

Procédures Stockées

Bases de Données

Raphaël Fournier-S'niehotta

Équipe Vertigo - Laboratoire CEDRIC

Conservatoire National des Arts & Métiers, Paris, France

Contenu du cours

- PL/SQL
 - Variables
 - Structures de contrôle
 - Interaction avec la base et Curseurs
 - Sous-programmes, paquetages
 - Exceptions
 - Transactions

(Supports de Nicolas Travers, d'après Michel Crucianu, Cédric du Mouza et Philippe Rigaux)

Bibliographie

Bales, D.K. *Java programming with Oracle JDBC*. O'Reilly, 2002.

Bizoi, R. *PL/SQL pour Oracle 10g*, Eyrolles. 2006.

Date, C. *Introduction aux bases de données*. Vuibert, 2004 (8^{ème} édition).

Gardarin, G. *Bases de données*, Eyrolles. 2003.

Reese, G. *JDBC et Java : guide du programmeur*. O'Reilly, 2001.

Soutou, C. *SQL pour Oracle*. Eyrolles, 2008 (3^{ème} édition).

PL/SQL

- *Procedural Language / Structured Query Language*
 - PL/SQL : langage **propriétaire** Oracle
 - Langage procédural :
 - Variables, boucles, tests, curseurs, fonctions/procédures, exceptions
- Syntaxe de PL/SQL inspirée du langage Ada (Pascal)
 - Avantages de SQL
 - Programmation en plus
- PL/SQL n'est pas très éloigné du langage normalisé *Persistent Stored Modules (PSM)*

PL/SQL

- Qui ?
 - DBA
 - Programmeur d'application de BD
- Existe dans d'autres SGBDR
 - *MySQL* : PL/SQL like
 - *Sybase* et *Microsoft SQL server* : Transact-SQL
 - *PostgreSQL* : PL/pgSQL
 - *DB2* (IBM) : SQL Procedural Language
- Documentation Oracle (en anglais)
http://download.oracle.com/docs/cd/B10501_01/appdev.920/a96624/toc.htm
- Documentation MySQL
<http://dev.mysql.com/doc/refman/5.0/fr/stored-procedure-syntax.html>

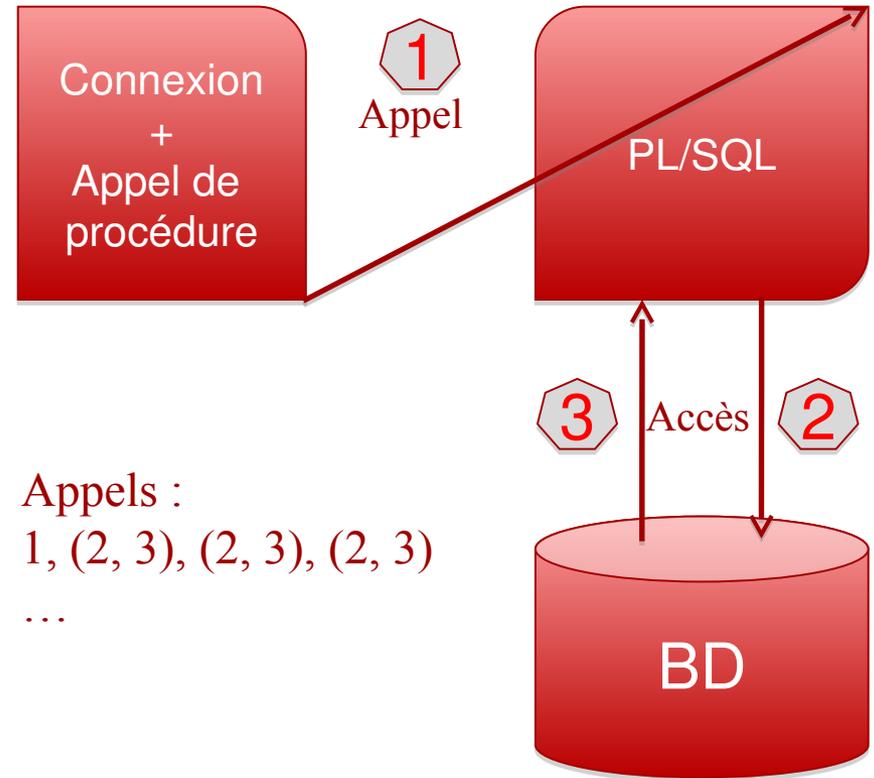
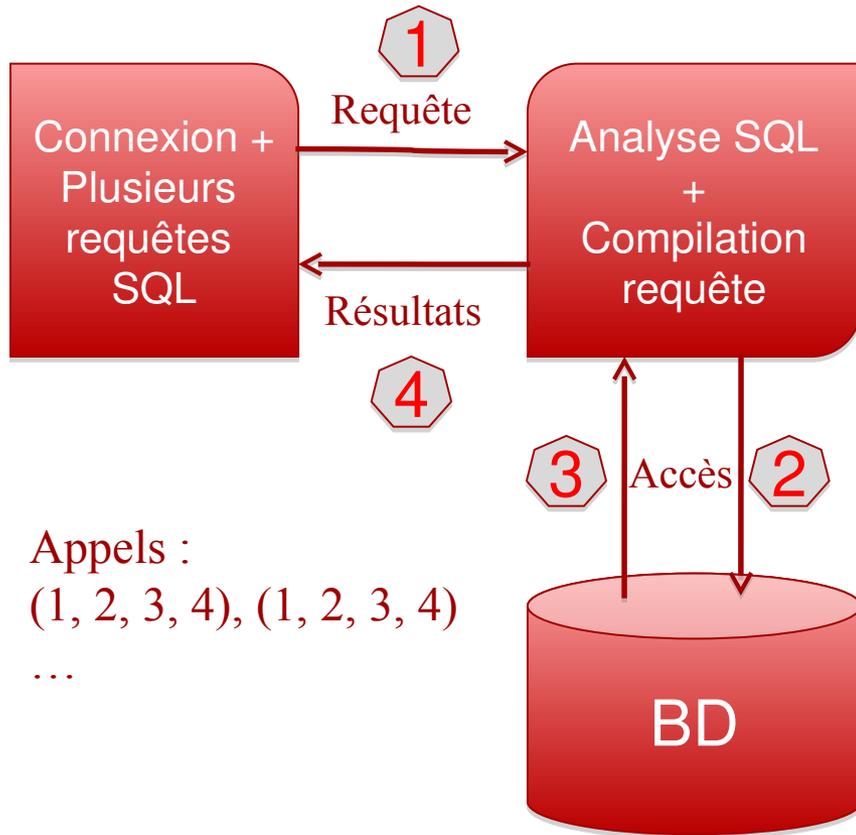
Quel est l'intérêt de PL/SQL ?

