IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE
1-INTRODUCTION

1- OBJECTIF

-> Compléter les faiblesses du modèle relationnelle :

- Les contraintes d'intégrité NON relationnelles : hors des dépendances fonctionnelles

- Le traitement individualisé

- La communication entre les unités de traitement

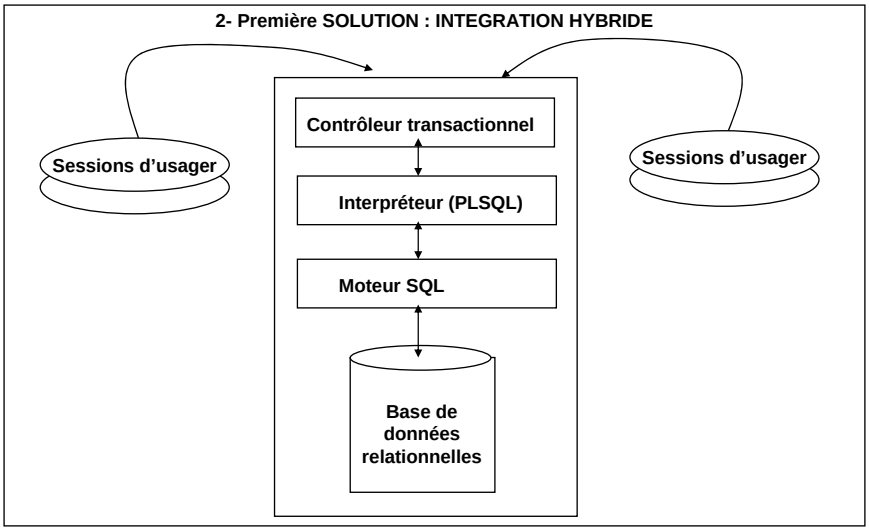
- La réutilisation des composants logiciels et de données entre programmes

- L'ouverture vers des types de données et des traitements externes de la base de données

- Le développement des applications complexes (CAO, GPAO, ...)

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE
1-INTRODUCTION

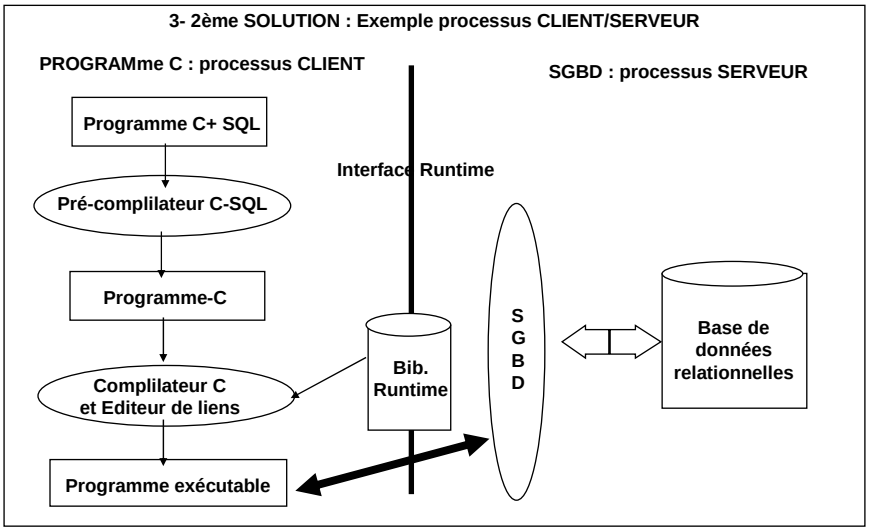
2- Première SOLUTION : INTEGRATION HYBRIDE



- 3 -

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE
1-INTRODUCTION

3- 2ème SOLUTION : Exemple processus CLIENT/SERVEUR



- 4 -

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

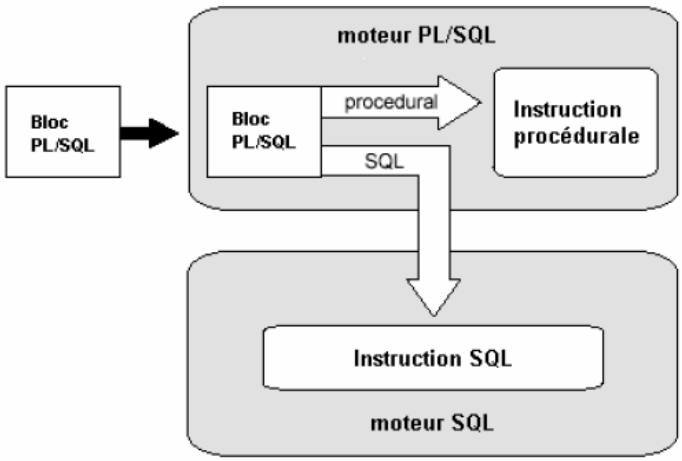
PRINCIPAUX POINTS ABORDÉS

- **Bloc PL SQL**
  - **introduction : bloc anonyme, bloc nommé**
  - **Section déclarative : types, variable, expression, assignation**
  - **Section d'exécution : instructions procédurales**
  - **Section d'exécution : instructions transactionnelles**
  - **Section d'exécution : instructions non procédurales**
  - **Section d'exécution : instructions mixtes – curseurs explicites et implicites**
  - **Section de traitement d'erreur: définie et manipulation des exceptions**
  - **Section de traitement d'erreur: définie et manipulation des exceptions**
- **Fonctions, procédures et paquetages**
  - **fonctions de typage : chaînes, arithmétiques, conversion, dates**
  - **Fonction et procédures déclaratives**
  - **paquetage**
- **Déclencheurs**
  - **Déclencheurs sur les tables**
  - **Déclencheurs sur les vues**
  - **Déclencheurs sur les évènement**
  - **Maintenance des déclencheurs**

- 5 -

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL
1- Bloc PL SQL
1- INTRODUCTION : Principe d'exécution d'un bloc PLSQL



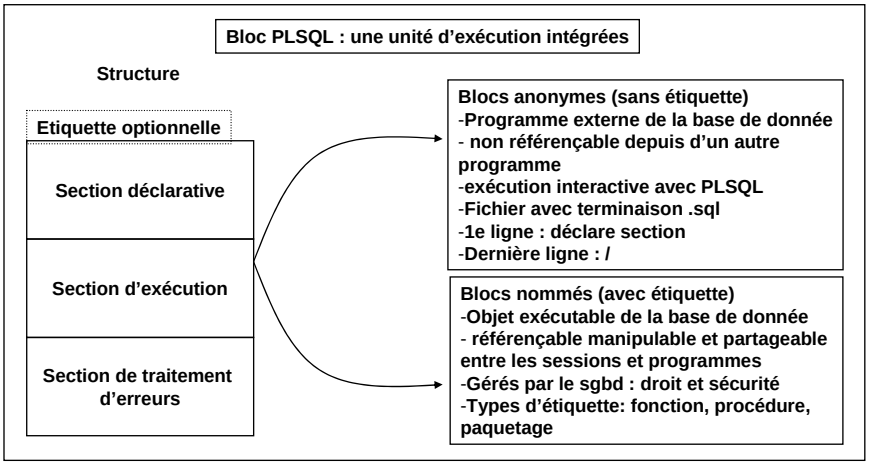
- 6 -

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL

1. INTRODUCTION : Structure, bloc anonyme, bloc nommé



- 7 -

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL

1. INTRODUCTION : Structure, bloc anonyme, bloc nommé

- Chaque bloc PL/SQL peut être constitué de 3 sections :
  - **Une section facultative de déclaration et initialisation de types, variables et constantes.**
    - » Cette section ne peut pas contenir d'instructions exécutables. Toutefois, il est possible de définir dans cette section des procédures ou des fonctions contenant une section exécutable.
    - » Toute variable doit avoir été déclarée avant de pouvoir être utilisée dans la section exécutable.
  - **Une section obligatoire contenant les instructions d'exécution**
  - **Une section facultative de gestion des erreurs qui** débute par le mot clé EXCEPTION
    - » Elle contient le code exécutable mis en place pour la gestion des erreurs
    - » Lorsqu'une erreur intervient dans l'exécution, le programme est stoppé et le code erreur est transmis à cette section

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

- Un bloc PL/SQL minimum peut être représenté de la façon suivante :
  BEGIN
  Null ;
  END ;
  - Le mot clé **BEGIN** détermine le début de la section des instructions exécutables
  - Le mot clé **END;** indique la fin de la section des instructions exécutables
  - Une seule instruction figure dans ce bloc : **Null;** qui ne génère aucune action

- 8 -

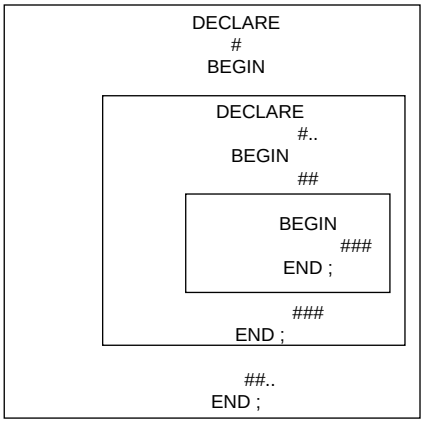
IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL

1. INTRODUCTION : Structure, bloc anonyme, bloc nommé

- **Les blocs PL/SQL peuvent être imbriqués les uns dans les autres**



- 9 -

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL

2. Section déclarative : types, variable, littéral, expression

Variables

- **Syntaxe**

  var 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
  - **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.
- **Note :** PL/SQL n'est pas sensible à la casse.

- 10 -

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
2. Section déclarative : types, variable, littéral, expression

Variables (2)

- **Catégories de variables**
  - **Variables locales** : les variables définies dans le bloc PLSQL courant
  - **Variables externes** : les variables définies dans une autres unités d'ORACLE (SQLPLUS, FORMS, TRIGGER, REPORT, PRO-C,…), et utilisées dans le bloc PLSQL : elles doit-être préfixées par le caractère # ou $ (le cas de SQLPLUS)
  - **Variables bases de données** : les variables représentant les objets dans la bases de données en particulier les attributs. Elles peuvent être utilisées directement. Dans le cas de conflit de nom avec une variable locale, il faut préfixer la variable d'attribut par le nom de relation dans laquelle il appartient

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
2. Section déclarative : types, variable, littéral, expression

Variables (3)

- **Affectation**
  - L'initialisation ou l'affectation, s'effectue avec l'opérateur **:=**

```
DECLARE
  LN$Nbre NUMBER(3) := 0 ;
  LD$Date DATE := SYSDATE ;
  LC$Nom VARCHAR2(10) := 'PL/SQL' ;
```

- **Constante**
  - Une constante est une variable dont l'initialisation est obligatoire et dont la valeur ne pourra pas être modifiée en cours d'exécution
  - Elle est déclarée avec le mot clé : CONSTANT qui doit précéder le type

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
2. Section déclarative : types, variable, littéral, expression

Types

- **Types scalaires**

Types scalaires				
BINARY_INTEGER	DEC	DECIMAL	DOUBLE PRECISION	FLOAT
INT	INTEGER	NATURAL	NATURALN	**NUMBER**
NUMERIC	**PLS_INTEGER**	POSITIVE	POSITIVEN	REAL
SIGNTYPE	SMALLINT	**CHAR**	CHARACTER	**LONG**
LONG RAW	**NCHAR**	**NVARCHAR2**	**RAW**	**ROWID**
STRING	**UROWID**	**VARCHAR**	**VARCHAR2**	
BOOLEAN	**DATE**			
INTERVAL DAY TO SECOND(9i)	**INTERVAL YEAR TO MONTH(9i)**	**TIMESTAMP(9i)**	**TIMESTAMP WITH LOCAL TIME ZONE(9i)**	**TIMESTAMP WITH TIME ZONE(9i)**

Note : dans le premier tableau, **les types Oracle sont en gras**, les sous-types compatible ANSI/ISO en normal

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
2. Section déclarative : types, variable, littéral, expression

Types (2)

- **Types composée et types complexe**

Types composés		
RECORD	TABLE	VARRAY

Types références	
REF CURSOR	REF type_objet

Types grands objets			
BFILE	BLOB	CLOB	NCLOB

Types suplémentaires				
SYS.ANYDATA	SYS.ANYTYPE	SYS.ANYDATASET		
XMLTYPE	URITYPE			
MDSYS.SDO_GEOMETRY				
ORDSYS.ORDAUDIO	ORDSYS.ORDIMAGE	ORDSYS.ORDVIDEO	ORDSYS.ORDDOC	ORDSYS.ORDIMAGESIGNATUI

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
2. Section déclarative : types, variable, littéral, expression

Types (3)

- **Sous-types et types dérivés**
  - **SUBTYPE nom_sous-type IS type ;**
    - » Exemple 1: SUBTYPE entier_court IS SMALLINT ;
    - » Exemple 2: i entier_court ;
  - **%TYPE**
    - » référence à un type existant qui est soit une colonne d'une table soit un type défini précédemment
    - » **Modèle 1 : nom_variable nom_table.nom_colonne%TYPE ;**
    - » **Modèle 2 : nom_variable nom_variable_ref%TYPE ;**
  - **ROWTYPE**
    - » référence à une ligne d'une table ou d'un curseur
    - » **Modèle 1 : nom_variable nom_table%ROWTYPE ;**
    - » **Modèle2 : nom_variable nom_curseur%ROWTYPE ;**

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
2. Section déclarative : types, variable, littéral, expression

Types (4)

- **Sous-types et types dérivés**
  - **Exemples :**

```
Declare
-- variable de même type que le colonne ENAME de la table EMP
LC$Nom EMP.ENAME%TYPE ;
-- variable de même type qu'une ligne de la table EMP
LR$EMP EMP%ROWTYPE ;
LC$Dat1 DATE ;
-- variable de même type que LC$Dat1 (DATE)
LC$Dat2 LC$Dat1%TYPE ;
-- Curseur --
Cursor C_EMP is
   Select empno, ename, job From EMP ;
-- variable de type ligne du curseur C_EMP
LR$C_emp C_EMP%ROWTYPE ;
```

