Cours 1 : Introduction au langage PL/SQL



Auteur : Riadh ZAAFRANI

2ème année SI
Janvier 2010

Présentation de PL/SQL

- **PL/SQL** (pour **PROCEDURAL LANGUAGE extensions to SQL**) est un langage procédural d'Oracle corporation étendant SQL.
- Il s'agit d'un langage de 3éme génération assimilable à un langage de programmation au dessus d'un langage de requêtes.
- Il permet de combiner les avantages d'un langage de programmation classique avec les possibilités de manipulation de données offertes par SQL.

Présentation de PL/SQL

- Le langage **PL/SQL,** étend SQL en lui ajoutant des éléments tels que :
- **Les variables et les types.**
- **Les structures de contrôle et les boucles.**
- **Les procédures et les fonctions.**
- **Les types d'objets et les méthodes.**
- Ce ne sont plus des ordres SQL qui sont transmis un à un au moteur de base de données Oracle, mais un bloc de programmation.

Présentation de PL/SQL

- Le langage **PL/SQL,** est simple d'apprentissage et de mise en œuvre.
- Sa syntaxe claire offre une grande lisibilité en phase de maintenance de vos applications.
- **PL/SQL** est un langage structuré.
- Les programmes PL/SQL sont écrits sous forme de blocs de code définissant plusieurs sections comme la **déclaration de variables**, le **code exécutable** et la **gestion des erreurs**.

Structure d'un programme PL/SQL

- Le code PL/SQL peut être stocké dans la base sous forme d'un **sous-programme** doté d'un nom ou il peut être codé directement dans SQL*Plus en tant que **"bloc de code anonyme"**, c'est-à-dire sans nom.
- Lorsqu'il est stocké dans la base, le sous-programme inclut une **section d'en-tête** dans laquelle il est nommé, mais qui contient également la déclaration de son type et la définition d'arguments optionnels.

Structure d'un programme PL/SQL

- La structure de base d'un programme PL/SQL a généralement la forme suivante:
- **[DECLARE]**
  /* section de déclaration */
  BEGIN
  /* corps du bloc de programme
  Il s'agit de la seule zone
  dont la présence est obligatoire */
  [EXCEPTION]
  /* gestion des exceptions */
  END;

Structure d'un programme PL/SQL

- Le corps du programme (entre le **BEGIN** et le **END**) contient des instructions **PL/SQL** (assignements, boucles, appel de procédure) ainsi que des instructions SQL.
- Il s'agit de la seule partie qui soit **obligatoire**.
- Les deux autres zones, dites **zone de déclaration** (définition et initialisation des structures et des variables utilisés dans le bloc) et zone de gestion des exceptions (des erreurs) sont **facultatives**.

Structure d'un programme PL/SQL

- Les seuls ordres SQL que l'on peut trouver dans un bloc PL/SQL sont les instructions de type **Langage de Manipulation de Données (LMD)** : SELECT, INSERT, UPDATE, DELETE.
- Les autres types d'instructions (par exemple CREATE, DROP, ALTER) ne peuvent se trouver qu'a l'extérieur d'un tel bloc.

Structure d'un programme PL/SQL

- **PL/SQL** ne se soucie pas de la casse (majuscule vs. minuscule).
- On peut inclure des commentaires par**--** (en début de chaque ligne commentée) ou par **/* .... */** (pour délimiter des blocs de commentaires).

Structure d'un programme PL/SQL

- Chaque instruction se termine par un « **;** ».
- Lorsque vous exécutez une instruction SQL dans **SQL*Plus**, elle se termine par un point-virgule.
- Il ne s'agit que de la terminaison de l'instruction, non d'un élément qui est constitutif.
- A la lecture du point-virgule, **SQL*Plus** est informé que l'instruction est complète et l'envoie à la base de données.

Structure d'un programme PL/SQL

- Dans un bloc **PL/SQL**, tout au contraire, le point-virgule n'est pas un simple indicateur de terminaison, mais fait partie de la syntaxe même du bloc.
- Lorsque vous spécifiez le mot-clé **DECLARE** ou **BEGIN**, **SQL*Plus** détecte qu'il s'agit d'un bloc PL/SQL et non d'une instruction SQL. Il doit cependant savoir quand se termine le bloc.
- La barre oblique « **/** », raccourci de la commande SQL*Plus **RUN**, lui en fournit l'indication.

Structure d'un programme PL/SQL

- **SQL> begin**
  Delete Employés WHERE NUMEMPLOYÉ = 2;
  INSERT INTO Employés
  Values(2,1,'Chater','Med','Agent',
  To_Date('10/10/1970'),To_Date('10/10/2000'));
  COMMIT;
  end;
  /
  Procédure PL/SQL terminée avec succès.

Bloc imbriqué

- PL/SQL permet d'imbriquer ou d'encapsuler des blocs anonymes dans d'autres blocs PL/SQL.

```
DECLARE
...
BEGIN
...
    DECLARE
    ...
    BEGIN
    ...
    EXCEPTION
    ...
    END;
...
EXCEPTION
....
END;
```

Bloc imbriqué

- On peut également imbriquer des blocs anonymes dans d'autres blocs anonymes à plusieurs niveaux.
- Un bloc PL/SQL imbriqué à l'intérieur d'un autre bloc PL/SQL peut être appelé : **Bloc imbriqué**, **Bloc secondaire**, **Bloc enfant** ou **sous-bloc**.
- Un bloc PL/SQL qui appelle un autre bloc PL/SQL peut être appelé **bloc principal** ou **bloc parent**.
- Le principal avantage, et l'une des raisons d'utiliser, du bloc imbriqué est qu'il fournit une portée à tous les objets et à toutes les commandes de ce bloc. Cette portée permet de contrôler les actions effectuées par votre programme.

