PL/SQL
et Triggers

Procedural Language/
Structured Query Language

Plan

- Introduction
- Structure d'un programme
- Les variables
- Les instructions
- Les curseurs
- Les exceptions
- Triggers

PL/SQL

- Langage de programmation de quatrième génération (L4G)
- Langage procédural propriétaire Oracle, combine des requêtes SQL avec l'utilisation des variables et des instructions procédurales (alternatives, boucles)
- Syntaxe inspirée du langage Ada
- Proche de la norme SQL/PSM (SQL/Persistent Stored Modules) comme PostgreSQL, MySql stored procedures et Transact-SQL

Utilisation de PL/SQL (1)

- ***SQL*** = langage
  - Sémantique (multi-)ensembliste
  - Type déclaratif (décrit le résultat souhaité)

- ***« PL/SQL »***
  - Langage procédural: variables, tests, boucles, curseurs, fonctions, procédures, exceptions
  - Association forte avec SQL
  - Regrouper sur le serveur les requêtes SQL et le traitement des données (compilation à la création, exécution rapide à l'appel)

Utilisation de PL/SQL (2)

- On stocke du code propre aux données dans la base :
  - *Meilleures performances :* permet de regrouper les instructions SQL et les envoyer au serveur dans un bloc => réduit les échanges entre client/serveur
  - *Sécurité :* le code traitant les données est stocké dans le serveur de la base donc limite les échanges, et accessible uniquement par le DBA (ou personne autorisée)

- Effectuer des traitements complexes sur les données, notamment en reliant plusieurs requêtes entre elles

- Portabilité des programmes, mécanismes de traitement des erreurs et des exceptions

- Possibilité d'utiliser ce code par plusieurs utilisateurs ayant les droits requis

Plan

- Introduction
- Structure d'un programme
- Les variables
- Les instructions
- Les curseurs
- Les exceptions
- Triggers

PL/SQL

- *Blocs anonymes* (anonymous blocks)
- *Procédures et fonctions stockées* (Stored procedures/functions)
- Procédures et fonctions utilisées dans des applications(Application procedures/functions)
- *Déclencheurs associés à la BD* (Database triggers)
- Déclencheurs utilisés dans des applications (Application triggers)
- Paquetages (Packages)

Structure d'un bloc PL/SQL

- Bloc PL/SQL : unité de base de tout programme PL/SQL
- Sections d'un **bloc anonyme**:

DECLARE …	Déclarations : variables, curseurs, types, sous-blocks (optionnelle)
BEGIN …	instructions PL/SQL (obligatoire)
EXCEPTION …	Traitement de avertissements et des erreurs (optionnelle)
END ; /	← lance l'exécution sous SQL*Plus

Procédures et fonctions stockées

Déclaration d'une procédure

```
CREATE [OR REPLACE] PROCEDURE nomProcedure [(liste_paramètres)]
AS|IS
[déclarations de variables] ← Pas de mot clé DECLARE
BEGIN
…
[EXCEPTION]
END [nomProcedure];
/
```

Déclaration d'une fonction

```
CREATE [OR REPLACE] FONCTION nomFonction[(liste_paramètres)]
RETURN typeDonnée AS|IS
[déclarations de variables] ← Pas de mot clé DECLARE
BEGIN
…
RETURN valeurRetour ;
…
[EXCEPTION]
END [nomFonction];
/
```

Procédures et des fonctions stockées

Appel :

- Directement dans du code PL/SQL :

  nomProcedure/nomFonction [(liste de paramètres)]

- Depuis la console SQL*PLUS :

  exec nomProcedure[(paramètres)]

- Dans une requête SQL :

  SELECT nomFonction[(paramètres)], listeAttr FROM maTable …

Suppression :

DROP PROCEDURE nomProcedure ;

DROP FUNCTION nomFunction ;

Plan

- Introduction
- Structure d'un programme
- Les variables
- Les instructions
- Les curseurs
- Les exceptions
- Triggers

Variables

- Doivent être déclarées avant BEGIN (dans la section **DECLARE** pour un bloc anonyme)
- **Nom de variable** : commence obligatoirement par une lettre, ne doit pas dépasser 30 caractères, ne peut pas être un mot réservé
- **Types de variables** PL/SQL ou SQL:
  - *Scalaires* : VARCHAR2, DATE, NUMBER , BOOLEAN , BINARY_INTEGER (PLS_INTEGER), ROWID (adresse d'une ligne), etc.
  - *Composites* (on peut accéder aux composantes individuellement): RECORD, TABLE OF
  - *Référence* : REF (pointers)
  - LOB (Large Object) : pointers vers des objets de grande taille, stockés séparément : texte, images, vidéos, son.
- Variables non PL/SQL :
  - Variables globales ou de substitution définies dans SQL*Plus ou définies dans d'autres programmes