Types (5)

- **Types prédéfinis (chaînes)**
  - **CHAR[(n)] :** Chaîne de caractères de longueur fixe avec n compris entre 1 et 32767 (par défaut 1)
  - **VARCHAR2(n) :** Chaîne de caractères de longueur variable avec n compris entre 1 et 32767
  - **Note :** Ces types PL/SQL ont une capacité supérieure à celle des colonnes de tables de même type. Une colonne CHAR ne peut excéder 2000 caractères et une colonne de type VARCHAR2 4000 caractères.
  - **LONG :** Chaîne de caractères de longueur variable avec au maximum 32760 octets
  - **RAW[(n)] :** Chaîne de caractères ou données binaires de longueur variable avec n compris entre 1 et 32767. Le contenu d'une variable de ce type n'est pas interprété par PL/SQL (pas de gestion des caractères nationaux)
  - **LONG RAW :** Identique au type LONG qui peut contenir des données binaires

Types (6)

- **Types prédéfinis (chaînes)**
  - **NCHAR[(n)] :** Chaîne de caractères de longueur fixe avec n compris entre 1 et 32767 (par défaut 1)
  - **NVARCHAR2[(n)] :** Chaîne de caractères de longueur variable avec n compris entre 1 et 32767
    - » **Note :** Le nombre de caractères réellement stockés dépend du nombre d'octets utilisés pour coder chaque caractère
  - **UROWID, ROWID :** Permet de stocker l'adresse absolue d'une ligne dans une table sous la forme d'une chaîne de caractères
    - » Le format d'une telle variable est le suivant : 000000FFFBBBBBBRRR
      - 000000 représente le numéro de l'objet qui possède cette ligne (dans le cas de cluster, plusieurs objets peuvent partager le même segment)
      - FFF représente le numéro du fichier qui contient la ligne
      - BBBBBB représente le numéro du bloc dans le fichier
      - RRR représente le numéro de ligne dans le bloc

Types (7)

- **Types prédéfinis (numériques)**
  - **NUMBER[(e,d)] :** Nombre réel avec e chiffres significatifs stockés et d décimales
  - **BINARY_INTEGER :** Nombre entier compris entre -2 147 483 647 et +2 147 483 647 (Utilise les fonctions de la librairie arithmétique)
  - **(10g)BINARY_FLOAT :** Nombre à virgule flottante simple précision au format IEEE 754 un littéral de ce type est écrit avec un f terminateur (ex. 3.125f)
  - **(10g)BINARY_DOUBLE :** Nombre à virgule flottante double précision au format IEEE 754 un littéral de ce type est écrit avec un d terminateur (ex. 3.12548d)
  - **PLS_INTEGER :** Nombre entier compris entre -2 147 483 647 et +2 147 483 647 (Plus rapide que BINARY_INTEGER car il utilise les registres du processeur)

Types (8)

- **Types prédéfinis (grand volume)**
  - **BFILE :** Stocke la référence vers un fichier du système d'exploitation
  - **BLOB :** Permet de stocker un objet binaire jusqu'à 4 Go
  - **CLOB :** Permet de stocker un ensemble de caractères, jusqu'à 4 Go
  - **NCLOB :** Permet de stocker un ensemble de caractères, codés sur un ou plusieurs octets, jusqu'à 4 Go
- **Types prédéfinis (objets spécifiques ANYDATA)**
  - **SYS.ANYTYPE, SYS.ANYDATA :** Une variable de ce type peut contenir un objet de n'importe quel type scalaire ou objet
    - » Définie comme colonne d'une table, elle pourrait contenir une variable de type NUMBER dans une ligne, une variable de type VARCHAR2 dans une autre, une variable de type objet dans une troisième, etc.
    - » Il faut utiliser les méthodes associées pour insérer la valeur correspondant au type désiré sur chaque ligne

Types (9)

- **Types prédéfinis (objets spécifiques ANYDATA)**
  - **SYS.ANYDATA.CONVERT...**
    - » SYS.ANYDATA.ConvertNumber(1500) pour insérer une variable numérique
    - » SYS.ANYDATA.ConvertVarchar2('Hello') pour insérer une variable caractère
  - **Liste des fonctions de conversion**
    - » ConvertNumber(num IN NUMBER) RETURN AnyData
    - » ConvertDate(dat IN DATE) RETURN AnyData
    - » ConvertChar(c IN CHAR) RETURN AnyData
    - » ConvertVarchar(c IN VARCHAR) RETURN AnyData
    - » ConvertVarchar2(c IN VARCHAR2) RETURN AnyData
    - » ConvertRaw(r IN RAW) RETURN AnyData
    - » ConvertBlob(b IN BLOB) RETURN AnyData
    - » ConvertClob(c IN CLOB) RETURN AnyData
    - » ConvertBfile(b IN BFILE) RETURN AnyData

Types (10)

- **Types prédéfinis (objets spécifiques ANYDATA)**
  - **Méthodes nom_variable.GET... pour retrouver les valeurs insérées :**
    - » GetNumber(self IN AnyData, num OUT NOCOPY NUMBER) RETURN PLS_INTEGER
    - » GetDate(self IN AnyData, dat OUT NOCOPY DATE) RETURN PLS_INTEGER
  - **Liste des fonctions de conversion**
    - » ConvertObject(obj IN "(object_type)") RETURN AnyData
    - » ConvertRef(rf IN REF "(object_type)") RETURN AnyData
    - » ConvertCollection(col IN "(COLLECTION_1)") RETURN AnyData
    - » GetChar(self IN AnyData, c OUT NOCOPY CHAR) RETURN PLS_INTEGER
    - » GetVarchar(self IN AnyData, c OUT NOCOPY VARCHAR) RETURN PLS_INTEGER
    - » GetVarchar2(self IN AnyData, c OUT NOCOPY VARCHAR2) RETURN PLS_INTEGER
    - » GetRaw(self IN AnyData, r OUT NOCOPY RAW) RETURN PLS_INTEGER
    - » GetBlob(self IN AnyData, b OUT NOCOPY BLOB) RETURN PLS_INTEGER

Types (11)

- **Types prédéfinis (objets spécifiques ANYDATA)**
  - **Méthodes nom_variable.GET... pour retrouver les valeurs insérées :**
    - » GetClob(self IN AnyData, c OUT NOCOPY CLOB) RETURN PLS_INTEGER
    - » GetBfile(self IN AnyData, b OUT NOCOPY BFILE) RETURN PLS_INTEGER
    - » GetObject(self IN AnyData, obj OUT NOCOPY "(object_type)") RETURN PLS_INTEGER
    - » GetRef(self IN AnyData, rf OUT NOCOPY REF "(object_type)") RETURN PLS_INTEGER
    - » GetCollection(self IN AnyData, col OUT NOCOPY "(collection_type)") RETURN PLS_INTEGER

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
2. Section déclarative : types, variable, littéral, expression

Types (12)

- **Types prédéfinis (objets spécifiques ANYDATA)**
  - **Le type ANYDATA supporte également les méthodes suivantes:**
    - » Procédure BEGINCREATE pour la création d'un nouveau type
    - » Procédure membre PIECEWISE pour définir le mode d'accès à la valeur courante
    - » Procédure membre SET... Pour positionner les valeurs
    - » Procédure membre ENDCREATE Pour terminer la création d'un nouveau type
    - » Fonction membre GETTYPENAME Pour retrouver la définition complète du type
    - » Fonction membre GETTYPE Pour retrouver le type de l'objet

Types (13)

- **Types prédéfinis (objets spécifiques ANYDATA)**
  - **SYS.ANYDATASET :** Ce type contient à la fois la description et un ensemble de données de même type.
  - **Liste des fonctions attachées à ce type**
  - · Procédure membre ADDINSTANCE Pour l'ajout d'une nouvelle instance de données
  - · Procédure BEGINCREATE pour la création d'un nouveau type
  - · Procédure membre PIECEWISE pour définir le mode d'accès à la valeur courante
  - · Procédure membre SET... Pour positionner les valeurs
  - · Procédure membre ENDCREATE Pour terminer la création d'un nouveau type
  - · Fonction membre GETTYPENAME Pour retrouver la définition complète du type
  - · Fonction membre GETTYPE Pour retrouver le type de l'objet
  - · Fonction membre GETINSTANCE Pour retrouver l'instance suivante
  - · Fonctions membre GET... Pour retrouver les valeurs
  - · Fonction membre GETCOUNT Pour retrouver le nombre d'instances du type

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
2. Section déclarative : types, variable, littéral, expression

Types (14)

- **Types prédéfinis (objets spécifiques Types XML)**
  - Ces types sont utilisés pour stocker des objets XML. Le type XMLTYPE possède des fonctions membres pour insérer, extraire et interroger les données XML via les expressions de type XPATH
  - Pour manipuler les données XML, Oracle met à disposition les fonctions
    - » XMLAGG
    - » XMLCOLATTVAL
    - » XMLCONCAT
    - » XMLDATA
    - » XMLELEMENT
    - » XMLFOREST
    - » XMLSEQUENCE
    - » XMLTRANSFORM

Types (15)

- **Types prédéfinis (objets spécifiques Types XML)**
  - Ainsi que les paquetages spécifiques pour manipuler les types XML
    - » DBMS_XMLDOM
    - » DBMS_XMLGEN
    - » DBMS_XMLPARSER
    - » DBMS_XMLQUERY
    - » DBMS_XMLSAVE
    - » DBMS_XMLSCHEMA
  - Les types URI (URITYPE, DBURITYPE, XDBURITYPE et HTTPURITYPE) permettent de gérer les données sous forme d'URL.
    - » Pour manipuler les données URI, Oracle met à disposition le paquetage URIFACTORY

Types (16)

- **Types prédéfinis (objets spécifiques Géométriques et média)**
  - **MDSYS.SDO_GEOMETRY :** Pour la manipulation d'objets Oracle Spatial
  - **Les types URI** (URITYPE, DBURITYPE, XDBURITYPE et HTTPURITYPE) permettent de gérer les données sous forme d'URL.
    - » Pour manipuler les données URI, Oracle met à disposition le paquetage URIFACTORY
  - **Les types médias** sont utilisés pour stocker des objets multi-média avec Oracle interMedia
    - » **ORDSYS.ORDAUDIO :** Pour le stockage de données audio
    - » **ORDSYS.ORDIMAGE :** Pour le stockage des images
    - » **ORDSYS.ORDIMAGESIGNATURE :** Pour le stockage des propriétés des images
    - » **ORDSYS.ORDVIDEO :** Pour le stockage des données vidéo
    - » **ORDSYS.ORDDOC :** Pour le stockage de tout type de données multi-média

Types (17)

- **Types et sous-types définis par l'utilisateur** avec le mot clé **TYPE** ou **SUBTYPE**
  - **SUBTYPE nom_sous-type IS type_base[(précision)] [NOT NULL]**
    - » **nom_sous-type** représente le nom du sous-type déclaré
    - » **type_base** représente le nom du type prédéfini
    - » **précision** représente une longueur pour les caractères et longueur + décimales pour les numériques
  - **Exemples :**
    - » **SUBTYPE chaine_courte IS VARCHAR2(10);**
      - Le sous-type utilisateur chaine_courte définit un VARCHAR2(10)
      - à la suite de cette définition, toute déclaration de variable de type chaine_courte sera égale à VARCHAR2(10)
    - » **SUBTYPE NOM_EMP IS EMP.ename%Type;**
      - le sous-type NOM_EMP définit un type identique à la colonne ename de la table EMP
    - » **SUBTYPE REC_EMP IS EMP%ROWTYPE;**
      - le sous-type REC_EMP définit un type identique à une ligne de la table EMP

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
2. Section déclarative : types, variable, littéral, expression

Littéraux et expressions

- **Les littéraux** : Un littéral ou valeur constante désigne une valeur fixe.
  - **Exemples :**
    - » 'LUNDI', 'Montpellier', '2012' représentent des valeurs littérales de type caractère
    - » 12.3, 25 représentent des valeurs littérales de type numérique
    - » Ces valeurs peuvent apparaître dans des initialisations de variables, des calculs ou transmises à des procédures ou fonctions.
  - **Littéral de type caractère :** Désigne une valeur fixe comme étant de type caractère
    - » La valeur peut contenir n'importe quel caractère à l'exception d'une simple apostrophe
    - » **Note :** pour saisir une apostrophe dans un littéral, il faut la doubler ('')
    - » Il doit être encadré d'une paire d'apostrophes
    - » Il peut être précédé du caractère N pour indiquer qu'il doit être transformé dans le jeu de caractères national
    - » Il a les mêmes propriétés que les types CHAR et VARCHAR2
    - » Sa longueur ne peut pas dépasser 4000 octets

Littéraux et expressions (2)

- **Les littéraux** :.
  - **Littéral de type entier :** Désigne une valeur fixe comme étant de type caractère
    - » Désigne une valeur fixe comme étant de type entier
    - » Ne peut contenir que les chiffres de 0 à 9
    - » Il peut être précédé des signes + ou -
    - » Il peut contenir jusqu'à 38 chiffres de précision
  - **Littéral de type décimal :**
    - » Désigne une valeur fixe comme étant de type numérique
    - » Ne peut contenir que les chiffres de 0 à 9
    - » Il peut être précédé des signes + ou -
    - » Il peut contenir jusqu'à 38 chiffres de précision
    - » Il peut contenir le caractère e ou E qui indique que la valeur est spécifiée en notation scientifique. Les chiffres après le E indiquent l'exposant. Ce dernier est valide dans un intervalle de -130 à 125

Littéraux et expressions (3)

- **Les littéraux** :.
  - **Littéral de type intervalle :** Désigne une valeur fixe comme étant de type caractère
    - » Spécifie une période de temps, déclinée en années et mois ou en jours, heures, minutes et secondes.
    - » Les deux types de littéraux de type intervalle sont YEAR TO MONTH et DAY TO SECOND
    - » Chaque type contient un préfixe et peut contenir un suffixe. Le préfixe désigne l'unité de base de date ou d'heure.
    - » Le suffixe définit les parties d'incrément associées à l'unité de base.
    - » Note : Si vos données sont sous forme numérique, vous pouvez utiliser les fonctions de conversion
      - NUMTOYMINTERVAL ou NUMTODSINTERVAL pour les convertir en littéraux de type intervalle.