Sortie à l'écran

- Le langage **PL/SQL** ne dispose d'aucune **gestion intégrée des entrées/sorties**.
- Il s'agit en fait d'un choix de conception, car l'affichage des valeurs de variables ou de structures de données n'est pas une fonction utile à la manipulation des données stockées dans la base.
- La possibilité de gérer les sorties a toutefois été introduite, sous la forme d'une application intégrée **DBMS_OUTPUT**.

Sortie à l'écran

- L'application **DBMS_OUTPUT** permet d'envoyer des messages depuis un bloc **PL/SQL**. La procédure **PUT_LINE** de cette application permet de placer des informations dans un tampon qui pourra être lu par un autre bloc PL/SQL.
- Si la récupération et l'affichage des informations placées dans le tampon ne sont pas gérés et si l'exécution ne se déroule pas sous **SQL*Plus**, alors les informations sont ignorées.
- Le principal intérêt de ce package est de faciliter la mise au point des programmes.

Sortie à l'écran

- **Oracle Entreprise Manager** et **SQL*Plus**, possèdent le paramètre **SERVEROUTPUT** qu'il faut activer à l'aide de la commande **SET SERVEROUTPUT ON** pour connaître les informations qui ont été écrites dans le tampon après l'exécution d'une commande INSERT, UPDATE, DELETE, d'une fonction, d'une procédure ou d'un bloc PL/SQL anonyme.
- Le script qui crée **DBMS_OUTPUT** accorde au groupe **PUBLIC** la permission **EXECUTE** sur cette application et crée un synonyme public pour ce dernier.

Sortie à l'écran : Exemple

```
SET SERVEROUTPUT ON
DECLARE
BEGIN
dbms_output.put_line('Bonjour utilisateur : ' || User || '
Aujourd''hui est le : ' || to_char(sysdate,'dd month yyyy'));
dbms_output.put_line(uid);
dbms_output.put_line(user);
dbms_output.put_line(sysdate);
end;
/
```

```
Bonjour utilisateur : ZAAFRANI Aujourd'hui est le : 26 février
2008
68
ZAAFRANI
26/02/08
Procédure PL/SQL terminée avec succès.
```

Sortie à l'écran

- Dans l'exemple précédent, vous pouvez remarquer que, dans le bloc **PL/SQL**, il y a quatre ordres qui se terminent par un point virgule.
- La procédure **PUT_LINE** accepte comme argument, soit une expression de type chaîne de caractères, soit une expression numérique ou une expression de type date.

Variables et types PL/SQL

- Dans un programme **PL/SQL** vous avez besoin de manipuler des chaînes de texte, nombres, valeurs booléennes, enregistrements, tableaux, dates, etc.
- La manipulation est possible grâce à des conteneurs pour ces valeurs de travail, ces conteneurs sont **des variables**.

Variables et types PL/SQL

- L'utilisation des variables est diverse; elles peuvent servir à stocker des données récupérées dans les colonnes de tables, ou à conserver des résultats de calculs internes au programme.
- Les variables peuvent être **scalaires** (une valeur simple) ou **composées** (de valeurs ou composants divers).
- La variable est une **zone mémoire nommée** permettant de stocker une valeur, elle est définie par son nom, son type et sa valeur.

Variables et types PL/SQL

- Le **nom d'une variable PL/SQL** doit respecter les conditions suivantes :
- La longueur ne doit pas dépasser **30 caractères**.
- Il est composé des lettres A..Z et a..z, chiffres 0..9, $, _ ou #.
- Il doit commencer par une lettre, mais peut être suivi par un des caractères autorisés.
- Il n'est pas un mot réservé.
- **Attention :** Les variables doivent être obligatoirement déclarées avant leur utilisation.

Variables et types PL/SQL

- Toutes les constantes et toutes les variables ont un **type**. Le type de données définit le format de stockage, les restrictions d'utilisation de la variable, et les valeurs qu'elle peut prendre.
- On trouve trois sortes de types de variables en PL/SQL :
  - Un des **types utilisés en SQL** pour les colonnes de tables.
  - Un **type particulier à PL/SQL**.
  - Un **type faisant référence à celui d'une (suite de) colonne(s) d'une table**.

Types de données scalaires

- Les types scalaires se répartissent en quatre catégories :
- **- Les types numériques** :
- **REAL**, **INTEGER**, **NUMBER** (précision de 38 chiffres par défaut), **NUMBER(x)** (nombres avec x chiffres de précision), **NUMBER(x,y)** (nombres avec x chiffres de précision dont y après la virgule), **BINARY_INTEGER** nombre entier signé.