Déclaration des variables scalaires

Syntaxe :

identificateur [CONSTANT] type [NOT NULL] [:= | DEFAULT valeur_initiale] ;

- CONSTANT : constante, sa valeur ne peut pas être modifiée, doit être initialisée.
- DEFAULT : permet d'initialiser des variables sans utiliser l'opérateur :=
- NOT NULL : la variable doit être initialisée

Pas de déclaration multiple : v1, v2 NUMBER ; incorrect

Exemple de déclaration

```
DECLARE
   nom varchar2(10) not null := 'John';
   address varchar2(20);
   x NUMBER := 1;
   pi constant FLOAT := 3.14159;
   rayon FLOAT := 1;
   surface DOUBLE := pi * rayon ** 2;
   …
```

Variable composite : RECORD

- Permet de déclarer une variable composite personnalisée
- *Syntaxe :*

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

- *Déclaration :* `nomVariableRecord nomTypeRecord;`

- *Utilisation dans le code :*

```
nomVariableRecord.nomChamp1 : = ...
```

Exemple de RECORD

```
DECLARE

TYPE adresse IS RECORD
  (no          INTEGER,
   rue         VARCHAR(40),
   ville       VARCHAR(40) NOT NULL := 'Paris',
   codePostal VARCHAR(10));

MonAdresse adresse;

BEGIN

   MonAdresse.no :=5;

   MonAdresse.rue :='Place Jussieu';

   Etc...
```

Variable composite : TABLE

- Permet de définir des tableaux dynamiques (dimension initiale non précisée) composés d'une clé primaire et d'une valeur qui peut être un scalaire ou une variable de type composite)
- Modélisée comme une table, mais ne peut pas être manipulée avec des commandes SQL
- Chaque élément a un index unique (les index ne sont pas nécessairement séquentiels)

- *Définition du type :*

```
TYPE nomTypeTableau IS TABLE OF
{typeScalaire | typeRecord | ...} [NOT NULL]
[INDEX BY BINARY_INTEGER| PLS_INTEGER| VARCAHR2(n)];
```

- *Déclaration de variable:* `nomVariableTableau nomTypeTableau;`
- Si `INDEX BY ...` manque : table imbriquée, les indices sont des nombres séquentiels
- Si `INDEX BY ...` présent : tableau associatif, indices arbitraires, similaire table de hachage

Fonctions pour tableaux PL/SQL

- `EXISTS(x)` : retourne vrai si l'élément de clé x existe
- `PRIOR(x) et NEXT(x)` : renvoie *la clé* de l'élément du tableau qui précède/suit celui de clé x, NULL si pas de prédécesseur/successeur
- `DELETE(x,...)` : supprime l'élément de clé x
- `COUNT` : retourne le nombre d'éléments du tableau
- `FIRST et LAST` : renvoie la première/dernière *valeur de clé*
- `DELETE` : vide le tableau

Exemple de TABLE(1)

```
DECLARE
 --tableau associatif
 TYPE jours IS TABLE OF VARCHAR2(10) INDEX BY
BINARY_INTEGER;
 --déclaration de variable
UneSemaine jours;

 --table imbriquée (nested)
TYPE mois IS TABLE OF VARCHAR2(10); ← table imbriquée
DesMois mois := mois('Janvier', 'Février');

 intIndex BINARY_INTEGER;

BEGIN
   UneSemaine(3):='Lundi';
   intIndex := UneSemaine.FIRST; /* 3 */
   intIndex := DesMois.LAST; /* 2 */

   IF UneSemaine.EXISTS(2)=FALSE
   THEN UneSemaine(2):='Mardi'; END IF;
   …
```

Exemple de TABLE(2)

```
DECLARE

  - tableau associatif
  TYPE joursSem IS TABLE OF NUMBER INDEX BY VARCHAR2(10);

  ListeJours joursSem;

  CharIndex VARCHAR2(10);

BEGIN

    ListeJours('Mardi') := 2;
    ListeJours('Lundi') := 1;

    CharIndex := ListeJours.LAST; /* 'Lundi' */
    CharIndex := ListeJours.FIRST; /* 'Mardi' */

    …
```

VARRAY

- Tableaux de dimension fixe
- *Syntaxe :*

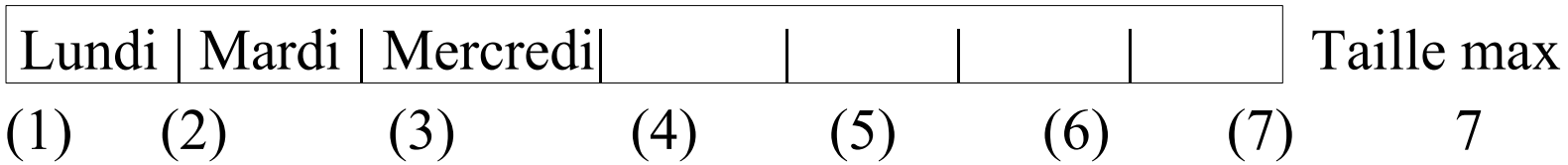
```
TYPE nomTypeVarray IS Varray(N) OF
{typeScalaire | typeRecord | ...} [NOT NULL]
```