Littéraux et expressions (4)

- **Les littéraux** :
  - **Littéral de type intervalle de type YEAR TO MONTH :**
    - » **Syntaxe**

      INTERVAL 'nombre_entier [-nombre_entier]' YEAR ou MONTH (précision) TO YEAR ou MONTH
      - **Nombre_entier [-nombre_entier]** spécifie une valeur entière pour le préfixe et éventuellement le suffixe du littéral. Si le préfixe est YEAR et le suffixe est MONTH, nombre_entier pour le mois doit être entre 0 et 11
      - **Précision** représente le nombre maximum de chiffres pour le préfixe compris entre 0 et 9. Par défaut sa valeur est 2
    - » **Exemples**
      - INTERVAL '12-3' YEAR TO MONTH : intervalle de 12 ans et 3 mois
      - INTERVAL '115' YEAR(3) : intervalle de 115 ans (la précision du suffixe doit être spécifiée YEAR(3) si elle est supérieure à la valeur par défaut)
      - INTERVAL '24' MONTH : intervalle de 24 mois
    - » **Opérations :** additionner ou soustraire un littéral de type intervalle à un autre
      - INTERVAL '6-4' YEAR TO MONTH - INTERVAL '6' MONTH

Littéraux et expressions (5)

- **Les littéraux** :
  - **Littéral de type intervalle de type DAY TO SECOND :**
    - » **Syntaxe**

      INTERVAL 'nombre_entier' DAY ou HOUR ou MINUTE ou SECOND (précision) TO DAY ou HOUR ou MINUTE ou SECOND (fractions de secondes)
      - Nombre_entier peut représenter
        - soit : Un nombre de jours
        - soit : Une heure au format HH[:MI[:SS[.fractions_de_secondes]]]
      - **Précision** représente le nombre maximum de chiffres pour le préfixe compris entre 0 et 9. Par défaut sa valeur est 2
      - Fractions_de_secondes représente le nombre de chiffres des fractions de secondes, compris entre 1 et 9. Par défaut sa valeur est 6
      - Les valeurs correctes pour les champs sont :
        - HOUR 0 à 23
        - MINUTE 0 à 59
        - SECOND 0 à 59.999999999

Littéraux et expressions (7)

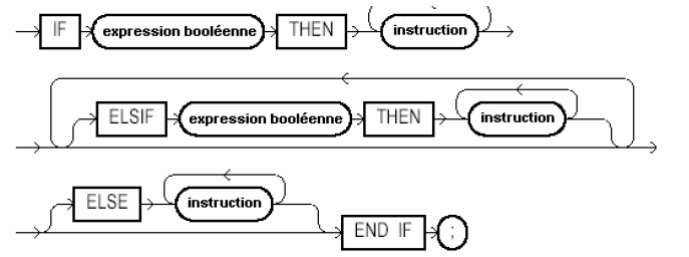
- **Les littéraux** :
  - **Littéral de type intervalle de type DAY TO SECOND :**
    - » **Exemples**
      - INTERVAL '6 4 :10 :22.356' DAY TO SECOND(3) : intervalle de 6 jours, 4 heures, 10 minutes, 22 secondes et 356 millièmes de secondes
      - INTERVAL '6 4 :10' DAY TO MINUTE : intervalle de 6 jours, 4 heures et 10 minutes
      - INTERVAL '365 12' DAY(3) TO HOUR : intervalle de 365 jours et 12 heures
      - INTERVAL '8 :10 :20.3333333' HOUR TO SECOND(7) : Intervalle de 8 heures, 10 minutes, 20.3333333 secondes
      - INTERVAL '18 :30' HOUR TO MINUTE : intervalle de 18 heures et 30 minutes
      - INTERVAL '20' MINUTE : intervalle de 20 minutes
      - INTERVAL '4.12345' SECOND(2,4) : intervalle arrondi à 4.1235 secondes car la précision demandée sur les

      fractions de secondes est de 4 chiffres
    - » **Opérations :** additionner ou soustraire un littéral de type intervalle à un autre
      - INTERVAL '30' DAY - INTERVAL '18' HOUR

Affectation

- L'assignation d'une valeur à une variable peut être faite de 2 façons différentes
  - **Affectation par l'opérateur :=** :
  - **Exemple :**
    - » a_variable := 10 ;
    - » Ma_chaine := 'Chaîne de caractères' ;
  - **Affectation par la clause INTO :**
    - » SELECT … INTO ….
    - » FETCH …. INTO ….
  - **Exemple :**
    - » Fetch C_EMP Into LC$Nom_emp ;
    - » Select C_NOM Into unNOM ;

Instructions conditionnelles (IF)



Note
• **expression booléenne** représente un test générant un booléen TRUE ou FALSE
•Seuls les mots clé **IF** et **END IF;** sont obligatoires. Les clauses **ELSIF** et **ELSE** sont facultatives

Instructions conditionnelles (CASE)

- **Cette instruction permet de mettre en place des structures de test conditionnel de type IF .. ELSE .. END IF, à la grande différence qu'elle est utilisable dans les requêtes SQL**

- **Deux syntaxes sont possibles**
  - **CASE simple**
    - » **[<<label>>] CASE opérateur { WHEN contenu_opérateur THEN { instruction;} ... }... [ELSE { instruction;}...] END CASE [label];**
  - **CASE de recherche**
    - » **[<<label>>] CASE { WHEN expression_booléenne THEN { instruction;} ... }... [ELSE { instruction;}...] END CASE [label];**

Instructions conditionnelles (CASE) (2)

- **Notes :**
  - opérateur peut être n'importe quel type PL/SQL à l'exception des objets suivants : BLOB, BFILE, Type objet, Enregistrement, Collection (NESTED TABLE, INDEX-BY TABLE, VARRAY)
  - Pour le CASE simple, chaque mot clé **WHEN** vérifie l'égalité entre opérateur et contenu_opérateur. Dans l'affirmative, l'instruction suivant le mot clé **THEN** est exécutée, puis la structure **CASE** est quittée et l'exécution du programme est reprise après le mot clé **END CASE;**
  - Pour le CASE de recherche, l'omission de la clause ELSE provoque une erreur

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
3. Section d'exécution : instructions procédurales

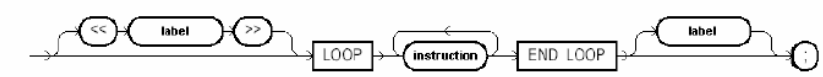
- **Exemple : CASE simple**

```
SQL> Declare
2 LN$Num pls_integer := 0 ;
3 Begin
4 Loop
5 LN$Num := LN$Num + 1 ;
6 CASE LN$Num
7 WHEN 1 Then dbms_output.put_line( '1' ) ;
8 WHEN 2 Then dbms_output.put_line( '2' ) ;
9 WHEN 3 Then dbms_output.put_line( '3' ) ;
10 ELSE
11 EXIT ;
12 END CASE ;
13 End loop ;
14 End ;
15 /
123
Procédure PL/SQL terminée avec succès.
```

- **Exemple : CASE recherche**

```
SQL> Declare
2 LN$Num pls_integer := 0 ;
3 Begin
4 Loop
5 LN$Num := LN$Num + 1 ;
6 CASE
7 WHEN LN$Num between 1 and 3 Then dbms_output.put_line(
        To_char( LN$Num ) || ' -> 1-3'
) ;
8 WHEN LN$Num < 5 Then dbms_output.put_line( To_char(
        LN$Num ) || ' < 5' ) ;
9 ELSE dbms_output.put_line( To_char( LN$Num ) || ' >= 5' ) ;
10 END CASE ;
11 exit when LN$Num = 5 ;
12 End loop ;
13 End ;
14 /
1 -> 1-3
2 -> 1-3
3 -> 1-3
4 < 5
5 >= 5
Procédure PL/SQL terminée avec succès.
```

Instructions répétitives : boucle sans étiquette

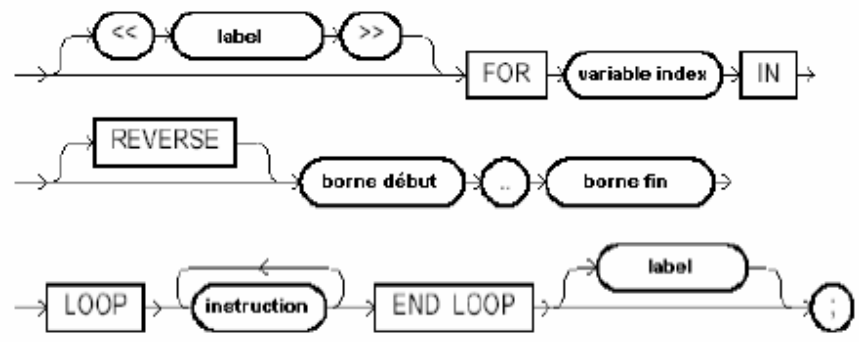


LOOP
instruction;[instruction;[...]]
END LOOP;

• Cette syntaxe met en place une boucle simple ou aucune condition de sortie n'est indiquée
• Il faut donc une instruction **EXIT** pour sortir de ce type de boucle

```
SQL> Declare
2 LN$I pls_integer := 0 ;
3 Begin
4 Loop
5 LN$I := LN$I + 1 ;
6 dbms_output.put_line( to_char( LN$I) ) ;
7 exit when LN$I > 2 ;
8 End loop ;
9 End ;
10 /
123
```


Instructions répétitives : boucle avec étiquette « for »



Instructions répétitives : boucle avec étiquette « for »

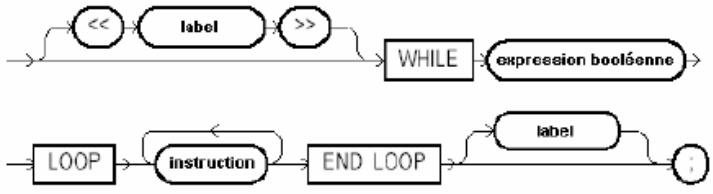
FOR variable index IN [REVERSE] borne_début..borne_fin
LOOP instruction;[instruction;[...]]
END LOOP;

• Cette syntaxe permet de mettre en place une boucle dont le nombre d'itérations est fixé dès l'entrée

• **Variable index** représente le nom de la variable qui servira d'indice. Cette variable ne nécessite pas de définition préalable dans la section déclarative
• **Reverse** permet de faire varier l'indice dans le sens contraire (décrémentation)
•**borne début** représente l'indice de départ
• **borne fin** représente l'indice de fin

```
SQL> Declare
2 LN$I pls_integer := 0 ;
3 Begin
4 For i in 1..3 Loop
6 dbms_output.put_line( to_char( i ) ) ;
7 End loop ;
8 End ;
9 /
123
SQL> Declare
2 LN$I pls_integer := 0 ;
3 Begin
4 For i in reverse 1..3 Loop
6 dbms_output.put_line( to_char( i ) ) ;
7 End loop ;
8 End ;
9 /
321
```


Instructions répétitives : boucle avec étiquette « while »

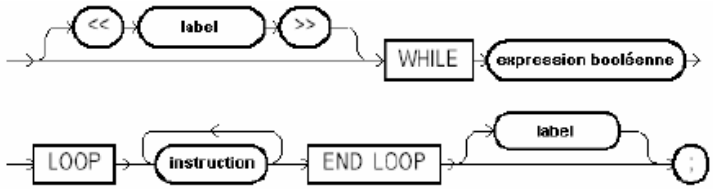


WHILE expression booléenne LOOP instruction;[instruction;[...]] END LOOP;

• Cette syntaxe permet de mettre en place une boucle dont la condition de test est évaluée au début.
• Si expression booléenne donne le résultat FALSE, les instructions suivantes jusqu'au mot clé **END LOOP;** ne seront pas exécutées

```
SQL> Declare
2 LN$I pls_integer := 0 ;
3 Begin
4 For i in 1..3 Loop
6 dbms_output.put_line( to_char( i ) ) ;
7 End loop ;
8 End ;
9 /
123
```

Instructions répétitives : boucle avec étiquette « while »



WHILE expression booléenne LOOP instruction;[instruction;[...]] END LOOP;

• Cette syntaxe permet de mettre en place une boucle dont la condition de test est évaluée au début.
• Si expression booléenne donne le résultat FALSE, les instructions suivantes jusqu'au mot clé **END LOOP;** ne seront pas exécutées

```
SQL> Declare
2 LN$I pls_integer := 0 ;
3 Begin
4 For i in 1..3 Loop
6 dbms_output.put_line( to_char( i ) ) ;
7 End loop ;
8 End ;
9 /
123
```


Instructions branchement : étiquette et goto

Etiquette

GOTO

• Cette instruction permet d'exécuter un saut dans le code vers le label précisé
• Une instruction valide doit suivre la déclaration du label

```
2 LN$I pls_integer := 0 ;
3 LN$J pls_integer := 0 ;
4 Begin
5 Loop
6 LN$I := LN$I + 1 ;
7 Loop
8 LN$J := LN$J + 1 ;
9 dbms_output.put_line( to_char( LN$I ) || ',' ||
to_char( LN$J ) ) ;
```

```
10 If LN$J > 3 Then GOTO sortie ; End if ;
11 End loop ;
12 End loop ;
13 <<sortie>>
14 null ;
15 End ;
16 /
1,1  1,2  1,3   1,4
Procédure PL/SQL terminée avec succès.
```

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
3. Section d'exécution : instructions procédurales

