Plan du cours

1. Introduction
2. Modèle conceptuel de données Entité-Association
3. Modèle relationnel de données
4. Le langage SQL
5. **PL/SQL**
6. Transactions

PL/SQL

1. **Introduction**
2. Structure d'un bloc PL/SQL
3. Variables (types, déclaration, affectation)
4. Structures de contrôle `(IF,CASE,LOOP,WHILE,FOR)`
5. Curseurs
6. Sous-programmes : procédures et fonctions
7. Exceptions
8. Packages
9. Triggers (CI dynamiques)

1. Introduction

- PL/SQL : langage de programmation pour SQL
  - Extension procédurale du langage SQL
  - Développement d'applications complexes autour de BD
    - Structures de contrôle (conditionnelles, itérations …)
    - Éléments procéduraux (procédures, fonctions…)
- Principaux objectifs de PL/SQL
  - Enchaîner plusieurs instructions SQL
  - Augmenter l'expressivité de SQL
  - Traiter les résultats d'une requête un tuple à la fois (curseurs)
  - Optimiser l'exécution d'ensemble de commandes SQL
  - Réutiliser le code des programmes
- Structure de base dans PL/SQL : le bloc

PL/SQL

1. Introduction
2. **Structure d'un bloc PL/SQL**
3. Variables (types, déclaration, affectation)
4. Structures de contrôle `(IF,CASE,LOOP,WHILE,FOR)`
5. Curseurs
6. Sous-programmes : procédures et fonctions
7. Packages
8. Exceptions
9. Triggers (CI dynamiques)

2. Structure d'un bloc PL/SQL

```
[entête-de-bloc ]          -- entête de bloc facultatif
[DECLARE
  constantes
  variables                -- partie déclarative facultative
  curseurs
  exceptions-utilisateur ]
BEGIN                      -- partie exécutable obligatoire
  commandes-PL/SQL
  [EXCEPTION               -- partie gestion d'erreurs facultative
  gestion-exceptions ]
END;
```

2. Structure d'un bloc PL/SQL

- **entête-de-bloc** : indique si le bloc est une procédure, une fonction, un package (module)
  - Un bloc sans entête est un **bloc anonyme**
- Commandes de SQL utilisables dans un bloc PL/SQL
  - Toutes les commandes de SQL/LMD (`SELECT, INSERT, UPDATE...`)
    - Attention ! forme particulière de la commande `SELECT` (`SELECT...INTO...`)
  - Les commandes LDD ne sont pas utilisables dans les blocs PL/SQL (create table, create view, create index, drop table...)
- **Commentaires** dans un bloc PL/SQL

```
-- ceci commentaire sur une ligne
/* ceci est un exemple de commentaire long et verbeux sur
   plusieurs lignes */
```

Exemple de bloc PL/SQL

```
DECLARE                          -- bloc anonyme
   qté_en_stock NUMBER(4);
BEGIN
   SELECT quantité INTO qté_en_stock FROM inventaire
              WHERE nom_produit = 'RAQUETTE TENNIS';
   IF qté_en_stock > 0 THEN
       UPDATE inventaire SET quantité = quantité-1
              WHERE nom_produit = 'RAQUETTE TENNIS';
       INSERT INTO ventes VALUES
              ('vente raquette tennis', SYSDATE);
   ELSE INSERT INTO ventes VALUES
              ('rupture stock raquette tennis', SYSDATE);
   END IF;
   COMMIT;
END;
```

Exemples de bloc PL/SQL

```
BEGIN                                   -- bloc anonyme
   INSERT INTO Produit VALUES(430,'lecteur DVD', 9.99);
   UPDATE Produit  SET prixUnitaire = prixUnitaire*1.05
      WHERE  libellé IN ('CD-ROM', 'DVD', 'ZIP')
    COMMIT;
END;

CREATE PROCEDURE nom_majus AS     -- bloc procédure
   BEGIN
        UPDATE client SET nom = UPPER(nom);
        COMMIT;
        DBMS_OUTPUT.PUT_LINE ('opération faite');
   END;
```

Imbrication de blocs

- Un bloc PL/SQL est une commande PL/SQL
  - Imbrication possible de blocs PL/SQL
  - Règles de visibilité des variables : idem que dans les langages à structure de bloc

Une variable est visible dans le bloc où elle est définie et dans les blocs imbriqués (dans ce bloc) sauf si elle y est redéfinie (variable de même nom).

```
DECLARE                                  -- début du 1er bloc anonyme
V1 int; V2 date;
BEGIN
      -------
      -------
      DECLARE                            -- début d'un bloc imbriqué dans le 1er
      V3 char; V1 char;          -- variable V1 redéfinie
      BEGIN
            ---
            ---
      EXCEPTION
        --
        --
      END;                               -- fin du bloc imbriqué
      --------
      --------
      --------
EXCEPTION
   --
   --
END;                                     -- fin du 1er bloc
```

V1 de type int

V1 de type char

V1 de type int

Affichage à l'écran à partir d'un bloc

- Les procédures du package DBMS_OUTPUT permettent d'écrire/lire des lignes dans un tampon (buffer) depuis un bloc PL/SQL ou une procédure
- Ces lignes peuvent être **affichées** à l'écran (sortie standard) si la variable SERVEROUTPUT est positionnée à ON
  - SET SERVEROUTPUT ON
  - variable SQL*Plus positionnée une fois pour la session
- Il n'y a pas de mécanisme similaire pour lire une donnée au clavier
- Procédures d'écriture (dans le buffer ou à l'écran)
  - PUT : ajout d'un texte sur la ligne courante
  - NEW_LINE : ajout d'un retour à ligne
  - PUT_LINE : put + new_line

Exemple : DBMS_OUTPUT.PUT_LINE('Coucou du bloc PL/SQL');

PL/SQL

1. Introduction
2. Structure d'un bloc PL/SQL
3. **Variables (types, déclaration, affectation)**
4. Structures de contrôle `(IF,CASE,LOOP,WHILE,FOR)`
5. Curseurs
6. Sous-programmes : procédures et fonctions
7. Exceptions
8. Packages
9. Triggers (CI dynamiques)