- N est la taille maximale, doit toujours être spécifiée

  Exemple :

- *Déclaration du type:* `TYPE Jours IS VARRAY(7) OF VARCHAR2(10) ;`

- *Déclaration/Initialisation d'une variable:* `lesJours Jours:= Jours('Lundi', 'Mardi', 'Mercredi');`



Types dérivés

%TYPE

- Permet de donner à une variable le même type qu'un attribut d'une table ou qu'une autre variable
- Syntaxe:
  - `maVariable nomTable.nomAttribut%TYPE`
  - `maVariable nomAutreVariable%TYPE`

%ROWTYPE

- Permet de donner à une variable composite le même type qu'un enregistrement d'une table ou du résultat d'une requête
- Syntaxe:
  - `maVariable nomTable%ROWTYPE;`
  - `maVariable nomCurseur%ROWTYPE;`

Exemples de types dérivés

EMPLOYE(nom,prénom,âge,adresse)

```
DECLARE
   nomFamille EMPLOYE.nom%TYPE ;
   unEmploye EMPLOYE%ROWTYPE ;
BEGIN
   nomFamille := 'Martin' ;
   unEmploye.prénom :='Lucie' ;
   unEmploye.âge :=24 ;
…
```

Avantage : *si on change le type d'un attribut dans la table employé, pas besoin de changer dans tout le code.*

Plan

- Introduction
- Structure d'un programme
- Les variables
- Les instructions
- Les curseurs
- Les exceptions
- Triggers

Le bloc instructions

- Séquence d'ordres SQL et d'instructions PL/SQL, après **BEGIN**:
  - Select [BULK COLLECT] **INTO** ….FROM...
  - Commandes DML: *Insert, update, delete (*Syntaxe identique à SQL).
  - Transactions : *COMMIT, ROLLBACK*, etc.
  - Fonctions : *to_char, to_date, round,* etc.
  - Commentaires :

```
-- ceci est un commentaire, sur une ligne

/*  et cela
    aussi, sur plusieurs */
```

  - Affichage: DBMS_OUTPUT.PUT_LINE('j"affiche un âge' || maVarAge)
    (pour voir les messages exécuter: SQL> SET SERVEROUTPUT ON)
  - Pas de commandes DDL.

L'instruction Select en PL/SQL

- *INTO*: Permet d'intialiser des variables avec des valeurs d'une seule ligne calculée par une requête
  - Select a1,...,an **INTO** v1,..., vn FROM...
    - a1,...,an sont des attributs, v1,...vn des variables
  - La requête SELECT doit retourner **une seule ligne**
    - Si aucune ligne n'est retournée → erreur NO_DATA_FOUND (ORA-01403)
    - Si plusieurs lignes sont retournées → erreur TOO_MANY_ROWS (ORA-01422)
  - *Exemple:*

```
DECLARE
    sonAge EMPLOYE.age%TYPE;
BEGIN
    select age INTO sonAge FROM EMPLOYE where eno=666;
END;
/
```

L'instruction Select en PL/SQL

- *BULK COLLECT INTO*: Permet de stocker toutes les lignes du résultat
  - *Exemple:*

```
DECLARE
    TYPE l_age IS TABLE OF  age EMPLOYE.age%TYPE INDEX BY
    PLS_INTEGER;
    liste_age l_age;
BEGIN
    select age BULK COLLECT INTO liste_age FROM EMPLOYE;
    FOR indx IN 1 .. liste_age.COUNT
    LOOP
        …
    END LOOP;
END;
    /
```

Les instructions

Affectation

- Simple : nomVariable := valeur
- Par un SELECT ... INTO ….

Comparaison

- NomVariable = valeur

Structures de contrôle conditionnelles

- IF THEN ELSE
- CASE WHEN THEN

Boucles itératives

- FOR IN LOOP
- WHILE LOOP
- LOOP EXIT WHEN

IF THEN ELSE

Syntaxe :

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

Exemple :

```
IF (âge < 18 or âge > 65) THEN
    DBMS_OUTPUT.PUT_LINE('employé hors tranche d''âge') ;
    Delete from EMPLOYE where eno=10 ;
    Update TABLE_SYNTHESE set nb_employés=nb_employés – 1 ;
END IF ;
```

CASE WHEN THEN

Syntaxe :

```
CASE variable
WHEN expression1 THEN instructions1;
WHEN expression2 THEN instructions2; ...
[ELSE instructions;]
END CASE ;
```

- Le premier cas valide est traité, les autres sont ignorés.
- Si aucun est valide, exécute l'instruction du ELSE, et si pas de ELSE, lève une exception `CASE_NOT_FOUND`

