# SGBD

# PL/SQL

# PLAN

- Introduction
- Blocs PL/SQL : structure
- Variables
- Traitements conditionnels
- Traitements répétitifs
- Curseurs
- Exceptions
- Procédures et fonctions
- Triggers
- Packages

# INTRODUCTION

- Le **langage Oracle PL SQL** est :
  - un langage procédural
  - une extension procédurale du langage SQL
- Il permet de grouper des traitements et de les soumettre au noyau en un bloc unique de traitement.
- Le langage SQL est non procédural alors que le **PL SQL** est un langage procédural. Le **PL SQL** sert à programmer des procédures, des fonctions, des triggers, des packages.
- Des variables permettent l'échange d'information entre les requêtes SQL et le reste du programme

# EXEMPLE

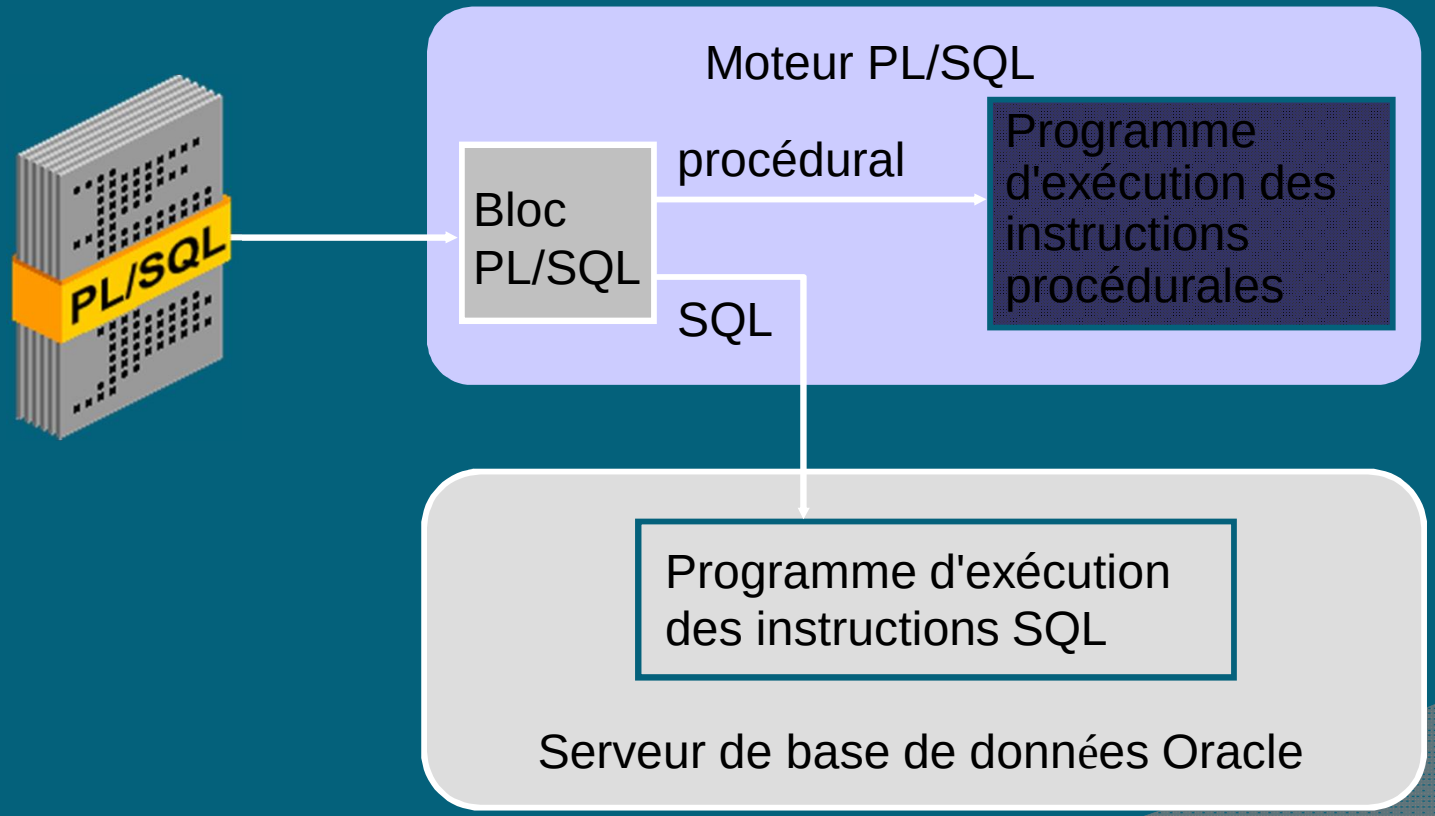
- Supposons que nous voulons accorder un bonus à chaque employé en fonction des heures effectuées sur un projet.
  - Le problème serait simplifié si vous disposiez d'instructions conditionnelles.

  → Le langage PL/SQL est conçu pour faire face à ces besoins

# PL/SQL : Structure

- Le langage PL/SQL :
  - Offre une structure de bloc pour les unités de code exécutables. La maintenance du code est facilitée avec une structure bien définie.
  - Fournit des structures procédurales telles que :
    - Variables, constantes et types
    - Structures de contrôle, telles que les instructions conditionnelles et les boucles
    - Programmes réutilisables écrits une fois et exécutés plusieurs fois

# PL/SQL : Environnement (1/2)



# PL/SQL : Environnement (2/2)

- Un bloc PL/SQL contient des instructions procédurales et des instructions SQL.
- Si on soumet un bloc PL/SQL au serveur, le moteur PL/SQL commence par analyser le bloc : Il identifie les instructions procédurales et les instructions SQL.
  - Il transmet les instructions procédurales au programme d'exécution des instructions procédurales
  - et transmet les instructions SQL au programme d'exécution des instructions SQL.
- Par conséquent, toutes les instructions procédurales sont exécutées localement et seules les instructions SQL sont exécutées dans la base de données.

# PL/SQL : Avantages (1/2)

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

# PL/SQL : Avantages (2/2)

- **Amélioration des performances :** le langage PL/SQL vous permet de combiner toutes ces instructions SQL dans un même programme. L'application peut envoyer le bloc entier à la base de données, plutôt que d'envoyer les instructions SQL une par une. Il en résulte une réduction significative du nombre d'appels de la base de données
- **Traitement des exceptions :** le langage PL/SQL permet de traiter efficacement les exceptions. On peut définir des blocs distincts pour la gestion des exceptions.

# BLOCS PL/SQL : Structure (1/6)

**La structure de base d'un programme PL/SQL est celle de bloc (possiblement imbriqué)**
La délimitation des blocs est faite avec les mots réservés:

- **DECLARE (facultatif)**
  **Variables, curseurs, exceptions définies par l'utilisateur**
- **BEGIN (obligatoire)**
  **- Instructions SQL**
  **- Instructions PL/SQL**
- **EXCEPTION (facultatif)**
  **Actions à effectuer lorsque des erreurs se produisent**
- **END; (obligatoire)**

# BLOCS PL/SQL : Structure (2/6)

- Le corps du programme (entre le BEGIN et le END) contient des instructions PL/SQL (assignements, boucles, appel de procédure) ainsi que des instructions SQL.
- Il s'agit de la seule partie qui soit obligatoire. Les deux autres zones, dites zone de déclaration et zone de gestion des exceptions sont facultatives.
- Les seuls ordres SQL que l'on peut trouver dans un bloc PL/SQL sont: SELECT, INSERT, UPDATE, DELETE.
- Les autres types d'instructions (par exemple CREATE, DROP, ALTER) ne peuvent se trouver qu'a l'extérieur d'un tel bloc. Chaque instruction se termine par un ";".
- Le PL/SQL ne se soucie pas de la casse. On peut inclure des commentaires par - - (en début de chaque ligne commentée) ou par /*.... */ (pour délimiter des blocs de commentaires).

# BLOCS PL/SQL : Structure (3/6)

- Ainsi :
  - La partie de déclaration est optionnelle, mais chaque objet utilisé doit être déclaré: Variables, Constantes, Types, Curseurs, ...
  - La partie des commandes exécutables est toujours présente. Elle est constituée de :
    - Ordres SQL
    - Manipulations des variables, constantes, et de structures de programmation (itérations, sélections, etc.).
  - La partie des exceptions est optionnelle
    - Elle gère les exceptions et les reprises d'erreurs.