Instructions branchement : exit

• Cette instruction permet de quitter une structure itérative

• **label** facultatif permet de nommer précisément la structure dont on veut sortir.
• **expression booléenne** permet de spécifier une condition de sortie
• Exit saute à l'instruction suivant le mot clé **END LOOP;**
• Dans le cas de boucles imbriquées, l'indication d'un label permet de quitter tout ou partie des boucles imbriquées

```
SQL> Declare
2 LN$Num pls_integer := 0 ;
3 Begin
4 Loop
5 LN$Num := LN$Num + 1 ;
6 dbms_output.put_line( to_char( LN$Num ) ) ;
7 EXIT WHEN LN$Num > 3 ;
-- sortie de la boucle lorsque LN$Num est supétieur à 3
8 End loop ;
9 End ;
10 /
1234
Procédure PL/SQL terminée avec succès.
```


IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
3. Section d'exécution : instructions transactionnelles

Commit

•Cette instruction permet d'enregistrer en base toutes les modifications effectuées au cours de la transaction en cours

• Le mot clé **WORK** est facultatif et n'a aucun effet particulier

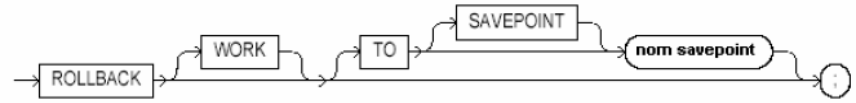
• Un commentaire d'un maximum de 50 caractères peut apparaître entre apostrophes derrière le mot clé **COMMENT**

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
3. Section d'exécution : instructions transactionnelles

Rollback et Savepoint



• **Cette instruction permet d'annuler en base toutes les modifications effectuées au cours de la transaction**
• **Savepoint** permet au traitement d'annuler, avec l'instruction **ROLLBACK**, les modifications effectuées à partir de cette étiquette
•**nom savepoint** représente le nom d'une étiquette savepoint préalablement définie dans le corps du code avec l'instruction **SAVEPOINT**
• Avec **TO SAVEPOINT** nom savepoint, l'annulation porte sur toutes les modifications effectuées à partir de l'étiquette nom savepoint

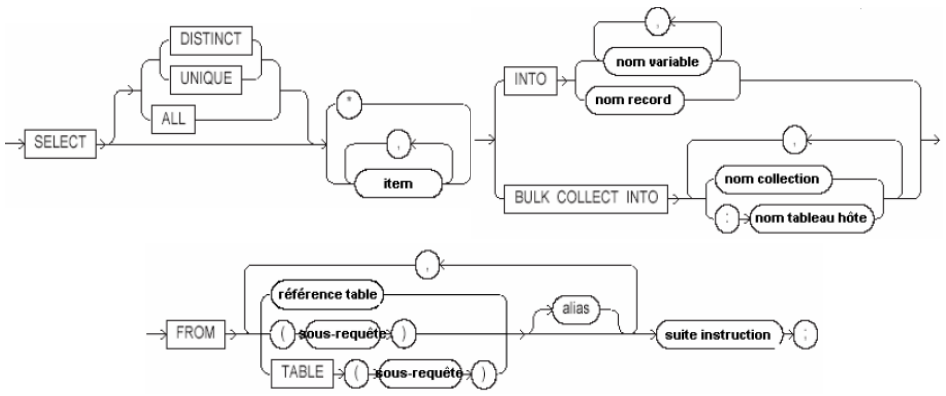
```
SQL> Begin
2 Insert Into EMP( empno, ename, job )
3 values( 9991, 'Dupontont', 'CLERK' ) ;
4 Insert Into EMP( empno, ename, job )
5 values( 9992, 'Duboudin', 'CLERK' ) ;
6 SAVEPOINT mise_a_jour ;
7 Update EMP Set sal = 2500 Where empno >
9990 ;
10 ROLLBACK TO SAVEPOINT mise_a_jour ;
11
12 Commit ;
13 End ;
14 /
```


IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
3. Section d'exécution : instructions non procédurales

SELECT … INTO



IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
3. Section d'exécution : instructions non procédurales

SELECT … INTO (2)

- Cette instruction permet d'exécuter un ordre Select implicite.
- Cet ordre ne doit ramener qu'une ligne sous peine de générer l'exception NO_DATA_FOUND si aucune ligne n'est ramenée ou TOO_MANY_ROWS si plus d'une ligne sont ramenées
- Utilisée avec la clause **BULK COLLECT**, elle permet de charger une collection avec les lignes ramenées

- Exemple

```
SQL> Declare
2 LN$Num EMP.empno%Type ;
3 LC$Nom EMP.ename%Type ;
4 LC$Job EMP.job%Type ;
5 Begin
6 Select empno,ename,job
10 Into LN$Num,LC$Nom,LC$Job
14 From EMP
16 Where empno = 7369 ;
19 End ;
20 /
Procédure PL/SQL terminée avec succès.
```

- 51 -

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
3. Section d'exécution : instructions non procédurales

SELECT … INTO (3)

- **item** représente un littérale ou un nom de colonne
- **nom variable** représente le nom d'une variable d'accueil. Son type doit être identique à celui de item
- **nom record** représente le nom d'un enregistrement composé de champs de même type que les items ramenés
- **nom collection** représente le nom d'une collection d'accueil
- **nom tableau hôte** représente le nom de la variable tableau passée par un programme tiers
- **référence table** représente la liste des tables et/ou vues de l'ordre SQL
- **sous-requête** représente le texte d'une sous-requête SQL conforme
- **suite instruction** représente la suite de l'ordre Select (clauses Where, Group by, Order by, etc.)

- Exemple

```
SQL> Declare
2 LR$Emp EMP%Rowtype ;
3 Begin
4 Select  *
6 Into LR$Emp
8 From EMP
10 Where empno = 7369;
13 End ;
14 /
Procédure PL/SQL terminée avec
    succès.
```

- 52 -

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
3. Section d'exécution : instructions non procédurales

Mise à jour de la BD

```
Insert
 into nom_relation
Values (nouveau tuple);
```

```
Delete
 from nom_relation
Where -- Condition ;
```

```
Update relation
(Nom_att=expression_valeur)*
Where -- Condition ;
```

- Toute instruction SQL valide est exécutable
- Pas de commande sur les structures : Create / Drop / Alter

- 53 -

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

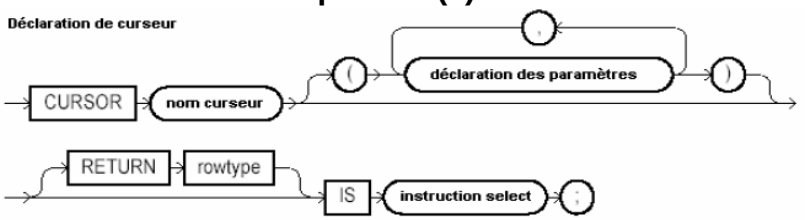
1- Bloc PL SQL
3. Section d'exécution : instructions mixtes

Curseurs explicites

- Un curseur est une zone mémoire de taille fixe, utilisée par le moteur SQL pour analyser et interpréter un ordre SQL
- Un curseur explicite, contrairement au curseur implicite (SELECT INTO) est géré par l'utilisateur pour traiter un ordre Select qui ramène plusieurs lignes
- Tout curseur explicite géré dans la section exécution doit avoir été déclaré dans la section déclarative
- Un curseur nommé C_EMP est déclaré avec l'ordre Select correspondant (CURSOR C_EMP IS...)
- Il est ouvert avec l'instruction **OPEN,** lu avec l'instruction **FETCH** et fermé avec l'instruction **CLOSE**
- Un curseur est paramétrable. On peut donc utiliser le même curseur pour obtenir différents résultats

- 54 -

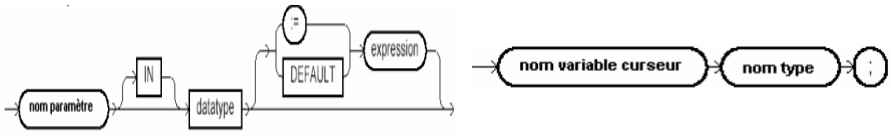
Curseurs explicites (2) : déclaration



- **nom curseur** représente le nom du curseur que l'on déclare
- **déclaration des paramètres**(facultatif) représente la liste des paramètres transmis au curseur
- **instruction select** représente l'ordre SQL Select d'alimentation du curseur

```
Cursor C  is
Select V.v#, P.nom, V.A# , P.nom
 from vol V, pilote P
 where P.pl#=V.pl# and A.a#=V.v#
and VD='Paris' and VA='Nice'
;
```


Curseurs explicites (3) : déclaration des paramètres



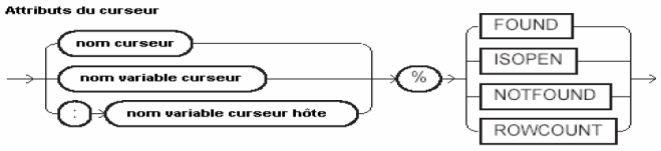
- nom paramètre représente le nom de la variable paramètre
- datatype représente le type SQL de la variable paramètre (doit correspondre en type avec la colonne visée)
- expression représente la valeur par défaut du paramètre (doit correspondre en type avec celui du paramètre)

```
SQL> Declare
CURSOR C_EMP ( PN$Num IN
    EMP.empno%Type )IS
Select  empno, ename, job
From EMP
Where empno = PN$Num ;
```

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
3. Section d'exécution : instructions mixtes

Curseurs explicites (4) : Attributs d'un curseur



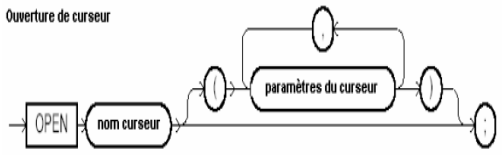
Chaque curseur dispose de 4 attributs

- **%FOUND :** Cet attribut prend la valeur TRUE lorsque une ligne est ramenée, sinon il prend la valeur FALSE
- **%NOTFOUND :** Cet attribut prend la valeur FALSE lorsque une ligne est ramenée, sinon il prend la valeur TRUE
- **%ISOPEN :** Cet attribut prend la valeur TRUE lorsque le curseur indiqué est ouvert, sinon il prend la valeur FALSE
- **%ROWCOUNT :** Cet attribut retourne le nombre de lignes impactées par la dernière instruction SQL

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
3. Section d' exécution : instructions mixtes

Curseurs explicites : OPEN (5)

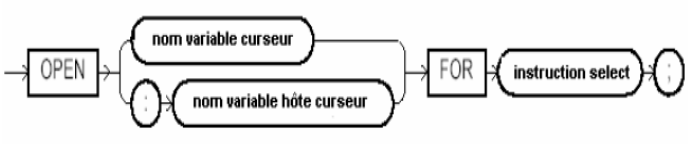



```
Begin
Open C_EMP ( 1024 );
Fetch C_EMP Into LC$Nom_emp ;
Close C_EMP ;
End ;
```

- **nom curseur** représente le nom donné au curseur qui permettra de le référencer dans les instructions suivantes
- **paramètres du curseur** représente la liste des paramètres transmis au curseur
- **le curseur doit avoir été préalablement défini dans la section déclarative (paramétrable comme dans l'exemple)**

- Declare
- LC$Nom_emp EMP.ENAME%Type ;
- Cursor C_EMP (LN$Numemp IN EMP.EMPNO%Type) Is
- Select ename
- From EMP
- Where Empno = LN$Numemp;

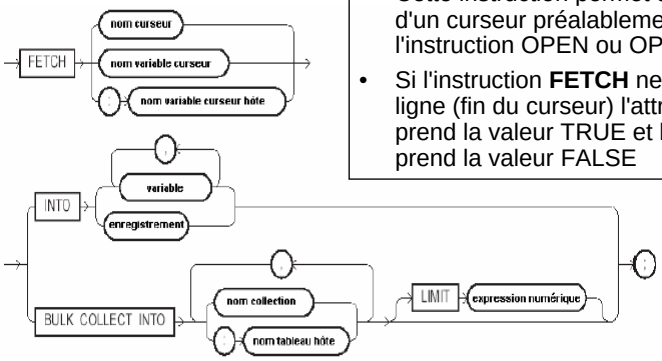
Curseurs explicites (6)



OPEN FOR permet de déclarer le curseur au moment de l'ouverture

```
Declare
LC$Nom_emp EMP.ENAME%Type ;
Begin
Open C_EMP For 'Select ename From EMP Where empno = 1024' ;
Fetch C_EMP Into LC$Nom_emp ;
Close C_EMP ;
End ;
```


Curseurs explicites (7)



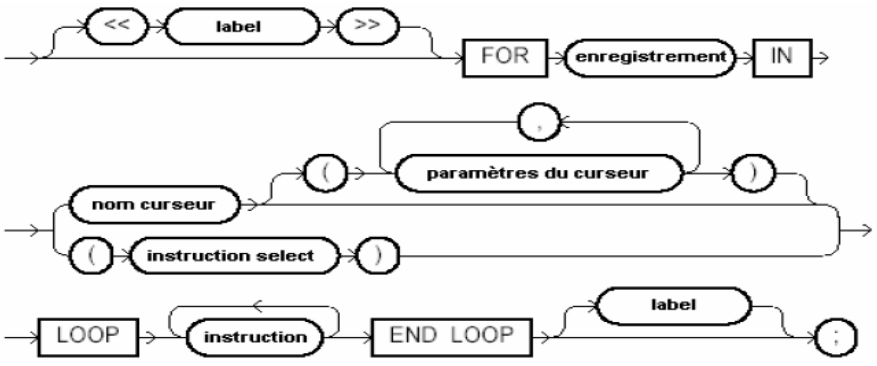
- Cette instruction permet de ramener une ligne d'un curseur préalablement ouvert avec l'instruction OPEN ou OPEN FOR
- Si l'instruction **FETCH** ne ramène plus de ligne (fin du curseur) l'attribut %NOTFOUND prend la valeur TRUE et l'attribut %FOUND prend la valeur FALSE

Curseurs explicites (8)