Exemple :

```
CASE day_of_week
   WHEN 'Saturday' THEN DBMS_OUTPUT.PUT_LINE(' Samedi ') ;
   WHEN 'Sunday' THEN DBMS_OUTPUT.PUT_LINE(' Dimanche ') ;
   ELSE DBMS_OUTPUT.PUT_LINE(' Le week-end est fini ...') ;
END CASE ;
```

FOR IN LOOP

Syntaxe :

```
FOR compteur IN [REVERSE] borneInf..borneSup LOOP
instructions;
END LOOP;
```

- Après chaque itération, le compteur est incrémenté de 1 (ou décrémenté si REVERSE), le pas est toujours de 1 et pas besoin de déclarer de variable compteur

Exemple :

```
For i IN 1..7 LOOP
   DBMS_OUTPUT.PUT_LINE('Le jour de la semaine est '
   || maSemaine(i)) ;
END LOOP ;
```

WHILE LOOP

Syntaxe :

```
WHILE condition LOOP
    instructions;
END LOOP
```

Exemple :

```
i:=1 ;
WHILE (i<8) LOOP
 DBMS_OUTPUT.PUT_LINE('Le jour de la semaine est'
|| maSemaine(i));
i := i+1
END LOOP;
```

LOOP EXIT WHEN

Syntaxe :

```
LOOP
   instructions1;
EXIT [WHEN condition];
   [instructions2;]
END LOOP ;
```

Exemple :

```
i:=1 ;
LOOP
   DBMS_OUTPUT.PUT_LINE('Le jour de la semaine est' ||
maSemaine(i)) ;
   i:=i+1 ;
   EXIT WHEN (i>7) ;
END LOOP ;
```

Plan

- Introduction
- Structure d'un programme
- Les variables
- Les instructions
- Les curseurs
- Les exceptions
- Triggers

Curseurs

```
SELECT nom, prénom INTO v_nom,v_prenom
  FROM EMPLOYE WHERE eno = 666
```

- Récupère l'unique ligne du résultat de la requête (eno est clé)
- La place dans la paire de variables (v_nom,v_prenom)

```
SELECT nom, prénom INTO v_nom,v_prenom
  FROM EMPLOYE WHERE âge > 35
```

- La requête retourne plusieurs lignes
- Plusieurs lignes peuvent être retournées → exception TOO_MANY_ROWS

Les curseurs

- Un curseur est une zone de travail privée de SQL (zone tampon)
- Deux types de curseurs: implicite et explicite
- Les curseurs permettent de parcourir le résultat, n-uplet par n-uplet
- A chaque étape on peut placer le n-uplet courant dans une variable dont le type est soit dérivé du curseur (%ROWTYPE) soit un type structuré (RECORD)

Cycle de vie d'un curseur

Définition dans la section **DECLARE**

```
CURSOR nomCurseur[(paramètres)] IS  < requête >
```

Après **BEGIN** :

Ouverture du curseur (résultat déterminé par la requête mais pas forcément exécuté) :

```
OPEN nomCurseur[(paramètres)];
```

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

```
FETCH nomCurseur INTO Variable;
```

(On peut utiliser BULK COLLECT INTO avec un curseur pour stocker plusieurs lignes du curseur dans un tableau)

Fermeture du curseur (avec libération zone mémoire) :

```
CLOSE nomCurseur;
```

Attributs des curseurs

- `<nomCurseur>%FOUND`

Retourne vrai si le dernier FETCH a retourné une ligne (avant le premier FETCH %FOUND retourne NULL)

- `<nomCurseur>%NOTFOUND`

  l'inverse de %FOUND

- `<nomCurseur>%ROWCOUNT`

Nombre total de lignes traitées jusqu'à présent (nombre de FETCH faits, 0 avant le premier FETCH)

- `<nomCurseur>%ISOPEN`

vrai si la variable curseur est ouverte

Exemple

```
DECLARE
  CURSOR listeEmployes IS SELECT nom, prenom FROM EMPLOYE ;
  v_maliste listeEmployes%ROWTYPE ;
BEGIN
  OPEN listeEmployes ;
  FETCH listeEmployes INTO v_maliste ;
  WHILE(listeEmployes%FOUND) LOOP
    DBMS_OUTPUT.PUT_LINE('employé ' || listeEmployes%ROWCOUNT ||
          's"appelle ' || v_maliste.prenom || ' ' || v_maliste.nom) ;
    FETCH listeEmployes INTO v_maliste ;
  END LOOP ;
  CLOSE listeEmployes ;
END ;
```

Curseurs paramétrés

- Il est possible de paramétrer la requête définissant le curseur

  Exemple :

CURSOR listeDepartement(numDep NUMBER(2)) IS SELECT prenom, nom FROM EMPLOYE WHERE dno=numDep ;

- Les valeurs des paramètres sont transmises lors du OPEN

OPEN listeDepartement(3) ; (employés du département 3)