3. Variables et constantes - Déclaration

- On doit déclarer des variables et des constantes avant de les utiliser dans un bloc
- Lors de l'exécution du bloc, la valeur d'une variable peut changer contrairement à la valeur d'une constante

nom_variable type [[NOT NULL] [{DEFAULT|:= } expression];

nom_constante CONSTANT type := valeur;

- On peut affecter une valeur initiale ou une valeur par défaut à une variable
  - Dans le cas contraire, la variable est initialisée à NULL
- La contrainte NOT NULL **doit** être suivie d'une initialisation
- L'affectation d'une valeur à une constante est obligatoire

Exemples de déclaration de variables

DECLARE
qty_in_stock NUMBER(4);
birthday DATE;
employees_count SMALLINT := 0;
pi CONSTANT REAL := 3.14159;
radius REAL := 1;
area REAL := pi*radius**2;
groupe_sanguin CHAR NOT NULL DEFAULT ‘O’;
heures_travail INTEGER DEFAULT 40;
credit_maximum CONSTANT REAL := 500.00;
BEGIN

Expressions dans PL/SQL

- Opérateurs arithmétiques

  + , - , / , * ,**

- Opérateur de concaténation de chaînes

  ||

- Opérateurs de comparaisons

  = , < , > , <= , >= , <> ,

  IS NULL , LIKE , BETWEEN , IN

- Opérateurs logiques

  AND , OR, NOT

- Priorité décroissante :

  **, - (op unaire), { *, / },{+, -, || }, {=, <, … IN}, NOT,{AND, OR}

Types de données pour les variables PL/SQL (1/2)

- Types scalaires
  - Numériques
    - `NUMBER, DECIMAL, FLOAT, INTEGER, SMALLINT...`
    - `BINARY_INTEGER, PLS_INTEGER` : entiers relatifs
      - Espace de stockage : moins important que le type NUMBER
      - Opérations sur le type `PLS_INTEGER` : plus rapides que sur `BINARY_INTEGER`
  - Caractères
    - `CHAR, LONG, VARCHAR2, VARCHAR, STRING, ROWID...`
  - Logique
    - `BOOLEAN (3 valeurs : true, false, null)`
  - Date
    - `DATE , TIMESTAMP`

Types de données pour les variables PL/SQL (2/2)

- Types LOB (Large OBjects)
  - `BFILE, BLOB, CLOB, NCLOB`
- Types référence
  - `REF CURSOR, REF type_objet`
- Types composites (ou collections)
  - `RECORD,TABLE,VARRAY`

Types composites : tableaux "classiques"

```
DECLARE
TYPE   tab_entiers   IS  VARRAY(10)   OF   INTEGER ; -- taille maximale = 10
mon_tab1    tab_entiers ;
mon_tab2 tab_entiers;

BEGIN
      mon_tab2 := tab_entiers ();              -- initialisation obligatoire
      mon_tab1 := tab_entiers (4,10,7,3);  -- initialisation  avec 4 éléments
            -- mon_tab1.count   vaut  4,    mon_tab1.limit   vaut   10

      mon_tab1.extend(1);   -- on "agrandit" le tableau (5ème élément)

      mon_tab1(5) := 23 ;      -- on positionne la valeur du 5ème élément
END ;
```

Types composites : tableaux associatifs clé-valeur (1/2)

```
DECLARE
TYPE type_tableau IS TABLE OF REAL INDEX BY  BINARY_INTEGER;
                                -- définir un type tableau
T type_tableau ;                -- utiliser ce type
                                -- T est une variable de type type_tableau
S type_tableau;

BEGIN
    T(1) := 12.5;    T(2) := 5.2;
     S(1) := 12.0;   S(2) := 10.5; S(3) := 15.5; S(4) := 12.5;
    dbms_output.put_line ('cardinal de S :' || S.count);      -- (4)
    dbms_output.put_line ('1ère clé (indice) :' || S.first);    -- dans l'ordre des clés (1)
    dbms_output.put_line ('clé suivant la clé 3 :' || S.next(3));
    dbms_output.put_line ('dernière clé :' || S.last);       -- dans l'ordre des clés (4)
    dbms_output.put_line ('clé précédant la clé 2 :' || S.prior(2));
    S.delete(4);                                              -- supprime le 4ème élément de S
END;
```

Types composites : tableaux associatifs clé-valeur (2/2)

```
DECLARE
    TYPE population_type IS TABLE OF NUMBER INDEX BY VARCHAR2(64);
    continent_population population_type;
    howmany NUMBER;
    which VARCHAR2(64);

BEGIN
    continent_population('Australia') := 30000000;
    -- Chercher la valeur associée à une chaîne
    howmany := continent_population('Australia');
    continent_population('Antarctica') := 1000;          -- Crée une nouvelle entrée
    continent_population('Antarctica') := 1001;          -- Remplacement
    -- Retourne 'Antarctica' , premier dans l'ordre alphabétique
    which := continent_population.FIRST;
    -- Retourne 'Australia' , dernier dans l'ordre aphabétique
    which := continent_population.LAST;
END;
```

Types composites : record (enregistrement)

```
DECLARE
TYPE t_adresse IS RECORD   (rue varchar2(40),
                           ville varchar2(40),
                           code_postal varchar2(10));
TYPE t_personne IS RECORD  (nom varchar2(40),
                           prénom varchar2(40),
                           adresse t_adresse);
employé t_personne;
BEGIN
employé.nom := 'dupond';
employé.prénom := 'pierre' ;
employé.adresse.ville := 'nancy';
employé.adresse.code_postal := '54000';
---
END;
```

N.B. : les types TABLE et RECORD ne peuvent pas être utilisés dans le schéma d'une table dans le modèle relationnel (1ère forme normale)

Typage dynamique : %TYPE

- **%TYPE** : attribut fournissant le type d'une variable ou le type d'une colonne d'une table

```
DECLARE
crédit NUMBER(7,2);
débit crédit%TYPE;      /*variable débit de même type que crédit */
débit_bis crédit%TYPE := 0;
                                    -- initialisation possible

numéro toto.Produit.numProduit%TYPE;
/* type de l'attribut numProduit de la table  Produit appartenant à toto      */
BEGIN
…
```