Types de données scalaires

- **- Les types alphanumériques : CHAR(x)** (chaîne de caractères de longueur fixe x), **VARCHAR(x)** (chaîne de caractères de longueur variable jusqu'a x), **VARCHAR2** (idem que précédent excepté que ce type supporte de plus longues chaînes et que l'on n'est pas obligé de spécifier sa longueur maximale).
- **- Les types date/heure** (type **DATE**) sous différents formats.
- **- Le type booléen** (type **BOOLEAN**) stocke des valeurs logiques TRUE, FALSE ou la valeur NULL.

Les variables de substitution

- **SQL*Plus** prévoit les variables de substitution qui sont utiles pour recevoir les entrées utilisateur et stocker des informations à travers plusieurs exécutions successives.
- Dans une instruction SQL, les variables de substitution sont introduites par le caractère **&**.
- Avant l'envoi de l'instruction SQL au serveur, **SQL*Plus** effectuera une substitution textuelle complète de la variable.
- Il est à remarquer qu'aucune mémoire n'est effectivement allouée aux variables de substitution.

Les variables de substitution

- La définition des variables de substitution peut être réalisée de trois manières :
  - Préfixer une variable par un simple **&**.
  - Préfixer une variable par un double **&&**.
  - Utiliser les commandes **DEFINE** et **ACCEPT**.
- Il existe deux types de variables de substitution :
  - **&** : pour une **variable temporaire**, doivent être introduites à chaque utilisation.
  - **&&** : pour une **variable permanente**, ne sont introduites que lors de la première utilisation.

Les variables de substitution : Exemples

- Soit la table station composée des 4 enregistrements suivants :
- **SQL>** select * from station;

NOMSTATION	CAPACITÉ	LIEU	RÉGION	TARIF
Venusa	350	Guadeloupe	Antilles	1200
Farniente	200	Seychelles	Océan Indien	1500
Santalba	150	Martinique	Antilles	2000
Passac	400	Alpes	Europe	1000

- Dans l'exemple suivant la variable **&var_tarif** doit être renseignée à chaque utilisation par contre la variable permanente **&&var_région** n'est renseignée que lors de la première utilisation.

Exemple : 1ère exécution

- **SQL>** select nomstation, capacité, lieu
  2 from station
  3 where région = &&var_région and
  4 tarif = &var_tarif;
- **Entrez une valeur pour var_région :** 'Antilles'
  ancien 3 : where région = &&var_région and
  nouveau 3 : where région = 'Antilles' and
- **Entrez une valeur pour var_tarif :** 1200
  ancien 4 : tarif = &var_tarif
  nouveau 4 : tarif = 1200

NOMSTATION	CAPACITÉ	LIEU
Venusa	350	Guadeloupe

Exemple : 2ème exécution

- **SQL>** select nomstation, capacité, lieu
  2 from station
  3 where région = &&var_région and
  4 tarif = &var_tarif;
- **ancien 3 : where région = &&var_région and**
  nouveau 3 : where région = 'Antilles' and
  Entrez une valeur pour var_tarif : 2000
- **ancien 4 : tarif = &var_tarif**
  nouveau 4 : tarif = 2000

NOMSTATION	CAPACITÉ	LIEU
Santalba	150	Martinique

Les variables de substitution : Exemples

- Dans l'exemple suivant, la variable de substitution remplace toute une partie de l'ordre SQL (Elle porte bien son nom).

SQL> select nomstation, capacité, lieu
2 from station
3 &var_substitution;

Entrez une valeur pour var_substitution : where région = 'Antilles'
ancien 3 : &var_substitution
nouveau 3 : where région = 'Antilles'

NOMSTATION	CAPACITÉ	LIEU
Venusa	350	Guadeloupe
Santalba	150	Martinique

Les variables de substitution : Exemples

- Pour éviter l'affichage de vérification de substitution, l'utilisateur peut activer ou désactiver cette option par la commande suivante :
  SET VERIFY [ON | OFF]
- **SQL>** SET VERIFY OFF
- **SQL>** select nomstation, capacité, lieu
  2 from station
  3 where tarif = &var_tarif;
- **Entrez une valeur pour var_tarif :** 1000

NOMSTATION	CAPACITÉ	LIEU
Passac	400	Alpes

Définition des variables de substitution avec ACCEPT

- La commande ACCEPT permet de lire une valeur entrée par un utilisateur et de stocker la valeur saisie dans une variable à l'aide de la syntaxe suivante :
- **ACC[EPT] nom_variable {NUM[BER] | CHAR | DATE} [PROMPT "Invite :" [HIDE]]**
- **Nom_variable** Nom de la variable dans laquelle vous voulez stocker une valeur.
- **PROMPT** Texte affiché à l'écran avant de saisir la valeur de la variable.
- **HIDE** L'option permet de supprimer la visualisation sur l'écran quand l'utilisateur tape sur son clavier, généralement utilisée pour saisir un mot de passe.

Définition des variables de substitution avec ACCEPT

- **SQL>** ACCEPT var_tarif NUMBER PROMPT "Entrez le tarif :"
- **Entrez le tarif : 1500**
- **SQL>** ACCEPT var_région CHAR PROMPT "Entrez la région :"
- **Entrez la région :**'Océan Indien'
- **SQL>** select nomstation, capacité, lieu
  2 from station
  3 where région = &&var_région and tarif = &var_tarif;

NOMSTATION	CAPACITÉ	LIEU
Farniente	200	Seychelles

- On initialise les variables var_région et var_tarif à l'aide la commande **SQL*Plus ACCEPT**. A l'exécution de la requête, les variables sont déjà renseignées et sont remplacées automatiquement.