- Une fois fermé, on peut rappeler le curseur avec d'autres valeurs de paramètres

Curseur implicite

- Créés par défaut après l'exécution des opérations DML : INSERT, UPDATE, DELETE, ou après un SELECT INTO
- Le curseur est désigné par "curseur sql"
- Attributs :
  - *sql%found* : tester si l'opération DML a affecté au moins une ligne ou un **select into** a retourné au moins une ligne (inverse : sql%notfound)
  - *sql%rowcount* : nombre de lignes affectées par l'operation DML ou retournées par **SELECT INTO**
  - *sql%isopen* : retourne toujours FALSE (le curseur est fermé automatiquement après l'opération)

Curseur implicite : FOR IN LOOP

Syntaxe :

```
FOR variable IN (requête) LOOP
instructions;
END LOOP;
```

- On parcourt le résultat d'une requête qu'on stocke ligne par ligne dans une variable (éventuellement structurée)

Exemple : table JOUR(numéro, nom) et variable monjour JOUR%ROWTYPE

```
For monjour IN (SELECT * FROM JOUR) LOOP
   DBMS_OUTPUT.PUT_LINE('Le jour '||monjour.numéro||
   ' de la semaine est '|| monjour.nom) ;
END LOOP ;
```

Curseurs de mises à jour

- Curseurs permettant de verrouiller les données retournées par la requête pour les modifier
- *Syntaxe :*

```
CURSOR nomCurseur[(paramètres)] IS
SELECT listeAttr1 FROM nomTable
WHERE condition
FOR UPDATE [OF listeAttr2]
```

- Tous les attributs de listeAttr2 listeAttr1 sont verrouillés, si pas de OF tous les attributs de listeAttr1 sont verrouillés

Curseurs de mises à jour (suite)

On fait référence à la ligne courante d'un curseur grâce à la syntaxe `CURRENT OF`

Exemples :

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

Remarque : dans les curseurs de mises à jour on ne peut utiliser d'agrégation, d'opérateurs ensemblistes ou de distinct

Plan

- Introduction
- Structure d'un programme
- Les variables
- Les instructions
- Les curseurs
- Les exceptions
- Triggers

Exceptions : principes

- Il existe deux types d'exceptions :
  - générées automatiquement par le système correspondant à une erreur dans l'exécution du code PL/SQL
  - créées par l'utilisateur pour un événement particulier pour lequel on souhaite faire un traitement

- Les exceptions sont « interceptées » dans la section EXCEPTION:
  `WHEN <nomException> THEN<traitementException>`
- Propagation d'une exception déclenchée:
  - PL/SQL cherche d'abord un gestionnaire pour cette excéption (WHEN adéquat) dans la partie EXCEPTION du bloc courant
  - Si pas de gestionnaire, l'erreur est envoyée au bloc englobant...
  - Si pas de gestionnaire dans tous les blocs, on envoie un message d'erreur à l'application appelante

Exceptions : syntaxe

```
EXCEPTION
   WHEN liste_exceptions_1 THEN instructions1;
   WHEN liste_exceptions_2 THEN instructions2;
   ….
   WHEN OTHERS THEN instructions;
END;
```

`liste_exceptions_i` : nomException1 OR nomException2...

`OTHERS` capture toutes les exceptions qui n'ont pas été spécifiées auparavant, doit être après tous les autres identificateur d'exception.

Les fonctions `SQLCODE` et `SQLERRM` permettent de connaître le code et le message d'erreur

Les exceptions prédéfinies

- Sont générées automatiquement par le système suite à l'apparition d'une des erreurs prédéfinies par Oracle (chaque erreur a un numéro)
- Exceptions prédéfinies = quelques erreurs parmi les plus répandues, auxquelles Oracle associe un nom

- Quelques exceptions prédéfinies :
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - CURSOR_ALREADY_OPEN – on essaie d'ouvrir un curseur déjà ouvert
  - INVALID_CURSOR – on essaie de fermer un curseur qui n'est pas ouvert
  - ZERO_DIVIDE – division par zéro
  - VALUE_ERROR – erreur de conversion
  - STORAGE_ERROR – plus de mémoire

Exemple : exceptions prédéfinies

Emp (Eno, Ename, Title, City) **Project**(Pno, Pname, Budget, City)

Pay(Title, Salary) **Works**(Eno, Pno, Resp, Dur)

```
DECLARE
    v_name  EMP.Ename%TYPE ;
BEGIN
    SELECT Ename INTO v_name FROM Emp WHERE Title='Analyst' ;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
            dbms_output.put_line('Aucun analyste trouvé !') ;
     WHEN TOO_MANY_ROWS THEN
            dbms_output.put_line('Il existe plusieurs analystes !') ;
      WHEN OTHERS THEN
            dbms_output.put_line('Autre erreur rencontrée :  ' || SQLERRM) ;
END ;
/
```

Créer ses propres exceptions

Créer ses propres exceptions:

