

Cours 2 : PL/SQL

Procedural Language/SQL

Blocs, variables, instructions, structures de
contrôle, curseurs, gestion des erreurs,
procédures/fonctions stockées, packages,
triggers

PL/SQL

- ❖ Chapitre 3 de la norme SQL3 sous le nom SQL/PSM (Persistent Stored Modules)
- ❖ Langage procédural plus portable
- ❖ Un script SQL Developpeur peut contenir des blocs de sous-programmes en PL/SQL
- ❖ Traitement de transactions

PL/SQL (2)

- ❖ Construction de procédures ou fonctions stockées qui améliorent le mode client-serveur par stockage des procédures ou fonctions souvent utilisées au niveau serveur
- ❖ Gestion des erreurs (à la ADA)
- ❖ Construction de triggers (ou déclencheurs)

Structure d'un bloc

❖ Un programme ou une procédure PL/SQL est un ensemble de un ou plusieurs blocs. Chaque bloc comporte trois sections :

1. Section déclaration
2. Section corps du bloc
3. Section traitement des erreurs

1. Section déclaration

- ❖ Contient la description des structures et des variables utilisées dans le bloc
- ❖ Section facultative
- ❖ Commence par le mot clé **DECLARE**

2. Section corps du bloc

- ❖ Contient les instructions du programme et éventuellement, à la fin, la section de traitement des erreurs
- ❖ Obligatoire
- ❖ Introduite par le mot clé **BEGIN**
- ❖ Se termine par le mot clé **END**

3. Section traitement des erreurs

- ❖ Facultative

- ❖ Introduite par le mot clé **EXCEPTION**

Syntaxe

DECLARE

déclaration

BEGIN

corps-du-bloc

EXCEPTION

traitement-des-erreurs

END;

**/ ← A ajouter obligatoirement
dans l'exécution d'un script**

Exemple

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
  x VARCHAR2(10);
```

```
BEGIN
```

```
  x := 'Bonjour';
```

```
  DBMS_OUTPUT.PUT_LINE(x);
```

```
END;
```

```
/
```

Exemple (2)

DECLARE

erreurNb EXCEPTION;

nom Voyage.nomVoyage%TYPE;

min Voyage.nbMin%TYPE;

max Voyage.nbMax%TYPE;

BEGIN

...

Exemple (2 – suite)

• • •

BEGIN

```
SELECT nomVoyage, nbMin, nbMax  
INTO nom, min, max  
FROM Voyage WHERE numVoyage = 1;  
IF max < min
```

```
    THEN RAISE erreurNb; END IF;
```

```
DBMS_OUTPUT.PUT_LINE (nom || ' OK' );
```

EXCEPTION

• • •

Exemple (2 – suite et fin)

...

EXCEPTION

WHEN NO_DATA_FOUND THEN

DBMS_OUTPUT.PUT_LINE(
 'numéro inconnu');

WHEN erreurNb THEN

DBMS_OUTPUT.PUT_LINE(
 nom || ' NBMIN > NBMAX');

END;

/

Types de variables

❖ Variables scalaires

❖ Types composés

- Record
- Table

Variables scalaires

- ❖ Types issus de SQL : **CHAR**, **NUMBER**, **DATE**, **VARCHAR2**
- ❖ Types PL/SQL : **BOOLEAN**, **SMALLINT**, **BINARY_INTEGER**, **DECIMAL**, **FLOAT**, **INTEGER**, **REAL**, **ROWID**
- ❖ Les variables hôtes sont préfixées par « : »

Déclaration des variables scalaires

nom-variable nom-du-type;

nom-variable nom-table.nom-attribut%TYPE;

Déclaration pour un enregistrement (record)

- ❖ Soit par référence à une structure de table ou de curseur en utilisant **ROWTYPE**

nom-variable ***nom-table***%ROWTYPE;

nom-variable ***nom-curseur***%ROWTYPE;

Déclaration pour un enregistrement (record) (2)

- ❖ Soit par énumération des rubriques qui la composent. Cela se fait en deux étapes :
 - Déclaration du type enregistrement
TYPE *nom-du-type* **IS RECORD** (
nom-attribut₁ *type-attribut₁*,
nom-attribut₂ *type-attribut₂*, ...);
 - Déclaration de la variable de type enregistrement
nom-variable *nom-du-type*;

Exemple

```
DECLARE
TYPE recVoyage IS RECORD (
libelle Voyage.nomVoyage%TYPE,
prixPropose NUMBER(8,2));
voyPerso recVoyage; ...
BEGIN
voyPerso.prixPropose := 3601.43;

...
```

Tables

- ❖ Structure composée d'éléments d'un même type *scalaire*
- ❖ L'accès à un élément de la table s'effectue grâce à un indice, ou clé primaire
- ❖ Cet index est déclaré de type **BINARY_INTEGER** (valeurs entières signées)

Déclaration pour une table

❖ Deux étapes :

- Déclaration du type de l'élément de la table :

TYPE *nom-du-type* IS TABLE OF *type-argument* INDEX BY BINARY_INTEGER;

- Déclaration de la variable de type table :
nom-variable nom-du-type;

Exemple