Définition des variables de substitution avec DEFINE

- La création d'une variable à l'aide de la commande **DEFINE** a la syntaxe suivante :
- **DEF[INE] nom_variable = "valeur_texte"**
- **Nom_variable** Nom de la variable dans laquelle vous voulez stocker une valeur.
- **Valeur_texte** Une valeur de type CHAR affectée à la variable. La variable créée est obligatoirement de type texte.
- Pour annuler la déclaration d'une variable, vous pouvez quitter **SQL*Plus** ou utiliser la commande **UNDEFINE** avec la syntaxe :
- **UNDEF[INE] nom_variable**

Définition des variables de substitution avec DEFINE

- **SQL>** DEFINE var_tarif = 1500
- **SQL>** select nomstation, capacité, lieu
  2 from station where tarif = &var_tarif;

NOMSTATION	CAPACITÉ	LIEU
Farniente	200	Seychelles

- **SQL>** UNDEFINE var_tarif
- **SQL>** select nomstation, capacité, lieu
  2 from station where tarif = &var_tarif;
- **Entrez une valeur pour var_tarif :** 1000

NOMSTATION	CAPACITÉ	LIEU
Passac	400	Alpes

Déclaration de variables dans le bloc DECLARE de PL/SQL

- Les variables sont déclarées dans la section **DECLARE** du bloc **PL/SQL** à l'aide de la syntaxe suivante :
- **Nom_Variable [CONSTANT] TYPE [NOT NULL] [{DEFAULT | :=} VALEUR];**
- **Nom_Variable** doit être unique dans le bloc.
- **CONSTANT** La variable est une constante. Sa valeur ne change plus dans le bloc
- **NOT NULL** La variable doit être automatiquement renseignée, sinon une erreur est affichée à la compilation du bloc.
- **:= VALEUR** La variable est affectée avec **VALEUR**. Il faut respecter le type et la précision de la variable.

Déclaration de variables

- 1er exemple :
- **DECLARE**
  utilisateur_id NUMBER := UID;
  utilisateur VARCHAR 2(12) := USER;
  date_du_jour DATE := SYSDATE;
  BEGIN
  dbms_output.put_line('L''identificateur utilisateur : ' ||uid || ' Utilisateur : ' || Utilisateur || ' Aujourd'hui : ' || date_du_jour);
  utilisateur := ' Karim Hmam ';
  dbms_output.put_line('Utilisateur : ' || Utilisateur);
  end;
  /
- **L'identificateur utilisateur : 68 Utilisateur : ZAAFRANI Aujourd'hui : 26/02/08**
- **Utilisateur : Karim Hmam**

Déclaration de variables

- 2ème exemple :
- **DECLARE**
  utilisateur CONSTANT VARCHAR2(12) DEFAULT 'Karim Hmam';
  BEGIN
  utilisateur := USER;
  dbms_output.put_line('Utilisateur : ' || Utilisateur);
  end;
  /
- **utilisateur := USER;**
  *****
  ERREUR à la ligne 4 : ORA-06550: Ligne 4, colonne 5 : PLS-00363: expression 'UTILISATEUR' ne peut être utilisée comme cible d'affectation
- La variable utilisateur est **CONSTANT** et ne peut pas être modifiée.

Déclaration de variables

- 3ème exemple :
- **DECLARE**
  utilisateur_id NUMBER NOT NULL;
  BEGIN
  utilisateur_id := 13;
  dbms_output.put_line('Utilisateur : ' || Utilisateur_id);
  end;
  /
- **utilisateur_id NUMBER NOT NULL;**
  *****
  ERREUR à la ligne 2 : ORA-06550: Ligne 2, colonne 23 : PLS-00218: une variable déclarée NOT NULL doit avoir une affectation d'initialisation
- L'option **NOT NULL** implique automatiquement, pour une variable l'affectation dans sa déclaration.

Les variables de liaison

- **SQL*Plus** prévoit en plus des variables de substitution, les **variables de liaison**, qui sont utiles pour recevoir les entrées utilisateur et stocker des informations à travers plusieurs exécutions successives.
- En effet, aucune mémoire n'est allouée aux variables de substitution. **SQL*Plus** peut néanmoins allouer un espace mémoire sous forme d'une variable de liaison, dont le contenu est utilisable à l'intérieur d'un **bloc PL/SQL** ou d'une **instruction SQL**.

Les variables de liaison

- Etant donné que l'espace alloué est extérieur au bloc, son contenu peut être utilisé successivement par plusieurs blocs ou instructions et faire l'objet d'un affichage en fin de traitement.
- L'allocation d'une variable de liaison est réalisée au moyen de la commande **VARIABLE** de **SQL*Plus**.
- Sachez que celle-ci n'est valide **qu'à partir de l'invite de commande** de **SQL*Plus** et pas à l'intérieur d'un bloc **PL/SQL**.