1. Déclarer ses exceptions dans la section `DECLARE`:

```
MonException EXCEPTION;
```

2. On “lève” l'exception dans le codePL/SQL:

```
RAISE MonException;
```

(RAISE permet aussi de lever des exceptions prédéfinies)

3. On la capture dans le bloc `EXCEPTION`:

```
WHEN MonException THEN...
```

Exceptions non nommées

- Codes d'erreur Oracle sans nom prédéfini
- Traitées dans la partie WHEN OTHERS de la section EXCEPTION

  Exemple :

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

- Si l'on souhaite leurs associer un traitement spécifique :
  - Dans la section DECLARE :

    monException EXCEPTION ; ← Déclarer un nom associé

    PRAGMA EXCEPTION_INIT(monException, codeErrOracle) ← Associer le code d'erreur Oracle avec l'identificateur monException
  - Dans la section EXCEPTION :

    WHEN monException THEN ← traitement spécifique

Exemple : exceptions utilisateur

Emp (Eno, Ename, Title, City) **Project**(Pno, Pname, Budget, City)

Pay(Title, Salary) **Works**(Eno, Pno, Resp, Dur)

```
DECLARE
    v_sal  NUMBER ;
    v_total NUMBER ;
    mon_exception EXCEPTION ;
BEGIN
     SELECT COUNT(*) INTO v_sal FROM Pay WHERE Salary < 2000 ;
     SELECT COUNT(*) INTO v_total FROM PAY ;
     IF v_sal/v_total > 0.5 THEN RAISE mon_exception ;

EXCEPTION
     WHEN mon_exception THEN
              dbms_output.put_line('Plus de la moitié des salaires sont < 2000') ;
END ;
/
```

Exceptions avec message et code d'erreur personnalisés

- Créer et lever en même temps une exception avec un code d'erreur et un message personnalisés:

```
raise_application_error(codeErr,messageErr[,TRUE|FALSE])
```

- Arrête l'exécution du programme, annule les modifications des données et génère une erreur

  - (code_erreur entre -20000 et -20999)
  - TRUE (par défaut): erreur mise dans une pile d'erreurs à propager
  - FALSE: remplace les erreurs précédentes dans la pile

- Peuvent également être interceptées et traitées en utilisant:

  PRAGMA EXCEPTION_INIT...

- Ne peut pas être utilisé dans des blocs anonymes, seulement dans des programmes (procédures et fonctions) stockés

Exemple (1)

Emp (Eno, Ename, Title, City) **Project**(Pno, Pname, Budget, City)
Pay(Title, Salary) **Works**(Eno, Pno, Resp, Dur)

```
CREATE OR REPLACE PROCEDURE test_budget(nom Projet.Pname%TYPE)  AS
    v_budget Projet.Budget%TYPE ;
    budget_manquant EXCEPTION ;
    PRAGMA   EXCEPTION_INIT(budget_manquant, -20001) ;
BEGIN
    SELECT  Budget  INTO v_budget FROM Projet WHERE Pname=nom ;
    IF Budget  IS  NULL THEN
        raise_application_error(-20001, 'Budget du projet '||
                                    nom || 'n''est pas renseigné') ;
    END IF ;
    EXCEPTION
        WHEN budget_manquant THEN
            dbms_output.put_line(SQLERRM) ;
      END;
      /
```

Exemple (2)

Emp (Eno, Ename, Title, City) **Project**(Pno, Pname, Budget, City)

Pay(Title, Salary) **Works**(Eno, Pno, Resp, Dur)

```
CREATE OR REPLACE PROCEDURE test_budget (nom Projet.Pname%TYPE) AS
    v_budget Projet.Budget%TYPE ;
    v_codeErr NUMBER ;
BEGIN
    SELECT  Budget  INTO v_budget FROM Projet WHERE Pname=nom ;
    IF Budget  IS  NULL THEN
        raise_application_error(-20001, 'Budget du projet '|| nom || 'n''est pas renseigné') ;
    END IF ;
    EXCEPTION
        WHEN  OTHERS THEN
            v_codeErr  := SQLCODE ;
            IF (v_codeErr = -20001) THEN
                dbms_output.put_line(SQLERRM) ;
            END IF ;
  END;
```

Exemples

COMMERCIAL(id, nomC, prénom,ancienneté)

SECTEUR(code, nomS, surfaceS)

VILLE(nomV, codeSecteur, population, surfaceV)

AFFECTATION(idCommercial, codeSecteur, dateDébut, dateFin)

Exemple 1

```
COMMERCIAL(id, nomC, prénom,ancienneté) SECTEUR(code, nomS, surfaceS)
VILLE(nomV, codeSecteur, population, surfaceV)
AFFECTATION(idCommercial, codeSecteur, dateDébut, dateFin)
```

