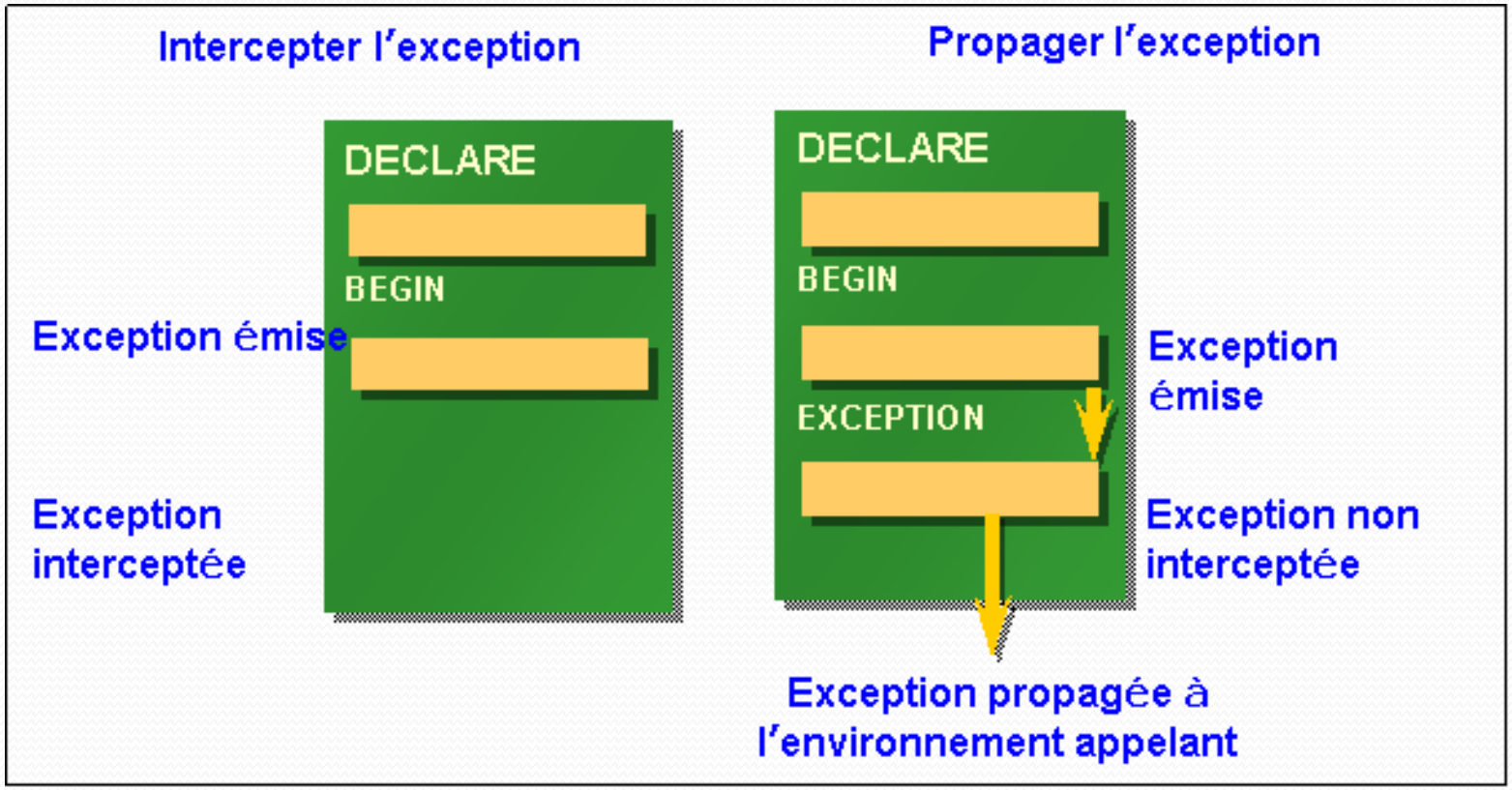
Types d'exceptions (2/2)

- Les exception systèmes anonymes, sont des exceptions qui se déclenchent à la suite d'une erreur de traitement de PL/SQL ou oracle, mais auxquelles PL/SQL n'a pas attribué de nom, seuls les erreurs les plus courant sont nommées; les autres sont numérotés et on peut leur attribuer des noms avec le pragma EXCEPTION_INIT.
- Les exceptions utilisateurs anonymes sont définies et déclenchées par le programmeur. Celui- ci définit un code comprise entre -20 000 et -22 999, et un message d'erreur. Il déclenche l'exception avec l'ordre RAISE_APPLICATION_ERROR.

Interception d'une Exception

- Si une exception se produit dans la partie exécutable du bloc, le traitement se débranche au sous-programme correspondant dans la section Exception du bloc.
- Si le PL/SQL traite convenablement l'exception, alors elle n'est pas propagée au bloc supérieur ou vers l'environnement appelant. Le bloc PL/SQL s'exécute correctement.
- Chaque traitement d'exception contient une commande WHEN, qui spécifie les conditions de l'exception, suivie d'une séquence d'instructions à exécuter lorsque cette exception est déclenchée.

Propagation d'une Exception



Syntaxe

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    instruction1;
    instruction2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    instruction3;
    instruction4;
    . . .]
  [WHEN OTHERS THEN
    instruction5;
    instruction6;
    . . .]
```

Règles pour intercepter les Exceptions

- Commencer la section Exception du bloc par le mot clé EXCEPTION.
- Définir plusieurs traitements d'Exceptions pour un bloc, chacune ayant sa propre séquence d'actions.
- Lorsqu'une exception se produit, le PL/SQL n'exécute *qu'un Seul* traitement avant de sortir du bloc.
- Placer la commande OTHERS après toutes les autres commandes de traitement d'exceptions.
- WHEN OTHERS est la dernière clause.
- Le mot-clé EXCEPTION débute la section de gestion des exceptions.
- Une seule exception est exécutée avant de sortir d'un bloc.

Erreurs Oracle pré-définies

- Utiliser le nom standard à l'intérieur de la section Exception.
- Exemple d'exceptions pré-définies :
- NO_DATA_FOUND
- TOO_MANY_ROWS
- INVALID_CURSOR
- ZERO_DIVIDE

Interception des erreurs Oracle pré-définies

```
BEGIN
  SELECT … COMMIT;
EXCEPTION
 WHEN NO_DATA_FOUND THEN
   instruction1;
   instruction2;
 WHEN TOO_MANY_ROWS THEN
   instruction1;
 WHEN OTHERS THEN
   instruction1;
   instruction2;
   instruction3;
END;
```

Interception d'erreurs Oracle non pré-définies

- Vous pouvez intercepter une erreur Oracle non pré-définie en la déclarant au préalable, ou en utilisant la commande OTHERS. L'exception déclarée est implicitement déclenchée.
- En PL/SQL, la clause *pragma EXCEPTION_INIT* permet d'associer un nom d'exception à un code d'erreur Oracle.
- Ceci vous permet de faire référence à n'importe quelle exception interne Oracle, par un nom et d'écrire un traitement spécifique pour celle-ci.

Exemple: Intercepter une erreur de violation de contraintes (Erreur Oracle –2292)

```
DECLARE
Viol_Contraint EXCEPTION;
PRAGMA EXCEPTION_INIT(Viol_Contraint,-2292);
 v_deptno      dept.deptno%TYPE := &p_deptno;
BEGIN
 DELETE FROM dept
 WHERE  deptno = v_deptno;
 COMMIT;
EXCEPTION
 WHEN   Viol_Contraint    THEN
 DBMS_OUTPUT.PUT_LINE('Suppression Impossible du
 dep:'||TO_CHAR(v_deptno)||'Il ya des employé s affectés à ce
   département');
END;
```

Exceptions définies par l'utilisateur

- Le langage PL/SQL vous permet de définir vos propres exceptions. Les exceptions définies par l'utilisateur en PL/SQL doivent être :
  - Déclarées dans la section DECLARE du bloc PL/SQL
  - Déclenchées explicitement à l'aide de l'instruction RAISE
- Vous pouvez intercepter une exception définie par l'utilisateur en la déclarant et en la déclenchant explicitement.
  - Déclarer le nom de l'exception dans la section DECLARE.
  - Utiliser l'instruction RAISE pour déclencher explicitement l'exception dans la section EXECUTABLE.
  - Traiter l'exception ainsi déclarée dans la section EXCEPTION.

Exemple