```
DECLARE
TYPE tabNom IS TABLE OF VARCHAR2(20)
INDEX BY BINARY_INTEGER;
tableNom tabNom; ...
BEGIN
tableNom(3) := 'Dupont';
...

```

Variables (scalaires ou composées)

❖ Valeur initiale :

nom-variable *nom-du-type* := *valeur*;

❖ Constante :

nom-variable *nom-du-type* DEFAULT *valeur*;

ou

nom-variable CONSTANT *nom-du-type* := *valeur*;

Variables (scalaires ou composées) (2)

- ❖ Visibilité : une variable est utilisable dans le bloc où elle a été définie ainsi que dans les blocs imbriqués dans le bloc de définition, sauf si elle est redéfinie dans un bloc interne

Conversion de type

- ❖ Explicite avec
**TO_CHAR, TO_DATE, TO_NUMBER,
RAWTOHEX, HEXTORAW**
- ❖ Implicites, par conversion automatique

Instructions

- ❖ Affectations
- ❖ Instructions du langage SQL : CLOSE, COMMIT, DELETE, FETCH, INSERT, LOCK, OPEN, ROLLBACK, SAVEPOINT, SELECT, SET TRANSACTION, UPDATE

Instructions (2)

- ❖ Instructions de contrôle itératif ou répétitif
- ❖ Instructions de gestion de curseurs
- ❖ Instructions de gestion des erreurs

Affectation

- ❖ Opérateur d'affectation **:=**
- ❖ Option **INTO** dans un ordre **SELECT**
- ❖ Instruction **FETCH** avec un curseur

Exemple

```
DECLARE
  TYPE tabNom IS TABLE OF VARCHAR2(20)
  INDEX BY BINARY_INTEGER;
  tableNom tabNom;
  i BINARY_INTEGER;
  TYPE recVoyage IS RECORD (
    libelle Voyage.nomVoyage%TYPE,
    prixPropose NUMBER(8,2));
  voyPerso recVoyage;
  convers NUMBER(8,6);
BEGIN
  ...
```

Exemple (suite)

```
...  
BEGIN  
  converts := 6.55957;  
  tableNom(5) := 'Dupont';  
  i := 10;  
  tableNom(i) := 'Dupond';  
  voyPerso.libelle := 'Découverte du Japon';  
  voyPerso.prixPropose := 1989.08;  
END;
```

Exemple (2)

DECLARE

v_nom Voyage.nomVoyage%TYPE;

v_min Voyage.nbMin%TYPE;

TYPE recVoyage IS RECORD (

 r_nom Voyage.nomVoyage%TYPE,

 r_min Voyage.nbMin%TYPE);

r_voyage recVoyage;

rr_Voyage Voyage%ROWTYPE;

BEGIN

Exemple (2 – suite et fin)

...

BEGIN

SELECT nomVoyage, nbMin

INTO v_nom, v_min

FROM Voyage **WHERE** numVoyage = 517;

SELECT nomVoyage, nbMin **INTO** r_Voyage

FROM Voyage **WHERE** numVoyage = 364;

SELECT * **INTO** rr_Voyage

FROM Voyage **WHERE** numVoyage = 618;

END;

Structures de contrôle

❖ Structure alternative

❖ Structure répétitives

Structures alternatives

```
IF condition THEN instructions;  
END IF;
```

```
IF condition THEN instructions;  
ELSE instructions; END IF;
```

```
IF condition THEN instructions;  
ELSIF condition THEN instructions;  
ELSE instructions; END IF;
```

Structures répétitives

```
LOOP instructions; END LOOP;
```

```
LOOP instructions; ...  
EXIT WHEN condition; ...  
END LOOP;
```

```
LOOP ...  
IF condition THEN EXIT; END IF;  
... END LOOP;
```

Structures répétitives (2)

```
FOR variable-indice IN [REVERSE]  
  val-début .. val-fin  
  LOOP instructions; END LOOP;
```

- ❖ *variable-indice* est une variable locale (locale à la boucle) **non déclarée**
- ❖ *val-début* et *val-fin* sont des variables locales **déclarées** et initialisées ou alors des constantes
- ❖ le pas est -1 si **REVERSE** est présent, sinon il est égal à +1

Structures répétitives (3)

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

Les curseurs

- ❖ Il y a création d'un curseur dès qu'on exécute une instruction SQL. C'est une zone de travail de l'environnement utilisateur qui contient les informations relatives à l'instruction SQL :
 - Le texte source de l'ordre SQL
 - Le texte « compilé » de l'ordre SQL
 - Un tampon pour une ligne du résultat
 - Le statut (*cursor status*)
 - Des informations de travail et de contrôle

Curseurs implicites

- ❖ Gérés automatiquement par le noyau dans les cas suivants :
 - Une instruction **SELECT** exécutée sous SQL Developer
 - Une instruction **SELECT** donnant une seule ligne de résultat sous PL/SQL
 - Les instructions **UPDATE**, **INSERT** et **DELETE**
 - ...

Curseurs explicites

- ❖ Obligatoires pour un **SELECT** susceptible de produire plusieurs lignes résultat
- ❖ Quatre étapes :
 - 1) Déclaration du curseur
 - 2) Ouverture du curseur
 - 3) Traitement des lignes du résultat
 - 4) Fermeture du curseur

1) Déclaration du curseur

- ❖ Association d'un nom de curseur à une requête **SELECT**
- ❖ Se fait dans la section **DECLARE** d'un bloc PL/SQL

CURSOR *nom-curseur* **IS** *requête*;

- ❖ Un curseur peut être paramétré :

CURSOR *nom-curseur* (*nom-p₁* *type-p₁*
[**:=** *val-défaut*], ...) **IS** *requête*;

Exemple