Les variables de liaison

- A l'intérieur du **bloc PL/SQL**, la variable de liaison est introduite par le signe **« : »**.
- **SQL>** VARIABLE utilisateur varchar2(12);
- **SQL>** begin
  2 :utilisateur := user;
  3 dbms_output.put_line(:utilisateur);
  4 end;
  5 /
  ZAAFRANI
- **Procédure PL/SQL terminée avec succès.**

Les variables de liaison

- La commande **PRINT** affiche la valeur de la variable après exécution du bloc.
- **SQL>** PRINT utilisateur
  UTILISATEUR
  ZAAFRANI
- Dans cet exemple, vous pouvez remarquer que la variable de liaison **utilisateur** bénéficie d'une allocation mémoire dans laquelle on peut stocker une valeur de même type que la variable, en l'occurrence le nom de l'utilisateur.

Les variables de liaison

- L'exemple suivant montre la différence entre une **variable de liaison** et une **variable de substitution**, à savoir qu'une variable de substitution **n'a pas d'allocation mémoire**.
- **SQL>** VAR v_liaison VARCHAR2(20);
- **SQL>** declare
  2 v_plsql VARCHAR2(20) := 'Tintin';
  3 begin
  4 :v_liaison := v_plsql;
  5 &&v_substitution := USER;
  6 dbms_output.put_line('v_plsql ='||v_plsql);
  7 dbms_output.put_line('v_liaison ='||:v_liaison);
  8 dbms_output.put_line
  ('&&v_substitution='||&&v_substitution);
  9 end;
  10 /

Les variables de liaison

- **Entrez une valeur pour v_substitution :**v_plsql
- **ancien 5 : &&v_substitution := USER;**
  nouveau 5 : v_plsql := USER;
  ancien 8 : dbms_output.put_line
  ('&&v_substitution='||&&v_substitution);
  nouveau 8 :
  dbms_output.put_line('v_plsql='||v_plsql);
  v_plsql =ZAAFRANI
  v_liaison =Tintin
  v_plsql=ZAAFRANI

Les variables de liaison

- La variable de substitution est initialisée avec la pseudo colonne **USER**.
- A l'exécution du bloc **PL/SQL**, l'environnement demande la valeur de la variable de substitution **v_substitution** : toute occurrence de cette variable dans le bloc est remplacée par la chaîne de substitution **v_plsql**.
- Vous pouvez remarquer, que même la chaîne de caractères **&&v_substitution** de la troisième opération d'affichage put_line est remplacée.
- En conclusion, une **variable de substitution** est un moyen simple de remplacer des parties du code par une saisie utilisateur.

Visibilité des variables PL/SQL

- La **portée d'une variable** est la partie du programme dans laquelle vous pouvez faire référence à cette variable.
- La visibilité d'une variable porte sur le bloc où elle a été déclarée, et dans tous les blocs imbriqués si le nom n'a pas été utilisé pour une déclaration.

Visibilité des variables PL/SQL

```
SQL> VARIABLE utilisateur varchar2(50);
SQL> declare
  2    utilisateur varchar2(50) := 'Bloc principal :' || USER;
  3  begin
  4    :utilisateur := 'Variable de liaison :' || USER;
  5    declare
  6         utilisateur varchar2(50) := 'Premier Bloc imbriqué :' || USER;
  7    begin
  8         declare
  9            utilisateur varchar2(50) := 'Deuxième Bloc imbriqué :' ||
USER;
 10         begin
 11            dbms_output.put_line(utilisateur);
 12            dbms_output.put_line(:utilisateur);
 13         end;
 14         dbms_output.put_line(utilisateur);
 15    end;
 16    dbms_output.put_line( utilisateur) ;
 17  end;
```

Visibilité des variables PL/SQL

- **Le résultat de l'exécution :**
- **Deuxième Bloc imbriqué :ZAAFRANI**
  Variable de liaison :ZAAFRANI
  Premier Bloc imbriqué :ZAAFRANI
  Bloc principal :ZAAFRANI
- Vous pouvez voir la création des 3 blocs imbriqués. Dans chaque bloc, il y a une déclaration et un affichage de la variable utilisateur.
- La variable utilisateur affichée dans chaque bloc est la variable définie dans le bloc respectif.
- Vous pouvez remarquer aussi que la variable de liaison peut être utilisée même dans les blocs imbriqués.

Visibilité des variables PL/SQL

```
SQL> declare
  2    date_du_jour date;
  3  begin
  4    begin
  5      date_du_jour := sysdate;
  6    end;
  7    dbms_output.put_line(date_du_jour);
  8  end;
  9  /
28/02/07
```

- La variable **date_du_jour** est définie dans le bloc principal et elle peut être référencée dans le bloc secondaire.

Types définis par l'utilisateur

- Dans le langage **PL/SQL**, il est possible de définir des **types de données dérivés** des types prédéfinis.
- Un **type dérivé** est une déclinaison d'un type original, qui en prend les règles mais peut en restreindre le domaine de valeurs.
- Il y a 2 catégories de types dérivés :
  - Les **types bornés**
  - Les **types non bornés**

Types définis par l'utilisateur

- Un **type dérivé borné** restreint le domaine des valeurs autorisées par le type original.
- Un **type dérivé non borné** ne restreint pas le domaine des valeurs possibles du type original.
- **SUBTYPE NOM_SUBTYPE IS**
  TYPE_BASE [(CONSTRAINT)]
  [NOT NULL];

Types définis par l'utilisateur

```
SQL> declare
  2      SUBTYPE Numeral IS NUMBER(1,0);
  3      x Numeral;
  4  BEGIN
  5      x :=10;
  6  END;
  7  /
```

- **Declare**
  *****

ERREUR à la ligne 1 : ORA-06502: PL/SQL : erreur numérique ou erreur sur une valeur: précision de NUMBER trop élevée

- Vous pouvez remarquer que le type dérivé **Numéral** ne peuvent contenir que des valeurs entre –9 et 9, l'affectation déclenche une erreur.

