M2106
Programmation et administration des bases de données

http://cedric.cnam.fr/~ferecatm
Marin Ferecatu
(supports de cours : Michel Crucianu)

Contenu du cours

- PL/SQL
  - Variables, structures de contrôle
  - Curseurs, interaction avec la base
  - Sous-programmes, paquetages
  - Exceptions
  - Transactions
  - Déclencheurs (*triggers*)
- Administration (utilisateurs, rôle, droits, vues, accès, sécurité)

Bibliographie

- Anne-Sophie LACROIX, Jérôme GABILLAUD, Programmez avec SQL et PL/SQL, Editions ENI 2015
- Joan Casteel, *Oracle 11g: PL/SQL Programming, Cengage Learning*, 2012
- Michael McLaughlin, *Oracle Database 12c PL/SQL Programming*, McGraw-Hill 2014
- Date, C. *Introduction aux bases de données*. Vuibert, 2004 (8ème édition).
- Gardarin, G. *Bases de données*, Eyrolles. 2003.
- Reese, G. *Database programming with JDBC and Java,* O'Reilly, 2006.
- Soutou, C. *SQL pour Oracle*. Eyrolles, 2015.

PL/SQL

- *Procedural Language / Structured Query Language* (PL/SQL) : langage propriétaire Oracle
- Syntaxe de PL/SQL inspirée du langage Ada
- D'autres éditeurs de SGBDR utilisent des langages procéduraux similaires (ex: MS SQL Server - T-SQL)
- Documentation Oracle (en anglais) :

  http://docs.oracle.com/cd/E11882_01/appdev.112/e25519/toc.htm

Quel est l'intérêt de PL/SQL ?

- La nature relationnelle, déclarative de SQL rend l'expression des requêtes très naturelle

…mais les applications complexes exigent plus :

  - Pour la facilité et l'efficacité de développement : gérer le contexte et lier entre elles plusieurs requêtes, créer des librairies de procédures cataloguées réutilisables
  - Pour l'efficacité de l'application : diminuer le volume des échanges entre client et serveur (un programme PL/SQL est exécuté sur le serveur)

⇒ Nécessité d'étendre les fonctionnalités de SQL : PL/SQL est une extension procédurale

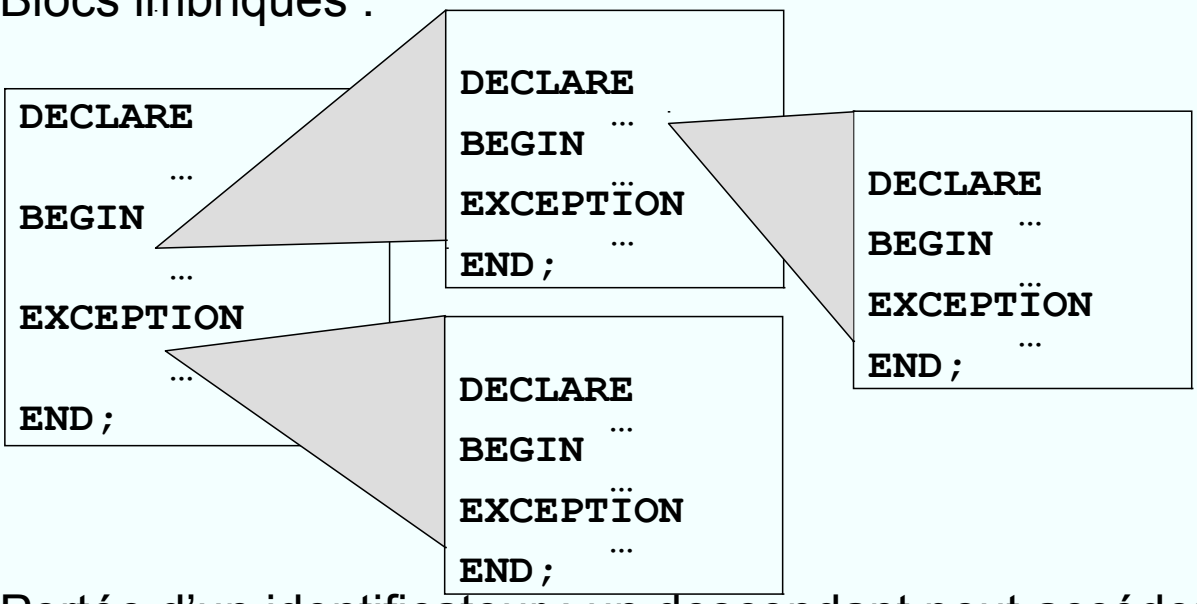
Structure d'un programme

- Programme PL/SQL = bloc (procédure anonyme, procédure nommée, fonction nommée) :

```
DECLARE
        -- section de déclarations
        -- section optionnelle
        …
BEGIN
        -- traitement, avec d'éventuelles directives SQL
        -- section obligatoire
        …
EXCEPTION
        -- gestion des erreurs retournées par le SGBDR
        -- section optionnelle
        …
END;
/       ← lance l'exécution sous SQL*Plus
```

Structure d'un programme (2)

- Blocs imbriqués :



- Portée d'un identificateur : un descendant peut accéder aux identificateurs déclarés par un parent, pas l'inverse

Identificateurs, commentaires

- Identificateur (variable, curseur, exception, etc.) :
  - Commence par une lettre
  - Peut contenir : lettres, chiffres, $, #, _
  - Interdits : &, -, /, *espace*
  - Jusqu'à 30 caractères
  - Insensible à la casse ! (`nompilote = NomPILOTE`)

- Commentaires :

```
-- Commentaire sur une seule ligne
/* Commentaire sur plusieurs
   lignes */
```

Variables

- Types de variables PL/SQL :
  - Scalaires : par exemple **NUMBER(5,2), VARCHAR2, DATE, BOOLEAN,** …
  - Composites : **%ROWTYPE, RECORD, TABLE**
  - Référence : **REF**
  - LOB (*Large Object* jusqu'à 4 Go ; pointeur si externe)
- Un programme PL/SQL peut également manipuler des variables non PL/SQL :
  - Variables de substitution ou globales définies dans SQL*Plus
  - Variables hôtes (définies dans d'autres programmes et utilisées par le programme PL/SQL)
- Toute variable PL/SQL doit obligatoirement être déclarée dans une section **DECLARE** avant utilisation

Variables scalaires

- Syntaxe de déclaration :

  Identificateur [CONSTANT] type [[NOT NULL] {:= | DEFAULT} expression];
  - **CONSTANT** : c'est une constante (sa valeur ne peut pas changer, doit être initialisée lors de la déclaration)
  - **NOT NULL** : on ne peut pas lui affecter une valeur nulle (en cas de tentative, une exception **VALUE_ERROR** est levée)
  - Initialisation : affectation **:=** ou **DEFAULT** (sinon : **NULL** !)
- Pas de déclaration multiple dans PL/SQL !

  number1, number2 NUMBER; ← déclaration incorrecte !
- Autre possibilité pour affecter une valeur à une variable

  SELECT … INTO identificateur FROM … ;

Nouveaux types PL/SQL

- Nouveaux types prédéfinis :

  BINARY_INTEGER : entiers signés entre -2^{31} et 2^{31}

  PLS_INTEGER : entiers signés entre -2^{31} et 2^{31} ; plus performant que **NUMBER** et **BINARY_INTEGER** au niveau des opérations arithmétiques ; en cas de dépassement, exception levée

- Sous-types PL/SQL : restriction d'un type de base
  - Prédéfinis : **CHARACTER, INTEGER, NATURAL, POSITIVE, FLOAT, SMALLINT, SIGNTYPE**, etc.
  - Utilisateur (restriction : précision ou taille maximale) :

```
SUBTYPE nomSousType IS typeBase [(contrainte)]
  [NOT NULL];
```

  - Exemple de sous-type utilisateur :

```
SUBTYPE numInsee IS NUMBER(13) NOT NULL;
```


Base pour les exemples

avion :

Numav	Capacite	Type	Entrepot
14	25	A400	Garches
345	75	B200	Lille

pilote :

Matricule	Nom	Ville	Age	Salaire
1	Durand	Cannes	45	28004
2	Dupont	Touquet	24	11758

vol :

Numvol	Heure_depart	Heure_arrivee	Ville_depart	Ville_arrivee
AL12	08-18	09-12	Paris	Lille
AF8	11-20	23-54	Paris	Rio