```
DECLARE
```

```
CURSOR C1 IS SELECT numVoyage  
FROM Tarif WHERE prix > 1000;
```

```
CURSOR C2 (p NUMBER(8), q NUMBER(8)) IS  
SELECT nomVoyage FROM Voyage  
WHERE nbMin >= p  
AND nbMax <= q;
```

```
BEGIN
```

```
...
```

2) Ouverture d'un curseur

- ❖ Alloue un espace mémoire au curseur et positionne les éventuels verrous

OPEN *nom-curseur*;

ou

OPEN *nom-curseur*(*liste-par-effectifs*) ;

- ❖ Pour les paramètres, association par position ou par nom sous la forme
paramètre-formel => *paramètre-réel*

Exemple

❖ OPEN C1 ;

❖ OPEN C2 (6 , 20) ;

❖ OPEN C2 (q => 20 , p => 6) ;

3) Traitement des lignes

- ❖ Autant d'instructions **FETCH** que de lignes résultats :

FETCH *nom-curseur*
INTO *liste-variables*;

ou

FETCH *nom-curseur*
INTO *nom-enregistrement*;

- ❖ Au moins quatre formes possibles

Première forme : exemple

```
DECLARE
  CURSOR C3 IS SELECT numVoyage,
    nomVoyage FROM Voyage;
  v_num Voyage.numVoyage%TYPE;
  v_nom Voyage.nomVoyage%TYPE;
BEGIN
  OPEN C3; LOOP
    FETCH C3 INTO v_num, v_nom;
    EXIT WHEN C3%NOTFOUND;
    Traitement; END LOOP; CLOSE C3;
END;
```

Deuxième forme : exemple

```
DECLARE
  CURSOR C3 IS SELECT numVoyage,
    nomVoyage FROM Voyage;
  TYPE recVOY IS RECORD(
    v_num Voyage.numVoyage%TYPE,
    v_nom Voyage.nomVoyage%TYPE);
  r_voy recVOY;
BEGIN
  OPEN C3; LOOP
    FETCH C3 INTO r_voy;
    EXIT WHEN C3%NOTFOUND;
    Traitement; END LOOP; CLOSE C3;
END;
```

Troisième forme : exemple

```
DECLARE
  CURSOR C3 IS SELECT numVoyage,
    nomVoyage FROM Voyage;
  r_voy C3%ROWTYPE;
BEGIN
  OPEN C3; LOOP
    FETCH C3 INTO r_voy;
    EXIT WHEN C3%NOTFOUND;
    Traitement; END LOOP; CLOSE C3;
END;
```

Quatrième forme : exemple

```
DECLARE
  CURSOR C3 IS SELECT numVoyage,
    nomVoyage FROM Voyage;
  v_num Voyage.numVoyage%TYPE;
  v_nom Voyage.nomVoyage%TYPE;
BEGIN
  FOR rec IN C3 LOOP
    v_num := rec.numVoyage;
    v_nom := rec.nomVoyage;
    Traitement; END LOOP;
END;
```


Statut d'un curseur

Attribut	Valeur
%FOUND	Vrai si exécution correcte de l'ordre SQL
%NOTFOUND	Vrai si exécution incorrecte de l'ordre SQL
%ISOPEN	Vrai si curseur ouvert
%ROWCOUNT	Nombre de lignes traitées par l'ordre SQL, évolue à chaque ligne traitée par un FETCH (zéro au départ)

Statut d'un curseur (2)

Curseur implicite	Curseur explicite
SQL%FOUND	<i>nom-curseur</i> %FOUND
SQL%NOTFOUND	<i>nom-curseur</i> %NOTFOUND
SQL%ISOPEN	<i>nom-curseur</i> %ISOPEN
SQL%ROWCOUNT	<i>nom-curseur</i> %ROWCOUNT

Modification des données

- ❖ Se fait habituellement avec **INSERT**, **UPDATE** ou **DELETE**
- ❖ Possibilité d'utiliser la clause **FOR UPDATE** dans la déclaration du curseur. Cela permet d'utiliser la clause

CURRENT OF *nom-curseur*

dans la clause **WHERE** des instructions **UPDATE** et **DELETE**. Cela permet de modifier la ligne du curseur traitée par le dernier **FETCH**, et donc d'accélérer l'accès à cette ligne

Example