- **nom curseur** représente le nom d'un curseur préalablement ouvert avec l'instruction OPEN ou OPEN FOR
- **nom variable curseur** représente le nom d'une variable curseur
- **nom variable curseur hôte** représente le nom d'une variable curseur transmise par un programme tiers (ex : Pro*C, Pro*Cobol, etc.)
- **variable** représente le nom d'une variable préalablement définie dans la section déclarative, qui doit être du même type que la colonne ramenée par l'instruction Select
- **enregistrement** représente le nom d'un enregistrement préalablement défini dans la section déclarative qui doit être du même type que la ligne ramenée par l'instruction Select
- **nom collection** représente le nom d'une collection préalablement définie dans la section déclarative **nom tableau hôte** représente le nom du tableau transmis par un programme tiers

Boucle For avec curseur : For in …

Cette instruction permet de gérer un curseur sans utiliser OPEN, FETCH et CLOSE



Boucle For avec curseur : For in … (2)

```
SQL> Declare
3 CURSOR C_EMP IS
4 Select *
6 From EMP
8 Where  job = 'CLERK' ;
11Begin
13 For c IN C_EMP Loop
14 dbms_output.put_line( To_char( c.Empno ) || ' - ' || c.Nom) ;
15 End loop ;
16 End ;
17 /
1214 - DUPONT
1412 - DUFER
1242 – DUNOIR
Procédure PL/SQL terminée avec succès.
```

- La variable de curseur implicite **Cur**, non définie dans la section déclarative, doit être utilisée pour manipuler dans la boucle,
- les objets du curseur (To_char(Cur.empno),Cur.ename)
- Après l'instruction **END LOOP;** l'utilisation de cette variable génère une erreur
- Avec cette syntaxe, les instructions OPEN, FETCH et CLOSE sont inutiles

Boucle For avec curseur paramétré

```
SQL> Declare
3 CURSOR C_EMP ( PC$Job IN EMP.job%Type ) IS
4 Select  * From EMP Where job = PC$Job ;
11 Begin
13 For C IN C_EMP( 'vendeur' ) Loop
14 dbms_output.put_line( To_char( C.empno ) || ' - ' || C.ename ) ;
15 End loop ;
16 End ;
17 /
1400 - DUBLANC
1414 - DUNOIR
1144 - DUGRIS
Procédure PL/SQL terminée avec succès.
```

- **Attention :**

  Le passage des paramètres s'effectue sur le curseur déclaré (C_EMP) et non sur la variable curseur **(C)**

Section de traitement d'exception

- Débutée par le mot clé **EXCEPTION**, elle contient le code mis en oeuvre pour la gestion des erreurs générées par la section d'exécution (des milliers codes d'erreur d'Oracle)
- Une erreur survenue lors de l'exécution du code déclenche ce que l'on nomme une exception. Le code erreur
- associé est transmis à la section **EXCEPTION**, pour vous laisser la possibilité de la gérer et donc de ne pas mettre fin prématurément à l'application.
- En plus des erreurs Oracle, on peut intercepter ses propres erreurs en déclarant des variables dont le type est **exception** et en provoquant soi-même le saut dans la section de gestion des erreurs à l'aide de l'instruction **RAISE**

Principe de traitement d'exception

Phase 1 (optionnelle)

Déclarer les exceptions dans la DECLARE SECTION (sauf les exceptions prédéfinies par le système) avec le type EXCEPTION

Phase 2 (optionnelle)

Générer les exceptions dans la SECTION d'EXECUTION (sauf les exceptions prédéfinies par le système) avec l'instruction RAISE

Phase 3 (optionnelle)

Traiter les exceptions capturées dans la SECTION d'EXCEPTION avec l'instruction WHEN

Principe de traitement d'exception

- Il n'est pas possible de déclarer la même exception deux fois dans le même bloc. Toutefois, dans le cas de blocs imbriqués, vous pouvez déclarer la même exception dans la section **EXCEPTION** de chaque bloc

```
DECLARE
LE$Fin Exception ;
BEGIN
DECLARE
LE$Fin Exception ;
BEGIN
...
```

```
EXCEPTION
WHEN LE$Fin Then
...
END ;
EXCEPTION
WHEN LE$Fin Then
...
END ;
```


Génération des exceptions : RAISE

Si la variable LN$I est > 2,
alors on provoque le saut
dans la section EXCEPTION avec
l'erreur utilisateur LE$Fin

```
SQL> Declare
LN$I pls_integer := 0 ;
LE$Fin exception ;
Begin
Loop
LN$I := LN$I + 1 ;
dbms_output.put_line( to_char( LN$I) ) ;
```

```
If LN$I > 2 Then RAISE LE$Fin ;
End if ; End loop ;
Exception
When LE$Fin Then Null ;
End ;
/
123
```

Section de traitement d'exception : exceptions prédéfinies

Exception prédéfinie	Erreur Oracle	Valeur de SQLCODE
ACCESS_INTO_NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Toutes les autres exceptions doivent être interceptées via leur code erreur numérique.

Association à une erreur système : PRAGMA

On peut associer un code erreur Oracle à vos propres variables exception à l'aide du mot clé **PRAGMA EXCEPTION_INIT**, dans le cadre de la section déclarative de la façon suivante :

Nom_exception EXCEPTION ;
PRAGMA EXCEPTION_INIT(nom_exception, -code_error_oracle);

- Lorsque l'on tente d'insérer plus de caractères dans une variable que sa déclaration ne le permet, Oracle déclenche une erreur -6502. Nous pouvons "nommer" cette erreur en e_trop_long et l'intercepter dans la section exception

```
Declare $Chaine varchar2(10) ;
e_trop_long exception ;
pragma exception_init( e_trop_long, -6502);
Begin
$Chaine := rpad( ' ', 30)
Exception
when e_trop_long then dbms_output.put_line( 'Chaîne
     de caractères trop longue') ;
End;
```

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
4. Section de traitement d'exception

Fonction d'erreur : SQLCODE

Le code erreur numérique Oracle ayant généré la plus récente erreur est récupérable en interrogeant la fonction **SQLCODE**.
Le libellé erreur associé est récupérable en interrogeant la fonction **SQLERRM**

```
SQL> Declare
$Chaine varchar2(10) ;
Begin
$Chaine := rpad( ' ', 30) ;
Exception
when others then
dbms_output.put_line( 'Code erreur : ' || to_char( SQLCODE )) ;
dbms_output.put_line( 'libellé erreur : ' || to_char( SQLERRM )) ;
End ;
/
```


II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

1- Bloc PL SQL
4. Section de traitement d'exception

Poursuite de l'exécution après l'interception d'une exception

Une fois dans la section **EXCEPTION**, il n'est pas possible de retourner dans la section exécution juste après l'instruction qui a généré l'erreur.
Par contre il est tout à fait possible d'encadrer chaque groupe d'instructions voire même chaque instruction avec les mots clé

BEGIN # EXCEPTION # END;

Cela permet de traiter l'erreur et de continuer l'exécution

```
Declare
$Ch1 varchar2(20) := 'Phrase longue';
$Chaine varchar2(10) ;
e_trop_long exception ;
pragma exception_init(e_trop_long, -6502 ) ;
 Begin
  Begin
    $Chaine := LC$Ch1;
```

```
Exception
when e_trop_long then
$Chaine := Substr( LC$Ch1, 1, 10 ) ;
End ;  -- poursuite du traitement --
dbms_output.put_line(LC$Chaine ) ;
End ;
/
Phrase lon
```

Message d'erreur : RAISE_APPLICATION_ERROR

Vous pouvez également définir vos propres messages d'erreur avec la procédure RAISE_APPLICATION_ERROR
DBMS_STANDARD.raise_application_error(numero_erreur, message[, {TRUE | FALSE}])

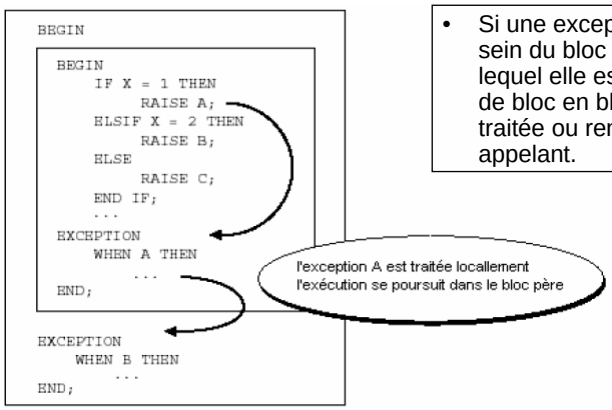
- **numero_erreur** représente un entier négatif compris entre -20000 et -20999
- **message** représente le texte du message d'une longueur maximum de 2048 octets
- **TRUE** indique que l'erreur est ajoutée à la pile des erreurs précédentes
- **FALSE** indique que l'erreur remplace toutes les erreurs précédentes

Remarque

Du fait que cette procédure appartienne a un paquetage standard, il n'est pas nécessaire de préfixer cette procédure

L'appel de la procédure raise_application_error ne peut être exécuté que depuis une procédure stockée, et déclenche un retour immédiat au programme appelant en lui transmettant le code et le libellé de l'erreur

Propagation des exceptions



- Si une exception n'est pas traitée au sein du bloc **BEGIN # END;** dans lequel elle est générée, elle remonte de bloc en bloc jusqu'à ce qu'elle soit traitée ou rende la main au programme appelant.

- Dans cet exemple, l'exception A est traitée dans le bloc local. Le traitement se poursuit dans le bloc parent

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage
1. Introduction

- Une procédure est un ensemble de code PL/SQL nommé, défini par l'utilisateur et généralement stocké dans la BDD
- Une fonction est identique à une procédure à la différence qu'elle retourne une valeur
- Un paquetage est le regroupement de plusieurs procédures et fonctions dans un objet distinct
- Ces ensembles nommés sont stockés dans la base de données, offrant les avantages suivants :
  - Le code relatif aux règles de gestion est centralisé. Cela permet de dissocier les fonctions au sein d'une équipe. La partie traitement des règles de gestion est confiée à une partie de l'équipe et la conception des interfaces est confiée à l'autre partie
  - Ces traitements stockés sont donc déportés des interfaces clientes, permettant le partage du code entre plusieurs applications ainsi qu'une amélioration des performances, car le code stocké est pré-compilé
  - Ces traitements sont accessibles par toute application tierce supportant l'appel des procédures stockées (Sql*Plus, Forms, Reports, Pro*C, Pro*Cobol, etc.)
  - Cela permet également de tirer parti de la réutilisation des requêtes dans la base qui se trouvent dans le pool partagé de la zone SGA(System Global Area)

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

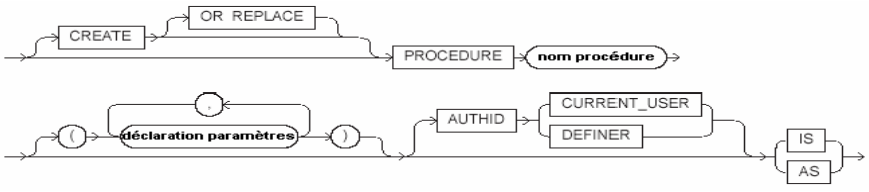
II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage
1. Introduction

- **Pour créer un objet procédural, vous devez disposer du privilège système**
  - **CREATE PROCEDURE pour votre schéma ou**
  - **du privilège système CREATE ANY PROCEDURE pour la création dans un autre schéma**
- **Pour autoriser un autre schéma à exécuter une procédure de votre schéma, vous devez lui octroyer le privilège EXECUTE**

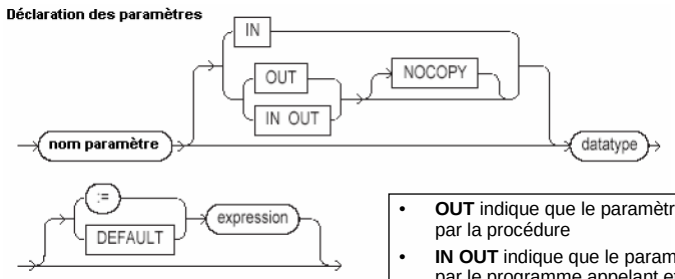
GRANT EXECUTE ON ma_procedure TO autre_schéma

2- Blocs nommés : Procédure, fonctions et paquetage
2. Procédure



- **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
- **AUTHID** indique sur quel schéma la procédure s'applique :
  - **CURRENT_USER :** Indique que la procédure utilise les objets du schéma de l'utilisateur qui appelle la procédure
  - **DEFINER**(défaut) / Indique que la procédure utilise les objets du schéma de création de la procédure

2- Blocs nommés : Procédure, fonctions et paquetage
2. Procédure



- **nom paramètre** est le nom donné par l'utilisateur au paramètre transmis
- **IN**(valeur par défaut) indique que le paramètre transmis par le programme appelant n'est pas modifiable par la procédure

- **OUT** indique que le paramètre est modifiable par la procédure
- **IN OUT** indique que le paramètre est transmis par le programme appelant et renseigné par la procédure
- **NOCOPY** indique que le paramètre est transmis par référence (pointeur) et non par copie de la valeur
- Par défaut, les paramètres sont transmis par copie, c'est à dire qu'un espace mémoire est créé pour recevoir une copie de la valeur

2- Blocs nommés : Procédure, fonctions et paquetage
2. Procédure

```
SQL> CREATE OR REPLACE
     PROCEDURE Augmentation (Numemp
     IN EMP.empno%Type -- numéro de
     l'employé , Pourcent IN NUMBER
     -- pourcentage d'augmentation) IS
BEGIN,-- augmentation de l'employé
     Update EMP Set sal = sal *
     PN$Pourcent
•    Where empno = PN$Numemp ;
END;
 /
Procédure créée.
```

- La procédure Augmentation reçoit deux paramètres
- **Numemp** en entrée (IN) de même type que la colonne empno de la table EMP qui reçoit le numéro d'employé
- **Pourcent** en entrée (IN) de type NUMBER qui reçoit le pourcentage d'augmentation