- SQL est déclaratif
 - Requêtes naturelles
 - Mais les applications complexes exigent plus,
 - Pour la **facilité et l'efficacité de développement** :
 - gérer le contexte,
 - lier plusieurs requêtes entre elles,
 - créer des librairies de procédures cataloguées réutilisables
 - Pour l'**efficacité de l'application** :
 - factoriser les traitements proches des données
 - réduire les échanges client et serveur
(un programme PL/SQL est exécuté sur le serveur)
- ⇒ Besoin d'étendre SQL :
PL/SQL est une extension procédurale

PL/SQL - Modes

- Interactif :
 - Exécution de code
 - par exemple, contrôler ou corriger des données
- Stocké :
 - Procédures, fonctions ou de triggers
 - Appel interne
- Programme :
 - Appel depuis langages généralistes (JDBC)

Architecture



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
```

Exemple

DECLARE

```
-- Quelques variables  
v_nbFilms    INTEGER;  
v_nbArtistes INTEGER;
```

BEGIN

```
-- Compte le nombre de films  
SELECT COUNT(*) INTO v_nbFilms FROM Film;  
-- Compte le nombre d'artistes  
SELECT COUNT(*) INTO v_nbArtistes FROM Artiste;  
  
-- Affichage des résultats  
DBMS_OUTPUT.PUT_LINE ('Nombre de films: ' || v_nbFilms);  
DBMS_OUTPUT.PUT_LINE ('Nombre d'artistes: ' || v_nbArtistes)
```

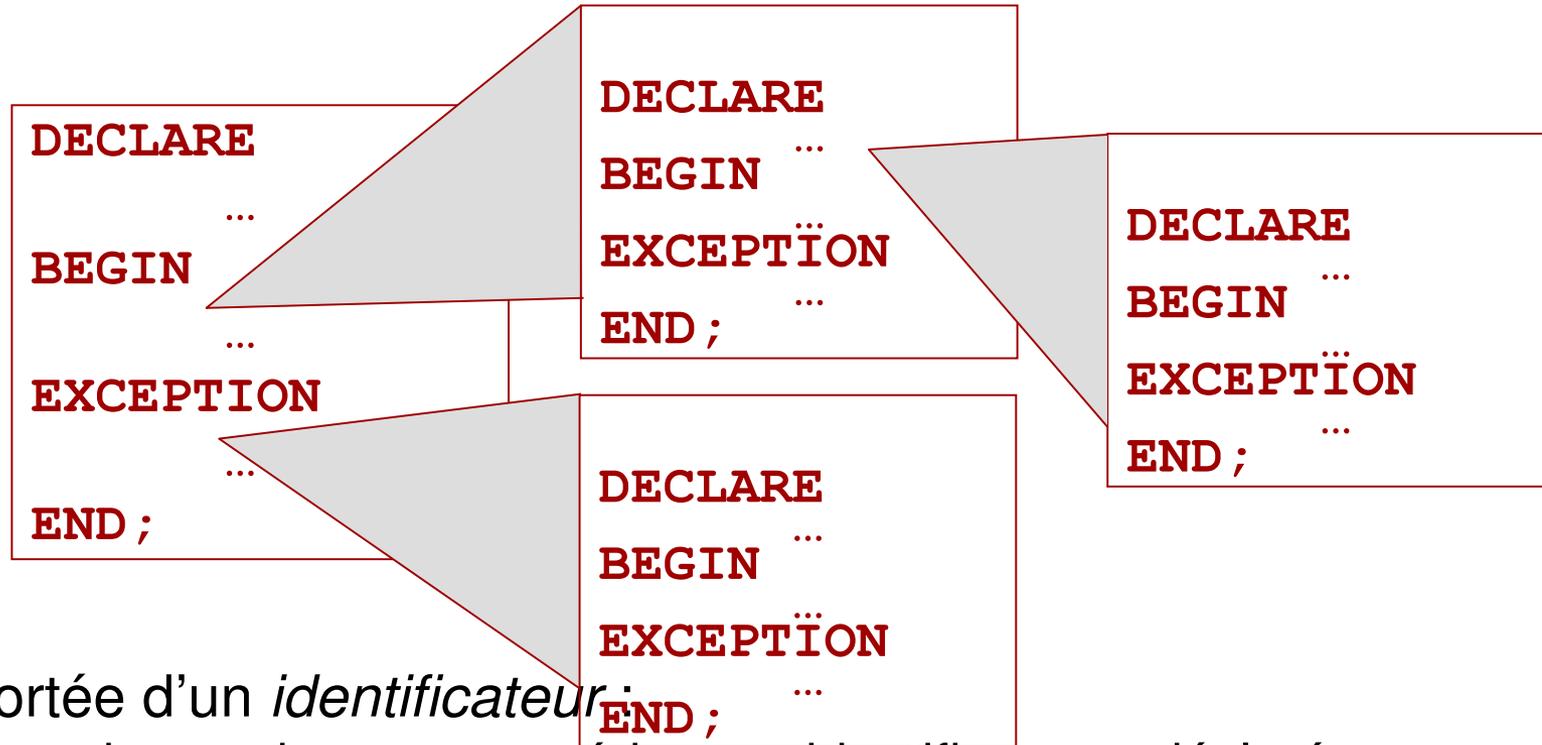
EXCEPTION