Typage dynamique : %ROWTYPE

- **%ROWTYPE** : attribut fournissant le type d'un tuple d'une table, d'une vue ou d'un curseur (type record)

```
DECLARE
  employee_rec  HR.EMPLOYEES%ROWTYPE
BEGIN
  SELECT  *  INTO employee_rec
  FROM    HR.EMPLOYEES
  WHERE   emp_id = 123;
  IF (employee_rec.manager_id IS NULL) THEN
    dbms_output.put_line('salarié 123 sans chef');
END;
```

Affectation d'une valeur à une variable

nom_variable := expression

Affectation de la valeur de l'expression dans la variable ssi :

- la variable est déjà déclarée
- le type de l'expression est compatible avec celui de la variable

```
DECLARE
        total INTEGER;                          -- total initialisé à  NULL
        pi CONSTANT REAL := 3.14159;
        radius REAL := 1;          area REAL;
BEGIN
        total := total + 1;                     -- total est NULL
        area := pi*radius**2;
END;
```

Affectation de valeur à une variable de type BOOLEAN

nom_variable_booléenne := expression

`expression` est soit :

- `TRUE`, `FALSE` ou `NULL`
- une expression de condition (ou comparaison)

```
DECLARE
        fini BOOLEAN;              -- fini initialisé à NULL
        total INTEGER := 24;
BEGIN
        fini := FALSE;
        fini := (total >= 10);     -- fini est TRUE
        IF fini THEN dbms_output.put_line ('fini!');
        END IF;
END;
```

Affectation du résultat d'un SELECT à une variable (1/2)

- Il est possible de récupérer dans une(des) variable(s) le résultat d'une requête SELECT **lorsqu'il est réduit à un tuple**

```
SELECT liste_attributs  INTO liste_variables   FROM ...
```

- Si on sélectionne plusieurs colonnes, il est possible d'utiliser une variable de type **record**.
- Si la requête ne renvoie **aucune** ligne, l'exception NO_DATA_FOUND est déclenchée.
- Si au contraire elle renvoie **plus** d'une ligne, l'exception TOO_MANY_ROWS est déclenchée
  - Utilisation d'un curseur

Affectation du résultat d'une requête SELECT à une variable (2/2)

```
DECLARE
    numéro Produit.noProduit%TYPE;
    nom     Produit.libelle%TYPE
    prixTTC  Produit.prixUnitaire%TYPE;

BEGIN
SELECT noProduit , nomProduit, prixUnitaire*1.5  INTO numero, nom,  prixTTC
FROM    Produit  WHERE  noProduit = 'F12345';
END;
```

```
DECLARE
        produit_rec  Produit %ROWTYPE;

BEGIN
SELECT noProduit , nomProduit, prixUnitaire*1.5  INTO  produit_rec
FROM    Produit  WHERE  noProduit = 'F12345';
END;
```

PL/SQL

1. Introduction
2. Structure d'un bloc PL/SQL
3. Variables (types, déclaration, affectation)
4. **Structures de contrôle `(IF,CASE,LOOP,WHILE,FOR)`**
5. Curseurs
6. Sous-programmes : procédures et fonctions
7. Exceptions
8. Packages
9. Triggers (CI dynamiques)

4. Structures de contrôle
a- Conditionnelle (IF)

```
IF condition
THEN commandes;
[ELSE commandes ;]
END IF;
```

```
--
BEGIN
  IF (x>y) THEN max := x END IF;

  IF nb_ventes > quota THEN
       UPDATE Ventes set --- WHERE --;
  ELSE
       UPDATE Ventes SET --- WHERE --;
  END IF;
END;
```

4. Structures de contrôle
b- Conditionnelle multiple (CASE)

```
CASE expression
WHEN  valeur THEN commandes;
WHEN  valeur THEN commandes;
…
[ELSE commandes ; ]
END CASE;
```

```
CASE note
   WHEN 'A' THEN dbms_output.put_line ('bon');
   WHEN 'B' THEN dbms_output.put_line ('moyen');
   WHEN 'C' THEN dbms_output.put_line ('médiocre');
   ELSE dbms_output.put_line ('note inexistante');
END CASE;
```

---

4. Structures de contrôle
c- Boucle "infinie" (LOOP)

```
[<<nom_boucle>>]
LOOP
commandes;
END LOOP;
```

Pour sortir de la boucle :

```
EXIT [<<nom_boucle>>]  [WHEN condition];
```

4. Structures de contrôle
d- Boucle " tant-que" (WHILE)

```
[<<nom_boucle>>]
WHILE condition LOOP
   commandes;
END LOOP [nom_boucle];
```

```
DECLARE
X NUMBER(2) := 1;
BEGIN
WHILE X <= 100 LOOP
  ---
  X := X+2;
END LOOP;
END;
```

4. Structures de contrôle
e- Boucle "pour" (FOR)

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

```
<<boucle>>
FOR i IN 1..100 LOOP
 dbms_output.put_line (i);
END LOOP boucle;
       /*  1, 2, 3 … 100 */
```

```
<<boucleInverse>>
FOR i IN REVERSE 1..100 LOOP
  dbms_output.put_line(i)
END LOOP boucleInverse;
       /* 100, 99, 98, … 1 */
```