```
DECLARE
  e_invalid_emp  EXCEPTION;
BEGIN
  UPDATE emp
  SET             ename = '&nom_employee'
  WHERE  empno = &emp_matricule;
  IF SQL%NOTFOUND THEN
    RAISE e_invalid_emp;
  END IF;
  COMMIT;
EXCEPTION
WHEN e_invalid_emp   THEN
DBMS_OUTPUT.PUT_LINE('matricule d''employee invalide.');
END;
```

Fonctions d'interception des Erreurs

- Lorsqu'une exception se produit, vous pouvez identifier le code et le message d'erreur associer à l'aide de deux fonctions.
- Suivant le message ou la valeur du code, vous pouvez décider quelle action effectuer après cette erreur.
- SQLCODE renvoie le code Oracle de l'erreur pour les exceptions internes.
- Vous pouvez transmettre le code de l'erreur à SQLERRM, qui renvoie alors le message associé ainsi que le code de l'erreur.

Exemple

```
DECLARE
 v_error_code     NUMBER;
 v_error_message   VARCHAR2(255);
BEGIN
…
EXCEPTION
…
 WHEN OTHERS THEN
   ROLLBACK;
   v_error_code := SQLCODE ;
   v_error_message := SQLERRM ;
   INSERT INTO erreurs VALUES(v_error_code,
                              v_error_message);
END;
```

Propagation des exceptions

- Au lieu d'intercepter une exception dans un bloc PL/SQL, vous pouvez propager l'exception afin de permettre à l'environnement appelant de la traiter. Chaque environnement a sa propre façon d'accéder et d'afficher les erreurs.
- Lorsqu'un sous-bloc traite une exception, il se termine normalement,et l'exécution du traitement se poursuit dans le bloc supérieur, juste après l'instruction END du sous-bloc.
- Cependant, si un programme PL/SQL produit une exception et que le bloc ne prévoit pas de la traiter, elle se propage successivement dans les blocs supérieurs jusqu'à trouver une fonction qui la gère.
- Si aucun de ces blocs ne traite l'exception, le résultat donne une exception de non gérée dans l'environnement hôte.
- Lorsque l'exception se propage à un bloc supérieur, les actions exécutables restantes de ce bloc sont ignorées.

Exemple

```
DECLARE
 . . .
 e_no_rows    exception;
 e_integrity    exception;
 PRAGMA EXCEPTION_INIT (e_integrity, -2292);
BEGIN
 FOR c_record IN emp_cursor LOOP
       BEGIN
        SELECT ...
        UPDATE ...
        IF SQL%NOTFOUND THEN
          RAISE e_no_rows;
        END IF;
       EXCEPTION
        WHEN e_integrity THEN ...
        WHEN e_no_rows THEN ...
       END;
       END LOOP;
EXCEPTION
 WHEN NO_DATA_FOUND THEN . . .
 WHEN TOO_MANY_ROWS THEN . . .
END;
```

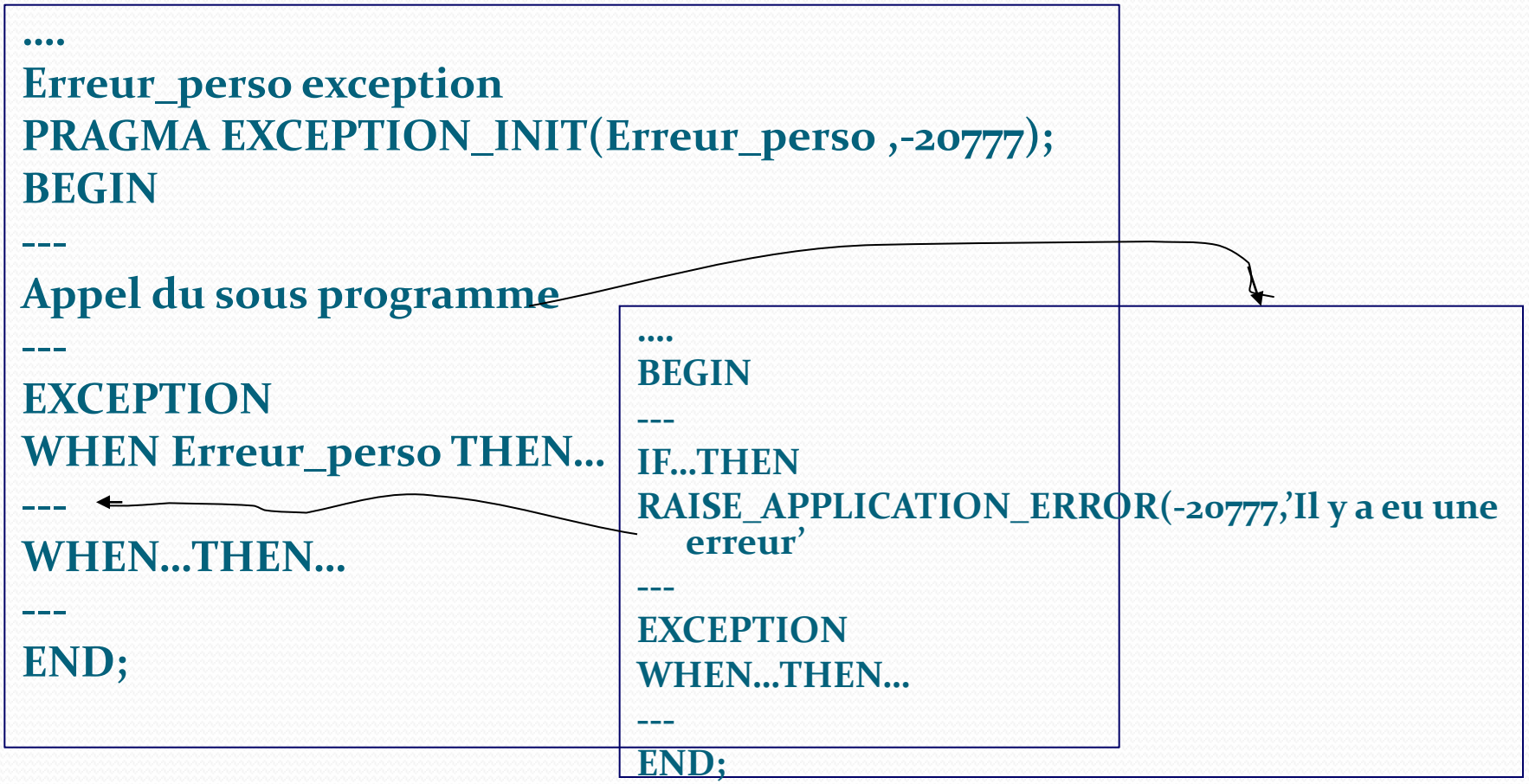
Procédure RAISE_APPLICATION_ERROR

- Cette procédure vous permet de délivrer des messages d'erreur définis par l'utilisateur à partir de sous-programmes stockés
- Elle ne peut être appelée que durant l'exécution d'un sous-programme stocké dans la base de données
- Utilisez la procédure RAISE_APPLICATION_ERROR pour renvoyer, une exception pré-définie, en utilisant un code d'erreur non standard et un message d'erreur. Avec RAISE_APPLICATION_ERROR, vous pouvez reporter des erreurs dans votre application et éviter ainsi de renvoyer des exceptions non gérées.

Syntaxe

- raise_application_error (*error_number,message*[, {TRUE | FALSE}]);
- **Error_number : valeur défini par l'utilisateur pour l'exception, entre -20 000 et -20 999.**
- **TRUE | FALSE** est un paramètre Booléen optionnel (Si TRUE, l'erreur est rangée dans la pile des erreurs précédentes. Si FALSE , valeur par défaut l'erreur remplace toutes les erreurs précédentes.)

Exemple


....
Erreur_perso exception
PRAGMA EXCEPTION_INIT(Erreur_perso ,-20777);
BEGIN

Appel du sous programme

EXCEPTION
WHEN Erreur_perso THEN...

WHEN...THEN...

END;
....
BEGIN

IF...THEN
RAISE_APPLICATION_ERROR(-20777,'Il y a eu une erreur'

EXCEPTION
WHEN...THEN...

END;

Fonctions stockées

Fonctions stockées

- Une fonction stockée est un bloc PL/SQL nommé qui peut prendre des paramètres et être appelé. En règle générale, vous utilisez une fonction pour calculer une valeur.
- Une fonction doit comporter une clause RETURN dans l'en-tête, et au moins une instruction RETURN dans la section exécutable.
- Les fonctions ont pour avantage d'être réutilisables et révisables.
- En cas de changement de définition, seule la fonction se trouve affectée, ce qui simplifie grandement la maintenance.

- Les fonctions peuvent être appelées en tant que partie d'une expression SQL ou d'une expression PL/SQL.
- Dans une expression SQL, une fonction doit obéir à certaines règles destinées à contrôler les effets de bord qu’elle est susceptible d’entraîner.
- Dans une expression PL/SQL, l'identificateur de la fonction agit comme une variable dont la valeur dépend des paramètres qui lui ont été passés.

Syntaxe

CREATE [OR REPLACE] FUNCTION *function_name*
(*parameter1* [*mode1*] *datatype1*,
***parameter2* [*mode2*] *datatype2*,. . .)**
RETURN *datatype*
IS|AS
PL/SQL Block;

Création d'une fonction stockée avec SQL*Plus

- Entrer le texte de l'instruction CREATE FUNCTION dans un éditeur.
- Exécuter le fichier script pour compiler la fonction (@ ou SATART fichier_script.sql).
- Utiliser SHOW ERRORS pour visualiser les erreurs de compilation.
- Une fois compilée avec succès, la fonction est prête pour exécution.

Exemple