Afficher la surface totale des secteurs auxquels est affecté le commercial.

```
CREATE OR REPLACE PROCEDURE test_occupation(v_id COMMERCIAL.id%TYPE) AS
  surfaceTotale SECTEUR.surfaceS%TYPE;
BEGIN
        SELECT sum(surfaceS) INTO surfaceTotale
        FROM secteur, affectation
        WHERE code=codeSecteur and idCommercial=v_id
                    and sysdate between dateDebut and dateFin;
        IF(surfaceTotale IS NULL) THEN
         DBMS_OUTPUT.PUT_LINE('Le commercial '||v_id || ' n'existe pas ou ne travaille pas
aujourd'hui');
         ELSE DBMS_OUTPUT.PUT_LINE('Le commercial '||v_id|| ' couvre '|| surfaceTotale ||'
km2 aujourd'hui');
 END IF;
END;
/
```

Exemple 2

COMMERCIAL(id, nomC, prénom,ancienneté) **SECTEUR**(code, nomS, surfaceS)

VILLE(nomV, codeSecteur, population, surfaceV)

AFFECTATION(idCommercial, codeSecteur, dateDébut, dateFin)

Afficher les noms des villes avec les noms de leurs secteurs + total des villes listées

```
CREATE OR REPLACE PROCEDURE lister_villes AS
  cursor listeVilles IS SELECT nomV, nomS  FROM ville, secteur where codesecteur=code;
 v_ville listeVilles%ROWTYPE;
BEGIN
   OPEN listeVilles;
    FETCH listeVilles INTO v_ville;
    WHILE(listeVilles%FOUND) LOOP
        DBMS_OUTPUT.PUT_LINE(v_ville.nomV || ' ' || v_ville.nomS);
        FETCH listeVilles INTO v_ville;
     END LOOP;
     DBMS_OUTPUT.PUT_LINE('Nombre total de villes: '|| listeVilles%rowCount);
     close listeVilles;
END;
/
```

Exemple 3 (1)

Lister secteurs avec commerciaux affectés à ces secteurs.

```
CREATE OR REPLACE PROCEDURE listeSecteurs AS
      cursor listeSecteur IS SELECT * SELECT secteur;
      v_secteur listeSecteur%ROWTYPE;
      cursor listeAffectation(p_secteur SECTEUR.code%TYPE) IS
     SELECT nomC, prenom, dateDebut, dateFin
     FROM commercial, affectation
     WHERE id=idcommercial and codeSecteur=p_secteur
     ORDER BY dateDebut;
      v_affectation listeAffectation%ROWTYPE;
BEGIN
      OPEN listeSecteur;
      FETCH listeSecteur INTO v_secteur;
```

Exemple 3 (2)

```
WHILE(listeSecteur%FOUND) LOOP
    DBMS_OUTPUT.PUT_LINE('Affectations pour le secteur '||
v_secteur.nomS
            || 'de surface '|| v_secteur.surfaceS);
            OPEN listeAffectation(v_secteur.code);
            FETCH listeAffectation INTO v_affectation;
            WHILE(listeAffectation%FOUND) LOOP
                DBMS_OUTPUT.PUT_LINE('   '|| v_affectation.dateDebut||'
a ' ||v_affectation.dateFin || ': ' ||
                v_affectation.nomC|| ' '||v_affectation.prenom);
                FETCH listeAffectation INTO v_affectation;
            END LOOP;
            CLOSE listeAffectation;
            FETCH listeSecteur INTO v_secteur;
      END LOOP;
      CLOSE listeSecteur;
END;
/
```

Exemple 4

Tester si un secteur n'est pas affecté à un créneau donné

```
CREATE OR REPLACE FUNCTION test_secteur(p_code
AFFECTATION.codeSecteur%TYPE, p_debut AFFECTATION.dateDebut%TYPE,
p_fin
AFFECTATION.dateFin%TYPE) RETURN BOOLEAN AS
    v_affect AFFECTATION.idCommercial%TYPE;
BEGIN
    SELECT idCommercial into v_affect
    FROM affectation
    WHERE  codeSecteur=p_code and p_debut<=dateFin and p_fin>=dateDebut;
    return true;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN return false;
END;
/
```

Exemple 5

Fonction qui retourne la liste des villes rattachées à un secteur donné

```
CREATE OR REPLACE FUNCTION listeVilles(p_code
VILLE.codeSecteur%TYPE)
RETURN VARCHAR IS
      cursor lesVilles IS   SELECT  nomV FROM ville
                            WHERE codeSecteur=p_code;
      v_ville lesVilles%ROWTYPE;
      v_liste VARCHAR2(400):='';
BEGIN
  OPEN lesVilles;
   FETCH lesVilles INTO v_ville;
   WHILE(lesVilles%FOUND) LOOP
      v_liste:=v_liste || ' ' || v_ville.nomv ; FETCH lesVilles INTO v_ville;
   END LOOP;
   CLOSE lesVilles;
  RETURN v_liste;
END;
/
```