PL/SQL

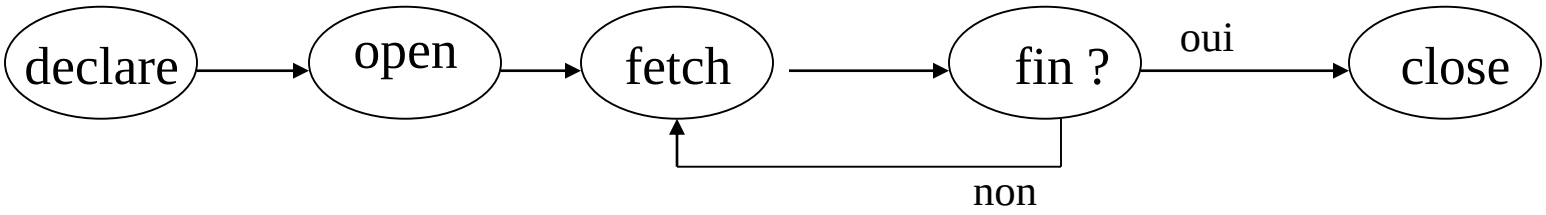
1. Introduction
2. Structure d'un bloc PL/SQL
3. Variables (types, déclaration, affectation)
4. Structures de contrôle `(IF,CASE,LOOP,WHILE,FOR)`
5. **Curseurs**
6. Sous-programmes : procédures et fonctions
7. Exceptions
8. Packages
9. Triggers (CI dynamiques)

5. Curseurs (1/7)

- Un curseur est une sorte de pointeur permettant de parcourir le résultat d'une requête tuple par tuple :

  - Déclaration du curseur (`CURSOR IS`)
    - on associe une requête SELECT au curseur
    - aucun effet visible
  - Ouverture du curseur (`OPEN`)
    - la requête SELECT est évaluée
    - le curseur pointe vers le premier tuple
  - Lecture du tuple courant et passage au suivant (`FETCH`)
  - Fermeture du curseur (`CLOSE`)

5. Curseurs (2/7)



5. Curseurs (3/7)

- Déclaration de curseur (dans la partie déclarative d'un bloc)

```
CURSOR nom_curseur[(argument1 type:=valeur, …)]
          IS requête;
```

```
DECLARE
CURSOR c1
  IS  SELECT      numProduit, libellé
      FROM        produit
      ORDER BY    numProduit;
CURSOR c2 (arg1 Produit.numProduit%TYPE := 5)
 IS   SELECT      numProduit, libelle
      FROM        produit
      WHERE       numProduit > arg1;
                        -- arg1 vaut 5 par défaut
```

5. Curseurs (4/7)

- Ouverture d'un curseur (dans la partie exécutable d'un bloc)

```
OPEN nom_curseur [(valeur_argument1 , …)];
```

```
DECLARE
CURSOR C1 IS  SELECT numProduit, libelle
              FROM   produit
              ORDER BY numProduit;
CURSOR C2 (arg1 Produit.numProduit%TYPE := 5)
 IS    SELECT numProduit, libelle FROM  produit
       WHERE  numProduit > param1;
BEGIN
OPEN C1;
OPEN C2 (10);
```

5. Curseurs (5/7)

- Lecture du tuple courant et passage au tuple suivant

```
FETCH nom_curseur INTO nom_variable_type_record;
```

ou

```
FETCH nom_curseur INTO nom_var1, nom_var2 …;
```

- Attributs d'un curseur

```
nom_curseur%ATTRIBUT
```

`%FOUND` : retourne T(rue) si un tuple a été trouvé (par FETCH)
`%NOTFOUND` : retourne T(rue) si aucun tuple trouvé
`%ISOPEN` : retourne T(rue) si le curseur est déjà ouvert
`%ROWCOUNT` : nombre de tuples déjà traités

5. Curseurs (6/7)

- Fermeture explicite d'un curseur

```
CLOSE nom_curseur;
```

- Parcours des tuples d'un curseur à l'aide d'une boucle FOR

```
  -- ouverture implicite
FOR variable_record IN nom_curseur LOOP
  -- disposer du tuple courant
END LOOP ;
  -- fermeture implicite du curseur
```

Soit la table **Produit** (numProduit, libelle, prix, numFournisseur)

```
DECLARE
CURSOR C1 (no_four Produit.numFournisseur%TYPE)
   IS  SELECT libelle, prixUnitaire
       FROM   Produit
       WHERE  numFornisseur = no_four;
un_produit C1%ROWTYPE ;
BEGIN
OPEN C1(123);
IF C1%ISOPEN THEN
  FETCH C1 INTO un_produit;
  WHILE C1%FOUND LOOP
     DBMS_OUTPUT.PUT_LINE ('Le produit : ' ||
un_produit.libelle|| ' coûte: ' ||un_produit.prix);
     FETCH C1 INTO un_produit;
   END LOOP;
ELSE DBMS_OUTPUT.PUT_LINE ('erreur lors de l\' ouverture du
curseur');
END IF;
CLOSE C1;
END;
```

Même bloc avec parcours du curseur avec une boucle FOR

```
DECLARE

CURSOR CUR (no_four Produit.numFournisseur%TYPE)
   IS  SELECT libelle, prixUnitaire
       FROM   Produit
       WHERE  numFornisseur = no_four;

un_produit CUR%ROWTYPE ;

BEGIN

FOR un_produit IN CUR(123) LOOP
DBMS_OUTPUT.PUT_LINE ( un_produit.libelle|| ' coûte: ' ||un_produit.prix);
END LOOP;

END;
```

5. Curseurs (7/7)

- Mise à jour à travers un curseur
  - Déclaration d'un curseur en vue d'une mise à jour

```
CURSOR nom_curseur IS requête FOR UPDATE;
```

  - Modification ou suppression du tuple courant désigné par le curseur

```
UPDATE nom_table ... WHERE CURRENT OF nom_curseur;
DELETE nom_table  WHERE CURRENT OF nom_curseur;
```

PL/SQL

1. Introduction
2. Structure d'un bloc PL/SQL
3. Variables (types, déclaration, affectation)
4. Structures de contrôle `(IF,CASE,LOOP,WHILE,FOR)`
5. Curseurs
6. **Sous-programmes : procédures et fonctions**
7. Exceptions
8. Packages
9. Triggers (CI dynamiques)

6. Sous-programmes : Procédures

- Création ou modification d'une procédure

```
CREATE [OR REPLACE] PROCEDURE nom_procédure [(nom_paramètre
   [IN| OUT |IN OUT] type_non_contraint
                        [{DEFAULT | :=} valeur], …)]
 {IS | AS}
   [déclaration de variables locales] --sans le mot DECLARE
 BEGIN
   commandes;
 [EXCEPTION
   commandes;]
 END [nom_procédure];
```