# BLOCS PL/SQL : Structure (4/6)

- Premiers pas en pl/sql
  - Programme1
    - Begin
    - End;
  - Programme2
    - Begin
    - End;
    - /

Un bloc pl/sql doit de terminer par un /

```
SQL> Begin
  2  End;
  3
  4
  5
  6
  7
  8
```

Un bloc pl/sql doit contenir au moins une instruction

```
SQL> begin
  2  end;
  3  /
end;
*
ERREUR à la ligne 2 :
ORA-06550: Ligne 2, colonne 1 :
PLS-00103: Symbole "END" rencontré à la place d'un des symboles suivants :
begin case declare exit for goto if loop mod null pragma
raise return select update while with <identificateur>
<identificateur entre guillemets> <variable bind> << close
current delete fetch lock insert open rollback savepoint set
sql execute commit forall merge pipe
```

# BLOCS PL/SQL : Structure (5/6)

- Programme 3
  - Declare
    - Mot varchar2(20);
  - End;
  - /

Le bloc d'exécution Begin/End est obligatoire

```
SQL> Declare
  2  Mot varchar2(20);
  3  End;
  4  /
End;
*
ERREUR à la ligne 3 :
ORA-06550: Ligne 3, colonne 1 :
PLS-00103: Symbole "END" rencontré à la place d'un des symboles suivants :
begin function package pragma procedure subtype type use
<identificateur> <identificateur entre guillemets> form
current cursor
```

# BLOCS PL/SQL : Structure (6/6)

- Programme 4 : Affichage
  - Set serveroutput on

Permet l'activation de l'affichage sur ecran

```
SQL> set serveroutput on
SQL>
```

  - Declare
    - Chaine varchar2(10) := ' Bonjour ';
  - Begin
    - Dbms_output.put_line (chaine);
  - End;
  - /

Toute variable doit avoir été déclarée avant de pouvoir être utilisée dans la section exécutable.

```
SQL> Declare
  2  Chaine varchar2(10) := ' Bonjour ';
  3  Begin
  4  Dbms_output.put_line (chaine);
  5  End;
  6  /
 Bonjour

Procédure PL/SQL terminée avec succès.
```

# VARIABLES(1/16)

- La section déclarative
  - Syntaxe :

**nom variable [CONSTANT] type [ [NOT NULL] := expression ] ;**

- **nom variable** représente le nom de la variable composé de lettres, chiffres, $, # ou _
  Le nom de la variable ne peut pas excéder 30 caractères, doit commencer par une lettre, n'est pas sensible a la casse et doit être déclarée avant d'être utilisée.
- **CONSTANT** indique que la valeur ne pourra pas être modifiée dans le code du bloc PL/SQL
- **NOT NULL** indique que la variable ne peut pas être NULL, et dans ce cas **expression** doit être indiqué.
- **type** représente le type de la variable (→)

# VARIABLES(2/16)

- (→)
- Plusieurs types de variables sont manipulés par un programme PL/SQL :
  - **Variables PL/SQL :**
    - Types scalaires recevant une seule valeur :

      **INTEGER, REAL, STRING, DATE, BOOLEAN + %TYPE**
      + types SQL (Tous les types SQL sont utilisables en PL/SQL)
    - Types composites (%ROWTYPE, RECORD)
  - **Variables non PL/SQL** :
    - définies sous SQL*Plus (de substitution et globales), Variables hôtes (déclarées dans des programmes précompilés).

# VARIABLES(3/16)

- Exemples
  - **age integer;**
  - **nom varchar2(30);**
  - **dateNaissance date;**
  - **ok boolean := true;**

Rq: Déclarations multiples **interdites** :

~~**Exemple: i, j integer;**~~

# VARIABLES(4/16)

- Exemple

```
declare
date_naissance DATE := ‘DD/MM/YYYY’;
compteur integer default 0;
id char(5) not null ;
begin
dbms_output.put_line(date_naissance||' '||compteur||' '||id);
end;
/
```

```
ERREUR à la ligne 4 :
ORA-06550: Ligne 4, colonne 4 :
PLS-00218: une variable déclarée NOT NULL doit avoir une affectation
d'initialisation
```

# VARIABLES(5/16)

- Le type %Type
  - référence à un type existant qui est soit une colonne d'une table (ou d'une vue) soit un type défini précédemment

    **nom_variable nom_table.nom_colonne%TYPE ;**

    **nom_variable nom_variable_ref%TYPE ;**

    **Exemple1**
    **Declare**
    **idProjet projet.numproj%Type;**
    **Exemple2**
    **Declare**
    **Date1 DATE;**
    **Date2 Date1%Type;**

# VARIABLES(6/16)

- Le type %RowType
  - Ce type est composé d'un ensemble de colonnes d'un enregistrement. L'enregistrement peut contenir toutes les colonnes d'une table ou seulement certaines.
  - Il fait référence à une ligne d'une table
  - Exemple

```
Declare
Employe  personne%Rowtype;
```

  - Utilité:
    - diminue les changements à apporter au code en cas de modification des types des colonnes de la table.
    - Il est aussi possible d'insérer dans une table ou de modifier une table en utilisant une variable de type %ROWTYPE.

# VARIABLES(7/16)

- Exemple
  - Créer le type un service composé d'un enregistrement de la table service et dont tous les composants sont de même type que la table service avec la valeur du numéro accordé au service étant choisie par défaut et les autres valeurs quelconques.
  - Insérer  le service crée au niveau de la table service puis le visualiser.

```
declare
Un_service service%rowtype;
Numservice service.numserv%type:=65;
nomservice service.nomserv%type;
locservice service.locserv%type;
begin
Un_service.numserv:=NumService;
Un_service.nomserv:='comptabilité';
Un_service.locserv:= 'ariana';
insert into service values
    (Un_service.numserv,Un_service.nomserv, Un_service.locserv);
end;
/
```

# VARIABLES(8/16)

- Le type Record : permet de déclarer des structures de données personnalisées.

**TYPE *nomRecord IS RECORD***

( *nomChamp1 typeDonnées [[NOT NULL] {:= | DEFAULT} expression]*

[,*nomChamp2 typeDonnées… ]… );*

- *Exemple*

```
DECLARE
TYPE employe is record
(numemp number(4),
Nomemp varchar2(16),
postemp varchar2(12) := ‘ingénieur’);

Emp1 employe;
```

Déclaration du RECORD employe contenant trois champs ; initialisation du champ postemp par défaut à ingénieur.

Déclaration d’une variable de type employe

- Rq! il est possible qu’un champ d’un RECORD soit lui-même un RECORD, ou soit déclaré avec les directives %TYPE ou %ROWTYPE.

# VARIABLES(9/16)

- Exemple
  - Déclarer une structure « point » composée d'une abscisse et d'une ordonnée. Afficher l'abscisse et l'ordonnée d'un point de votre choix.

```
DECLARE
TYPE point IS RECORD
(
abscisse NUMBER,
ordonnee NUMBER
) ;
p point;
BEGIN
p.abscisse := 1 ;
p.ordonnee := 3 ;
DBMS_OUTPUT.PUT_LINE ( 'p.abscisse = ' || p . abscisse || ' et p .
   ordonnee =' || p . ordonnee ) ;
END;
/
```

# VARIABLES(10/16)

- Affectation
  - Il existe plusieurs façons de donner une valeur à une variable :
    - :=
    - Par la directive default
    - Par la directive INTO de la requête SELECT

  Exemples :
    - age := 25;
    - Afficher en utilisant la variable nompersonne la personne ayant le numéro 7501

# VARIABLES(11/16)

- Le type Varray

**TYPE *nomTypeTableau* IS VARRAY (taille) OF typeElements ;**
NomTableau nomTypeTableau;

- Le type Table

**TYPE *nomTypeTableau IS TABLE OF***
*{typeScalaire | variable%TYPE | table.colonne%TYPE} [NOT NULL]*
| table.%ROWTYPE [INDEX BY BINARY_INTEGER];

*nomTableau nomTypeTableau;*

- *Permet de définir et manipuler des* tableaux dynamiques (car définis sans dimension initiale).
- Un tableau est composé d'une clé primaire (de type BINARY_INTEGER) et d'une colonne (de type scalaire, TYPE, ROWTYPE ou RECORD) pour stocker chaque élément.
- La plage de valeurs du type `BINARY_INTEGER` est comprise entre -2 147 483 647 et 2 147 483 647, ce qui signifie que la valeur de la clé primaire peut être négative. L'indexation ne doit pas nécessairement commencer à 1.

# VARIABLES(12/16)

- **Fonctions pour les tableaux**
  - EXISTS(*x*) *Retourne TRUE si le x$^{ème}$ élément du tableau existe.*
  - COUNT Retourne le nombre d'éléments du tableau.
  - FIRST / LAST Retourne le premier/dernier indice du tableau (NULL si tableau vide).
  - PRIOR(x) / NEXT(*x*) *Retourne l'élément avant/après le x$^{ème}$ élément du tableau.*
  - DELETE; DELETE(*x*); DELETE(*x,y*) Supprime un ou plusieurs éléments du tableau.

# VARIABLES(13/16)

- Exemple
  - Déclarer un tableau dynamique de chaine de caractères dont les indices -3, -2, 0 contiennent des noms de personnes.
  - Afficher le premier indice, le dernier indice ainsi que le nombre d'éléments du tableau.
  - Afficher le contenu du tableau correspondant à l'indice 0.

# VARIABLES(14/16)

- Variables non pl/sql
  - Variables de substitution
    - Il est possible de passer en paramètres d'entrée d'un bloc PL/SQL des variables définies sous SQL*Plus.

**ACCEPT nom variable type variable PROMPT 'expression' ;**

    - Ces variables sont dites de substitution.
    - On accède aux valeurs d'une telle variable dans le code PL/SQL en faisant préfixer le nom de la variable du symbole « & » (avec ou sans côtes suivant qu'il s'agit d'un nombre ou pas).

# VARIABLES(15/16)

- Exemple

  Ecrire un programme qui permet de demander la saisie d'un numéro puis de l'afficher.