```
SQL> CREATE OR REPLACE FUNCTION get_sal
  2    (v_id  IN  emp.empno%TYPE)
  3    RETURN NUMBER
  4  IS
  5    v_salary   emp.sal%TYPE :=0;
  6  BEGIN
  7    SELECT sal
  8    INTO       v_salary
  9    FROM       emp
 10    WHERE     empno = v_id;
 11    RETURN (v_salary);
 12  END get_sal;
 13  /
```

Exécuter une fonction

- Une fonction peut comporter un ou plusieurs paramètres IN, mais ne doit retourner qu'une seule valeur. Vous appelez des fonctions utilisées au sein d'expressions PL/SQL, en utilisant des variables pour recevoir la valeur de retour.
- Syntaxe d'éxécution :

Execute Name_fct(valeur1,valeur2,…);:

- **SQL> START get_salary.sql**
- **Fonction créée.**
- **SQL> VARIABLE g_salary number**
- **SQL> EXECUTE :g_salary := get_sal(7934)**
- **Procédure PL/SQL terminée avec succès.**
- **SQL> PRINT g_salary**
- **G_SALARY**
- **----------**
- **1300**

Appeler des fonctions stockées à partir d'expressions SQL

- Les expressions SQL peuvent référencer des fonctions PL/SQL définies par l'utilisateur. Là où une fonction SQL intégrée peut être placée, une fonction définie par l'utilisateur peut l'être également.
- Avantages
  - Autorise des calculs trop complexes, mal conçus, ou indisponibles avec SQL
  - Renforce l'indépendance des données en traitant des analyses de données complexes dans Oracle Server, plutôt qu'en récupérant des données dans une application
  - Accroît l'efficacité des requêtes en réalisant des fonctions dans la requête plutôt que dans une application

Exemple

- Dans SQL*Plus, appelez la fonction TAX au sein d'une requête en affichant l'ID, le nom et le salaire de l'employé.
- **SQL> SELECT empno, ename, sal, tax(sal)**
  2 FROM emp;

Restrictions lors des appels de fonctions à partir d'expressions SQL(1/2)

- Pour qu'une expression SQL puisse être appelée, une fonction PL/SQL définie par l'utilisateur doit satisfaire quelques exigences.
  - Seules les fonctions stockées peuvent être appelées à partir d'instructions SQL. Les procédures stockées ne peuvent pas être appelées.
  - Les paramètres d'une fonction PL/SQL appelée à partir d'une instruction SQL doivent utiliser la notation de positionnement. La notation nommée n'est pas supportée.

Restrictions lors des appels de fonctions à partir d'expressions SQL (2/ 2)

- Les fonctions PL/SQL stockées ne peuvent pas être appelées à partir de la clause de contrainte CHECK d'une commande CREATE ou ALTER TABLE ou être utilisées pour spécifier une valeur par défaut d'une colonne.
- Vous devez être propriétaire de la fonction, ou avoir le privilège EXECUTE, pour l'appeler à partir d'une instruction SQL.

Contrôler les effets de bord(1/2)

- Pour exécuter une instruction SQL qui appelle une fonction stockée, Oracle Server doit savoir si la fonction n'entraînera pas d'effets de bord.
- Les effets de bord sont des changements apportés aux tables de la base de données.
- Les effets de bord pourraient retarder l'exécution d'une requête ou générer des résultats dépendant de l'ordre (par conséquent indéterminés).

Contrôler les effets de bord (2/2)

- Restrictions
  - La fonction ne peut pas modifier les tables de la base de données ; elle ne peut exécuter une instruction INSERT, UPDATE ou DELETE.
  - La fonction ne peut pas appeler un autre sous-programme qui annule l'une des restrictions mentionnées ci-dessus.

Suppression des fonctions

- Lorsqu'une fonction stockée n'est plus requise, vous pouvez utiliser une instruction SQL dans le volet de l'interpréteur de SQL*Plus ou de Procedure Builder pour la supprimer (DROP).
- Syntaxe
- **DROP FUNCTION *function_name***
- **Exemple**
- **SQL> DROP FUNCTION get_sal;**
- **Fonction supprimée.**

Procédures stockées

Qu'est-ce qu'une procédure

- Une procédure est un bloc PL/SQL qui accepte des paramètres (arguments), et qui peut être appelée.
- En règle générale, vous utilisez une procédure pour effectuer une action.
- Une procédure peut être stockée dans la base de données, comme objet de base de données, en vue d'exécutions répétées.
- Les procédures ont pour avantage d'être réutilisables et révisables. Une fois validées, elles peuvent être utilisées dans nombreux applications.
- En cas de changement de définition, seule la procédure se trouve affectée, ce qui simplifie grandement la maintenance.

Créer des procédures

- L'instruction CREATE PROCEDURE vous permet de créer de nouvelles procédures.
- Cette instruction permet de déclarer une liste de paramètres et doit définir les actions à réaliser par le bloc PL/SQL standard.
- Les blocs PL/SQL commencent par BEGIN ou par la déclaration de variables locales et finissent par END ou END *procedure_name*.
- **CREATE [OR REPLACE] PROCEDURE *procedure_name***
- **(*parameter1* [*mode1*] *datatype1*,**
- **p*parameter2* [*mode2*] *datatype2*,. . .)**
- **IS|AS Bloc PL/SQL ;**

Paramètres IN (2/ 2)

```
SQL> CREATE OR REPLACE PROCEDURE
  raise_salary
 2  (v_id in emp.empno%TYPE)
 3  IS
 4  BEGIN
 5   UPDATE emp
 6   SET    sal = sal * 1.10
 7   WHERE  empno = v_id;
 8  END raise_salary;
 9  /
```

Paramètres IN (2/ 2)

- L'exécution de cette instruction dans SQL*Plus crée la procédure RAISE_SALARY. Une fois appelée, RAISE_SALARY prend le paramètre du numéro d'employé et met à jour l'enregistrement relatif à l'employé et lui affecte une augmentation de 10 pour cent.
- Pour appeler une procédure dans SQL*Plus, utilisez la commande EXECUTE.
- **SQL> EXECUTE raise_salary (7369)**
- Les paramètres IN sont transmis comme des constantes de l'environnement appelant vers la procédure. Essayer de changer la valeur d'un paramètre IN se soldera par une erreur.

Paramètres OUT