- 3 modes de passage d'un paramètre : par valeur (IN), par référence ou adresse (OUT) ou les deux (IN OUT)
- Initialisation et valeur par défaut impossibles pour OUT et IN OUT

6. Sous-programmes : Procédures

- Appel d'une procédure dans un bloc PL/SQL

```
nom_procédure [(paramètre_effectif, …)];

-- ou alors
nom_procédure [(nom_paramètre=>valeur, …)];
```

- Appel en dehors d'un bloc (commande SQL)

```
EXEC[UTE] nom_procédure [paramètres_effectifs];
```

6. Sous-programmes : Fonctions

- Création ou modification d'une fonction

```
CREATE [OR REPLACE] FUNCTION nom_fonction [(nom_paramètre
   type_non_contraint [{DEFAULT|:=} valeur],
                …)]
 RETURN type_non_contraint
 {IS | AS}
   [déclaration de variables locales]
 BEGIN
   ---
   RETURN valeur;     -- dans le corps de la fonction
   ---
 END [nom_fonction];
```

- Appel d'une fonction : partout où un type de la valeur retournée est utilisable (expressions)
- Les paramètres d'une fonction sont passés en mode IN

6. Procédures et Fonctions : compléments

- Types non contraints: `CHAR, VARCHAR2, NUMBER`
  - Types contraints : `CHAR(n),VARCHAR2(n),NUMBER(p[,c])`
- `CREATE OR REPLACE` : (re)compilation de la procédure/fonction
- `RETURN` dans le corps d'une fonction : arrêt de l'exécution de la fonction et retour à l'appelant
- Suppression d'une procédure

```
DROP PROCEDURE nom_procédure;
```

- Suppression d'une fonction

```
DROP FUNCTION nom_fonction;
```

- Utile en TP pour voir les erreurs de compilation

  `SHOW ERRORS` (commande du client SQL*Plus)

Exemple de procédure

```
CREATE OR REPLACE PROCEDURE conversion_FF_euro
   (prix_FF IN REAL, prix_euro OUT REAL)
 IS
  taux CONSTANT REAL := 6.55957;
 BEGIN
  IF prix_FF IS NOT NULL THEN
      prix_euro := prix_FF/taux;
  ELSE
      dbms_output.put_line ('calcul impossible');
  END IF;

 END conversion_FF_euro ;
```

Exemple de fonction (récursive)

```
CREATE  FUNCTION factorielle (n INTEGER)
 RETURN INTEGER
 IS
 BEGIN
   IF n=1 THEN
   RETURN 1;
   ELSE
       RETURN n*factorielle (n-1);
   END IF;
 END;
```

PL/SQL

1. Introduction
2. Structure d'un bloc PL/SQL
3. Variables (types, déclaration, affectation)
4. Structures de contrôle `(IF,CASE,LOOP,WHILE,FOR)`
5. Curseurs
6. Sous-programmes : procédures et fonctions
7. **Exceptions**
8. Packages
9. Triggers (CI dynamiques)

8. Exceptions

- A chaque erreur à l'exécution, une exception est "levée" ou déclenchée.
- Deux types d'exceptions
  1. Exceptions système prédéfinies
     - Levées automatiquement lorsque les erreurs se produisent
     - ex : cursor_already_open (`ORA-06511`), login_denied (`ORA-01017`), zero_divide (`ORA-01476`), too_many_rows, no_data_found, storage_error ...
  2. Exceptions définies par l'utilisateur
     - Déclarées, levées et traitées dans les blocs PL/SQL

Exceptions définies par l'utilisateur

- Déclarer l'exception dans la partie déclarative du bloc PL/SQL

```
nom_exception EXCEPTION ;
```

- Lever l'exception dans la partie exécutable du bloc

```
RAISE nom_exception ;
```

N.B. : La levée d'un exception interrompt l'exécution du bloc

- Traiter l'exception dans la partie EXCEPTION du bloc

```
WHEN nom_exception1 THEN commandes;
WHEN nom_exception2 THEN commandes;
…
WHEN OTHERS THEN commandes;
```

Exemple : exception utilisateur traitée dans le bloc

```
CREATE OR REPLACE PROCEDURE conversion_FF_euro (prix_FF IN
  REAL, prix_euro OUT REAL)
 IS
   taux CONSTANT REAL := 6.55957;
   prix_null EXCEPTION;              -- déclaration exception
 BEGIN
   IF prix_FF IS NOT NULL THEN
       prix_euro := prix_FF/taux;
   ELSE
       RAISE prix_null;              -- exception levée
   END IF;
 EXCEPTION
   WHEN prix_null THEN               -- exception traitée
       dbms_output.put_line ('Pb prix null');
 END conversion_FF_euro ;
```

Traitement des exceptions

- Si une exception se produit dans un bloc, l'exécution est interrompue et la partie EXCEPTION du bloc est exécutée.
  - Si l'exception correspond à une clause WHEN, les instructions sont exécutées et fin du programme
  - Sinon
    - S'il existe une clause WHEN OTHERS , les instructions correspondantes sont exécutées et fin du prog
    - Sinon, l'exception est propagée au programme appelant. Si aucun traitement d'exception n'est rencontré, la transaction qui a déclenché l'exception est annulée.
- Conseil : associer des codes aux erreurs !!!