2- Blocs nommés : Procédure, fonctions et paquetage
2. Procédure

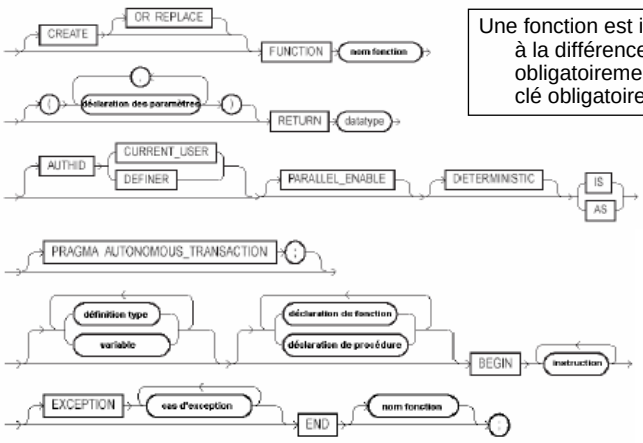
```
SQL> Declare
unEmp EMP%Rowtype ;
Begin
Select * Into unEmp From EMP
Where empno = 7369 ; -- lecture ligne avant
     mise à jour
dbms_output.put_line( 'Avant augmentation ' ||
     To_char( unEmp.empno )
|| ' ' || unEMp.ename || ' --> ' || To_char( LR$
unEmp.sal ) ) ;
Augmentation( 7369, 1.1 ) ; -- appel de
     la;procédure
Select * Into LR$Emp From EMP Where
     empno = 7369 ; -- lecture ligne après
     mise à jour
dbms_output.put_line( 'Après augmentation '
     || To_char( LR$Emp.empno )
|| ' ' || LR$EMp.ename || ' --> ' || To_char(
     LR$Emp.sal ) ) ;
End ;
/
Avant augmentation 7369 SMITH --> 880
Après augmentation 7369 SMITH --> 968
Procédure PL/SQL terminée avec succès.
```

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage

3. Fonction



Une fonction est identique à une procédure à la différence qu'elle retourne obligatoirement une valeur d'où le mot clé obligatoire **RETURN**

- 81 -

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage

3. Fonction

- **CREATE** indique que l'on veut créer une fonction stockée dans la base
- La clause facultative **OR REPLACE** permet d'écraser une fonction existante portant le même nom
- **nom fonction** est le nom donné par l'utilisateur à la fonction
- **AUTHID** indique sur quel schéma la fonction s'applique :
  - **CURRENT_USER** Indique que la fonction utilise les objets du schéma de l'utilisateur qui appelle la fonction
  - **DEFINER**(défaut) : Indique que la fonction utilise les objets du schéma de création de la fonction
- **nom paramètre** est le nom donné par l'utilisateur au paramètre transmis
- **IN**(valeur par défaut) indique que le paramètre transmis par le programme appelant n'est pas modifiable par la fonction
- **OUT** indique que le paramètre est modifiable par la procédure

- **IN OUT** indique que le paramètre est transmis par le programme appelant et renseigné par la fonction
- **NOCOPY** indique que le paramètre est transmis par référence (pointeur) et non par copie de la valeur
- **datatype** représente le type SQL ou PL/SQL du paramètre
- **:=** représente le symbole d'assignation d'une valeur par défaut
- **DEFAULT** identique à :=
- **expression** représente la valeur par défaut du paramètre (doit être conforme au type du paramètre)
- **PRAGMA AUTONOMOUS_TRANSACTION** indique que la fonction sera exécutée dans une transaction autonome

- 82 -

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage
3. Fonction

```
SQL> CREATE OR REPLACE FUNCTION
     F_Test_Augmentation (Numemp IN
     EMP.empno%Type, Pourcent IN
     NUMBER ) Return NUMBER IS
Salaire EMP.sal%Type ;
BEGIN
Select sal Into Salaire From EMP Where
     empno = Numemp ;
 -- augmentation virtuelle de l'employé
Salaire := Salaire * Pourcent ;
Return(Salaire ) ; -- retour de la valeur
END;
/
Fonction créée.
```

```
SQL> Declare
Salaire emp.sal%Type ;
Begin
Select sal Into Salaire From EMP Where
     empno = 7369 ;
dbms_output.put_line( 'Salaire de 7369
     avant augmentation ' || To_char( Salaire
     ));
dbms_output.put_line( 'Salaire de 7369
     après augmentation ' || To_char(
     F_Test_Augmentation(7369, 1.1 ) ) ) ;
End ;
/
Salaire de 7369 avant augmentation 880
Salaire de 7369 après augmentation 968
Procédure PL/SQL terminée avec succès.
```

- 83 -

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage
4. Paquetage : introduction

- **Un paquetage est un ensemble de procédures et fonctions regroupées dans un objet nommé**
  - Par exemple le paquetage Oracle DBMS_LOB regroupe toutes les fonctions et procédures manipulant les grands objets (LOBs)
  - Le paquetage UTL_FILE regroupe les procédures et fonctions permettant de lire et écrire des fichiers du système d'exploitation
- **Un paquetage est organisé en deux parties distinctes**
  - la partie spécification d'un paquetage est déclarée avec l'instruction **CREATE [OR REPLACE] PACKAGE**
  - la partie corps est déclarée avec l'instruction **CREATE [OR REPLACE] PACKAGE BODY**
- **L'accès à un objet d'un paquetage est réalisé avec la syntaxe suivante :**

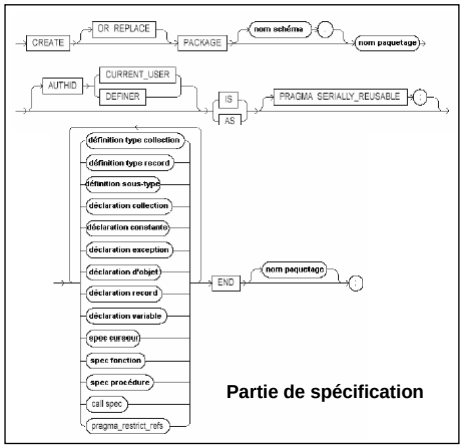
nom_paquetage.nom_objet[(liste paramètres)]

- 84 -

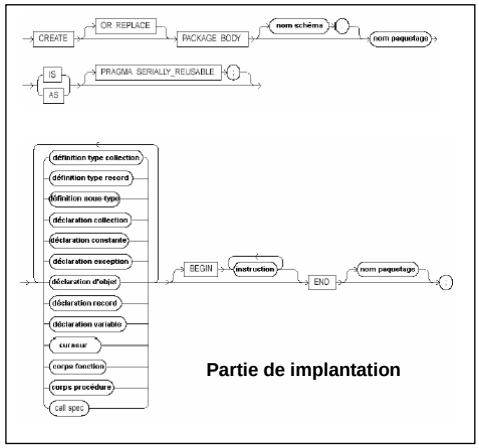
IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage
4. Paquetage : Spécification et Implantation



Partie de spécification



Partie de implantation

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage
4. Paquetage : exemple

```
SQL> CREATE OR REPLACE PACKAGE
     Pkg_Finance IS
-- Variables globales et publiques
Salaire EMP.sal%Type ;
-- Fonctions publiques
FUNCTION F_Test_Augmentation ( Numemp IN
     EMP.empno%Type, Pourcent IN NUMBER)
     Return NUMBER ;
-- Procédures publiques
14 PROCEDURE Test_Augmentation (
Numemp IN EMP.empno%Type
-- numéro de l'employé
Pourcent IN OUT NUMBER
-- pourcentage d'augmentation ) ;
End Pkg_Finance ;
/
```

```
SQL> CREATE OR REPLACE PACKAGE BODY
     Pkg_Finance IS -- Variables globales privées
Emp EMP%Rowtype ;
-- Procédure privées
PROCEDURE Affiche_Salaires IS
CURSOR C_EMP IS select * from EMP ;
BEGIN
OPEN C_EMP ;
Loop
FETCH C_EMP Into GR$Emp ;
Exit when C_EMP%NOTFOUND ;
dbms_output.put_line( 'Employé ' |
GR$Emp.ename || ' --> ' || Lpad( To_char( GR$Emp.sal), 10 ) )
     ;
End loop ;
CLOSE C_EMP ;
END Affiche_Salaires ;-- Fonctions publiques
```

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage
4. Paquetage : exemple (2)

```
FUNCTION F_Test_Augmentation (Numemp IN
      EMP.empno%Type, Pourcent IN NUMBER
 Return NUMBER
IS
$Salaire EMP.sal%Type ;
BEGIN
Salaire := Salaire * Pourcent ;
-- Affectation de la variable globale publique
Salaire := LN$Salaire ;
Return( LN$Salaire ) ; -- retour de la valeur
END F_Test_Augmentation;
--Procédures publiques
PROCEDURE Test_Augmenta (
Numemp IN EMP.empno%Type
Pourcent IN OUT NUMBER ) IS
Salaire EMP.sal%Type ;
BEGIN
Select sal Into Salaire From EMP Where
      empno = -- augmentation virtuelle de
      l'employé
Pourcent := LN$Salaire * PN$Pourcent ;
-- appel procédure privée
Affiche_Salaires ;
END Test_Augmentation;
55
56
END Pkg_Finance;
58 /
Corps de package créé.
```


II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage
4. Paquetage : exemple (3)

- La spécification du paquetage est créée avec
  - une variable globale et publique : GN$Salaire
  - une procédure publique : Test_Augmentation
  - une fonction publique : F_Test_Augmentation qui sont visibles depuis l'extérieur (le programme appelant)
- Le corps du paquetage est créé avec
  - une procédure privée : Afiche_Salaires qui n'est visible que dans le corps du paquetage
  - une variable globale au corps du paquetage : GR$Emp utilisée par la procédure privée
- La syntaxe d''accès à un objet d'un paquetage :
  - **nom_paquetage.nom_objet[(liste paramètres)]**

- Exemple d'un appel de la fonction F_Test_Augmentation :

```
SQL> Declare
LN$Salaire emp.sal%Type ;
Begin
Select sal Into LN$Salaire From EMP Where empno =
      7369 ;
dbms_output.put_line( 'Salaire de 7369 avant
      augmentation ' || To_char( LN$Salaire ) ) ;
dbms_output.put_line( 'Salaire de 7369 après
      augmentation ' || To_char(
      Pkg_Finance.F_Test_Augmentation( 7369, 1.1 ) )
      ) ;
End ;
/
Salaire de 7369 avant augmentation 880
Procédure PL/SQL terminée avec succès.
```

2- Blocs nommés : Procédure, fonctions et paquetage
4. Paquetage : exemple (4)

- Exemple d'un appel de la procédure Test_Augmentation

```
SQL> Declare
LN$Pourcent NUMBER := 1.1 ;
Begin
Pkg_Finance.Test_Augmentation( 7369, LN$Pourcent ) ;
dbms_output.put_line( 'Employé 7369 après augmentation : ' || To_char( LN$Pourcent ) ) ;
End ;
/
Employé SMITH --> 880
Employé 7369 après augmentation : 968
Procédure PL/SQL terminée avec succès.
```

- Exemple d'un appel de la fonction F_Test_Augmentation :

```
SQL> Declare
LN$Salaire emp.sal%Type ;
Begin
Select sal Into LN$Salaire From EMP Where empno = 7369 ;
dbms_output.put_line( 'Salaire de 7369 avant augmentation ' || To_char( LN$Salaire ) ) ;
dbms_output.put_line( 'Salaire de 7369 après augmentation ' || To_char( Pkg_Finance.F_Test_Augmentation( 7369, 1.1 ) ) ) ;
End ;
/
Salaire de 7369 avant augmentation 880
Procédure PL/SQL terminée avec succès.
```


2- Blocs nommés : Procédure, fonctions et paquetage
6. Fonctions natives du système: de chaînes de caractères

- Concarténation : 2 syntaxes :
  - **chaîne || chaîne**
  - **CONCAT(chaîne, chaîne)**
- Ajout : Ces fonctions formatent une chaîne sur une longueur donnée par ajout de caractères avant (LPAD) ou après (RPAD) la chaîne passée en argument
  - **LPAD(chaîne, longueur [, 'caractères'])**
  - **RPAD(chaîne, longueur [, 'caractères'])**
  - chaîne représente le nom d'une colonne, d'une variable ou un littéral
  - longueur représente le nombre total de caractères du résultat
  - caractères représente le ou les caractères de remplissage

- Exemple :
  - RPAD('Total', 20, '.')

    Total.....
  - LPAD('Hello', 20, '.')