Les enregistrements

- Le langage **PL/SQL** connaît deux types composés : **TABLE** et **RECORD**.
- Leur utilisation est particulière. Ils doivent tout d'abord faire l'objet d'une déclaration préalable de type de données.
- Ensuite seulement, une **table** ou un **record PL/SQL** peuvent être déclarés comme correspondant au type en question.

Les enregistrements

- Un **enregistrement** est une structure de données composée, ce qui signifie qu'il comprend plus d'un élément ou composant, avec chacun une valeur propre; il permet de stocker des données et d'y accéder en tant que groupe.
- Pour déclarer un enregistrement, on doit :
  - Déclarer ou définir un **TYPE** d'enregistrement comprenant la structure voulue pour l'enregistrement.
  - Utiliser ce **TYPE** d'enregistrement comme base de déclaration des enregistrements de même structure.

Les enregistrements

- On déclare le type d'**enregistrement** avec l'ordre **TYPE**.
- L'ordre **TYPE** définit le nom de la nouvelle structure d'enregistrement, et les éléments ou zones qui composent cet enregistrement :
- **TYPE NOM_TYPE IS RECORD (NOM_CHAMP1 TYPE [NOT NULL] [:= EXPRESSION1], [,…]);**

Les enregistrements

- Une fois que l'on a créé ses propres types d'enregistrement, on peut les utiliser pour déclarer des enregistrements spécifiques.
- La déclaration de l'enregistrement réel possède le format suivant :
- **NOM_ENREGISTREMENT TYPE_ENREGISTRMENT;**
- Les champs d'une variable de type enregistrement peuvent être référencés à l'aide de l'opérateur **(.)**.

Les enregistrements

```
SQL> declare
  2    TYPE adresse IS RECORD ( ADRESSE
VARCHAR2(60),
  3                    VILLE      VARCHAR2(15),
  4                    CODE_POSTAL  VARCHAR2(10));
  5    TYPE employé IS RECORD ( NOM        VARCHAR2(20),
  6                    PRENOM     VARCHAR2(10),
  7                    adr_emp    adresse    );
  8  mon_employé employé;
  9  begin
 10    mon_employé.NOM          := 'HMAM';
 11    mon_employé.PRENOM       := 'Karim';
 12    mon_employé.adr_emp.ADRESSE  := '12, Rue Assad Ibn
Fourat';
 13    mon_employé.adr_emp.VILLE    := 'KAIROUAN';
 14    mon_employé.adr_emp.CODE_POSTAL  := '3100';
```

Les enregistrements

```
 15  dbms_output.put_line(mon_employé.NOM     || ' ' ||
 16        mon_employé.PRENOM                 || ' ' ||
 17        mon_employé.adr_emp.ADRESSE        || ' ' ||
 18        mon_employé.adr_emp.CODE_POSTAL || ' ' ||
 19        mon_employé.adr_emp.VILLE                 );
 20  end;
 21  /
HMAM Karim 12, Rue Assad Ibn Fourat 3100
KAIROUAN
```

- Cet exemple montre la création d'un enregistrement **adresse** qui a son tour est utilisé comme type de base pour un des champs du deuxième enregistrement **employé**.

Les tableaux

- Les tableaux sont conçus comme les tables de la base de données.
- Ils possèdent une **clé primaire (index)** pour accéder aux lignes du tableau.
- Un **tableau**, comme une table, ne possède pas de limite de taille. De cette façon, le nombre d'éléments d'un tableau va croître dynamiquement.
- La colonne peut être de n'importe quel type scalaire, mais l'index doit être du type **BINARY_INTEGER.**

Les tableaux

- Pour déclarer un tableau, on doit :
  - Déclarer ou définir un **TYPE** de tableaux.
  - Utiliser ce **TYPE** de tableau comme base de déclaration des tableaux.
- Vous pouvez déclarer un type **TABLE** dans la partie déclarative d'un bloc, d'un sous-programme ou d'un package :
- **TYPE NOM_TYPE IS TABLE OF TYPE [NOT NULL] INDEX BY BINARY_INTEGER;**

Les tableaux

- Lorsque le type est déclaré, vous pouvez déclarer des tableaux de ce type :
- **NOM_TABLE TYPE_TABLE;**
- Pour accéder à un élément du tableau, vous devez spécifier une valeur d'index :
- **NOM_TABLE(VALEUR_INDEX);**

Les tableaux

```
SQL> declare
  2 TYPE mon_type_tableau IS TABLE OF
  VARCHAR2(20)
  3                 INDEX BY BINARY_INTEGER;
  4    mon_tableau mon_type_tableau;
  5 begin
  6    mon_tableau(1) := 'Ligne numéro : 1';
  7    mon_tableau(2) := 'Ligne numéro : 2';
  8    mon_tableau(3) := 'Ligne numéro : 3';
  9    mon_tableau(4) := 'Ligne numéro : 4';
 10    mon_tableau(5) := 'Ligne numéro : 5';
```

Les tableaux

```
 11    dbms_output.put_line(mon_tableau(1));
 12    dbms_output.put_line(mon_tableau(2));
 13    dbms_output.put_line(mon_tableau(3));
 14    dbms_output.put_line(mon_tableau(4));
 15    dbms_output.put_line(mon_tableau(5));
 16 end;
 17 /
Ligne numéro : 1
Ligne numéro : 2
Ligne numéro : 3
Ligne numéro : 4
Ligne numéro : 5
```

- Cet exemple montre la création d'un type tableau, il est alimenté avec 5 valeurs qui sont affichées.