Exceptions et codes d'erreur

- Possible de lier un nom d'exception à un code d'erreur

PRAGMA EXCEPTION_INIT (NOM_EXCEPTION, CODE_ERREUR)

  - Si une exception n'est pas traitée dans un programme, le programme appelant reçoit le code d'erreur associé

- Possible de déclencher une erreur sans la lier à une exception

RAISE_APPLICATION_ERROR(CODE_ERREUR,MESSAGE_ERREUR)

  - Intervalle de codes réservés aux erreurs non prédéfinies

    [-20999 , -20000]

Exemple précédent : liaison d'un code erreur à une exception

```
CREATE OR REPLACE PROCEDURE conversion_FF_euro (prix_FF IN
   REAL, prix_euro OUT REAL)
 IS
   taux CONSTANT REAL := 6.55957;
   prix_null EXCEPTION;                -- déclaration exception
   PRAGMA EXCEPTION_INIT(prix_null, -20150); --code erreur

 BEGIN
   IF prix_FF IS NOT NULL THEN
       prix_euro := prix_FF/taux;
   ELSE
       RAISE prix_null;      -- exception levée
                             -- code erreur transmis
                             -- puisque pas de partie Exception
   END IF;
END conversion_FF_euro ;
```

Exemple précédent : déclenchement d'une erreur (sans définir une exception)

```
CREATE OR REPLACE PROCEDURE conversion_FF_euro (prix_FF IN
   REAL, prix_euro OUT REAL)
 IS
   taux CONSTANT REAL := 6.55957;
 BEGIN
   IF prix_FF IS NOT NULL THEN
       prix_euro := prix_FF/taux;
   ELSE
       RAISE_APPLICATION_ERROR (-20150, 'Pb prix null');
             -- code et message erreur transmis à l'appelant

   END IF;
END conversion_FF_euro ;
```

PL/SQL

1. Introduction
2. Structure d'un bloc PL/SQL
3. Variables (types, déclaration, affectation)
4. Structures de contrôle `(IF,CASE,LOOP,WHILE,FOR)`
5. Curseurs
6. Sous-programmes : procédures et fonctions
7. Exceptions
8. **Packages**
9. Triggers (CI dynamiques)

7. Packages ou paquetages PL/SQL

- Un package ou paquetage permet de regrouper un ensemble de procédures, constantes, exceptions … liés entre eux (module)
- Un package comporte deux parties :
  - **Spécification** : interface publique (contient des éléments que l'on rend accessibles à tous les utilisateurs du package)
  - **Corps** : contient l'implémentation et ce que l'on veut cacher
- Droits : attribuer le droit d'exécuter un package donne accès à toute la spécification et pas au corps
- **Surcharge** autorisée : on peut avoir plusieurs procédures ou fonctions de même nom, avec des signatures différentes
- Pas de partage de variables : chaque session utilisant un package en possède une instance

7. Packages (2/4)

- **Contenu de la spécification**
  - des signatures de procédures et fonctions,
  - des constantes et des variables
  - des définitions d'exceptions
  - des définitions de curseurs
- **Contenu du corps**
  - Les corps des procédures et fonctions de la spécification (obligatoire)
  - D'autres procédures et/ou fonctions (cachées, facultatif)
  - Des déclarations que l'on veut garder privées
  - Un bloc d'initialisation du package si nécessaire.

Premier exemple de Package

```
-- PACKAGE ANNE-CÉCILE
-- SPÉCIFICATION DU PACKAGE
CREATE OR REPLACE PACKAGE MON_PAQ AS
   PROCEDURE P ;
   PROCEDURE P(I NUMBER) ;
   FUNCTION P(I NUMBER) RETURN NUMBER ;
   CPT NUMBER ;
   FUNCTION GET_CPT RETURN NUMBER ;
   MON_EXCEPTION EXCEPTION ;
   PRAGMA EXCEPTION_INIT(MON_EXCEPTION, -20101);
END ;
```

```
-- CORPS DU PACKAGE

CREATE OR REPLACE PACKAGE BODY MON_PAQ AS
    PROCEDURE P IS
        BEGIN DBMS_OUTPUT.PUT_LINE('TOTO'); END ;

    PROCEDURE P(I NUMBER) IS
        BEGIN DBMS_OUTPUT.PUT_LINE(I); END ;

    FUNCTION P(I NUMBER) RETURN NUMBER IS
        BEGIN  IF (I > 10) THEN RAISE MON_EXCEPTION ;
                    END IF ;
        RETURN I ;
        END ;
    FUNCTION GET_CPT RETURN NUMBER IS
        BEGIN RETURN CPT ; END ;
END ;
```

```
-- EXEMPLE D'UTILISATION
BEGIN
MON_PAQ.CPT := 20 ;
END ;
SELECT MON_PAQ.GET_CPT FROM DUAL ;
GET_CPT
---------
20
-- AUTRE UTILISATEUR :
SELECT MOI.MON_PAQ.GET_CPT FROM DUAL ;
GET_CPT
-------
NULL
SELECT MON_PAQ.P(11) FROM DUAL ;
-- AVEC LE PRAGMA EXCEPTION_INIT ... ON OBTIENT
ORA-20101:
ORA-06512: À "MOI.MON_PAQ",
LIGNE 12 ORA-06512: À LIGNE 1
```

Autre exemple de package

```
-- PACKAGE PKG_FINANCE

-- SPÉCIFICATION
CREATE OR REPLACE PACKAGE  PKG_FINANCE IS
 -- VARIABLES GLOBALES ET PUBLIQUES
 GN$SALAIRE    EMP.SAL%TYPE ;
-- FONCTIONS PUBLIQUES
FUNCTION    F_TEST_AUGMENTATION (
      PN$NUMEMP   IN   EMP.EMPNO%TYPE,
      PN$POURCENT IN   NUMBER   )
RETURN NUMBER ;
-- PROCÉDURES PUBLIQUES
PROCEDURE    TEST_AUGMENTATION ( PN$NUMEMP IN
EMP.EMPNO%TYPE ,    PN$POURCENT   IN   OUT NUMBER ) ;
END PKG_FINANCE ;
```