Variables scalaires : exemple

```
DECLARE
   villeDepart      CHAR(10) := 'Paris';
   villeArrivee     CHAR(10);
   numVolAller      CONSTANT CHAR := 'AF8';
   numVolRetour     CHAR(10);
BEGIN
   SELECT Ville_arrivee INTO villeArrivee FROM vol
   WHERE Numvol = numVolAller;  -- vol aller
   numVolRetour := 'AF9';
   INSERT INTO vol VALUES (numVolRetour, NULL, NULL,
   villeArrivee, villeDepart);  -- vol retour
END;
```


Conversions

- Conversions implicites :
  - Lors du calcul d'une expression ou d'une affectation
  - Si la conversion n'est pas autorisée (voir page suivante), une exception est levée
- Conversions explicites :

De A	CHAR	NUMBER	DATE	RAW	ROWID
CHAR		TO_NUMBER	TO_DATE	HEXTORAW	CHARTOROWID
NUMBER	TO_CHAR		TO_DATE		
DATE	TO_CHAR				
RAW	RAWTOHEX				
ROWID	ROWIDTOHEX				

Quelques conversions implicites

De \ A	CHAR	VARCHAR2	BINARY_ INTEGER	NUMBER	LONG	DATE	RAW	ROWID
CHAR		OUI	OUI	OUI	OUI	OUI	OUI	OUI
VARCHAR2	OUI		OUI	OUI	OUI	OUI	OUI	OUI
BINARY_ INTEGER	OUI	OUI		OUI	OUI			
NUMBER	OUI	OUI	OUI		OUI			
LONG	OUI	OUI					OUI	
DATE	OUI	OUI			OUI			
RAW	OUI	OUI			OUI			
ROWID	OUI	OUI						

Déclaration %TYPE

- Déclarer une variable de même type que
  - Une colonne (attribut) d'une table existante :

    ```
    nomNouvelleVariable NomTable.NomColonne%TYPE
    [{:= | DEFAULT} expression];
    ```
  - Une autre variable, déclarée précédemment :

    ```
    nomNouvelleVariable nomAutreVariable%TYPE
    [{:= | DEFAULT} expression];
    ```
- Cette propagation du type permet de réduire le nombre de changements à apporter au code PL/SQL en cas de modification des types de certaines colonnes

Déclaration %TYPE : exemple

```
DECLARE
   nomPilote pilote.Nom%TYPE;  /* table pilote,
                                   colonne nom */
   nomCoPilote nomPilote%TYPE;
BEGIN
   …
   SELECT Nom INTO nomPilote FROM pilote WHERE
   matricule = 1;
   SELECT Nom INTO nomCoPilote FROM pilote WHERE
       matricule = 2;
   …
END;
```


Variables %ROWTYPE

- Déclarer une variable composite du même type que les tuples d'une table :

  `nomNouvelleVariable NomTable%ROWTYPE;`
- Les composantes de la variables composite, identifiées par `nomNouvelleVariable.nomColonne`, sont du même type que les colonnes correspondantes de la table
- Les contraintes `NOT NULL` déclarées au niveau des colonnes de la table ne sont pas transmises aux composantes correspondantes de la variable !
- Attention, on peut affecter un seul tuple à une variable définie avec `%ROWTYPE` !

Variables %ROWTYPE : exemple

```
DECLARE
  piloteRecord    pilote%ROWTYPE;
  agePilote       NUMBER(2) NOT NULL := 35;
BEGIN
  piloteRecord.Age := agePilote;
  piloteRecord.Nom := 'Pierre';
  piloteRecord.Ville := 'Bordeaux';
  piloteRecord.Salaire := 45000;
  INSERT INTO pilote VALUES piloteRecord;
END;
```


Variables RECORD

- Déclarer une variable composite personnalisée (similaire à `struct` en C) :

```
TYPE nomTypeRecord IS RECORD
(nomChamp typeDonnee [[NOT NULL] {:= |
DEFAULT} expression]
    [, nomChamp typeDonnee  …]…);
…
nomVariableRecord nomTypeRecord;
```

- Les composantes de la variables composite peuvent être des variables PL/SQL de tout type et sont identifiées par `nomVariableRecord.nomChamp`

Variables RECORD : exemple

```
DECLARE
   TYPE aeroport IS RECORD
   (ville CHAR(10), distCentreVille NUMBER(3),
   capacite NUMBER(5));
   ancienAeroport aeroport;
   nouvelAeroport aeroport;
BEGIN
   ancienAeroport.Ville := 'Paris';
   ancienAeroport.DistCentreVille := 20;
   ancienAeroport.Capacite := 2000;
   nouvelAeroport := ancienAeroport;
END;
```


Variables TABLE

- Définir des tableaux dynamiques (dimension initiale non précisée) composés d'une clé primaire et d'une colonne de type scalaire, `%TYPE`, `%ROWTYPE` ou `RECORD`

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

- Si `INDEX BY BINARY_INTEGER` est présente, l'index peut être entre -2^{31} et 2^{31} (type `BINARY_INTEGER`) !
- Fonctions PL/SQL dédiées aux `TABLE` : `EXISTS(x)`, `PRIOR(x)`, `NEXT(x)`, `DELETE(x,…)`, `COUNT`, `FIRST`, `LAST`, `DELETE`

Variables TABLE : exemple

```
DECLARE
   TYPE pilotesProspectes IS TABLE OF pilote%ROWTYPE
       INDEX BY BINARY_INTEGER;
   tabPilotes pilotesProspectes;
   tmpIndex   BINARY_INTEGER;
BEGIN
   …
   tmpIndex := tabPilotes.FIRST;
   tabPilotes(4).Age := 37;
   tabPilotes(4).Salaire := 42000;
   tabPilotes.DELETE(5);
   …
END;
```


Variables de substitution

- Passer comme paramètres à un bloc PL/SQL des variables définies sous SQL*Plus :

```
SQL> ACCEPT s_matPilote PROMPT 'Matricule : '
SQL> ACCEPT s_nomPilote PROMPT 'Nom pilote : '
SQL> ACCEPT s_agePilote PROMPT 'Age pilote : '
DECLARE
salairePilote NUMBER(6,2) DEFAULT 32000;
BEGIN
INSERT INTO pilote VALUES ('&s_matPilote',
'&s_nomPilote', NULL, &s_agePilote,
salairePilote);
END;
/
```

Variables hôtes, variables de session

- Variables hôtes :
  - Définies en dehors de PL/SQL et appelées dans PL/SQL
  - Dans le code PL/SQL, préfixer le nom de la variable de « : »
- Variables de session (globales) :
  - Directive SQL*Plus **VARIABLE** avant le bloc PL/SQL concerné
  - Dans le code PL/SQL, préfixer le nom de la variable de « : »

```
SQL> VARIABLE compteurGlobal NUMBER := 0;
BEGIN
  :compteurGlobal := :compteurGlobal - 1;
END;
/
SQL> PRINT compteurGlobal
```


Résolution des noms

- Règles de résolution des noms :
  - Le nom d'une variable du bloc l'emporte sur le nom d'une variable externe au bloc (et visible)
  - Le nom d'une variable l'emporte sur le nom d'une table
  - Le nom d'une colonne d'une table l'emporte sur le nom d'une variable
- Étiquettes de blocs :

```
<<blocExterne>>
DECLARE
dateUtilisee DATE;
BEGIN
DECLARE
dateUtilisee DATE;
BEGIN
dateUtilisee := blocExterne.dateUtilisee;
END;
END blocExterne;
```

Entrées et sorties

- Paquetage `DBMS_OUTPUT` :
  - Mise dans le tampon d'une valeur :

```
PUT(valeur IN {VARCHAR2 | DATE | NUMBER});
```

  - Mise dans le tampon du caractère fin de ligne :

```
NEW_LINE(ligne OUT VARCHAR2, statut OUT INTEGER);
```

  - Mise dans le tampon d'une valeur suivie de fin de ligne :

```
PUT_LINE(valeur IN {VARCHAR2 | DATE | NUMBER});
```

- Rendre les sorties possibles avec SQL*Plus :

```
SET SERVEROUT ON
```

(ou `SET SERVEROUTPUT ON`)

Entrées et sorties (2)

  - Lecture d'une ligne :

```
GET_LINE(ligne OUT VARCHAR2(255),
     statut OUT INTEGER);
```

  - Lecture de plusieurs lignes :

```
GET_LINES(tableau OUT DBMS_OUTPUT.CHARARR,
nombreLignes IN OUT INTEGER);
```

- Exemple :

```
DECLARE
lgnLues DBMS_OUTPUT.CHARARR;
nombreLignes INTEGER := 2;
BEGIN
DBMS_OUTPUT.PUT_LINE('Nombre de lignes : ' ||
nombreLignes);
DBMS_OUTPUT.GET_LINES(lgnLues, nombreLignes);
END;
```

Entrées et sorties (3)