```
DECLARE
CURSOR C IS SELECT *
FROM Tarif FOR UPDATE OF prix;
aug Tarif.prix%TYPE;
BEGIN
FOR rec IN C LOOP
IF rec.prix < 1000 THEN aug := rec.prix*0.2;
ELSIF rec.prix < 2000 THEN aug := rec.prix*0.1;
ELSE aug := 0; DBMS_OUTPUT.PUT_LINE('OK ' ||
    rec.numvoyage || ' ' || rec.datedeb); END IF;
UPDATE Tarif
SET prix = prix+aug WHERE CURRENT OF C; ←
END LOOP;
END;
```

Modification des données (2)

- ❖ Dans le cas d'une clause **FOR UPDATE**, la table est verrouillée en mode row share (RS). Les lignes concernées par le verrou sont les lignes du **SELECT** de la définition du curseur
- ❖ En général, un **COMMIT** à l'emplacement de la flèche ferme le curseur. Mais ça n'est pas vrai sous Oracle en PL/SQL

Gestion des erreurs (erreurs standard)

Code d'erreur SQLCODE	Erreur
100	NO_DATA_FOUND
-1	DUP_VAL_ON_INDEX
-6502	VALUE_ERROR
-1001	INVALID_CURSOR
-1722	INVALID_NUMBER
-6501	PROGRAM_ERROR
-1017	LOGIN_DENIED
-1422	TOO_MANY_ROWS
-1476	ZERO_DIVIDE

Gestion des erreurs (erreurs standard) (2)

- ❖ La nature d'une erreur peut être connue par appel aux fonctions **SQLCODE** et **SQLERRM**
- ❖ **SQLCODE** renvoie le statut d'erreur de la dernière instruction SQL exécutée (0 si n'y a pas d'erreur)
- ❖ **SQLERRM** renvoie le message d'erreur correspondant à **SQLCODE**

Erreurs utilisateur

DECLARE

nom-anomalie EXCEPTION;

BEGIN

...

IF ... THEN RAISE *nom-anomalie*;

...

EXCEPTION

WHEN *nom-anomalie* THEN *traitement*;

END;

Erreurs anonymes

- ❖ Pour les codes d'erreur n'ayant pas de nom associé, il est possible de définir un nom d'erreur (code entre -20000 et -20999)

Exemple

```
DECLARE
  e EXCEPTION;
  PRAGMA EXCEPTION_INIT(e, -20091);
  ...
BEGIN
  ...
  IF ... THEN RAISE e;
EXCEPTION
  WHEN e THEN ...
END;
```

Exemple (2)

```
DECLARE
    ...
BEGIN
    ...
EXCEPTION
    IF ... THEN
        RAISE_APPLICATION_ERROR(
            -20099, 'nom inexistant');
END;
```

Description du traitement de l'erreur (syntaxe)

BEGIN

...

EXCEPTION

WHEN *nom-erreur₁* THEN *traitement-erreur₁* ;

...

WHEN *nom-erreur_n* THEN *traitement-erreur_n* ;

WHEN OTHERS THEN *traitement-autres-erreurs* ;

END ;

Description du traitement de l'erreur (syntaxe) (2)

❖ Possibilité d'écrire :

```
WHEN nom-erreur1 OR nom-erreur2  
THEN . . . ;
```

Example

```
...  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN ...;  
    WHEN OTHERS THEN  
        DBMS_OUTPUT.PUT_LINE(  
            'SQLCODE = ' || TO_CHAR(SQLCODE) );  
        DBMS_OUTPUT.PUT_LINE(  
            'SQLERRM : ' || TO_CHAR(SQLERRM) );  
END;
```

Procédures stockées

```
CREATE [OR REPLACE] PROCEDURE nom-procédure  
[ (argument [mode] type, ... ) ]  
[ IS | AS ]  
bloc-procédure;
```

- ❖ *argument* : nom d'un paramètre formel
- ❖ *mode* : définit si le paramètre formel est en entrée (**IN**), en sortie (**OUT**) ou en entrée-sortie (**IN OUT**).
Par défaut : **IN**
- ❖ *type* : le type du paramètre formel
- ❖ *bloc-procédure* : le corps PL/SQL de la procédure

Exemple

```
CREATE OR REPLACE PROCEDURE
modifierPrix (num IN Voyage.numVoyage%TYPE) IS
BEGIN
BEGIN
UPDATE Tarif SET prix = prix * .50
WHERE numVoyage = num AND dateDeb < '31-12-16';
END;
BEGIN
UPDATE Tarif SET prix = prix * .60
WHERE numVoyage = num AND dateDeb > '31-12-17';
END;
END modifierPrix;
```


Fonctions stockées

```
CREATE [OR REPLACE] FUNCTION nom-fonction  
[(argument [IN] type, ...)]  
RETURN type-retour  
[IS | AS]  
bloc-fonction;
```

- ❖ Les paramètres sont forcément en entrée (**IN**)
- ❖ Dans le *bloc-fonction* :