# VARIABLES(16/16)

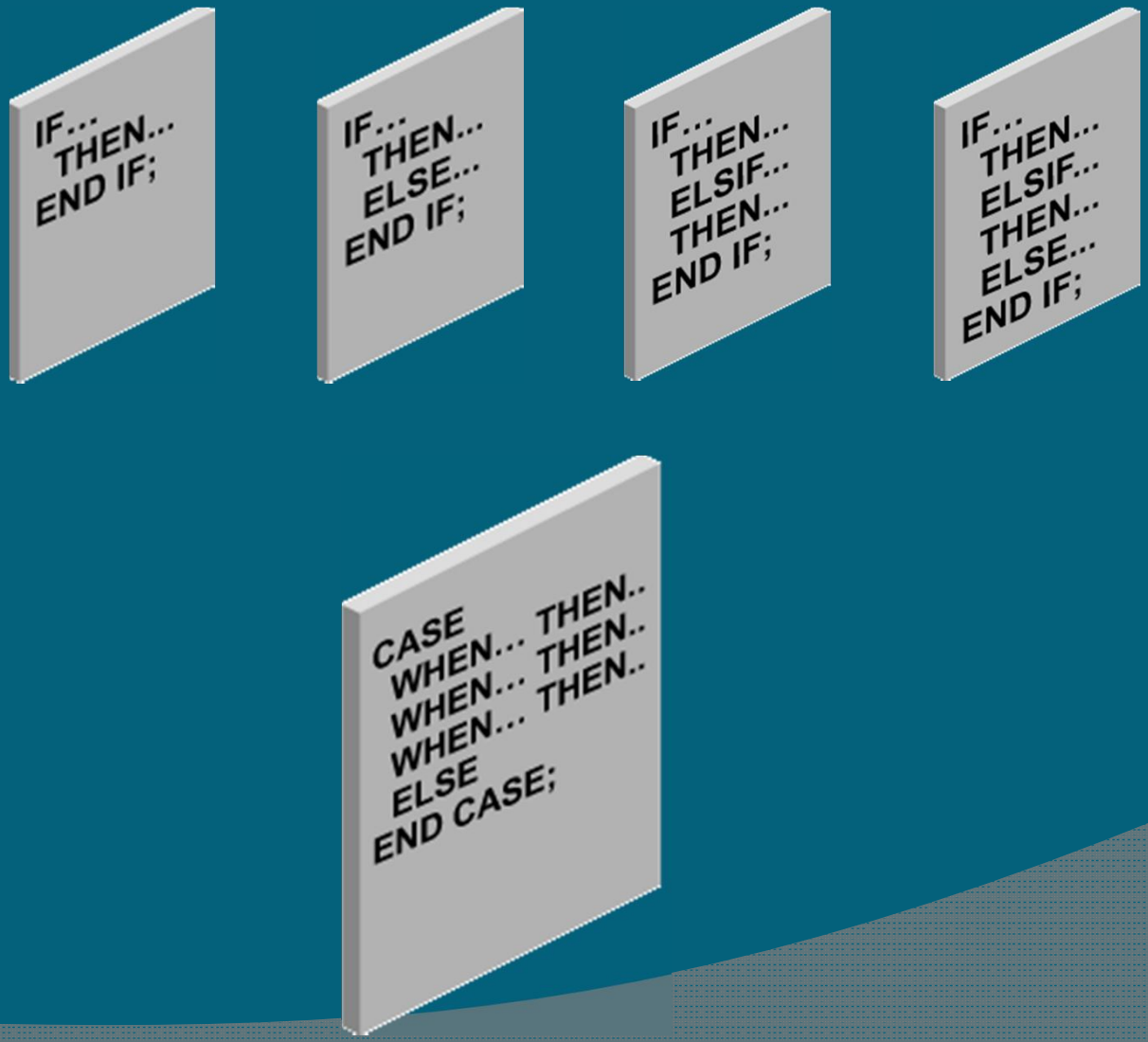
- Variables de session
  - Il est possible de définir des variables de session (globales) définies sous SQL*Plus au niveau d'un bloc PL/SQL.
  - La directive SQL*Plus à utiliser en début de bloc est VARIABLE.
  - Dans le code PL/SQL, il faut faire préfixer le nom de la variable de session du symbole « : ».
  - L'affichage de la variable sous SQL*Plus est réalisé par la directive PRINT.
- Exemple
  - Déclarer une variable globale var1
  - Affecter a cette variable la somme d'un nombre num + 5
  - Afficher cette variable

# A RETENIR

- **Les types de données les plus utilisés :**
  - **Pour les types numériques :**
    - REAL,
    - INTEGER,
    - NUMBER (précision de 38 chiffres par défaut),
    - NUMBER(x) (nombres avec x chiffres de précision),
    - NUMBER(x,y) (nombres avec x chiffres de précision dont y après la virgule).
  - **Pour les types alphanumériques :**
    - CHAR(x) (chaîne de caractère de longueur fixe x),
    - VARCHAR2 (chaîne de caractère de longueur variable)
  - **Pour les types Dates :**
    - PL/SQL permet de manipuler des dates (type DATE) sous différents formats.

- **PL/SQL supporte les opérateurs suivants :**
  - Arithmétique : +, - *, /, ** (exponentielle)
  - Concaténation : ||
  - Parenthèses (contrôle des priorités entre opérations): ()
  - Affectation: :=
  - Comparaison : =, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN
  - Logique : AND, OR, NOT

# TRAITEMENT CONDITIONNEL(1/5)



# TRAITEMENT CONDITIONNEL(2/5)

- Syntaxe de la structure IF :

**IF <condition> THEN commandes;**
**[ ELSIF <condition> THEN commandes; ]**
**[ ELSE commandes; ]**
**END IF;**

# TRAITEMENT CONDITIONNEL(3/5)

- Exemple1
  - Ecrire un programme qui permet d'afficher qu'une personne, selon son âge, est un enfant (avant 15 ans) ou un adulte (après 15 ans).
- Exemple2
  - Ecrire un programme permettant de multiplier par 2 le salaire minimal reçu s'il est inférieur à 300 et de diviser par 2 le salaire maximal reçu s'il est supérieur à 2000.
  - Afficher pour les deux personnes concernées leur noms, salaire avant et salaire après modification.

# TRAITEMENT CONDITIONNEL(4/5)

- Syntaxe de la structure CASE

```
CASE <variable>
WHEN <expr1> THEN instructions1;
WHEN <expr2> THEN instructions2;
....
[ELSE instructionsN+1;]
END CASE ;
```

# TRAITEMENT CONDITIONNEL(5/5)

- Exemple1
  - Donner en fonction de la note obtenue d'un étudiant la mention :
    - Très bien si la note est > 16
    - Bien si la note est entre 14 et 16
    - Assez bien si la note est entre 12 et 14
    - Passable si la note est entre 10 et 12
- Exemple2
  - Ecrire un programme permettant de dire pour les personnes 7501 , 7901, 7902 dans quel numéro de service elles sont affectées et combien de personnes travaillent dans ce service.

# TRAITEMENT DES VALEURS NULL

- En utilisant les valeurs NULL, certaines erreurs fréquentes doivent être évitées en tenant compte des règles suivantes :
  - Les comparaisons simples impliquant des valeurs NULL renvoient toujours une valeur NULL.
  - L'application de l'opérateur logique NOT à une valeur NULL renvoie une valeur NULL.
  - Dans les instructions de contrôle conditionnelles, si la condition renvoie une valeur NULL, la séquence d'instructions associée n'est pas exécutée.

# TABLES LOGIQUES

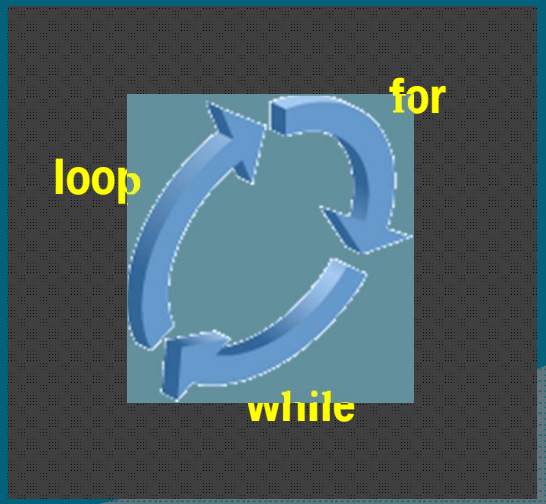
| AND | *TRUE* | *FALSE* | *NULL* |
|---|---|---|---|
| *TRUE* | TRUE | FALSE | NULL |
| *FALSE* | FALSE | FALSE | FALSE |
| *NULL* | NULL | FALSE | NULL |

| OR | *TRUE* | *FALSE* | *NULL* |
|---|---|---|---|
| *TRUE* | TRUE | TRUE | TRUE |
| *FALSE* | TRUE | FALSE | NULL |
| *NULL* | TRUE | NULL | NULL |

| NOT | |
|---|---|
| *TRUE* | FALSE |
| *FALSE* | TRUE |
| *NULL* | NULL |

# TRAITEMENT REPETITIF (1/6)

- Les boucles permettent d'exécuter plusieurs fois une instruction ou une séquence d'instructions.
- Il existe trois types de boucle :
  - Boucle de base `LOOP`
  - Boucle `WHILE`
  - Boucle `FOR`



# TRAITEMENT REPETITIF (2/6)

- La boucle loop
  - C'est une boucle potentiellement infinie.
  - Au moins une des instructions du corps de la boucle doit être une instruction de sortie.
  - Dès que la condition devient vraie (si elle le devient...), on sort de la boucle.
  - Syntaxe

```
LOOP
  commandes;
  . . .
  EXIT [WHEN condition];
END LOOP;
```

# TRAITEMENT REPETITIF (3/6)

- Exemple
  - A partir du numéro du dernier service inséré au niveau de la table service, utiliser une boucle pour insérer à l'aide d'un compteur (de valeur égale à 1) 3 nouveaux enregistrements de votre choix.