- Autres API pour des E/S spécifiques :
  - **DBMS_PIPE** : échanges avec les commandes du système d'exploitation
  - **UTL_FILE** : échanges avec des fichiers
  - **UTL_HTTP** : échanges avec un serveur HTTP (Web)
  - **UTL_SMTP** : échanges avec un serveur SMTP (courriel)
  - **HTP** : affichage des résultats sur une page HTML

Structures de contrôle

- Structures conditionnelles :
  - **IF … THEN … ELSIF… THEN … ELSE … END IF;**
  - **CASE … WHEN … THEN … ELSE … END CASE;**
- Structures répétitives :
  - **WHILE … LOOP … END LOOP;**
  - **LOOP … EXIT WHEN … END LOOP;**
  - **FOR … IN … LOOP … END LOOP;**

Structures conditionnelles : IF

- Syntaxe :

```
IF condition1 THEN instructions1;
    [ELSIF condition2 THEN instructions3;]
    [ELSE instructions2;]
END IF;
```

- Conditions : expressions dans lesquelles peuvent intervenir noms de colonnes, variables PL/SQL, valeurs
- Opérateurs AND et OR :

AND	T	F	NULL
T	T	F	NULL
F	F	F	F
NULL	NULL	F	NULL

OR	T	F	NULL
T	T	T	T
F	T	F	NULL
NULL	T	NULL	NULL

IF : exemple

```
SQL> ACCEPT s_agePilote PROMPT 'Age pilote : '
DECLARE
  salairePilote pilote.Salaire%TYPE;
BEGIN
  IF &s_agePilote = 45 THEN
      salairePilote := 28004;
  ELSIF &s_agePilote = 24 THEN
      salairePilote := 11758;
  END IF;
  DBMS_OUTPUT.PUT_LINE('Salaire de base : ' ||
  salairePilote);
END;
/
```

Structures conditionnelles : CASE

- Syntaxe avec expressions :

```
[<<etiquette>>]
CASE variable
    WHEN expression1 THEN instructions1;
    WHEN expression2 THEN instructions2; …
    [ELSE instructionsj;]
END CASE [etiquette];
```

- Valeur de la **variable** comparée au résultat de l'évaluation de **expression1**, si égalité alors sont exécutées **instructions1**, sinon comparaison au résultat de **expression2**, etc.
- Le premier cas valide est traité, les autres sont ignorés
- Aucun cas valide : exception **CASE_NOT_FOUND** levée !

CASE : exemple (avec expressions)

```
SQL> ACCEPT s_agePilote PROMPT 'Age pilote : '
DECLARE
   salairePilote pilote.Salaire%TYPE;
BEGIN
   CASE &s_agePilote
       WHEN 45 THEN salairePilote := 28004;
       WHEN 24 THEN salairePilote := 11758;
   END CASE;
   DBMS_OUTPUT.PUT_LINE('Salaire de base : ' ||
   salairePilote);
EXCEPTION
   WHEN CASE_NOT_FOUND THEN
       DBMS_OUTPUT.PUT_LINE('Évaluer expérience');
END;
```

Structures conditionnelles : CASE (2)

- Syntaxe avec conditions (*searched case*) :

```
[<<etiquette>>]
CASE
    WHEN condition1 THEN instructions1;
    WHEN condition2 THEN instructions2; …
    [ELSE instructionsj;]
END CASE [etiquette];
```

- La **condition1** est évaluée, si elle s'évalue à **TRUE** alors sont exécutées **instructions1**, sinon est évaluée **condition2**, etc.
- Le premier cas valide est traité, les autres sont ignorés
- Aucun cas valide : exception **CASE_NOT_FOUND** levée

CASE : exemple (avec conditions)

```
SQL> ACCEPT s_agePilote PROMPT 'Age pilote : '
DECLARE
   salairePilote pilote.Salaire%TYPE;
BEGIN
   CASE
       WHEN &s_agePilote=45 THEN salairePilote := 28004;
       WHEN &s_agePilote=24 THEN salairePilote := 11758;
   END CASE;
   DBMS_OUTPUT.PUT_LINE('Salaire de base : ' ||
   salairePilote);
EXCEPTION
   WHEN CASE_NOT_FOUND THEN
       DBMS_OUTPUT.PUT_LINE('Évaluer expérience');
END;
```

Structures répétitives : tant que

- Syntaxe :

```
[<<etiquette>>]
WHILE condition LOOP
    instructions;
END LOOP [etiquette];
```

- La condition est vérifiée au début de chaque itération ; tant que la condition est vérifiée, les instructions sont exécutées

Tant que (**WHILE**) : exemple

```
DECLARE
  valeurEntiere   INTEGER := 1;
  produitSerie    INTEGER := 1;
BEGIN
  WHILE (valeurEntiere <= 10) LOOP
      produitSerie := produitSerie * valeurEntiere;
      valeurEntiere := valeurEntiere + 1;
  END LOOP;
END;
```

- A la fin du bloc, **produitSerie** est 10! (3628800) et **valeurEntiere** est 11

Structures répétitives : répéter

- Syntaxe :

```
[<<etiquette>>]
LOOP
    instructions1;
    EXIT [WHEN condition];
    [instructions2;]
END LOOP [etiquette];
```

- La condition est vérifiée à chaque itération ; si elle est vraie, on quitte la boucle
- Si la condition est absente, les instructions sont exécutées une seule fois

Répéter (**LOOP**) : exemple

```
DECLARE
  valeurEntiere   INTEGER := 1;
  produitSerie    INTEGER := 1;
BEGIN
  LOOP
      produitSerie := produitSerie * valeurEntiere;
      EXIT WHEN valeurEntiere >= 10;
      valeurEntiere := valeurEntiere + 1;
  END LOOP;
END;
```

- A la fin du bloc, **produitSerie** est 10! (3628800) et **valeurEntiere** est 10

Répéter : boucles imbriquées

```
DECLARE
   …
BEGIN
   …
   <<boucleExterne>>
   LOOP
   …
   LOOP
        …
        EXIT boucleExterne WHEN …; /* quitter
   boucle externe */
        …
        EXIT WHEN …; -- quitter boucle interne
   END LOOP;
   …
   END LOOP;
   …
END;
```


Structures répétitives : pour (FOR)

- Syntaxe :

```
[<<etiquette>>]
FOR compteur IN [REVERSE] valInf..valSup LOOP
    instructions;
END LOOP;
```

- Les instructions sont exécutées une fois pour chaque valeur prévue du compteur
- Les bornes `valInf,valSup` sont évaluées une seule fois
- Après chaque itération, le compteur est incrémenté de 1 (ou décrémenté si `REVERSE`) ; la boucle est terminée quand le compteur sort de l'intervalle spécifié

Pour (FOR) : exemple

```
DECLARE
   valeurEntiere    INTEGER := 1;
   produitSerie     INTEGER := 1;
BEGIN
   FOR valeurEntiere IN 1..10 LOOP
       produitSerie := produitSerie * valeurEntiere;
   END LOOP;
END;
```

- A la fin du bloc, `produitSerie` est 10! (3628800) et `valeurEntiere` est 10

Interaction avec la base

- Interrogation directe des données : doit retourner 1 enregistrement (sinon `TOO_MANY_ROWS` ou `NO_DATA_FOUND`)

```
SELECT listeColonnes INTO var1PLSQL [, var2PLSQL …]
FROM nomTable [WHERE condition];
```

- Manipulation des données :
  - Insertions :

```
INSERT INTO nomTable (liste colonnes) VALUES (liste expressions);
```

  - Modifications :

```
UPDATE nomTable SET nomColonne = expression [WHERE condition];
```

  - Suppressions :

```
DELETE FROM nomTable [WHERE condition];
```

Exemple

```
DECLARE
  ancienPilote Pilote%ROWTYPE;
  agePilote Pilote.Age%TYPE;
  nomPilote Pilote.Nom%TYPE;
BEGIN
  SELECT * INTO ancienPilote FROM Pilote
       WHERE Matricule = 1;
  DELETE FROM Pilote WHERE Matricule = 1;
  SELECT Age, Nom INTO agePilote, nomPilote FROM Pilote
WHERE Matricule = 2;
  INSERT INTO Pilote (Nom, Ville, Age)
       VALUES ('Dupond', 'Lille', agePilote);
  UPDATE Pilote SET Age = agePilote+1 WHERE Matricule = 2;
END;
```