Les tableaux

- Les attributs et les méthodes d'un tableau sont :
- **EXISTS(n) :** Permet de tester la présence d'une valeur dans l'élément d'indice n.
- **COUNT :** Permet de compter le nombre d'éléments
- **FIRST/LAST :** Permet d'accéder au premier/dernier élément du tableau.
- **PRIOR/NEXT(n) :** Permet d'accéder à l'élément précédent/suivant de l'élément d'indice n du tableau.
- **TRIM(n) :** Supprime un ou plusieurs éléments(n) à la fin du tableau.
- **DELETE(n) :** Supprime l'élément d'indice n.

Les tableaux

- Dans l'exemple suivant vous créez un type utilisateur **NumTab** qui sera utilisé dans le bloc suivant à titre d'exemple les différentes méthodes et attributs du tableau.
- **SQL>** CREATE OR REPLACE TYPE **NumTab** AS TABLE OF NUMBER;
  2 /
  Type créé.

Les tableaux

```
SQL> declare
  2    mon_tableau NumTab := NumTab(1,2,3,4,5,6,7,8,9);
  3  begin
  4    dbms_output.put_line('count      :' || mon_tableau.count);
  5    dbms_output.put_line('first      :' || mon_tableau.first);
  6    dbms_output.put_line('last       :' || mon_tableau.last);
  7    dbms_output.put_line('Prior(3)   :' || mon_tableau.Prior(3));
  8    dbms_output.put_line('Next(3)    :' || mon_tableau.Next(3));
  9    mon_tableau.delete(2);
 10    dbms_output.put_line('count      :' || mon_tableau.count);
 11    mon_tableau(7) := NULL;
 12    mon_tableau.Trim(5);
 13    dbms_output.put_line('count      :' || mon_tableau.count);
```

Les tableaux

```
 14    dbms_output.put_line(mon_tableau(1));
 15    dbms_output.put_line(mon_tableau(3));
 16    dbms_output.put_line(mon_tableau(4));
 17 end;
 18 /
count    :9
first    :1
last     :9
Prior(3) :2
Next(3)  :4
count    :8
count    :3
1
3
4
```

Les tableaux

- Un tableau peut être défini d'une taille fixe qui doit être précisée lors de sa déclaration. Les éléments sont numérotés à partir de la valeur 1.
- Vous pouvez déclarer un type VARRAY dans la partie déclarative d'un bloc, d'un sous-programme ou d'un package en utilisant la syntaxe suivante :
- **TYPE NOM_TYPE IS VARRAY (TAILLE_MAXIMALE) OF TYPE [NOT NULL];**

Les tableaux

```
SQL> declare
  2    TYPE mon_type_tableau IS VARRAY(2) OF VARCHAR2(30);
  3    mon_tableau mon_type_tableau := mon_type_tableau('Ligne numéro : 1','Ligne numéro : 2');
  4  begin
  5  dbms_output.put_line(mon_tableau(1));
  6  dbms_output.put_line(mon_tableau(2));
  7  end;
  8  /
Ligne numéro : 1
Ligne numéro : 2
```

Les variables basées

- Le langage **PL/SQL**, donne la possibilité à la déclaration d'une variable de faire référence à une entité existante, qui a fait l'objet d'une déclaration préalable de type de données.
- On peut référencer plusieurs types d'entités existantes : **colonne**, **table**, **curseur** ou **variable**.
- L'attribut **%TYPE** permet de référencer soit une colonne d'une table, soit une variable précédemment définie :
- **NOM_VARIABLE (NOM_TABLE.COLONNE | NOM_VARIABLE)%TYPE;**

Les variables basées

```
SQL> declare
  2    date_embauche
employés.DATEEMBAUCHE%TYPE :=
ADD_MONTHS(TRUNC(SYSDATE,'MONTH'),1);
  3  begin
  4  INSERT INTO employés VALUES
  5    (10,1,'Ben Brahim','Nadra','Chef des
ventes','01/02/1968', date_embauche,1000,0);
  6  dbms_output.put_line(date_embauche);
  7  end;
  8  /
01/03/07
```

Les variables basées

- Dans l'exemple précédent, la déclaration de la variable **date_embauche** référence la colonne **dateEmbauche** de la table employés.
- La déclaration de la variable comporte aussi une affectation de la date du premier jour du mois suivant, et qui alimentera le champ **dateEmbauche** de l'employé inséré :
- **SQL>** select nom, dateembauche from employés where numemployé = 10;
- **NOM DATEEMBAUCHE**
  Ben Brahim 01/03/07

Les variables basées : %ROWTYPE

- **%ROWTYPE** permet de déclarer une variable composée qui est équivalente à une ligne dans la table spécifiée.
- Une telle variable est un enregistrement composé des noms de colonnes et des types de données référencés dans la table :
- **NOM_VARIABLE {NOM_TABLE | NOM_VARIABLE}%ROWTYPE;**