```
WHEN OTHERS THEN  
    DBMS_OUTPUT.PUT_LINE ('Problème rencontré dans StatsFilms');
```

END;

Imbrication des blocs PL/SQL

- Blocs imbriqués :



- Portée d'un *identificateur* :
 - un descendant peut accéder aux identificateurs déclarés par un parent, pas l'inverse
- Un bloc est compilé pour être ensuite exécuté

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

- Toute variable PL/SQL :
 - **Obligatoirement** défini dans **DECLARE** avant utilisation
- Types de variables PL/SQL :
 - Types Scalaires (Oracle) :
 - NUMBER (5 , 2) ,
 - VARCHAR2 ,
 - DATE ,
 - BOOLEAN , ...
 - Composites :
 - %TYPE (schéma d'un attribut),
 - %ROWTYPE (schéma d'une table ou résultat de requête),
 - RECORD (type complexe dérivé),
 - TABLE (tables dynamiques)
 - Référence :
 - REF
 - Large Object :
 - LOB (jusqu'à 4 Go ; pointeur si externe)

Variables scalaires

- Syntaxe de déclaration :

```
Identificateur [CONSTANT] type [[NOT NULL] {:= |  
DEFAULT} expression];
```

- **CONSTANT** :
c'est une constante (**doit** être initialisée)
 - **NOT NULL** :
on ne peut pas lui affecter une valeur nulle (sinon exception
`VALUE_ERROR`)
 - Initialisation :
:= (affectation)
DEFAULT
- Pas de déclaration multiple dans PL/SQL !
`number1, number2 NUMBER;` ← **déclaration incorrecte !**

Variables scalaires : Exemples

DECLARE

```
nom varchar2 (10) not null := 'Lucas';  
adresse varchar2 (20);  
x INT := 1;  
pi constant FLOAT := 3.14159;  
rayon FLOAT DEFAULT 1;  
surface DOUBLE := pi * rayon ** 2;
```

Variables et SQL

- Possibilité d'affecter une valeur grâce à une requête SQL

```
SELECT titre INTO mon_film  
FROM FILM  
WHERE id_film = mon_id_film ;
```

Variables composites

- TYPE adresse IS **RECORD**
(no INTEGER,
 rue VARCHAR(40) ,
 ville VARCHAR(40) ,
 codePostal VARCHAR(10)) ;
- titre Film.titre%**TYPE** ;
 - Même type qu'un attribut ou autre variable ;
 - Préserve des modifications de tables ;
- artiste Artiste%**ROWTYPE**
 - Contraintes **NOT NULL** de la table non transmises ;
 - **un seul tuple** affecté à une variable %**ROWTYPE** !
- Possibilité de dériver des types à partir du retour des requêtes (cf. curseurs)

Variables composites : Exercice

- Créer un **programme** :
 - Pour un id de Film (*mon_id_film*) donné
 - Récupère le *titre* du film correspondant
(Requête SQL : `SELECT ... INTO variable FROM ...`)
 - Récupère le *nom* et le *prénom* du metteur en scène (*id_mes*) dans la table *Artiste*
 - Affiche le titre et le nom à l'aide de `DBMS_OUTPUT.PUT_LINE`
- Schéma :
 - Film (*id_film*, *titre*, *id_mes*, *année*, *coût*, *recette*)
 - Artiste (*id*, *nom*, *prenom*, *date_naiss*)

Exercice - réponse

```
DECLARE
  v_titre Film.titre%TYPE;
  v_idMES Film.idMES%TYPE;
  v_mes Artiste%ROWTYPE;

BEGIN
  -- Recherche du film
  SELECT titre, idMES INTO v_titre, v_idMES
     FROM Film
     WHERE idFilm = mon_id_Film;

  -- Recherche du metteur en scène
  SELECT * INTO v_mes
     FROM Artiste WHERE idArtiste = v_idMES;

  DBMS_OUTPUT.PUT_LINE (v_titre || ', réalisé par ' ||
                        v_mes.prenom || ' ' || v_mes.nom);

END;
/
```

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 en opérations arithmétiques
- Sous-types PL/SQL :
 - Restriction d'un type de base
 - `CHARACTER`, `INTEGER`, `NATURAL`, `POSITIVE`, `FLOAT`, `SMALLINT`, `SIGNTYPE`, etc.
 - Restriction : précision ou taille maximale
 - `SUBTYPE nomSousType IS typeBase`
`[(contrainte)] [NOT NULL];`
 - Exemple de sous-type utilisateur :
 - `SUBTYPE numInsee IS NUMBER(13) NOT NULL;`

Conversions implicites

- Lors du calcul d'une expression ou d'une affectation
- Exception si conversion non autorisée

De	A	CHAR	VARCHAR 2	BINARY _ INTEGE R	NUMBE R	LONG	DATE	RAW	ROWID
CHAR			OUI	OUI	OUI	OUI	OUI	OUI	OUI
VARCHAR 2	OUI			OUI	OUI	OUI	OUI	OUI	OUI
BINARY_ INTEGER	OUI	OUI			OUI	OUI			
NUMBER	OUI	OUI	OUI	OUI		OUI			
LONG	OUI	OUI	OUI					OUI	
DATE	OUI	OUI	OUI			OUI			
RAW	OUI	OUI	OUI			OUI			
ROWID	OUI	OUI	OUI						

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				

Variables **TABLE**

- Tableaux dynamiques, composé de :
 - Clé primaire
 - Colonne de type scalaire
 - `%TYPE`, `%ROWTYPE` OU `RECORD`
- Fonctions **PL/SQL** dédiées aux tableaux :
EXISTS (x) , **PRIOR (x)** , **NEXT (x)** ,
DELETE (x, ...) , **COUNT** , **FIRST** , **LAST** ,
DELETE

Variables **TABLE** : exemple

```
DECLARE
  TYPE FilmSF IS TABLE OF Film%ROWTYPE
    INDEX BY BINARY_INTEGER;
  tabFilms FilmSF;
  tmpIndex BINARY_INTEGER;
BEGIN
  ...
  tmpIndex := tabFilms.FIRST;
  tabFilms(4).Titre := 'Star Wars - Ep 4';
  tabFilms(4).MES := 54;
  tabFilms.DELETE(5);
  ...
END;
```

Affectation de Variables : ligne de commande