```
SQL> CREATE OR REPLACE PROCEDURE query_emp
  1  (v_id     IN emp.empno%TYPE,
  2   v_name       OUT       emp.ename%TYPE,
  3   v_salary OUT     emp.sal%TYPE,
  4   v_comm      OUT       emp.comm%TYPE)
  5  IS
  6  BEGIN
  7    SELECT   ename, sal, comm
  8    INTO     v_name, v_salary, v_comm
  9    FROM     emp
 10    WHERE    empno = v_id;
 11  END query_emp;
 12  /
```

Paramètres OUT et SQL*Plus

```
SQL> VARIABLE g_name    VARCHAR2(15)
SQL> VARIABLE g_sal           NUMBER
SQL> VARIABLE g_comm   NUMBER
SQL> EXECUTE query_emp
   (7654,:g_name,:g_sal,:g_comm)
PL/SQL procedure successfully completed.
SQL> PRINT g_name
G_NAME
--------------
MARTIN
```

Paramètres IN OUT

- Avec un paramètre IN OUT, vous pouvez transmettre des valeurs dans une procédure et renvoyer une valeur vers l'environnement appelant.
- La valeur retournée est soit l'original, soit la valeur inchangée soit encore une nouvelle valeur définie dans la procédure.
- Un Paramètre IN OUT se comporte en variable initialisée.
- Exemple : Créez une procédure avec un paramètre IN OUT pour accepter une chaîne de caractères contenant 9 chiffres et renvoyer un numéro de téléphone formaté de la façon suivante : (+212) 063-305-175.

Exemple

```
SQL> CREATE OR REPLACE PROCEDURE
  format_phone
 2  (v_phone_no IN OUT VARCHAR2)
 3  IS
 4  BEGIN
 5   v_phone_no := '+212 ' || SUBSTR(v_phone_no,1,3)
   ||
 6              ' ' || SUBSTR(v_phone_no,4,3) ||
 7              ' ' || SUBSTR(v_phone_no,7);
 8  END format_phone;
 9 /
```

Appel de FORMAT_PHONE à partir de SQL*Plus

```
SQL> VARIABLE g_phone_no VARCHAR2(15)
SQL> BEGIN :g_phone_no := '060330575'; END;
  2  /
SQL> PRINT g_phone_no
G_PHONE_NO
---------------
060330575
SQL> EXECUTE format_phone (:g_phone_no)
SQL> PRINT g_phone_no
G_PHONE_NO
---------------
(+212)060-330-575
```

Méthodes de transfert des paramètres

```
SQL> CREATE OR REPLACE PROCEDURE add_dept
  1  (v_name  IN dept.dname%TYPE  DEFAULT
   'unknown',
  2   v_loc   IN dept.loc%TYPE    DEFAULT 'unknown')
  3  IS
  4  BEGIN
  5     INSERT INTO dept
  6     VALUES (dept_deptno.NEXTVAL, v_name, v_loc);
  7  END add_dept;
```

```
SQL> BEGIN
  2  add_dept;
  3  add_dept ( 'TRAINING', 'NEW YORK');
  4  add_dept ( v_loc => 'DALLAS', v_name =>'EDUCATION');
  5  add_dept ( v_loc => 'BOSTON') ;
  6  END;
  7  /
Procédure PL/SQL terminée avec succès.
SQL> SELECT * FROM dept;
DEPTNO   DNAME          LOC
------      -------------      ------------
   ...       ...                  ...
    41      unknown          unknown
    42      TRAINING         NEW YORK
    43      EDUCATION              DALLAS
    44   unknown             BOSTON
```

Déclaration de sous-programmes

- Vous pouvez déclarer des sous-programmes dans tout bloc PL/SQL. Voici une alternative à la création d'une procédure autonome LOG_EXEC.
- Vous devez déclarer le sous-programme, dans la section déclaration du bloc, mais il doit être le dernier élément, après tous les autres éléments du programme. Par exemple, une variable déclarée après la fin du sous-programme, avant le BEGIN de la procédure, génèrera une erreur de compilation.

Exemple

```
CREATE OR REPLACE PROCEDURE LEAVE_EMP2
  (v_id   IN   emp.empno%TYPE)
IS
  PROCEDURE log_exec
  IS
  BEGIN
    INSERT INTO log_table (user_id, log_date)
    VALUES (user,sysdate);
  END log_exec;
BEGIN
  DELETE FROM emp
  WHERE empno = v_id;
  log_exec;
END leave_emp2;
```

Triggers de manipulation des données

Présentation (1/2)

- Les triggers de manipulation de données s'exécutent de manière implicite lorsqu'une instruction INSERT, UPDATE ou DELETE (instruction de déclenchement) est émise sur la table connexe, et ce, quel que soit l'utilisateur connecté ou l'application utilisée.
- Ils s'exécutent de la même manière lorsque certaines actions se produisent (connexion d'un utilisateur ou fermeture de la base de données par l'administrateur, par exemple).

Présentation (2/2)

- Les triggers de base de données peuvent être définis sur des tables et des vues. Si une opération LMD est effectuée sur une vue, le trigger INSTEAD OF définit les actions qui doivent se produire. Si, au nombre des actions, figurent des opérations LMD sur des tables, tout trigger présent sur la (les) tables(s) de base sera déclenché.
- Un trigger est un bloc PL/SQL qui s'exécute implicitement à chaque fois qu'un événement spécifique se produit.
- Il existe deux types de triggers : les triggers de base de données et les triggers d'application.

Quand-t-il utiliser les triggers?

- Utilisez des triggers pour vous assurer que, lors de l'exécution d'une opération spécifique, les actions connexes seront, elles aussi, exécutées.
- N'utilisez ces triggers que pour les opérations globales et centralisées devant être déclenchées pour l'instruction de déclenchement, et ce, quel que soit l'utilisateur ou l'application qui a émis l'instruction.
- Ne définissez pas de triggers censés remplacer ou reproduire une fonctionnalité déjà intégrée dans la base de données Oracle. (implémenter des règles d'intégrité)
- L'usage abusif de triggers peut provoquer des interdépendances complexes, difficiles à gérer dans des applications volumineuses. prenez garde aux effets récursifs et en cascade.

Trigger de manipulation de données

- Avant de coder le corps du trigger, vous devez définir ses éléments constitutifs : temporisation du trigger, événement déclencheur et type de trigger.
- Temporisation du trigger
  - Pour la table : BEFORE, AFTER
  - Pour la vue : INSTEAD OF
- Événement déclencheur : INSERT, UPDATE ou DELETE
- Nom de l'objet : sur la table ou la vue
- Type de trigger : ligne ou instruction
- Clause "When" : condition restrictive
- Corps du trigger : bloc PL/SQL

Triggers BEFORE

- Ce type de trigger est utilisé fréquemment dans les situations suivantes :
- Lorsque le trigger doit déterminer si l'instruction de déclenchement doit, ou non, être autorisée à s'exécuter. (Cette situation vous permet d'éviter tout traitement inutile de l'instruction de déclenchement, ainsi que le rollback qui s'ensuit, en cas d'apparition d'une exception dans l'action de déclenchement).
- Pour obtenir les valeurs de colonne avant d'exécuter une instruction de déclenchement INSERT ou UPDATE.

Triggers AFTER

- Ce type de trigger est utilisé fréquemment dans les situations suivantes :
- Si vous souhaitez que l'instruction de déclenchement soit terminée avant l'exécution de l'action de déclenchement.
- S'il existe déjà un trigger BEFORE, et qu'un trigger AFTER peut effectuer différentes actions sur la même instruction de traitement.

Triggers INSTEAD OF

- Ce type de trigger permet de modifier, en toute transparence, des vues qu'il est impossible de modifier directement à l'aide d'instructions LMD SQL en raison de leur nature intrinsèquement non modifiable.
- Vous pouvez écrire des instructions INSERT, UPDATE et DELETE sur la vue. Le trigger INSTEAD OF s'exécute alors de manière invisible en tâche de fond, et effectue directement sur les tables sous-jacentes l'action codée dans le corps du trigger.

Temporisation du trigger

- quand le trigger doit-il se déclencher ?
- BEFORE : exécution du corps du trigger avant le déclenchement de l'événement LMD sur la table.
- AFTER : exécution du corps du trigger après le déclenchement de l'événement LMD sur la table.
- INSTEAD OF : exécution du corps du trigger au lieu de l'instruction de déclenchement. Utilisé pour les VUES qui, autrement, ne pourraient être modifiées.

Événement utilisateur déclencheur

- Toute instruction INSERT, UPDATE ou DELETE exécutée sur une table.
- S'il s'agit d'une instruction UPDATE, vous pouvez inclure une liste de colonnes afin d'identifier la ou les colonnes qui doivent être modifiées pour déclencher le trigger.
- . . . UPDATE OF sal . . .
- Cette spécification est impossible dans le cas des instructions INSERT et DELETE, car elles concernent l'ensemble des lignes
- L'événement déclencheur peut contenir plusieurs opérations LMD
- . . . INSERT or UPDATE or DELETE
- . . . INSERT or UPDATE OF job . . .

Type de Trigger

- Combien de fois le corps du trigger doit-il s'exécuter lorsque survient l'événement déclencheur ?
  - Instruction : par défaut, le corps du trigger s'exécute une seule fois pour l'événement déclencheur.
  - Ligne : le corps du trigger s'exécute une seule fois pour chaque ligne concernée par l'événement déclencheur.
- Si l'instruction de manipulation de données à la base du déclenchement concerne plusieurs lignes, le trigger d'instruction se déclenche à une seule reprise. Quant au trigger de ligne, il s'exécute une fois pour chacune des lignes affectées.

Corps du trigger

- Quelle action le trigger doit-il effectuer ?
- Le corps du trigger est un bloc PL/SQL ou un appel vers une procédure.
- Le bloc PL/SQL peut contenir des instructions SQL et PL/SQL, définir des structures PL/SQL telles que des variables, curseurs, exceptions, etc.
- Il convient encore d'ajouter que les triggers de ligne ont recours à des noms de corrélation pour accéder aux valeurs de colonne, anciennes ou nouvelles, de la ligne en cours de traitement.

Syntaxe relative à la création d'un trigger d'instruction

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

Exemple d’un trigger BEFORE

- Vous pouvez créer un trigger chargé de limiter à certaines heures ouvrables, du lundi au vendredi, les insertions effectuées dans la table EMP.
- Si, par exemple, un utilisateur tente d'insérer une ligne un dimanche, un message apparaît, le trigger échoue et l'instruction de déclenchement est annulée. RAISE_APPLICATION_ERROR est une procédure intégrée côté serveur qui imprime un message à l'attention de l'utilisateur et provoque l'échec du bloc PL/SQL.
- En cas d'échec d'un trigger de base de données, Oracle Server annule automatiquement l'instruction de déclenchement.