```
RETURN nom-variable;
```

Exemple

```
CREATE OR REPLACE FUNCTION
moyennePrix (dd IN DATE ) RETURN NUMBER IS
moy Tarif.prix%TYPE := 0;
e EXCEPTION;
BEGIN
SELECT AVG(prix) INTO moy FROM Tarif
WHERE dateDeb = dd;
IF moy IS NULL THEN RAISE e; END IF;
RETURN moy;
EXCEPTION
WHEN e THEN RETURN 0;
END moyennePrix;
```

Informations à propos des procédures/fonctions

❖ Erreurs

- **USER_ERRORS**
- **ALL_ERRORS**
- **DBA_ERRORS**

❖ Infos sur les procédures/fonctions :

- **USER_OBJECTS**
- **ALL_OBJECTS**
- **DBA_OBJECTS**

❖ Infos sur les textes source :

- **USER_SOURCE**
- **ALL_SOURCE**
- **DBA_SOURCE**

Suppression d'une procédure/fonction stockée

DROP PROCEDURE *nom-procedure*;

DROP FUNCTION *nom-fonction*;

Appel d'une procédure/fonction stockée dans un bloc PL/SQL

***nom-procédure** (**liste-paramètres-effectifs**) ;*

***nom-variable** := **nom-fonction**(**liste-paramètres-effectifs**) ;*

Appel d'une procédure/fonction stockée dans un script SQL Developer

EXECUTE *nom-procédure* (*liste-paramètres-effectifs*) ;

EXECUTE : *nom-variable* := *nom-fonction*
(*liste-paramètres-effectifs*) ;

EXECUTE DBMS_OUTPUT.PUT_LINE(*nom-fonction* (*liste-paramètres-effectifs*)) ;

Exemple

1. Création procédure

CREATE OR REPLACE PROCEDURE

nvVoy(**num** Voyage.numVoyage%TYPE,
 nom Voyage.nomVoyage%TYPE)

IS

BEGIN

 INSERT INTO **Voyage** (numVoyage,
 nomVoyage) VALUES (num, nom);

END;

Exemple

2. Exécution procédure

EXECUTE

nvVoy (400, 'Voyage en Irlande');

Exemple

2^{bis}. Exécution procédure

```
EXECUTE nvVoy (&num, &nom);
```

Enter Substitution Variable

NUM: 400

Enter Substitution Variable

NOM: 'Voyage en Irlande'

Exemple

2^{ter}. Exécution procédure

PROMPT Entrez les infos sur un
nouvel voyage

ACCEPT num **PROMPT** Numéro

ACCEPT nom **PROMPT** Nom

Numéro : 400

Nom : 'Voyage en Irlande'

EXECUTE nvVoy (&num, &nom);

Exemple

1. Création fonction

```
CREATE OR REPLACE FUNCTION
```

```
moyenne(nom Voyage.nomVoyage%TYPE)
```

```
RETURN NUMBER IS moy Tarif.prix%TYPE;
```

```
BEGIN
```

```
SELECT AVG(prix) INTO moy FROM Tarif
```

```
WHERE numVoyage IN (SELECT numVoyage
```

```
FROM Voyage WHERE nomVoyage = nom);
```

```
RETURN moy;
```

```
END;
```

Exemple

2. Exécution fonction

```
EXECUTE DBMS_OUTPUT.PUT_LINE (  
    moyenne('Circuit Heleni'));
```

Exemple

2^{bis}. Exécution fonction

```
EXECUTE DBMS_OUTPUT.PUT_LINE (  
    moyenne( &nom ) );
```

```
Enter Substitution Variable  
NOM: 'Circuit Heleni'
```

Exemple

2^{ter}. Exécution fonction

```
ACCEPT nom PROMPT Nom
```

```
Nom : 'Circuit Heleni'
```

```
EXECUTE DBMS_OUTPUT.PUT_LINE (  
    moyenne( &nom ) );
```

Exemple

2^{quater}. Exécution fonction

```
VARIABLE moy NUMBER
EXECUTE :moy := moyenne( 'Circuit
    Heleni' );
PRINT moy
```

Gestion des erreurs

- ❖ Erreur détectée par le SGBD
- ❖ Erreur générée par l'utilisateur

Chaque catégorie peut être prise en compte dans la section **EXCEPTION** ou par l'environnement appelant

Gestion par une section EXCEPTION

- ❖ Dans ce cas, l'exécution de la procédure ou de la fonction est toujours considérée comme **réussie** par l'environnement appelant. En général, on stocke les messages d'erreurs dans une table spécifique accessible à l'environnement
- ❖ Exemple : en cas de suppression d'un voyage, vérifier qu'il n'a pas de dates de départ prévues

Exemple

```
CREATE PROCEDURE delVoyage(  
num IN Voyage.numVoyage%TYPE) IS  
filler CHAR(1); erreur EXCEPTION;  
BEGIN  
SELECT 'x' INTO filler FROM Tarif  
WHERE numVoyage = num; RAISE erreur;  
EXCEPTION  
WHEN NO_DATA_FOUND THEN DELETE FROM Voyage  
WHERE numVoyage = num; COMMIT;  
WHEN erreur OR TOO_MANY_ROWS THEN  
    DBMS_OUTPUT.PUT_LINE ('Le voyage numéro :  
    ' || num || ' a déjà des dates de départ  
    prevues'); COMMIT;  
END delVoyage;
```

Gestion des erreurs par l'environnement

- ❖ Erreur émise par le SGBD : le code d'erreur, sous la forme **ORA_XXXXX**, et le message associé sont transmis au bloc appelant
- ❖ Erreur générée par l'utilisateur : utilisation de la procédure standard

**RAISE_APPLICATION_ERROR(*numéro*,
texte) ;**

- Le numéro doit être compris entre -20000 et -20999
- L'exécution de **RAISE_APPLICATION_ERROR** annule la transaction en cours

Exemple (on ne tient pas compte des CIR)