```
SQL> ACCEPT s_titre PROMPT 'Titre Film : '  
SQL> ACCEPT s_annee PROMPT 'Année de sortie : '  
SQL> ACCEPT s_MES PROMPT 'Metteur en scène : '
```

```
DECLARE
```

```
    id_film NUMBER(6,2) DEFAULT 1;
```

```
BEGIN
```

```
    INSERT INTO Film VALUES
```

```
        (id_film, '&s_titre', &s_annee, &s_MES, 0, 0);
```

```
END;
```

```
/
```

Résolution des noms

- Lors de doublons de noms
 - Variable, Table, Colonne
- Règles de résolution des noms :
 - Variable **du bloc** prioritaire sur variable **externe** au bloc (et visible)
 - *Variable* prioritaire sur *nom d'une table*
 - *Nom d'une colonne d'une table* prioritaire sur *Variable*

Entrées et Sorties

- Paquetage `DBMS_OUTPUT` :

- Sortie d'une valeur :

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

- Sortie d'une valeur suivie de fin de ligne :

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

- Entrée d'une valeur :

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

Entrées et Sorties (2)

- 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 else
 - Case when
- Structures Répétitives
 - While
 - Loop
 - For

Structures conditionnelles

```
▪ IF <condition> THEN  
    <instructions> ;  
ELSIF <condition> THEN  
    <instructions> ;  
ELSE  
    <instructions> ;  
END IF;
```

```
▪ CASE <variable>  
WHEN <value> THEN  
    <instructions> ;  
...  
WHEN <value> THEN  
    <instructions> ;  
ELSE  
    <instructions> ;  
END CASE;
```

IF : exemple

```
DECLARE
  titre Film.titre%TYPE;
BEGIN
  IF episode = 4 THEN
    DBMS_OUTPUT.PUT_LINE ('A new Hope');
  ELSIF episode = 5 THEN
    DBMS_OUTPUT.PUT_LINE ('Empire strikes Back');
  ELSIF episode = 7 THEN
    DBMS_OUTPUT.PUT_LINE ('The Force Awakens');
  END IF;
END;
/
```

CASE

- Seul le cas valide est traité
- Si aucun cas valide : exception `CASE_NOT_FOUND`
- Exemple :

```
DECLARE
  titre Film.titre%TYPE;
BEGIN
  CASE episode
  WHEN 1 THEN DBMS_OUTPUT.PUT_LINE ('The Fantom Menace');
  WHEN 2 THEN DBMS_OUTPUT.PUT_LINE ('The Clone Wars');
  WHEN 3 THEN DBMS_OUTPUT.PUT_LINE ('Revenge of the Siths');
  WHEN 4 THEN DBMS_OUTPUT.PUT_LINE ('A new Hope');
  WHEN 5 THEN DBMS_OUTPUT.PUT_LINE ('Empire strikes Back');
  WHEN 6 THEN DBMS_OUTPUT.PUT_LINE ('Return of the Jedi');
  WHEN 7 THEN DBMS_OUTPUT.PUT_LINE ('The Force Awakens');
  ELSE
    DBMS_OUTPUT.PUT_LINE ('Unkown episode');
  END IF;
END;
/
```

Structures répétitives

- **WHILE** <condition> **LOOP**
 <instructions> ;
END LOOP;

- **LOOP**
 <instructions> ;
EXIT WHEN <condition> ;
 <instructions>
END LOOP;

- **FOR** <variable> **IN** <value> .. <value> **LOOP**
 <instructions> ;
END LOOP;

Tant que (**WHILE**) : exemple

```
DECLARE
  a INTEGER := 1;
  b INTEGER := 1;
BEGIN
  WHILE b <= 10 LOOP
    a := a * b;
    b := b + 1;
  END LOOP;
END;
```

Répéter (**LOOP**) : exemple

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

Pour (**FOR**) : exemple

- Compteur est incrémenté de **1** (ou décrémente si **REVERSE**) ;

```
DECLARE
  a  INTEGER := 1;
  b  INTEGER := 1;
BEGIN
  FOR b IN 1..10 LOOP
    a := a * b;
  END LOOP;
END;
```

Boucles imbriquées

- Chaque structure répétitives peut avoir une étiquette : <<étiquette>>

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

Interaction avec la base

- Interrogation **directe** des données :
 - `SELECT titre INTO varTitre`
`FROM Film WHERE année = 2015;`
 - Doit retourner **1** enregistrement
 - Sinon `TOO_MANY_ROWS` OU `NO_DATA_FOUND`
- Manipulation des données :
 - `INSERT INTO nomTable (liste colonnes) VALUES`
`(liste expressions);`
 - `UPDATE nomTable SET nomColonne = expression`
`[WHERE condition];`
 - `DELETE FROM nomTable [WHERE condition];`

Curseurs

- Problème :
 - Accès direct : ne prend qu'un seul tuple
 - Comment récupérer plusieurs tuples ?
 - curseur parcourt un par un chaque tuple résultat ('pointeur' sur résultats).
 - Chaque tuple récupéré pourra être mis dans une variable

Curseurs : instructions