-- LE CORPS DU PACKAGE PKG_FINANCE

```
CREATE OR REPLACE PACKAGE BODY PKG_FINANCE IS
-- VARIABLES GLOBALES PRIVÉES
 GR$EMP   EMP%ROWTYPE ;
-- PROCÉDURE PRIVÉES
PROCEDURE  AFFICHE_SALAIRES
IS
 CURSOR C_EMP IS SELECT * FROM EMP ;
BEGIN
FOR GR$EMP IN C_EMP
      LOOP
      DBMS_OUTPUT.PUT_LINE('EMPLOYÉ ' || GR$EMP.ENAME
|| ' --> ' || LPAD( TO_CHAR (GR$EMP.SAL), 10)) ;
      END LOOP ;
END AFFICHE_SALAIRES ;
```

```
-- FONCTIONS PUBLIQUES
FUNCTION    F_TEST_AUGMENTATION (PN$NUMEMP IN
EMP.EMPNO%TYPE, PN$POURCENT IN NUMBER ) RETURN NUMBER
IS

LN$SALAIRE EMP.SAL%TYPE ;

BEGIN
 SELECT SAL INTO LN$SALAIRE FROM EMP WHERE EMPNO =
PN$NUMEMP ;
 -- AUGMENTATION VIRTUELLE DE L'EMPLOYÉ
 LN$SALAIRE := LN$SALAIRE * PN$POURCENT ;
 -- AFFECTATION DE LA VARIABLE GLOBALE PUBLIQUE
 GN$SALAIRE := LN$SALAIRE ;
 RETURN( LN$SALAIRE ) ; -- RETOUR DE LA VALEUR
END   F_TEST_AUGMENTATION;
```

```
-- PROCÉDURES PUBLIQUES
PROCEDURE     TEST_AUGMENTATION (PN$NUMEMP IN
EMP.EMPNO%TYPE, PN$POURCENT IN OUT NUMBER ) IS

LN$SALAIRE EMP.SAL%TYPE ;

BEGIN
 SELECT SAL INTO LN$SALAIRE FROM EMP
      WHERE EMPNO = PN$NUMEMP ;
 -- AUGMENTATION VIRTUELLE DE L'EMPLOYÉ
 PN$POURCENT := LN$SALAIRE * PN$POURCENT ;
 -- APPEL PROCÉDURE PRIVÉE
 AFFICHE_SALAIRES ;
 END TEST_AUGMENTATION;
 END PKG_FINANCE;
```

```
-- APPEL DE LA PROCÉDURE TEST_AUGMENTATION DU
-- PACKAGE PKG_FINANCE
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 ;
```

PL/SQL

1. Introduction
2. Structure d'un bloc PL/SQL
3. Variables (types, déclaration, affectation)
4. Structures de contrôle `(IF,CASE,LOOP,WHILE,FOR)`
5. Curseurs
6. Sous-programmes : procédures et fonctions
7. Packages
8. Exceptions
9. **Triggers (CI dynamiques)**

Rappels sur les Contraintes d'Intégrité

- Les contraintes d'intégrité (CI) : contraintes que doivent vérifier les données pour être en phase avec le monde réel

- Une base de données est dite *cohérente* si ses contraintes d'intégrité sont satisfaites

Contraintes exprimées lors de la création d'une table (Rappels)

```
CREATE TABLE Depot
   (dep#       INT NOT NULL,
    adr        VARCHAR(50),
    capacite   INT,
    PRIMARY KEY dep#,
    CHECK (capacite>1000))

CREATE TABLE Stock
   (prod#      INT,
    dep#       INT CHECK (dep# BETWEEN 10 AND 50),
    qte        INT DEFAULT 0,
    PRIMARY KEY (prod#, dep#),
    FOREIGN KEY (prod#) REFERENCES Produit(prod#),
    FOREIGN KEY (dep#) REFERENCES Depot(dep#),
    CONSTRAINT ctr_qte CHECK (qte>=0))
```

Typologie des contraintes d'intégrité (1/2)

- **Les contraintes de domaine :** concernent un attribut

  (contraintes de domaine ou de valeur obligatoire/facultative)

```
CREATE TABLE Student
   ( stid INT NOT NULL,
     name VARCHAR(50),
     age INT CHECK (age BETWEEN 12 AND 90),
     major VARCHAR(10) ENUM ('biology', 'cs', 'law')
)
```

- **Les contraintes intra-relation :**

  qui portent sur un seul tuple (cf contrainte de domaine)

  ou sur plusieurs tuples d'une même relation (clé primaire, unicité, cardinalité) "*l'étudiant X ne peut pas avoir le même stid# que l'étudiant Y*"

```
CREATE TABLE Student
   ( stid INT PRIMARY KEY,
     socialsecuritynum INT UNIQUE,
     name VARCHAR(50),
     age INT )
```

Typologie des contraintes d'intégrité (2/2)

- **Les contraintes inter-relations :** dépendances référentielles ou existentielles (clé étrangère) "*toute étudiant inscrit à un cours doit être présent dans la relation étudiant*"

```
CREATE TABLE ENROLLMENT
  ( stid INT REFERENCES Student,
    courseid INT REFERENCES Course(cid),
    year INT)
```

- **Les contraintes dynamiques :** les valeurs des données dépendent de l'état de la base ou d'un des états précédents ("*le salaire d'un employé ne peut pas diminuer*")

- **Les contraintes de gestion** :

Exemple : "*un enseignant ne peut pas être responsable d'un module après sa retraite*"

```
CREATE TABLE COURSE(
   cid INT PRIMARY KEY,
   profid INT REFERENCES Prof(pid),
   year INT),
   coursename VARCHAR2(40) )
```

```
CREATE TABLE PROF(
   pid INT PRIMARY KEY,
   profname VARCHAR2(150),
   retirementdate DATE )
```

Les *triggers* (déclencheurs)

- Trigger : procédure associée à une relation qui se déclenche quand un type d'événement survient
  - Objet persistant dans la BD
- Créé par le propriétaire de la relation ou un utilisateur ayant le privilège CREATE TRIGGER.
- Les triggers permettent de :
  - Définir des CI complexes qu'on ne peut pas spécifier lors de la création d'une table (CI dynamiques, CI de gestion)
  - Tracer les informations concernant divers événements

Triggers Oracle (1/4)

- Triggers **avant** ou **après** un type d'événement (BEFORE ou AFTER)
- Types d'événements : insertion/suppression/mise à jour de tuples
- Triggers de niveau instruction ou de niveau ligne (FOR EACH ROW)