```
CREATE PROCEDURE delVoy (
num IN Voyage.numVoyage%TYPE) IS
v_nom Voyage.nomVoyage%TYPE;
BEGIN
SELECT nomVoyage INTO v_nom FROM Voyage
WHERE numVoyage = num;
DELETE FROM Voyage WHERE numVoyage = num;
DBMS_OUTPUT.PUT_LINE('Voyage ' || v_nom || '
    supprimé');
EXCEPTION
WHEN NO_DATA_FOUND THEN
RAISE_APPLICATION_ERROR (-20002,'Le voyage
    ' || num || ' n''existe pas');
END delVoy;
```

Packages

- ❖ Un package contient deux types de procédures ou fonctions :
 - Publiques
 - Privées

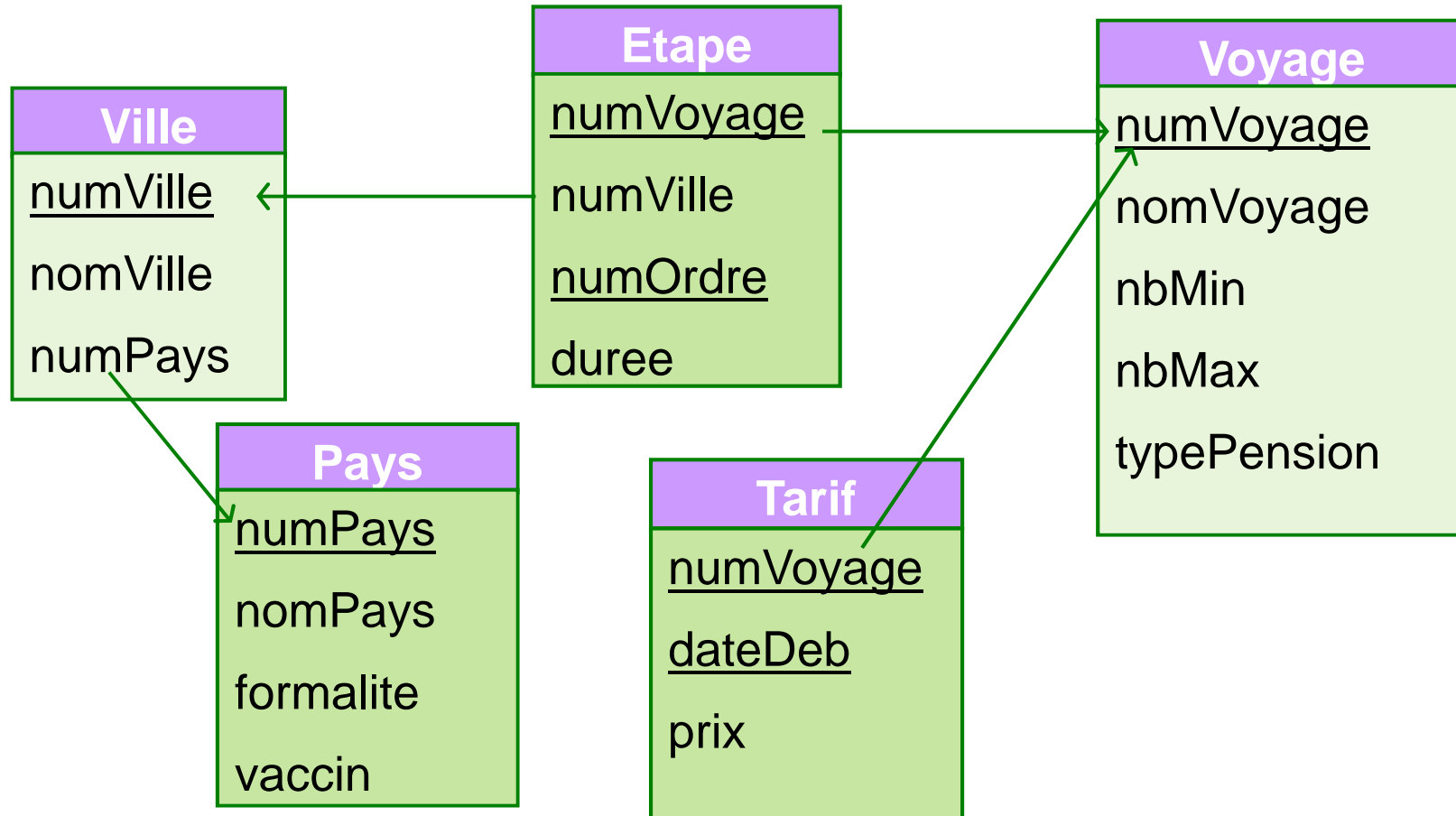
Packages (2)

- ❖ Deux parties distinctes dans un package (*chaque partie doit être créée et compilée séparément*) :
 - La **partie déclaration** ou spécification qui contient la déclaration des procédures, fonctions, variables et traitement d'exceptions de type public (accessibles de l'extérieur du package)
 - La **partie corps**, ou body, qui contient la définition des procédures ou fonctions de type public déclarées dans la partie spécification ainsi que les déclarations de procédures ou fonctions de type privé

Partie déclaration

```
CREATE [OR REPLACE] PACKAGE  
nom-package [IS | AS]  
[ déclaration-de-variable; ]  
[ déclaration-de-curseur; ]  
[ déclaration-de-procédure; ]  
[ déclaration-de-fonction; ]  
[ déclaration-d'exception; ]  
END nom-package;
```

Exemple



Déclaration : exemple