```
DECLARE
```

```
-- Définition du curseur sur tous les films
```

```
CURSOR lesFilms IS SELECT * FROM Film;
```

```
-- Variable d'affectation des tuples
```

```
leFilm Film%ROWTYPE;
```

```
BEGIN
```

```
-- Ouverture du curseur et exécution de la requête
```

```
OPEN lesFilms;
```

```
-- Chargement d'un tuple, et positionnement sur le suivant
```

```
FETCH lesFilms INTO leFilm;
```

```
-- Fermeture du curseur et libération mémoire
```

```
CLOSE lesFilms;
```

Curseurs explicites : attributs

- **nomCurseur%ISOPEN**
 - **TRUE** si le curseur est ouvert
- **nomCurseur%FOUND**
 - **TRUE** si le dernier **FETCH** contient un tuple
- **nomCurseur%NOTFOUND**
 - **TRUE** si le dernier **FETCH** ne contient pas de tuples
- **nomCurseur%ROWCOUNT**
 - nombre total de lignes traitées **jusqu'à présent** (par ex. nombre de **FETCH**)

Curseur : Exercice

- Programme avec curseur
 - Récupérer tous les films dont 'Georges Lucas' est le metteur en scène (MES)
(curseur + open + boucle + fetch + close)
 - Afficher pour chaque film le titre et l'année
 - *Bonus*: Afficher le gain total de tous les films
 - Somme(recette) - somme(cout)
- Schéma :
 - Film (id_film, titre, id_mes, année, coût, recette)
 - Artiste (id, nom, prenom, date_naiss)

Curseurs : exemple

```
DECLARE
  CURSOR films IS
    SELECT * FROM Film
    WHERE MES = (SELECT id
                 FROM Artiste
                 WHERE Nom='Lucas' and Prenom='Georges');

  leFilm      Film%ROWTYPE;
  total       NUMBER(11,2) := 0;

BEGIN
  OPEN films;
  LOOP
    FETCH films INTO leFilm;
    EXIT WHEN films%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(leFilm.titre || ' : ' || leFilm.année);
    total := total - leFilm.cout + leFilm.recette;
  END LOOP;
  CLOSE leFilm;
  DBMS_OUTPUT.PUT_LINE('Gain total : ' || total);

END;
/
```

Curseur paramétré

- Objectif :
 - paramétrer la requête associée à un curseur
 - Provient d'une variable temporaire, paramètre du programme

- Syntaxe :

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

Paramètres :

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

(nomPar est inconnu en dehors de la définition !)

- Utilisation :

```
OPEN nomCurseur (valeurPar1 [, valeurPar2, ...]);
```

- Fermeture (**close**) avant d'utiliser avec d'autres paramètres

Curseur Paramétré: Exercice

- Programme avec curseur paramétré
 - Récupérer tous les films d'un MES en paramètre (leMES)
 - Pour chaque film, affichage :
 - Si le gain (*recette - cout*) est supérieur à une valeur donnée (*seuil*)
 - Du titre et de l'année
 - Afficher le nombre de films précédents / total
- Schéma :
 - Film (id_film, titre, id_mes, année, coût, recette)
 - Artiste (id, nom, prenom, date_naiss)

Curseurs paramétrés : exemple

```
DECLARE
  CURSOR films(idMES Film.MES%TYPE) IS
    SELECT * FROM Film
    WHERE MES = idMES ;

  leFilm          Film%ROWTYPE;
  nbSup           NUMBER(11,2) := 0;

BEGIN
  OPEN films(leMES);
  LOOP
    FETCH films INTO leFilm;
    EXIT WHEN films%NOTFOUND;
    IF (leFilm.recette - leFilm.cout) > seuil THEN
      DBMS_OUTPUT.PUT_LINE
        (leFilm.titre || ' : ' || leFilm.année);
      nbSup := nbSup + 1;
    END IF;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Résultat: ' || nbSup || '/' || films%ROWCOUNT);
  CLOSE leFilm;
END;
```

Boucle FOR avec curseur

- Exécute les instructions pour chaque enregistrement

```
DECLARE
  CURSOR films (idMES Film.MES%TYPE) IS
    SELECT * FROM Film
      WHERE MES = idMES ;
  leFilm      Film%ROWTYPE;
  nbSup       NUMBER(11,2) := 0;
  total       NUMBER(11,2) := 0;
BEGIN

  -- l'ouverture du curseur se fait dans le FOR
  FOR leFilm IN films(leMES) LOOP
    IF (leFilm.recette - leFilm.cout) > seuil THEN
      DBMS_OUTPUT.PUT_LINE(leFilm.titre||' : '||leFilm.année);
      nbSup := nbsup + 1;
    END IF;
    total = films%ROWCOUNT;
  END LOOP;
  -- Fermeture du curseur (plus de %ROWCOUNT)

  DBMS_OUTPUT.PUT_LINE('Résultat: '|| nbSup || '/' || total);
END;
```

Programmes et Sous-Programmes

- Nommage et paramétrage de Blocs
 - **Procédure**
 - Eventuellement retourne DES résultats
 - **Fonction**
 - Résultat unique obligatoire
 - Appel possible dans une requête SQL
- Programmes stockés dans la base
 - Modularité (conception et maintenance),
 - Réutilisation
 - Intégrité (regroupement de traitements dépendants)
 - Sécurité (gestion des droits/contraintes sur données)
- Récursivité autorisée (à utiliser avec précaution) !
- Sous-Programmes
 - Défini dans le `DECLARE`

Appel de programme

- Appel de procédure/fonction depuis un **bloc PL/SQL** :
`nomProcEDURE (listeParEffectifs) ;`
- Appel de procédure stockée sous **SQL*Plus** :
`SQL> EXECUTE nomProcEDURE (listeParEffectifs) ;`
- Appel de fonction stockée sous **SQL*Plus** :
`SQL> nomFonction (listeParEffectifs) ;`

Procédures

- Syntaxe :

```
CREATE [OR REPLACE] 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];
```

Paramètres

- Types de paramètres :
 - Entrée (**IN**)
 - Valeur constante (pas d'affectation)
 - Toujours passé par **référence** !
 - Sortie (**OUT**)
 - Valeur de retour
 - Ne peut être qu'affecté (pas utilisé)
 - Par défaut (sans **NOCOPY**) passé par **valeur** !
 - Entrée et sortie (**IN OUT**)
 - Passé en référence
 - Valeur de retour
 - Peut être utilisé et affecté
 - Par défaut (sans **NOCOPY**) passé par **valeur** !
- **NOCOPY**
 - Données retours par référence (paramètres volumineux)

Procédure locale : exemple