```
SQL> CREATE OR REPLACE TRIGGER secure_emp
  2  BEFORE INSERT ON emp
  3  BEGIN
  4   IF (TO_CHAR (sysdate,'DY') IN ('SAM.','DIM.')) OR
  5     (TO_CHAR(sysdate,'HH24') NOT BETWEEN
  6                          '08' AND '18')
  7   THEN RAISE_APPLICATION_ERROR (-20500,
  8   'You may only insert into EMP during normal
    hours.');
  9   END IF;
 10  END;
```

Combinaison d'événements déclencheurs

- Il est possible de rassembler plusieurs événements déclencheurs au sein d'un seul événement grâce aux prédicats conditionnels spéciaux INSERTING, UPDATING et DELETING disponibles dans le corps du trigger.
- Exemple
- Créez un trigger pour empêcher l'exécution de tout événement de manipulation de données sur la table EMP en dehors de certaines heures ouvrables, du lundi au vendredi.
- Faites également appel aux triggers d'instruction BEFORE pour initialiser des variables ou indicateurs globaux et valider les règles de gestion complexes.

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT OR UPDATE OR DELETE ON emp
BEGIN
 IF (TO_CHAR (sysdate,'DY') IN ('SAM.',‘DIM.')) OR
   (TO_CHAR (sysdate, 'HH24') NOT BETWEEN '08' AND '18')   THEN
  IF  DELETING
  THEN RAISE_APPLICATION_ERROR (-20502,
   'You may only delete from EMP during normal hours.');
   ELSIF   INSERTING
   THEN RAISE_APPLICATION_ERROR (-20500,
    'You may only insert into EMP during normal hours.');
     ELSE
    RAISE_APPLICATION_ERROR (-20504,
    'You may only update EMP during normal hours.');
   END IF;
  END IF;
END;
```

Syntaxe relative à la création d'un trigger de ligne

```
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
```

Exemple

- Vous pouvez créer un trigger de ligne BEFORE afin d'empêcher l'aboutissement d'une opération de déclenchement si une certaine condition est enfreinte.
- Créez un trigger permettant uniquement à certains employés d'avoir un salaire supérieur à 5000 unités.
- Si un utilisateur tente d'enfreindre cette condition, le trigger génère une erreur.

```
SQL> CREATE OR REPLACE TRIGGER CHECK_SALARY
  2    BEFORE  INSERT OR UPDATE OF sal   ON emp
  3  FOR EACH ROW
  4  BEGIN
  5    IF  NOT (:NEW.JOB IN ('MANAGER' , 'PRESIDENT'))
  6        AND :NEW.SAL > 5000
  7    THEN
  8      RAISE_APPLICATION_ERROR
  9        (-20202, 'EMPLOYEE CANNOT EARN THIS AMOUNT');
 10    END IF;
 11  END;
```

```
SQL> UPDATE EMP SET SAL = 6500 WHERE ENAME =
  'MILLER'
    2  /
  UPDATE EMP SET SAL = 6500 WHERE ENAME =
'MILLER'
         *
  ERROR at line 1:
  ORA-20202: EMPLOYEE CANNOT EARN THIS
AMOUNT
  ORA-06512: at "A_USER.CHECK_SALARY, line 5
  ORA-04088: error during execution of trigger
    'A_USER.CHECK_SALARY
```

Utilisation des qualificatifs OLD et NEW

- Dans un trigger ROW, faites référence à la valeur d'une colonne avant et après la modification des données en la faisant précéder du qualificatif OLD ou NEW.
- Les qualificatifs OLD et NEW sont disponibles uniquement dans les triggers ROW.
- Dans toute instruction SQL et PL/SQL, faites précéder les qualificatifs de deux points (:).
- Ne pas utiliser le préfixe (:) quand les qualificatifs sont référencés dans la condition de restriction WHEN.

Exemple

- La clause WHEN permet de limiter l'action du trigger aux lignes qui répondent à une certaine condition.
- Créez un trigger sur la table EMP pour calculer la commission d'un employé lors de l'ajout d'une ligne dans une table EMP ou de la modification du salaire d'un employé.
- Le qualificatif NEW ne doit pas être précédé de deux points dans la clause WHEN.

```
SQL>CREATE OR REPLACE TRIGGER
   derive_commission_pct
 2 BEFORE INSERT OR UPDATE OF sal ON emp
 3 FOR EACH ROW
 4 WHEN (NEW.job = 'SALESMAN')
 5 BEGIN
 6   IF  INSERTING
 7    THEN  :NEW.comm := 0;
 8   ELSIF :OLD.comm IS NULL
 9    THEN  :NEW.comm := 0;
10   ELSE  :NEW.comm := :OLD.comm * (:NEW.sal/:OLD.sal);
11   END IF;
12 END;
13 /
```