    .....Hello
  - RPAD(LPAD('2320 euros', 20,'*'), 30, '*')

    **********2320 euros**********
  - LPAD('Hello', 20, '-*-')

    -*--*--*--*--*-Hello
- Note : Si caractères n'est pas spécifié, le caractère par défaut est l'espace (CHR(32))

2- Blocs nommés : Procédure, fonctions et paquetage
6. Fonctions natives du système: de chaînes de caractères(2)

- Suppression :
  - **LTRIM(chaîne [, 'caractères'])**
    Suppression à gauche de la chaîne
  - **RTRIM(chaîne [, 'caractères'])**
    Suppression à droite de la chaîne
  - **TRIM(chaîne, longueur [, 'caractères'])**
    Suppression à gauche et/ou à droite de la chaîne

-**chaîne** représente la chaîne en entrée de la fonction
- **LEADING** : tous les caractères à gauche identiques à **caractère cité** sont supprimés
- **TRAILING** : tous les caractères à droite identiques à **caractère** sont supprimés
-**BOTH** ou rien est spécifié, tous les caractères à gauche et à droite identiques à **caractère** sont supprimés
-si caractère n'est pas cité la valeur par défaut est l'espace

- si seule **chaîne** est spécifiée tous les espaces en début et fin de chaîne sont supprimés
- si **caractère** ou **chaîne** est NULL, la fonction retourne NULL

- Exemple :

```
SQL> Declare
LC$Ch1 Varchar(20) := ' libellé ' ;
LC$Ch2 Varchar(20) := '***libellé***' ;
Begin
dbms_output.put_line( TRIM( LC$Ch1 ) ) ;
dbms_output.put_line( TRIM( '*' FROM LC$Ch2 ) ) ;
End ;
/
libellé
libellé
Procédure PL/SQL terminée avec succès.
```


2- Blocs nommés : Procédure, fonctions et paquetage
6. Fonctions natives du système: de chaînes de caractères(3)

- Position de sous chaîne :
  - **INSTR(chaîne, sous-chaîne [, début [,nombre occurrences]])**
  - **INSTRB(chaîne, sous-chaîne [, début [,nombre occurrences]])**
  - **chaîne** représente le nom d'une colonne, d'une variable ou un littéral passé en argument
  - **sous-chaîne** représente le nom d'une variable ou un littéral dont on cherche la position
  - **début** (optionnel) représente la position de départ de la recherche dans chaîne
  - **nombre occurrences** (si début renseigné) représente le nombre d'occurrences trouvées à ignorer
  - Lorsque **sous-chaîne** représente plusieurs caractères, la fonction retourne la position du premier caractère de la sous-chaîne

- Autres fonctions :
  - **LOWER(chaîne)**
  - **NLS_LOWER(chaîne [, nls_paramètre])**
  - **UPPER(chaîne)**
  - **NLS_UPPER(chaîne [, nls_paramètre])**
  - **INITCAP(chaîne)**
  - **NLS_INITCAP(chaîne [, nls_paramètre])**
  - **LENGTH(chaîne)**
- Exemple :
  - INITCAP('le sgbd oracle') Le Sgbd Oracle
  - LENGTH('le sgbd oracle') 14
  - INSTR('le sgbd oracle', 'oracle') 9
  - INSTR('le sgbd oracle d''oracle corporation', 'oracle', 1, 2) 18
  - INSTR('le sgbd oracle d''oracle corporation', 'texte', 1, 2) 0

(si la sous-chaîne n'est pas trouvée, la fonction retourne 0)

2- Blocs nommés : Procédure, fonctions et paquetage
6. Fonctions natives du système: de chaînes de caractères(4)

- Extraction :
  - **SUBSTR(chaîne, début [, 'nombre'])**
  - **chaîne** représente le nom d'une colonne, d'une variable ou un littéral
  - **début** représente la position de départ de recherche dans la chaîne
  - **nombre** représente le nombre de caractères à extraire
  - si **nombre** est négatif l'extraction débute à partir de la fin de la chaîne
- Exemple :
  - SUBSTR('le sgbd oracle', 4, 4)

  sgbd
  - SUBSTR('le sgbd oracle', 4)

  sgbd oracle

Note : si **nombre** est négatif l'extraction débute à partir de la fin de la chaîne

- Remplacement, translation :
  - **REPLACE(chaîne, chaîne source, chaîne cible)**
  - **TRANSLATE(chaîne, chaîne source, chaîne cible)**
  - **chaîne** représente le nom d'une colonne, d'une variable ou un littéral passé en argument
  - **chaîne source** représente le nom d'une variable ou un littéral de recherche
  - **chaîne cible** représente le nom d'une variable ou un littéral de remplacement
- Note : dans Translate Chaque caractère présent dans **chaîne** qui est également présent dans **chaîne source** est remplacé par le caractère qui occupe la même position dans **chaîne cible**

2- Blocs nommés : Procédure, fonctions et paquetage
6. Fonctions natives du système: de chaînes de caractères(5)

- Exemple :
  - REPLACE('banjaur', 'a', 'o')

    Bonjour
  - REPLACE('"champs1","champs2"', '"', '')

    champs1,champs2
  - TRANSLATE('Pas d''accents : éèàùôö', 'éèàùôö', 'eeauoo')

    Pas d'accents : eeauoo

- Exemple (suite) :
  - TRANSLATE('Nom*de[fichier<unix>', '_ /\<>|(){}[]*&''"$;','_____________________')

    Nom_de_fichier_unix_
  - TRANSLATE('Nom+de|fichier!unix', 'A+|!', 'A')

    Nomdefichierunix
  - Le premier caractère A de la chaîne source est un leurre qui indique à la fonction de remplacer outes les occurrences de A par A et de remplacer les autres caractères (+|!) par rien.

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage
6. Fonctions natives du système: arithmétiques

- **ABS(valeur) :** Valeur absolue d'un nombre
- **CEIL(valeur) :** Entier supérieur ou égal à **valeur**
- **FLOOR(valeur) :** Entier inférieur ou égal à **valeur**
- **MOD(valeur, diviseur) :** Reste d'une division
  - Lorsque **diviseur** est supérieur à **valeur**, la fonction retourne la valeur MOD(5, 12) = 5
  - Si **valeur** est un entier alors MOD(valeur, 1) = 0 (idéal pour tester que valeur est entier)
  - Si **valeur** est un entier pair alors MOD(valeur, 2) = 0
- **POWER(valeur, exposant) :** élévation d'un nombre à une puissance
- **SQRT(valeur) :** Racine carrée d'un nombre
- **EXP(valeur) :** e (2,71828182845905) élevé à une puissance
- **LN(valeur) :** Logarithme naturel, ou base e, d'une valeur
- **LOG(base, valeur) :** Logarithme d'une valeur
- **ROUND(valeur, précision) :** Arrondi d'une valeur à un certain nombre de chiffres de précision
- **TRUNC(valeur, précision) :** Suppression d'une partie des chiffres de précision décimale
- **SIGN(valeur) :** Signe d'une valeur
- **SIN(valeur), COS(valeur), TAN(valeur) :** Renvoie la valeur trigonométrique d'un angle exprimé en radians
- **ASIN(valeur), ACOS(valeur), ATAN(valeur):** Renvoie respectivement l'arc sinus, cosinus et tangente en radians
- **AVG(colonne) :** Valeur moyenne des valeurs de **colonne**
- **COUNT(colonne) :** Nombre de valeurs de **colonne** (les valeurs NULL ne sont pas prises en compte)
- **MAX(colonne) :** Valeur maximum des valeurs de **colonne**

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage
6. Fonctions natives du système: arithmétiques

- **MIN(colonne) :** Valeur minimum des valeurs de **colonne**
- **SUM(colonne)**
- Somme des valeurs de colonne
- **GREATEST(valeur1, valeur2, valeur3,#) :** Valeur la plus grande de la liste
- **LEAST(valeur1, valeur2, valeur3,#) :** Valeur la plus petite de la liste
- **BIN_TO_NUM(bit [,bit[...]]) :** Conversion d'une suite de bits en nombre
- **BITAND(arg1, arg2) :** Applique un ET logique sur les deux arguments
- **CONVERT(chaîne, jeu caract source, jeu caract cible) :** Conversion d'une chaîne d'un jeu de caractères à un autre
- **DECODE(valeur, si1, alors1, si2, alors2,#.., sinon) :** Substitution valeur par valeur
- **BIN_TO_NUM(bit [,bit[...]]) :** Conversion d'une suite de bits en nombre
- **DUMP(expr [,format_retour [,début [,longueur]]]) :** Retourne le type interne, longueur en octets de l'expression passée en argument
- **format_retour :** peut prendre l'une des quatre valeurs suivantes :
  - **8** octal
  - **10** décimal
  - **16** hexadécimal
  - **17** caractères
- **HEXTORAW(char) :** Conversion d'un nombre hexadécimal en un nombre binaire
- **RAWTOHEX(raw) :** Conversion d'un nombre binaire en nombre hexadécimal
- **ROWIDTOCHAR(rowid) :** Conversion d'un ROWID en chaîne
- **TO_CHAR(date [, 'format']) et**
- **TO_CHAR(nombre [, 'format']) :** Transformation d'un type DATE ou NUMBER en chaîne

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage
6. Fonctions natives du système: arithmétiques

- **TO_DSINTERVAL(chaîne)(9i) :** Transformation d'un type CHAR ou VARCHAR2 en INTERVAL DAY TO SECOND
- **TO_LOB(LONG) et**
- **TO_LOB(LONG RAW) :** Conversion d'une valeur de type LONG ou LONG RAW en valeur de type LOB
- **TO_CLOB(char) et**
- **TO_CLOB(colonne lob) :** Conversion d'une valeur de type CHAR ou LOB en valeur de type CLOB
- **TO_MULTI_BYTE :** Conversion d'une chaîne de caractères mono-octet en chaîne de caractères multi-octets
  - Cette fonction n'est nécessaire que si votre jeu de caractères contient à la fois des caractères mono-octets et des caractères multi-octets
- **TO_NUMBER(chaîne [, 'format']) :** Transformation d'un type CHAR ou VARCHAR2 en nombre
- **TO_SINGLE_BYTE :** Conversion d'une chaîne de caractères multi-octets en chaîne de caractères mono-octet
- Cette fonction n'est nécessaire que si votre jeu de caractères contient à la fois des caractères mono-octets et des caractères multi-octets
- **TO_TIMESTAMP(chaîne)(9i)** : Transformation d'un type CHAR / VARCHAR2 en TIMESTAMP
- **TO_TIMESTAMP_TZ(chaîne)(9i) :** Transformation d'un type CHAR ou VARCHAR2 en TIMESTAMP + TIME ZONE
- **TO_YMINTERVAL(chaîne)(9i) :** Transformation d'un type CHAR ou VARCHAR2 en INTERVAL YEAR TO MONTH
  - Chaîne représente un couple année-mois 'AA-MM'
- **VSIZE(argument) :** Retourne le nombre d'octets nécessaires au stockage de l'argument

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage
6. Fonctions natives du système: Dates

- **ADD_MONTHS(date, nombre de mois) :** Ajoute ou soustrait un nombre de mois à une date**TO_LOB(LONG) et**
  - select SYSDATE, ADD_MONTHS(SYSDATE, 1) from dual ;
  - select SYSDATE, ADD_MONTHS(SYSDATE, -12) from dual ;
  - ---------- ----------
  - 31/01/2004 31/01/2003
- **EXTRACT (datetemps FROM valeur)(9i) :** Extraction d'un segment d'une valeur de type date ou d'un littéral de type intervalle (Pour une raison inconnue, cette fonction ne permet pas d'extraire l'heure d'une date...)
  - datetemps peut être l'un des arguments suivants
  - valeur est l'un des arguments suivants
  - Exemple :

```
SQL> SELECT EXTRACT(YEAR FROM
          TO_DATE('10/01/2004','DD/MM/YYYY')) FROM DUAL;
---------------------------------------------------
2004
```

- **LAST_DAY(date) :** Dernier jour du mois de la date passée en argument
- **MONTHS_BETWEEN(date2, date1) :** Nombre de mois qui séparent **date2** de **date1**
- **NEXT_DAY(date, 'jour') :** Date du prochain jour après **date** ou **jour** est un jour de la semaine
- **NEW_TIME(date, fuseau1, fuseau2) :** Date et heure de **date** en time zone **fuseau2** lorsque **date** est en time zone **fuseau1**
- **NUMTODSINTERVAL(nombre, 'type')(9i) :** Conversion d'un nombre en intervalle de type DAY TO SECOND
  - **'type'** peut prendre l'une des quatre valeurs suivantes :
  - **'DAY'** précise que nombre indique un nombre de jours
  - **'HOUR'** précise que nombre indique un nombre d'heures
  - **'MINUTE'** précise que nombre indique un nombre de minutes
  - **'SECOND'** précise que nombre indique un nombre de secondes

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

2- Blocs nommés : Procédure, fonctions et paquetage
6. Fonctions natives du système: Dates

- **NUMTOYMINTERVAL(nombre, 'type')(9i) :** Conversion d'un nombre en intervalle de type YEAR TO MONTH
  - **'type'** peut prendre l'une des deux valeurs suivantes :
  - **'YEAR'** précise que nombre indique un nombre d'années
  - **'MONTH'** précise que nombre indique un nombre de mois
- **ROUND(date [, format]) :** sans l'argument **format**, la date est arrondie au matin 00:00:00 si l'heure transmise dans date est située avant 12:00, sinon elle est arrondie au lendemain 00:00:00
- **TRUNC(date [, format]) :** sans l'argument **format**, la date est arrondie au matin 00:00:00
- **SYS_EXTRACT_UTC(timestamp)(9i) :** Conversion d'une date au fuseau Greenwich
- Exemples :

```
SQL> -- Ramener une date/heure au format Greenwich
SELECT SYSDATE,
    SYS_EXTRACT_UTC(SYSTIMESTAMP) FROM
    DUAL;
```

- Exemples :
  - select TRUNC(to_date('01/01/2004 10:25','DD/MM/YYYY HH24:MI')) from dual ;
  - select TRUNC(to_date('01/01/2004 18:25','DD/MM/YYYY HH24:MI')) from dual ;
  - SELECT SYSDATE + NUMTOYMINTERVAL(6,'MONTH') FROM DUAL ;$
  - SELECT SYSDATE, SYSDATE + NUMTODSINTERVAL(30,'SECOND') FROM DUAL ;
  - SELECT SYSDATE, SYSDATE + NUMTODSINTERVAL(8,'MINUTE') FROM DUAL ;
  - select sysdate, NEW_TIME(sysdate, 'AST', 'PST') from dual ;
  - select NEXT_DAY('31/01/2004', 'Lundi') from dual ;
  - select MONTHS_BETWEEN('01/10/2004', '01/01/2004') from dual ;
  - select LAST_DAY('01/01/2004') from dual ;
  - SELECT EXTRACT(YEAR FROM TO_DATE('10/01/2004','DD/MM/YYYY')) FROM DUAL;