```
DECLARE
```

```
-- Procédure locale
PROCEDURE lesFilms (prenomParam VARCHAR2(30), nomParam VARCHAR2(30))
IS
  CURSOR films(lePrenom VARCHAR2(20), leNom VARCHAR2(20)) IS
  SELECT * FROM Film
  WHERE MES = (SELECT id
  FROM Artiste
  WHERE Nom=leNom and Prenom=lePrenom);
  leFilm          Film%ROWTYPE;
BEGIN
  FOR leFilm IN films(prenomParam,nomParam) LOOP
    DBMS_OUTPUT.PUT_LINE('  ||leFilm.titre||' : '||leFilm.année);
  END LOOP;
END lesFilms;
  CURSOR mes IS
  SELECT * FROM Artiste where id in (select MES from Films);
  leMes Film%ROWTYPE;
```

```
BEGIN
```

```
  FOR leMes IN mes() LOOP
    DBMS_OUTPUT.PUT_LINE(leMes.prenom||' '||leMes.nom||' : ');
    -- Appel de la procédure
    lesFilms (leMes.prenom, lesMes.nom) ;
  END LOOP;
```

```
END;
```

Fonctions

- Syntaxe :

```
CREATE [OR REPLACE] FUNCTION nomFonction  
[(par1 [IN | OUT | IN OUT] [NOCOPY] type1  
[{::= | DEFAULT} expression]  
[, par2 [IN | OUT | IN OUT] [NOCOPY] type2  
[{::= | DEFAULT} expression ... )]  
-- OUT a éviter (effets de bords)
```

```
RETURN typeRetour  
{IS | AS} [declarations;]  
BEGIN  
<instructions>;  
...  
RETURN varRetour;  
-- return obligatoire
```

```
[EXCEPTION  
<traitementExceptions>;  
]  
END [nomFonction];
```

Fonctions : Exemple

```
DECLARE
FUNCTION nbFilms (idMES Film.MES%TYPE)
RETURN INTEGER
IS
    nbFilm INTEGER;
BEGIN
    SELECT COUNT(*) INTO nbFilm FROM Film
        WHERE MES = idMES;
    RETURN nbFilm;
END nbFilms;
CURSOR mes IS
    SELECT * FROM Artiste where id in (select MES from Film);
leMes Film%ROWTYPE;
nbFilm INTEGER;
BEGIN
    FOR leMes IN mes() LOOP
        nbFilm := nbFilms(leMes.id);
        DBMS_OUTPUT.PUT_LINE(leMes.prenom||' '
            ||leMes.nom||' : '||nbFilm);
    END LOOP;
END;
```

Fonction stockée : exemple

- Appel depuis SQL*Plus :

```
SQL> SELECT prenom, nom, nbFilms(id)
      FROM Artiste
      WHERE id in (select MES from Film);
```

Manipulation de programme

- Création ou modification de sous-programme :
`CREATE [OR REPLACE] {PROCEDURE | FUNCTION} nom ...`
- Recompilation automatique lors d'une modification
 - 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`

Paquetages

- Paquetage
 - Regroupement variables, curseurs, fonctions, procédures, etc.
 - Ensemble cohérent de services
- Encapsulation
 - Accès extérieurs
 - Accès privés (internes au paquetage)
- Structure
 - Section de **spécification**
 - Déclaration des variables et curseurs,
 - Déclaration 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 gestionMES AS  
  ...  
  FUNCTION nbFilms (idMES Film.MES%TYPE)  
    RETURN INTEGER;  
  
  FUNCTION leMES (titreFilm Film.MES%TYPE)  
    RETURN INTEGER;  
  
  PROCEDURE filmParMES ();  
  ...  
END gestionMES;
```

Section d'implémentation

- **Syntaxe :**

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

Implémentation : exemple (1/2)

```
CREATE PACKAGE BODY gestionMES AS
```

```
-- Fonction publique : Nombre de films pour un Metteur en scène
```

```
FUNCTION nbFilms(idMES Film.MES%TYPE) RETURN INTEGER IS  
nbFilmINTEGER;  
BEGIN  
SELECT COUNT(*) INTO nbFilm FROM Film WHERE MES = idMES;  
RETURN nbFilm;  
END nbFilms;
```

```
-- Fonction publique : Retourne l'id du metteur en scène à partir d'une titre de film
```

```
FUNCTION leMES(titreFilm Film.MES%TYPE) RETURN INTEGER IS  
idMES Film.MES%TYPE;  
BEGIN  
SELECT MES INTO idMES FROM Film WHERE title like titreFilm;  
RETURN idMES;  
END leMES;
```

```
/* Fonction publique : Affiche pour chaque metteur en scène provenant un curseur privé :
```

```
- son prenom et son nom
```

```
- appel une procédure d'affichage de tous ses films */
```

```
PROCEDURE filmParMES() IS  
leMES Artiste%ROWTYPE;  
BEGIN  
FOR leMES IN lesMES() LOOP  
DBMS_OUTPUT.PUT_LINE(leMES.prenom||' '||leMES.nom);  
lesFilms (leMES.id);  
END LOOP;  
END filmParMES;
```

Implémentation : exemple (2/2)

```
-- Curseur privé : liste tous les artistes metteurs en scène
CURSOR lesMES ()
    IS SELECT * FROM Artiste where id in (select MES from Films);

-- Procédure privée : affiche tous les films d'un metteur en scène
PROCEDURE lesFilms (leMES Film.MES%TYPE) IS
    CURSOR films (idMES Film.MES%TYPE) IS
        SELECT * FROM Film where MES = idMES
        ORDER BY année;
    leFilm FILM%ROWTYPE;
    BEGIN
        FOR leFilm IN films(leMES) LOOP
            DBMS_OUTPUT.PUT_LINE(' ||leFilm.titre||' ' ||leFilm.année);
        END LOOP;
    END lesFilms;

END gestionMES;
```

Référence au paquetage

- Uniquement sur les objets et programmes **publics**
- Syntaxe :
 - `nomPaquetage.nomObjet`
 - `nomPaquetage.nomSousProgramme (...)`