Curseurs

- Les échanges entre l'application et la base sont réalisés grâce à des curseurs : zones de travail capables de stocker un ou plusieurs enregistrements et de gérer l'accès à ces enregistrements
- Types de curseurs :
  - Curseurs implicites :
    - Déclarés implicitement et manipulés par SQL
    - Pour toute requête SQL du LMD et pour les interrogations qui retournent un seul enregistrement
  - Curseurs explicites :
    - Déclarés et manipulés par l'utilisateur
    - Pour les interrogations qui retournent plus d'un enregistrement

Curseurs implicites

- Nom : `SQL` ; actualisé après chaque requête LMD et chaque SELECT non associé à un curseur explicite
- À travers ses attributs, permet au programme PL/SQL d'obtenir des infos sur l'exécution des requêtes :
  - `SQL%FOUND` : `TRUE` si la dernière requête LMD a affecté au moins 1 enregistrement
  - `SQL%NOTFOUND` : `TRUE` si la dernière requête LMD n'a affecté aucun enregistrement
  - `SQL%ROWCOUNT` : nombre de lignes affectées par la requête LMD
- Exemple :

```
DELETE FROM pilote WHERE Age >= 55;
DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' pilotes partent
    à la retraite ');
```


Curseurs explicites : instructions

- Définition = association à une requête ; dans la section des déclarations :

```
CURSOR nomCurseur IS SELECT … FROM … WHERE …;
```

- Ouverture du curseur (avec exécution de la requête) :

```
OPEN nomCurseur;
```

- Chargement de l'enregistrement courant et positionnement sur l'enregistrement suivant :

```
FETCH nomCurseur INTO listeVariables;
```

- Fermeture du curseur (avec libération zone mémoire) :

```
CLOSE nomCurseur;
```

Curseurs explicites : attributs

- `nomCurseur%ISOPEN` : `TRUE` si le curseur est ouvert

  `IF nomCurseur%ISOPEN THEN … END IF;`
- `nomCurseur%FOUND` : `TRUE` si le dernier accès (`FETCH`) a retourné une ligne

  `IF nomCurseur%FOUND THEN … END IF;`
- `nomCurseur%NOTFOUND` : `TRUE` si le dernier accès (`FETCH`) n'a pas retourné de ligne (fin curseur)

  `IF nomCurseur%NOTFOUND THEN … END IF;`
- `nomCurseur%ROWCOUNT` : nombre total de lignes traitées jusqu'à présent (par ex. nombre de `FETCH`)

  ```
  DBMS_OUTPUT.PUT_LINE('Lignes traitées : ' ||
  nomCurseur%ROWCOUNT);
  ```


Curseurs explicites : exemple

```
DECLARE
   CURSOR curseur1 IS SELECT Salaire FROM pilote
   WHERE (Age >= 30 AND Age <= 40);
   salairePilote     pilote.Salaire%TYPE;
   sommeSalaires     NUMBER(11,2) := 0;
   moyenneSalaires   NUMBER(11,2);
BEGIN
   OPEN curseur1;
   LOOP
   FETCH curseur1 INTO salairePilote;
   EXIT WHEN curseur1%NOTFOUND;
   sommeSalaires := sommeSalaires + salairePilote;
   END LOOP;
   moyenneSalaires := sommeSalaires / curseur1%ROWCOUNT;
   CLOSE curseur1;
   DBMS_OUTPUT.PUT_LINE('Moyenne salaires 30 à 40 ans : '
       || moyenneSalaires);
END;
```

Curseurs explicites paramétrés

- Objectif : paramétrer la requête associée à un curseur (procédures, éviter de multiplier les curseurs similaires)
- Syntaxe de définition :

  `CURSOR nomCurseur(param1[, param2, …]) IS …;`

  avec, pour chaque paramètre,

  `nomPar [IN] type [{:= | DEFAULT} valeur];`

  (`nomPar` est inconnu en dehors de la définition !)
- Les valeurs des paramètres sont transmises à l'ouverture du curseur :

  `OPEN nomCurseur(valeurPar1[, valeurPar2, …]);`
- Il faut évidemment fermer le curseur avant de l'appeler avec d'autres valeurs pour les paramètres

Curseurs paramétrés : exemple

```
DECLARE
   CURSOR curseur1(ageInf NUMBER, ageSup NUMBER) IS
   SELECT Salaire FROM pilote
   WHERE (Age >= ageInf AND Age <= ageSup);
   salairePilote      pilote.Salaire%TYPE;
   sommeSalaires      NUMBER(11,2) := 0;
   moyenneSalaires    NUMBER(11,2);
BEGIN
   OPEN curseur1(30,40);
   LOOP
   FETCH curseur1 INTO salairePilote;
   EXIT WHEN curseur1%NOTFOUND;
   sommeSalaires := sommeSalaires + salairePilote;
   END LOOP;
   moyenneSalaires := sommeSalaires / curseur1%ROWCOUNT;
   CLOSE curseur1;
   DBMS_OUTPUT.PUT_LINE(…);
END;
```

Boucle FOR avec curseur

- Exécuter des instructions pour tout enregistrement retourné par la requête associée à un curseur :

```
DECLARE
CURSOR nomCurseur(…) IS …;
BEGIN
/* un seul OPEN nomCurseur(…), implicite,
puis un FETCH à chaque itération ;
   enregistrement déclaré implicitement de
type nomCurseur%ROWTYPE */
FOR enregistrement IN nomCurseur(…) LOOP
…
END LOOP; -- implicitement CLOSE nomCurseur
…
END;
```


Boucle FOR avec curseur : exemple

```
DECLARE
   CURSOR curseur1(ageInf NUMBER, ageSup NUMBER) IS
     SELECT Salaire FROM pilote
     WHERE (Age >= ageInf AND Age <= ageSup);
   sommeSalaires     NUMBER(11,2) := 0;
   moyenneSalaires   NUMBER(11,2) := 0;
   nbPilotes  NUMBER(11,0) := 0;
BEGIN
   FOR salairePilote IN curseur1(30,40) LOOP
     sommeSalaires := sommeSalaires + salairePilote;
     nbPilotes := curseur1%ROWCOUNT;
   END LOOP;
   /* curseur1 fermé, plus possible de lire %ROWCOUNT */
   IF nbPilotes > 0 THEN
     moyenneSalaires := sommeSalaires / nbPilotes;
   END IF;
   DBMS_OUTPUT.PUT_LINE(…);
END;
```

Curseurs et verrouillage

- Objectif : lorsqu'un curseur est ouvert, verrouiller l'accès aux colonnes référencées des lignes retournées par la requête afin de pouvoir les modifier
- Syntaxe de déclaration :

```
CURSOR nomCurseur[(parametres)] IS
    SELECT listeColonnes1 FROM nomTable
    WHERE condition
    FOR UPDATE [OF listeColonnes2]
          [NOWAIT | WAIT intervalle]
```

- Absence de `OF` : toutes les colonnes sont verrouillées
- Absence de `NOWAIT | WAIT intervalle` : on attend (indéfiniment) que les lignes visées soient disponibles

Modification des lignes verrouillées

- Restrictions : `DISTINCT`, `GROUP BY`, opérateurs ensemblistes et fonctions de groupe ne sont pas utilisables dans les curseurs `FOR UPDATE`
- Modification de la ligne courante d'un curseur :

```
UPDATE nomTable SET modificationsColonnes
WHERE CURRENT OF nomCurseur;
```

- Suppression de la ligne courante d'un curseur :

```
DELETE FROM nomTable
    WHERE CURRENT OF nomCurseur;
```

Modification des lignes : exemple

```
DECLARE
   CURSOR curseur1(villePrime pilote.Ville%TYPE) IS
        SELECT Salaire FROM pilote
        WHERE (Ville = villePrime) FOR UPDATE;
   prime pilote.Salaire%TYPE := 5000;
BEGIN
   -- salaireActuel : implicitement curseur1%ROWTYPE
   FOR salaireActuel IN curseur1('Paris') LOOP
        UPDATE pilote
        SET Salaire = salaireActuel.Salaire + prime
        WHERE CURRENT OF curseur1;
   END LOOP;
END;
```


Variables curseurs

- Éviter de manipuler de nombreux curseurs associés à des requêtes différentes : définir des variables curseurs, associées dynamiquement aux requêtes
- Syntaxe de déclaration :

```
TYPE nomTypeCurseur IS REF CURSOR [RETURN type];
nomVariableCurseur nomTypeCurseur;
```

- Le curseur est typé si `RETURN type` est présente (en général, `type` est `nomDeTable%ROWTYPE`) ; ne peut être associé qu'aux requêtes ayant le même type de retour
- Association à une requête et ouverture :

```
OPEN nomVariableCurseur FOR
     SELECT … FROM … WHERE …;
```

Variables curseurs : exemple

```
DECLARE
   TYPE curseurNonType IS REF CURSOR;
   curseur1 curseurNonType;
   salaireInf pilote.Salaire%TYPE := 25000;
   salairePilote pilote.Salaire%TYPE;
   ageInf pilote.Age%TYPE := 27;
BEGIN
   OPEN curseur1 FOR SELECT Salaire FROM pilote
   WHERE Salaire <= salaireInf;
   LOOP
   FETCH curseur1 INTO salairePilote;
   EXIT WHEN curseur1%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE('Salaire : '|| salairePilote);
   END LOOP;
   CLOSE curseur1;
   OPEN curseur1 FOR SELECT Age FROM pilote
   WHERE Age <= ageInf;
   …
END;
```


Sous-programmes

- Blocs PL/SQL nommés et paramétrés
  - Procédures : réalisent des traitements ; peuvent retourner ou non un ou plusieurs résultats
  - Fonctions : retournent un résultat unique ; peuvent être appelées dans des requêtes SQL
- Sont stockés avec la base
- Intérêt de l'utilisation de sous-programmes :
  - Productivité de la programmation : modularité (avantage pour la conception et la maintenance), réutilisation
  - Intégrité : regroupement des traitements dépendants
  - Sécurité : gestion des droits sur les programmes qui traitent les données
- La récursivité est permise (à utiliser avec précaution) !