Plan

- Introduction
- Structure d'un programme
- Les variables
- Les instructions
- Les curseurs
- Les exceptions
- Triggers

Triggers : utilisation

« Règles actives » (ECA) *généralisant* les contraintes d'intégrité **:**

- génération automatique de valeurs manquantes (ex. valeur dérivée, par défaut)
- éviter des modifications invalides (C: test, A: abort)
- implantation de règles applicatives (« business rules »)
- génération de traces d'exécution, statistiques, ...
- maintenance de répliques
- propagation de mises-à-jour sur des vues vers les tables
- intégrité référentielle entre des données distribuées
- interception d'événements utilisateur / système (LOGIN, STARTUP, …)

Trigger ou règle active ou règle ECA

Définition ECA :

Événement (E) :

- une mise-à-jour de la BD qui active le trigger, ou d'autres (opérations DDL, logon, serverror, ..)

  ex.: réservation de place

Condition (C):

- un test ou une requête devant être vérifié lorsque le trigger est activé (une requêtes est *vraie* si sa réponse n'est pas vide)

  ex.: nombre de places disponibles ?

Action (A):

- une procédure exécutée lorsque le trigger est activé et la condition est *vraie* : E,C → A

  ex.: annulation de réservation

Exécution des triggers (1)

Moment de déclenchement du trigger par rapport à l'événement *E* (maj. activante) :

- avant (before) E
- après (after) *E*
- à la place de (instead of) de *E* (spécifique aux vues => màj des données de la base)

Nombre d'exécutions de l'action A par déclenchement :

- une exécution de l'action *A* par n-uplet modifié (ROW TRIGGER)
- une exécution de l'action *A* par événement (STATEMENT TRIGGER)

Exécution des triggers (2)

Delta structure : « les données considérées par le trigger »

- *:old* avant l'événement, *:new* après l'événement (peuvent être renommés)
- *for each row* : un n-uplet, *for each statement*: un ensemble de n-uplets
- *:new* peut être modifié par l'action, mais effet seulement si *before*
- Pour agir avec un trigger *after*, il faut modifier directement la base
- *:old* (resp. *:new*) n'a pas de sens pour *insert* (resp. *delete*)

Syntaxe (Oracle)

```
{ CREATE | REPLACE } TRIGGER <nom>
// Événement
{BEFORE | AFTER | INSTEAD OF}
{INSERT | DELETE | UPDATE [OF <attribut, …>]}
    ON <table>
[REFERENCING [NEW AS <nouv>] [OLD AS <anc>]]
[FOR EACH ROW] // ROW TRIGGER
// Condition
[WHEN  (<condition SQL>) THEN]
// Action
<Procédure SQL>
```

Contrôle d'intégrité

Emp (Eno, Ename, Title, City)

Vérification de la contrainte de clé à l'insertion d'un nouvel employé :

```
CREATE TRIGGER InsertEmp
BEFORE INSERT ON Emp
REFERENCING NEW AS N
FOR EACH ROW
WHEN EXISTS
     (SELECT * FROM Emp WHERE Eno=N.Eno)
THEN
     ABORT;
```

Contrôle d'intégrité

Emp (Eno, Ename, Title, City) **Pay**(Title, Salary)

Suppression d'un titre et des employés correspondants (« ON DELETE CASCADE ») :

```
CREATE TRIGGER DeleteTitle
BEFORE DELETE ON Pay
REFERENCING OLD AS O
FOR EACH ROW
BEGIN
    DELETE FROM Emp WHERE Title=O.Title
END;
```

Mise-à-jour automatique

Emp (Eno, Ename, Title, City)

Création automatique d'une valeur de clé (autoincrément) :

```
CREATE TRIGGER SetEmpKey
BEFORE INSERT ON Emp
REFERENCING NEW AS N
FOR EACH ROW
BEGIN
  N.Eno := SELECT COUNT(*) FROM Emp
END;

/* le premier Eno sera 0 */
```

Mise-à-jour automatique

Pay(Title, Salary, Raise)

Maintenance des augmentations (raise) de salaire :

```
CREATE TRIGGER UpdateRaise
AFTER UPDATE OF Salary ON Pay
REFERENCING OLD AS O, NEW AS N
FOR EACH ROW
BEGIN
     UPDATE Pay
     SET Raise = N.Salary - O.Salary
     WHERE Title = N.Title;
END
```

Analyse des triggers

Plusieurs triggers de type différent peuvent être affectés au même événement :

- Ordre par défaut : BEFORE STATEMENT → BEFORE ROW → AFTER ROW → AFTER STATEMENT

Un trigger activé peut en activer un autre :

- longues chaînes d'activation => problème de performances
- boucles d'activation => problème de terminaison

Recommandations :

- pour l'intégrité, utiliser si possible le mécanisme des contraintes plus facile à optimiser par le système.
- associer les triggers à des règles de gestion.