Manipulation d'un paquetage

- Re-compilation d'un paquetage :
 - **CREATE OR REPLACE PACKAGE**
 - Modification d'une sections
 - re-compilation automatique de l'autre section
 - Erreurs de compilation avec SQL*Plus :
SHOW ERRORS
- Suppression d'un paquetage :
DROP BODY nomPaquetage ;
DROP nomPaquetage ;

Exceptions

- Conditions d'erreur lors de l'exécution
- **EXCEPTION**
 - Clause de récupération d'erreur
 - Evite l'arrêt systématique du programme
- Possibilité de définir des erreurs

Traitement des exceptions

- Syntaxe :

```
BEGIN
```

```
...
```

```
EXCEPTION
```

```
    WHEN nomException1 [OR nomException2 ...] THEN  
        instructions1;
```

```
    WHEN nomException3 [OR nomException4 ...] THEN  
        instructions3;
```

```
    WHEN OTHERS THEN  
        instructionsAttrapeTout;
```

```
END;
```

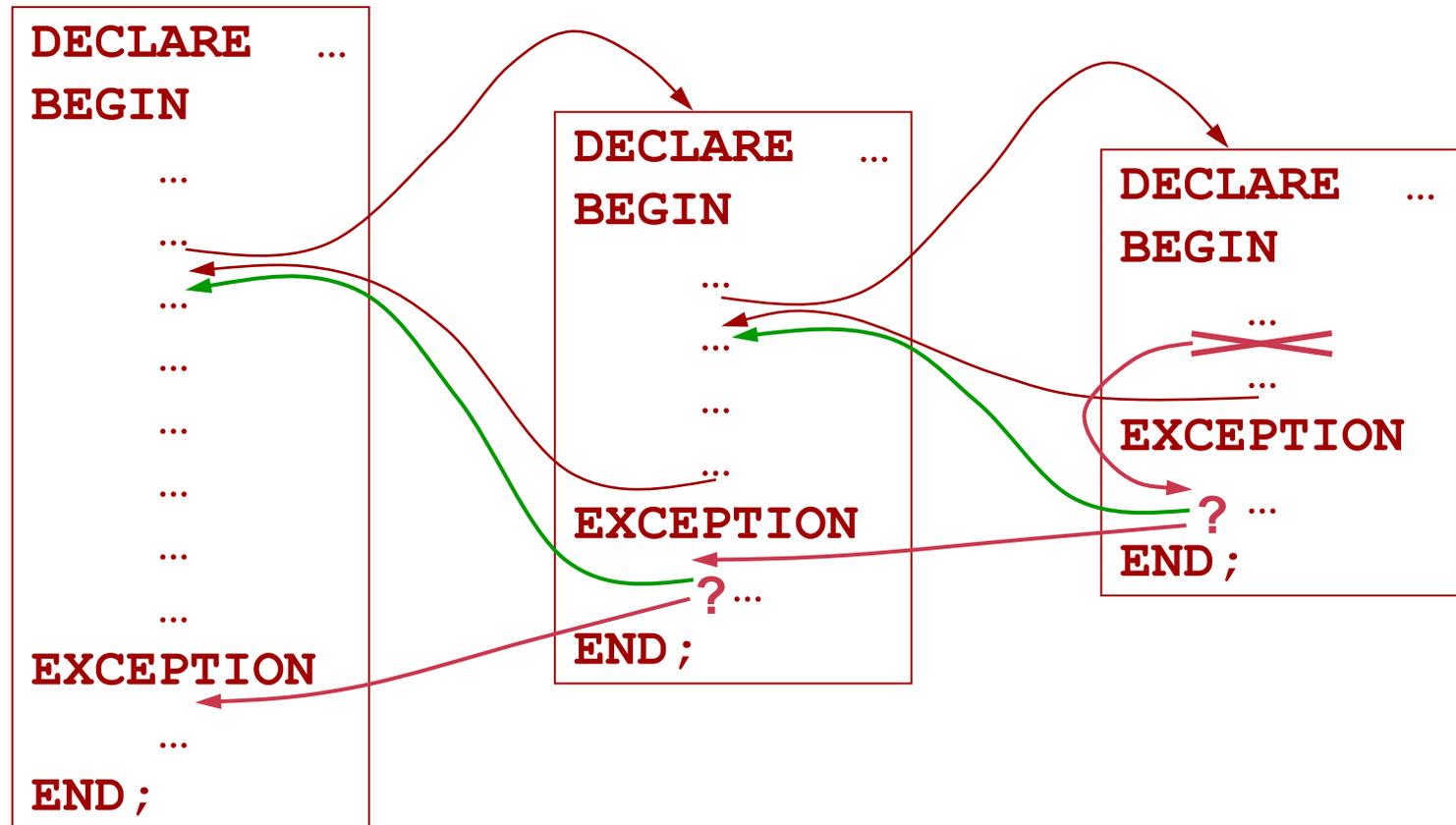
- Affichage de l'erreur

```
DBMS_OUTPUT.PUT_LINE (SQLERRM || ' : ' || SQLCODE) ;
```

Lors d'une exception

1. Aucun traitement n'est prévu
 - le programme s'arrête
2. Un traitement est prévu :
 - Arrêt de l'exécution du bloc PL/SQL courant
 - L'exception est recherchée dans la section **EXCEPTION**
 - Associée au bloc courant,
 - Sinon dans les blocs parents
 - Sinon le programme appelant
 - Exception traitée suivant les instructions trouvées
 - Spécifiques
 - Attrape-tout
 - Exécution du traitement prévu par l'exception (**THEN...**)
 - Suite de l'exécution dans le bloc/programme parent

Suivi des exceptions



Mécanismes de déclenchement

1. Déclenchement automatique

- Erreurs **prédéfinies** Oracle
- `VALUE_ERROR`, `ZERO_DIVIDE`, `TOO_MANY_ROWS`, etc.