# TRAITEMENT REPETITIF (4/6)

- La boucle While
  - Elle permet la sortie selon une condition prédéfinie.
  - Elle est utilisée pour répéter des instructions tant que la condition choisie renvoie TRUE.
  - Si la condition renvoie la valeur NULL, la boucle est ignorée et l'exécution du programme reprend à l'instruction suivant la fin de la boucle.
  - **Syntaxe**

    **WHILE <condition> LOOP**

    **commandes;**

    **END LOOP;**
  - Exemple
    - Faire le même exemple que le précédent en utilisant la boucle While.

# TRAITEMENT REPETITIF (5/6)

- La boucle For
  - Ce type de boucle permet de répéter un nombre défini de fois un même traitement.
  - Cette boucle est utilisée pour simplifier le contrôle du nombre d'itérations.
  - La déclaration du compteur est implicite.
  - La syntaxe '`lower_bound .. upper_bound`' est obligatoire.
  - Syntaxe

```
FOR <compteur> IN [REVERSE]
        <limite_inf> .. <limite_sup>  loop
commandes;
END LOOP;
```

  - Exemple
    - Supprimer  les 6 dernières lignes ajoutées.

# TRAITEMENT REPETITIF (6/6)

- Règles à respecter pour la boucle FOR
  - Le compteur ne doit être référencé qu'à l'intérieur de la boucle ; il n'est pas défini en dehors.
  - Le compteur  ne doit pas être utilisé en tant que cible d'une affectation.
  - Aucune limite de boucle ne doit être `NULL`.
- Autre règles pour les boucles
  - Utilisez la boucle `LOOP` lorsque ses instructions doivent s'exécuter au moins une fois.
  - Utilisez la boucle `WHILE` si la condition doit être évaluée au début de chaque itération.
  - Utilisez une boucle `FOR` si le nombre d'itérations est connu.

# CURSEURS (1/26)

- Dans un bloc PL/SQL, le langage PL/SQL prend en charge les instructions LMD (pour extraire et modifier des données de la table de base de données ) ainsi que les commandes de gestion des transactions (commit, rollback).
- L'utilisation du résultat de la requête se fait à travers la commande INTO

# CURSEURS (2/26)

- Il faut définir autant de variables dans la clause `INTO` que de colonnes de base de données dans la clause `SELECT` en s'assurant qu'elles correspondent de manière appropriée et que les types de données sont compatibles.
- Les fonctions de groupe, telles que `SUM`, `COUNT`, … dans une instruction SQL peuvent être utilisées puisqu'elles s'appliquent à des ensembles de lignes dans une table

# CURSEURS (3/26)

- Soit la requête suivante

```
Declare
Nompers personne.nomp%type;
Salairepers personne.salairep%type;
Postepers personne.postep%type;
Begin
SELECT nomp, postep, salairep into
nompers, postepers, salairepers
from personne where numservp = 20;
 end;
/
```

Les interrogations doivent renvoyer une seule ligne

```
ERREUR à la ligne 1 :
ORA-01422: l'extraction exacte ramène plus que le nombre de lignes demandé
ORA-06512: à ligne 6
```

- Les instructions de type SELECT ... INTO ... manquent de souplesse, elles ne fonctionnent que sur des requêtes retourant une et une seule valeur.
- Ne serait-il donc pas interessant de pouvoir placer dans des variables le résultat d'une requête retournant plusieurs lignes ?

# CURSEURS (4/26)

- Etant donné que les requêtes renvoient très souvent un nombre important et non prévisible de lignes.
- On introduit, donc une notion de “curseur” pour récupérer (et exploiter) les résultats de requêtes.
- Un curseur est une zone mémoire de taille fixe, utilisée par le moteur SQL pour analyser et interpréter un ordre SQL
- Il contient le résultat d'une requête (0, 1 ou plusieurs lignes).

# CURSEURS (5/26)

- Il existe deux types de curseur :
  - Curseurs implicites : créés et gérés en interne par le serveur Oracle afin de traiter les instructions SQL
  - Curseurs explicites : déclarés explicitement par le programmeur

# CURSEURS (6/26)

- Attributs des curseurs implicites
  - Grâce aux attributs de curseur SQL, on peut tester le résultat de l'exécution des instructions SQL.

| | |
|---|---|
| **SQL%FOUND** | **Attribut booléen qui prend la valeur TRUE si la dernière instruction SQL a renvoyé au moins une ligne** |
| **SQL%NOTFOUND** | **Attribut booléen qui prend la valeur TRUE si la dernière instruction SQL n'a renvoyé aucune ligne** |
| **SQL%ROWCOUNT** | **Valeur entière qui représente le nombre de lignes affectées par l'instruction SQL la plus récente** |

# CURSEURS (7/26)

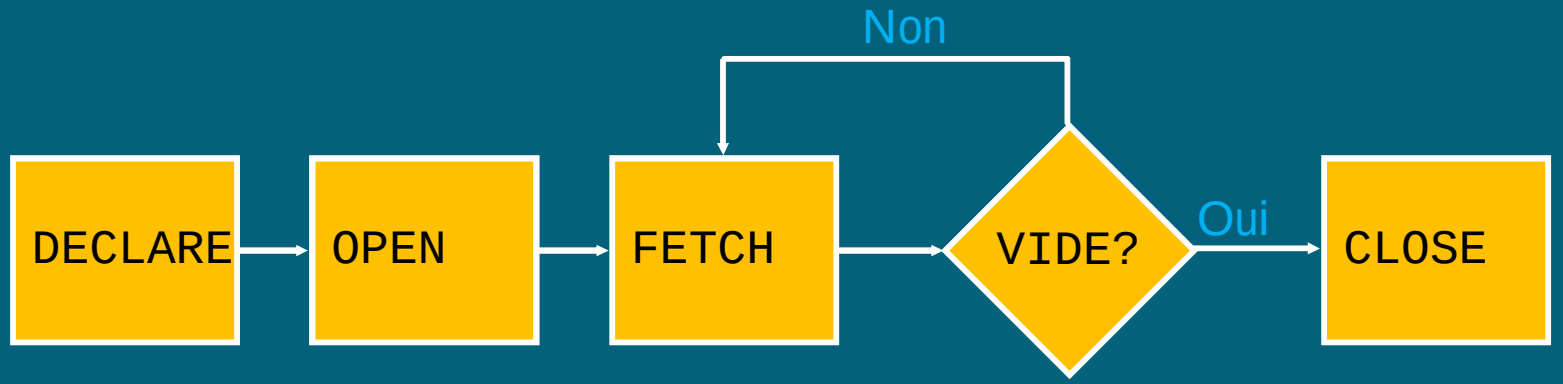
- Exemple
- Supprimer de la table `personne` les lignes avec un numéro d'employé désigné. Afficher le nombre de lignes supprimées. (Utiliser une variable de session)

# CURSEURS (8/26)

- Les curseurs explicites
  - Un curseur explicite, contrairement au curseur implicite est géré par l'utilisateur pour traiter un ordre Select qui ramène plusieurs lignes

# CURSEURS (9/26)

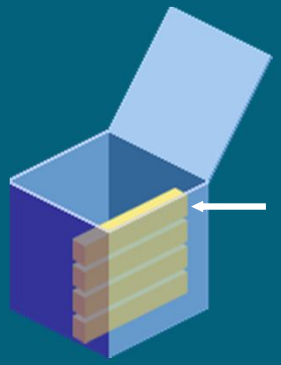
- Etapes d'utilisation des curseurs




- Déclarer le curseur dans la section déclarative d'un bloc PL/SQL en le nommant et en définissant la structure de l'interrogation à y associer.
- Ouvrez le curseur. L'instruction `OPEN` exécute l'interrogation et attache toutes les variables référencées. Les lignes identifiées par l'interrogation constituent l'ensemble actif et peuvent désormais être extraites (fetch).
- Procédez à l'extraction (fetch) des données à partir du curseur. Dans le diagramme de flux présenté dans la diapositive ci-dessus, après chaque extraction (fetch), vous testez l'existence de la ligne dans le curseur. S'il n'y a plus de lignes à traiter, vous devez fermer le curseur.
- Fermez le curseur. L'instruction `CLOSE` libère l'ensemble actif de lignes. Il est désormais possible de rouvrir le curseur pour établir un nouvel ensemble actif.

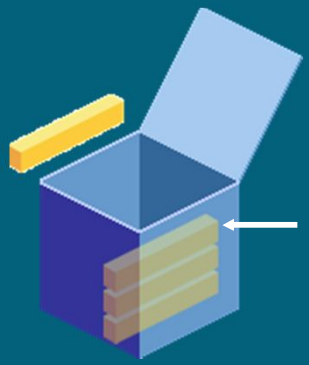
# CURSEURS (10/26)

1 Ouverture du curseur



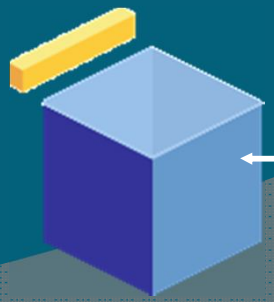
Pointeur de curseur

2 Extraction (fetch) d'une ligne



Pointeur de curseur

3 Fermeture du curseur



Pointeur de curseur

# CURSEURS (11/26)

- Syntaxe
  - Déclaration du curseur

CURSOR nomcurseur IS requête ;

  - Exemple

```
DECLARE
   CURSOR curseur_pers IS
   SELECT nump, nomp
   FROM personne
   WHERE numservp=20;

   numero  personne.nump%type;
   nom     personne.nomp%type;
```

# CURSEURS (12/26)

- Ouverture du curseur

```
Begin
   OPEN curseur_pers;
```

- Extraction des lignes

```
LOOP
   FETCH curseur_pers INTO numero, nom;
   EXIT WHEN curseur_pers %NOTFOUND;
   DBMS_OUTPUT.PUT_LINE('le numero est:' || numero
   ||' le nom est'||nom);
END LOOP;
```

- Fermeture du curseur

```
CLOSE curseur_pers;
```

# CURSEURS (13/26)

- Exemple
  - Sélectionner  l'ensemble des employés dont le salaire ne dépasse pas 400 dinars et les augmenter de 100 dinars.