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

3- Déclencheurs
1. Généralité

- Un déclencheur est un bloc PL/SQL associé à une vue ou une table, qui s'exécutera lorsqu'une instruction du langage de manipulation de données (DML) sera exécutée
- L'avantage principal du déclencheur réside dans le fait que le code est centralisé dans la base de données, et se déclenchera quel que soit l'outil utilisé pour mettre à jour ces données, donnant ainsi l'assurance qu'une utilisation d'un ordre DML depuis Sql*Plus, Forms ou n'importe quelle application tierce procurer un résultat identique sur les données l'inconvénient principal du déclencheur réside dans le fait que son exécution utilise des ressources qui peuvent augmenter sensiblement les temps de traitement, notamment lors de modifications massives apportées sur une table
- Un déclencheur s'exécute dans le cadre d'une transaction. Il ne peut donc pas contenir d'instruction COMMIT ou ROLLBACK ou toute instruction générant une fin de transaction implicite (ordre DDL)
- Les ordres SQL (SELECT, INSERT, UPDATE, DELETE) contenus dans le bloc PL/SQL et qui se réfèrent à la table sur laquelle s'exécute le déclencheur peuvent générer l'exception ORA-04091 TABLE IS MUTATING (pour une explication détaillée du problème des tables en mutation, référez-vous à l'article Résolution du problème de la table mutante (ora-04091) par Pomalaix

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

3- Déclencheurs
1. Généralité

- Le bloc PL/SQL qui constitue le trigger peut être exécuté avant ou après la vérification des contraintes d'intégrité
- Il peut être exécuté pour chaque ligne affectée par l'ordre DML ou bien une seule fois pour la commande
- Seules les colonnes de la ligne en cours de modification sont accessibles par l'intermédiaire de 2 variables de type enregistrement **OLD** et **NEW**
  - **OLD** représente la valeur avant modification
  - **OLD** n'est renseignée que pour les ordres DELETE et UPDATE. Elle n'a aucune signification pour un ordre INSERT, puisqu'aucune ancienne valeur n'existe
  - **NEW** représente la nouvelle valeur
  - **NEW** n'est renseignée que pour les ordres INSERT et UPDATE. Elle n'a aucune signification pour un ordre DELETE, puisqu'aucune nouvelle valeur n'existe

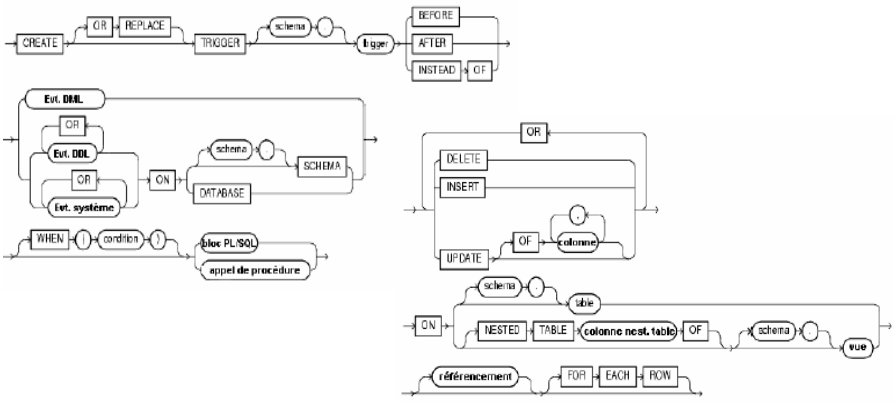
- Ces deux variables peuvent être utilisées dans la clause **WHEN** du déclencheur et dans la section exécutable
- Dans cette section, elle doivent être préfixées comme des variables hôtes avec l'opérateur **:**
  - Les noms de ces deux variables sont fixés par défaut, mais il est possible de les modifier en précisant les nouveaux noms dans la clause **REFERENCING**
  - **REFERENCING OLD AS nouveau_nom NEW AS nouveau_nom**

- 101 -

IV- PROGRAMMATION PROCEDURALE ET EVENEMENTIELLE

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

3- Déclencheurs
2. Syntaxe



- 102 -

3- Déclencheurs
2. Syntaxe

- **Evt. DML** représente l'évènement INSERT ou UPDATE ou DELETE
- **Evt. DDL** représente un évènement utilisateur
- **Evt. système** représente un évènement système
- **colonne nest. table** représente le nom d'une colonne de table imbriquée
- **référencement** représente le renommage des valeurs OLD et NEW

- Dans le cas d'un déclencheur **BEFORE UPDATE** ou **AFTER UPDATE**, la clause **OF** peut être ajoutée après le mot clé **UPDATE** pour spécifier la liste des colonnes modifiées.
- Cela permet de restreindre l'activation du déclencheurs sur les seules colonnes visées.
- Le mot clé **WHEN(condition)** permet également de restreindre le champs d'activation du déclencheur en ajoutant une clause restrictive

3- Déclencheurs
3. Déclencheur sur table

- Créons un déclencheur très basique qui ne fait qu'afficher le numéro et le nom d'un employé que l'on veut supprimer de la table EMP :

```
CREATE OR REPLACE TRIGGER TRG_BDR_EMP
BEFORE DELETE -- avant supression
ON EMP -- sur la table EMP
FOR EACH ROW -- pour chaque ligne
Declare
LC$Chaine VARCHAR2(100);
Begin
dbms_output.put_line( 'Suppression de l"employé n° ' ||
     To_char( :OLD.empno ) || ' -> ' || :OLD.ename ) ;
End ;
 /
Déclencheur créé.
```

- Supprimons maintenant un employé

```
delete from emp where empno = 7369
/
Suppression de l'employé n° 7369 ->
    SMITH
1 ligne supprimée.
SQL> rollback;
Annulation (rollback) effectuée.
```

La DRH annonce que désormais, tout nouvel employé devra avoir un numéro supérieur ou égal à 10000

3- Déclencheurs
3. Déclencheur sur table

- La DRH annonce que désormais, tout nouvel employé devra avoir un numéro supérieur ou égal à 10000. Il faut donc interdire toute insertion qui ne reflète pas cette nouvelle directive:

```
CREATE OR REPLACE TRIGGER
    TRG_BIR_EMP
BEFORE INSERT ON EMP -- sur la table EMP
FOR EACH ROW -- pour chaque ligne
Begin
If :NEW.empno < 10000 Then
RAISE_APPLICATION_ERROR ( -20010,
    'Numéro employé inférieur à 10000' ) ;
End if ;
End ;
/
Déclencheur créé.
```

- Tentons d'insérer un nouvel employé avec le numéro 9999 :

```
insert into emp (empno, ename, job)
    values( 9999, 'Burger', 'CLERK' ) ;
ERREUR à la ligne 1 :
ORA-20010: Numéro employé inférieur
    à 10000
ORA-06512: "SCOTT.TRG_BIR_EMP",
    ligne 3
ORA-04088: erreur lors d'exécution du
    déclencheur
    'SCOTT.TRG_BIR_EMP'
```

L'ordre d'insertion est rejeté

- Il est possible de gérer dans le même déclencheur des ordres différents en combinant les termes de la clause BEFORE avec le mot clé **OR**

3- Déclencheurs
3. Déclencheur sur table

- Exemple :

```
DROP TRIGGER TRG_BIR_EMP ;
CREATE OR REPLACE TRIGGER
    TRG_BIUDR_EMP
2BEFORE INSERT OR UPDATE OR
    DELETE ON EMP
FOR EACH ROW
Begin
If INSERTING Then
dbms_output.put_line( 'Insertion dans la
    table EMP' ) ;
End if ;
If UPDATING Then
dbms_output.put_line( 'Mise à jour de la
    table EMP' ) ;
End if ;
If DELETING Then
dbms_output.put_line( 'Suppression
    dans la table EMP' ) ;
End if ;
End ;
/
Déclencheur créé
insert into emp (empno, ename, job)
    values( 9993, 'Burger', 'CLERK') ;
Insertion dans la table EMP
1 ligne créée.
update emp set sal = 5000 where
    empno = 9993 ;
Mise à jour de la table EMP
1 ligne mise à jour.
```

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

3- Déclencheurs
3. Déclencheur sur vue

- La syntaxe d'un déclencheur sur vue est identique à celle du déclencheur sur table, à la différence que la clause INSTEAD OF est ajoutée
- Ce type de déclencheur est particulier dans la mesure ou son exécution remplace celle de la commande à laquelle il est associé
- **Ce type de déclencheur n'est définissable que sur les vues et lui seul peut être mis en place sur les vues**

- Exemple : Nous mettons à la disposition de certains utilisateurs une vue permettant de sélectionner les employés qui ont le job CLERK

```
CREATE OR REPLACE VIEW
    VW_EMP_CLERK AS
Select empno "Numéro", ename "Nom",
    deptno "Dept.", sal "Salaire"
From EMP Where JOB = 'CLERK';
Vue créée.
SQL> select * from VW_EMP_CLERK ;
Numéro Nom Dept. Salaire
---------- ---------- ---------- ----------
7369 SMITH 20 880
7876 ADAMS 20 1210
2 ligne(s) sélectionnée(s).
```


II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

3- Déclencheurs
3. Déclencheur sur vue

- A travers cette vue, ces utilisateurs peuvent insérer des lignes :

```
Insert into VW_EMP_CLERK values( 9994,
    'Schmoll', 20, 2500 ) ;
1 ligne créée.
```

- Cependant, ils ne peuvent pas voir leurs insertions car la colonne job (inutile dans ce cas) ne fait pas partie de la vue et donc de l'insertion !

```
SQL> select * from VW_EMP_CLERK ;
Numéro Nom Dept. Salaire
---------- ---------- ---------- ----------
7369 SMITH 20 880
7876 ADAMS 20 1210
2 ligne(s) sélectionnée(s)
```

- Exemple : Nous allons donc créer un déclencheur sur vue qui va résoudre ce problème :

```
CREATE OR REPLACE TRIGGER
    TRG_BIR_VW_EMP_CLERK
 INSTEAD OF INSERT -- à la place de
    l'insertion
ON VW_EMP_CLERK
FOR EACH ROW
Begin
Insert into EMP ( empno, ename, deptno, sal,
    job ) -- on valorise la colonne JOB
Values (:NEW."Numéro", :NEW."Nom",
    :NEW."Dept.", :NEW."Salaire", 'CLERK' ) ;
End ;
/Déclencheur créé.
```

– L'utilisateur peut désormais visualiser ses insertions

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

3- Déclencheurs
4. Déclencheur sur évènement

- depuis la version Oracle8i, il est désormais possible d'utiliser des déclencheurs pour suivre les changements d'état du système ainsi que les connexions / déconnexions utilisateur et la surveillance des ordres description ou manipulation de données
- 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
- **Les attributs :**
  - **ora_client_ip_adress**
  - **ora_database_name**
  - **ora_des_encrypted_password**
  - **ora_dict_obj_name**
  - **ora_dict_obj_name_list**
  - **ora_dict_obj_owner**
  - **ora_dict_obj_owner_list**
  - **ora_dict_obj_type**
  - **ora_grantee**
  - **ora_instance_num**
  - **ora_is_alter_column**
  - **ora_is_creating_nested_table**
  - **ora_is_drop_column**
  - **ora_is_servererror**
  - **ora_login_user**
  - **ora_privileges**
  - **ora_revokee**
  - **ora_server_error**
  - **ora_sysevent**
  - **ora_with_grant_option**

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

3- Déclencheurs
4. Déclencheur sur évènement

- Evènements systèmes

  CREATE TRIGGER nom_déclencheur {BEFORE|AFTER} évènement_système ON{DATABASE|SCHEMA} bloc PL/SQL
- *Les vènements systèmes*
  - **STARTUP**
  - · **SHUTDOWN**
  - **SERVERERROR**
- Evènements utilisateurs

  CREATE TRIGGER nom_déclencheur {BEFORE|AFTER} évènement_utilisateur ON{DATABASE|SCHEMA} bloc PL/SQL

Les vènements utilisateurs
- **LOGON**
- **LOGOFF**
- **CREATE**
- **ALTER**
- **DROP**
- **ANALYZE**
- **ASSOCIATE STATISTICS**
- **AUDIT**
- **NOAUDIT**
- **COMMENT**
- **DDL**
- **DISSOCIATE STATISTICS**
- **GRANT**
- **RENAME**
- **REVOKE**
- **TRUNCATE**

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

3- Déclencheurs
5. Entretien des déclencheur

- **Activation/désactivation d'un déclencheur.**
- Il est possible de désactiver un déclencheur avec la commande suivante
- **ALTER TRIGGER nom_déclencheur DISABLE**
- et de l'activer avec la commande suivante
- **ALTER TRIGGER nom_déclencheur ENABLE**
- De la même façon, on peut désactiver tous les déclencheurs définis sur une table

- **ALTER TABLE nom_table DISABLE ALL TRIGGERS**
- et de les activer avec la commande suivante
  - **ALTER TABLE nom_table ENABLE ALL TRIGGERS**
    - » Les informations sur les déclencheurs sont visibles à travers les vues du dictionnaire de données
  - **USER_TRIGGERS** pour les déclencheurs appartenant au schéma
  - **ALL_TRIGGERS** pour les déclencheurs appartenant aux schémas accessibles
  - **DBA_TRIGGERS** pour les déclencheurs appartenant à tous les schémas

II- PROGRAMMATION INTEGREE : LANGAGE PL SQL

3- Déclencheurs
5. Entretien des déclencheur

- La colonne BASE_OBJECT_TYPE permet de savoir si le déclencheur est basé sur une table, une vue, un schéma ou la totalité de la base
- La colonne TRIGGER_TYPE permet de savoir s'il s'agit d'un déclencheur BEFORE, AFTER ou INSTEAD OF si son mode est FOR EACH ROW ou non
- s'il s'agit d'un déclencheur évènementiel ou non
- La colonne TRIGGERING_EVENT permet de connaître l'évèvement concerné par le déclencheur
- La colonne TRIGGER_BODY contient le code du bloc PL/SQL

- **ALTER TABLE nom_table DISABLE ALL TRIGGERS**
- et de les activer avec la commande suivante
  - **ALTER TABLE nom_table ENABLE ALL TRIGGERS**
    - » Les informations sur les déclencheurs sont visibles à travers les vues du dictionnaire de données
  - **USER_TRIGGERS** pour les déclencheurs appartenant au schéma
  - **ALL_TRIGGERS** pour les déclencheurs appartenant aux schémas accessibles
  - **DBA_TRIGGERS** pour les déclencheurs appartenant à tous les schémas