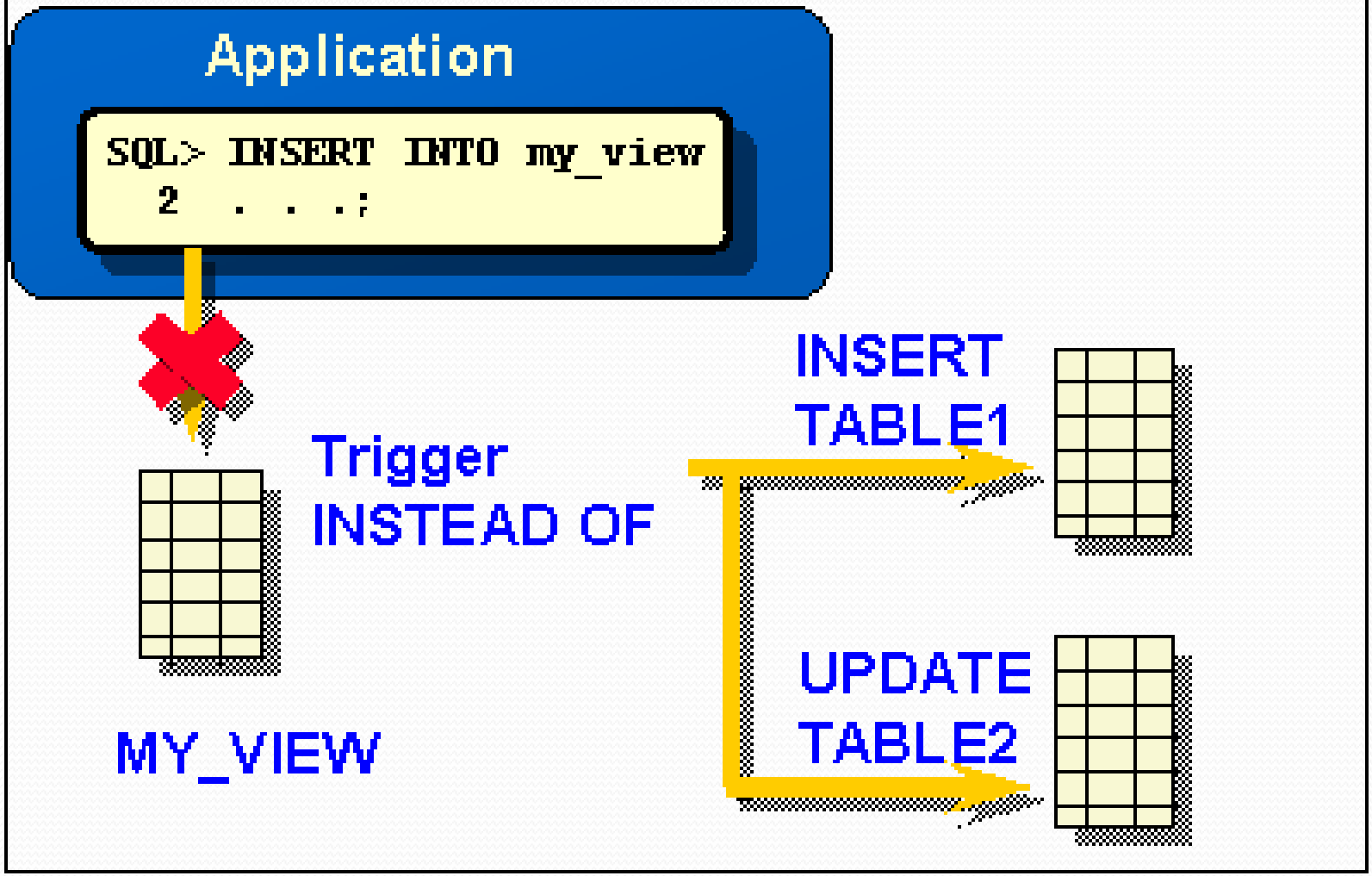
Triggers INSTEAD OF

- Les triggers INSTEAD OF vous permettent de modifier les données lorsqu'une instruction LMD a été émise relativement à une vue intrinsèquement non modifiable.
- On parle, dans ce cas, de "triggers INSTEAD OF" car, contrairement aux autres triggers, Oracle Server les déclenche au lieu d'exécuter l'instruction de déclenchement. Ce type de trigger sert à effectuer une opération INSERT, UPDATE ou DELETE directement sur les tables sous-jacentes.
- Supposons que vous écriviez des instructions INSERT, UPDATE ou DELETE sur une vue. Dans ce cas, le trigger INSTEAD OF s'exécute de manière invisible en tâche de fond afin de permettre l'exécution des actions appropriées.

Pourquoi utiliser des triggers INSTEAD OF?

- Si une vue comprend plusieurs tables, une insertion effectuée dans la vue pourrait entraîner une insertion dans une table et une mise à jour dans une autre. Vous pouvez donc écrire un trigger INSTEAD OF qui se déclenchera lorsque vous créez une insertion sur la vue. Le cas échéant, le corps du trigger s'exécute en lieu et place de l'insertion originale, ce qui se traduit par une insertion dans une table et une mise à jour dans une autre.

Trigger INSTEAD OF



Syntaxe relative à la création d'un trigger INSTEAD OF

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

Modes du trigger : Activé ou Désactivé

- Lors de sa création, le trigger est activé automatiquement.
- Dans le cas des triggers activés, Oracle Server vérifie les contraintes d'intégrité et s'assure que les triggers ne pourront y porter atteinte. De surcroît, Oracle Server fournit des vues à lecture cohérente pour les requêtes et les contraintes, il gère les dépendances et propose une validation à deux phases si un trigger met à jour des tables distantes dans une base de données répartie.
- Utilisez la syntaxe ALTER TRIGGER pour désactiver un trigger spécifique ou ALTER TABLE pour désactiver *tous* les triggers d'une table.

- Désactivation ou réactivation d'un trigger de base de données :
- **ALTER TRIGGER *trigger_name* *DISABLE* | *ENABLE***
- Désactivation ou réactivation de tous les triggers d'une table :
- **ALTER TABLE *table_name* *DISABLE* | *ENABLE* ALL TRIGGERS**
- Recompilation d'un trigger pour une table :
- **ALTER TRIGGER *trigger_name* COMPILE**

Suppression de triggers

- Pour supprimer un trigger de la base de données, utilisez la syntaxe DROP TRIGGER :
- **DROP TRIGGER *trigger_name***

- Exemple
- SQL> DROP TRIGGER secure_emp;
- Trigger dropped

Triggers de base de données

Création de triggers de base de données 1/2

- Avant de coder un corps de trigger, vous devez en définir les éléments constitutifs.
- Dans le cas d'un événement système, les triggers peuvent être définis au niveau du schéma ou de la base de données. Ainsi, un trigger de fermeture de base de données sera-t-il défini au niveau de la base de données. S'agissant des instructions LDD, les triggers peuvent être définis au niveau de la base de données ou du schéma.
- Cela est également valable pour la connexion et la déconnexion d'un utilisateur. En ce qui concerne les instructions LMD, les triggers sont définis sur une table ou une vue spécifique.

Création de triggers de bd (2/2)

- Un trigger défini au niveau de la base de données se déclenche pour tous les utilisateurs. Défini au niveau du schéma, ou de la table, le trigger se déclenchera uniquement si l'événement déclencheur concerne ce schéma ou cette table.
- Voici la liste des événements à la base du déclenchement d'un trigger :
- Instruction de définition de données sur un objet de la base de données ou du schéma.
- Connexion ou déconnexion d'un utilisateur spécifique ou quelconque.
- Démarrage ou fermeture d'une base de données.
- Génération d'une erreur spécifique ou quelconque.