Procédures

- Syntaxe :

```
PROCEDURE nomProcedure
[(par1 [IN | OUT | IN OUT] [NOCOPY] type1
[{:= | DEFAULT} expression]
[, par2 [IN | OUT | IN OUT] [NOCOPY] type2
[{:= | DEFAULT} expression … )]
{IS | AS} [declarations;]
BEGIN
instructions;
[EXCEPTION
traitementExceptions;]
END[nomProcedure];
```

- Se termine à la fin du bloc ou par une instruction `RETURN`

Fonctions

- Syntaxe :

```
FUNCTION nomFonction
    [(par1 [IN | OUT | IN OUT] [NOCOPY] type1
          [{:= | DEFAULT} expression]
    [, par2 [IN | OUT | IN OUT] [NOCOPY] type2
          [{:= | DEFAULT} expression … )]
    RETURN typeRetour
    {IS | AS} [declarations;]
BEGIN
    instructions;
[EXCEPTION
    traitementExceptions;]
END[nomFonction];
```

- Se termine obligatoirement par `RETURN` qui doit renvoyer une valeur de type `typeRetour`

Paramètres de sous-programme

- Types de paramètres :
  - Entrée (**IN**) : on ne peut pas lui affecter une valeur dans le sous-programme ; le paramètre effectif associé peut être une constante, une variable ou une expression ; toujours passé par référence !
  - Sortie (**OUT**) : on ne peut pas l'affecter (à une variable ou à lui-même) dans le sous-programme ; le paramètre effectif associé doit être une variable ; par défaut (sans **NOCOPY**) passé par valeur !
  - Entrée et sortie (**IN OUT**) : le paramètre effectif associé doit être une variable ; par défaut (sans **NOCOPY**) passé par valeur !
- **NOCOPY** : passage par référence de paramètre **OUT | IN OUT**, utile pour paramètres volumineux ; attention aux effets de bord !
- Paramètres **OUT | IN OUT** pour **FUNCTION** : mauvaise pratique qui risque de produire des effets de bord. Lorsqu'une fonction doit être appelée depuis SQL, seuls des paramètres **IN** sont autorisés !

Définition de sous-programme

- Définition de procédure ou fonction locale dans un bloc PL/SQL : à la fin de la section de déclarations

```
DECLARE …
PROCEDURE nomProcedure …; END nomProcedure;
FUNCTION nomFonction …; END nomFonction;
BEGIN … END;
```

- Définition de procédure ou fonction stockée (isolée ou dans un paquetage) :

```
CREATE [OR REPLACE] PROCEDURE nomProcedure …;
END nomProcedure;
CREATE [OR REPLACE] FUNCTION nomFonction …;
END nomFonction;
```

Manipulation de sous-programme

- Création ou modification de sous-programme :

```
CREATE [OR REPLACE] {PROCEDURE | FUNCTION} nom …
```

- Oracle recompile automatiquement un sous-programme quand la structure d'un objet dont il dépend a été modifiée
  - Pour une compilation manuelle :

```
ALTER {PROCEDURE | FUNCTION} nom COMPILE
```

  - Affichage des erreurs de compilation sous SQL*Plus :

```
SHOW ERRORS
```

- Suppression de sous-programme :

```
DROP {PROCEDURE | FUNCTION} nom
```


Appel de sous-programme

- Appel de procédure depuis un bloc PL/SQL :

```
nomProcedure(listeParEffectifs);
```

- Appel de procédure stockée depuis SQL*Plus :

```
SQL> EXECUTE nomProcedure(listeParEffectifs);
```

- Appel de fonction depuis un bloc PL/SQL : introduction dans une instruction PL/SQL ou SQL de `nomFonction(listeParEffectifs)`
- Appel de fonction stockée depuis SQL*Plus : introduction dans une instruction SQL de `nomFonction(listeParEffectifs)`

Procédure locale : exemple

```
DECLARE
   nbPilotesPrimes INTEGER := 0;
   PROCEDURE accordPrime(villePrime IN pilote.Ville%TYPE,
                         valPrime  IN NUMBER,
                         nbPilotes OUT INTEGER) IS
   BEGIN
       UPDATE pilote SET Salaire = Salaire + valPrime
              WHERE (Ville = villePrime);
       nbPilotes := SQL%ROWCOUNT;
   END accordPrime;
BEGIN
   accordPrime('Toulouse', 1000, nbPilotesPrimes);
   DBMS_OUTPUT.PUT_LINE('Nombre pilotes primés : ' ||
                        nbPilotesPrimes);
END;
```


Procédure stockée : exemple

```
CREATE OR REPLACE PROCEDURE
    accordPrime(villePrime IN pilote.Ville%TYPE,
                valPrime IN NUMBER,
                nbPilotes OUT INTEGER) IS
BEGIN
    UPDATE pilote SET Salaire = Salaire + valPrime
           WHERE (Ville = villePrime);
    nbPilotes := SQL%ROWCOUNT;
END accordPrime;
```

- Appel depuis SQL*Plus :

```
SQL> EXECUTE accordPrime('Gap',1000,nbPilotesPrimes);
```

- Appel depuis un bloc PL/SQL :

```
BEGIN …
    accordPrime('Gap', 1000, nbPilotesPrimes);
    …
END;
```

Fonction locale : exemple

```
DECLARE
   salaireMoyInt pilote.Salaire%TYPE;
   FUNCTION moyInt(ageInf IN pilote.Age%TYPE,
                   ageSup IN pilote.Age%TYPE)
       RETURN pilote.Salaire%TYPE IS
   BEGIN
       -- la variable du parent est visible ici !
       SELECT AVG(Salaire) INTO salaireMoyInt FROM pilote
              WHERE (Age >= ageInf AND Age <= ageSup);
       RETURN salaireMoyInt;
   END moyInt;
BEGIN
   salaireMoyInt := moyInt(32,49);
   DBMS_OUTPUT.PUT_LINE('Salaire moyen pour âge 32-49 '
                        || salaireMoyInt);
END;
```


Fonction stockée : exemple

```
CREATE OR REPLACE FUNCTION
moyInt(ageInf IN pilote.Age%TYPE,
ageSup IN pilote.Age%TYPE)
RETURN pilote.Salaire%TYPE IS
salaireMoyInt pilote.Salaire%TYPE;
BEGIN
SELECT AVG(Salaire) INTO salaireMoyInt FROM pilote
WHERE (Age >= ageInf AND Age <= ageSup);
RETURN salaireMoyInt;
END moyInt;
```

- Appel depuis SQL*Plus :

```
SQL> SELECT … FROM pilote WHERE (Salaire > moyInt(32,49));
```

- Appel depuis un bloc PL/SQL :

```
salaireMoyenIntervalle := moyInt(32,49);
…
```

Paquetages

- Paquetage = regroupement de variables, curseurs, fonctions, procédures, etc. PL/SQL qui fournissent un ensemble cohérent de services
- Distinction entre ce qui est accessible depuis l'extérieur et ce qui n'est accessible qu'à l'intérieur du paquetage → encapsulation
- Structure :
  - Section de spécification : déclarations des variables, curseurs, sous-programmes accessibles depuis l'extérieur
  - Section d'implémentation : code des sous-programmes accessibles depuis l'extérieur + sous-programmes accessibles en interne (privés)