```
CREATE OR REPLACE PACKAGE gestionVoy IS
dateMax CONSTANT DATE := SYSDATE + 30;
FUNCTION lePays (maVille
ville.nomVille%TYPE) RETURN
Pays.nomPays%TYPE;
PROCEDURE etablirAgenda(monVoyage
Voyage.numVoyage%TYPE);
END gestionVoy;
```

Partie corps

```
CREATE [OR REPLACE] PACKAGE BODY  
nom-package [IS | AS]  
[ définition-de-variable; ]  
[ définition-de-curseur; ]  
[ définition-de-procédure; ]  
[ définition-de-fonction; ]  
[ définition-d'exception; ]  
END nom-package;
```

Corps : exemple

```
CREATE OR REPLACE PACKAGE BODY gestionVoy IS

FUNCTION lePays (maVille Ville.nomVille%TYPE)
RETURN Pays.nomPays%TYPE IS P Pays.nomPays%TYPE;
BEGIN SELECT nomPays INTO P FROM Pays WHERE
numPays = (SELECT numPays FROM Ville WHERE
nomVille = maVille); RETURN P; END lePays;

PROCEDURE etablirAgenda(monVoyage
Voyage.numVoyage%TYPE) IS
CURSOR voyageCUR IS SELECT * FROM Tarif WHERE
numVoyage = monVoyage; BEGIN FOR r IN voyageCUR
LOOP IF r.dateDeb <= dateMax
THEN DBMS_OUTPUT.PUT_LINE (r.dateDeb || r.prix);
END IF; END LOOP; END etablirAgenda;
END gestionVoy;
```

Référence à un élément d'un package

❖ En PL/SQL :

nom-package.nom-élément;

❖ En mode interactif :

EXECUTE ***nom-package.nom-variable := ...***

EXECUTE ***nom-package.nom-procedure***
(paramètres-effectifs)

EXECUTE ***:nom-variable := nom-***
package.nom-fonction (paramètres-
effectifs) ;

Suppression d'un package

❖ Pour la totalité du package :

DROP PACKAGE ***nom-package;***

❖ Pour seulement le corps :

DROP PACKAGE BODY ***nom-package;***

Surcharge d'une procédure ou d'une fonction

- ❖ A l'intérieur d'un package, il est possible de *surcharger* une procédure ou une fonction, c'est-à-dire de définir plusieurs procédures ou fonctions avec le même nom mais avec une liste de paramètres différente

Exemple

- ❖ On peut prévoir deux fonctions de même nom qui calculent, pour un réalisateur passé en paramètre, le nombre de films mis en scène par cette personne. Une fonction aura le nom comme paramètre, c'est-à-dire une chaîne de caractères, l'autre le numéro d'individu, c'est-à-dire un **NUMBER**

Packages (3)

- ❖ Lorsqu'un package est utilisé par plusieurs sessions, chaque session utilise sa propre copie des variables et des curseurs.
- ❖ Un utilisateur doit posséder le privilège **CREATE PROCEDURE** pour créer un package qui utilise ses propres objets
- ❖ Un utilisateur doit posséder le privilège **CREATE ANY PROCEDURE** pour créer un package qui utilise n'importe quels objets

Les déclencheurs (triggers)

- ❖ Un traitement déclenché par un événement
- ❖ L'exécution d'un déclencheur est un succès ou un échec
- ❖ En cas d'échec, l'exécution du traitement est stoppée, mais la transaction qui l'a appelé peut soit continuer soit être annulée

12 types de déclencheurs

❖ 3 événements

- **INSERT**
- **UPDATE**
- **DELETE**

❖ 2 modes

- Ordre
- Ligne (**FOR EACH ROW**)

❖ 2 moments

- **BEFORE**
- **AFTER**

Ordre d'exécution

- ❖ Il est possible d'associer un et un seul déclencheur de chaque type à chaque table. Lorsque plusieurs déclencheurs sont associés à la même table, l'ordre d'exécution est le suivant :
 1. Déclencheur par ordre **BEFORE**
 2. Pour chaque ligne (**FOR EACH ROW**)
 - Déclencheur par ligne **BEFORE**
 - Déclencheur par ligne **AFTER**
 3. Déclencheur par ordre **AFTER**

Déclencheur par ordre

```
CREATE [OR REPLACE] TRIGGER  
nom-déclencheur  
moment  
événement [OR événement]  
ON nom-table  
bloc-PL/SQL;
```

Exemple

```
CREATE OR REPLACE TRIGGER
ajoutVoyage
BEFORE
INSERT ON Voyage
BEGIN
IF USER != 'FFIOREN' THEN
RAISE_APPLICATION_ERROR (-20001,
    'Utilisateur interdit'); END IF;
END ajoutVoyage;
```

Déclencheur par ordre (2)

- ❖ Pour l'événement **UPDATE**, on peut spécifier les attributs concernés en mettant

UPDATE OF *nom-attribut*₁, ...

Exemple