# CURSEURS (14/26)

- Curseurs et boucle FOR
  - Il existe une boucle FOR se chargeant de l'ouverture, de la lecture des lignes du curseur et de sa fermeture.
  - Elle simplifie le traitement des curseurs explicites.
  - Des opérations d'ouverture, d'extraction (fetch), de sortie et de fermeture ont lieu de manière implicite.
  - L'enregistrement est déclaré implicitement.

# CURSEURS (15/26)

- Curseurs et boucle FOR
  - Syntaxe

    FOR *nom_enregistrement* IN *nom_curseur* LOOP

    *instruction1;*

    *instruction2*;

    . . .

    END LOOP;

    - Rq! Le nom de l'enregistrement est déclaré implicitement

# CURSEURS (16/26)

- Exemple
  - Sélectionner  l'ensemble des employés dont le salaire dépasse 2000 dinars et les afficher.

# CURSEURS (17/26)

- Attributs d'un curseur explicite

| Attribut | Type | Description |
|---|---|---|
| %ISOPEN | Booléen | Prend la valeur TRUE si le curseur est ouvert |
| %NOTFOUND | Booléen | Prend la valeur TRUE si la dernière extraction (fetch) ne renvoie pas de ligne |
| %FOUND | Booléen | Prend la valeur TRUE si la dernière extraction renvoie une ligne ; complément de %NOTFOUND |
| %ROWCOUNT | Nombre | Prend la valeur correspondant au nombre total de lignes renvoyées jusqu'à présent |

# CURSEURS (18/26)

- Attribut %ISOPEN
  - Les lignes ne peuvent être extraites que si le curseur est ouvert.
  - L'attribut de curseur `%ISOPEN` sert à déterminer si le curseur est ouvert.
  - `%ISOPEN` renvoie l'état du curseur à `TRUE` s'il est ouvert et `FALSE` s'il est fermé.
  - Exemple
    - On peut tester si un curseur est ouvert avant de commencer a extraire les lignes

```
IF NOT emp_cursor%ISOPEN THEN
      OPEN emp_cursor;
END IF;
LOOP
  FETCH emp_cursor...
```

# CURSEURS (19/26)

- Attribut %ROWCOUNT
  - Sert à :
    - Extraire un nombre exact de lignes
    - Extraire (fetch) les lignes avec une boucle et déterminer dans quels cas la sortie de la boucle doit s'effectuer
  - Exemple
    - Ecrire un programme pl/sql permettant d'extraire et d'afficher les 4 premiers employés appartenant au service 10 s'ils existent sinon afficher uniquement ceux qui existent.
    - Rq! Il faut faire attention à la condition de sortie de la boucle.

# CURSEURS (20/26)

- Utilisation de sous interrogation dans la boucle FOR du curseur
  - On peut ne pas déclarer un curseur dans la boucle for et utiliser à la place directement un sous interrogation
  - Syntaxe

    FOR *nom_enregistrement* IN *Requête* LOOP

    *instruction1;*

    *instruction2*;

    . . .

    END LOOP;
  - Exemple
    - Extraire et afficher  les personnes travaillant au service 20.

# CURSEURS (21/26)

- Curseurs avec paramètres
  - Permettent de transmettre des paramètres au curseur au moment de son ouverture et de l'exécution de l'interrogation.
  - Ouvrir un curseur explicite à plusieurs reprises, en renvoyant un ensemble actif différent à chaque fois.
  - Syntaxe

    CURSOR *nom_curseur* [(nom*_parametre type*, ...)]
    IS *select_statement*;

    OPEN *nom_curseur (val_parametre*,.....) ;
    instructions;
    CLOSE nom curseur;
  - Exemple
    - Extraire et afficher en fonction du numéro de service, le numéro de service et les personnes y travaillant.

# CURSEURS (22/26)

- Clause FOR UPDATE
  - Lorsque plusieurs sessions pour une même base de données sont ouvertes, les lignes d'une table particulière peuvent être mises à jour par plusieurs utilisateurs après l'ouverture du curseur.
  - Les données mises à jour ne peuvent être vues que lorsque le curseur est ouvert de nouveau.
  - Il est donc préférable de placer des verrous sur les lignes avant de les mettre à jour ou de les supprimer.
  - Le verrouillage des lignes se fait avec la clause `FOR UPDATE` dans l'interrogation du curseur.

# CURSEURS (23/26)

- Syntaxe

  SELECT ...
  FROM ...
  FOR UPDATE [OF *column_reference*] [NOWAIT | WAIT *n*];
  - Le verrouillage explicite sert à interdire l'accès aux autres sessions pendant la durée d'une transaction.
  - Le verrouillage des lignes se fait *avant* la mise à jour ou la suppression.
- L'instruction `SELECT ... FOR UPDATE` identifie les lignes qui seront mises à jour ou supprimées, puis verrouille chaque ligne de l'ensemble de résultats.
- Cela s'avère utile lorsqu'une mise à jour est basée sur les valeurs existantes d'une ligne. Dans ce cas, il faut s'assurer que cette ligne n'est pas modifiée par une autre session avant la mise à jour.

# CURSEURS (24/26)

- Le mot-clé facultatif `NOWAIT` indique au serveur Oracle de ne pas attendre si les lignes demandées soient verrouillées par un autre utilisateur.
- Le programme reprend immédiatement le contrôle ; il peut ainsi effectuer d'autres travaux avant de réessayer d'obtenir le verrouillage.
- Si vous omettez le mot-clé `NOWAIT`, le serveur Oracle attend que les lignes soient disponibles mais il peut attendre indéfiniment.
- Si les lignes sont verrouillées par une autre session et que le mot-clé `NOWAIT` est indiqué, l'ouverture du curseur provoque une erreur (il faut essayer de le rouvrir ultérieurement).
- On peut utiliser `WAIT` plutôt que `NOWAIT` et indiquer le nombre de secondes pendant lesquelles attendre et vérifier que les lignes sont déverrouillées. Si les lignes sont toujours verrouillées après *n* secondes, une erreur est renvoyée.

# CURSEURS (25/26)

- Clause WHERE CURRENT OF
  - La clause `WHERE CURRENT OF` est utilisée conjointement avec la clause `FOR UPDATE` afin de faire référence à la ligne en cours dans un curseur explicite.
  - La clause `WHERE CURRENT OF` est utilisée dans l'instruction `UPDATE` ou `DELETE`, tandis que la clause `FOR UPDATE` est définie dans la `déclaration du curseur`.
  - La clause `FOR UPDATE` peut être incluse dans l'interrogation du curseur afin que les lignes soient verrouillées lors de l'ouverture (`OPEN`).

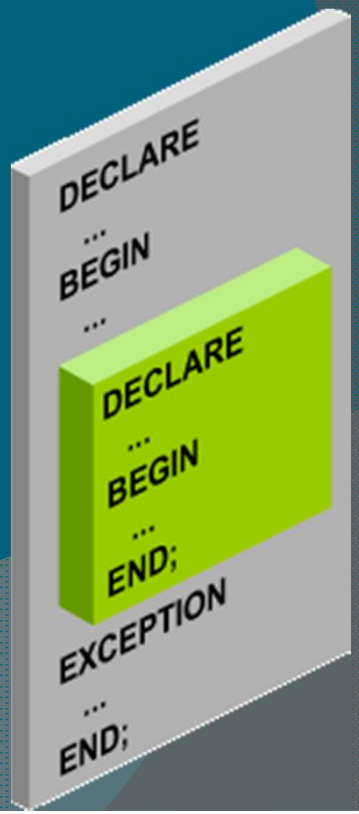
# CURSEURS (26/26)

- Exemple
  - Mettre à jour les salaires des personnes du service 10 en les augmentant de 10%.

# PORTEE DES VARIABLES

- Les blocs PL/SQL peuvent être imbriqués.
  - Une section exécutable (`BEGIN … END`) peut contenir des blocs imbriqués.
  - Une section de traitement des exceptions peut contenir des blocs imbriqués.
- Pour cela, il faut bien étudier les portées des différentes variables utilisées.
- Exemple

```
DECLARE
 var_gl VARCHAR2(20):= 'VARIABLE GLOBALE';
BEGIN
      DECLARE
       var_loc VARCHAR2(20):='VARIABLE LOCALE';
      BEGIN
       DBMS_OUTPUT.PUT_LINE(var_loc);
       DBMS_OUTPUT.PUT_LINE(var_gl);
      END;
     DBMS_OUTPUT.PUT_LINE(var_loc);
     DBMS_OUTPUT.PUT_LINE(var_gl);
    END;
    /
```



# EXCEPTIONS (1/22)

- PL/SQL permet de définir dans une zone particulière (de gestion d'exception), l'attitude que le programme doit avoir lorsque certaines erreurs définies ou prédéfinies se produisent.
- Une erreur survenue lors de l'exécution du code déclenche ce que l'on nomme une exception.
- Un certain nombre d'exceptions sont prédéfinies sous Oracle.
- Exemples :
  - NO DATA FOUND (Aucune donnée trouvée) (devient vrai dès qu'une requête renvoie un résultat vide)
  - TOO MANY ROWS (l'extraction exacte ramène plus que le nombre de lignes demandé)
  - CURSOR ALREADY OPEN (curseur déjà ouvert)
  - INVALID CURSOR (curseur invalide)...
- Le code erreur associé est transmis à la section **EXCEPTION,** pour laisser à l'utilisateur la possibilité de la gérer et donc de ne pas mettre fin prématurément à l'application.