Section de spécification

- Syntaxe :

```
CREATE [OR REPLACE] PACKAGE nomPaquetage {IS | AS}
    [declarationTypeRECORDpublique …; ]
    [declarationTypeTABLEpublique …; ]
    [declarationSUBTYPEpublique …; ]
    [declarationRECORDpublique …; ]
    [declarationTABLEpublique …; ]
    [declarationEXCEPTIONpublique …; ]
    [declarationCURSORpublique …; ]
    [declarationVariablePublique …; ]
    [declarationFonctionPublique …; ]
    [declarationProcedurePublique …; ]
END [nomPaquetage];
```

Spécification : exemple

```
CREATE PACKAGE gestionPilotes AS
   …
   CURSOR accesPilotes(agePilote pilote.Age%TYPE)
       RETURN pilote%ROWTYPE;
   FUNCTION moyInt(ageInf IN pilote.Age%TYPE,
                   ageSup IN pilote.Age%TYPE)
       RETURN pilote.Salaire%TYPE;
   PROCEDURE accordPrime(villePrime IN pilote.Ville%TYPE,
             valPrime IN NUMBER,nbPilotes OUT INTEGER);
   …
END gestionPilotes;
```


Section d'implémentation

- Syntaxe :

```
CREATE [OR REPLACE] PACKAGE BODY nomPaquetage
    {IS | AS}
    [declarationTypePrive …; ]
    [declarationObjetPrive …; ]
    [definitionFonctionPrivee …; ]
    [definitionProcedurePrivee …; ]
    [instructionsFonctionPublique …; ]
    [instructionsProcedurePublique …; ]
END [nomPaquetage];
```

Implémentation : exemple

```
CREATE PACKAGE BODY gestionPilotes AS
   CURSOR accesPilotes(agePilote pilote.Age%TYPE)
        RETURN pilote%ROWTYPE
        IS SELECT * FROM pilote WHERE Age = agePilote;
   FUNCTION moyInt(ageInf IN pilote.Age%TYPE,
                   ageSup IN pilote.Age%TYPE)
        RETURN pilote.Salaire%TYPE IS
   BEGIN
        SELECT AVG(Salaire) INTO salaireMoyInt FROM pilote
               WHERE (Age >= ageInf AND Age <= ageSup);
        RETURN salaireMoyInt;
   END moyInt;
   PROCEDURE accordPrime(villePrime IN pilote.Ville%TYPE,
               valPrime IN NUMBER, nbPilotes OUT INTEGER) IS
   BEGIN
        UPDATE pilote SET Salaire = Salaire + valPrime
               WHERE (Ville = villePrime);
        nbPilotes := SQL%ROWCOUNT;
   END accordPrime;
END gestionPilotes;
```


Référence au contenu d'un paquetage

- Naturellement, seuls les objets et sous-programmes publics peuvent être référencés depuis l'extérieur
- Syntaxe :

  `nomPaquetage.nomObjet`

  `nomPaquetage.nomSousProgramme(…)`
- Les paquetages autorisent la surcharge des noms de fonctions ou de procédures
  - Toutes les versions (qui diffèrent par le nombre et/ou le type des paramètres) doivent être déclarées dans la section de spécification
  - Les références seront les mêmes pour les différentes versions, le choix est fait en fonction des paramètres effectifs

Manipulation d'un paquetage

- Re-compilation d'un paquetage :
  - Utiliser `CREATE OR REPLACE PACKAGE` et terminer par `/` une des sections (sous SQL*Plus)
  - La modification d'une des sections entraîne la re-compilation automatique de l'autre section
  - Affichage des erreurs de compilation avec SQL*Plus : `SHOW ERRORS`
- Suppression d'un paquetage :

```
DROP BODY nomPaquetage;
DROP nomPaquetage;
```


Exceptions

- Les exceptions correspondent à des conditions d'erreur constatées lors de l'exécution d'un programme PL/SQL ou de requêtes SQL
- PL/SQL propose un mécanisme de traitement pour les exceptions déclenchées, permettant d'éviter l'arrêt systématique du programme
- Les programmeurs peuvent se servir de ce mécanisme non seulement pour les erreurs Oracle, mais pour toute condition qu'ils peuvent définir (→ communication plus riche entre appelants et appelés ou blocs imbriqués)

Traitement des exceptions

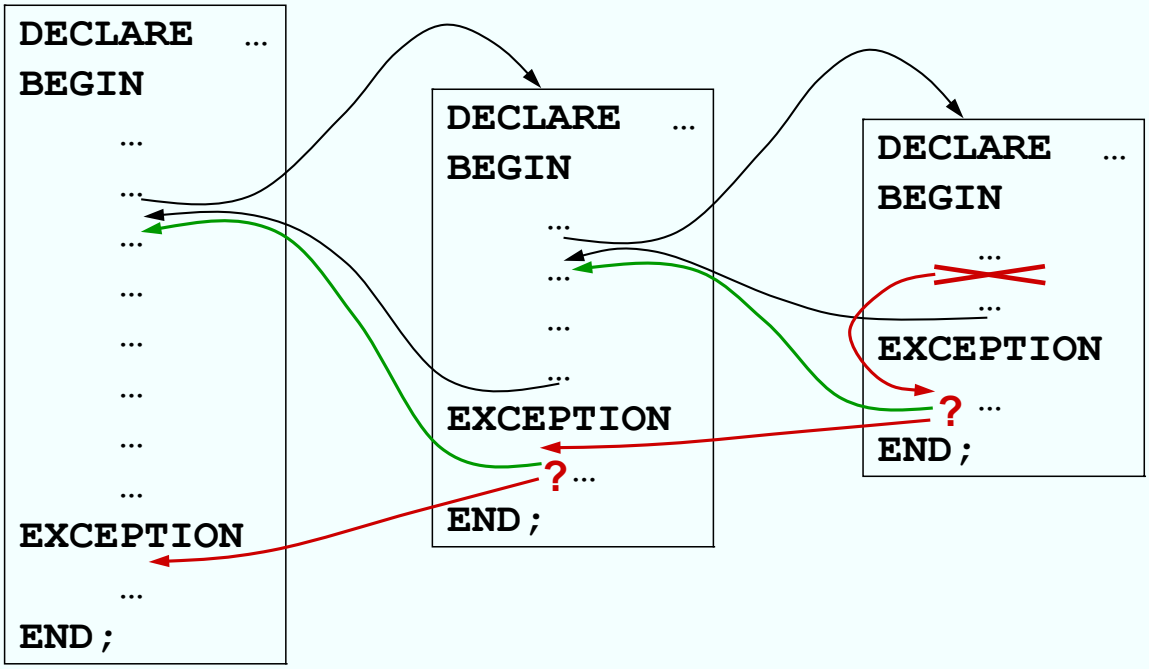
- Syntaxe :

```
…
EXCEPTION
    WHEN nomException1 [OR nomException2 …] THEN
          instructions1;
    WHEN nomException3 [OR nomException4 …] THEN
          instructions3;
    WHEN OTHERS THEN
          instructionsAttrapeTout;
END;
```


Suites de l'apparition d'une exception

1. Aucun traitement n'est prévu : le programme s'arrête
2. Un traitement est prévu :
   - L'exécution du bloc PL/SQL courant est abandonnée
   - Le traitement de l'exception est recherché dans la section **EXCEPTION** associée au bloc courant, sinon dans les blocs parents (englobant le bloc courant) ou le programme appelant
   - L'exception est traitée suivant les instructions trouvées (spécifiques ou attrape-tout)
   - L'exécution se poursuit normalement dans le bloc parent (ou le programme appelant) de celui qui a traité l'exception

Suites de l'apparition d'une exception



Mécanismes de déclenchement

1. Déclenchement automatique suite à l'apparition d'une des erreurs prédéfinies Oracle : `VALUE_ERROR`, `ZERO_DIVIDE`, `TOO_MANY_ROWS`, etc. ou non nommées
2. Déclenchement programmé (permet au programmeur d'exploiter le mécanisme de traitement des erreurs) :
   - Déclaration (dans `DECLARE`) : `nomException EXCEPTION;`
   - Déclenchement (dans `BEGIN`) : `RAISE nomException;`

   Il est également possible de déclencher avec `RAISE` une exception prédéfinie Oracle !