```
CREATE OR REPLACE TRIGGER
updateVoyage
BEFORE
UPDATE OF nomVoyage ON Voyage
BEGIN
IF USER != 'FFIOREN' THEN
RAISE_APPLICATION_ERROR (-20001,
    'Utilisateur interdit'); END IF;
END updateVoyage;
```

Déclencheur par ordre **BEFORE**

- ❖ Un déclencheur par ordre avec l'option **BEFORE** peut servir à soumettre l'exécution d'un traitement de mise à jour d'une table à certaines conditions, avec émission d'un message d'erreur si les conditions ne sont pas vérifiées

Déclencheur par ordre AFTER

- ❖ Un déclencheur par ordre avec l'option **AFTER** peut servir à faire des validations a posteriori afin de vérifier que les modifications se sont bien déroulées. Il peut aussi permettre de propager des modifications dans plusieurs tables

Déclencheur par ligne

```
CREATE [OR REPLACE] TRIGGER  
nom-déclencheur  
moment  
événement [OR événement]  
ON nom-table  
FOR EACH ROW  
[WHERE condition]  
bloc-PL/SQL;
```

Déclencheur par ligne (2)

- ❖ On peut faire référence, dans la condition **WHERE** ou dans le bloc PL/SQL associé au déclencheur à la valeur d'un attribut avant modification en préfixant le nom de l'attribut par **:OLD**, et/ou à la valeur après modification en préfixant le nom de l'attribut par **:NEW**

Déclencheur par ligne (3)

- ❖ La valeur prise en compte dépend de l'ordre SQL :

Ordre SQL	:OLD	:NEW
INSERT	NULL	Valeur créée
DELETE	Valeur avant suppression	NULL
UPDATE	Valeur avant modification	Valeur après modification

Exemple

```
CREATE OR REPLACE TRIGGER auditVoy
AFTER DELETE OR UPDATE OR INSERT
ON Voyage FOR EACH ROW
BEGIN
  IF DELETING OR UPDATING THEN
    INSERT INTO reportVoyage
    VALUES (SYSDATE, :OLD.numVoyage,
            :OLD.nomVoyage, :OLD.nbMin, :OLD.nbMax,
            :OLD.typePension);
  END IF;
  IF INSERTING THEN NULL; END IF;
END;
```

On suppose ici qu'il existe une table reportVoyage qui a la meme structure de la table Voyage

Déclencheur par ligne (4)

- ❖ La clause **WHERE** ne peut pas contenir de requête SQL
- ❖ Un déclencheur par ligne avec l'option **BEFORE** peut servir à effectuer des traitements d'initialisation avant l'exécution des modifications sur la table
- ❖ Un déclencheur par ligne avec l'option **AFTER** permet de propager les modifications ou de gérer l'historique

Tables système

❖ **USER_TRIGGERS**

❖ **ALL_TRIGGERS**

❖ **DBA_TRIGGERS**

Suppression

DROP TRIGGER *nom-déclencheur*;

Activation/Désactivation

- ❖ **ALTER TRIGGER *nom-déclencheur* DISABLE;**
- ❖ **ALTER TABLE *nom-table* DISABLE ALL TRIGGERS;**
- ❖ **ALTER TRIGGER *nom-déclencheur* ENABLE;**
- ❖ **ALTER TABLE *nom-table* ENABLE ALL TRIGGERS;**

Restrictions

- ❖ L'exécution d'un déclencheur dont le bloc PL/SQL inclut des ordres **INSERT**, **DELETE** ou **UPDATE** peut entraîner la mise en œuvre d'un autre déclencheur associé à la table modifiée par ces actions
- ❖ Dans ce cas, lors de l'exécution d'un déclencheur de type ligne :
 - Aucun ordre SQL ne doit consulter ou modifier une table déjà utilisée en mode modification par un autre utilisateur

Restrictions (suite)

- Un déclencheur ne peut modifier la valeur d'un attribut déclaré avec l'une des contraintes **PRIMARY KEY**, **UNIQUE** ou **FOREIGN KEY**
- Un déclencheur ne peut pas consulter les données d'une table en *mutation* : une **table en mutation** est une table directement ou indirectement concernée par l'événement qui a provoqué la mise en œuvre du déclencheur

Exemple

```
CREATE OR REPLACE TRIGGER verifPrix
AFTER UPDATE OF prix OR INSERT ON Tarif
FOR EACH ROW
DECLARE
v_min Tarif.prix%TYPE;
v_max Tarif.prix%TYPE;
BEGIN
SELECT MIN(prix), MAX(prix)
INTO v_min, v_max
FROM Tarif;
IF :NEW.prix > v_max OR :NEW.prix < v_min
THEN RAISE_APPLICATION_ERROR (-20002, 'Le
    prix ' || TO_CHAR(:NEW.prix) || ' est hors
    limites'); END IF;
END verifPrix;
```

Exemple (suite)

❖ Exécution :

```
INSERT INTO Tarif VALUES (6, SYSDATE, 1000)
```

Error starting at line 1 in command:

```
INSERT INTO Tarif VALUES (6, SYSDATE, 1000)
```

Error report:

SQL Error: ORA-04091: la table VOYAGE.TARIF est en mutation ; le déclencheur ou la fonction ne peut la voir

Erreurs de compilation (warnings)

- ❖ Sous SQL Developer, pour afficher les erreurs de compilation :
 - Se positionner sur l'objet créé avec des erreurs
 - Avec la touche droite de la souris, sélectionner « Compile for Debug »