- ***Création de triggers sur des instructions LDD***

CREATE [OR REPLACE] TRIGGER *trigger_name*
timing
[*ddl_event1* [OR *ddl_event2* OR ...]]
ON {DATABASE|SCHEMA}
trigger_body

- ***Création de triggers sur des événements système***

CREATE [OR REPLACE] TRIGGER trigger_name
timing
[database_event1 [OR database_event2 OR ...]]
ON {DATABASE|SCHEMA}
trigger_body

Trigger LOGON et LOGOFF

- Vous pouvez créer ce type de trigger pour contrôler le nombre de connexions et de déconnexions.
- Il se peut également que vous désiriez écrire un rapport faisant état de la durée de connexion. Si tel est le cas, et si vous êtes administrateur de base de données, remplacez SCHEMA par DATABASE.

```
SQL> CREATE OR REPLACE TRIGGER LOGON_TRIG
  2  AFTER logon  ON  SCHEMA
  3  BEGIN
  4    INSERT INTO log_trig_table  (user_id, log_date,
   action)
  5    VALUES (user, sysdate, 'Logging on');
  6  END;
SQL> CREATE OR REPLACE TRIGGER LOGOFF_TRIG
  2  BEFORE logoff  ON  SCHEMA
  3  BEGIN
  4    INSERT INTO  log_trig_table (user_id, log_date,
   action)
  5    VALUES (user, sysdate, 'Logging off');
  6  END;
```

Instruction CALL

- Cette instruction vous permet d'appeler une procédure stockée, plutôt que de coder le corps PL/SQL dans le trigger proprement dit.
- **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*]**
- **CALL *procedure_name***

Table en mutation :

- On qualifie de table en mutation une table qui est actuellement modifiée par une instruction UPDATE, DELETE ou INSERT, ou une table qui pourrait éventuellement être mise à jour à la suite d'une action d'intégrité référentielle déclarative DELETE CASCADE. Une table n'est pas considérée comme étant en mutation pour les triggers de niveau instruction.
- La table sur laquelle porte le trigger est elle-même une table en mutation, au même titre que toute table lui faisant référence avec la contrainte FOREIGN KEY. Cette restriction empêche un trigger ligne de visualiser un ensemble de données incohérentes.

Exemple de table en mutation

```
SQL> CREATE OR REPLACE TRIGGER check_salary
  2  BEFORE INSERT OR UPDATE OF sal, job ON emp
  3  FOR EACH ROW
  4  WHEN (new.job <> 'PRESIDENT')
  5  DECLARE
  6    v_minsalary emp.sal%TYPE;
  7    v_maxsalary emp.sal%TYPE;
  8  BEGIN
  9    SELECT MIN(sal), MAX(sal)
 10    INTO       v_minsalary, v_maxsalary
 11    FROM       emp
 12    WHERE  job = :new.job;
 13    IF :new.sal < v_minsalary OR
 14       :new.sal > v_maxsalary THEN
 15     RAISE_APPLICATION_ERROR(-20505,'Out of range');
 16    END IF;
 17  END;
 18  /
```

- SQL> UPDATE emp
- 2 SET sal = 1500
- 3 WHERE ename = 'SMITH';
- *
- ERROR at line 2:
- ORA-04091: table EMP is mutating, trigger/function
- may not see it
- ORA-06512: at "CHECK_SALARY", line 5
- ORA-04088: error during execution of trigger
- 'CHECK_SALARY'

Exemple de table en mutation

- Ce trigger, CHECK_SALARY, garantit l'adéquation du salaire d'un employé avec l'échelle de rémunération établie pour le poste à chaque fois qu'un nouvel employé est ajouté dans la table EMP ou qu'une désignation de poste ou le salaire d'un employé existant est modifié.
- Essayez de lire des données d'une table en mutation.
- Toute tentative de limitation d'un salaire entre une valeur plancher et une valeur plafond existantes génère une erreur d'exécution. La table EMP est en mutation ou dans un état de modification ; par conséquent, le trigger est incapable d'y lire des données.

Implémentation de triggers

```
SQL>CREATE OR REPLACE TRIGGER secure_emp
 2 BEFORE INSERT OR UPDATE OR DELETE ON emp
 3 DECLARE
 4   v_dummy VARCHAR2(1);
 5 BEGIN
 6   IF TO_CHAR (sysdate, 'DY' IN ('SAT','SUN'))
 7   THEN RAISE_APPLICATION_ERROR (-20506,
 8    'You may only change data during normal business
 9     hours.');
10   END IF;
11   SELECT COUNT(*) INTO v_dummy FROM holiday
12   WHERE holiday_date = TRUNC (sysdate);
13   IF v_dummy > 0 THEN RAISE_APPLICATION_ERROR (-20507,
14    'You may not change data on a holiday.');
15   END IF;
16 END;
17 /
```

audit au moyen d'un trigger

```
SQL>CREATE OR REPLACE TRIGGER audit_emp_values
 2 AFTER DELETE OR INSERT OR UPDATE ON emp
 3 FOR EACH ROW
 4 BEGIN
 5   IF audit_emp_package.g_reason IS NULL THEN
 6     RAISE_APPLICATION_ERROR (-20059, 'Specify a reason
 7     for the data operation with the procedure
 8     SET_REASON before proceeding.');
 9   ELSE
10     INSERT INTO audit_emp_table (user_name, timestamp, id,
11     old_last_name, new_last_name, old_title, new_title,
12     old_salary, new_salary, comments)
13    VALUES  (user, sysdate, :old.empno,:old.ename,
14     :new.ename, :old.job, :new.job, :old.sal,
15     :new.sal, audit_emp.package.g_reason);
16  END IF;
17 END;
18 /
```

```
SQL>CREATE TRIGGER cleanup_audit_emp
  2 AFTER INSERT OR UPDATE OR DELETE ON emp
  3 BEGIN
  4   audit_emp_package.g_reason := NULL;
  5 END;
  6 /
```

Intégrité des données

```
SQL>CREATE OR REPLACE TRIGGER check_salary
 2 BEFORE UPDATE OF sal ON emp
 3 FOR EACH ROW
 4 WHEN (new.sal < old.sal) OR
 5      (new.sal > old.sal * 1.1)
 6 BEGIN
 7   RAISE_APPLICATION_ERROR (-20508,
 8   'Do not decrease salary nor increase by
 9    more than 10%.');
10 END;
11 /
```

Intégrité référentielle

- Lors de la suppression d'un département de la table parent DEPT, définissez une suppression en cascade des lignes correspondantes dans la table enfant EMP.
-
- **SQL> ALTER TABLE emp**
- **2 ADD CONSTRAINT emp_deptno_fk**
- **3 FOREIGN KEY (deptno) REFERENCES dept(deptno)**
- **4 ON DELETE CASCADE;**

Intégrité référentielle avec un trigger

```
SQL>CREATE OR REPLACE TRIGGER
   cascade_updates
  2 AFTER UPDATE OF deptno ON dept
  3 FOR EACH ROW
  4 BEGIN
  5   UPDATE emp
  6   SET    emp.deptno = :new.deptno
  7   WHERE  emp.deptno = :old.deptno;
  8 END;
  9 /
```

Les packages

Présentation des packages

- Regroupement logique des types PL/SQL, éléments et sous-programmes
- Les packages comprennent deux parties :
  - Spécification
  - Corps
- Ils ne peuvent être appelés, paramétrés ou imbriqués
- Le package proprement dit ne peut pas être appelé, paramétré ou imbriqué. Cependant, le format d'un package est analogue à celui d'un sous-programme. Une fois écrit et compilé, le contenu peut être partagé par de nombreuses applications.
- La première fois que vous appelez une construction PL/SQL de package, l'intégralité du package est chargée en mémoire. Par conséquent, les appels ultérieurs vers ces constructions ne nécessiteront pas d'E/S disque.

Création d'une spécification de package

- Pour créer des packages, vous devez déclarer toutes les constructions publiques au sein de la spécification du package.
- Spécifiez l'option REPLACE si la spécification du package existe déjà.
- Initialisez, s'il y a lieu, une variable avec une formule ou une valeur de constante ; sinon, la variable est initialisée implicitement à NULL.
- Définition de la syntaxe
- Syntaxe
- **CREATE [OR REPLACE] PACKAGE *package_name***
- **IS|AS**
- ***public type and item declarations***
- ***subprogram specifications***
- **END *package_name;***

```
CREATE OR REPLACE PACKAGE emp_package IS
      l_comm   NUMBER  := 10;
      Cursor c_emp return emp%rowtype;
      e_valid_dept Exception;
      FUNCTION valid_dept return BOOLEAN;
    PROCEDURE reset_comm
       (v_comm   IN  NUMBER);
  END emp_package;
  /
```

Création d'un corps de package

```
SQL> CREATE OR REPLACE PACKAGE BODY emp_package
  2  IS
  3    FUNCTION  validate_comm
  4      (v_comm   IN   NUMBER)
  5      RETURN BOOLEAN
  6    IS
  7      v_max_comm    NUMBER;
  8    BEGIN
  9      SELECT   max(comm)
 10       INTO    v_max_comm
 11      FROM     emp;
 12      IF   v_comm > v_max_comm
 13      THEN   RETURN(FALSE);
 14      ELSE   RETURN(TRUE);
 15      END IF;
 16    END validate_comm;
```

```
PROCEDURE  reset_comm
 18     (v_comm   IN  NUMBER)
 19    IS
 20    BEGIN
 21     IF  validate_comm(v_comm)
 22     THEN   l_comm := v_comm;
 23     ELSE
 24        RAISE_APPLICATION_ERROR
 25              (-20210,'Invalid commission');
 26     END IF;
 27    END reset_comm;
 28  END emp_package;
 29  /
```

Exécution d'une procédure publique de package

- SQL> EXECUTE *package_name.procedure_name*(*parameters*)
- Exemple
- SQL> EXECUTE comm_package.reset_comm(15)

Variables globales

- Vous pouvez déclarer des variables publiques (globales) dont la durée de persistance correspondra à la session utilisateur. Vous pouvez également créer une spécification de package ne nécessitant pas de corps de package.
- Exemple
- Une spécification de package contenant plusieurs taux de conversion.
- Aucun corps de package n'est nécessaire pour prendre en charge cette spécification de package.