   `RAISE nomExceptionPreDefinie;`

Exceptions non nommées

- Traitement non spécifique grâce à l'attrape-tout :

```
WHEN OTHERS THEN instructions;
```

Exemple :

```
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE(SQLERRM || '(' ||
   SQLCODE || ')');
```

- Identification et traitement spécifique :
  - Identification dans la section **DECLARE** :

```
nomAPrendre EXCEPTION;
PRAGMA EXCEPTION_INIT(nomAPrendre, numErrOracle);
```

  - Traitement spécifique :

```
WHEN nomAPrendre THEN instructions;
```


Mécanismes de déclenchement (2)

- Propagation explicite au parent ou à l'appelant après traitement local :

```
WHEN nomException1 THEN
…;
RAISE; -- la même exception nomException1
WHEN autreException THEN …
```

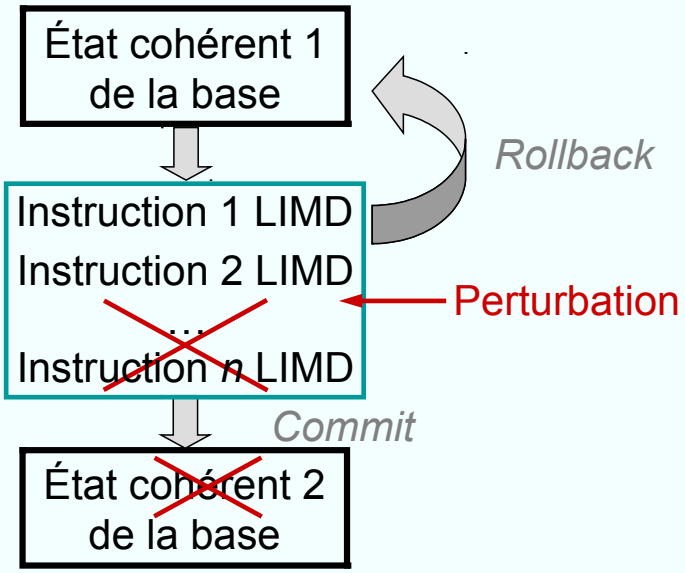
- Déclenchement avec message et code d'erreur personnalisé :

```
RAISE_APPLICATION_ERROR(numErr, messageErr,
[TRUE | FALSE]);
```

  - Utilise le mécanisme des exceptions non nommées, avec **SQLERRM** et **SQLCODE**
  - **TRUE** : mise dans une pile d'erreurs à propager (par défaut) ; **FALSE** : remplace les erreurs précédentes dans la pile

Transactions

- Objectif : rendre indivisible une suite de manipulations des données



Transactions : exigences

- Indivisibilité (atomicité) des instructions regroupées dans une transaction
- Maintien de la cohérence : passage d'un état initial cohérent à un état cohérent (qui peut être le même que l'état initial si la transaction ne s'est pas terminée avec succès)
- Isolation entre plusieurs transactions (sérialisation) ; pour des raisons de performance, des niveaux intermédiaires d'isolation peuvent être employés
- Durabilité des opérations : une fois la transaction terminée avec succès, le résultat des opérations qu'elle regroupe ne peut pas être remis en question

Transactions : contrôle

- Début d'une transaction : pas d'instruction explicite
  - À la première instruction SQL après le **BEGIN** du bloc
  - À la première instruction SQL après la fin d'une autre transaction
- Fin explicite d'une transaction :
  - Avec succès : **COMMIT [WORK];**
  - Échec : **ROLLBACK [WORK];**
- Fin implicite d'une transaction :
  - Avec succès : à la fin normale de la session ou à la première instruction du LDD ou LCD
  - Échec : à la fin anormale d'une session

Transactions : contrôle (2)

- Les points de validation intermédiaires rendent possible l'annulation d'une partie des opérations :

```
SAVEPOINT nomPoint;      -- insertion point de validation
ROLLBACK TO nomPoint;    -- retour à l'état d'avant nomPoint
```

- Remarques :
  - Oracle place chaque instruction SQL dans une transaction implicite ; si l'instruction échoue (par exemple, une exception est levée), l'état redevient celui qui précède l'instruction
  - La sortie d'un sous-programme suite à une exception non traitée ne produit pas de **ROLLBACK** implicite des opérations réalisées dans le sous-programme !

Transactions : exemple

```
DECLARE …
BEGIN    -- Échanger les heures de départ de 2 vols
   SELECT * INTO vol1 FROM Vol WHERE Numvol = numVol1;
   SELECT * INTO vol2 FROM Vol WHERE Numvol = numVol2;
   dureeVol1 := vol1.Heure_arrivee - vol1.Heure_depart;
   dureeVol2 := vol2.Heure_arrivee - vol2.Heure_depart;
   UPDATE Vol SET Heure_depart = vol1.Heure_depart,
       Heure_arrivee = vol1.Heure_depart +     dureeVol2
   WHERE Numvol = numVol2;
   UPDATE Vol SET Heure_depart = vol2.Heure_depart,
       Heure_arrivee = vol2.Heure_depart +     dureeVol1
   WHERE Numvol = numVol1;
   COMMIT;
EXCEPTION
   WHEN OTHERS THEN
   ROLLBACK;
END;
```


Transactions : exemple (2)

```
BEGIN
   UPDATE …
   SAVEPOINT Point1;
   <<Boucle2>>
   LOOP -- essais multiples, boucle si exception levée
      BEGIN  -- bloc PL/SQL imbriqué
         UPDATE …
         UPDATE …
         EXIT Boucle2;  -- quitter la boucle si réussite
      EXCEPTION
         WHEN OTHERS THEN ROLLBACK TO Point1;
      END;
   END LOOP;
   COMMIT;
EXCEPTION
   WHEN OTHERS THEN
   ROLLBACK;
END;
```

Déclencheurs

- Déclencheur (*trigger*) = programme dont l'exécution est déclenchée (*fired*) par un événement (un déclencheur n'est pas appelé explicitement par une application)
- Événements déclencheurs :
  - Instruction LMD : **INSERT**, **UPDATE**, **DELETE**
  - Instruction LDD : **CREATE**, **ALTER**, **DROP**
  - Démarrage ou arrêt de la base
  - Connexion ou déconnexion d'utilisateur
  - Erreur d'exécution
- Usage fréquent : implémenter les règles de gestion non exprimables par les contraintes au niveau des tables

Définition d'un déclencheur

- Structure :
  1. Description de l'événement déclencheur
  2. Éventuelle condition supplémentaire à satisfaire pour déclenchement
  3. Description du traitement à réaliser après déclenchement
- Syntaxe pour déclenchement sur instruction LMD :

```
CREATE [OR REPLACE] TRIGGER nomDeclencheur
{BEFORE | AFTER | INSTEAD OF}
{DELETE | INSERT | UPDATE [OF colonne 1, …] [OR …]}
ON {nomTable | nomVue}
[REFERENCING {OLD [AS] nomAncien | NEW [AS] nomNouveau
       | PARENT [AS] nomParent } …]
[FOR EACH ROW]
[WHEN conditionSupplementaire]
{[DECLARE …] BEGIN … [EXCEPTION …] END;
| CALL nomSousProgramme(listeParametres)}
```

Déclencheurs sur instruction LMD

- Quand le déclenchement a lieu (si concevable) :
  - Avant l'événement : **BEFORE**
  - Après l'événement : **AFTER**
  - À la place de l'événement : **INSTEAD OF** (uniquement pour vues multi-tables)
- Description de l'événement (pour instructions LMD) :
  - La ou les (**OR**) instructions,
  - Si l'événement concerne des colonnes spécifiques (**[OF colonne 1, …]**) ou non,
  - Le nom de la table (ou vue) (**ON {nomTable | nomVue}**)

Déclencheurs sur instruction LMD

- Changement des noms par défaut : **REFERENCING**
  - **:OLD** désigne un enregistrement à effacer (déclencheur sur **DELETE**, **UPDATE**) : **REFERENCING OLD AS nomAncien**
  - **:NEW** désigne un enregistrement à insérer (déclencheur sur **INSERT**, **UPDATE**) : **REFERENCING NEW AS nomNouveau**
  - **:PARENT** pour des *nested tables* : **REFERENCING PARENT AS nomParent**
- **FOR EACH ROW** :
  - Avec **FOR EACH ROW**, 1 exécution par ligne concernée par l'instruction LMD (*row trigger*)
  - Sans **FOR EACH ROW**, 1 exécution par instruction LMD (*statement trigger*)

Déclencheurs et exceptions

- Exception levée lors de l'exécution d'un déclencheur
  - Si présence de traitement de l'exception dans le déclencheur : poursuite avec l'exécution du traitement de l'exception
  - Si absence de traitement de l'exception dans le déclencheur
    - Annulation des opérations de l'instruction ayant déclenché le déclencheur
    - Annulation des opérations déjà effectuées dans le déclencheur
    - Fin du déclencheur