# EXCEPTIONS (2/22)

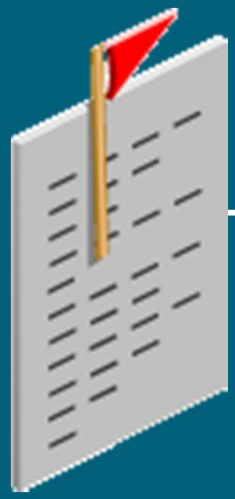
- Exemple1 : Ecrire un programme plsql qui donne le nom des personnes embauchées avant le 01/01/1990.
- Comme la requête ne ramène aucune ligne, l'exception prédéfinie NO_DATA_FOUND est générée et transmise à la section **EXCEPTION qui peut traiter le cas et poursuivre l'exécution de l'application.**

# EXCEPTIONS (3/22)

- Exemple2 :

```
declare
   nompers personne.nomp%type;
begin
    Select nomp into nompers from personne
    where datembp > TO_DATE
                          ('01/01/1995','DD/MM/YYYY');
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
  DBMS_OUTPUT.PUT_LINE ('Utilisez un curseur');
end;
 /
```

# EXCEPTIONS (4/22)



Exception générée

Exception interceptée ?

Non

Interrompre brutalement l'exécution

Propager l'exception

Oui

Exécuter les instructions de la section `EXCEPTION`

Interrompre correctement l'exécution

# EXCEPTIONS (5/22)

- Il existe 3 types d'exceptions
  - Les exception prédéfinies du serveur Oracle. Ce sont les erreurs les plus fréquentes en langage pl/SQL. Elles sont *déclenchées implicitement* .
  - Les exceptions non prédéfinies du serveur Oracle. Ce sont les erreurs standards d'oracle Server. Elles sont *déclanchées implicitement*.
  - Les exception définies par l'utilisateur . Ce sont des conditions, définies par le programmeur, comme anormales. Elles sont *déclanchées explicitement*.

# EXCEPTIONS (6/22)

- Syntaxe

```
EXCEPTION
WHEN <exception1> [OR <exception2> OR ...]
  THEN <instructions>;
WHEN <exception3> [OR <exception2> OR ...]
  THEN <instructions>;
WHEN OTHERS THEN <instructions>;
END;
```

# EXCEPTIONS (7/22)

- La section de gestion des exceptions commence avec le mot-clé EXCEPTION.
- Lorsqu'une exception se produit, un seul traitement est exécuté avant la sortie du bloc.
- La section de traitement des exceptions intercepte seulement les exceptions qui sont définies ; les autres ne sont pas interceptées sauf si la clause WHEN OTHERS est précisée.
- Cette dernière permet d'intercepter les exceptions qui n'ont pas encore été traitées.
- C'est pour cela que WHEN OTHERS doit être la dernière instruction définie.

# EXCEPTIONS (8/22)

- Quelques exceptions prédéfinies

| Exception prédéfinie | Erreur Oracle | Description |
|---|---|---|
| ACCESS_INTO_NUL | ORA-06530 | Assignation d'une valeur à un objet non initialisé |
| CURSOR_ALREADY_OPEN | ORA-06511 | Ouverture d'un curseur déjà ouvert |
| DUP_VAL_ON_INDEX | ORA-00001 | Insertion d'une ligne en doublon |
| INVALID_CURSOR | ORA-01001 | Opération interdite sur un curseur |
| INVALID_NUMBER | ORA-01722 | Echec sur une conversion d'une chaine de caractères vers un type number |
| LOGIN_DENIED | ORA-01017 | Connexion à oracle avec un utilisateur ou un mot de passe invalide |
| PROGRAM_ERROR | ORA-06501 | PL/SQL a un problème interne |
| VALUE_ERROR | ORA-06502 | Erreur d'arithmétique, de conversion, de troncature ou de limite de taille |
| ZERO_DIVIDE | ORA-01476 | Division par zero |

# EXCEPTIONS (9/22)

- Interception d'exceptions pré-définies
  - Elle se fait en utilisant le nom standard de l'exception à l'intérieur de la section.
  - Une seule exception à la fois est déclenchée et traitée.
  - Exemple : Ecrire un bloc PL/SQL permettant de sélectionner le nom d'un employé en connaissant le montant de son salaire *sal*.
    - Si le salaire entré renvoie plus d'une ligne, traiter l'exception en affichant le message « Il y a plus d'un employé avec le salaire *sal ».*
    - Si le salaire entré ne renvoie aucune ligne, traiter l'exception en affichant le message « Aucun employé avec le salaire *sal ».*
    - Toute autre exception sera affichée en utilisant le message « Autre erreur ».

# EXCEPTIONS (10/22)

- Interception d'exceptions non pré-définies
  - Une exception non pré-définie peut être interceptée soit en la déclarant préalablement soit en utilisant la commande WHEN OTHERS.
  - La déclaration de l'exception se fait dans la partie DECLARE du bloc PL/SQL en la nommant et en lui associant un code d'erreur à l'aide de la clause PRAGMA EXCEPTION_INT.
  - Ceci permet de faire référence et d'écrire un traitement spécifique pour n'importe quelle exception interne.

# EXCEPTIONS (11/22)

- Syntaxe

Nommer l'exception

Associer l'exception à un code PRAGMA

```
DECLARE
  nom_exception   EXCEPTION;
  PRAGMA  EXCEPTION_INIT(nom_exception,error_number);
    .....
BEGIN
    …
EXCEPTION
WHEN nom_exception THEN ….
END;
```

Traiter l'exception déclenchée

# EXCEPTIONS (12/22)

- Exemple : Ecrire un programme PL/SQL qui permet d'afficher le message d'erreur suivant « *Opération d'insertion impossible, contrainte not null non vérifiée* » correspondant à l'insertion d'un ligne dans la table service contenant un nom de service NULL.

# EXCEPTIONS (13/22)

- Exceptions définies par l'utilisateur
  - Les exceptions déclarées par l'utilisateur doivent être déclarées et nommées dans la partie DECLARE du bloc PL/SQL, déclenchées explicitement dans la section exécutable (BEGIN) à l'aide de l'instruction *RAISE* et traitées dans la partie EXCEPTION.

# EXCEPTIONS (14/22)

- Syntaxe

*DECLARE*
  *nom_exception*  EXCEPTION;
…
BEGIN
  …
  RAISE nom_exception;
  …
EXCEPTION
WHEN *nom_exception THEN ….*
*END;*

# EXCEPTIONS (15/22)

- Exemple : Ecrire un programme PL/SQL permettant de mettre à jour le nom d'un service et le valider en saisissant au préalable un numéro et un nom.

  Si le numéro de service saisi n'existe pas, une exception est produite et un message d'erreur s'affiche.

# EXCEPTIONS (16/22)

- Fonctions d'interception des erreurs
  - SQLCODE : Renvoie la valeur numérique associée au code de l'erreur.
  - SQLERRM : Renvoie le message associé au code de l'erreur.
  - Ces fonctions ne peuvent pas être directement utilisées dans une instruction SQL (de type INSERT ou UPDATE, …), il faut assigner leurs valeurs à des variables.

# EXCEPTIONS (17/22)

- Exemple

```
DECLARE
   Code_erreur  NUMBER;
   Msg_erreur    VARCHAR2(200);
…
BEGIN
…
EXCEPTION
…
WHEN OTHERS THEN
Rollback;
Code_erreur := SQLCODE;
Msg_erreur:= SQLERRM;
INSERT INTO table_erreurs  VALUES (Code_erreur,Msg_erreur);
END;
```

# EXCEPTIONS (18/22)

- Propagation des exceptions
  - Exemple

```
DECLARE
...
except1 exception;
except2 exception;
PRAGMA EXCEPTION_INIT (except, -2292);
BEGIN
FOR enreg IN nom_curseur LOOP
  BEGIN
    SELECT ...
    UPDATE ...
    IF SQL%NOTFOUND THEN
      RAISE except1;
    END IF;
  EXCEPTION
    WHEN except1 THEN ...
    WHEN except2 THEN ...
  END;
END LOOP;
EXCEPTION
WHEN NO_DATA_FOUND THEN . . .
WHEN TOO_MANY_ROWS THEN . . .
END;
```

- Si un sous-bloc traite une exception, il se termine normalement et l'exécution se poursuit dans le bloc supérieur.
-Si le programme produit une exception qui n'est pas traitée par le sous-bloc, elle se propage jusqu'aux blocs supérieurs.
  - Si elle trouve une fonction qui la gère, elle est traitée.
  -Sinon, le résultat donne une exception non gérée.