Les variables basées : %ROWTYPE

```
SQL> declare
  2     Client Clients%ROWTYPE;
  3  begin
  4     Client.CODECLIENT := '10000';
  5     Client.SOCIÉTÉ   := 'GSM';
  6     Client.ADRESSE   := '12, Rue Assad Ibn Fourat';
  7     Client.VILLE     := 'KAIROUAN';
  8     Client.CODE_POSTAL:= '3100';
  9     dbms_output.put_line(Client.CODECLIENT     || ' ' ||
 10                  Client.SOCIÉTÉ        || ' ' ||
 11                  Client.ADRESSE        || ' ' ||
 12                  Client.VILLE          || ' ' ||
 13                  Client.CODE_POSTAL );
 14  end;
 15  /
```

- **10000 GSM 12, Rue Assad Ibn Fourat KAIROUAN 3100**

La syntaxe SELECT

- Il existe deux façons d'affecter des valeurs à des variables. La première utilise l'opérateur d'assignation, le signe « **:=** ».
- La deuxième façon d'attribuer des valeurs à des variables consiste à effectuer un **SELECT** de valeurs en provenance de la base de données :
- **SELECT Expression1 [,…] INTO Variable1 [,…] FROM Nom_Table [Where PREDICAT];**
- **Attention** : L'ordre SELECT doit rapporter une seule ligne, sans quoi une erreur est générée.

La syntaxe SELECT

```
SQL> declare
  2     v_employé employés%ROWTYPE;
  3  begin
  4     SELECT * INTO v_employé FROM
employés Where numemployé = 10;
  5     dbms_output.put_line(v_employé.NOM
|| ' ' || v_employé.prénom);
  6  end;
  7  /
```

- **Ben Brahim Nadra**

Structures de contrôle PL/SQL

- Comme n'importe quel langage procédural, **PL/SQL** possède un certain nombre de structures de contrôles évoluées comme les branchements conditionnels et les boucles.
- **1) Les branchements conditionnels**
- **IF-THEN-ELSEIF :**

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

Structures de contrôle PL/SQL

- Exemple :
- **IF nomEmploye='TOTO' THEN**

```
   salaire:=salaire*2;
ELSIF salaire>10000 THEN
   salaire:=salaire/2;
ELSE
   salaire:=salaire*3;
END IF;
```

Structures de contrôle PL/SQL

- **CASE :**
- **CASE <expression>**

```
   WHEN {<valeur>|<condition>
            THEN <commandes>
   [,…]
END;
```

Structures de contrôle PL/SQL

- **SQL>** declare

```
 2   v_employé employés%ROWTYPE;
 3  begin
 4   case &&valeur
 5   when 1 then
 6      dbms_output.put_line('la valeur saisie est : 1');
 7   when 2 then
 8      dbms_output.put_line('la valeur saisie est : 2');
 9   when 3 then
10      dbms_output.put_line('la valeur saisie est : 3');
11    else
12      dbms_output.put_line('Toute autre valeur');
13    end case;
14  end;
15  /
```

- **Entrez une valeur pour valeur :** 3
  la valeur saisie est : 3

Structures de contrôle PL/SQL

- **2) boucles**
- PL/SQL admet trois sortes de boucles.
- La première est une boucle potentiellement infinie :
- **LOOP**

```
   commandes;
END LOOP;
```

- Au moins une des instructions du corps de la boucle doit être une instruction de sortie :
- **EXIT WHEN <condition>;**
- Des que la condition devient vraie (si elle le devient...), on sort de la boucle.

Structures de contrôle PL/SQL

- Le deuxième type de boucle permet de répéter un nombre défini de fois un même traitement:
- **FOR <compteur> IN [REVERSE] <limite_inf> .. <limite_sup>**
  commandes;
  END LOOP;
- Enfin, le troisième type de boucle permet la sortie selon une condition prédéfinie.
- **WHILE <condition> LOOP**
  commandes;
  END LOOP;

Structures de contrôle PL/SQL

- Toutes ces structures de contrôles sont évidemment imbriquables les unes dans les autres. Voici un même exemple traité de trois manières différentes suivant le type de boucle choisi.

```
DECLARE
   x NUMBER(3):=1;
BEGIN
    LOOP
INSERT INTO employé(noemp,nomemp,job,nodept)
 VALUES (x, 'TOTO', 'PROGRAMMEUR', 1);
  x:=x+1;
EXIT WHEN x>=100
   END LOOP;
END;
```

Structures de contrôle PL/SQL

- Deuxième exemple :

```
DECLARE
   x NUMBER(3);
BEGIN
   FOR x IN 1..100
       INSERT INTO employe(noemp,
nomemp, job, nodept)
       VALUES (x, 'TOTO',
'PROGRAMMEUR', 1);
   END LOOP;
END;
```

Structures de contrôle PL/SQL

- Troisième exemple :

```
DECLARE
x NUMBER(3):=1;
BEGIN
WHILE x<=100 LOOP
  INSERT INTO employe(noemp, nomemp, job,
nodept)
  VALUES (x, 'TOTO', 'PROGRAMMEUR', 1);
  x:=x+1;
END LOOP;
END;
```

Fin du cours 1 : Introduction au langage PL/SQL

MERCI pour votre patiente attention !