Base exemple

- Tables de la base (la clé primaire est soulignée) :

  immeuble (Adr, NbEtg, DateConstr, NomGerant)

  appart (Adr, Num, Type, Superficie, Etg, NbOccup)

  personne (Nom, Age, CodeProf)

  occupant (Adr, NumApp, NomOccup, DateArrivee, DateDepart)

  propriete (Adr, NomProprietaire, QuotePart)
- Exemples de contraintes à satisfaire :
  - Intégrité référentielle (clé étrangère ← clé primaire) : lors de la création de la table `propriete` : `CONSTRAINT prop_pers FOREIGN KEY (NomProprietaire) REFERENCES personne (Nom)`
  - Condition entre colonnes : lors de la création de la table `occupant` : `CONSTRAINT dates CHECK (DateArrivee < DateDepart)`
  - Règles de gestion : somme quotes-parts pour une propriété = 100 ; date construction immeuble < dates arrivée de tous ses occupants… → déclencheurs

Déclencheur sur INSERT

- Pour un nouvel occupant, vérifie si occupant.DateArrivee > immeuble.DateConstr (FOR EACH ROW est nécessaire pour avoir accès à :NEW, l'enregistrement ajouté) :

```
CREATE TRIGGER TriggerVerificationDates
BEFORE INSERT ON occupant FOR EACH ROW
DECLARE
Imm immeuble%ROWTYPE;
BEGIN
SELECT * INTO Imm FROM immeuble
WHERE immeuble.Adr = :NEW.Adr;
IF :NEW.DateArrivee < Imm.DateConstr THEN
RAISE_APPLICATION_ERROR(-20100, :NEW.Nom || ' arrivé
avant construction immeuble ' ||      Imm.Adr);
END IF;
END;
```

- Événement déclencheur : INSERT INTO occupant … VALUES …;

Déclencheur sur INSERT (2)

- Si chaque nouvel immeuble doit avoir au moins un appartement, insérer un appartement après la création de l'immeuble (FOR EACH ROW est nécessaire pour avoir accès à :NEW, l'enregistrement ajouté) :

```
CREATE TRIGGER TriggerAppartInitial
    AFTER INSERT ON immeuble FOR EACH ROW
BEGIN
    INSERT INTO appart (Adr, Num, NbOccup)
    VALUES (:NEW.Adr, 1, 0);
END;
```

- Événement déclencheur : INSERT INTO immeuble … VALUES …;

Déclencheur sur DELETE

- Au départ d'un occupant, décrémente `appart.NbOccup` après effacement de l'occupant (`FOR EACH ROW` est nécessaire car la suppression peut concerner plusieurs occupants, ainsi que pour avoir accès à `:OLD`, l'enregistrement éliminé) :

```
CREATE TRIGGER TriggerDiminutionNombreOccupants
    AFTER DELETE ON occupant FOR EACH ROW
BEGIN
    UPDATE appart SET NbOccup = NbOccup - 1
           WHERE  appart.Adr = :OLD.Adr
           AND    appart.Num = :OLD.NumApp;
END;
```

- Événement déclencheur : `DELETE FROM occupant WHERE …;`

Déclencheur sur UPDATE

- En cas de modification d'un occupant, met à jour les valeurs de `appart.NbOccup` pour 2 les appartements éventuellement concernés (utilise à la fois `:OLD` et `:NEW`) :

```
CREATE TRIGGER TriggerMAJNombreOccupants
AFTER UPDATE ON occupant FOR EACH ROW
BEGIN
IF :OLD.Adr <>:NEW.Adr OR :OLD.NumApp <>:NEW.NumApp
THEN       UPDATE appart SET NbOccup = NbOccup - 1
WHERE      appart.Adr = :OLD.Adr
AND appart.Num = :OLD.NumApp;
UPDATE appart SET NbOccup = NbOccup + 1
WHERE      appart.Adr = :NEW.Adr
AND appart.Num = :NEW.NumApp;
END IF;
END;
```

- Événement déclencheur : `UPDATE occupant SET … WHERE …;`

Déclencheur sur conditions multiples

- Un seul déclencheur pour INSERT, DELETE, UPDATE qui met à jour les valeurs de `appart.NbOccup` pour le(s) appartement(s) concerné(s) :

```
CREATE TRIGGER TriggerCompletMAJNombreOccupants
AFTER INSERT OR DELETE OR UPDATE
ON occupant FOR EACH ROW
BEGIN
IF (INSERTING) THEN
…
ELSIF (DELETING) THEN
…
ELSIF (UPDATING) THEN
…
END IF;
END;
```

- Exemple d'événement déclencheur :

```
INSERT INTO occupant … VALUES …;
```


Déclencheurs sur instruction LDD

- Syntaxe pour déclenchement sur instruction LDD :

```
CREATE [OR REPLACE] TRIGGER nomDeclencheur
BEFORE | AFTER action [OR action …]
ON {[nomSchema.]SCHEMA | DATABASE}
{[DECLARE …] BEGIN … [EXCEPTION …] END;
| CALL nomSousProgramme(listeParametres)}
```

- `SCHEMA` : déclencheur valable pour schéma courant
- Quelques actions :
  - `CREATE, RENAME, ALTER, DROP` sur un objet du dictionnaire
  - `GRANT, REVOKE` privilège(s) à un utilisateur

Déclencheur sur LDD : exemple

- Enregistrement des changements de noms des objets du dictionnaire :

```
historiqueChangementNoms(Date,NomObjet,NomProprietaire)

CREATE TRIGGER TriggerHistoriqueChangementNoms
    AFTER RENAME ON DATABASE
BEGIN
    -- On se sert de 2 attributs système
    -- ora_dict_obj_name  : nom objet affecté
    -- ora_dict_obj_owner : propriétaire objet affecté

    INSERT INTO historiqueChangementNoms
        VALUES (SYSDATE, ora_dict_obj_name,
                         ora_dict_obj_owner);
END;
```


Déclencheurs d'instance

- Syntaxe :

```
CREATE [OR REPLACE] TRIGGER nomDeclencheur
BEFORE | AFTER evenement [OR evenement …]
ON {[nomSchema.]SCHEMA | DATABASE}
{[DECLARE …] BEGIN … [EXCEPTION …] END;
| CALL nomSousProgramme(listeParametres)}
```

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

Déclencheur d'instance : exemple

- Afficher l'identité de l'objet ayant provoqué un débordement :

```
CREATE TRIGGER TriggerDebordement
    AFTER SERVERERROR ON DATABASE
DECLARE
    eno NUMBER;
    typ VARCHAR2; owner VARCHAR2; ts VARCHAR2;
    obj VARCHAR2; subobj VARCHAR2;
BEGIN
    IF (space_error_info(eno,typ,owner,ts,obj,subobj)
=       TRUE) THEN
          DBMS_OUTPUT.PUT_LINE('L'objet' || obj ||
                 ' de ' || owner || ' a débordé !');
    END IF;
END;
```


Manipulation d'un déclencheur

- Tout déclencheur est actif dès sa compilation !
- Re-compilation d'un déclencheur après modification :

```
ALTER TRIGGER nomDeclencheur COMPILE;
```

- Désactivation de déclencheurs :

```
ALTER TRIGGER nomDeclencheur DISABLE;
ALTER TABLE nomTable DISABLE ALL TRIGGERS;
```

- Réactivation de déclencheurs :

```
ALTER TRIGGER nomDeclencheur ENABLE;
ALTER TABLE nomTable ENABLE ALL TRIGGERS;
```

- Suppression d'un déclencheur :

```
DROP TRIGGER nomDeclencheur;
```

Droits de création et manipulation

- Déclencheurs d'instance : privilège

  ADMINISTER DATABASE TRIGGER

- Autres déclencheurs :
  - Dans tout schéma : privilège **CREATE ANY TRIGGER**
  - Dans votre schéma : privilège **CREATE TRIGGER**

    (rôle **RESOURCE**)

ALL_TRIGGERS, DBA_TRIGGERS, USER_TRIGGERS

- **ALL_TRIGGERS** : déclencheurs de l'utilisateur et des tables de l'utilisateur
  - **OWNER**
  - **TRIGGER_NAME**
  - **TRIGGER_TYPE (BEFORE STATEMENT, BEFORE EACH ROW, …)**
  - **TRIGGERING_EVENT**
  - **TABLE_OWNER**
  - **BASE_OBJECT_TYPE (TABLE, VIEW, SCHEMA, DATABASE)**
  - **TABLE_NAME**
  - **COLUMN_NAME**
  - **REFERENCING_NAMES**
  - **WHEN_CLAUSE**
  - **STATUS (ENABLED, DISABLED)**
  - **DESCRIPTION**
  - **ACTION_TYPE (CALL, PL/SQL)**
  - **TRIGGER_BODY**
- **DBA_TRIGGERS** : de la base, **USER_TRIGGERS** : de l'utilisateur