# EXCEPTIONS (19/22)

- Procédure raise_application_error
  - C'est une procédure qui permet de délivrer des messages d'erreurs définis par l'utilisateur.
  - Elle ne peut être appelée que durant l'exécution d'un sous-programme.
  - Syntaxe

Code erreur spécifié par l'utilisateur compris entre -20000 et -20999

Message (chaine de caracteres)défini par l'utilisateur pour l'exception

```
Raise_application_error (numero_erreur,message [,{TRUE|FALSE}]);
```

True : l'erreur est rangée dans la pile des erreurs précédentes

False(par défaut) : l'erreur remplace toutes les erreurs précédentes

# EXCEPTIONS (20/22)

- Cette procédure peut être utilisée à deux endroits:
  - Dans la section exécutable
  - Exemple

```
BEGIN
…
DELETE FROM personne WHERE nump  = &num;
IF SQL%NOTFOUND THEN
Raise_application_error(-20222, ‘employe inexistant’);
END IF;
…
```

# EXCEPTIONS (21/22)

- Dans la section exception
- Exemple

```
EXCEPTION
WHEN NO_DATA_FOUND THEN
Raise_application_error(-20222, ‘employe inexistant’);
END;
```

- Les erreurs ainsi renvoyées par la procédure Raise_application_error seront plus cohérentes par rapport aux erreurs du serveur oracle.

# EXCEPTIONS (22/22)

- Exercice
  - Ecrire un programme PL/SQL qui affiche le nombre d'employés qui gagnent 100d de plus ou de moins que le montant d'un salaire saisi préalablement.
  - S'il n'y a pas d'employés dans cette tranche de salaires, afficher un message à l'utilisateur en utilisant une exception.
  - S'il y a au moins un employé dans cette tranche, le message doit indiquer le nombre d'employés
  - Traiter toute autre exception avec un message adéquat.

# PROCEDURES ET FONCTIONS (1/14)

- Il est possible de créer des procédures et des fonctions dans PL/SQL comme dans n'importe quel langage de programmation classique.
- Les procédures et fonctions sont des blocs PL/SQL nommés, appelés sous-programmes .
- Contrairement aux blocs anonymes:
  - Ils sont Compilés une seule fois et sont stockés dans la base de données.
  - Il est possible de les appeler par d'autres applications
  - Ils peuvent accepter des paramètres et dans le cas des fonctions, doivent renvoyer des valeurs.

# PROCEDURES ET FONCTIONS (2/14)

- Toute fonction ou procédure créée devient un objet à part entière de la base (comme une table ou une vue, par exemple).
- Elle est souvent appelée “procédure ou fonction stockée”.
- Elle est donc, entre autres, sensible à la notion de droit : son créateur peut décider par exemple d’en permettre l’utilisation à d’autres utilisateurs.
- Elle est aussi appelable depuis n’importe quel bloc PL/SQL .
- Sur le même principe, on peut créer des bibliothèques de fonctions et de procédures appelées ”packages”.

# PROCEDURES ET FONCTIONS (3/14)

- PROCEDURES
  - Syntaxe

**CREATE [OR REPLACE] PROCEDURE**
**<nom_*procedure*>**
***[(argument1 [mode1] type1,***
***argument2 [mode2] type2,***
***. . .)]***
**IS**
**<zone de déclaration de variables>**
**BEGIN**
**<corps de la procédure>**
**EXCEPTION**
**<traitement des exceptions>**
**END;**

# PROCEDURES ET FONCTIONS (4/14)

- **CREATE** indique que l'on veut créer une procédure stockée dans la base
- La clause facultative **OR REPLACE** permet d'écraser une procédure existante portant le même nom
- **Nom procédure** est le nom donné par l'utilisateur à la procédure

# PROCEDURES ET FONCTIONS (5/14)

- Il y a trois **modes** pour passer les paramètres dans une procédure : *IN* (lecture seule), *OUT* (écriture seule), *INOUT* (lecture et écriture).
- Le mode *IN* est réservé au paramètres qui ne doivent pas être modifiés par la procédure.
- Le mode *OUT*, pour les paramètres transmis en résultat, le mode *INOUT* pour les variables dont la valeur peut être modifiée en sortie et consultée par la.

# PROCEDURES ET FONCTIONS (6/14)

- L'appel de la procédure se fait :
  - En utilisant l'instruction

```
call nom_procédure();
```

  - Dans un bloc PLSQL :

```
BEGIN
   nom_procédure;
END;
```

# PROCEDURES ET FONCTIONS (7/14)

- Exemple1

  Ecrire une procédure PL/SQL permettant d'extraire chaque employé et le poste qui lui est associé sous la frome « l'employé … a pour poste… ».

  Appeler cette procédure.

- Exemple2

  Ecrire une procédure PL/SQL permettant d'afficher le compte à rebours d'un nombre.

# PROCEDURES ET FONCTIONS (8/14)

- FONCTIONS
  - Syntaxe

**CREATE [OR REPLACE]** ***FUNCTION nom_fonction***
**[(argument1 [mode1]** ***type1,***
**argument2 [mode2]** ***type2, . . .)]***
**RETURN** ***type de donnée***
**IS**
**<zone de declaration de variables>**
**BEGIN**
**<corps de la fonction>**
**EXCEPTION**
**<traitement des exceptions>**
**END;**

# PROCEDURES ET FONCTIONS (9/14)

- La différence entre une procédure et une fonction est qu'une fonction doit renvoyer une valeur au programme appelant.
- L'instruction RETURN devra se trouver dans le corps pour spécifier quel résultat est renvoyé.
- La liste des arguments est facultative dans la déclaration d'une fonction.

# PROCEDURES ET FONCTIONS (10/14)

- L'appel de la fonction se fait :
  - En utilisant l'instruction

```
select nom_fonction() from dual;
```

  - Dans un bloc PLSQL :

```
BEGIN
   nom_fonction;
END;
```

# PROCEDURES ET FONCTIONS (11/14)

- Exemple1

  Ecrire une fonction nbre_emp permettant d'indiquer le nombre total d'employés.

- Exemple2

  Ecrire une fonction « minimum » permettant , à partir de deux nombres donnés a et b, d'afficher :

  - a tel que a<b (si a>b appeler a-b)

  Exécuter cette fonction.

# PROCEDURES ET FONCTIONS (12/14)

- Fonctions recevant des paramètres
  - Exemple1

    Ecrire une fonction nbre_emp _serv permettant d'indiquer le nombre d'employés pour un service donné.

    Appeler cette fonction

# PROCEDURES ET FONCTIONS (13/14)

- Exemple2

  Ecrire une fonction `verif_sal` permettant de déterminer si le salaire d'un employé donné est supérieur ou inférieur au salaire moyen de tous les employés de son département. La fonction renvoie `TRUE` si le salaire de l'employé est supérieur au salaire moyen des employés de son département ; sinon, elle renvoie `FALSE`. La fonction renvoie `NULL` si une exception `NO_DATA_FOUND` est générée.

  Appeler cette fonction pour un employé choisi

# PROCEDURES ET FONCTIONS (14/14)

- Les procédures et fonctions PL/SQL supportent assez bien la surcharge (oexistence de procédures de même nom ayant des listes de paramètres différentes). C'est le système qui, au moment de l'appel, inférera, en fonction du nombre d'arguments et de leur types, quelle est la bonne procédure à appeler.

- Les procédures et fonctions sont des objets stockés. On peut, donc, les supprimer par des instructions similaires aux instructions de suppression de tables…

**DROP PROCEDURE <nom de procedure>**
**DROP FUNCTION <nom de fonction>**

# DECLENCHEURS/TRIGGERS(1/21)

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

# DECLENCHEURS/TRIGGERS(2/21)

- Principe
  - Un trigger se lance automatiquement lorsqu'un événement se produit.
  - Par événement, on entend toute modification des données se trouvant dans les tables.
  - On s'en sert pour contrôler ou appliquer des contraintes qu'il est impossible de formuler de façon déclarative.

# DECLENCHEURS/TRIGGERS(3/21)

- Evénements déclencheurs
  - Lors de la création d'un trigger, il convient de préciser quel est le type d'événement qui le déclenche:
    - Insert
    - Delete
    - Update
    - …
- Moments d'exécution
  - Il faut également préciser si le trigger doit être exécuté avant (BEFORE) ou après (AFTER) l'événement.
- Durée
  - L'action associée à un trigger est un bloc PL/SQL enregistré dans la base.
  - Un trigger est opérationnel jusqu'à la suppression de la table à laquelle il est lié.
- Le nom du trigger doit être unique dans la base de données

# DECLENCHEURS/TRIGGERS(4/21)

- Types de triggers
  - Les triggers lignes sont exécutés **séparément** **pour chaque ligne modifiée dans la table.**
    - **Ils sont très utiles s'il faut mesurer une évolution** pour certaines valeurs, effectuer des opérations pour chaque ligne en question.
  - Les triggers de table sont exécutés **une seule fois lorsque des** modifications surviennent sur une table (même si ces modifications concernent plusieurs lignes de la table).
    - Ils sont utiles si des opérations de groupe doivent être réalisées (comme le calcul d'une moyenne, d'une somme totale, d'un compteur, …).
  - Pour des raisons de performance, il est préférable d'employer les triggers de table plutôt que les triggers lignes.

# DECLENCHEURS/TRIGGERS(5/21)

- Syntaxe

```
CREATE [OR REPLACE] TRIGGER NomTrigger
{ BEFORE | AFTER} liste_Instructions
ON nom_de_la_Table
[FOR EACH ROW]
[ WHEN condition]
BLOC PL/SQL
```

# DECLENCHEURS/TRIGGERS(6/21)

- L'option BEFORE /AFTER indique le moment du déclenchement du trigger.
- Les instructions SQL (Par exemple : INSERT OR UPDATE OR DELETE) peuvent être toutes présentes comme on peut en avoir juste une.
- Pour un UPDATE, on peut spécifier une liste de colonnes (UPDATE OF listeAttributs). Dans ce cas, le trigger ne se déclenchera que s'il porte sur l'une des colonnes précisées dans la liste.
- FOR EACH ROW est utilisée pour les triggers de niveau ligne.
- WHEN condition : le trigger est déclenché si la condition est vraie pour chaque ligne.