Exemple

```
SQL> CREATE OR REPLACE PACKAGE global_vars IS
  2   mile_2_kilo   CONSTANT  NUMBER  :=  1.6093;
  3   kilo_2_mile   CONSTANT  NUMBER  :=  0.6214;
  4   yard_2_meter  CONSTANT  NUMBER  :=  0.9144;
  5   meter_2_yard  CONSTANT  NUMBER  :=  1.0936;
  6  END global_vars;
  7  /
SQL> EXECUTE DBMS_OUTPUT.PUT_LINE   -
>  ('20 miles ='||20* global_vars.mile_2_kilo||' km')
20 miles =32.186 km
```

Collection

- **Ces types de données n'existent qu'en PL/SQL et n'ont pas d'équivalent dans la base Oracle**

Définition :

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 (mais en créant des collections de collections on peut obtenir des tableaux à plusieurs dimensions)
On peut distinguer trois types différents de collections :

Types de collection

- **Les tables (INDEX-BY TABLES) qui peuvent être indicées par des variables numériques ou alpha-numériques**
- **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**

- Les collections de type NESTED TABLE et VARRAY doivent-être initialisées après leur déclaration, à l'aide de leur constructeur qui porte le même nom que la collection

- Déclaration d'une collection de type nested table
  TYPE nom type IS TABLE OF type élément [**NOT NULL**] ;
  Déclaration d'une collection de type index by
  TYPE nom type IS TABLE OF type élément [**NOT NULL**]
  INDEX BY index_by_type ;
  index_by_type représente l'un des types suivants :

  BINARY_INTEGER

  PLS_INTEGER(9i)

  VARCHAR2(taille)

  LONG

```
SQL> declare
2 -- collection de type nested table
3 TYPE TYP_NES_TAB is table of varchar2(100) ;
4 -- collection de type index by
5 TYPE TYP_IND_TAB is table of number index by binary_integer ;
6 tab1 TYP_NES_TAB ;
7 tab2 TYP_IND_TAB ;
8 Begin
9 tab1 := TYP_NES_TAB('Lundi','Mardi','Mercredi','Jeudi' ) ;
10 for i in 1..10 loop
11 tab2(i):= i ;
12 end loop ;
13 End;
14 /
```

VARRAY

- **Les collections de type VARRAY**
- Ce type de collection possède une dimension maximale qui doit être précisée lors de sa déclaration
  Elle possède une longueur fixe et donc la suppression d'éléments ne permet pas de gagner de place en mémoire
  Ses éléments sont numérotés à partir de la valeur 1

  Déclaration d'une collection de type VARRAY
  TYPE nom type IS VARRAY (taille maximum) OF type élément [NOT NULL] ;

```
SQL>declare
2 -- collection de type VARRAY
3 TYPE TYP_VAR_TAB is VARRAY(30) of varchar2(100);
4 tab1 TYP_VAR_TAB := TYP_VAR_TAB('','','','','','','','','','');
5 Begin
6 for i in 1..10 loop
7 tab1(i):= to_char(i) ;
8 end loop ;
9 End;
10 /
```

```
SQL>declare
2 -- Record –
3 TYPE T_REC_EMP IS RECORD (
4 Num emp.empno%TYPE,
5 Nom emp.ename%TYPE,
6 Job emp.job%TYPE );
7 -- Table de records –
8 TYPE TAB_T_REC_EMP IS TABLE OF T_REC_EMP index by binary_integer ;
9 t_rec TAB_T_REC_EMP ;
-- variable tableau d'enregistrements
10 Begin
11 t_rec(1).Num := 1 ;
12 t_rec(1).Nom := 'Scott' ;
13 t_rec(1).job := 'GASMAN' ;
14 t_rec(2).Num := 2 ;
15 t_rec(2).Nom := 'Smith' ;
16 t_rec(2).job := 'CLERK' ;
17 End; 18 /
```

Lecture et index

La syntaxe d'accès à un élément d'une collection est la suivante :

Nom_collection(indice)

Pour une collection de type NESTED TABLE, l'indice commence par 1 .

Pour une collection de type VARRAY, l'indice doit être un nombre valide compris entre 1 et la taille maximum du tableau.

```
1 Declare
2 Type TYPE_TAB_EMP IS TABLE OF Varchar2(60)
   INDEX BY BINARY_INTEGER ;
3 emp_tab TYPE_TAB_EMP ;
4 i pls_integer ;
5 Begin
6 For i in 0..10 Loop
7 emp_tab( i+1 ) := 'Emp ' || ltrim( to_char( i ) ) ;
8 End loop ;
9 End ;
```

```
1 Declare
2 Type TYPE_TAB_JOURS IS TABLE OF PLS_INTEGER
    INDEX BY VARCHAR2(20) ;
3 jour_tab TYPE_TAB_JOURS ;
4 Begin
5 jour_tab( 'LUNDI' ) := 10 ;
6 jour_tab( 'MARDI' ) := 20 ;
7 jour_tab( 'MERCREDI' ) := 30 ;
8 End ;
/
```

Méthodes des collections :

Les méthodes sont des fonctions ou des procédures qui s'appliquent uniquement aux collections.
L'appel de ces méthodes s'effectue en préfixant le nom de la méthode par le nom de la collection

Nom_collection.nom_méthode[(paramètre, ...)]

```
If ma_collection.EXISTS(10) Then
Ma_collection.DELETE(10) ;
End if ;
Declare
nombre number(2) ;
Begin
nombre := ma_collection.COUNT ;
End ;
```

FIRST
Cette méthode retourne le plus petit indice d'une collection.
Elle retourne NULL si la collection est vide
Pour une collection de type VARRAY cette méthode retourne toujours 1
LAST
Cette méthode retourne le plus grand indice d'une collection.
Elle retourne NULL si la collection est vide
Pour une collection de type VARRAY cette méthode retourne la même valeur que la méthode COUNT
PRIOR(indice)
Cette méthode retourne l'indice de l'élément précédent l'indice donné en argument
Elle retourne NULL si indice est le premier élément de la collection

NEXT(indice)

Cette méthode retourne l'indice de l'élément suivant l'indice donné en argument
Elle retourne NULL si indice est le dernier élément de la collection

- CREATE OR REPLACE package pk is
- Type t_client is table of Client%rowtype INDEXED BY BINARY_INTEGER;
- FUNCTION listclient () return t_client;
- End pk;

- CREATE OR REPLACE PACKAGE BODY pk is
- FUNCTION listClient() return t_client
- Is
- CURSOR C_client is select * from client;
- List t_client;
- Begin
- Open c_client
- Loop
- Fetch c_client BULK COLLECT into list;
- Exit when c_client%notfound;
- End loop;
- Return list;
- End listclient;

- DECLARE
- List pk.t_client;
- BEGIN
- List:=pk.listclient();
- For i in list.first..list.last loop
- Dbms_output.put_line('N°'|| tochar(i)|| list(i).idclient||list(i).nom);
- End loop;
- En,d;

FIN