2. Déclenchement programmé

- Dans `DECLARE` :

```
nomException EXCEPTION;
```

- Dans `BEGIN` :

```
RAISE nomException;
```

- 2. Dans `EXCEPTION` :

```
WHEN nomException THEN
```

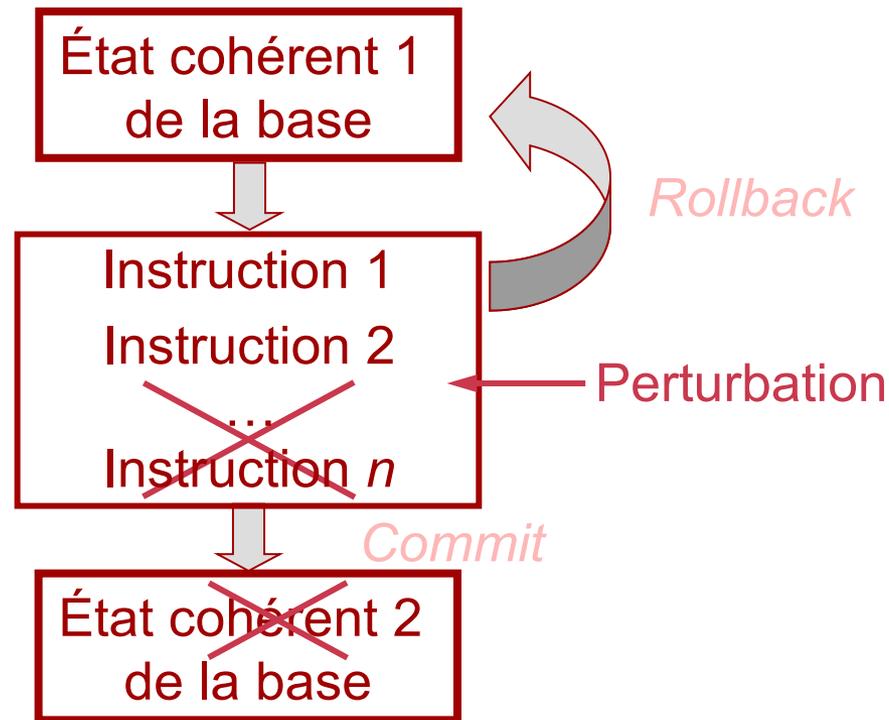
Mécanismes de déclenchement (2)

- Déclenchement
 - avec message et code d'erreur personnalisé :

```
RAISE_APPLICATION_ERROR(  
  numErr, messageErr, [TRUE | FALSE]);
```
 - **TRUE** : mise dans une pile d'erreurs à propager (par défaut) ;
FALSE : remplace les erreurs précédentes dans la pile

Transactions

- Objectif :
 - Cohérence d'une suite de maj sur des données



Transactions : contrôle

- Début
 - 1° **SQL** après le **BEGIN**
 - 1° **SQL** après une transaction
- Fin
 - Avec succès : **COMMIT [WORK] ;**
 - Échec : **ROLLBACK [WORK] ;**
 - Fin implicite
 - Avec succès : fin normale d'une session
 - Échec : fin anormale d'une session

Transactions : contrôle (2)

- Annulations partielles

SAVEPOINT nomPoint; -- insertion point de validation

ROLLBACK TO nomPoint; -- retour à l'état au nomPoint

- Remarques :

- Sortie suite à une exception non traitée

- Pas de **ROLLBACK** implicite

- Opérations réalisées dans le sous-programme non annulées

Transactions/Exceptions : exemple

```
CREATE PROCEDURE ajoutFilm () IS
  nvFilm Film%ROWTYPE;
  nbFilm INTEGER;
  mesErreur EXCEPTION;
  filmErreur EXCEPTION;
BEGIN
  SELECT MAX(id)+1 INTO film.id FROM Film;
  nvFilm.titre := DBMS_OUTPUT.GET_LINE('Titre=',1);
  nvFilm.MES := trouver_idMES();

  IF nvFilm.MES IS NULL OR nvFilm.MES = 0 THEN RAISE mesErreur; END IF;
  nvFilm.annee := DBMS_OUTPUT.GET_LINE('Année=',1);
  nvFilm.cout := DBMS_OUTPUT.GET_LINE('Cout=',1);
  nvFilm.recette := DBMS_OUTPUT.GET_LINE('Recette=',1);
  INSERT INTO Film values
  (nvFilm.id, nvFilm.titre, nvFilm.MES, nvFilm.cout, nvFilm.recette);
  SELECT COUNT(*) INTO nbFilm
  FROM Film where titre = nvFilm.titre AND annee = nvFilm.annee;

  IF nbFilm > 1 THEN RAISE filmErreur; END IF;
  COMMIT;
EXCEPTION
  WHEN mesErreur THEN
    DBMS_OUTPUT.PUT_LINE('Metteur en scène inconnu');
  WHEN filmErreur THEN
    DBMS_OUTPUT.PUT_LINE('Film déjà existant');
  ROLLBACK;
  WHEN OTHERS THEN
    ROLLBACK;
END;
```

PL/SQL et MySQL

- Quelques différences :
 - Déclaration d'une variable : DECLARE var <type>;
 - Affectation de variable : SET variable = <expr>;
 - Curseurs :
 - DECLARE cur CURSOR FOR <SQL>;
 - FETCH cur INTO a, b, c; (plusieurs variables)
 - Pas de boucles FOR
 - Afficher des données : SELECT
 - SELECT concat('texte : ', var, '. Texte') ;
 - Exceptions
 - Déclarer un "maître" (Handler) pour l'erreur
 - Valable pour la procédure à partir de la déclaration
 - Différents types d'actions

Mysql et Erreurs

```
DECLARE <type d'action> HANDLER  
  FOR <condition d'erreur> [, <condition>] ...  
  <statement>
```

- Type d'action :
 - CONTINUE / EXIT / UNDO
- Condition d'erreur :
 - Erreur MySQL : SQLSTATE [VALUE] <sqlstate value>
 - <http://dev.mysql.com/doc/refman/5.0/fr/error-handling.html>
 - Ex : 23000 : valeur unique, 42000 : incorrect/inconnu/interdit
 - SQLWARNING / NOT FOUND / SQLEXCEPTION