# DECLENCHEURS/TRIGGERS(7/21)

- Conditions de déclenchement d'un trigger
  - Le trigger se déclenche lorsqu'un événement précis survient : BEFORE UPDATE, AFTER DELETE, AFTER INSERT, …
  - Ces événements sont importants car ils définissent **le moment d'exécution du trigger.**
  - **Ainsi, lorsqu'un trigger BEFORE** INSERT est programmé, il sera exécuté *juste avant l'insertion d'un nouvel* élément dans la table.

# DECLENCHEURS/TRIGGERS(8/21)

- Si plusieurs triggers sont présents pour une même table, l'ordre d'activation est :
  1. BEFORE niveau table
  2. BEFORE niveau ligne, aussi souvent que de lignes concernées
  3. AFTER niveau ligne, aussi souvent que de lignes concernées
  4. AFTER niveau table

# DECLENCHEURS/TRIGGERS(9/21)

- Exemple1 : Trigger de table

  Ecrire un déclencheur PL/SQL permettant d'afficher un message d'erreur « Ne supprimez pas la table service » si une opération de suppression des lignes de la table service est demandée.

  Effectuer, par la suite, les instructions suivantes:

  select count(*) from service;

  delete from service

  select count(*) from service;

# DECLENCHEURS/TRIGGERS(10/21)

- Au niveau de l'instruction FOR EACH ROW, il est possible avant la modification de chaque ligne, de lire l'ancienne ligne et la nouvelle ligne par l'intermédiaire de deux variables structurées :
  - :OLD.nomAttribut : correspond à la valeur avant la transaction UPDATE ou DELETE
  - :NEW.nomAttribut : correspond à la valeur après la transaction UPDATE ou INSERT
  - *Remarque :* lorsque les variables OLD et NEW sont utilisées dans la partie WHEN condition, il ne faut pas utiliser les ":"

# DECLENCHEURS/TRIGGERS(11/21)

- Exemple de trigger sur Delete

  Ecrire un déclencheur PL/SQL «msg_supp » permettant d'afficher le message « vous avez supprimé la personne numéro …» si une opération de suppression d'une ligne dans la table personne est effectuée.

  Vérifier le déclenchement du trigger

# DECLENCHEURS/TRIGGERS(12/21)

- Exemple de trigger sur INSERT

  Ecrire un déclencheur PL/SQL «ctrl_insertion» permettant d'afficher un message d'erreur « on ne peut pas avoir un employé embauché après cette date» si une opération d'insertion d'une ligne dans la table personne contenant une date d'embauche > 01/01/2014 est effectuée.

  Insérer une ligne contenant une date > 01/01/2014 pour vérifier le déclenchement du trigger

# DECLENCHEURS/TRIGGERS(13/21)

- Exemple de trigger sur Update

  Ecrire un déclencheur PL/SQL « ctrl_maj » permettant de contrôler la mise à jour des salaires : le nouveau salaire ne doit pas être plus petit que l'ancien salaire et d'afficher le message d'erreur « Pas de baisse de salaire ».

# DECLENCHEURS/TRIGGERS(14/21)

- Déclencheur sur conditions multiples
  - Lorsqu'un trigger porte sur les opérations LMD, des prédicats peuvent être ajoutés dans le code, pour indiquer les opérations de déclenchement : Inserting, Updating, Deleting
  - Syntaxe

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

# DECLENCHEURS/TRIGGERS(15/21)

- Exemple

  Ecrire un déclencheur PL/SQL « msg_operations » permettant d'indiquer par un message pour chaque opération effectuée si c'est une opération d'insertion, de modification ou de suppression.

# DECLENCHEURS/TRIGGERS(16/21)

- Autres Déclencheurs
  - il est possible d'utiliser des déclencheurs pour suivre les changements d'état du système ainsi que les connexions/déconnexions utilisateur et la surveillance des ordres LDD et LMD.
  - Lors de l'écriture de ces déclencheurs, il est possible d'utiliser des attributs pour identifier précisément l'origine des évènements et adapter les traitements en conséquence .

# DECLENCHEURS/TRIGGERS(17/21)

- Déclencheurs sur instructions LDD
  - Syntaxe

```
CREATE TRIGGER nom_déclencheur
{BEFORE|AFTER} action
ON{DATABASE|SCHEMA}
bloc PL/SQL ;
```

**Quelques actions :**

- **Create, Alter, Drop, Rename, …**
- **Grant, Revoke  privilège(s) à un utilisateur**

**Quelques attributs:**

- **ora_database_name :** Nom de la base de données
- **ora_dict_obj_name  :** Nom de l'objet visé par l'opération DDL
- **ora_login_user :** Nom de la connexion
- **ora_dict_obj_owner : propriétaire objet affecté**

# DECLENCHEURS/TRIGGERS(18/21)

- Exemple

  Ecrire un déclencheur PL/SQL «nouvelle_table» permettant d'indiquer par un message contenant le propriétaire de la table ainsi que le nom de la table pour chaque opération de création de table.

# DECLENCHEURS/TRIGGERS(19/21)

- Déclencheurs d'instance
  - Syntaxe

    CREATE [OR REPLACE] TRIGGER nomDeclencheur
    BEFORE | AFTER evenement [OR evenement …]
    ON {SCHEMA | DATABASE}
    bloc PL/SQL;

Les événements déclencheurs :

- Démarrage ou arrêt de la base : **SHUTDOWN ou STARTUP**
- Connexion ou déconnexion d'utilisateur : **LOGON ou LOGOFF**
- Erreurs : **SERVERERROR, NO_DATA_FOUND, …**

# DECLENCHEURS/TRIGGERS(20/21)

- Un trigger peut être activé ou désactivé par les instructions:
  - ALTER TRIGGER <nom>  {ENABLE|DISABLE};
  - ALTER TABLE nomTable DISABLE ALL TRIGGERS;
  - ALTER TABLE nomTable ENABLE ALL TRIGGERS;
- et détruit par :
  - DROP TRIGGER <nom>;
- Tout déclencheur est actif dès sa compilation

  Pour Recompiler  un déclencheur après modification :
  - ALTER TRIGGER nomDeclencheur COMPILE;

# DECLENCHEURS/TRIGGERS(21/21)

- **Pour savoir quels sont les triggers crées**

```
SELECT   owner, object_name
FROM     all_objects
WHERE    object_type = ‘TRIGGER'
ORDER BY owner, object_name;
```

# PACKAGES (1/5)

- Un package est un module de programmes incluant procédures et / ou fonctions fonctionnellement dépendantes.
- Un package est composé de 2 parties :
  - La spécification (introduite par 'CREATE PACKAGE') qui liste les entêtes des procédures et fonctions contenues dans le package,
  - Le corps du package (introduit par 'CREATE PACKAGE BODY') qui contient le code effectif des procédures et fonctions déclarées précédemment.

# PACKAGES (2/5)

- Un package satisfait les points suivants :
  - Encapsulation : certains traitements sont masques, seule la spécification du package est visible. Cela a pour avantage de simplifier la tâche de celui qui va utiliser le package.
  - Modularité : il est possible de développer séparément les diverses parties de l'application. le développement devient ainsi un assemblage de package.
- Ces deux aspects fournissent une souplesse certaine au niveau du développement : il est possible de modifier le corps d'un package sans changer sa spécification, donc sans modifier le fonctionnement de l'application.

# PACKAGES (3/5)

- Syntaxe
  - Partie Spécification

```
CREATE OR REPLACE PACKAGE nompackage IS
        /*declarations*/
        procedure...
        function...return...
END nomPackage;
```

  - Partie Corps du Package

```
CREATE OR REPLACE PACKAGE BODY nompackage IS
        /* implementation */
            procedure...
            function...
END nomPackage ;
```

# PACKAGES (4/5)

- Exécution du package

  execute
  **nom_package.[nom_procedure|nom_fonction]**
  (liste_des_paramètres) ;

# PACKAGES (5/5)

- Exemple

  Ecrire un package pl/sql « ajout » pour créer :

  - une procédure « ajout_serv» permettant d'insérer une nouvelle ligne dans la table service
  - une procédure « ajout_pers» permettant d'insérer une nouvelle ligne dans la table personne

  Implémenter ce package puis l'exécuter pour des données de votre choix

  execute nom_package.nom_procedure (liste_param) ;

# EXERCICE

- Ecrire un package PLSQL «augmentation» permettant de:
  - Définir une fonction publique «calcul_augmentation» qui reçoit le numéro d'une personne et le pourcentage d'augmentation accordé et qui renvoie le montant de l'augmentation.
  - Définir une procédure publique «nouveau_salaire » qui, à partir, du numéro de personne entré, nous donne son nouveau salaire augmenté (Utiliser une variable globale).
  - Définir une procédure privée «affiche_salaire» permettant d'afficher tous les salaires sous la forme Employé nom_employé ayant le numéro num --> salaire
  - La mise à jour de la table n'est pas demandée.
  - Tenir compte d'éventuelles exceptions
  - Afficher, si demandé, le nouveau salaire.

# EXERCICE (Suite)

- Reprendre l'exercice en effectuant une mise à jour du salaire augmenté dans la table personne.
- Programmer un déclencheur vous empêchant de faire une mise à jour de plus du double du salaire.