  (une commande de mise à jour peut affecter pluieurs tuples)

```
CREATE [OR REPLACE] TRIGGER nom_trigger
   {BEFORE | AFTER}
   {INSERT |DELETE |UPDATE [of liste_attributs]}
   ON nom_table
   [WHEN condition]
   [FOR EACH ROW]
   [DECLARE …]
   BEGIN
   …          -- bloc PL/SQL ou appel de procédure
              -- sans LDD ni contrôle de transaction
              -- possible de lever des exceptions
   END [nom_trigger];
```

Triggers Oracle (2/4)

Algorithme de contrôle de l'exécution d'une instruction de mise à jour (insert/delete/update) sur une table :

1. Exécuter le trigger BEFORE de niveau instruction, s'il en existe

2. Pour chaque ligne affectée par l'instruction :
   a) exécuter le trigger BEFORE de niveau ligne, s'il en existe
   b) exécuter l'instruction
   c) exécute le trigger AFTER de niveau ligne, s'il en existe

3. Exécuter le trigger AFTER de niveau instruction, s'il en existe

Triggers Oracle (3/4)

- Pour les **triggers de niveau ligne**, on accède aux données du tuple en cours de mise à jour au moyen de deux identifiants de corrélation :
  - **:OLD** : valeurs d'origine du tuple avant la mise à jour
  - **:NEW** : valeurs qui seront insérées ou remplaceront celles d'origine au terme de la mise à jour
- :NEW et :OLD sont définis pour l'instruction UPDATE
- :OLD est NULL pour l'instruction INSERT
- :NEW est NULL pour l'instruction DELETE

Triggers Oracle (4/4)

Rappel : Procédure **`raise_application_error (error_number, error_message)`**

`error_number` : entier compris entre -20000 et -20999

`error_message` : chaîne de 500 caractères

Quand cette procédure est appelée dans un trigger :

1) elle termine le trigger
2) annule les effets de la transaction (ROLLBACK)
3) renvoie numéro et message d'erreur à l'application.

- Un trigger peut être activé/désactivé

  ALTER TRIGGER ENABLE/DISABLE

- Un trigger peur être supprimé (DROP TRIGGER)
- Attention aux triggers récursifs !

Exemple

```
CREATE TABLE Employee
(id NUMBER(10) PRIMARY KEY,…
 salary NUMBER(12,2));
```

CI : Le salaire d'un employé ne doit pas baisser

```
CREATE OR REPLACE TRIGGER check_salary
AFTER UPDATE of (salary)
ON Employee
FOR EACH ROW
WHEN (NEW.salary < OLD.salary)
DECLARE
BEGIN
raise_application_error(-20600, 'Le salaire ne peut pas baisser !');
END ;
```

Noter la syntaxe new et pas :new dans la condition WHEN

Table en mutation

- Il ne faut pas, dans un trigger de niveau ligne, interroger une table qui est en cours de modification (*mutating table*)

```
CREATE TABLE Emp
(id NUMBER(3) PRIMARY KEY,
 name VARCHAR2(20));
-- CI : attribuer un numéro en séquence si nécessaire

CREATE OR REPLACE TRIGGER CLE_AUTO
BEFORE INSERT
ON Emp
FOR EACH ROW
WHEN (NEW.id IS NULL)
DECLARE
  next_id NUMBER (3);
BEGIN
  SELECT nvl(max(id),0) +1 INTO
                      next_id FROM Emp;
  :NEW.id := next_id ;
END;
```

```
INSERT INTO Emp (name)
VALUES ('AB');

ORA-04091: mutating table
USER.Emp
Trigger can not see it.
```

Table en mutation : remède

- Disposer d'une copie à jour de la table

```
CREATE TABLE Emp
(id NUMBER(3) PRIMARY KEY,
 name VARCHAR2(20));

CREATE TABLE Emp_copy
(id NUMBER(3) PRIMARY KEY);

-- CI : attribuer un numéro en séquence si nécessaire

CREATE OR REPLACE TRIGGER CLE_AUTO
BEFORE INSERT
ON Emp
FOR EACH ROW
WHEN (NEW.id IS NULL)
DECLARE
   next_id NUMBER (3);
BEGIN
   SELECT nvl(max(id),0)+1 INTO next_id
        FROM Emp_copy ;
   :NEW.id := next_id ;
   INSERT INTO Emp_copy VALUES (next_id);
END ;
```

```
INSERT INTO Emp (name)
VALUES  ('J. VAIS');

INSERT INTO Emp (name)
VALUES  ('B. THAON');

INSERT INTO Emp (id, name)
 VALUES  (10, 'C. CLAIR');
```

Deux exemples de trigger (1/2)

```
-- employe(numemp, numserv,…)
-- service(numserv,...)
-- CI : vérifier que le service d'un nouvel employé existe bien
CREATE TRIGGER verif_service
BEFORE INSERT OR UPDATE OF numserv
ON employe
FOR EACH ROW
WHEN (new.numserv is not null)
DECLARE
  noserv integer;
BEGIN
  noserv:=0;
  SELECT COUNT(numserv) INTO noserv
  FROM SERVICE
  WHERE numserv=:new.numserv;
  IF (noserv=0) THEN
    raise_application_error(-20501, 'service inexistant');
  END IF;
END;
```

Deux exemples de trigger (2/2)

```
-- employe(numemp,salaire,grade,...)
-- grille(grade,salmin,salmax)

-- s'assurer que le salaire est dans les bornes
-- correspondant au grade de l'employé

CREATE TRIGGER verif_grade_salaire
BEFORE INSERT OR UPDATE OF salaire,
  grade
ON employe
FOR EACH ROW
DECLARE
  minsal NUMBER;
  maxsal NUMBER;
```

```
…BEGIN
-- retrouver le salaire min et max du grade
SELECT salmin, salmax
            INTO minsal, maxsal
FROM  grille  WHERE grade = :new.grade;
--  si problème, erreur
IF (:new.salaire<minsal OR
  :new.salaire>maxsal)
THEN
          raise_application_error (-20300,
      'Salaire ' ||TO_CHAR (:new.salaire) ||
            ' incorrect pour ce grade');

EXCEPTION
  WHEN  NO_DATA_FOUND  THEN
            raise_application_error
            (-20301,'Grade incorrect');
END;
```